INTERNATIONAL STANDARD

ISO/IEC 9075-2

Sixth edition
2023-06

Information technology — Database languages SQL —

Part 2: Foundation (SQL/Foundation)

Technologies de l'information — Langages de base de données SQL — Partie 2: Fondations (SQL/Fondations)

Reference number
ISO/IEC 9075-2:2023(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Tables

Table

Page

Figures

Figure

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC have not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This sixth edition cancels and replaces the fifth edition (ISO/IEC 9075-2:2016), which has been technically revised. It also incorporates the Technical Corrigenda ISO/IEC 9075-2:2016/Cor.1:2019 and ISO/IEC 9075-2:2016/Cor.2:2022.

The main changes are as follows:

— native JSON datatype;

— more types of numeric literals;

— additional SQL functions:

  — ANY_VALUE,
  — BTRIM,
  — GREATEST,
  — JSON_SCALAR,
  — JSON_SERIALIZE,

  - LEAST,
  - LPAD,
  - LTRIM,
  - RPAD,
  - RTRIM;

- improve the presentation and accuracy of the summaries of implementation-defined and implementation-dependent aspects of this document;

- introduction of several digital artifacts;

- alignment with updated ISO house style and other guidelines for creating standards.

This sixth edition of ISO/IEC 9075-2 is designed to be used in conjunction with the following editions of other parts of the ISO/IEC 9075 series, all published 2023:

- ISO/IEC 9075-1, sixth edition;

- ISO/IEC 9075-3, sixth edition;

- ISO/IEC 9075-4, seventh edition;

- ISO/IEC 9075-9, fifth edition;

- ISO/IEC 9075-10, fifth edition;

- ISO/IEC 9075-11, fifth edition;

- ISO/IEC 9075-13, fifth edition;

- ISO/IEC 9075-14, sixth edition;

- ISO/IEC 9075-15, second edition;

- ISO/IEC 9075-16, first edition.

A list of all parts in the ISO/IEC 9075 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The organization of this document is as follows:

1) Clause 1, "Scope", specifies the scope of this document.
2) Clause 2, "Normative references", identifies additional standards that, through reference in this document, constitute provisions of this document.
3) Clause 3, "Terms and definitions", defines the terms and definitions used in this document.
4) Clause 4, "Concepts", presents concepts used in the definition of SQL.
5) Clause 5, "Lexical elements", defines the lexical elements of the language.
6) Clause 6, "Scalar expressions", defines the elements of the language that produce scalar values.
7) Clause 7, "Query expressions", defines the elements of the language that produce rows and tables of data.
8) Clause 8, "Predicates", defines the predicates of the language.
9) Clause 9, "Additional common rules", specifies the rules for assignments that retrieve data from or store data into SQL-data, and formation rules for set operations.
10) Clause 10, "Additional common elements", defines additional language elements that are used in various parts of the language.
11) Clause 11, "Schema definition and manipulation", defines facilities for creating and managing a schema.
12) Clause 12, "Access control", defines facilities for controlling access to SQL-data.
13) Clause 13, "SQL-client modules", defines SQL-client modules and externally-invoked procedures.
14) Clause 14, "Data manipulation", defines the data manipulation statements.
15) Clause 15, "Additional data manipulation rules", defines additional rules for data manipulation.
16) Clause 16, "Control statements", defines the SQL-control statements.
17) Clause 17, "Transaction management", defines the SQL-transaction management statements.
18) Clause 18, "Connection management", defines the SQL-connection management statements.
19) Clause 19, "Session management", defines the SQL-session management statements.
20) Clause 20, "Dynamic SQL", defines the SQL dynamic statements.
21) Clause 21, "Embedded SQL", defines the host language embeddings.
22) Clause 22, "Direct invocation of SQL", defines direct invocation of SQL language.
23) Clause 23, "Diagnostics management", defines the diagnostics management facilities.
24) Clause 24, "Status codes", defines values that identify the status of the execution of SQL-statements and the mechanisms by which those values are returned.
25) Clause 25, "Conformance", defines the criteria for conformance to this document.
26) Annex A, "SQL conformance summary", is an informative Annex. It summarizes the conformance requirements of the SQL language.

27) Annex B, "Implementation-defined elements", is an informative Annex. It lists those features for which the body of this document states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or other aspect is partly or wholly implementation-defined.

28) Annex C, "Implementation-dependent elements", is an informative Annex. It lists those features for which the body of this document states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or other aspect is partly or wholly implementation-dependent.

29) Annex D, "SQL optional feature taxonomy", is an informative Annex. It identifies the optional features of the SQL language specified in this document by an identifier and a short descriptive name. This taxonomy is used to specify conformance.

30) Annex E, "Deprecated features", is an informative Annex. It lists features that the responsible Technical Committee intends not to include in a future edition of this document.

31) Annex F, "Incompatibilities with ISO/IEC 9075:2016", is an informative Annex. It lists incompatibilities with the previous edition of this document.

32) Annex G, "Defect Reports not addressed in this edition of this document", is an informative Annex. It describes the Defect Reports that were known at the time of publication of this document. Each of these problems is a problem carried forward from the previous edition of the ISO/IEC 9075 series. No new problems have been created in the drafting of this edition of this document.

33) Annex H, "SQL mandatory feature taxonomy", is an informative Annex. It identifies mandatory features and subfeatures of the SQL language specified in this document by an identifier and a short descriptive name. This taxonomy is used to specify conformance to Core SQL.

In the text of this document, in Clause 5, "Lexical elements", through Clause 24, "Status codes", Subclauses begin new pages. Any resulting blank space is not significant.

Information technology — Database language SQL —

Part 2:
Foundation (SQL/Foundation)

1 Scope

This document defines the data structures and basic operations on SQL-data. It provides functional capabilities for creating, accessing, maintaining, controlling, and protecting SQL-data.

This document specifies the syntax and semantics of a database language:

— for specifying and modifying the structure and the integrity constraints of SQL-data;

— for declaring and invoking operations on SQL-data and cursors;

— for declaring database language procedures;

— for embedding SQL-statements in a compilation unit that is otherwise written in a particular programming language (host language);

— for deriving an equivalent compilation unit in the host language. In that equivalent compilation unit, each embedded SQL-statement has been replaced by one or more statements in the host language, some of which invoke an SQL externally-invoked procedure that, when executed, has an effect equivalent to executing the SQL-statement;

— for direct invocation of SQL-statements;

— to support dynamic preparation and execution of SQL-statements.

This document provides a vehicle for portability of data definitions and compilation units between SQL-implementations.

This document provides a vehicle for interconnection of SQL-implementations.

Implementations of this document can exist in environments that also support application programming languages, end-user query languages, report generator systems, data dictionary systems, program library systems, and distributed communication systems, as well as various tools for database design, data administration, and performance optimization.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies. [1]

ISO/IEC 646, *Information technology — ISO 7-bit coded character set for information interchange*

ISO/IEC 1539-1:2018, *Information technology — Programming languages — Fortran — Part 1: Base language*

ISO 1989:2014, *Information technology — Programming languages — COBOL*

ISO 6160:1979, *Programming languages — PL/I (Endorsement of ANSI X3.53-1976)*

ISO/IEC 6429, *Information technology — Control functions for coded character sets*

ISO 7185:1990, *Information technology — Programming languages — Pascal*

ISO 8601-1:2019, *Date and time — Representations for information interchange — Part 1: Basic rules*

ISO/IEC 8652:2012, *Information technology — Programming languages — Ada*

ISO/IEC 8652:2012/Cor.1:2016, *Information technology — Programming languages — Ada — Technical Corrigendum 1*

ISO/IEC 8859-1, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

ISO/IEC 9075-1, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*

ISO/IEC 9075-11, *Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata)*

ISO/IEC 9579, *Information technology — Remote database access for SQL with security enhancement*

ISO/IEC 9899:2018, *Information technology — Programming languages — C*

ISO/IEC 10206:1991, *Information technology — Programming languages — Extended Pascal*

ISO/IEC 10646:2020, *Information technology — Universal Multi-Octet Coded Character Set (UCS)*

ISO/IEC 11756:1999, *Information technology — Programming languages — M*

ISO/IEC 14651:2020, *Information technology — International string ordering and comparison — Method for comparing character strings and description of the common template tailorable ordering*

ECMA International. *ECMA-262 — ECMAScript® Language Specification 5.1 Edition* [online]. [Place of publication unknown]: Available at `https://262.ecma-international.org/5.1/-ECMA-262.pdf`

1 In this document, ECMAScript Language Specification 5.1 Edition is referenced for the purpose of defining the lexical elements of the SQL/JSON path language specified in Subclause 9.45, "SQL/JSON path language: lexical elements", and Subclause 9.46, "SQL/JSON path language: syntax and semantics". There are no intentions to update this reference to a newer edition of ECMA-262.

Internet Engineering Task Force (IETF) RFC 8259 *The JavaScript Object Notation (JSON) Data Interchange Format*. Edited by: Miller, Matthew December 2018
Available at: https://datatracker.ietf.org/doc/rfc8259/

The Unicode Consortium. *Unicode Regular Expressions* [online]. 21. Mountain View, California, USA: The Unicode Consortium, 2020-06-17 . Available at https://www.unicode.org/reports/tr18/-tr18-21.html

W3C XQuery and XPath Functions and Operators 3.1 *XQuery and XPath Functions and Operators 3.1, W3C Recommendation*. Edited by: Malhotra, Ashok et al. 21 March 2017
Available at: https://www.w3.org/TR/xpath-functions/

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9075-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at `https://www.iso.org/obp`

— IEC Electropedia: available at `https://www.electropedia.org/`

3.1 Definitions taken from ISO/IEC 10646:2020

3.1.1
character encoding form
form that determines how each UCS *code point* (3.1.2) for a UCS character is to be expressed as one or more code units used by the encoding form

[SOURCE: ISO/IEC 10646:2020, 3.22]

3.1.2
code point
value in the UCS codespace

[SOURCE: ISO/IEC 10646:2020, 3.9]

3.1.3
code unit
minimal bit combination that can represent a unit of encoded text for processing or interchange

[SOURCE: ISO/IEC 10646:2020, 3.10]

3.1.4
control character
control function the coded representation of which consists of a single *code point* (3.1.2)

[SOURCE: ISO/IEC 10646:2020, 3.17]

3.1.5
noncharacter code point
code point (3.1.2) that consists of FDD0-FDEF and any *code point* (3.1.2) ending in the value FFFE or FFFF

[SOURCE: ISO/IEC 10646:2020, 7.3.7]

3.1.6
normalization form
mechanisms allowing the selection of a unique coded representation among alternative but equivalent coded text representations of the same text

[SOURCE: ISO/IEC 10646:2020, 22]

3.1.7
repertoire
specified set of characters that are represented in a coded character set

[SOURCE: ISO/IEC 10646:2020, 3.46]

3.2 Definitions taken from ISO/IEC 14651:2020

3.2.1
collation
process by which, given two strings, it is determined whether the first one is less than, equal to, or greater than the second one

[SOURCE: ISO/IEC 14651:2020, 3.7]

3.3 Definitions taken from ISO 8601-1:2019

3.3.1
UTC
Coordinated Universal Time
Time scale with the same rate as International Atomic Time (TAI), but differing from TAI only by an integral number of seconds

[SOURCE: ISO 8601-1:2019, 3.1.1.12]

3.3.2
date
time on the calendar time scale

[SOURCE: ISO 8601-1:2019, 3.1.1.1]

3.3.3
calendar day of month
ordinal number of a calendar day within a calendar month

[SOURCE: ISO 8601-1:2019, 3.1.2.13]

3.3.4
calendar day of year
ordinal number of a calendar day within a calendar year

[SOURCE: ISO 8601-1:2019, 3.1.2.14]

3.3.5
calendar month
time scale unit resulting from a defined division of a calendar year, each containing a specific number of calendar days

[SOURCE: ISO 8601-1:2019, 3.1.2.19]

3.3.6
calendar year
time scale unit defined by the calendar system

[SOURCE: ISO 8601-1:2019, 3.1.2.21]

3.3.7
clock hour
time scale unit whose duration is one hour

[SOURCE: ISO 8601-1:2019, 3.1.2.6]

3.3.8
clock minute
time scale unit whose duration is one minute

[SOURCE: ISO 8601-1:2019, 3.1.2.4]

3.3 Definitions taken from ISO 8601-1:2019

3.3.9
clock second
time scale unit whose duration is one second

[SOURCE: ISO 8601-1:2019, 3.1.2.2]

3.3.10
day
duration of a calendar day

[SOURCE: ISO 8601-1:2019, 3.1.2.10]

3.3.11
Gregorian calendar
calendar in general use that defines a calendar year that closely approximates the tropical year

[SOURCE: ISO 8601-1:2019, 3.1.1.19]

3.3.12
hour
duration of 60 minutes

[SOURCE: ISO 8601-1:2019, 3.1.2.5]

3.3.13
leap second
intentional time step of one second to adjust UTC to ensure appropriate agreement with UT1, a time scale based on the rotation of the Earth

[SOURCE: ISO 8601-1:2019, 3.1.1.24]

3.3.14
minute
duration of 60 seconds

[SOURCE: ISO 8601-1:2019, 3.1.2.3]

3.3.15
month
duration of a calendar month

[SOURCE: ISO 8601-1:2019, 3.1.2.18]

3.3.16
second
base unit of duration measurement in the International System of Units (SI)

[SOURCE: ISO 8601-1:2019, 3.1.2.1]

3.3.17
year
duration of a calendar year

[SOURCE: ISO 8601-1:2019, 3.1.2.20]

3.4 Definitions taken from XQuery and XPath Functions and Operators 3.1

3.4.1
XQuery option flag
valid value of the `$flags` argument of `fn:matches`, to set options for the interpretation of the regular expression

[SOURCE: XQuery and XPath Functions and Operators 3.1, 5.6.2]

3.4.2
XQuery regular expression
regular expression syntax used by functions as defined in terms of the regular expression syntax specified in XML Schema

[SOURCE: XQuery and XPath Functions and Operators 3.1, 5.6.1]

3.4.3
XQuery regular expression parenthesized subexpression
regular expression parenthesized subexpression

[SOURCE: XQuery and XPath Functions and Operators 3.1, 5.6.1.3]

3.4.4
XQuery replacement string
valid value of the `$replacement` argument of `fn:replace`

[SOURCE: XQuery and XPath Functions and Operators 3.1, 5.6.4]

3.5 Definitions provided in this document

3.5.1
assignable, adj.
<of data types, taken pairwise> characteristic of a data type *T1* that permits a value of *T1* to be assigned to a site of a specified data type *T2*, where *T1* and *T2* may be the same data type

3.5.2
assignment
operation whose effect is to ensure that the value at a site *T* (known as the *target*) is identical to a given value *S* (known as the *source*)

Note 1 to entry: Assignment is frequently indicated by the use of the phrase "*T* is set to *S*" or "the value of *T* is set to *S*".

3.5.3
attribute
component of a *structured type* (3.5.68)

Note 1 to entry: Each value *V* in structured type *T* has exactly one attribute value for each attribute *A* of *T*. The characteristics of an attribute are specified by an attribute descriptor. The value of an attribute may be retrieved as the result of the invocation *A*(*V*) of the *observer function* (3.5.38) for that attribute.

3.5.4
cardinality
<of a collection> number of elements in that collection

Note 1 to entry: Those elements need not necessarily have distinct values. The objects to which this concept applies include tables and the values of collection types.

3.5.5
comparable, adj.
<of a pair of values> capable of being compared

Note 1 to entry: In most, but not all, cases, the values of a data type can be compared one with another. For the specification of comparability of individual data types, see Subclause 4.3, "Character strings", through Subclause 4.12, "Collection types".

3.5.6
constructor function
niladic (3.5.37) *SQL-invoked function* (3.5.62) of which exactly one is implicitly specified for every *structured type* (3.5.68)

Note 1 to entry: An invocation of the constructor function for data type *T* returns a value *V* of the most specific type of *T* such that *V* is not the null value and, for every *observer function* (3.5.38) *O* defined for *T*, the invocation *O*(*V*) returns the default value of the *attribute* (3.5.3) corresponding to *O*.

3.5.7
data model
<general> definition of the kinds of data that belong to a particular universe of discourse, including the operations on those kinds of data

3.5.8
declared type
<of an expression denoting a value or anything that can be referenced to denote a value, such as, for example, a parameter, column, or variable > unique data type that is common to every value that might result from evaluation of that expression

3.5.9
distinct, adj.
<of a pair of comparable values> capable of being distinguished within a given context

Note 1 to entry: Informally, two values are distinct if neither is null and the values are not equal. A null value and a non-null value are distinct. Two null values are not distinct. See Subclause 4.2.5, "Properties of distinct", and the General Rules of Subclause 8.15, "<distinct predicate>".

3.5.10
distinct type
user-defined type (3.5.76) derived from a predefined type or a collection type

3.5.11
duplicates
two or more members of a multiset that are not distinct

3.5.12
dyadic, adj.
<of operators, functions, and procedures> having exactly two operands or parameters

Note 1 to entry: An example of a dyadic operator in this document is "–", specifying the subtraction of the right operand from the left operand. An example of a dyadic function is POSITION.

3.5.13
element type
<of a collection type> declared type *DT* specified in the definition of a collection type *CT*

Note 1 to entry: The declared type of every element of every value of type *CT* is *DT*.

3.5.14
empty row pattern match
row pattern match (3.5.51) that matches no rows

3.5.15
external routine
SQL-invoked routine (3.5.64) whose routine body is an external body reference that identifies a program written in a programming language other than SQL

3.5.16
fixed-length, adj.
characteristic of the declared type of sites whose types are character string types or binary string types that restricts values in those sites to contain exactly one number of characters or octets, respectively, known as the length in characters or octets, respectively, of the site

3.5.17
fully qualified, adj.
<of a name of some SQL object> with all optional components specified explicitly

Note 1 to entry: A fully qualified name does not necessarily identify an object uniquely. For example, although a fully qualified specific name, consisting of a catalog name, a schema name and a specific name, uniquely identifies a routine, a fully qualified routine name does not necessarily do so.

3.5.18
identical, adj.
<of a pair of values> indistinguishable, in the sense that it is impossible, by any means specified in this document, to detect any difference between them

Note 1 to entry: For the full definition, see Subclause 9.10, "Determination of identical values".

3.5.19
interface
<of an *user-defined type* (3.5.76)> set comprising every function such that the declared type of at least one of its parameters or its result is that *user-defined type* (3.5.76)

3.5.20
JSON array
structure represented by a "[", zero or more elements separated by ",", and "]"

3.5.21
JSON Boolean
JSON literal (3.5.24) "**true**" or *JSON literal* (3.5.24) "**false**"

3.5.22
JSON element
JSON text (3.5.30) fragment that is a *JSON value* (3.5.33) in a *JSON array* (3.5.20)

3.5.23
JSON data model
implicit data model associated with JSON

3.5.24
JSON literal
JSON text fragment (3.5.31) that is any of the key words "true", "false", or "null"

3.5.25
JSON member
JSON string (3.5.29) followed by a colon followed by a *JSON value* (3.5.33) in a *JSON object* (3.5.28)

Note 1 to entry: This is also known as a "name-value pair"; the name is sometimes called a "key" and the second value is sometimes called a "bound value".

3.5.26
JSON null
JSON literal (3.5.24) "null"

Note 1 to entry: A JSON null is distinct from an SQL null value and from an SQL/JSON null.

3.5.27
JSON number
Unicode character string comprising an integer part, optionally followed by a fractional part and/or an exponent part

3.5.28
JSON object
structure represented by a "{", zero or more members separated by ",", and "}"

3.5.29
JSON string
Unicode character string

Note 1 to entry: Some characters must be "escaped" by preceding them with a reverse solidus ("\"), while any or all characters can be represented in "Unicode notation" comprising the string "\u" followed by four hexadecimal digits or two such strings representing the UTF-16 surrogate pairs representing characters not on the Basic Multilingual Plane (strings are surrounded by double-quote characters, which are not part of the value of the strings).

Note 2 to entry: This applies only to *JSON tokens* (3.5.32) in *JSON text* (3.5.30).

3.5.30
JSON text
sequence of *JSON tokens* (3.5.32), which must be encoded in Unicode (UTF-8 by default); insignificant whitespace may be used anywhere in *JSON text* (3.5.30) except within strings (where all whitespace is significant), numbers, and literals

3.5.31
JSON text fragment
substring of a *JSON text* (3.5.30) that conforms to any BNF non-terminal in RFC 8259

3.5.32
JSON token
one of six structural characters ("{", "}", "[", "]", ":", ","), *JSON strings* (3.5.29), *JSON numbers* (3.5.27), and *JSON literals* (3.5.24)

3.5.33
JSON value
JSON object (3.5.28), *JSON array* (3.5.20), *JSON number* (3.5.27), *JSON string* (3.5.29), or one of three *JSON literals* (3.5.24)

3.5.34
monadic, adj.
<of operators, functions, and procedures> having exactly one operand or parameter

Note 1 to entry: An example of a monadic arithmetic operator in this document is "–", specifying the negation of the operand. An example of a monadic function is CHARACTER_LENGTH, specifying the length in characters of the argument.

3.5.35
most specific type
<of a value> unique data type of which every data type of that value is a supertype

3.5.36
mutator function
dyadic, type-preserving *SQL-invoked function* (3.5.62) implicitly defined by the definition of an *attribute* (3.5.3) of a *structured type* (3.5.68) that, when invoked, modifies the value of the *attribute* (3.5.3) with which it is associated

3.5.37
niladic, adj.
<of functions and procedures> having no parameters

3.5.38
observer function
monadic *SQL-invoked function* (3.5.62) associated with an *attribute* (3.5.3) of a *structured type* (3.5.68) that, when invoked, returns the value of the *attribute* (3.5.3) with which it is associated.

Note 1 to entry: The observer function is implicitly defined by the definition of the attribute.

3.5.39
primary row pattern variable
row pattern variable (3.5.58) that appears in a <row pattern>

3.5.40
redundant duplicates
all except one of any collection of duplicate values or rows

3.5.41
REF value
value that references some site

3.5.42
reference type
data type all of whose values are potential references to sites of one specified data type

3.5.43
referenced type
declared type of the values at sites referenced by values of a particular *reference type* (3.5.42)

3.5.44
result data type
declared type of the result of an *SQL-invoked function* (3.5.62)

3.5.45
result set
sequence of rows specified by a <cursor specification> that is brought into existence by opening a cursor and ranged over by that cursor

3.5.46
result set sequence
sequence of returned *result sets* (3.5.45)

3.5.47
result SQL parameter
SQL parameter (3.5.66), the *most specific type* (3.5.35) of the value of which determines the *most specific type* (3.5.35) of the result of the *SQL-invoked function* (3.5.62) of which it is a parameter

3.5.48
returned result set
result set (3.5.45) created during execution of an *SQL-invoked procedure* (3.5.63) and not destroyed when that execution terminates

Note 1 to entry: Such a *result set* (3.5.45) can be accessed by using a cursor other than the one that brought it into existence (a received cursor).

3.5.49
row pattern
form of regular expressions that specifies criteria (patterns) used to identify collections of rows in a table based on their relationships to one another

3.5.50
row pattern input table
table that is the input operand of a <row pattern recognition clause> or of a <window definition> that specifies window row pattern recognition

3.5.51
row pattern match
sequence of consecutive rows in a *row pattern partition* (3.5.57) that collectively meet the match criteria of a <row pattern common syntax>

3.5.52
row pattern matching
process of identifying collections of rows in a table based on the patterns specified in *row patterns* (3.5.49)

3.5.53
row pattern measure
named scalar expression defined with respect to the *row pattern variables* (3.5.58)

3.5.54
row pattern measure column
exported column of the *row pattern output table* (3.5.56), the value for which is specified by a *row pattern measure* (3.5.53)

3.5.55
row pattern measure function
window function whose value is specified by a *row pattern measure* (3.5.53)

3.5.56
row pattern output table
table that is the result of a <row pattern recognition clause> applied to its *row pattern input table* (3.5.50)

3.5.57
row pattern partition
maximal collection of rows of the *row pattern input table* (3.5.50) in which all rows have non-distinct values for a set of one or more columns, called the *row pattern partitioning columns*; or a window partition in a window whose descriptor has window row pattern recognition

3.5.58
row pattern variable
primary row pattern variable (3.5.39) or an *union row pattern variable* (3.5.74)

3.5.59
savepoint
point within an SQL-transaction, identified by a savepoint name, to which that SQL-transaction may be restored

3.5.60
signature
<of an *SQL-invoked routine* (3.5.64)> name of an *SQL-invoked routine* (3.5.64), the position and declared type of each of its *SQL parameters* (3.5.66), and an indication of whether it is an *SQL-invoked function* (3.5.62) or an *SQL-invoked procedure* (3.5.63)

3.5.61
SQL argument
expression denoting a value to be substituted for an *SQL parameter* (3.5.66) in an invocation of an *SQL-invoked routine* (3.5.64)

3.5.62
SQL-invoked function
function that is allowed to be invoked only from within SQL

3.5.63
SQL-invoked procedure
procedure that is allowed to be invoked only from within SQL

3.5.64
SQL-invoked routine
routine that is allowed to be invoked only from within SQL

3.5.65
SQL/JSON data model
data model created for operating on JSON data within the SQL language

3.5.66
SQL parameter
parameter declared as part of the signature of an *SQL-invoked routine* (3.5.64)

3.5.67
SQL routine
SQL-invoked routine (3.5.64) whose routine body is written in SQL

3.5.68
structured type
user-defined type (3.5.76) specified as a set of *attributes* (3.5.3)

3.5.69
subfield
<of a row type> field that is a field of a row type *RT* or a field of a row type *RT2* that is the declared type of a field that is a subfield of *RT*

3.5.70
subtype
<of a data type> data type *T2* such that every value of *T2* is also a value of data type *T1*

Note 1 to entry: If *T1* and *T2* are not compatible, then *T2* is a *proper subtype* of *T1*. "Compatible" is defined in Subclause 4.2.4, "Data type terminology". See also *supertype* (3.5.71).

3.5.71
supertype
<of a data type> data type *T1* such that every value of *T2* is also a value of data type *T1*

Note 1 to entry: If *T1* and *T2* are not compatible, then *T1* is a *proper supertype* of *T2*. "Compatible" is defined in Subclause 4.2.4, "Data type terminology". See also *subtype* (3.5.70).

3.5.72
transliteration
method of translating characters in one character set into characters of the same or a different character set

3.5.73
type-preserving function
SQL-invoked function (3.5.62), one of whose parameters is a result *SQL parameter* (3.5.66)

Note 1 to entry: The *most specific type* (3.5.35) of the value returned by an invocation of a type-preserving function is identical to the *most specific type* (3.5.35) of the SQL argument value substituted for the result *SQL parameter* (3.5.66).

3.5.74
union row pattern variable
row pattern variable (3.5.58) that is defined as a union of *primary row pattern variables* (3.5.39) in a <row pattern subset clause>

3.5.75
universal row pattern variable
implementation-defined *union row pattern variable* (3.5.74) that is defined implicitly as the union of all *primary row pattern variables* (3.5.39)

3.5.76
user-defined type
data type whose interface is user-defined

3.5.77
variable-length, adj.
characteristic of the declared type of sites whose types are character string types or binary string types that allows values in those sites to contain any number of characters or octets, respectively, between 0 (zero) and some maximum number, known as the maximum length in characters or octets, respectively, of the site

3.5.78
with-return cursor
cursor that, when opened, creates a result set that is capable of becoming a returned result set

Note 1 to entry: The WITH RETURN option of <declare cursor>, <dynamic declare cursor>, and <allocate extended dynamic cursor statement> specifies a with-return cursor. WITH RETURN may also be specified in the content of an <attributes variable> in a <prepare statement>, to indicate that the prepared statement, when opened as a dynamic cursor, creates a with-return cursor.

4 Concepts

This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-3.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-4.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-9.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-10.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-11.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-13.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-14.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-15.
This Clause is modified by Clause 4, "Concepts", in ISO/IEC 9075-16.

4.1 Notations and conventions

This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-11.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 4.1, "Notations and conventions", in ISO/IEC 9075-16.

4.1.1 Notations

This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-11.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 4.1.1, "Notations", in ISO/IEC 9075-16.

The notations used in this document are defined in ISO/IEC 9075-1.

The syntax defined in this document is available from the ISO website as a "digital artifact". See `https://standards.iso.org/iso-iec/9075/-2/ed-6/en/` to download digital artifacts for this document. To download the syntax defined in a plain-text format, select the file named `ISO_IEC_9075-2(E)_Foundation.bnf.txt`. To download the syntax defined in an XML format, select the file named `ISO_IEC_9075-2(E)_Foundation.bnf.xml`.

4.1.2 Use of terms

This Subclause is modified by Subclause 4.1.2, "Use of terms", in ISO/IEC 9075-4.

An SQL-statement *S1* is said to be executed as a *direct result of executing an SQL-statement* if *S1* is the SQL-statement contained in an <externally-invoked procedure> or <SQL-invoked routine> that has been executed.

04 An SQL-statement *S1* is said to be executed as a *direct result of executing an SQL-statement* if *S1* is the value of an <SQL statement variable> referenced by an <execute immediate statement> contained in an <externally-invoked procedure> that has been executed, or if *S1* was the value of the <SQL statement variable> that was associated with an <SQL statement name> by a <prepare statement> and that same <SQL statement name> is referenced by an <execute statement> contained in an <externally-invoked procedure> that has been executed.

4.2 Data types

This Subclause is modified by Subclause 4.2, "Data types", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.8, "Data types", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.2, "Data types", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.2, "Data types", in ISO/IEC 9075-15.

4.2.1 General introduction to data types

This Subclause is modified by Subclause 4.2.1, "General introduction to data types", in ISO/IEC 9075-15.

A data type is a set of representable values. Every representable value belongs to at least one data type and some belong to several data types. The physical representation of a value of a data type is implementation-dependent (UV060).

Exactly one of the data types of a value *V*, namely the most specific type of *V*, is a subtype of every data type of *V*. A <value expression> *E* has exactly one declared type, common to every possible result of evaluating *E*. Items that can be referenced by name, such as SQL parameters, columns, fields, attributes, and variables, also have declared types.

SQL supports three sorts of data types: *predefined data types*, *constructed types*, and *user-defined types*. Predefined data types are sometimes called "built-in data types", though not in this document. User-defined types can be defined by a standard, by an SQL-implementation, or by an application.

15 A constructed type is specified using data type constructors. The constructed types and the associated data type constructors are:

— 15 array type: ARRAY;
— multiset type: MULTISET;
— reference type: REF;
— row type: ROW.

15 Array types and multiset types are known generically as *collection types*. SQL defines a category of data types known as *collection types*. The collection types are:

— array type;
— multiset type.

Every predefined data type is a subtype of itself and of no other data types. It follows that every predefined data type is a supertype of itself and of no other data types. The predefined data types are individually described in each of Subclause 4.3, "Character strings", through Subclause 4.8, "JSON types".

Row types, reference types, and collection types are described in Subclause 4.10, "Row types", Subclause 4.11, "Reference types", Subclause 4.12, "Collection types", respectively.

A user-defined type is either a *distinct type* or a *structured type*. User-defined types are described in Subclause 4.9, "User-defined types".

4.2.2 Naming of predefined types

This Subclause is modified by Subclause 4.2.1, "Naming of predefined types", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.2.1, "Naming of predefined types", in ISO/IEC 9075-14.

09 14 SQL defines predefined data types named by the following <key word>s:

CHARACTER

CHARACTER VARYING

CHARACTER LARGE OBJECT

BINARY

BINARY VARYING

BINARY LARGE OBJECT

NUMERIC

DECIMAL

SMALLINT

INTEGER

BIGINT

FLOAT

REAL

DOUBLE PRECISION

DECFLOAT

BOOLEAN

DATE

TIME

TIMESTAMP

INTERVAL

JSON

These names are used in the *type designators* that constitute the *type precedence lists* specified in Subclause 9.7, "Type precedence list determination". In addition, TABLE is the type designator for the generic table parameter type, and DESCRIPTOR is the type designator for the descriptor parameter type.

NOTE 1 — The generic table parameter type and the descriptor parameter type only occur as parameter types, and only in the parameter lists of polymorphic table functions; they are not available as types for columns, domains, or parameters of non-polymorphic table functions.

09 14 For reference purposes:

— The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are collectively referred to as *character string types* and the values of character string types are known as *character strings*.

— The data types BINARY, BINARY VARYING, and BINARY LARGE OBJECT are referred to as *binary string types* and the values of binary string types are referred to as *binary strings*.

— The data types CHARACTER LARGE OBJECT and BINARY LARGE OBJECT are collectively referred to as *large object string types* and the values of large object string types are referred to as *large object strings*.

— Character string types and binary string types are collectively referred to as *string types* and values of string types are referred to as *strings*.

— The data types NUMERIC, DECIMAL, SMALLINT, INTEGER, and BIGINT are collectively referred to as *exact numeric types*.

— The data types FLOAT, REAL, and DOUBLE PRECISION are collectively referred to as *approximate numeric types*.

— The data type DECFLOAT is referred to as the *decimal floating-point type*.

— Exact numeric types, approximate numeric types, and the decimal floating-point type are collectively referred to as *numeric types*. Values of numeric types are referred to as *numbers*.

— The data types TIME WITHOUT TIME ZONE and TIME WITH TIME ZONE are collectively referred to as *time types* (or, for emphasis, as time with or without time zone).

— The data types TIMESTAMP WITHOUT TIME ZONE and TIMESTAMP WITH TIME ZONE are collectively referred to as *timestamp types* (or, for emphasis, as timestamp with or without time zone).

— The data types DATE, TIME, and TIMESTAMP are collectively referred to as *datetime types*.

— Values of datetime types are referred to as *datetimes*.

— The data type INTERVAL is referred to as an *interval type*. Values of interval types are called *intervals*.

Each data type has an associated data type descriptor; the contents of a data type descriptor are determined by the specific data type that it describes. A data type descriptor includes an identification of the data type and all information needed to characterize a value of that data type.

Subclause 6.1, "<data type>", describes the semantic properties of each data type.

4.2.3 Host language data types

This Subclause is modified by Subclause 4.8.1, "Host language data types", in ISO/IEC 9075-13.

13 Each host language has its own data types, which are separate and distinct from SQL data types, even though similar names may be used to describe the data types. Mappings of SQL data types to data types in host languages are described in:

— Subclause 11.60, "<SQL-invoked routine>"

— Subclause 21.1, "<embedded SQL host program>"

Not every SQL data type has a corresponding data type in every host language.

4.2.4 Data type terminology

This Subclause is modified by Subclause 4.2.2, "Data type terminology", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.2.2, "Data type terminology", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.2.2, "Data type terminology", in ISO/IEC 9075-15.

The notion of a *constituent* of a declared type *DT* is defined recursively as follows:

— *DT* is a constituent of *DT*;

— If *DT* is a row type, then the declared type of each field of *DT* is a constituent of *DT*;

— If *DT* is a collection type, then the element type of *DT* is a constituent of *DT*;

— Every constituent of a constituent of *DT* is a constituent of *DT*.

A data type *TY* is *usage-dependent* on a user-defined type *UDT* if and only if exactly one of the following is true:

— *TY* is *UDT*;

— *TY* is a reference type whose referenced type is *UDT*;

— *TY* is a row type, and the declared type of some field of *TY* is usage-dependent on *UDT*;

— *TY* is a collection type, and the declared element type of *TY* is usage-dependent on *UDT*.

A structured type *ST* is *directly based on* a data type *DT* if and only if at least one of the following is true:

— *DT* is the declared type of some attribute of *ST*;

— *DT* is a direct supertype of *ST*;

— *DT* is a direct subtype of *ST*;

— *DT* is compatible with *ST*.

A collection type *CT* is *directly based on* a data type *DT* if *DT* is the element type of *CT*.

A row type *RT* is *directly based on* a data type *DT* if *DT* is the declared type of some field (or the data type of the domain of some field) whose descriptor is included in the descriptor of *RT*.

A data type *DT1* is *based on* a data type *DT2* if *DT1* is compatible with *DT2*, *DT1* is directly based on *DT2*, or *DT1* is directly based on some data type that is based on *DT2*.

Two data types, *T1* and *T2*, are said to be *compatible* if *T1* is assignable to *T2*, *T2* is assignable to *T1*, and their descriptors include the same data type name. If they are row types, it shall further be the case that the declared types of their corresponding fields are pairwise compatible. If they are collection types, it shall further be the case that their element types are compatible. If they are reference types, it shall further be the case that their referenced types are compatible.

NOTE 2 — The data types "CHARACTER(*n*) CHARACTER SET *CS1*" and "CHARACTER(*m*) CHARACTER SET *CS2*", where *CS1* ≠ *CS2*, have descriptors that include the same data type name (CHARACTER), but are not mutually assignable; therefore, they are not compatible.

Ordering and comparison of values of the predefined data types requires knowledge only about those predefined data types. However, to be able to compare and order values of constructed types or of user-defined types, additional information is required. We say that some type *T* is *S*-ordered, for some set of types *S*, if, in order to compare and order values of type *T*, information about ordering at least one of the types in *S* is first required. A definition of *S*-ordered is required for several *S* (that is, for several sets of types), but not for all possible such sets.

The general definition of *S*-ordered is as follows.

Let *T* be a type and let *S* be a set of types. *T* is *S-ordered* if and only if exactly one of the following is true:

— *T* is a member of *S*;

— *T* is a row type and the declared type of some field of *T* is *S*-ordered;

— *T* is a collection type and the element type of *T* is *S*-ordered;

— *T* is a structured type whose comparison form is STATE and the declared type of some attribute of *T* is *S*-ordered;

— *T* is a user-defined type whose comparison form is MAP and the return type of the SQL-invoked function that is identified by the <map function specification> is *S*-ordered;

— *T* is a reference type with a derived representation and the declared type of some derivational attribute of the derived representation is *S*-ordered.

09 14 15 The notion of *S*-ordered is applied in the following definitions:

— A type *T* is *JSON-ordered* if *T* is *S*-ordered, where *S* is the set of JSON types.

— A type *T* is *LOB-ordered* if *T* is *S*-ordered, where *S* is the set of large object types.

— A type *T* is *array-ordered* if *T* is *S*-ordered, where *S* is the union of the set of array types and the set of distinct types whose source type is an array type.

— A type *T* is *multiset-ordered* if *T* is *S*-ordered, where *S* is the union of the set of multiset types and the set of distinct types whose source type is a multiset type.

— A type *T* is *row-ordered* if *T* is *S*-ordered, where *S* is the set of row types.

— A type *T* is *reference-ordered* if *T* is *S*-ordered, where *S* is the set of reference types.

— A type *T* is *DT-EC-ordered* if *T* is *S*-ordered, where *S* is the set of distinct types with EQUALS ONLY comparison form (DT-EC stands for "distinct type-equality comparison").

— A type *T* is *DT-FC-ordered* if *T* is *S*-ordered, where *S* is the set of distinct types with FULL comparison form.

— A type *T* is *DT-NC-ordered* if *T* is *S*-ordered, where *S* is the set of distinct types with no comparison form.

— A type *T* is *ST-EC-ordered* if *T* is *S*-ordered, where *S* is the set of structured types with EQUALS ONLY comparison form.

— A type *T* is *ST-FC-ordered* if *T* is *S*-ordered, where *S* is the set of structured types with FULL comparison form.

— A type *T* is *ST-NC-ordered* if *T* is *S*-ordered, where *S* is the set of structured types with no comparison form.

— A type *T* is *ST-ordered* if *T* is ST-EC-ordered, ST-FC-ordered, or ST-NC-ordered.

— A type *T* is *UDT-EC-ordered* if *T* is either DT-EC-ordered or ST-EC-ordered (UDT stands for "user-defined type").

— A type *T* is *UDT-FC-ordered* if *T* is either DT-FC-ordered or ST-FC-ordered

— A type *T* is *UDT-NC-ordered* if *T* is either DT-NC-ordered or ST-NC-ordered.

4.2.5 Properties of distinct

Two comparable values are distinct if they are capable of being distinguished within a given context.

Two null values are not distinct.

A null value and a non-null value are distinct.

Two non-null values are distinct if the General Rules of Subclause 8.15, "<distinct predicate>", return *True*.

The result of evaluating whether or not two comparable values are distinct is never *Unknown*. The result of evaluating whether or not two values that are not comparable (for example, values of a user-defined type that has no comparison type) are distinct is not defined.

4.3 Character strings

This Subclause is modified by Subclause 4.2, "Character strings", in ISO/IEC 9075-10.

4.3.1 Introduction to character strings

A character string is a sequence of characters. All the characters in a character string are taken from a single character set. A character string has a length, which is the number of characters in the sequence. The length is 0 (zero) or a positive integer.

A *character string type* is described by a character string type descriptor. A character string type descriptor contains:

— The name of the specific character string type (CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT; NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, and NATIONAL CHARACTER LARGE OBJECT are represented as CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT, respectively);

— The length or maximum length in characters of the character string type;

— The catalog name, schema name, and character set name of the character set of the character string type;

— The catalog name, schema name, and collation name of the collation of the character string type.

A *character large object type* is a character string type where the name of the specific character string type is CHARACTER LARGE OBJECT. A value of a character large object type is a *large object character string*.

The character set of a character string type may be specified explicitly or implicitly.

The <key word>s NATIONAL CHARACTER are used to specify the character type with an implementation-defined (IV099) character set. Special syntax (`N'string'`) is provided for representing literals in that character set.

With two exceptions, a character string expression is assignable only to sites of a character string type whose character set is the same. The exceptions are as specified in Subclause 4.3.8, "Universal character sets", and such other cases as may be implementation-defined (IA014). If a store assignment would result in the loss of non-<truncating whitespace> characters due to truncation, then an exception condition is raised. If a retrieval assignment or evaluation of a <cast specification> would result in the loss of characters due to truncation, then a warning condition is raised.

Character sets fall into three categories: those defined by national or international standards, those defined by SQL-implementations, and those defined by applications. The character sets defined by ISO/IEC 10646:2020 are known as *Universal Character Sets* (UCS) and their treatment is described in Subclause 4.3.8, "Universal character sets". Every character set contains the <space> character (equivalent to U+0020). An application defines a character set by assigning a new name to a character set from one of the first two categories. They can be defined to "reside" in any schema chosen by the application. Character sets defined by standards or by SQL-implementations reside in the Information Schema (named INFORMATION_SCHEMA) in each catalog, as do collations defined by standards and collations, transliterations, and transcodings defined by SQL-implementations.

NOTE 3 — The Information Schema is defined in ISO/IEC 9075-11.

4.3.2 Comparison of character strings

Two character strings are comparable if and only if either they have the same character set or there exists at least one collation that is applicable to both their respective character sets (which is possible only if the character sets share the same repertoire).

NOTE 4 — There are syntactic possibilities in which a comparison operation has two equally worthy collations available to compare two character string operands. Such syntax is a syntax error, even though the operands could be compared if one or the other of the two collations was designated as the collation to use. For example, if *T.C1* is a column whose collation is *COLLATION1* and *T.C2* is a column whose collation is some other collation *COLLATION2*, then

```
T.C1 > T.C2
```

is a syntax error, whereas as

```
T.C1 > T.C2 COLLATE COLLATION2
```

is not a syntax error.

A *collation* is defined by ISO/IEC 14651:2020 as "a process by which two strings are determined to be in exactly one of the relationships of less than, greater than, or equal to one another". Each collation known in an SQL-environment is applicable to one or more character sets, and for each character set, one or more collations are applicable to it, one of which is associated with it as its *character set collation*.

Anything that has a declared type can, if that type is a character string type, be associated with a collation applicable to its character set; this is known as a *declared type collation*. Every declared type that is a character string type has a collation derivation, this being either *none*, *implicit*, or *explicit*. The collation derivation of a declared type with a declared type collation that is explicitly or implicitly specified by a <data type> is *implicit*. If the collation derivation of a declared type that has a declared type collation is not *implicit*, then it is *explicit*. The collation derivation of an expression of character string type that has no declared type collation is *none*.

An operation that explicitly or implicitly involves character string comparison is a *character comparison operation*. At least one of the operands of a character comparison operation shall have a declared type collation.

There may be an SQL-session collation for some or all of the character sets known to the SQL-implementation (see Subclause 4.45, "SQL-sessions").

The collation used for a particular character comparison is specified by Subclause 9.15, "Collation determination".

The comparison of two character string expressions depends on the collation used for the comparison (see Subclause 9.15, "Collation determination"). When values of unequal length are compared, if the collation for the comparison has the NO PAD characteristic and the shorter value is equal to some prefix of the longer value, then the shorter value is considered less than the longer value. If the collation for the comparison has the PAD SPACE characteristic, for the purposes of the comparison, the shorter value is effectively extended to the length of the longer by concatenation of <space>s on the right.

For every character set, there is at least one collation.

4.3.3 Operations involving character strings

4.3.3.1 Regular expression syntaxes

This document utilizes two syntaxes for regular expressions, a POSIX-based regular expression syntax and the XQuery regular expression syntax.

The POSIX-based regular expression syntax is similar to the syntax specified in ISO/IEC/IEEE 9945, but is normatively defined in the General Rules of Subclause 8.6, "<similar predicate>".

The XQuery regular expression syntax is normatively defined in XQuery and XPath Functions and Operators 3.1.

4.3.3.2 Operators that operate on character strings and return character strings

<concatenation operator> is an operator, | |, that returns the character string made by joining its character string operands in the order given.

<character substring function> is a triadic function, SUBSTRING, that returns a string extracted from a given string according to a given numeric starting position and a given numeric length.

<regular expression substring function> is a triadic function, SUBSTRING, distinguished by the keywords SIMILAR and UESCAPE. It has three parameters: a source character string, a pattern string, and an escape character. It returns a result string extracted from the source character string by pattern matching using POSIX-based regular expressions.

— **Step 1**: The escape character is exactly one character in length. As indicated in Figure 1, "Operation of <regular expression substring function>", the escape character precedes two instances of <double quote> that are used to partition the pattern string into three subpatterns (identified as *R1*, *R2*, and *R3*).

— **Step 2**: If the source string *S* does not satisfy the predicate

```
'S' SIMILAR TO 'R1' || 'R2' || 'R3'
```

then the result is the null value.

— **Step 3**: Otherwise, *S* is partitioned into two substrings *S1* and *S23* such that *S1* is the shortest initial substring of *S* such that the following is true:

```
'S1' SIMILAR TO 'R1' AND
  'S23' SIMILAR TO '(' || 'R2' || 'R3' || ')'
```

— **Step 4**: Next, *S23* is partitioned into two substrings *S2* and *S3* such that *S3* is the shortest final substring such that the following is true:

```
'S2' SIMILAR TO 'R2' AND 'S3' SIMILAR TO 'R3'
```

The result of the <regular expression substring function> is *S2*.

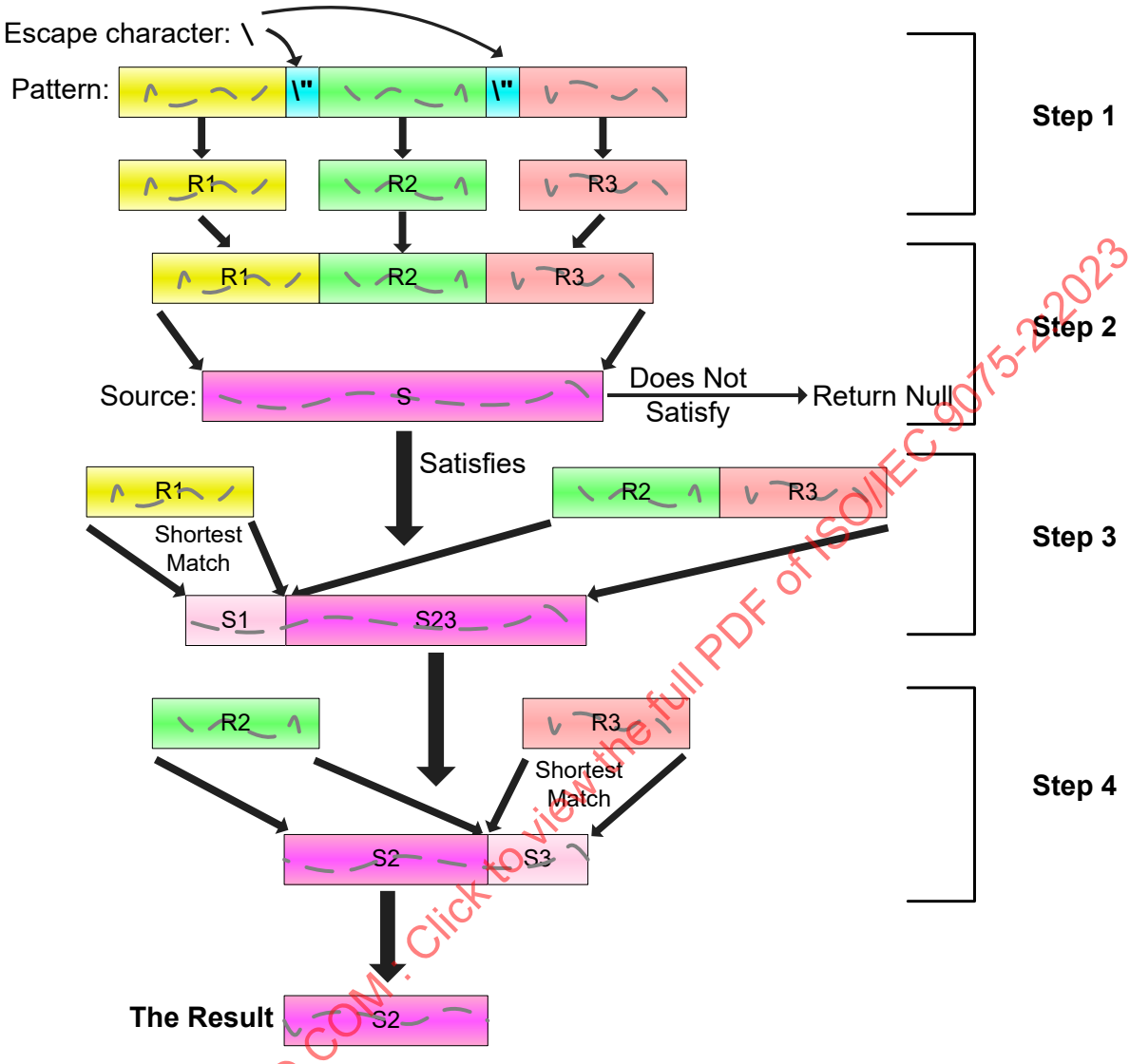

Figure 1 — Operation of <regular expression substring function>

<regex substring function> is a function, SUBSTRING_REGEX, that searches a string for an XQuery regular expression pattern and returns one occurrence of the matching substring.

<regex transliteration> is a function, TRANSLATE_REGEX, that searches a string for an XQuery regular expression pattern and returns the string with either one or every occurrence of the XQuery regular expression pattern replaced using an XQuery replacement string.

<character overlay function> is a function, OVERLAY, that modifies a string argument by replacing a given substring of the string, which is specified by a given numeric starting position and a given numeric length, with another string (called the replacement string). When the length of the substring is zero, nothing is removed from the original string and the string returned by the function is the result of inserting the replacement string into the original string at the starting position.

<fold> is a pair of functions for converting all the lower-case and title case characters in a given string to upper-case (UPPER) or all the upper-case and title case characters to lower-case (LOWER). A lower-case character is a character in the Unicode General Category class "Ll" (lower-case letters). An upper-case character is a character in the Unicode General Category class "Lu" (upper-case letters). A title case character is a character in the Unicode General Category class "Lt" (title-case letters).

NOTE 5 — Case correspondences are not always one-to-one: the result of case folding is sometimes of a different length in characters than the source string. For example, U+00DF, "ß", Latin Small Letter Sharp S, becomes "SS" when folded to upper-case.

<transcoding> is a function that invokes an installation-supplied transcoding to return a character string *S2* derived from a given character string *S1*. It is intended, though not enforced by this document, that *S2* be exactly the same sequence of characters as *S1*, but encoded according to some different character encoding form. A typical use might be to convert a character string from two-octet UCS to one-octet Latin1 or *vice versa*.

<single-character trim function> is a function that returns its first string argument with leading and/or trailing pad characters removed. The second argument indicates whether leading, or trailing, or both leading and trailing pad characters shall be removed. The third argument specifies the pad character that is to be removed.

<multi-character trim function> is a function that returns its first string argument with leading pad characters, trailing pad characters, or both leading and trailing pad characters removed. The second argument specifies the pad characters that are to be removed. In contrast to the <single-character trim function>, this function accepts a sequence of characters as the second argument and will remove all leading, trailing, or both occurrences of any of these characters, regardless of their ordering.

<pad function> is a function that returns its first string argument padded to the total length of characters specified by the second argument with the sequence of characters specified by the third argument.

<character transliteration> is a function for changing each character of a given string according to some many-to-one or one-to-one mapping between two not necessarily distinct character sets. The mapping, rather than being specified as part of the function, is some external function identified by a <transliteration name>.

For any pair of character sets, there are zero or more transliterations that may be invoked by a <character transliteration>. A transliteration is described by a transliteration descriptor. A transliteration descriptor includes:

— The name of the transliteration;

— The name of the character set from which it translates;

— The name of the character set to which it translates;

— The specific name of the SQL-invoked routine that performs the transliteration.

4.3.3.3 Other operators involving character strings

<length expression> returns the length of a given character string, as an exact numeric value, in characters or octets according to the choice of function.

<position expression> determines the first position, if any, at which one string, *S1*, occurs within another, *S2*. If *S1* is of length zero, then it occurs at position 1 (one) for any value of *S2*. If *S1* does not occur in *S2*, then zero is returned. The declared type of a <position expression> is exact numeric.

<like predicate> uses the triadic operator LIKE (or the inverse, NOT LIKE), operating on three character strings and returning a Boolean. LIKE determines whether or not a character string "matches" a given "pattern" (also a character string). The characters <percent> and <underscore> have special meaning when they occur in the pattern. The optional third argument is a character string containing exactly one

character, known as the “escape character”, for use when a <percent>, <underscore>, or the “escape character” itself is required in the pattern without its special meaning.

<similar predicate> uses the triadic operator SIMILAR (or the inverse, NOT SIMILAR), operating on three character strings and returning a Boolean. SIMILAR determines whether or not a character string “matches” a given “pattern” (also a character string) using a POSIX-based regular expression. The pattern is in the form of a “regular expression”. In this regular expression, certain characters (<left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>) have a special meaning. The optional third argument specifies the “escape character”, for use when one of the special characters or the “escape character” itself is required in the pattern without its special meaning.

<regex occurrences function> is a function, OCCURRENCES_REGEX, that searches a string for an XQuery regular expression pattern and returns an integer indicating the count of occurrences of the matched substring.

<regex position expression> is a function, POSITION_REGEX, that searches a string for an XQuery regular expression pattern and returns an integer indicating the beginning position, or 1 (one) plus the ending position, of one occurrence of the matched substring.

<regex like predicate> is a predicate, LIKE_REGEX, that performs XQuery regular expression matching.

4.3.3.4 Operations involving large object character strings

Large object character strings cannot be operated on by all string operations. Large object character strings can, however, be operated on by the following operations:

— <null predicate>;

— <like predicate>;

— <similar predicate>;

— <position expression>;

— <regex like predicate>;

— <regex position expression>;

— <regex occurrences function>;

— <regex substring function>;

— <regex transliteration>;

— <comparison predicate> with an <equals operator> or <not equals operator>;

— <quantified comparison predicate> with the <equals operator> or <not equals operator>.

As a result of these restrictions, large object character strings cannot be used in (among other places):

— predicates other than those listed above and the <exists predicate>;

— <general set function>;

— <group by clause>;

— <order by clause>;

— <unique constraint definition>;

— <referential constraint definition>;

— <select list> of a <query specification> that has a <set quantifier> of DISTINCT;

— UNION, INTERSECT, and EXCEPT;

— columns used for matching when forming a <joined table>.

All the operations described within Subclause 4.3.3.2, "Operators that operate on character strings and return character strings", and Subclause 4.3.3.3, "Other operators involving character strings", are supported for large object character strings.

4.3.4 Character repertoires

An SQL-implementation supports one or more character repertoires. These character repertoires may be defined by a standard or be implementation-defined (IA015).

A character repertoire is described by a character repertoire descriptor. A character repertoire descriptor includes:

— The name of the character repertoire;

— The name of the default collation for the character repertoire.

The following character repertoire names are specified as part of the ISO/IEC 9075 series:

— SQL_CHARACTER is a character repertoire that consists of the <SQL language character>s as specified in Subclause 5.1, "<SQL terminal character>". The name of the default collation is SQL_CHARACTER;

— GRAPHIC_IRV is the character repertoire that consists of the 95-character graphic subset of the International Reference Version (IRV) as specified in ISO/IEC 646. Its repertoire is a proper superset of that of SQL_CHARACTER. The name of the default collation is GRAPHIC_IRV;

— LATIN1 is the character repertoire defined in ISO/IEC 8859-1. The name of the default collation is LATIN1;

— ISO8BIT is the character repertoires formed by combining the character repertoire specified by ISO/IEC 8859-1 and the "control characters" specified by ISO/IEC 6429. The repertoire consists of all 255 characters, each consisting of exactly 8 bits, as, including all control characters and all graphic characters except the character corresponding to the numeric value 0 (zero). The name of the default collation is ISO8BIT;

— UCS is the Universal Character Set repertoire specified by ISO/IEC 10646:2020. It is implementation-defined (ID007) whether the name of the default collation is UCS_BASIC or UNICODE;

— SQL_TEXT is a character repertoire that is an implementation-defined (IA016) subset of the repertoire of the Universal Character Set that includes every <SQL language character> and every character in every character set supported by the SQL-implementation. The name of the default collation is SQL_TEXT;

— SQL_IDENTIFIER is an implementation-defined (IA029) character repertoire consisting of the <SQL language character>s and all other characters that the SQL-implementation supports for use in <regular identifier>s. The name of the default collation is SQL_IDENTIFIER.

Each character repertoire includes one or more characters defined to be whitespace; see Clause 3, "Terms and definitions".

NOTE 6 — The normative provisions of this document impose no requirement that any character repertoire have equivalents for any of these characters except U+0020 (<space>); however, by reference to this definition of whitespace, they do impose the requirement that every equivalent for one of these is recognized as a whitespace character.

NOTE 7 — If and when the set of characters with the Unicode property White_Space is modified to add new characters or remove characters, those modifications can implemented by SQL-implementations without affecting conformance to this document.

4.3.5 Character encoding forms

An SQL-implementation supports one or more character encoding forms for each character repertoire that it supports. These character encoding forms may be defined by a standard or be implementation-defined (IA017).

A character encoding form is described by a character encoding form descriptor. A character encoding form descriptor includes:

— The name of the character encoding form;

— The name of the character repertoire to which it is applicable.

The following character encoding form names are specified as part of the ISO/IEC 9075 series:

— SQL_CHARACTER is an implementation-defined (IV116) character encoding form. It is applicable to the SQL_CHARACTER character repertoire;

— GRAPHIC_IRV is the character encoding form in which the coded representation of each character is specified in ISO/IEC 646. It is applicable to the GRAPHIC_IRV character repertoire;

— LATIN1 is the character encoding form specified in ISO/IEC 8859-1. It is applicable to the LATIN1 character repertoire;

— ISO8BIT is the character encoding form specified in ISO/IEC 8859-1, augmented by ISO/IEC 6429. When restricted to the LATIN1 characters, it is the same character encoding form as LATIN1. It is applicable to the ISO8BIT character repertoire;

— UTF32 is the character encoding form specified in ISO/IEC 10646:2020, in which each character is encoded as four octets. It is applicable to the UCS character repertoire;

— UTF16 is the character encoding form specified in ISO/IEC 10646:2020, in which each character is encoded as two or four octets. It is applicable to the UCS character repertoire;

— UTF8 is the character encoding form specified in ISO/IEC 10646:2020, in which each character is encoded as from one to four octets. It is applicable to the UCS character repertoire;

— SQL_TEXT is an implementation-defined (IV117) character encoding form. It is applicable to the SQL_TEXT character repertoire;

— SQL_IDENTIFIER is an implementation-defined (IV118) character encoding form. It is applicable to the SQL_IDENTIFIER character repertoire.

If an SQL-implementation supplies more than one character encoding form for a particular character repertoire, then it shall specify a precedence ordering of the character encoding forms of that character repertoire. The precedence of character encoding forms applicable to the UCS character repertoire and defined in this document is:

UTF8 ≺ UTF16 ≺ UTF32

4.3.6 Collations

An SQL-implementation supports one or more collations for each character repertoire that it supports, and one or more collations for each character set that it supports. A collation is described by a collation descriptor. A collation descriptor includes:

— The name of the collation;

— The name of the character repertoire to which it is applicable;

— A list of the names of the character sets to which the collation can be applied;

— Whether the collation has the NO PAD or the PAD SPACE characteristic.

The following supported collation names are specified as part of the ISO/IEC 9075 series:

— SQL_CHARACTER is an implementation-defined (IA003) collation. It is applicable to the SQL_CHARACTER character repertoire;

— GRAPHIC_IRV is a collation in which the ordering is determined by treating the code points defined by ISO/IEC 646 as unsigned integers. It is applicable to the GRAPHIC_IRV character repertoire;

— LATIN1 is a collation in which the ordering is determined by treating the code points defined by ISO/IEC 8859-1 as unsigned integers. It is applicable to the LATIN1 character repertoire;

— ISO8BIT is a collation in which the ordering is determined by treating the code points defined by ISO/IEC 8859-1 as unsigned integers. When restricted to the LATIN1 characters, it produces the same collation as LATIN1. It is applicable to the ISO8BIT character repertoire;

— UCS_BASIC is a collation in which the ordering is determined entirely by the Unicode scalar values of the characters in the strings being sorted. It is applicable to the UCS character repertoire. Since every character repertoire is a subset of the UCS repertoire, the UCS_BASIC collation is potentially applicable to every character set;

NOTE 8 — The Unicode scalar value of a character is its code point treated as an unsigned integer.

— UNICODE is the collation that conforms to ISO/IEC 14651:2020 without delta. It is applicable to the UCS character repertoire. Since every character repertoire is a subset of the UCS repertoire, the UNICODE collation is potentially applicable to every character set;

— SQL_TEXT is an implementation-defined (IA003) collation. It is applicable to the SQL_TEXT character repertoire;

— SQL_IDENTIFIER is an implementation-defined (IA003) collation. It is applicable to the SQL_IDENTIFIER character repertoire.

The collations, including standard defined collations, supported by the SQL-implementation are implementation-defined (IA003).

4.3.7 Character sets

An SQL <character set specification> allows a reference to a character set name defined by a standard, an SQL-implementation, or a user.

A character set is described by a character set descriptor. A character set descriptor includes:

— The name of the character set;

— The name of the character repertoire for the character set;

— The name of the character encoding form for the character set;

— The name of the default collation for the character set.

The following SQL supported character set names, many of which are further described in Table 1, "Overview of character sets", are specified as part of the ISO/IEC 9075 series:

— SQL_CHARACTER is a character set whose repertoire is SQL_CHARACTER and whose character encoding form is SQL_CHARACTER. The name of its default collation is SQL_CHARACTER;

— GRAPHIC_IRV is a character set whose repertoire is GRAPHIC_IRV and whose character encoding form is GRAPHIC_IRV. The name of its default collation is GRAPHIC_IRV;

— ASCII_GRAPHIC is a synonym for GRAPHIC_IRV;

— LATIN1 is a character set whose repertoire is LATIN1 and whose character encoding form is LATIN1. The name of its default collation is LATIN1;

— ISO8BIT is a character set whose repertoire is ISO8BIT and whose character encoding form is ISO8BIT. The name of its default collation is ISO8BIT;

— ASCII_FULL is a synonym for ISO8BIT;

— UTF32 is a character set whose repertoire is UCS and whose character encoding form is UTF32. It is implementation-defined (ID011) whether the name of its default collation is UCS_BASIC or UNICODE;

— UTF16 is a character set whose repertoire is UCS and whose character encoding form is UTF16. It is implementation-defined (ID012) whether the name of its default collation is UCS_BASIC or UNICODE;

— UTF8 is a character set whose repertoire is UCS and whose character encoding form is UTF8. It is implementation-defined (ID013) whether the name of its default collation is UCS_BASIC or UNICODE;

— SQL_TEXT is a character set whose repertoire is SQL_TEXT and whose character encoding form is SQL_TEXT. The name of its default collation is SQL_TEXT;

— SQL_IDENTIFIER is a character set whose repertoire is SQL_IDENTIFIER and whose character encoding form is SQL_IDENTIFIER. The name of its default collation is SQL_IDENTIFIER;

The result of evaluating a character string expression whose most specific type has character set *CS* is constrained to consist of characters drawn from the character repertoire of *CS*.

Table 1 — Overview of character sets

Character Set	Character Repertoire	Character Encoding Form	Collation	Synonym
GRAPHIC_IRV	GRAPHIC_IRV	GRAPHIC_IRV	GRAPHIC_IRV	ASCII_GRAPHIC
ISO8BIT	ISO8BIT	ISO8BIT	ISO8BIT	ASCII_FULL
LATIN1	LATIN1	LATIN1	LATIN1	
SQL_CHARACTER	SQL_CHARACTER	SQL_CHARACTER	SQL_CHARACTER	
SQL_TEXT	SQL_TEXT	SQL_TEXT	SQL_TEXT	
SQL_IDENTIFIER	SQL_IDENTIFIER	SQL_IDENTIFIER	SQL_IDENTIFIER	
UTF16	UCS	UTF16	UCS_BASIC or UNICODE	
UTF32	UCS	UTF32	UCS_BASIC or UNICODE	
UTF8	UCS	UTF8	UCS_BASIC or UNICODE	

NOTE 9 — The provision of additional character sets and/or additional character encoding forms and collations for character sets defined in this document has no effect on conformance of SQL-implementations.

4.3.8 Universal character sets

A *UCS string* is a character string whose character repertoire is UCS and whose character encoding form is one of UTF8, UTF16, or UTF32. Any two UCS strings are comparable.

An SQL-implementation may assume that all UCS strings are normalized in one of Normalization Form C (NFC), Normalization Form D (NFD), Normalization Form KC (NFKC), or Normalization Form KD (NFKD), as specified by ISO/IEC 10646:2020. <normalized predicate> may be used to verify the normalization form to which a particular UCS string conforms. Applications may also use <normalize function> to enforce a particular <normal form>. With the exception of <normalize function> and <normalized predicate>, the result of any operation on an unnormalized UCS string is implementation-defined (IV100).

Conversion of UCS strings from one character set to another is automatic.

Detection of a noncharacter code point in a UCS-string causes an exception condition to be raised. The detection of an unassigned code point does not.

4.4 Binary strings

4.4.1 Introduction to binary strings

A binary string is a sequence of octets that does not have either a character set or collation associated with it.

A binary string type is described by a binary string type descriptor. A binary string type descriptor contains:

— The name of the data type (BINARY, BINARY VARYING, or BINARY LARGE OBJECT);

— The length or maximum length in octets of the binary string type.

A binary string is assignable only to sites of binary string type. If a store assignment would result in the loss of non-zero octets due to truncation, then an exception condition is raised. If a retrieval assignment would result in the loss of octets due to truncation, then a warning condition is raised.

4.4.2 Binary string comparison

All binary string values are comparable. When binary large object string values are compared, they shall have exactly the same length (in octets) to be considered equal. Binary large object string values can be compared only for equality.

For binary string values other than binary large object string values, it is implementation-defined (IA022) whether trailing X'00's are considered significant when comparing two binary string values that are otherwise equivalent.

4.4.3 Operations involving binary strings

4.4.3.1 Operators that operate on binary strings and return binary strings

<binary concatenation> is an operator, | |, that returns a binary string by joining its binary string operands in the order given.

<binary substring function> is a triadic function identical in syntax and semantics to <character substring function> except that its arguments and the returned value are all binary strings.

<binary overlay function> is a function identical in syntax and semantics to <character overlay function> except that the first argument, second argument, and returned value are all binary strings.

<binary trim function> is a function identical in syntax and semantics to <single-character trim function> except that its arguments and the returned value are all binary strings.

4.4.3.2 Other operators involving binary strings

<length expression> returns the length of a given binary string, as an exact numeric value, in octets.

<position expression> when applied to binary strings is identical in syntax and semantics to the corresponding operation on character strings except that the operands are binary strings.

<like predicate> when applied to binary strings is identical in syntax and semantics to the corresponding operation on character strings except that the operands are binary strings.

Binary large object strings cannot be used in:

— Predicates other than <comparison predicate> with an <equals operator> or a <not equals operator>, <quantified comparison predicate> with an <equals operator> or a <not equals operator>, and <exists predicate>;

— <general set function>;

— <group by clause>;

— <order by clause>;

— <unique constraint definition>;

— <referential constraint definition>;

— <select list> of a <query specification> that has a <set quantifier> of DISTINCT;

— UNION, INTERSECT, and EXCEPT;

— Columns used for matching when forming a <joined table>.

4.5 Numbers

This Subclause is modified by Subclause 4.3, "Numbers", in ISO/IEC 9075-15.

4.5.1 Introduction to numbers

A number is either an exact numeric value, an approximate numeric value, or a decimal floating-point value. Any two numbers are comparable.

A numeric type is described by a numeric type descriptor. A numeric type descriptor contains:

— The name of the specific numeric type (NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DOUBLE PRECISION, or DECFLOAT);

— The implemented precision of the numeric type;

— For an exact numeric type, the implemented scale of the numeric type;

— An indication of whether the precision (and scale) are expressed in decimal or binary terms;

— The name by which the numeric type was declared (but the corresponding full name if INT or DEC was specified);

— The explicit declared precision, if any, of the numeric type;

— For an exact numeric type, the explicit declared scale, if any, of the numeric type.

If <precision> or <scale> is not specified explicitly, then the corresponding element of the descriptor effectively contains the null value.

An SQL-implementation is permitted to regard certain <exact numeric type>s as equivalent, if they have the same precision, scale, and radix, as permitted by the Syntax Rules of Subclause 6.1, "<data type>". When two or more <exact numeric type>s are equivalent, the SQL-implementation chooses one of these equivalent <exact numeric type>s as the *normal form* representing that equivalence class of <exact numeric type>s. The normal form determines the name of the exact numeric type in the numeric type descriptor.

Similarly, an SQL-implementation is permitted to regard certain <approximate numeric type>s as equivalent, as permitted by the Syntax Rules of Subclause 6.1, "<data type>", in which case the SQL-implementation chooses a *normal form* to represent each equivalence class of <approximate numeric type> and the normal form determines the name of the approximate numeric type.

For every numeric type, the least value is less than zero and the greatest value is greater than zero.

4.5.2 Characteristics of numbers

An exact numeric type has a precision *P* and a scale *S*. *P* is a positive integer that determines the number of significant digits in a particular radix *R*, where *R* is either 2 or 10. *S* is a non-negative integer. Every value of an exact numeric type of scale *S* is of the form $n \times 10^{-S}$, where *n* is an integer such that $-R^P \leq n < R^P$.

NOTE 10 — Not every value in that range is necessarily a value of the type in question.

An exact numeric value consists of either one or more decimal digits followed by an optional decimal point and zero or more decimal digits or a decimal point followed by one or more decimal digits.

Approximate numeric and decimal floating-point values consist of a mantissa and an exponent. The mantissa is a signed numeric value, and the exponent is a signed integer that specifies the magnitude of the mantissa. Approximate numeric and decimal floating-point values have a precision. The precision of approximate numeric values is a positive integer that specifies the number of significant binary digits in the mantissa. The precision of decimal floating-point values is a positive integer that specifies the number of significant decimal digits in the mantissa. The value of an approximate numeric or decimal floating-point type is the mantissa multiplied by a factor determined by the exponent.

An <exact numeric literal> *ENL* consists of either an <unsigned integer>, a <period> followed by an <unsigned decimal integer>, or an <unsigned decimal integer> followed by a <period> and an optional <unsigned decimal integer>. The declared type of *ENL* is an exact numeric type.

An <approximate numeric literal> *ANL* consists of a <mantissa> that is an <exact numeric literal>, the letter 'E' or 'e', and an <exponent> that is a <signed decimal integer>. It is implementation-defined (IV119) whether the declared type of *ANL* is an approximate numeric type or the decimal floating-point type. If *M* is the value of the <mantissa> and *E* is the value of the <exponent>, then $M * 10^E$ is the *apparent value* of *ANL*. If the declared type of *ANL* is an approximate numeric type, then the actual value of *ANL* is approximately the apparent value of *ANL*, according to implementation-defined (IA065) rules. If the declared type of *ANL* is the decimal floating-point type, the actual value of *ANL* is either exactly or approximately the apparent value of *ANL*, according to implementation-defined (IA065) rules.

A number is assignable only to sites of numeric type. If an assignment of some number would result in a loss of its most significant digit, an exception condition is raised. If least significant digits are lost, implementation-defined (IA195) rounding or truncating occurs, with no exception condition being raised. The rules for arithmetic are specified in Subclause 6.30, "<numeric value expression>".

Whenever a numeric value is assigned to an exact numeric value site, an approximation of its value that preserves leading significant digits after rounding or truncating is represented in the declared type of the target. The value is converted to have the precision and scale of the target. The choice of whether to truncate or round is implementation-defined (IA195).

An approximation obtained by truncation of a numeric value *N* for an exact numeric type *T* is a value *V* in *T* such that *N* is not closer to zero than *V* and there is no value in *T* between *V* and *N*.

An approximation obtained by rounding of a numeric value *N* for an exact numeric type *T* is a value *V* in *T* such that the absolute value of the difference between *N* and the numeric value of *V* is not greater than half the absolute value of the difference between two successive numeric values in *T*. If there is more than one such value *V*, then it is implementation-defined (IA192) which one is taken.

All numeric values between the smallest and the largest value, inclusive, in a given exact numeric type have an approximation obtained by rounding or truncation for that type; it is implementation-defined (IA196) which other numeric values have such approximations.

An approximation obtained by truncation or rounding of a numeric value *N* for an approximate numeric type *T* or the decimal floating-point type *T* is a value *V* in *T* such that there is no numeric value in *T* distinct from that of *V* that lies between the numeric value of *V* and *N*, inclusive. If there is more than one such value *V* then it is implementation-defined (IA192) which one is taken. It is implementation-defined (IA193) which numeric values have approximations obtained by rounding or truncation for a given approximate numeric type or a given decimal floating-point type.

Whenever a numeric value is assigned to an approximate numeric value site or a decimal floating-point value site, an approximation of its value is represented in the declared type of the target. The value is converted to have the precision of the target.

Operations on numbers are performed according to the normal rules of arithmetic, within implementation-defined (IA194) limits, except as provided for in Subclause 6.30, "<numeric value expression>".

4.5.3 Operations involving numbers

This Subclause is modified by Subclause 4.3.1, "Operations involving numbers", in ISO/IEC 9075-15.

15 As well as the usual arithmetic operators, plus, minus, times, divide, unary plus, and unary minus, there are the following functions that return numbers:

— <position expression> (see Subclause 4.3.3, "Operations involving character strings", and Subclause 4.4.3, "Operations involving binary strings") takes two strings as arguments and returns an integer;

— <length expression> (see Subclause 4.3.3, "Operations involving character strings", and Subclause 4.4.3, "Operations involving binary strings") operates on a string argument and returns an integer;

— <extract expression> (see Subclause 4.7.4, "Operations involving datetimes and intervals") operates on a datetime or interval argument and returns an exact numeric;

— <cardinality expression> (see Subclause 4.12.5, "Operations involving arrays", and Subclause 4.12.6, "Operations involving multisets") operates on a collection argument and returns an integer;

— <max cardinality expression> (see Subclause 4.12.5, "Operations involving arrays") operates on an array argument and returns an integer;

— <absolute value expression> operates on a numeric argument and returns its absolute value in the same most specific type;

— <modulus expression> operates on two exact numeric arguments with scale 0 (zero) and returns the modulus (remainder) of the first argument divided by the second argument as an exact numeric with scale 0 (zero);

— <trigonometric function> computes one of sine, cosine, tangent, hyperbolic sine, hyperbolic cosine, hyperbolic tangent, inverse sine, inverse cosine, and inverse tangent of its argument;

— <general logarithm function> operates on two numeric arguments. The first argument specifies the base with which to compute the logarithm of its second argument;

— <common logarithm> computes the logarithm with base 10 of its argument;

— <natural logarithm> computes the natural logarithm of its argument;

— <exponential function> computes the exponential function, that is, *e*, (the base of natural logarithms) raised to the power equal to its argument;

— <power function> raises its first argument to the power of its second argument;

— <square root> computes the square root of its argument;

— <floor function> computes the greatest integer less than or equal to its argument;

— <ceiling function> computes the least integer greater than or equal to its argument;

— <width bucket function> is a function of four arguments, returning an integer between 0 (zero) and the value of the final argument plus 1 (one), by assigning the first argument to an equi-width partitioning of the range of numbers between the second and third arguments. Values outside the range between the second and third arguments are assigned to either 0 (zero) or the value of the final argument plus 1 (one);

NOTE 11 — The semantics of <width bucket function> are illustrated in Figure 2, "Illustration of WIDTH_BUCKET Semantics".

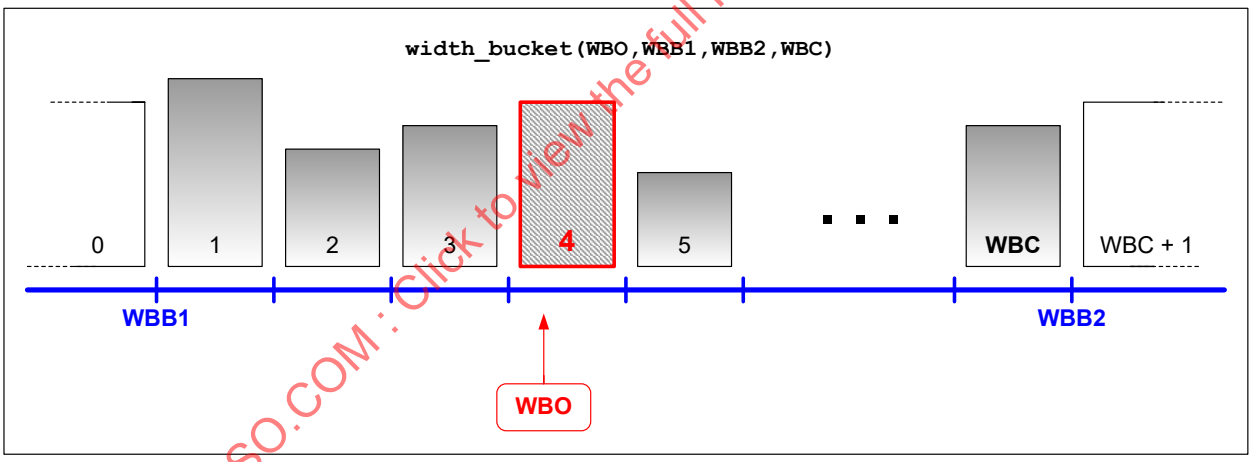

Figure 2 — Illustration of WIDTH_BUCKET Semantics

4.6 Boolean types

4.6.1 Introduction to Boolean types

The Boolean data type comprises the distinct truth values *True* and *False*. Unless prohibited by a NOT NULL constraint, the Boolean data type also supports the truth value *Unknown* as the null value. This specification does not make a distinction between the null value of the Boolean data type and the truth value *Unknown* that is the result of an SQL <predicate>, <search condition>, or <boolean value expression>; they may be used interchangeably to mean exactly the same thing.

The Boolean data type is described by the Boolean data type descriptor. The Boolean data type descriptor contains:

— The name of the Boolean data type (BOOLEAN).

4.6.2 Comparison and assignment of Booleans

All Boolean values and SQL truth values are comparable and all are assignable to a site of type Boolean. The value *True* is greater than the value *False*, and any comparison involving the null value or an *Unknown* truth value will return an *Unknown* result. The values *True* and *False* may be assigned to any site having a Boolean data type; assignment of *Unknown*, or the null value, is subject to the nullability characteristic of the target.

4.6.3 Operations involving Booleans

4.6.3.1 Operations on Booleans that return Booleans

The monadic Boolean operator NOT and the dyadic Boolean operators AND and OR take Boolean operands and produce a Boolean result (see Table 15, "Truth table for the AND Boolean operator", and Table 16, "Truth table for the OR Boolean operator").

4.6.3.2 Other operators involving Booleans

Every SQL <predicate>, <search condition>, and <boolean value expression> may be considered as an operator that returns a Boolean result.

4.7 Datetimes and intervals

4.7.1 Introduction to datetimes and intervals

A datetime data type is described by a datetime data type descriptor. An interval data type is described by an interval data type descriptor.

A datetime data type descriptor contains:

— The name of the specific datetime data type (DATE, TIME WITHOUT TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE);

— The value of the <time fractional seconds precision>, if it is a TIME WITHOUT TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE type.

An interval data type descriptor contains:

— The name of the interval data type (INTERVAL);

— An indication of whether the interval data type is a year-month interval or a day-time interval;

— The <interval qualifier> that describes the precision of the interval data type.

Values of interval data types are always signed.

Every datetime or interval data type has an implied *length in positions*. Let *D* denote a value in some datetime or interval data type *DT*. The length in positions of *DT* is constant for all *D*. The length in positions is the number of characters from the character set SQL_TEXT that it would take to represent any value in a given datetime or interval data type.

An approximation obtained by rounding of a datetime or interval value *D* for a datetime type or interval type *T* is a value *V* in *T* such that the absolute value of the difference between *D* and the numeric value of *V* is not greater than half the absolute value of the difference between two successive datetime or interval values in *T*. If there is more than one such value *V*, then it is implementation-defined (IA197) which one is taken.

NOTE 12 — No host language for which a binding is specified in this document has a data type corresponding to the datetime and interval data types specified by this document. Whenever an application written in one of those languages must transfer values of those data types from the SQL-implementation to the host language program, those values must be

transformed to values of some data type for which the host language has a corresponding data type. Typically, the rules of <cast specification> are used to transform values of those data types to values of some character string type. Similar solutions are required when transferring such values from a host program into the SQL-implementation.

4.7.2 Datetimes

Table 2, "Fields in datetime values", specifies the fields that can make up a datetime value; a datetime value is made up of a subset of those fields. Not all of the fields shown are required to be in the subset, but every field that appears in the table between the first included <primary datetime field> and the last included <primary datetime field> shall also be included. If either <time zone field> is in the subset, then both of them shall be included.

Table 2 — Fields in datetime values

Keyword	Meaning
YEAR	Calendar year
MONTH	Calendar month
DAY	Calendar day
HOUR	Clock hour
MINUTE	Clock minute
SECOND	Clock second, possibly possibly including a decimal fraction
TIMEZONE_HOUR	Hour value of time zone displacement
TIMEZONE_MINUTE	Minute value of time zone displacement

There is an ordering of the significance of <primary datetime field>s. This is, from most significant to least significant: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

The <primary datetime field>s other than SECOND contain non-negative integer values, constrained by the rules for dates using the Gregorian calendar, as defined in ISO 8601-1:2019. SECOND, however, can be defined to have a <time fractional seconds precision> that indicates the number of decimal digits maintained following the decimal point in the seconds value, a non-negative exact numeric value.

There are three classes of datetime data types defined within this document:

— DATE — contains the <primary datetime field>s YEAR, MONTH, and DAY;

— TIME — contains the <primary datetime field>s HOUR, MINUTE, and SECOND;

— TIMESTAMP — contains the <primary datetime field>s YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Items of type datetime are comparable only if they have the same <primary datetime field>s.

A datetime data type that specifies WITH TIME ZONE is a data type that is *datetime with time zone*, while a datetime data type that specifies WITHOUT TIME ZONE is a data type that is *datetime without time zone*.

The surface of the earth is divided into zones, called time zones, in which every correct clock tells the same time, known as *local time*. Local time is equal to UTC (Coordinated Universal Time) plus the *time zone displacement*. A time zone displacement is effectively a positive or negative HOUR TO MINUTE interval value. The *minimum permitted negative time zone displacement* is implementation-defined (IL061),

but shall not be greater than the <interval literal> INTERVAL '−12:00' HOUR TO MINUTE. The *maximum permitted positive time zone displacement* is implementation-defined (IL062), but shall not be less than the <interval literal> INTERVAL '+14:00' HOUR TO MINUTE. The time zone displacement is constant throughout a time zone, and may change at the beginning and end of daylight saving time, where applicable.

A datetime value, of data type TIME WITHOUT TIME ZONE or TIMESTAMP WITHOUT TIME ZONE, may represent a local time, whereas a datetime value of data type TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE represents UTC.

On occasion, UTC is adjusted by the omission of a second or the insertion of a "leap second" in order to maintain synchronization with sidereal time. This implies that sometimes, but very rarely, a particular minute will contain exactly 59, 61, or 62 seconds. Whether an SQL-implementation supports leap seconds, and the consequences of such support for date and interval arithmetic, is implementation-defined (IA198).

For the convenience of users, whenever a datetime value with time zone is to be implicitly derived from one without (for example, in a simple assignment operation), SQL assumes the value without time zone to be local, subtracts the current default time zone displacement of the SQL-session from it to give UTC, and associates that time zone displacement with the result.

Conversely, whenever a datetime value without time zone is to be implicitly derived from one with, SQL assumes the value with time zone to be UTC, adds the time zone displacement to it to give local time, and the result, without any time zone displacement, is local.

The preceding principles, as implemented by <cast specification>, result in data type conversions between the various datetime data types, as summarized in Table 3, "Datetime data type conversions".

Table 3 — Datetime data type conversions

	to DATE	**to TIME WITHOUT ZONE**	**to TIME WITH TIME ZONE**	**to TIMESTAMP WITHOUT TIME ZONE**	**to TIMESTAMP WITH TIME ZONE**
from DATE	*trivial*	*not supported*	*not supported*	Copy year, month, and day; set hour, minute, and second to 0 (zero)	*SV* ⇒ TSw/oTZ ⇒ TSw/TZ
from TIME WITHOUT TIME ZONE	*not supported*	*trivial*	*TV.UTC* = *SV* − *STZD* (modulo 24); *TV.TZ* = *STZD*	Copy date fields from CUR-RENT_DATE and time fields from *SV*	*SV* ⇒ TSw/oTZ ⇒ TSw/TZ
from TIME WITH TIME ZONE	*not supported*	*SV.UTC* + *SV.TZ* (modulo 24)	*trivial*	*SV* ⇒ TSw/TZ ⇒ TSwo/TZ	Copy date fields from CUR-RENT_DATE and time and time zone fields from *SV*

	to DATE	**to TIME WITHOUT ZONE**	**to TIME WITH TIME ZONE**	**to TIMESTAMP WITHOUT TIME ZONE**	**to TIMESTAMP WITH TIME ZONE**
from TIMESTAMP WITHOUT TIME ZONE	Copy date fields from *SV*	Copy time fields from *SV*	*SV* ⇒ TSw/TZ ⇒ TIMEw/TZ	*trivial*	*TV.UTC* = *SV* – *STZD*; *TV.TZ* = *STZD*
from TIMESTAMP WITH TIME ZONE	*SV* ⇒ TSw/oTZ ⇒ DATE	*SV* ⇒ TSw/oTZ ⇒ TIMEw/oTZ	Copy time and time zone fields from *SV*	*SV.UTC* + *SV.TZ*	*trivial*

SV: the source value

TV: the target value

UTC: the UTC component of *SV* or *TV* (if and only if the source or target has time zone)

TZ: the time zone displacement of *SV* or *TV* (if and only if the source or target has time zone)

STZD: the SQL-session default time zone displacement

⇒: cast from the type preceding the arrow to the type following the arrow, "TIMEw/TZ" is "TIME WITH TIME ZONE", "TIMEw/oTZ" is "TIME WITHOUT TIME ZONE", "TSw/TZ" is "TIMESTAMP WITH TIME ZONE", and "TSw/oTZ" is "TIMESTAMP WITHOUT TIME ZONE"

A datetime is assignable to a site only if the source and target of the assignment are both of type DATE, or both of type TIME (regardless whether WITH TIME ZONE or WITHOUT TIME ZONE is specified or implicit), or both of type TIMESTAMP (regardless whether WITH TIME ZONE or WITHOUT TIME ZONE is specified or implicit).

When a timestamp value is called for as part of a descriptor or context definition in this Clause (for example, "the creation timestamp" in a descriptor), and no additional information as to the specific type or precision is supplied, an implementation-defined (IV246) timestamp type is implicit.

4.7.3 Intervals

NOTE 13 — The term "interval" used in this document corresponds to the term "duration" defined in ISO 8601-1:2019.

There are two classes of intervals. One class, called *year-month* intervals, has an express or implied datetime precision that includes no fields other than YEAR and MONTH, though not both are required. The other class, called *day-time* intervals, has an express or implied interval precision that can include any fields other than YEAR or MONTH.

Table 4, "Fields in year-month INTERVAL values", specifies the fields that make up a year-month interval. A year-month interval is made up of a contiguous subset of those fields.

Table 4 — Fields in year-month INTERVAL values

Keyword	**Meaning**
YEAR	Years
MONTH	Months

Table 5, "Fields in day-time INTERVAL values", specifies the fields that make up a day-time interval. A day-time interval is made up of a contiguous subset of those fields.

Table 5 — Fields in day-time INTERVAL values

Keyword	Meaning
DAY	Days
HOUR	Hours
MINUTE	Minutes
SECOND	Seconds and possibly fractions of a second

The actual subset of fields that comprise a value of either type of interval is defined by an <interval qualifier> and this subset is known as the precision of the value.

Within a value of type interval, the first field is constrained only by the <interval leading field precision> of the associated <interval qualifier>. Table 6, "Valid values for fields in INTERVAL values", specifies the constraints on subsequent field values.

Table 6 — Valid values for fields in INTERVAL values

Keyword	Valid values of INTERVAL fields
YEAR	Unconstrained except by <interval leading field precision>
MONTH	Months (within years) (0-11)
DAY	Unconstrained except by <interval leading field precision>
HOUR	Hours (within days) (0-23)
MINUTE	Minutes (within hours) (0-59)
SECOND	Seconds (within minutes) (0-59.999...)

Values in interval fields other than SECOND are integers and have precision 2 when not the first field. SECOND, however, can be defined to have an <interval fractional seconds precision> that indicates the number of decimal digits maintained following the decimal point in the seconds value. When not the first field, SECOND has a precision of 2 places before the decimal point.

Fields comprising an item of type interval are also constrained by the definition of the Gregorian calendar.

Year-month intervals are comparable only with other year-month intervals. If two year-month intervals have different interval precisions, they are, for the purpose of any operations between them, effectively converted to the same precision by appending new <primary datetime field>s to either the most significant end of one interval, the least significant end of one interval, or both. New least significant <primary datetime field>s are assigned a value of 0 (zero). When it is necessary to add new most significant datetime fields, the associated value is effectively converted to the new precision in a manner obeying the natural rules for dates and times associated with the Gregorian calendar.

Day-time intervals are comparable only with other day-time intervals. If two day-time intervals have different interval precisions, they are, for the purpose of any operations between them, effectively con-

verted to the same precision by appending new <primary datetime field>s to either the most significant end of one interval or the least significant end of one interval, or both. New least significant <primary datetime field>s are assigned a value of 0 (zero). When it is necessary to add new most significant datetime fields, the associated value is effectively converted to the new precision in a manner obeying the rules for dates and times associated with the Gregorian calendar.

4.7.4 Operations involving datetimes and intervals

Table 7, "Valid operators involving datetimes and intervals", specifies the declared types of arithmetic expressions involving datetime and interval operands.

Table 7 — Valid operators involving datetimes and intervals

Operand 1	Operator	Operand 2	Result Type
Datetime	–	Datetime	Interval
Datetime	+ or –	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ or –	Interval	Interval
Interval	* or /	Numeric	Interval
Numeric	*	Interval	Interval

Arithmetic operations involving values of type datetime or interval obey the rules associated with dates and times and yield valid datetime or interval results according to the Gregorian calendar.

Operations involving values of type datetime require that the datetime values be comparable. Operations involving values of type interval require that the interval values be comparable.

Operations involving a datetime and an interval preserve the time zone of the datetime operand. If the datetime operand does not include a time zone displacement, then the result has no time zone displacement.

<overlaps predicate> uses the operator OVERLAPS to determine whether or not two chronological periods overlap in time. A chronological period is specified either as a pair of datetimes (starting and ending) or as a starting datetime and an interval. If the length of the period is greater than 0 (zero), then the period consists of all points of time greater than or equal to the lower endpoint, and less than the upper endpoint. If the length of the period is equal to 0 (zero), then the period consists of a single point in time, the lower endpoint. Two periods overlap if they have at least one point in common.

<extract expression> operates on a datetime or interval and returns an exact numeric value representing the value of one component of the datetime or interval.

<interval absolute value function> operates on an interval argument and returns its absolute value in the same most specific type.

4.8 JSON types

4.8.1 Introduction to JSON types

Values of the JSON type are referred to as SQL/JSON values. An SQL/JSON value is either the null value or an SQL/JSON item. SQL/JSON item is described in detail in Subclause 4.48.3, "SQL/JSON data model".

The JSON data type is described by the JSON data type descriptor. The JSON data type descriptor contains:

— The name of the JSON data type (JSON)

4.8.2 Comparison and assignment of JSON values

All SQL/JSON values are assignable to a site of type JSON. Any two SQL/JSON values are comparable.

The exact relationship between two SQL/JSON values is defined by the General Rules of Subclause 8.2, "<comparison predicate>", summarized as follows:

— SQL/JSON values compare equal, if they are equivalent as defined in Subclause 4.48.3, "SQL/JSON data model";

— An SQL/JSON null value compares less than any SQL/JSON scalar;

— An SQL/JSON scalar compares less than any SQL/JSON array;

— An SQL/JSON array compares less than any SQL/JSON object;

— Whether an SQL/JSON scalar that is a number compares less than an SQL/JSON scalar that is not a number is implementation-defined (IA224);

— Whether an SQL/JSON scalar that is a character string compares less than an SQL/JSON scalar that not a character string is implementation-defined (IA224);

— Whether an SQL/JSON scalar that is a Boolean compares less than an SQL/JSON scalar that is not a Boolean is implementation-defined (IA224);

— Whether an SQL/JSON scalar that is a datetime compares less than an SQL/JSON scalar that is not a datetime is implementation-defined (IA224);

— Two SQL/JSON scalars that are numbers compare according to the rules for SQL numbers;

— Two SQL/JSON scalars that are character strings compare according to the rules for SQL character strings using the Unicode codepoint collation;

— Two SQL/JSON scalars that are Booleans compare according to the rules for SQL Booleans;

— Two SQL/JSON scalars that are datetimes compare according to the rules for SQL datetimes with the following extensions:

  • It is implementation-defined (IA224) whether a datetime value of type TIME or TIME WITH TIME ZONE compares less than any datetime value of a type other than TIME or TIME WITH TIME ZONE;

  • Datetime values of type DATE are implicitly converted to TIMESTAMP for comparison only;

— An SQL/JSON array *A* compares less than another SQL/JSON array *B*, if one of the following is true:

  • The cardinality of *A* is less than the cardinality of *B* and the elements at the same array index in both SQL/JSON arrays compare equal for all elements of *A*;

  • There exists an element *E* in *A* that compares less than the corresponding element in *B* and each element that precedes *E* in *A* compares equal to the corresponding element in *B*;

— Whether an SQL/JSON object compares less than another SQL/JSON object that is not equivalent is implementation-defined (IA225);

— If an SQL/JSON value *A* compares less than an SQL/JSON value *B* and *B* compares less than an SQL/JSON value *C*, then *A* compares less than *C*.

4.8.3 Operations involving JSON values

<JSON parse> is an operator that parses a character string or binary string; i.e., converts JSON text to an SQL/JSON item, according to the rules of RFC 8259.

<JSON serialize> is an operator that, given a SQL/JSON value *JV*, returns either a character string or binary string value *SV* such that the result of a <JSON parse> with *SV* as input would be equivalent to *JV*.

<JSON object constructor> is an operator that, given zero or more name-value pairs, returns an SQL/JSON item that is an SQL/JSON object.

<JSON array constructor> is an operator that, given a list of values, returns an SQL/JSON item that is an SQL/JSON array.

<JSON scalar> is an operator that, given a value *V* of character string data type, numeric data type, Boolean data type, or datetime data type, returns an SQL/JSON item that is a SQL/JSON scalar whose value is *V*.

<JSON object aggregate constructor> is an operator that returns an SQL/JSON item that is an SQL/JSON object from a collection of rows.

<JSON array aggregate constructor> is an operator that returns an SQL/JSON item that is an SQL/JSON array from a collection of rows.

<JSON query> is an operator that, given an SQL/JSON item and an SQL/JSON path expression, returns an SQL/JSON item.

<JSON value function> is an operator that, given an SQL/JSON item and an SQL/JSON path expression, returns an SQL value of character string data type, numeric data type, Boolean data type, or datetime data type.

<JSON simplified accessor> provides a subset of the functionality of <JSON query> and <JSON value function> using a more convenient syntax.

<JSON table> is a kind of <derived table>, which may be used to query an SQL/JSON value as a table.

<JSON predicate> is a predicate that determines whether a given character string or JSON value satisfies the rules of RFC 8259 for JSON objects, JSON arrays, and/or JSON scalars.

<JSON exists predicate> is a predicates that evaluates an SQL/JSON path expression and determines if the result is an SQL null value, an empty SQL/JSON sequence, or a non-empty SQL/JSON sequence.

<JSON object constructor>, <JSON array constructor>, <JSON object aggregate constructor>, <JSON array aggregate constructor>, and <JSON query> have the option of returning a character string value or a binary string value instead of a JSON value, in which case the result is implicitly serialized.

4.9 User-defined types

This Subclause is modified by Subclause 4.9, "User-defined types", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.4, "User-defined types", in ISO/IEC 9075-15.

4.9.1 Introduction to user-defined types

This Subclause is modified by Subclause 4.9.1, "Introduction to user-defined types", in ISO/IEC 9075-13.

13 A user-defined type is a schema object, identified by a <user-defined type name>. The definition of a user-defined type specifies a representation for values of that type. A user-defined type is either a *distinct type* or a *structured type*. The representation of a distinct type is a single predefined type or collection type, known as the *source type*. The representation of a structured type consists of a list of attribute definitions. Although the attribute definitions are said to define the representation of the user-defined type, in fact they implicitly define certain functions (observer functions and mutator functions) that are

part of the interface of the user-defined type; physical representations of user-defined type values are implementation-dependent (UV060).

The definition of a user-defined type may include a <method specification list> consisting of one or more <method specification>s, each identifying an *SQL-invoked method*, or simply a *method*, of that user-defined type.

4.9.2 Distinct types

This Subclause is modified by Subclause 4.4.1, "Distinct types", in ISO/IEC 9075-15.

The definition of a distinct type specifies its name and the name of its source type.

The definition of a distinct type may specify FINAL; otherwise, FINAL is implicit. Consequently, the definition of a distinct type shall not specify a <subtype clause>.

The definition of a distinct type *DT* implicitly creates an SQL-invoked function *F1* that converts a value of *DT* to a value of its source type *ST*, an SQL-invoked function *F2* that converts a value of *ST* to a value of *DT*, and a transform *TR* with *F1* as the from-sql function and *F2* as the to-sql function.

15 A distinct type *DT* whose source type is a collection type *CT* is said to have an *element type*, which is the element type of *CT*.

A value of *DT* has a *cardinality*, which is the number of elements in that value. A distinct type whose source type is an array type *AT* is said to have a *maximum cardinality*, which is the maximum cardinality of *AT*.

4.9.3 Structured types

4.9.3.1 Introduction to structured types

The definition of a structured type specifies its name and a list of <attribute definition>s. Each <attribute definition> specifies the name of attribute, the data type of the attribute, and optionally a default value for the attribute.

The definition of a structured type may specify INSTANTIABLE or NOT INSTANTIABLE; otherwise, INSTANTIABLE is implicit. If the definition of a structured type *ST* specifies NOT INSTANTIABLE, then the most specific type of every value in *ST* is necessarily of some proper subtype of *ST*.

The definition of a structured type shall specify either FINAL or NOT FINAL. If the definition of a structured type specifies FINAL, then it shall not specify either a <subtype clause> or NOT INSTANTIABLE.

4.9.3.2 Observer functions and mutator functions

Corresponding to every attribute of every structured type is exactly one implicitly-defined observer function and exactly one implicitly-defined mutator function. These are both SQL-invoked functions. Further, the mutator function is a type-preserving function.

Let *A* be the name of an attribute of structured type *T* and let *AT* be the data type of *A*. The signature of the observer function for this attribute is FUNCTION *A*(*T*) and its result data type is *AT*. The signature of the mutator function for this attribute is FUNCTION *A*(*T* RESULT, *AT*) and its result data type is *T*.

Let *V* be a value in data type *T* and let *AV* be a value in data type *AT*. The invocation *A*(*V*,*AV*) returns *MV* such that "*A*(*MV*) is identical to *AV*" and for every attribute *A′* (*A′* ≠ *A*) of *T*, "*A′*(*MV*) is identical to *A′*(*V*)". The most specific type of *MV* is the most specific type of *V*.

4.9.3.3 Constructors

Associated with each structured type *ST* is one implicitly defined *constructor function*, if and only if *ST* is instantiable.

Let *TN* be the name of a structured type *T*. The signature of the constructor function for *T* is *TN*() and its result data type is *T*. The invocation *TN*() returns a value *V* such that *V* is not the null value and, for every attribute *A* of *T*, *A*(*V*) returns the default value of *A*. The most specific type of *V* is *T*.

For every structured type *ST* that is instantiable, zero or more SQL-invoked constructor methods can be specified. The names of those methods shall be equivalent to the name of the type for which they are specified.

NOTE 14 — SQL-invoked constructor methods are original methods that cannot be overloaded. An SQL-invoked constructor method and a regular SQL-invoked function can exist such that they have equivalent routine names, the types of the first parameter of the method's augmented SQL parameter declaration list and the function's parameter list are the same, and the types of the corresponding remaining parameters (if any) are identical according to the Syntax Rules of Subclause 9.30, "Data type identity".

4.9.3.4 Subtypes and supertypes

As a consequence of the <subtype clause> of <user-defined type definition>, two structured types T_a and T_b that are not compatible can be such that T_a is a subtype of T_b. See Subclause 11.51, "<user-defined type definition>".

A type T_a is a *direct subtype* of a type T_b if T_a is a proper subtype of T_b and there does not exist a type T_c such that T_c is a proper subtype of T_b and a proper supertype of T_a.

A type T_a is a subtype of type T_b if and only if exactly one of the following is true:

— T_a and T_b are compatible;

— T_a is a direct subtype of T_b;

— T_a is a subtype of some type T_c and T_c is a direct subtype of T_b.

By the same token, T_b is a supertype of T_a and is a direct supertype of T_a in the particular case where T_a is a direct subtype of T_b.

If T_a is a subtype of T_b and T_a and T_b are not compatible, then T_a is a proper subtype of T_b and T_b is a proper supertype of T_a. A type cannot be a proper supertype of itself.

A type with no proper supertypes is a *maximal supertype*. A type with no proper subtypes is a *leaf type*.

Let T_a be a maximal supertype and let *T* be a subtype of T_a. The set of all subtypes of T_a (which includes T_a itself) is called a *subtype family* of *T* or (equivalently) of T_a. A subtype family is not permitted to have more than one maximal supertype.

Every value in a type *T* is a value in the direct supertype, if any, of *T*. A value *V* in type *T* has exactly one most specific type *MST* such that *MST* is a subtype of *T* and *V* is not a value in any proper subtype of *MST*. The most specific type of value need not be a leaf type. For example, a type structure might consist of a type PERSON that has STUDENT and EMPLOYEE as its two subtypes, while STUDENT has two direct subtypes UG_STUDENT and PG_STUDENT. The invocation STUDENT() of the constructor function for STUDENT returns a value whose most specific type is STUDENT, which is not a leaf type.

If T_a is a subtype of T_b, then a value in T_a can be used wherever a value in T_b is expected. In particular, a value in T_a can be stored in a column of type T_b, can be substituted as an argument for an input SQL parameter of data type T_b, and can be the value of an invocation of an SQL-invoked function whose result data type is T_b.

A type *T* is said to be the *minimal common supertype* of a set of types *S* if *T* is a supertype of every type in *S* and a subtype of every type that is a supertype of every type in *S*.

NOTE 15 — Because a subtype family has exactly one maximal supertype, if two types have a common subtype, they also have a minimal common supertype. Thus, for every set of types drawn from the same subtype family, there is some member of that family that is the minimal common supertype of all of the types in that set.

If a structured type *ST* is defined to be not instantiable, then the most specific type of every value in *ST* is necessarily of some proper subtype of *ST*.

If the definition of a user-defined type *UDT* specifies or implies FINAL, then *UDT* has no proper subtypes. As a consequence, the most specific type of every value in *UDT* is necessarily *UDT*.

Users shall have the UNDER privilege on a type before they can define any direct subtypes of it. A type can have more than one direct subtype. A user-defined type or a reference type can have at most one direct supertype.

A <user-defined type definition> for type *T* can include references to components of the direct supertype, if any, of *T*. Effectively, components of the representation of the direct supertype are copied to the subtype's representation.

4.9.4 Methods

A method of a user-defined type *T* is either an *original method of T* or an *overriding method of T*. An original method is specified by an <original method specification>, while an overriding method is specified by an <overriding method specification>.

Each <original method specification> specifies:

— The <method name>;

— The <specific method name>;

— The <SQL parameter declaration list>;

— The <returns data type>;

— The <result cast from type> (if any);

— Whether the method is type-preserving;

— The <language clause>;

— If the language is not SQL, then the <parameter style>;

— Whether STATIC or CONSTRUCTOR is specified;

— Whether the method is deterministic;

— Whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL;

— Whether the method shall be evaluated as the null value whenever any argument is the null value, without actually invoking the method.

Each <overriding method specification> specifies:

— The <method name>;

— The <specific method name>;

— The <SQL parameter declaration list>;

— The <returns data type>.

For each <overriding method specification>, there shall be an <original method specification> with the same <method name> and <SQL parameter declaration list> in some proper supertype of the user-defined

type. Every SQL-invoked method in a schema shall correspond to exactly one <original method specification> or <overriding method specification> associated with some user-defined type existing in that schema.

A method *M* is a *method of type T1* if and only if exactly one of the following is true:

— *M* is an original method of *T1*;

— *M* is an overriding method of *T1*;

— There is a proper supertype *T2* of *T1* such that *M* is an original or overriding method of *T2* and such that there is no method *M3* such that *M3* has the same <method name> and <SQL parameter declaration list> as *M* and *M3* is an original method or overriding method of a type *T3* such that *T2* is a proper supertype of *T3* and *T3* is a supertype of *T1*.

4.9.5 User-defined type comparison and assignment

This Subclause is modified by Subclause 4.9.2, "User-defined type comparison and assignment", in ISO/IEC 9075-13.

Let the *comparison type* of a user-defined type T_a be the user-defined type T_b for which all of the following are true:

— The type designator of T_b is in the type precedence list of T_a;

— The user-defined type descriptor of T_b includes an ordering form that is EQUALS or FULL;

— The descriptor of no type T_c whose type designator precedes that of T_b in the type precedence list of T_a includes an ordering form that includes EQUALS or FULL.

If there is no such type T_b, then T_a has no comparison type.

Let *comparison form* of a user-defined type T_a be the ordering form included in the user-defined type descriptor of the comparison type of T_a.

Let *comparison category* of a user-defined type T_a be the ordering category included in the user-defined type descriptor of the comparison type of T_a.

13 Let *comparison function* of a user-defined type T_a be the ordering function included in the user-defined type descriptor of the comparison type of T_a.

13 Two values *V1* and *V2* whose most specific types are user-defined types *T1* and *T2* are comparable if and only if *T1* and *T2* are in the same subtype family and each have some comparison type *CT1* and *CT2*, respectively. *CT1* and *CT2* constrain the comparison forms and comparison categories of *T1* and *T2* to be the same and to be the same as those of all their supertypes.

— If the comparison category is either STATE or RELATIVE, then *T1* and *T2* are constrained to have the same comparison function;

— if the comparison category is MAP, they are not constrained to have the same comparison function.

NOTE 16 — Explicit cast functions or attribute comparisons can be used to make both values of the same subtype family or to perform the comparisons on attributes of the user-defined types.

NOTE 17 — "Subtype" and "subtype family" are defined in Subclause 4.9.3.4, "Subtypes and supertypes".

If there is no appropriate user-defined cast function, then an expression *E* whose declared type is some user-defined type *UDT1* is assignable to a site *S* whose declared type is some user-defined type *UDT2* if and only if *UDT1* is a subtype of *UDT2*. The effect of the assignment of *E* to *S* is that the value of *S* is *V*,

obtained by the evaluation of *E*. The most specific type of *V* is some subtype of *UDT1*, possibly *UDT1* itself, while the declared type of *S* remains *UDT2*.

An expression whose declared type is some distinct type whose source type is *SDT* is assignable to any site whose declared type is *SDT* because of the implicit cast functions created by the General Rules of Subclause 11.51, "<user-defined type definition>". Similarly, an expression whose declared type is some predefined data type *SDT* is assignable to any site whose declared type is some distinct type whose source type is *SDT*.

4.9.6 Transforms for user-defined types

Transforms are SQL-invoked functions that are automatically invoked when values of user-defined types are transferred from SQL-environment to host languages or vice-versa.

A transform is associated with a user-defined type. A transform identifies a list of *transform groups* of up to two SQL-invoked functions, called the *transform functions*, each identified by a group name. The group name of a transform group is an <identifier> such that no two transform groups for a transform have the same group name. The two transform functions are:

— **from-sql function** — This SQL-invoked function maps the user-defined type value into a value of an SQL pre-defined type, and gets invoked whenever a user-defined type value is passed to a host language program or an external routine;

— **to-sql function** — This SQL-invoked function maps a value of an SQL predefined type to a value of a user-defined type and gets invoked whenever a user-defined type value is supplied by a host language program or an external routine.

A transform is defined by a <transform definition>. A transform is described by a *transform descriptor*. A transform descriptor includes a possibly empty list of *transform group descriptors*, where each transform group descriptor includes:

— The group name of the transform group;

— The specific name of the from-sql function, if any, associated with the transform group;

— The specific name of the to-sql function, if any, associated with the transform group.

4.9.7 User-defined type descriptor

This Subclause is modified by Subclause 4.9.3, "User-defined type descriptor", in ISO/IEC 9075-13.

13 A user-defined type is described by a user-defined type descriptor. A user-defined type descriptor contains:

— The name of the user-defined type (<user-defined type name>). This is the type designator of that type, used in type precedence lists (see Subclause 9.7, "Type precedence list determination");

— An indication of whether the user-defined type is a structured type or a distinct type;

— The ordering form for the user-defined type, one of:

- EQUALS;
- FULL;
- NONE;

— 13 The ordering category for the user-defined type, one of:

- RELATIVE;
- MAP;

- STATE;

— A <specific routine designator> identifying the ordering function, depending on the ordering category;

— If the user-defined type is a direct subtype of another user-defined type, then the name of that user-defined type;

— If the user-defined type is a distinct type, then the descriptor of the source type; otherwise, the attribute descriptor of every originally-defined attribute and every inherited attribute of the user-defined type;

— An indication of whether the user-defined type is instantiable or not instantiable;

— An indication of whether the user-defined type is final or not final;

— [13] The transform descriptor of the user-defined type;

— If the user-defined type is a structured type, then:

- Whether the referencing type of the structured type has a user-defined representation, a derived representation, or a system-defined representation;
- If user-defined representation is specified, then the type descriptor of the representation type of the referencing type of the structured type; otherwise, if derived representation is specified, then the list of derivational attributes of the derived representation;

NOTE 18 — "user-defined representation", "derived representation", and "system-defined representation" of a reference type are defined in Subclause 4.11, "Reference types".

— If the <method specification list> is specified, then for each <method specification> contained in <method specification list>, a *method specification descriptor* that includes:

- The <method name>;
- The <specific method name>;
- An indication of whether the <method specification> is an <original method specification> or an <overriding method specification>;
- If the <method specification> is an <original method specification>, then an indication of whether STATIC or CONSTRUCTOR is specified;
- The <SQL parameter declaration list> augmented to include the implicit first parameter with parameter name SELF;
- For every <SQL parameter declaration> in the <SQL parameter declaration list>, a <locator indication>, if any;
- The <returns data type>;
- The <result cast from type>, if any;
- The <locator indication> contained in the <returns clause>, if any;
- The <language name>;
- If the <language name> is not SQL, then the <parameter style>;
- An indication whether the method is deterministic;
- An indication whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL;
- An indication whether the method shall not be invoked if any argument is the null value;

- The creation timestamp;
- The last-altered timestamp.

NOTE 19 — The characteristics of an <overriding method specification> other than the <method name>, <SQL parameter declaration list>, and <returns data type> are the same as the characteristics for the corresponding <original method specification>.

4.10 Row types

A row type is a sequence of (<field name> <data type>) pairs, called *fields*. It is described by a row type descriptor. A row type descriptor consists of the field descriptor of every field of the row type.

The most specific type of a row of a table is a row type. In this case, each column of the table corresponds to the field of the row type that has the same ordinal position as the column.

Row type *RT2* is a subtype of data type *RT1* if and only if *RT1* and *RT2* are row types of the same degree and, in every *n*-th pair of corresponding field definitions, $FD1_n$ in *RT1* and $FD2_n$ in *RT2*, the <field name>s are equivalent and the <data type> of $FD2_n$ is a subtype of the <data type> of $FD1_n$.

A value of row type *RT1* is assignable to a site of row type *RT2* if and only if the degree of *RT1* is the same as the degree of *RT2* and every field in *RT1* is assignable to the field in the same ordinal position in *RT2*.

A value of row type *RT1* is comparable with a value of row type *RT2* if and only if the degree of *RT1* is the same as the degree of *RT2* and every field in *RT1* is comparable with the field in the same ordinal position in *RT2*.

4.11 Reference types

4.11.1 Introduction to reference types

A *REF value* is a value that references a row in a *referenceable table* (see Subclause 4.17.9, "Referenceable tables, subtables, and supertables"). A referenceable table is necessarily also a *typed table* (that is, it has an associated structured type from which its row type is derived).

Given a structured type *T*, the REF values that can reference rows in typed tables defined on *T* collectively form a certain data type *RT* known as a *reference type*. *RT* is the *referencing type* of *T* and *T* is the *referenced type* of *RT*.

Let *TN* be name of *T*. The type designator of *RT* is REF(*TN*).

Values of two reference types are comparable if the referenced types of their declared types have some common supertype.

An expression *E* whose declared type is some reference type *RT1* is assignable to a site *S* whose declared type is some reference type *RT2* if and only if the referenced type of *RT1* is a subtype of the referenced type of *RT2*. The effect of the assignment of *E* to *S* is that the value of *S* is *V*, obtained by the evaluation of *E*. The most specific type of *V* is some subtype of *RT1*, possibly *RT1* itself, while the declared type of *S* remains *RT2*.

A site *RS* that is occupied by a REF value might have a *scope*, which determines the effect of an invocation of <reference resolution> *RR* on the value at *RS*. A scope is specified as a table name *STN* and consists at any time of every row in the table *ST* identified by *STN*. *ST* is the *scoped table* of *RR*. The scope of *RS* is specified in the declared type of *RS*. If no scope is specified in the declared type of *RS*, then <reference resolution> is not available.

A reference type is described by a reference type descriptor. The reference type descriptor for *RT* includes:

— The type designator of *RT*;

— The name of the referenceable table, if any, that is the scope of *RT*.

In a host variable, a REF value is materialized as an *N*-octet value, where *N* is implementation-defined (IV071).

Reference type *RT2* is a *subtype* of data type *RT1* (equivalently, *RT1* is a *supertype* of *RT2*) if and only if *RT1* is a reference type and the referenced type of *RT2* is a subtype of the referenced type of *RT1*.

Every value in a reference type *RT* is a value in every supertype of *RT*. A value *V* in type *RT* has exactly one most specific type *MST* such that *MST* is a subtype of *RT* and *V* is not a value in any proper subtype of *MST*.

A reference type has a *user-defined representation* if its referenced type is defined by a <user-defined type definition> that specifies <user-defined representation>. A reference type has a *derived representation* if its referenced type is defined by a <user-defined type definition> that specifies <derived representation>. A reference type has a *system-defined representation* if it does not have a user-defined representation or a derived representation.

4.11.2 Operations involving references

An operation is provided that takes a REF value and returns the value that is held in a column of the site identified by the REF value (see Subclause 6.22, "<dereference operation>"). If the REF value identifies no site, perhaps because a site it once identified has been destroyed, then the null value is returned.

An operation is provided that takes a REF value and returns a value of the referenced type; that value is constructed from the values of the columns of the site identified by that REF value (see Subclause 6.24, "<reference resolution>"). An operation is also provided that takes a REF value and returns a value acquired from invoking an SQL-invoked method on a value of the referenced type; that value is constructed from the values of the columns of the site identified by that REF value (see Subclause 6.23, "<method reference>").

4.12 Collection types

This Subclause is modified by Subclause 4.5, "Collection types", in ISO/IEC 9075-15.

4.12.1 Introduction to collection types

This Subclause is modified by Subclause 4.5.1, "Introduction to collection types", in ISO/IEC 9075-15.

15 A *collection* is a composite value comprising zero or more *elements*, each a value of some data type *DT*. If the elements of some collection *C* are values of *DT*, then *C* is said to be a collection of *DT*. The number of elements in *C* is the *cardinality* of *C*. The term "element" is not further defined in this document. The term "collection" is generic, encompassing various kinds of collection in connection with each of which, individually, this document defines primitive type constructors and operators. This document supports the following collection types:

— 15 Arrays;

— Multisets.

15 A specific <collection type> *CT* is a <data type> specified by pairing a keyword *KC* (ARRAY or MULTISET) with a specific data type *EDT*. Every element of every possible value of *CT* is a value of *EDT* and is permitted to be, more specifically, of some subtype of *EDT*. *EDT* is termed the *element type* of *CT*. *KC* specifies the kind of collection that every value of *CT* is, and thus determines the operators that are available for operating on or returning values of *CT*.

A maximum cardinality may optionally be specified for arrays.

Let *EDTN* be the type designator of *EDT*. The type designator of *CT* is *EDTN KC*.

15 A *collection type descriptor* describes a collection type. The collection type descriptor for *CT* includes:

— The type designator of *CT*;

— The descriptor of the element type of *CT*;

— An indication of the kind of the collection of *CT*:

  • 15 ARRAY (array type);

  • 15 MULTISET (multiset type);

— If *CT* is an array type, the maximum number of elements of *CT*.

Collection type *CT2* is a subtype of data type *CT1* (equivalently, *CT1* is a supertype of *CT2*) if and only if *CT1* is the same kind of collection as *CT2* and the element type of *CT2* is a subtype of the element type of *CT1*.

15 An external function (see Subclause 4.35, "SQL-invoked routines") can return values of a collection type; such a function is called a *collection-returning external function*. This document supports the following types of collection-returning external functions:

— Array-returning external functions;

— Multiset-returning external functions.

4.12.2 Arrays

An *array* is a collection *A* in which each element is associated with exactly one ordinal position in *A*. If *n* is the cardinality of *A*, then the ordinal position *p* of an element is an integer in the range 1 (one) ≤ *p* ≤ *n*. If *EDT* is the element type of *A*, then *A* can thus be considered as a function of the integers in the range 1 (one) to *n* into *EDT*.

An array site *AS* has a maximum cardinality *m*. The cardinality *n* of an array occupying *AS* is constrained not to exceed *m*.

An *array type* is a <collection type>. If *AT* is some array type with element type *EDT*, then every value of *AT* is an array of *EDT*.

Let *A1* and *A2* be arrays of *EDT*. *A1* and *A2* are identical if and only if *A1* and *A2* have the same cardinality *n* and if, for all *i* in the range 1 (one) ≤ *i* ≤ *n*, the element at ordinal position *i* in *A1* is identical to the element at ordinal position *i* in *A2*.

4.12.3 Multisets

A multiset is an unordered collection. Since a multiset is unordered, there is no ordinal position to reference individual elements of a multiset.

A multiset type is a <collection type>. If *MT* is some multiset type with element type *EDT*, then every value of *MT* is a multiset of *EDT*.

Let *M1* and *M2* be multisets of *EDT*. *M1* and *M2* are identical if and only if *M1* and *M2* have the same cardinality *n*, and for each element *x* in *M1*, the number of elements of *M1* that are identical to *x*, including *x* itself, equals the number of elements of *M2* that are identical to *x*.

Let *n1* be the cardinality of *M1* and let *n2* be the cardinality of *M2*. *M1* is a submultiset of *M2* if, for each element *x* of *M1*, the number of elements of *M1* that are not distinct from *x*, including *x* itself, is less than or equal to the number of elements of *M2* that are not distinct from *x*.

4.12.4 Collection comparison and assignment

This Subclause is modified by Subclause 4.5.3, "Collection comparison and assignment", in ISO/IEC 9075-15.

Two collections of ARRAY or MULTISET kind are comparable if and only if they are of the same kind and their element types are comparable.

A value of collection type *CT1* is assignable to a site of collection type *CT2* if and only if *CT1* is the same kind of collection (ARRAY or MULTISET) as *CT2* and the element type of *CT1* is assignable to the element type of *CT2*.

The array types have a defined *element order*. Comparisons are defined in terms of the element order of the arrays. The element order defines the pairs of corresponding elements from the arrays being compared. The element order of an array is implicitly defined by the ordinal position of its elements.

15 In the case of comparison of two arrays *C* and *D*, the elements are compared pairwise in element order. *C* = *D* is *True* if and only if *C* and *D* have the same cardinality and every pair of elements are equal.

Two multisets *C* and *D* of comparable element types are equal if they have the same cardinality *N* and there exist an enumeration CE_j, 1 (one) ≤ *j* ≤ *N* of the elements of *C* and an enumeration DE_j, 1 (one) ≤ *j* ≤ *N* of the elements of *D* such that for all *j*, $CE_j = DE_j$.

4.12.5 Operations involving arrays

4.12.5.1 Operators that operate on array values and return array elements

<array element reference> is an operation that returns the array element in the specified position within the array.

4.12.5.2 Operators that operate on array values and return array values

<array concatenation> is an operation that returns the array value made by joining its array value operands in the order given.

<trim array function> is an operation that returns an array value derived from the first input argument (an array value) by removing the last *n* elements, where *n* is the value of the second input argument.

4.12.5.3 Operators that operate on array values and return numbers

<cardinality expression> is an operation that returns the cardinality of the array.

<max cardinality expression> returns the maximum cardinality of the declared type of a given array as an exact numeric value.

4.12.6 Operations involving multisets

4.12.6.1 Operators that operate on multisets and return multiset elements

<multiset element reference> is an operation that returns the value of the element of a multiset, if the multiset has only one element.

4.12.6.2 Operators that operate on multisets and return multisets

<multiset set function> is an operation that returns the multiset obtained by removing duplicates from a multiset.

MULTISET UNION is an operator that computes the union of two multisets. There are two variants, specified using ALL or DISTINCT, to either retain duplicates or remove duplicates.

MULTISET INTERSECT is an operator that computes the intersection of two multisets. There are two variants, ALL and DISTINCT. The variant specified by ALL places in the result as many instances of each value as the minimum number of instances of that value in either operand. The variant specified by DISTINCT removes duplicates from the result.

MULTISET EXCEPT is an operator that computes the multiset difference of two multisets. There are two variants, ALL and DISTINCT. The variant specified by ALL places in the result a number of instances of a value, equal to the number of instances of the value in the first operand minus the number of instances of the value in the second operand. The variant specified by DISTINCT removes duplicates from the result.

4.12.6.3 Operators that operate on multiset values and return numbers

<cardinality expression> is an operation that returns the cardinality of the multiset.

4.13 Data conversions

This Subclause is modified by Subclause 4.4, "Data conversions", in ISO/IEC 9075-14.

Implicit type conversion can occur in expressions, fetch operations, single row select operations, inserts, deletes, and updates. Explicit type conversions can be specified by the use of the CAST operator.

Explicit data conversions can be specified by a *CAST operator*. A CAST operator defines how values of a source data type are converted into a value of a target data type according to the Syntax Rules and General Rules of Subclause 6.13, "<cast specification>". Data conversions between predefined data types and constructed types are defined by the rules of this document. Data conversions between a user-defined type and another data type are defined by a user-defined cast.

14 A user-defined cast identifies an SQL-invoked function, called the *cast function*, that has one SQL parameter whose declared type is the same as the source data type and a result data type that is the target data type. A cast function may optionally be specified to be implicitly invoked whenever values are assigned to targets of its result data type. Such a cast function is called an *implicitly invocable* cast function.

A user-defined cast is defined by a <user-defined cast definition>. A user-defined cast has a user-defined cast descriptor that includes:

— The name of the source data type;

— The name of the target data type;

— The specific name of the SQL-invoked function that is the cast function;

— An indication as to whether the cast function is implicitly invocable.

When a value *V* of declared type *TV* is assigned to a target *T* of declared type *TT*, a user-defined cast function *UDCF* is said to be an *appropriate user-defined cast function* if and only if all of the following are true:

— The descriptor of *UDCF* indicates that *UDCF* is implicitly invocable;

— The type designator of the declared type *DTP* of the only SQL parameter *P* of *UDCF* is in the type precedence list of *TV*;

— The result data type of *UDCF* is *TT*;

— No other user-defined cast function *UDCQ* with an SQL parameter *Q* with declared type *TQ* that precedes *DTP* in the type precedence list of *TV* is an appropriate user-defined cast function to assign *V* to *T*.

An <SQL procedure statement> *S* is said to be *dependent on* an appropriate user-defined cast function *UDCF* if and only if all of the following are true:

— *S* is a <select statement: single row>, <insert statement>, <update statement: positioned>, <update statement: searched>, or <merge statement>;

— *UDCF* is invoked during a store or retrieval assignment operation that is executed during the execution of *S* and *UDCF* is not executed during the invocation of an SQL-invoked function that is invoked during the execution of *S*.

4.14 Domains

A domain is a set of permissible values. A domain is defined in a schema and is identified by a <domain name>. The purpose of a domain is to constrain the set of valid values that can be stored in a column of a base table by various operations.

A domain definition specifies a data type. It may also specify a <domain constraint> that further restricts the valid values of the domain and a <default clause> that specifies the value to be used in the absence of an explicitly specified value or column default.

A domain is described by a domain descriptor. A domain descriptor includes:

— The name of the domain;

— The data type descriptor of the data type of the domain;

— The value of <default option>, if any, of the domain;

— The domain constraint descriptors of the domain constraints, if any, of the domain.

4.15 Columns, fields, and attributes

This Subclause is modified by Subclause 4.3, "Columns, fields, and attributes", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.2, "Columns, fields, and attributes", in ISO/IEC 9075-16.

The terms *column*, *field*, and *attribute* refer to structural components of tables, row types, and structured types, respectively, in analogous fashion. As the structure of a table consists of one or more columns, so does the structure of a row type consist of one or more fields and that of a structured type one or more attributes. Every structural element, whether a column, a field, or an attribute, is primarily a name paired with a declared type. The elements of a structure are ordered. Elements in different positions in the same structure can have the same declared type. Sometimes the Syntax Rules forbid two elements at different ordinal positions from having equivalent names (particularly when created through an <SQL schema statement>). When two elements with equivalent names are permitted, then the element name is ambiguous and attempts to reference either element by name will raise a syntax error. Although the elements of a structure are distinguished from each other by name, in some circumstances the compatibility of two structures (for the purpose at hand) is determined solely by considering the declared types of each pair of elements at the same ordinal position.

A table (see Subclause 4.17, "Tables") is defined on one or more columns and consists of zero or more rows. A column has a name and a declared type. Each row in a table has exactly one value for each column. Each value in a row is a value in the declared type of the corresponding column. A column is either *updatable* or *not updatable*.

NOTE 20 — The declared type includes the null value and values in proper subtypes of the declared type.

Every column has a *nullability characteristic* that indicates whether the value from that column can be the null value. A nullability characteristic is either *known not nullable* or *possibly nullable*.

Let *C* be a column of a base table *T*. *C* is *known not nullable* if and only if at least one of the following is true:

— There exists at least one constraint *NNC* that is enforced and not deferrable and that simply contains a <search condition> that is a <boolean value expression> that is a readily-known-not-null condition for *C*;

— *C* is based on a domain that has a domain constraint that is not deferrable and that simply contains a <search condition> that is a <boolean value expression> that is a readily-known-not-null condition for VALUE;

— *C* is a unique column of a non-deferrable unique constraint that is a PRIMARY KEY;

— *C* is the self-referencing column of *T*;

— The SQL-implementation is able to deduce that the <search condition> "*C* IS NULL" can never be *True* when applied to a row in *T* through some additional implementation-defined (IA199) rule or rules;

NOTE 21 — The notions "known not nullable" and "known-not-null condition" are examples of such additional rules that an SQL-implementation can adopt. If an SQL-implementation does adopt those rules (and possibly others), then the SQL-implementation can claim support for Feature T101, "Enhanced nullability determination".

16 The nullability characteristic of a column of a derived table is defined by the Syntax Rules of Subclause 7.6, "<table reference>" (for <PTF derived table>), Subclause 7.10, "<joined table>", Subclause 7.16, "<query specification>", and Subclause 7.17, "<query expression>".

A column *C* is described by a column descriptor. A column descriptor includes:

— The name of the column;

— Whether the name of the column is an implementation-dependent name.

— If the column is based on a domain, then the name of that domain; otherwise, the data type descriptor of the declared type of *C*;

— The value of <default option>, if any, of *C*;

— The nullability characteristic of *C*;

— The ordinal position of *C* within the table that contains it;

— An indication of whether *C* is updatable or not;

— An indication of whether *C* is a self-referencing column of a base table or not;

— An indication of whether *C* is an identity column or not;

— If *C* is an identity column, then an indication of whether values are always generated or generated by default;

— If *C* is an identity column, then the descriptor of the internal sequence generator for *C*, which includes the *start value* of *C*;

NOTE 22 — Identity columns and the meaning of "start value" are described in Subclause 4.17.12, "Identity columns".

— An indication of whether *C* is a generated column or not;

— If *C* is a generated column, then the generation expression of *C*;

NOTE 23 — Generated columns and the meaning of "generation expression" are described in Subclause 4.17.13, "Base columns and generated columns".

— An indication of whether *C* is a system-time period start column or not;

— An indication of whether *C* is a system-time period end column or not;

— If *C* is a system-time period start column or a system-time period end column, then an indication that values are always generated;

— If *C* is a column of a base table, then an indication of whether it is defined as NOT NULL and, if so, the constraint name of the associated table constraint definition;

NOTE 24 — This indication and the associated constraint name exist for definitional purposes only and are not exposed through the COLUMNS view in the Information Schema.

An attribute *A* is described by an attribute descriptor. An attribute descriptor includes:

— The name of the attribute;

— The data type descriptor of the declared type of *A*;

— The ordinal position of *A* within the structured type that contains it;

— The value of the implicit or explicit <attribute default> of *A*;

— The name of the structured type defined by the <user-defined type definition> that defines *A*.

09 A field *F* is described by a field descriptor. A field descriptor includes:

— The name of the field;

— The data type descriptor of the declared type of *F*;

— The ordinal position of *F* within the row type that simply contains it.

4.16 Periods

4.16.1 Introduction to periods

A period definition for a given table associates a period name with a pair of column names defined for that table. For a table *T* with a period definition *PD*, let *PN* be the period name, *START* be the first column name specified in *PD*, and *END* be the second column name specified in *PD*. *START* is called the *PN period start column* of *T*. *END* is called the *PN period end column* of *T*. The columns identified by *START* and *END* shall both be of a datetime data type and known not nullable. Further, the declared types of *START* and *END* shall be identical.

Every row in *T* is considered to be associated with the *PN period value*, with the value in the *START* column as the *PN period start value* and the value in the *END* column as the *PN period end value*. The *PN* period value of a row *R* is a set of all values of *DT*, where *DT* is the declared type of *START*, that are greater than or equal to the value of *START* and less than the value of *END*. For a table *T* with period *PN*, there is an implicit constraint that ensures that the *PN* period end value of a given row *R* is greater than the *PN* period start value of *R*.

A period is described by a *period descriptor*. A period descriptor includes:

— The name of the period;

— The name of the period start column;

— The name of the period end column;

— If the period name is an <application time period name>, then the name of the implicit period constraint.

If the table descriptor of a table *T* contains a period descriptor that includes a period name *PN*, then that period descriptor is also referred to as a *PN* period descriptor of *T*. A period whose period name is SYSTEM_TIME is also known as a *system-time period* and the corresponding period descriptor is also known as a *system-time period descriptor*. A period whose period name is not SYSTEM_TIME is also known as an *application-time period* and the corresponding period descriptor is also known as an *application-time period descriptor*.

Let *R1* and *R2* be any two rows in a table *T* with period *PN*. Let *S1* be the *PN* period start value of *R1*, *E1* be the *PN* period end value of *R1*, *S2* be the *PN* period start value of *R2*, and *E2* be the *PN* period end value

of *R2*. The *PN* period value of *R1* is said to *overlap* with the *PN* period value of *R2* if and only if (*S1* < *E2* AND *E1* > *S2*).

4.16.2 Operations involving periods

<period overlaps predicate> tests whether two periods overlap; that is, have at least one value in common.

<period equals predicate> tests whether two periods have the same period start and period end values.

<period contains predicate> tests whether a period contains a datetime value or another period; that is, whether the first period contains the datetime value or all the values that are contained in the second period, respectively.

<period precedes predicate> tests whether a period precedes another period; that is, all values contained in the first period are less than the start value of the second period.

<period succeeds predicate> tests whether a period succeeds another period; that is, all values contained in the first period are greater than or equal to the end value of the second period.

<period immediately precedes predicate> tests whether a period immediately precedes another period; that is, the end value of the first period equals the start value of the second period.

<period immediately succeeds predicate> tests whether a period immediately follows another period; that is, the start value of the first period equals the end value of the second period.

4.17 Tables

This Subclause is modified by Subclause 4.3, "Tables", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.4, "Tables", in ISO/IEC 9075-9.

4.17.1 Introduction to tables

This Subclause is modified by Subclause 4.4.1, "Introduction to tables", in ISO/IEC 9075-9.

A table is a collection of zero or more rows where each row is a sequence of one or more column values. The most specific type of a row is a row type. Every row of a given table has the same row type, called the *row type* of that table. The value of the *i*-th field of every row in a table is the value of the *i*-th column of that row in the table. The row is the smallest unit of data that can be inserted into a table and deleted from a table.

The *degree* of a table, and the degree of each of its rows, is the number of columns of that table. The number of rows in a table is its *cardinality*. A table whose cardinality is 0 (zero) is said to be *empty*.

09 A table is either a base table, a derived table, or a transient table.

4.17.2 Base tables

This Subclause is modified by Subclause 4.3.1, "Base tables", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.4.2, "Base tables", in ISO/IEC 9075-9.

4.17.2.1 Introduction to base tables

A base table is either a persistent base table or a temporary table.

A persistent base table is either a regular persistent base table or a system-versioned table.

A regular base table is either a regular persistent base table or a temporary table.

4.17.2.2 Regular persistent base tables

A regular persistent base table is a named table defined by a <table definition> that neither specifies TEMPORARY nor specifies WITH SYSTEM VERSIONING.

4.17.2.3 System-versioned tables

A system-versioned table is a named table defined by a <table definition> that specifies PERIOD SYSTEM_TIME and specifies WITH SYSTEM VERSIONING.

Columns of a system-versioned table include a *system-time period start column* and a *system-time period end column*. At any point in time, all rows that have their system-time period end column set to the highest value supported by the data type of that column are known as *current system rows*; all other rows are known as *historical system rows*.

4.17.2.4 Temporary tables

This Subclause is modified by Subclause 4.3.1.1, "Temporary tables", in ISO/IEC 9075-4.

A temporary base table is either a global temporary table, a created local temporary table, or a declared local temporary table

A global temporary table is a named table defined by a <table definition> that specifies GLOBAL TEMPORARY. Global temporary tables are effectively materialized only when referenced in an SQL-session. Every SQL-session that references a global temporary table causes a distinct instance of that global temporary table (i.e., a multiset of rows that is visible only to that SQL-session) to be materialized. That is, the multiset of rows that is referenced by the <table name> of a global temporary table cannot be shared between SQL-sessions.

For purposes of temporary tables, if an SQL-session is executing direct SQL, then the <direct SQL statement>s executed during the SQL-session effectively comprise a separate SQL-client module.

A created local temporary table is a named table defined by a <table definition> that specifies LOCAL TEMPORARY. Created local temporary tables are effectively materialized only when referenced in an SQL-session. Every SQL-client module in every SQL-session that references a created local temporary table causes a distinct instance of that created local temporary table (i.e., a multiset of rows that is visible only to that SQL-client module during that SQL-session) to be materialized. That is, the multiset of rows that is referenced by the <table name> of a created local temporary table cannot be shared between SQL-sessions, nor between SQL-client modules that execute during an SQL-session.

The definition of a global temporary table or a created local temporary table appears in a schema. In SQL language, the name and the scope of the name of a global temporary table or a created local temporary table are indistinguishable from those of a persistent base table.

A declared local temporary table is a named table defined by a <temporary table declaration>.

An SQL-client module declared local temporary table is a declared local temporary table defined in an <SQL-client module definition> or in a <direct SQL statement>. (Note that the <direct SQL statement>s executed by an SQL-session that is executing direct SQL effectively comprise a distinct SQL-client module.) An SQL-client module declared local temporary table is effectively materialized the first time any <externally-invoked procedure> in the <SQL-client module definition> that contains the <temporary table declaration> is executed, or when the <temporary table declaration> is executed as a <direct SQL statement>, and it persists for that SQL-session. Every SQL-client module in every SQL-session that references an SQL-client module declared local temporary table causes a distinct instance of that declared local temporary table (i.e., a multiset of rows that is visible only to that SQL-client module during that SQL-session) to be materialized. That is, the multiset of rows that is referenced by the <table name> of an SQL-client module declared local temporary table cannot be shared between SQL-sessions, nor between SQL-client modules that execute during an SQL-session.

04 All references to a declared local temporary table are prefixed by "MODULE.".

The materialization of a temporary table does not persist beyond the end of the SQL-session in which the table was materialized. Temporary tables are effectively empty at the start of an SQL-session. Temporary tables that are created within an SQL-session *S1* are also effectively empty after creation. If an <as subquery clause> that specifies WITH DATA is specified, the materialization of the temporary table within *S1* is initialized as specified by the General Rules of Subclause 11.3, "<table definition>".

4.17.3 Derived tables

A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of an expression, such as a <joined table>, <data change delta table>, <query expression>, or <table expression>. A <query expression> can contain an optional <order by clause>. The ordering of the rows of the table specified by the <query expression> is guaranteed only for the <query expression> that immediately contains the <order by clause>.

NOTE 25 — Derived tables, as defined in this Subclause, are not to be confused with the BNF non-terminal <derived table>, which is a specific syntax construct.

A <query expression> can also optionally contain a <result offset clause>, which may limit the cardinality of the derived table by removing a specified number of rows from the beginning of the derived table. If a <query expression> contains both an <order by clause> and a <result offset clause>, then the rows in the derived table are first sorted according to the <order by clause> and then limited by dropping the number of rows specified in the <result offset clause> from the beginning of the result produced by the <query expression>. If the cardinality of the result of an evaluation of a <query expression> is less than the offset value specified by a <result offset clause>, then the derived table is empty.

A <query expression> can also optionally contain a <fetch first clause>, which may limit the cardinality of the derived table. If a <query expression> contains both an <order by clause> and a <fetch first clause>, then the rows in the derived table are first sorted according to the <order by clause> and then limited to the number of rows specified in the <fetch first clause>. The <fetch first clause> may specify this limit either as an exact number of rows, or as a percentage. In addition, if WITH TIES is specified, then any peers of retained rows are also retained in the result. A <query expression> may contain both a <result offset clause> and a <fetch first clause>, in which case the <result offset clause> is applied first, followed by the <fetch first clause>. If the cardinality of the result of an evaluation of a <query expression> is less than the limit specified by a <fetch first clause>, then the <fetch first clause> has no effect on the derived table.

A *viewed table* is a named derived table defined by a <view definition>. A viewed table is sometimes called a *view*. Base tables and views are identified by <table name>s. The same <table name>, in its fully qualified form, cannot be used for both a base table and a view.

Optionally, an effectively updatable view *V* may specify CHECK OPTION, in either of two varieties: CASCADED CHECK OPTION or LOCAL CHECK OPTION.

LOCAL CHECK OPTION can be specified only if *V* is simply updatable. CHECK OPTION is not permitted if a target generally underlying table of *V* is trigger insertable, trigger updatable, or trigger deletable. If CHECK OPTION is specified, then *V* itself shall not subsequently be made trigger insertable, trigger updatable, or trigger deletable.

Let *TLUT* be a target leaf underlying table of *V*. The primary and secondary effects of the data change operation on *TLUT* are constrained as follows. For each row *R* that is in the new delta table of insert operation, the new delta table of update operation, or the new delta table of merge operation of *TLUT*, CHECK OPTION constrains *R* as follows:

— If CASCADED CHECK OPTION is specified, then *R* must be a row that can be seen by performing SELECT * FROM *VN* (where *VN* is the name of *V*);

— If LOCAL CHECK OPTION is specified, then, informally, *R* must satisfy the <search condition>(s) of *V* (if any), but need not satisfy the <search condition>s of any views that are generally underlying tables of *V* that do not specify CHECK OPTION;

NOTE 26 — The latter condition is difficult to formulate in a rigorous definition, because *V* or a generally underlying table of *V* that is itself a view can have a <derived table> in its <from clause> that renames columns. The rigorous specification is found in the General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION".

It is implementation-defined (IA056) whether new rows of *TLUT* that are the result of tertiary effects of a data change operation to a base table are checked for a view that specifies CHECK OPTION.

NOTE 27 — The primary, secondary, and tertiary effects of a data change operation on a base table are defined in Subclause 4.17.10, "Operations involving tables".

The original <query expression> *OQE* of a view *V*, and any <query specification>, <table value constructor>, <explicit table>, or <query expression> contained in *OQE* is called a *view component* of *V*.

4.17.4 Transient tables

A transient table is a named table that may come into existence implicitly during the evaluation of a <query expression> or the execution of a trigger. A transient table is identified by a <query name> if it arises during the evaluation of a <query expression>, or by a <transition table name> if it arises during the execution of a trigger. Such tables exist only for the duration of the executing SQL-statement containing the <query expression> or for the duration of the executing trigger.

4.17.5 Unique identification of tables

This Subclause is modified by Subclause 4.3.2, "Unique identification of tables", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.4.3, "Unique identification of tables", in ISO/IEC 9075-9.

04 09 The <table name> of a persistent base table, global temporary table, created local temporary table, or SQL-client module declared local temporary table is used to identify a particular multiset of rows, as follows:

— The <table name> of a persistent base table uniquely identifies a multiset of rows;

— The <table name> of a global temporary table, together with an SQL-session identifier, uniquely identifies a multiset of rows;

— The <table name> of a created local temporary table, together with an SQL-session identifier and an SQL-client module name, uniquely identifies a multiset of rows;

— 04 09 The <table name> of an SQL-client module declared local temporary table, together with an SQL-session identifier and an SQL-client module name, uniquely identifies a multiset of rows.

4.17.6 Table updatability

A table may be updatable. This document defines two grades of table updatability, using the adjectives "simply updatable" and "generally updatable". The set of simply updatable tables is a proper subset of the set of generally updatable tables. A table is *effectively updatable* if the table is simply updatable, or if the table is generally updatable and the SQL-implementation supports Feature T111, "Updatable joins, unions, and columns".

A generally updatable table has at least one *updatable column*. Every column of a simply updatable table is updatable. All base tables are simply updatable and every column of a base table is updatable. Transition tables are not generally updatable (and therefore not simply updatable) and their columns are not updatable. Derived tables and transient tables identified by <query name>s may be simply updatable or generally updatable. The Syntax Rules of Subclause 7.6, "<table reference>", Subclause 7.16, "<query specification>", and Subclause 7.17, "<query expression>", determine whether a derived table *T* is simply updatable or generally updatable, and which of the columns of *T* are updatable. A view is *simply updatable* or *generally updatable* if the user-specified <query expression> of the view is simply updatable or generally

updatable, respectively. A column of a view is *updatable* if the corresponding column of the user-specified <query expression> of the view is updatable. A view is *trigger updatable* if an update INSTEAD OF trigger is defined on that view. A view is *trigger deletable* if a delete INSTEAD OF trigger is defined on that view.

NOTE 28 — The definition that transition tables are not generally updatable merely states that they are not allowed to be the target of an SQL-data change statement. The value of a new transition table can be altered through assignment to a new transition variable column reference.

Some effectively updatable tables, including all base tables whose row type is not derived from a user-defined type that is not instantiable, are also *insertable-into*. Transition tables are not insertable-into. The Syntax Rules of Subclause 7.6, "<table reference>", Subclause 7.16, "<query specification>", Subclause 7.17, "<query expression>", determine whether a transient table identified by a <query name> or a derived table *T* is insertable-into. A view is *insertable-into* if the user-specified <query expression> of the view is insertable-into. A view is *trigger insertable-into* if an insert INSTEAD OF trigger is defined on that view.

The operation of update on a table *T* is permitted if *T* is effectively updatable or trigger-updatable, subject to constraining Access Rules and Conformance Rules. The operation of delete on a table *T* is permitted if *T* is effectively updatable or trigger deletable, subject to constraining Access Rules and Conformance Rules. The operation of insert on a table *T* is permitted if *T* is insertable-into or trigger insertable-into, subject to constraining Access Rules and Conformance Rules.

4.17.7 Table descriptors

This Subclause is modified by Subclause 4.4.4, "Table descriptors" in ISO/IEC 9075-9.

09 A table is described by a table descriptor. A table descriptor is either a transient table descriptor, a base table descriptor, a view descriptor, or a derived table descriptor (for a derived table that is not a view).

Every table descriptor includes:

— The column descriptor of each column in the table.

— The name of the structured type, if any, associated with the table;

— An indication of whether the table is insertable-into or not;

— An indication of whether the table is a referenceable table or not, and an indication of whether the self-referencing column is a system-generated, a user-generated, or a derived self-referencing column;

— The name of its direct supertable, if any;

— A list, possibly empty, of the names of its direct subtables.

A transient table descriptor describes a transient table. In addition to the components of every table descriptor, a transient table descriptor includes:

— If the transient table is defined by a <with list element> contained in a <query expression>, then the <query name>. If the transient table is defined by a <trigger definition>, then the <transition table name>.

A base table descriptor describes a base table. In addition to the components of every table descriptor, a base table descriptor includes:

— The name of the base table;

— An indication of whether the table is a regular persistent base table, a system-versioned table, a global temporary table, a created local temporary table, or a declared local temporary table;

— The descriptor of each period defined for the table;

— If the base table is a global temporary table, a created local temporary table, or a declared local temporary table, then an indication of whether ON COMMIT PRESERVE ROWS was specified or ON COMMIT DELETE ROWS was specified or implied;

— The descriptor of each table constraint specified for the table;

— A non-empty set of functional dependencies, according to the rules given in Subclause 4.26, "Functional dependencies";

— A non-empty set of candidate keys, according to the rules of Subclause 4.27, "Candidate keys";

— A preferred candidate key, which may be (but is not required to be) additionally designated the primary key, according to the Rules in Subclause 4.26, "Functional dependencies".

A derived table descriptor describes a derived table. In addition to the components of every table descriptor, a derived table descriptor includes:

— The <query expression> that defines how the table is to be derived; (When a derived table descriptor is included within a view descriptor, this is known as the hierarchical <query expression> of the view; it is used to find all rows of the view, including rows that may have proper subrows.)

— An indication of whether the derived table is effectively updatable or not;

— An indication of whether the derived table is simply updatable or not;

09 A view descriptor describes a view. In addition to the components of a derived table descriptor, a view descriptor includes:

— The name of the view;

— An indication of whether the view has the CHECK OPTION; if so, whether it is to be applied as CASCADED or LOCAL;

— The original <query expression> of the view; (This <query expression> is used to find the rows of the view that have no proper subrows.)

— The user-specified <query expression> of the view;

NOTE 29 — The user-specified <query expression> is identical to the original <query expression>, except for referenceable views whose reference type has a derived representation; see the General Rules of Subclause 11.32, "<view definition>".

— An indication of whether the view is trigger updatable;

— An indication of whether the view is trigger insertable-into;

— An indication of whether the view is trigger deletable.

4.17.8 Syntactic analysis of derived tables and cursors

This Subclause is modified by Subclause 4.4.5, "Syntactic analysis of derived tables and cursors", in ISO/IEC 9075-9.

An instance of certain BNF non-terminals *S* that define cursors and derived tables may be analyzed to define two directed graphs, known as the *graph of underlying table specifications* and the *graph of generally underlying table specifications*. In the absence of recursive queries, these graphs are acyclic (that is, they are trees).

The nodes of the graph of underlying table specifications *GUT* of *S* are labeled with BNF non-terminals contained in *S*. The nodes of the graph of generally underlying table specifications *GGUT* of *S* are labeled with BNF non-terminals generally contained in *S*. *GUT* is a subgraph of *GGUT*.

NOTE 30 — For example, *GUT* can have a node labeled with "SELECT *C* FROM *T*" as a <query expression> *QE*, another node labeled with "SELECT *C* FROM *T*" as a <query expression body>, and another node labeled with "SELECT *C* FROM *T*" as a

<query specification> contained within *QE*. In addition, *S* can contain more than one <query expression> which is equivalent to "SELECT *C* FROM *T*", and each such <query expression> is a separate node of *GUT*.

The terms *simply underlying table specification*, *underlying table specification*, *leaf underlying table specification*, *generally underlying table specification*, and *leaf generally underlying table specification* define a relationship between instances of certain BNF non-terminals that define a derived table or cursor.

The simply underlying table specifications of derived tables are defined in the Syntax Rules of Subclause 7.16, "<query specification>", and Subclause 7.17, "<query expression>". A <table or query name> has no simply underlying table specifications. The simply underlying table specification of a cursor is the <cursor specification> included in the cursor's result set descriptor (or, for standing cursors, the <cursor specification> included in the cursor declaration descriptor).

The underlying table specifications of a derived table or cursor *DTC* are the simply underlying table specifications of *DTC* and the underlying table specifications of the simply underlying table specifications of *DTC*.

The leaf underlying table specifications of a derived table or cursor *DTC* are the underlying table specifications of *DTC* that do not themselves have any underlying table specifications.

09 The generally underlying table specifications of a derived table or cursor *DTC* are the underlying table specifications of *DTDC* and, for each underlying table specification of *DTC* that is a <table or query name> *TORQN*, the generally underlying table specifications of *TORQN*, which are defined as follows:

— 09 If *TORQN* identifies a base table or if *TORQN* is a <transition table name>, then *TORQN* has no generally underlying table specifications;

— If *TORQN* is a <query name>, then the generally underlying table specifications of *TORQN* are the <query expression body> *QEB* of the <with list element> identified by *TORQN* and the generally underlying table specifications of *QEB*;

— If *TORQN* identifies a view *V*, then the generally underlying table specifications of *TORQN* are the hierarchical <query expression> *QEV* included in the view descriptor of *V* and the generally underlying table specifications of *QEV*.

The leaf generally underlying table specifications of a derived table or cursor *DTC* are the generally underlying table specifications of *DTC* that do not themselves have any generally underlying table specifications.

NOTE 31 — A leaf generally underlying table specification is either a <table or query name> that identifies a base table or a <transition table name>.

Let *S* be a BNF non-terminal. The *leaf underlying tables* of *S* are the tables identified by the leaf underlying table specifications of *S*, and the *leaf generally underlying tables* of *S* are the tables identified by the leaf generally underlying table specifications of *S*.

Let *C* be a cursor and let *CS* be the <cursor specification> of *C*. The *leaf underlying tables* of *C* are the leaf underlying tables of *CS*, and the *leaf generally underlying tables* of *C* are the leaf generally underlying tables of *CS*.

Let *V* be a view and let *HQE* be the hierarchical <query expression> of *V*. The *leaf underlying tables* of *V* are the leaf underlying tables of *HQE*, and the *leaf generally underlying tables* of *V* are the leaf generally underlying tables of *HQE*.

If *DTC* defines an updatable cursor or an effectively updatable derived table, then there are two additional directed graphs, known as the *graph of target underlying table specifications* and the *graph of target generally underlying table specifications*. The graph of target underlying table specifications is a subgraph of the graph of underlying table specifications. The graph of target generally underlying table specifications is a subgraph of the graph of generally underlying table specifications.

NOTE 32 — The purpose of these graphs is to identify the views and base tables that are subject to the primary effects of an SQL-data change statement targeting *DTC*.

The terms *target simply underlying table specification*, *target underlying table specification*, and *target leaf underlying table specification* are used to define the graph of target underlying table specifications. The additional terms *target generally underlying table specification* and *target leaf generally underlying table specification* are used to define the graph of target generally underlying table specifications.

The target simply underlying table specifications of effectively updatable derived tables are defined in the Syntax Rules of Subclause 7.16, "<query specification>", and Subclause 7.17, "<query expression>". The target simply underlying table specification of a <table or query name> is the <table name> or <query name> simply contained in the <table or query name>. (A <transition table name> does not identify an effectively updatable table, and therefore cannot appear in a graph of target underlying table specifications.) A <table name> has no target simply underlying table specifications. The target simply underlying table specification of a <query name> *QN* is the <query expression body> *QEB* of the <with list element> identified by *QN*. The target simply underlying table specification of an updatable cursor is the <cursor specification> included in the cursor's result set descriptor (or, for standing cursors, the <cursor specification> included in the cursor declaration descriptor).

The target underlying table specifications of an effectively updatable derived table or an updatable cursor *DTC* are the target simply underlying table specifications of *DTC* and the target underlying table specifications of the target simply underlying tables of *DTC*.

The target leaf underlying table specifications of *DTC* are the target underlying table specifications of *DTC* that do not themselves have any target underlying table specifications.

NOTE 33 — A target leaf underlying table specification is a <table name> that identifies a base table or an effectively updatable view.

The target generally underlying table specifications of *DTC* are the target underlying table specifications of *DTC* and, for each target underlying table of *DTC* that is a <table name> *VN* that identifies a view *V*, the hierarchical <query expression> *QEV* included in the view descriptor of *V* and the target generally underlying table specifications of *QEV*.

The target leaf generally underlying table specifications of *DTC* are the target generally underlying table specifications of *DTC* that do not themselves have any target generally underlying table specifications.

NOTE 34 — A target leaf generally underlying table is a <table name> that identifies a base table.

Let *S* be a BNF non-terminal. The *target leaf underlying tables* of *S* are the tables identified by the target leaf underlying table specifications of *S*, and the *target leaf generally underlying tables* of *S* are the tables identified by the target leaf generally underlying table specifications of *S*.

Let *C* be a cursor and let *CS* be the <cursor specification> of *C*. The *target leaf underlying tables* of *C* are the *target leaf underlying tables* of *CS*, and the *target leaf generally underlying tables* of *C* are the target leaf generally underlying tables of *CS*.

Let *V* is a view and let *HQE* be the hierarchical <query expression> of *V*. The *target leaf underlying tables* of *V* are the *target leaf underlying tables* of *HQE*, and the *target leaf generally underlying tables* of *V* are the target leaf generally underlying tables of *HQE*.

4.17.9 Referenceable tables, subtables, and supertables

A table *RT* whose row type is derived from a structured type *ST* is called a *typed table*. Only a base table or a view can be a typed table. A typed table has columns corresponding, in name and declared type, to every attribute of *ST* and one other column *REFC* that is the self-referencing column of *RT*; let *REFCN* be the <column name> of *REFC*. The declared type of *REFC* is necessarily REF(*ST*) and the nullability characteristic of *REFC* is *known not nullable*. If *RT* is a base table, then the table constraint "UNIQUE(*REFCN*)" is implicit in the definition of *RT*. A typed table is called a *referenceable table*. A self-referencing column cannot be updated. Its value is determined during the insertion of a row into the referenceable table. The value of a system-generated self-referencing column and a derived self-referencing column is automatically generated when the row is inserted into the referenceable table. The value of a user-generated self-referencing column is supplied as part of the candidate row to be inserted into the referenceable table.

A table T_a is a *direct subtable* of another table T_b if and only if the <table name> of T_b is contained in the <subtable clause> contained in the <table definition> or <view definition> of T_a. Both T_a and T_b shall be created on a structured type and the structured type of T_a shall be a direct subtype of the structured type of T_b.

A table T_a is a *subtable* of a table T_b if and only if at least one of the following is true:

— T_a and T_b are the same named table;

— T_a is a direct subtable of T_b;

— There is a table T_c such that T_a is a direct subtable of T_c and T_c is a subtable of T_b.

A table T is considered to be one of its own subtables. Subtables of T other than T itself are called its *proper subtables*. A table shall not have itself as a proper subtable.

A table T_b is called a *supertable* of a table T_a if T_a is a subtable of T_b. If T_a is a direct subtable of T_b, then T_b is called a *direct supertable* of T_a. If T_a is a proper subtable of T_b, then T_b is called a *proper supertable* of T_a. A table that is not a subtable of any other table is called a *maximal supertable*.

Let T_a be a maximal supertable and T be a subtable of T_a. The set of all subtables of T_a (which includes T_a itself) is called the *subtable family* of T or (equivalently) of T_a. Every subtable family has exactly one maximal supertable.

A *leaf table* is a table that does not have any proper subtables.

Those columns of a subtable T_a of a structured type ST_a that correspond to the inherited attributes of ST_a are called *inherited columns*. Those columns of T_a that correspond to the originally-defined attributes of ST_a are called *originally-defined columns*.

Let *TB* be a subtable of *TA*. Let *SLA* be the <value expression> sequence implied by the <select list> "*" in the <query specification> "SELECT * FROM *TA*". For every row *RB* in the value of *TB* there exists exactly one row *RA* in the value of *TA* such that *RA* is the result of the <row subquery> "SELECT *SLA* FROM VALUES *RRB*", where *RRB* is some <row value constructor> whose value is *RB*. *RA* is said to be the *superrow* in *TA* of *RB* and *RB* is said to be the *subrow* in *TB* of *RA*. If *TB* is a proper subtable of *TA*, then *RA* is a proper superrow of *RB*, and *RB* is a proper subrow of *RA*. If *TA* is a base table, then the one-to-one correspondence between superrows and subrows is guaranteed by the requirement for a unique constraint to be specified for some supertable of *TA*. If *TA* is a view, then such one-to-one correspondence is guaranteed by the requirement for a unique constraint to be specified on the leaf generally underlying table of *TA*.

Users shall have the UNDER privilege on a table before they can use the table in a subtable definition. A table can have more than one proper subtable. Similarly, a table can have more than one proper supertable.

4.17.10 Operations involving tables

Table values are operated on and returned by <query expression>s. The syntax of <query expression> includes various internal operators that operate on table values and return table values. In particular, every <query expression> effectively includes at least one <from clause>, which operates on one or more table values and returns a single table value. A table value operated on by a <from clause> is specified by a <table reference>.

In a <table reference>, ONLY can be specified to exclude from the result rows that have subrows in proper subtables of the referenced table.

In a <table reference>, <sample clause> can be specified to return a subset of result rows depending on the <sample method> and <sample percentage>. If the <sample clause> contains <repeatable clause>,

then repeated executions of that <table reference> return a result table with identical rows for a given <repeat argument>, provided certain implementation-defined (IA200) conditions are satisfied.

A <table reference> that specifies a system-versioned table by default returns the current system rows. A <query system time period specification> can be specified to return historical system rows.

Certain table updating operations, specified by SQL-data change statements, are available in connection with effectively updatable tables and trigger updatable tables, subject to applicable Access Rules and Conformance Rules. The value of an effectively updatable table *T* or a trigger updatable table *T* is determined by the result of the most recently executed SQL-data change statement (see Subclause 4.41.2, "SQL-statements classified by function") operating on *T*.

The effect of an SQL-data change statement on a base table is specified by the General Rules of Subclause 15.8, "Effect of deleting rows from base tables", Subclause 15.11, "Effect of inserting tables into base tables", and Subclause 15.14, "Effect of replacing rows in base tables".

The effect of an SQL-data change statement on a derived table is specified by the General Rules of Subclause 15.9, "Effect of deleting some rows from a derived table", Subclause 15.12, "Effect of inserting a table into a derived table", and Subclause 15.15, "Effect of replacing some rows in a derived table".

The effect of an SQL-data change statement on a viewed table is specified by the General Rules of Subclause 15.10, "Effect of deleting some rows from a viewed table", Subclause 15.13, "Effect of inserting a table into a viewed table", and Subclause 15.16, "Effect of replacing some rows in a viewed table".

An SQL-data change statement on a base table *T* has a primary effect (on *T* itself), zero or more secondary effects (not necessarily on *T*), and zero or more tertiary effects (not necessarily on *T*). If *T* is a system-versioned table, then the effects are restricted to the subset of rows that correspond to current system rows.

The primary effect of a <delete statement: positioned> on a regular base table *T* is to delete exactly one specified row from *T*. The primary effect of a <delete statement: searched> on a regular base table *T* is to delete zero or more rows from *T*.

The primary effect of a <delete statement: positioned> on a system-versioned table *T* is to replace exactly one specified row of *T* with its system-time period end column value set to the transaction timestamp of the SQL-transaction in which the <delete statement: positioned> executes.

The primary effect of a <delete statement: searched> on a system-versioned table *T* is to replace zero or more rows of *T* with their system-time period end column values set to the transaction timestamp of the SQL-transaction in which the <delete statement: searched> executes.

The primary effect of a <truncate table statement> on a regular base table *T* is to delete all rows from *T* without causing the execution of any trigger specified for *T*.

The primary effect of an <update statement: positioned> on a regular base table *T* is to replace exactly one specified row in *T* with some specified row. The primary effect of an <update statement: searched> on a regular base table *T* is to replace zero or more rows in *T*.

The primary effect of an <update statement: positioned> on a system-versioned table *T* is to replace exactly one specified row *R* in *T* with some specified row with the system-time period start column value set to the transaction timestamp *VT* of the SQL-transaction in which that <update statement: positioned> executes and the value of the system-time period end column is set to the greatest value supported by the declared type of the system-time period end column, followed by the insertion of a copy of *R* before the update, with the value of the system-time period end column set to *VT*.

The primary effect of an <update statement: searched> on a system-versioned table *T* is, for each candidate row *R* to be replaced, to replace *R* with some specified row with the system-time period start column value set to the transaction timestamp *VT* of the SQL-transaction in which that <update statement: searched> executes and the value of the system-time period end column set to the greatest value supported

by the declared type of the system-time period end column, followed by the insertion of a copy of *R* before the update, with the value of the system-time period end column set to *VT*.

The primary effect of an <insert statement> on a regular base table *T* is to insert into *T* each of the zero or more rows contained in a specified table.

The primary effect of an <insert statement> on a system-versioned table *T* is to insert into *T* each of the zero or more rows contained in a specified table with the value of the system-time period start column set to the transaction timestamp of the SQL-transaction in which that <insert statement> executes and the value of the system-time period end column set to the greatest value supported by the declared type of the system-time period end column.

The primary effect of a <merge statement> on a regular base table *T* is to delete zero or more rows from *T* and/or to replace zero or more rows in *T* with specified rows and/or to insert into *T* zero or more specified rows.

The primary effect of a <merge statement> on a system-versioned table *T* is to delete zero or more rows in T as described in the 11th paragraph of this Subclause and/or to replace zero or more rows in *T* with specified rows as described in the 16th paragraph of this Subclause and/or to insert into *T* zero or more specified rows as described in the 18th paragraph of this Subclause.

Each of the table updating operations, when applied to a base table *T*, can have various secondary and tertiary effects. Such secondary and tertiary effects may include alteration or reversal of the primary effect.

Secondary effects might arise from the existence of:

— Proper subtables and proper supertables of *T*, whose values might be affected by updating operations on *T*.

— BEFORE row triggers specified for *T*, which might alter the values of columns in rows to be inserted or replaced.

Tertiary effects might arise from the existence of:

— An application-time period of *T*, which may result in the insertion of up to two rows for each row that is replaced or deleted. These inserted rows retain the information for the portion of a replaced or deleted row that is outside the time period specified by the FOR PORTION OF clause.

— Cascaded operations specified in connection with integrity constraints of *T*, which might result in secondary effects on tables referenced by such constraints.

— AFTER triggers specified for *T*, which might specify table updating operations on effectively updatable tables other than *T*.

Columns that are identity columns and self-referencing columns that are system generated or derived are *overridable columns*. The values of overridable columns can be generated automatically when a row is inserted (using <insert statement> or a <merge insert specification> of a <merge statement>). It is permitted that they are not changed when a row is updated (by <update statement: positioned>, <update statement: searched>, or a <merge update specification> of a <merge statement>). The <override clause> may be used in <insert statement> or <merge insert specification>, with some restrictions found in the Syntax Rules, to coerce the value of an overridable column to either a user-supplied value or a system-generated value. OVERRIDING USER VALUE indicates that the values of overridable columns in the <insert column list> are to be system-generated even if the SQL-statement provides an explicit value for such columns. OVERRIDING SYSTEM VALUE indicates that the values assigned by the SQL-statement to the overridable columns in the <insert column list> are to be used rather than the automatically generated values. In the latter case, if the value assigned by the SQL-statement to an overridable column *OC* is DEFAULT, or if the overridable column is not listed in the <insert column list>, then the system-generated value is assigned to *OC*.

NOTE 35 — Self-referencing columns that are user-generated and generated columns are not overridable columns. Identity columns are overridable columns, even if they have an indication that they are generated always. In the latter case, "generated always" is understood to mean "generated automatically unless OVERRIDING SYSTEM VALUE is specified". One use of OVERRIDING SYSTEM VALUE is replication, when overridable columns must be copied and not automatically generated. System-time period columns are not overridable columns, though OVERRIDING USER VALUE can be specified with them. In this case, the user-supplied values are overridden with system-generated ones.

4.17.11 Range variables

An operation involving a table *T* may define one or more *range variable*s that ranges over rows of *T*, referencing each row in turn in an implementation-dependent (US052) order. Thus, each reference to a range variable *RV* of *T* references exactly one row of *T*. *T* is said to be the *table associated with RV*.

A range variable is a <correlation name>, a <table name>, or a <query name>.

A range variable *RV* is declared by a BNF non-terminal *BNF*, such as a <from clause>, <table reference>, <table factor>, or <table primary>. *BNF* is said to *expose RV*.

Every range variable has a *scope*. The scope of a range variable is the portion of an SQL-statement in which the range variable is visible. Two range variables that are equivalent (as <identifier>s or <table name>s) are nevertheless distinct if they have different scopes.

The primary use of a range variable *RV* is to qualify column names in <column reference>s and period names in <period reference>s contained within the scope of *RV*. Row pattern variables are a kind of range variable used only within <row pattern recognition clause>, where they are used to define row patterns, and may be used as an argument of <classifier function>, in addition to their use as qualifiers in <column reference>s. Subclause 6.6, "<identifier chain>", is responsible for determining the range variable that qualifies a column name in a <column reference> or a period name in a <period reference>, as well as disambiguating similar <period>-qualified BNF non-terminals.

Each range variable has an *associated column list* and an *associated period list*; these are the list of columns (not necessarily all columns) and list of periods (not necessarily all periods), respectively, of the table associated with that range variable. A range variable can appear as the qualifier in a <column reference> only if the name of the column is included in the associated column list of that range variable. Similarly, a range variable can appear as the qualifier in a <period reference> only if the name of the period is included in the associated period list of that range variable.

The following BNF non-terminals define a single range variable:

— <delete statement: positioned>.

NOTE 36 — The range variable defined by a <delete statement: positioned> has no scope and therefore can never be used to qualify a column name.

— <delete statement: searched>.

— <update statement: positioned>.

— <update statement: searched>.

The following BNF non-terminals may define one or more range variables:

— <table primary>.

— <table factor>.

— <table reference>.

— <from clause>.

— <trigger definition>.

The following BNF non-terminals may define two or more range variables:

— <joined table>.

— <merge statement>.

For a BNF non-terminal that defines a single range variable *RV*, the associated column list and the associated period list of *RV* include every column and every period, respectively, of the table associated with *RV*.

For a BNF non-terminal that defines multiple range variables, excluding <trigger definition> and <merge statement>, the associated column lists and the associated period lists of individual range variables are mutually exclusive and exhaustive, so that every column and every period in the result table of that BNF non-terminal has exactly one range variable that may be used to qualify that column or period, respectively.

4.17.12 Identity columns

The columns of a base table *BT* can optionally include not more than one *identity column*. The declared type of an identity column is either an exact numeric type with scale 0 (zero), INTEGER for example, or a distinct type whose source type is an exact numeric type with scale 0 (zero). An identity column has a *start value*, an *increment*, a *maximum value*, a *minimum value*, and a *cycle option*. An identity column is associated with an internal sequence generator *SG*. Let *IC* be the identity column of *BT*.

When a row *R* is presented for insertion into *BT*, if *R* does not contain a column corresponding to *IC*, then the General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are invoked to create a value *V*. The value for *IC* is *V*. If *R* contains a column corresponding to *IC* and the <insert statement> or the <merge statement> that inserts *R* specifies OVERRIDING SYSTEM VALUE, then the value in *R* is assigned to *IC*. If *R* contains a column corresponding to *IC* and the <insert statement> or the <merge statement> that inserts *R* specifies OVERRIDING USER VALUE, then the value in *R* is ignored and *IC* is assigned *V*. The definition of an identity column may specify GENERATED ALWAYS or GENERATED BY DEFAULT.

NOTE 37 — "Start value", "increment", "maximum value", "minimum value", and "cycle option" are defined in Subclause 4.29, "Sequence generators".

NOTE 38 — The notion of an internal sequence generator being associated with an identity column is used only for definitional purposes in this document.

4.17.13 Base columns and generated columns

A column of a base table is either a *base column* or a *generated column*. A base column is one that is not a generated column. A generated column is one whose values are determined by evaluation of a *generation expression*, a <value expression> whose declared type is by implication that of the column. A generation expression can reference base columns of the base table to which it belongs but cannot otherwise access SQL-data. Thus, the value of the field corresponding to a generated column in row *R* is determined by the values of zero or more other fields of *R*.

A generated column *GC* depends on each column that is referenced by a <column reference> in its generation expression, and each such referenced column is a *parametric column* of *GC*.

4.17.14 Grouped tables

A *grouped table* is a set of groups derived during the evaluation of a <group by clause>. A group *G* is a collection of rows in which, for every grouping column *GC*, if the value of *GC* in some row is not distinct from *GV*, then the value of *GC* in every row is *GV*; moreover, if *R1* is a row in group *G1* of grouped table *GT* and *R2* is a row in *GT* such that for every grouping column *GC* the value of *GC* in *R1* is not distinct from the value of *GC* in *R2*, then *R2* is in *G1*. Every row in *GT* is in exactly one group. A group may be considered as a table. Set functions operate on groups.

4.17.15 Windowed tables

A *windowed table* is a table together with one or more windows. A *window* is a transient data structure associated with a <table expression>. A window is defined explicitly by a <window definition> or implicitly by an <in-line window specification>. Implicitly defined windows have an implementation-dependent (UV061) window name. A window is used to specify window partitions and window frames, which are collections of rows used in the definition of <window function>s.

Every window defines a *window partitioning* of the rows of the <table expression>. The window partitioning is specified by a list of columns. Window partitioning is similar to forming groups of a grouped table. However, unlike grouped tables, each row is retained in the result of the <table expression>. The *window partition* of a row *R* is the collection of rows *R2* that are not distinct from *R*, for all columns enumerated in the window partitioning clause. The window partitioning clause is optional; if omitted, there is a single window partition consisting of all the rows in the result.

If a <table expression> is grouped and also has a window, then there is a syntactic transformation that segregates the grouping into a <derived table>, so that the window partitions consist of rows of the <derived table> rather than groups of rows.

A window may define a *window ordering* of rows within each window partition defined by the window. The window ordering of rows within window partitions is specified by a list of <value expression>s, followed by ASC (for ascending order) or DESC (for descending order). In addition, NULLS FIRST or NULLS LAST may be specified, to indicate whether a null value shall appear before or after all non-null values in the ordered sequence of each <value expression>.

A *window ordering group* is a maximal set of rows in a window partition that are peers according to the window ordering. Although the ordering of rows within a window ordering group is implementation-dependent (US028), it is possible to totally order the window ordering groups of a window partition, as follows: if *WOG1* and *WOG2* are two window ordering groups contained in the same window partition *P*, then *WOG1* precedes *WOG2* if and only if some row of *WOG1* precedes some row of *WOG2*.

Optionally, a window may define a *window frame* for each row *R*. A window frame is always defined relative to the current row. A window frame is specified by up to six syntactic elements:

— The choice of RANGE, to indicate a logical definition of the window frame by offsetting forward or backward from the current row by an increment or decrement to the value of the <sort key>; GROUPS, to indicate a logical definition of the window frame by counting forward or backward a number of window ordering groups (as defined by the <sort key>s) from the window ordering group containing the current row; or ROWS, to indicate a physical definition of the window frame, by counting rows forward or backward from the current row.

— A starting row, which may be the first row of the window partition of *R*, the current row, or some row determined by a logical or physical offset from the current row.

— An ending row, which may be the last row of the window partition of *R*, the current row, or some row determined by a logical or physical offset from the current row.

— A <window frame exclusion>, indicating whether to exclude the current row and/or its peers (if not already excluded by being prior to the starting row or after the ending row).

— A window row pattern recognition, which is specified by a <row pattern common syntax> and (optionally) a <row pattern measures>.

A window is described by a *window structure descriptor*, including:

— The window name.

— Optionally, the ordering window name—that is, the name of another window, called the *ordering window*, that is used to define the partitioning and ordering of the present window.

— The window partitioning clause—that is, a <window partition clause> if any is specified in either the present <window specification> or in the window descriptor of the ordering window.

— The window ordering clause—that is, a <window order clause> if any is specified in either the present <window specification> or in the window descriptor of the ordering window.

— The window framing clause—that is, a <window frame clause>, if any.

In general, two <window function>s are computed independently, each one performing its own sort of its data, even if they use the same data and the same <sort specification list>. Since sorts may specify partial orderings, the computation of <window function>s is inevitably non-deterministic to the extent that the ordering is not total. Nevertheless, the user may desire that two <window function>s be computed using the same ordering, so that, for example, two moving aggregates move through the rows of a partition in precisely the same order. Two <window function>s are computed using the same (possibly non-deterministic) window ordering of the rows if and only if at least one of the following is true:

— The <window function>s identify the same window structure descriptor.

— The <window function>s' window structure descriptors have window partitioning clauses that enumerate the same number of column references, and those column references are pairwise equivalent in their order of occurrence; and their window structure descriptors have window ordering clauses with the same number of <sort key>s, and those <sort key>s are all column references, and those column references are pairwise equivalent in their order of occurrence, and the <sort specification>s pairwise specify or imply <collate clause>s that specify equivalent <collation name>s, the same <ordering specification> (ASC or DESC), and the same <null ordering> (NULLS FIRST or NULLS LAST).

— The window structure descriptor of one <window function> is the ordering window of the other <window function>, or both window structure descriptors identify the same ordering window.

4.18 Data analysis operations

This Subclause is modified by Subclause 4.5, "Data analysis operations", in ISO/IEC 9075-14.

4.18.1 Introduction to data analysis operations

A data analysis function is a function that returns a value derived from a number of rows in the result of a <table expression>. A data analysis function may only be invoked as part of a <query specification>, <select statement: single row>, or simple table query, and then only in certain contexts, identified below. A data analysis function is one of:

— A group function, which is invoked on a grouped table and computes a grouping operation or an aggregate function from a group of the grouped table.

— A window function, which is invoked on a windowed table and computes a value for each row of the windowed table.

4.18.2 Group functions

A group function may only appear in the following:

— The <select list>, <having clause> or <window clause> of a <query specification> or a <select statement: single row>.

— In a <row pattern measures> or <row pattern definition search condition>.

A group function is one of:

— The *grouping operation*.

— A *group aggregate function*.

The grouping operation is of the form `GROUPING(<column reference>)`. The result of such an invocation is 1 (one) in the case of a row whose values are the results of aggregation over that <column reference> during the execution of a grouped query containing CUBE, ROLLUP, or GROUPING SETS, and 0 (zero) otherwise.

4.18.3 Window functions

A window function is a function whose result for a given row is derived from the window frame of that row as defined by a window structure descriptor of a windowed table. Window functions may only appear in the <select list> of a <query specification> or <select statement: single row>, or the <order by clause> simply contained in a <query expression> that is a simple table query.

A window function is one of:

— A rank function.

— A distribution function.

— The row number function.

— A window aggregate function.

— The ntile function.

— The lead function.

— The lag function.

— The first-value function.

— The last-value function.

— The nth-value function.

— A row pattern measure function.

The rank functions compute the ordinal rank of a row *R* within the window partition of *R* as defined by a window structure descriptor, according to the window ordering of those rows, also specified by the same window structure descriptor. Rows that are not distinct with respect to the window ordering within their window partition are assigned the same rank. There are two variants, indicated by the keywords RANK and DENSE_RANK.

— If RANK is specified, then the rank of row *R* is defined as 1 (one) plus the number of rows that precede *R* and are not peers of *R*.

NOTE 39 — This implies that if two or more rows are not distinct with respect to the window ordering, then there will be one or more gaps in the sequential rank numbering.

— If DENSE_RANK is specified, then the rank of row *R* is defined as the number of rows preceding and including *R* that are distinct with respect to the window ordering.

NOTE 40 — This implies that there are no gaps in the sequential rank numbering of rows in each window partition.

The distribution functions compute a relative rank of a row *R* within the window partition of *R* defined by a window structure descriptor, expressed as an approximate numeric ratio between 0.0 and 1.0. There are two variants, indicated by the keywords PERCENT_RANK and CUME_DIST.

— If PERCENT_RANK is specified, then the relative rank of a row *R* is defined as $(RK-1)/(NR-1)$, where *RK* is defined to be the RANK of *R* and *NR* is defined to be the number of rows in the window partition of *R*.

— If CUME_DIST is specified, then the relative rank of a row *R* is defined as *NP*/*NR*, where *NP* is defined to be the number of rows preceding or peer with *R* in the window ordering of the window partition of *R* and *NR* is defined to be the number of rows in the window partition of *R*.

The ROW_NUMBER function computes the sequential row number, starting with 1 (one) for the first row, of the row within its window partition according to the window ordering of the window.

The window aggregate functions compute an <aggregate function> (COUNT, SUM, AVG, etc.), the same as a group aggregate function, except that the computation aggregates over the window frame of a row rather than over a group of a grouped table. The hypothetical set functions are not permitted as window aggregate functions.

Aggregated arguments of the <aggregate function> of a window aggregate function *WAF* may contain <nested window function>s. There are two <nested window function>s: <nested row number function> and <value_of expression at row>. Both <nested window function>s use <row marker>s, which are keywords that denote specific rows. Let *R* be a particular row for which a <nested window function> is to be evaluated; let *F* be the window frame determined by *R*, and let *P* be the window partition containing *R*. The <row marker>s are:

— BEGIN_PARTITION, the first row of *P*.

— BEGIN_FRAME, the first row of *F*.

— END_FRAME, the last row of *F*.

— END_PARTITION, the last row of *P*.

— CURRENT_ROW, the row *R*.

— FRAME_ROW, the row within *F* that varies from the first row to the last row of *F* during evaluation of *WAF*.

<nested row number function> returns the window partition row number of a <row marker>. (Rows within a window partition are numbered sequentially starting with 1 (one) according to the window ordering).

<value_of expression at row> evaluates a <value expression> on a row indicated by a <row marker>, plus or minus an optional offset. If the combination of <row marker> plus or minus offset would lie outside the window partition, the value of <value_of expression at row> is the value of an optional argument, or the null value if this argument is omitted.

The ntile function takes a <simple value specification> or a <dynamic parameter specification> that evaluates to an exact numeric value *n* with scale 0 (zero) as an argument and computes an exact numeric value with scale 0 (zero), ranging from 1 (one) to *n*, for each row *R* within the window partition of *R* defined by a window structure descriptor.

The lead and lag functions each take three arguments, a <value expression> *VE*, an <exact numeric literal> *OFFSET*, and a <value expression> *DEFAULT*. For each row *R* within the window partition *P* of *R* defined by a window structure descriptor, the lag function returns the value of *VE* evaluated on a row that is *OFFSET* number of rows before *R* within *P*, and the lead function returns the value of *VE* evaluated on a row that is *OFFSET* number of rows after *R* within *P*. The value of *DEFAULT* is returned as the result if there is no row corresponding to the *OFFSET* number of rows before *R* within *P* (for the lag function) or after *R* within *P* (for the lead function). In addition, RESPECT NULLS or IGNORE NULLS can be specified to indicate whether the rows within *P* for which *VE* evaluates to the null value are preserved or eliminated.

The first-value and last-value functions take an arbitrary <value expression> *VE* as an argument and, for each row *R* of a windowed table, return the value of *VE* evaluated on the first row of the window frame of *R* (for the first-value function) or the last row of the window frame of *R* (for the last_value function) defined by a window structure descriptor. In addition, RESPECT NULLS or IGNORE NULLS can be specified to indicate whether the rows for which *VE* evaluates to the null value are preserved or eliminated.

The nth-value function takes an arbitrary <value expression> *VE* and a <simple value specification> or a <dynamic parameter specification> that evaluates to an exact numeric value *n* with scale 0 (zero) as arguments and, for each row *R* of a windowed table, returns the value of *VE* evaluated on the *n*-th row from the first (if FROM FIRST is specified or implied) or the last (if FROM LAST is specified) row of the window frame of *R* defined by a window structure descriptor. In addition, RESPECT NULLS or IGNORE NULLS can be specified to indicate whether the rows for which *VE* evaluates to the null value are preserved or eliminated.

A row pattern measure function is specified by a <measure name>, whose value is specified by a <row pattern measure expression>. See Subclause 4.18.5, "Row pattern measures".

4.18.4 Aggregate functions

This Subclause is modified by Subclause 4.5.1, "Aggregate functions", in ISO/IEC 9075-14.

An aggregate function is a function whose result is derived from an aggregation of rows defined by one of:

— The grouping of a grouped table, in which case the aggregate function is a group aggregate function, or set function, and for each group there is one aggregation, which includes every row in the group.

— The window frame of a row *R* of a windowed table relative to a particular window structure descriptor, in which case the aggregate function is a window aggregate function, and the aggregation consists of every row in the window frame of *R*, as defined by the window structure descriptor.

— Row pattern matching (either in <row pattern recognition clause> or <window clause>), in which case the aggregation is performed over a set of rows identified by a row pattern variable in a row pattern match.

Optionally, the collection of rows in an aggregation may be filtered, retaining only those rows that satisfy a <search condition> that is specified by a <filter clause>.

The result of the aggregate function COUNT(*) is the number of rows in the aggregation.

Every other aggregate function may be classified as a *unary group aggregate function*, a *binary group aggregate function*, an *inverse distribution*, a *hypothetical set function*, or a *JSON aggregate function*.

Every unary aggregate function takes an arbitrary <value expression> as the argument; most unary aggregate functions can optionally be qualified with either DISTINCT or ALL. Of the rows in the aggregation, the following are removed from the rows that qualify:

— If DISTINCT is specified, then redundant duplicates.

— Every row in which the <value expression> evaluates to the null value.

If no row qualifies, then the result of COUNT is 0 (zero), and the result of any other aggregate function is the null value.

14 Otherwise (i.e., at least one row qualifies), the result of the aggregate function is:

— If COUNT <value expression> is specified, then the number of rows that qualify.

— If SUM is specified, then the sum of <value expression> evaluated for each row that qualifies.

— If AVG is specified, then the average of <value expression> evaluated for each row that qualifies.

— If MAX is specified, then the maximum value of <value expression> evaluated for each row that qualifies.

— If MIN is specified, then the minimum value of <value expression> evaluated for each row that qualifies.

— If EVERY is specified, then *True* if the <value expression> evaluates to *True* for every row that qualifies; otherwise, *False*.

— If ANY or SOME is specified, then *True* if the <value expression> evaluates to *True* for at least one row that qualifies; otherwise, *False*.

— If VAR_POP is specified, then the population variance of <value expression> evaluated for each row that qualifies, defined as the sum of squares of the difference of <value expression> from the mean of <value expression>, divided by the number of rows that qualify.

— If VAR_SAMP is specified, then the sample variance of <value expression> evaluated for each row that qualifies, defined as the sum of squares of the difference of <value expression> from the mean of <value expression>, divided by the number of rows that qualify minus 1 (one).

— If STDDEV_POP is specified, then the population standard deviation of <value expression> evaluated for each row that qualifies, defined as the square root of the population variance.

— If STDDEV_SAMP is specified, then the sample standard deviation of <value expression> evaluated for each row that qualifies, defined as the square root of the sample variance.

— If ARRAY_AGG is specified, then an array value with one element formed from the <value expression> evaluated for each row that qualifies.

— If LISTAGG is specified, then a character string value that is the concatenation of the <character value expression> evaluated for each row that qualifies.

— If ANY_VALUE is specified, then the <value expression> is evaluated for an implementation-dependent (UV062) row that qualifies.

Neither DISTINCT nor ALL are allowed to be specified for VAR_POP, VAR_SAMP, STDDEV_POP, or STDDEV_SAMP; redundant duplicates are not removed from the rows that qualify when computing these functions.

The binary aggregate functions take a pair of arguments, the <dependent variable expression> and the <independent variable expression>, which are both <numeric value expression>s. Any row in which either argument evaluates to the null value is removed from the rows that qualify. If there are no rows that qualify, then the result of REGR_COUNT is 0 (zero), and the other binary aggregate functions result in the null value. Otherwise, the computation concludes and the result is:

— If REGR_COUNT is specified, then the number of rows remaining in the group.

— If COVAR_POP is specified, then the population covariance, defined as the sum of products of the difference of <independent variable expression> from its mean times the difference of <dependent variable expression> from its mean, divided by the number of rows that qualify.

— If COVAR_SAMP is specified, then the sample covariance, defined as the sum of products of the difference of <independent variable expression> from its mean times the difference of <dependent variable expression> from its mean, divided by the number of rows that qualify minus 1 (one).

— If CORR is specified, then the correlation coefficient, defined as the ratio of the population covariance divided by the product of the population standard deviation of <independent variable expression> and the population standard deviation of <dependent variable expression>.

— If REGR_R2 is specified, then the square of the correlation coefficient.

— If REGR_SLOPE is specified, then the slope of the least-squares-fit linear equation determined by the (<independent variable expression>, <dependent variable expression>) pairs.

— If REGR_INTERCEPT is specified, then the y-intercept of the least-squares-fit linear equation determined by the (<independent variable expression>, <dependent variable expression>) pairs.

— If REGR_SXX is specified, then the sum of squares of <independent variable expression>.

— If REGR_SYY is specified, then the sum of squares of <dependent variable expression>.

— If REGR_SXY is specified, then the sum of products of <independent variable expression> times <dependent variable expression>.

— If REGR_AVGX is specified, then the average of <independent variable expression>.

— If REGR_AVGY is specified, then the average of <dependent variable expression>.

There are two inverse distribution functions, PERCENTILE_CONT and PERCENTILE_DISC. Both inverse distribution functions specify an argument and an ordering of a value expression. The value of the argument shall be between 0 (zero) and 1 (one) inclusive. The value expression is evaluated for each row that qualifies, nulls are discarded, and the remaining rows are ordered. The computation concludes:

— If PERCENTILE_CONT is specified, by considering the pair of consecutive rows that are indicated by the argument, treated as a fraction of the total number of rows that qualify, and interpolating the value of the value expression evaluated for these rows.

— If PERCENTILE_DISC is specified, by treating the collection of rows that qualify as a window partition of the CUME_DIST window function, using the specified ordering of the value expression as the window ordering, and returning the first value expression whose cumulative distribution value is greater than or equal to the argument.

The hypothetical set functions are related to the window functions RANK, DENSE_RANK, PERCENT_RANK, and CUME_DIST, and use the same names, though with a different syntax. These functions take an argument *A* and an ordering of a value expression *VE*. *VE* is evaluated for all rows that qualify. This collection of values is augmented with *A*; the resulting collection is treated as a window partition of the corresponding window function whose window ordering is the ordering of the value expression. The result of the hypothetical set function is the value of the eponymous window function for the hypothetical "row" that contributes *A* to the collection.

The JSON aggregate functions transform information in rows of SQL tables into JSON objects (JSON_OBJECTAGG) and JSON arrays (JSON_ARRAYAGG).

4.18.5 Row pattern measures

A *row pattern measure* is a named scalar expression whose value is computed based on a match found by row pattern recognition. A row pattern measure is accessed using a row pattern measure column (if row pattern recognition is performed using <row pattern recognition clause>) or a row pattern measure function (if row pattern recognition is performed using a window). Row pattern measures are specified by <row pattern measure definition>s contained in a <row pattern measures>. Each <row pattern measure definition> defines a row pattern measure, with a <measure name> whose value is specified by a corresponding <row pattern measure expression>. A <row pattern measure expression> is a <value expression>, which may contain, for example, <set function specification>s and column references to any column of the row pattern input table. Row pattern measures may also use the following special functions:

— <row pattern navigation operation>, which can evaluate expressions by navigating to other rows in the row pattern partition by logical and/or physical offsets.

— <classifier function>, which returns a character string that is equivalent to the name of the primary row pattern variable that matches a row of a retained row pattern match.

— <match number function>, which returns the sequential number of the current retained row pattern match within its row pattern partition. (<match number function> is not available when performing row pattern matching within a window.)

4.19 Row pattern matching

4.19.1 Introduction to row pattern matching

Business process applications are composed of complex sequences of possibly many events. For example:

— Security applications must be able to identify unusual behaviors that may comprise efforts to defeat a system's security.

— Financial applications require the ability to detect fraud in, for example, stock trades.

— Material handling and shipping applications are expected to track material and packages through all possible valid paths.

In modern information systems technology, a *regular expression* is typically used to represent patterns for searching character strings and a variety of similar purposes. A form of regular expression can be used to detect patterns in rows of a table that is ordered in some manner (often sequentially by time of row creation).

Regular expressions are used to specify patterns of interest; the ordered rows of partitions of the row pattern input table are scanned to discover whether the patterns exist and, if so, to retrieve detailed or aggregated information contained in the rows that match certain components of the pattern. Row pattern variables defined as part of the regular expressions are used to reference those rows that match the components of the pattern represented by the row pattern variables, a special kind of range variable.

Columns of the rows referenced by each row pattern variable in a row pattern can be used in ordinary value expressions whose values can be returned in the row pattern output table upon successful matching of that pattern.

Row pattern recognition may be specified either in the <from clause> using the <row pattern recognition clause>, or in a window definition. This Subclause describes the syntax and semantics of <row pattern recognition clause>. Row pattern recognition using a window definition has much the same syntax, only embedded within a <window definition>. The shared syntax generally has the same semantics in either <row pattern recognition clause> or <window definition>. Row pattern recognition in windows is treated specifically in Subclause 4.17.15, "Windowed tables", and Subclause 4.18.3, "Window functions".

4.19.2 Matching rows with a pattern

The process of row pattern matching operates on an ordinary table (a base table, a view, or a derived table produced as an intermediate result during query processing), called the *row pattern input table*, and produces another (derived) table, called the *row pattern output table*. In this, it may be considered analogous to various components of <table expression>s. Because multiple tables that are joined in a <from clause> can be the subject of row pattern matching, the syntax that specifies row pattern matching is an optional clause available for use on most of the alternatives of <table primary>.

Row pattern recognition is specified through the use of the <row pattern recognition clause>. Although the process of using row patterns to match rows of a table has several optional steps, some of which complicate the process significantly, the basic process is relatively straightforward:

— If specified (<row pattern partition by>), the row pattern input table is partitioned based on the values of the column or columns specified as *row pattern partitioning columns*; each row pattern partition has equal (more precisely, non-distinct) values of the row pattern partitioning column or columns in all of its rows. If not specified, the row pattern input table is considered to have only one row pattern partition.

— If specified (<row pattern order by>), each row pattern partition is ordered based on the values of the column references simply contained in the <row pattern order by>; such ordering may be specified to be ascending or descending on a column-by-column basis (as cursor ordering is done). If

not specified, then the ordering of rows in each row pattern partition is implementation-dependent (US029).

NOTE 41 — Failure to order row pattern partitions decreases the likelihood that queries involving row pattern matching will behave the same in different SQL-implementations, and possibly from query execution to query execution on a given SQL-implementation.

— Rows of the ordered or unordered, partitioned or not partitioned table are inspected for potential matches to the specified row pattern (<row pattern>). The specification does not define an automaton capable of analyzing and classifying each row sequentially and independently of other rows; nevertheless, many elementary applications can be understood as implemented by such an automaton (especially if the predicates that define adjacent row pattern variables are mutually exclusive), as follows:

- The inspection scans each row pattern partition to find a row that meets the criteria of the first element (<row pattern term>, <row pattern factor>, <row pattern primary>, as modified by <row pattern quantifier>)) of the row pattern; if that element requires or allows additional consecutive rows that match those criteria, those rows are identified by scanning further in the partition.
- When a sequence of rows have been matched to the first element of the row pattern, those rows are associated with a <row pattern variable name> that is specified as part of the row pattern. Those rows are called *pattern-matched rows* and the <row pattern variable name> is called the *associated row pattern variable name*.

  NOTE 42 — That <row pattern variable name> is used as a <correlation name> to qualify the names of the columns of those rows. See the Syntax Rules of Subclause 6.6, "<identifier chain>".
- When there are no more matches to the first element of the row pattern, or that first element has been fully satisfied, the scan continues using the next element of the row pattern.
- That process continues until the row pattern has been fully satisfied by row matches in the row pattern partition, in which case a row pattern match is identified.
- If there is no row pattern match beginning at a particular row of a row pattern partition, then a row pattern match beginning at the next row of the row pattern partition is sought. When a row pattern match is successfully identified, then scanning continues at a row specified by <row pattern skip to>.

— When all rows of all row pattern partitions have been inspected, the row pattern output table is generated. The associated row pattern variable names specified in the row pattern are (if any matching rows were found for the corresponding component of the row pattern) associated with sequences of matched rows. Rows of the row pattern output table are generated by specifying one or more row pattern measure columns (<row pattern measure definition>s) whose values are derived from the values of columns in the matched rows corresponding to one or more of the associated row pattern variable names.

Additional optional syntax (<row pattern rows per match>) specifies whether a single row is generated in the row pattern output table for all rows of the table corresponding to a row pattern match of the row pattern (ONE ROW PER MATCH), or a row is generated for each row that is matched by some component of the row pattern (ALL ROWS PER MATCH). In the former case, the sequence of matching rows is treated similarly to the collection of rows created by the <group by clause>; that is, only row pattern partitioning columns and row pattern measure columns are columns of the row pattern output table. In the latter case, all columns of the row pattern input table, as well as row pattern measure columns, are columns of the row pattern output table.

4.19.3 Row pattern matching illustrated

Suppose it is necessary to examine a table containing the identifier of a particular stock, the time at which trades of shares of that stock were performed, and the price at which such trades took place. The business

goal is to identify sequences of trades that involve a trade as some (unspecified) price, followed by one or more trades at declining prices, followed by one or more trades at increasing prices, the last one or more of which are greater than the price of the initial trade. The business requirement is to identify the time and price of the first trade in the sequence that matches the pattern, the time at which the decline in prices finished and prices started to rise along with the lowest price, and the time and price of the last trade matched by the pattern.

For example, that pattern describes a trading day during which the stock of XYZ Corp. began the day at a price of $100/share, then steadily fell throughout the morning until it hit a price of $50/share at noon, after which the price rose throughout the afternoon until it hit a peak of $150/share at 15:00 and then began to decline again. The information to be returned would be 10:00 (starting time), $100 (starting price), 12:00 (time of turnaround), $50 (lowest price), 15:00 (time at which the share price was no longer rising), and $150 (final price).

A row pattern matching query that solves that particular problem is illustrated here:

```
SELECT a_symbol,      /* stock symbol */
       a_tstamp,      /* start time */
       a_price,       /* start price */
       max_c_tstamp,  /* inflection time */
       last_c_price,  /* lowest price */
       max_f_tstamp,  /* end time */
       last_f_price,  /* end price */
       matchno
FROM stock_ticker
  MATCH_RECOGNIZE (
  PARTITION BY symbol
  ORDER BY tstamp
  MEASURES A.symbol        AS a_symbol,
           A.tstamp        AS a_tstamp,
           A.price         AS a_price,
           MAX(C.tstamp)   AS max_c_tstamp,
           LAST(C.price)   AS last_c_price,
           MAX(F.tstamp)   AS max_f_tstamp,
           LAST(F.price)   AS last_f_price,
           MATCH_NUMBER () AS matchno
  ONE ROW PER MATCH
  AFTER MATCH SKIP PAST LAST ROW
  PATTERN ( A B C* D E* F+ )
  DEFINE /* A is unspecified, defaults to TRUE, matches any row */
         B AS (B.price < PREV (B.price)),
         C AS (C.price <= PREV (C.price)),
         D AS (D.price > PREV (D.price)),
         E AS (E.price >= PREV (E.price)),
         F AS (F.price >= PREV (F.price)
           AND F.price > A.price)) AS T
```

Figure 3, “Illustration of important concepts in example query”, illustrates the concepts used in that query.

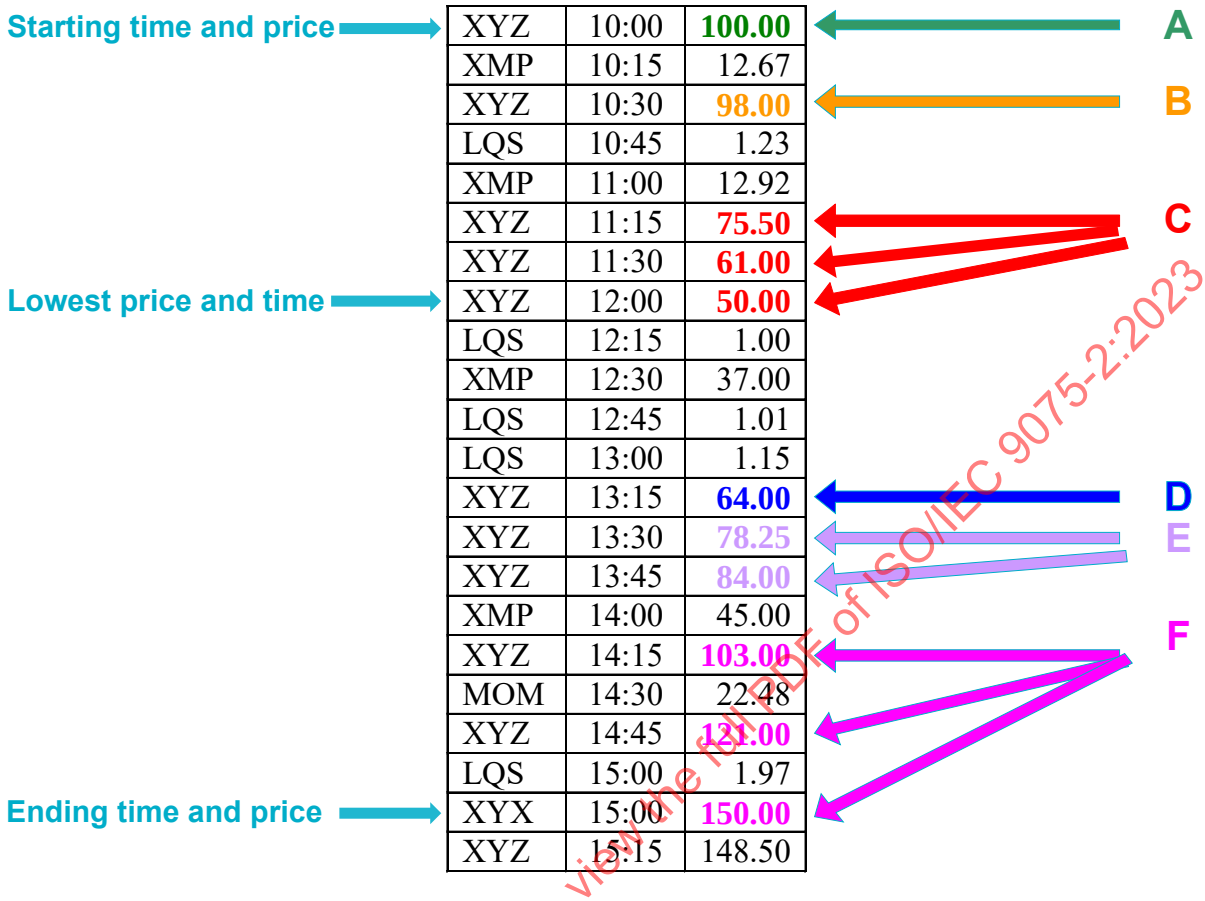

Figure 3 — Illustration of important concepts in example query

When evaluation of the <row pattern recognition clause> is complete for the row pattern partition containing the symbol "XYZ", a table having a single row of eight columns is returned. The single-row result is caused by the use of ONE ROW PER MATCH in the example query. The value of that row is shown in Table 8, "Result with ONE ROW PER MATCH".

Table 8 — Result with ONE ROW PER MATCH

a_symbol	a_tstamp	a_price	max_c_tstamp	last_c_price	max_f_tstamp	last_f_price	matchno
XYZ	10:00	100.00	12:00	50.00	15:00	150.00	1

Consider this variation of the sample query with changes highlighted by underscoring:

```
SELECT T.symbol,       /* row's symbol */
       T.tstamp,       /* row's timestamp */
       T.price,        /* row's price */
       T.classy,       /* row's classifier */
       T.matchno,      /* sequential row number within partition */
       T.a_tstamp,     /* start time */
       T.a_price,      /* start price */
       T.max_c_tstamp, /* inflection time */
       T.last_c_price, /* low price */
       T.max_f_tstamp, /* end time */
       T.last_f_price, /* end price */
       T.matchno
```

```
FROM ticker MATCH_RECOGNIZE (
              PARTITION BY symbol
              ORDER BY tstamp
              MEASURES A.symbol AS a_symbol,
                       A.tstamp        AS a_tstamp,
                       A.price         AS a_price,
                       MAX (C.tstamp)  AS max_c_tstamp,
                       LAST (C.price)  AS last_c_price,
                       MAX (F.tstamp)  AS max_f_tstamp,
                       LAST (F.price)  AS max_f_price,
                       MATCH_NUMBER () AS matchno,
                       CLASSIFIER ()   AS classy
              ALL ROWS PER MATCH
              AFTER MATCH SKIP PAST LAST ROW
              PATTERN (A B C* D E* F+)
              DEFINE /* A defaults to True, matches any row */
                B AS (B.price < PREV(B.price)),
                C AS (C.price <= PREV(C.price)),
                D AS (D.Price > PREV(D.price)),
                E AS (E.Price >= PREV(E.Price)),
                F AS (F.Price >= PREV(F.price)
                  AND F.price > A.price)) AS T
```

The result of that query is a table, shown in Table 9, "Result with ALL ROWS PER MATCH", having 11 rows created for the row pattern partition containing the symbol "XYZ".

Table 9 — Result with ALL ROWS PER MATCH

symbol	tstamp	price	classy	match-no	a_ tstamp	a_price	max_c_ tstamp	last_c_ price	max_f_ tstamp	last_f_ price
XYZ	10:00	100.00	A	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	10:30	98.00	B	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	11:15	75.50	C	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	11:30	61.00	C	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	12:00	50.00	C	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	13:15	64.00	D	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	13:30	78.25	E	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	13:45	84.00	E	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	14:15	103.00	F	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	14:45	121.00	F	1	10:00	100.00	12:00	50.00	15:00	150.00
XYZ	15:00	150.00	F	1	10:00	100.00	12:00	50.00	15:00	150.00

In the preceding table, the value of matchno in every row is 1 (one), because the collection of 11 rows collectively constitutes a single match of the pattern. If there were a subsequent match in the same row pattern partition, then the value of matchno in every row occasioned by that match would be 2. Matches in other row pattern partitions would be numbered starting from 1 (one) in each row pattern partition.

4.19.4 Row pattern partitioning

When a <row pattern recognition clause> contains a <row pattern partition by>, the rows of the row pattern input table are partitioned into one or more groups, or *row pattern partitions*. In each of those row pattern partitions, the values of the set of *row pattern partitioning columns* are not distinct in all rows of the partition (i.e., "equal" with an extension that considers two null values to be equal).

If no <row pattern partition by> is specified, then all rows of the row pattern input table are in a single row pattern partition.

NOTE 43 — The syntax and semantics of <row pattern partition by> are the same as the syntax and semantics of <window partition clause>.

4.19.5 Row ordering

When a <row pattern recognition clause> contains a <row pattern order by>, the rows of each row pattern partition are ordered by the values of the column references simply contained in the <row pattern order by>.

If no <row pattern order by> is specified, or if the ordering specified by <row pattern order by> does not determine the relative order of two or more rows in the row pattern partition, then the ordering of those rows is implementation-dependent (US029).

NOTE 44 — The syntax and semantics of <row pattern order by> are the same as the syntax and semantics of <order by clause>.

NOTE 45 — The syntax of <row pattern recognition clause> makes the <row pattern order by> optional. In practice, very few applications will omit <row pattern order by>.

4.19.6 Row pattern measure columns

When a <row pattern recognition clause> contains a <row pattern measures>, that <row pattern measures> defines "exported" columns that are specified by expressions over the <row pattern variable name>s defined in the <row pattern>. Those columns are known as *row pattern measure columns*.

4.19.7 Number of rows per match

The <row pattern recognition clause> supports row pattern matching that returns one row per match. When <row pattern rows per match> specifies ONE ROW PER MATCH (which is the default when <row pattern rows per match> is not specified), a single row is returned after completion of a match, including any empty matches. Columns of that row are referenced by column names that are specified by <row pattern partition by> and <row pattern measures>.

When <row pattern rows per match> specifies ALL ROWS PER MATCH, for each row identified by each row pattern match, a row is returned. Columns of that row are referenced by column names that are specified by <row pattern partition by> and <row pattern measures>, and by column names that identify columns of the row pattern input table.

With ALL ROWS PER MATCH, it is optional whether an empty row pattern match creates a row in the row pattern output table. If SHOW EMPTY MATCHES is specified, then each empty row pattern match generates one row in the row pattern output table. Columns of the row pattern output table that are derived from the row pattern input table (such as row pattern partitioning columns or ordering columns) show the values of the row of the row pattern input table at which the empty match occurred. If WITH UNMATCHED ROWS is specified, then each empty row pattern match generates one row in the row pattern output table (as if SHOW EMPTY MATCHES had been specified) and any row of the row pattern input table that is neither the starting row of an empty row pattern match, nor mapped by a non-empty row pattern match, generates one row in the row pattern output table in which each row pattern measure column is the null value. If OMIT EMPTY MATCHES is specified, then empty row pattern matches do not generate a row in the row pattern output table. The default is SHOW EMPTY MATCHES.

4.19.8 Skipping rows after matching

Once a row pattern match has been made, row pattern matching continues searching within a row pattern partition for additional matches (until every row pattern partition has been processed). After an empty match is detected, row pattern matching skips one row before looking for another match. Otherwise, row pattern matching can continue at any of four places within a row pattern partition, as specified in the optional <row pattern skip to> component of <row pattern recognition clause>:

— At the row immediately following the first row of the current match.

— At the row immediately following the last row of the current match.

— At the first row of the group of rows matched by a row pattern variable.

— At the last row of the group of rows matched by a row pattern variable.

If the skip is to the first row of the match, or if the skip is to a row pattern variable that did not match, then an exception condition is raised.

4.20 Row patterns

The heart of row pattern recognition is the row pattern itself. A *row pattern* is a form of regular expression in which the items for which matches are sought are not characters, sequences of characters, ranges of characters, and the like. Instead, the items for which matches are sought are expressed in terms of *primary row pattern variables*, each of which represents a Boolean condition (see Subclause 4.22, "Defining Boolean conditions") that may, for example, express a condition about a single row, the relationship between pairs of rows in the row pattern input table, or the relationship between sets of rows in the row pattern input table through the use of aggregates.

The regular expressions used in row pattern matching are very similar to those used in character string pattern matching. For example:

— Concatenation: indicated by the absence of any operator sign between two successive items in a pattern.

— Quantifiers: quantifiers are postfix operators with the following choices:

- `*` — zero (0) or more matches
- `+` — one (1) or more matches
- `?` — no match or one (1) match, optional
- `{ n }` — exactly *n* matches
- `{ n, }` — *n* or more matches
- `{ n, m }` — between *n* and *m* (inclusive) matches
- `{ , m }` — between zero (0) and *m* (inclusive) matches

— Reluctant quantifiers: indicated by an additional question mark (e.g., `*?`, `+?`, `??`, `{ n, m }?`). *Reluctant quantifiers* try to match as few rows as possible, whereas non-reluctant quantifiers (*greedy quantifiers*) try to match as many rows as possible. More precisely, with both reluctant and greedy quantifiers, the set of matches is ordered lexicographically, but when one match is an initial substring of another match, reluctant quantifiers prefer the shorter match (the substring), whereas greedy quantifiers prefer the longer match (the "superstring").

NOTE 46 — For example, given the pattern `(A|B){1,2}` with a greedy quantifier, the possible matches in order of decreasing preferment are:

```
A A
A B
```



```
A
B A
B B
B
```

Given the pattern `(A|B){1,2}?` with a reluctant quantifier, the possible matches in order of decreasing preferment are:

```
A
A A
A B
B
B A
B B
```

— `{- ... -}`: indicates that matching rows are to be excluded from the output. (This is useful only if ALL ROW PER MATCH is specified.)

— Alternation: indicated by a vertical bar " `|` ".

— Grouping: indicated by parentheses

— `^`: indicates the start of a row pattern partition

— `$`: indicates the end of a row pattern partition

The row pattern implicitly defines the primary row pattern variables, which are assigned meaning (Boolean conditions) in a subsequent clause of the <row pattern common syntax>.

4.21 Unions of row pattern variables

The optional <row pattern subset clause> component of <row pattern recognition clause> is used to define additional union row pattern variables that represent unions of the rows matched to the primary row pattern variables defined in the row pattern proper.

4.22 Defining Boolean conditions

The Boolean conditions used to define what rows are matched to specific parts of a row pattern are associated with primary row pattern variables. One or more primary row pattern variables are specified using syntax that requires the name of each primary row pattern variable and the Boolean condition associated with that primary row pattern variable. Each Boolean condition is expressed in terms of primary and union row pattern variables. The Boolean condition associated with a particular primary row pattern variable can reference the primary row pattern variable with which it is associated as well as other primary and union row pattern variables.

More precisely, each row pattern variable is used as a kind of qualifier (i.e., as a kind of <correlation name>) for the names of columns of the row pattern input table. The Boolean conditions are then defined as comparisons or other predicates of the values of columns in rows of the row pattern input table, the specific rows being used for the comparisons determined by the rows referenced by each row pattern variable.

NOTE 47 — Suppose a table whose name is `T` has three columns whose names are `C1`, `C2`, and `C3`, respectively. A Boolean condition associated with a row pattern variable whose name is `V1` can reference the values of those columns using a notation like this one: "`V1.C1`". Some other Boolean condition associated with a row pattern variable whose name is `V2` can reference the values of those columns with either "`V1.C1`" or "`V2.C1`". The values represented as "`V1.C1`" are those in rows matched by the Boolean condition associated with `V1` and those represented as "`V2.C1`" are those in rows matched by the Boolean condition associated with `V2`.

Row patterns may use the names of primary row pattern variables that are not defined in the <row pattern definition list>. Such primary row pattern variables always evaluate to *True* and thus will match any row.

Consider the following row pattern and Boolean condition definitions:

```
PATTERN ( A B* C+ D+ )
```

```
DEFINE
  A AS A.price > 100,
  B AS B.price > A.price,
  C AS C.price < AVG (B.price),
  D AS D.price > PREV(D.price)
```

The primary row pattern variable named A identifies a single row – which might be any row whose column named price has a value that is greater than 100. The primary row pattern variable named B identifies zero or more rows (which, in view of the row pattern, must be consecutive rows immediately following the single row identified by the primary row pattern variable named A), all of whose price column have values greater than the value of the price column associated with primary row pattern variable A.

Because B represents possibly many rows, the definition of C references aggregates of the columns of the rows identified by B. In this example, C will match all rows immediately following those identified by B that all have values of the price column that are less than the average of the values of the price column in the rows identified by B.

Finally, the definition of D illustrates how a primary row pattern variable can be defined in terms of itself. In this case, D represents all rows immediately following the last row represented by C whose price column's value is greater than the previous row's price column's value. (PREV is a function that is used to access the values of the previous row of the row pattern partition.)

4.23 Scalar expressions in row pattern matching

4.23.1 Introduction to scalars in row pattern matching

There are two contexts in which a scalar expression may appear in row pattern matching:

— In the <row pattern measure expression> that defines a row pattern measure.

— In a <row pattern definition search condition> that defines a primary row pattern variable.

In addition to the scalar expression syntax specified in Clause 6, "Scalar expressions", this document specifies three kinds of scalar expression that are only available in row pattern matching:

— <row pattern navigation operation>

— <classifier function>

— <match number function>

4.23.2 Running vs. final semantics

There are two semantics for scalar expressions within row pattern matching, known as *running semantics* and *final semantics*. These two semantics govern expression evaluation when a collection of rows is required, namely, aggregate functions and the row pattern navigation operations FIRST and LAST. Running semantics is the default, and may be indicated by the optional keyword RUNNING. Final semantics is indicated by the keyword FINAL.

Under running semantics, the collection of rows for the aggregation, FIRST, or LAST operation consists of the rows that are mapped to a specified row pattern variable, up to and including the current row. Under final semantics, the group of rows consists of all rows that are mapped to a specified row pattern variable. Final semantics are the same as running semantics when positioned on the last row of a row pattern match.

Running semantics is the only semantics supported in a <row pattern definition search condition>. As for measures, if row pattern matching is performed in a window, or if ONE ROW PER MATCH is specified, then expression evaluation is performed in the last row of the row pattern match, so that running and

final semantics are the same. If row pattern matching is performed using ALL ROWS PER MATCH, then there is a meaningful choice between running and final semantics for measures.

4.23.3 Row pattern navigation operations

<row pattern navigation operation> evaluates a <value expression> *VE* in a row *NR*, which may be different than the current row *CR*. The navigated row *NR* is specified by a physical and/or a logical offset from *CR*. The physical and logical offsets are non-negative integers.

VE must specify or imply exactly one row pattern variable *RPV*. The row pattern variable can be specified or implied by either the qualifier of column reference(s) or the argument of <classifier function>(s).

FIRST and LAST are row pattern navigation operations that navigate by a logical offset. The collection of rows that are mapped to *RPV* is identified (when using running semantics, only rows up to and including the current row are identified). FIRST locates a row within this collection by offsetting from the beginning of the collection in the row pattern ordering, whereas LAST offsets from the end of the collection.

PREV and NEXT are row pattern navigation operations that navigate by a physical offset. It is permitted to nest FIRST or LAST within PREV or NEXT. If such nesting is not specified, then RUNNING LAST is implicit.

Given such nesting, PREV and NEXT start at the row identified by the nested FIRST or LAST operation. PREV offsets from that identified row to a prior row in the partition according to the row pattern ordering, whereas NEXT offsets to a later row in the partition.

After identifying the navigated row *NR*, <row pattern navigation operation> evaluates *VE* as if *RPV* were bound to that row. The result is the null value if there is no such row.

4.23.4 Row pattern classifier function

<classifier function> is a function that returns a character string value equivalent to a primary row pattern variable name. <classifier function> has one optional argument, a row pattern variable. If omitted, the universal row pattern variable is the default.

NOTE 48 — For an example of <classifier function>, see the second example of a row pattern match query (the one using ALL ROWS PER MATCH) in Subclause 4.19.3, "Row pattern matching illustrated".

4.23.5 Row pattern match number function

<match number function> is a nullary function that returns an integer value that specifies the ordinal position of a row pattern match in a row pattern partition, according to the ordering of the starting rows of row pattern matches using the row pattern ordering. <match number function> is not available to row pattern matching in windows.

4.24 Determinism

In general, an operation is *deterministic* if that operation assuredly computes identical results when repeated with identical input values. The inputs of an operation are defined as follows:

— For an SQL-invoked routine, the values in the argument list are regarded as the input. Regarding polymorphic table functions, the following considerations apply:

  - For a polymorphic table function, the value of a table argument with row semantics is a multiset of rows; the value of a table argument with set semantics is a set of partitions of the table argument. A polymorphic table function is regarded as non-deterministic if the ordering of a partition of a table argument can determine the result of the polymorphic table function. SQL-data that is not passed as a table argument is not regarded as input to a polymorphic table function.

- For a PTF component procedure, the value of an argument that is a PTF extended name is the descriptor or the cursor that is identified by the PTF extended name. A PTF fulfill component procedure is regarded as non-deterministic if the ordering of a cursor can determine the result of the PTF fulfill component procedure. SQL-data that is not passed as a PTF extended cursor is not regarded as input to a PTF component procedure.

— Otherwise, the SQL-data and the set of privileges by which they are accessed is regarded as the input. Differences in the ordering of rows, as permitted by General Rules that specify implementation-defined behavior, are not regarded as significant to the question of determinism.

NOTE 49 — Transaction isolation levels have a significant impact on determinism, particularly transaction isolation levels other than SERIALIZABLE. However, this document does not address that impact, particularly because of the difficulty in clearly specifying that impact without appearing to mandate implementation techniques (such as row or page locking) and because different SQL-implementations almost certainly resolve the issue in significantly different ways.

Recognizing that an operation is deterministic is a difficult task, it is in general not mandated by this document. SQL-invoked routines are regarded as deterministic if the routine is declared to be DETERMINISTIC; that is, the SQL-implementation trusts the definer of the SQL-invoked routine to correctly declare that the routine is deterministic. For other operations, this document does not label an operation as deterministic; instead it identifies certain operations as "potential sources of non-determinism". The definition is found in Subclause 9.16, "Potential sources of non-determinism".

Certain <boolean value expression>s are identified as "retrospectively deterministic". A retrospectively deterministic <boolean value expression> has the property that if it is *True* at one point time, then it is *True* for all later points in time if re-evaluated for the identical SQL-data by an arbitrary user with the identical set of privileges. The precise definition is found in Subclause 6.46, "<boolean value expression>".

4.25 Integrity constraints

4.25.1 Overview of integrity constraints

Integrity constraints, generally referred to simply as constraints, define the valid states of SQL-data by constraining the values in the base tables. A constraint is described by a constraint descriptor. A constraint is either a table constraint, a domain constraint, or an assertion and is described by, respectively, a table constraint descriptor, a domain constraint descriptor, or an assertion descriptor. Every constraint descriptor includes:

— The name of the constraint.

— An applicable <search condition>.

NOTE 50 — The applicable <search condition> included in the descriptor is not necessarily the <search condition> that might be contained in the SQL-statement whose execution brings the constraint descriptor into existence. The General Rules for the SQL-statement in question specify the applicable <search condition> to be included in the constraint descriptor, in some cases deriving it from a given <search condition>. For example, the syntax for table constraints allows universal quantification over the rows of the table in question to be implicit; in the applicable <search condition> included in the descriptor, that universal quantification is made explicit, to allow for uniform treatment of all types of constraint.

— An indication of whether or not the constraint is deferrable.

— An indication of whether the initial constraint mode is *deferred* or *immediate*.

No integrity constraint shall be defined using a <search condition> that is not retrospectively deterministic.

4.25.2 Checking of constraints

Every constraint is either *deferrable* or *non-deferrable*. Within an SQL-transaction, every constraint has a constraint mode; if a constraint is non-deferrable, then its constraint mode is always *immediate*; otherwise, it is either immediate or *deferred*. Every constraint has an initial constraint mode that specifies the constraint mode for that constraint at the start of each SQL-transaction and immediately after definition

of that constraint. If a constraint is deferrable, then its constraint mode may be changed (from immediate to deferred, or from deferred to immediate) by execution of a <set constraints mode statement>.

Table constraints are either *enforced* or *not enforced*. Domain constraints and assertions are always enforced.

The checking of an enforced constraint depends on its constraint mode within the current SQL-transaction. Whenever an SQL-statement is executed, every enforced constraint whose mode is immediate is checked, at a certain point after any changes to SQL-data and schemas resulting from that execution have been effected, to see if it is satisfied. A constraint is *satisfied* if and only if the applicable <search condition> included in its descriptor evaluates to *True* or *Unknown*. If any enforced constraint is not satisfied, then any changes to SQL-data or schemas resulting from executing that statement are canceled. (See the General Rules of Subclause 9.17, "Executing an <SQL procedure statement>".)

NOTE 51 — This includes SQL-statements that are executed as a direct result or an indirect result of executing a different SQL-statement. It also includes statements whose effects explicitly or implicitly include setting the constraint mode to immediate.

The constraint mode can be set to immediate either explicitly by execution of a <set constraints mode statement>, or implicitly at the end of the current SQL-transaction.

When a <commit statement> is executed, all enforced constraints are effectively checked and, if any enforced constraint is not satisfied, then an exception condition is raised and the SQL-transaction is terminated by an implicit <rollback statement>.

In contrast to constraint checking, the referential actions specified by a referential constraint are never deferred.

4.25.3 Table constraints

4.25.3.1 Introduction to table constraints

A *table constraint* is a constraint whose descriptor is included in a base table descriptor. Being associated with a particular base table allows for convenient syntactic shorthands in which universal quantification over the rows of the table in question is implied.

A table constraint is either a unique constraint, a referential constraint or a table check constraint. A table constraint is described by a table constraint descriptor.

In addition to the components of every constraint descriptor, a table constraint descriptor includes:

— An indication of whether the constraint is enforced or not enforced.

NOTE 52 — A unique constraint is always enforced. A table check constraint or a referential constraint can either be enforced or not enforced.

A table constraint descriptor is either a unique constraint descriptor, a referential constraint descriptor, or a table check constraint descriptor, respectively.

Every table constraint specified for base table *T* is implicitly a constraint on every subtable of *T*, by virtue of the fact that every row in a subtable is considered to have a corresponding superrow in every one of its supertables.

4.25.3.2 Unique constraints

In addition to the components of every table constraint descriptor, a unique constraint descriptor includes:

— An indication of whether it was defined with PRIMARY KEY or UNIQUE.

— The names and positions of the *unique columns* specified in the <unique column list>.

— If <without overlap specification> is specified, then the name of the period specified.

— The implicit or explicit <unique null treatment>.

Let *T* be a table and let *R1* and *R2* be two rows of *T*. If there is some column for which the corresponding values from *R1* and *R2* are not equal, then *R1* and *R2* are *unique with nulls distinct*. If there is some column for which the corresponding values from *R1* and *R2* are distinct, then *R1* and *R2* are *unique with nulls not distinct*. If all column values of both *R1* and *R2* are the null value, then an SQL-implementation may regard *R1* and *R2* as unique with nulls not distinct; whether an SQL-implementation does this is implementation-defined (IA201).

NOTE 53 — The implementation-defined treatment of rows consisting entirely of null values allows a common implementation technique, whereby all-null rows are omitted from uniqueness checking data structures entirely. Such SQL-implementations can choose NULLS NOT DISTINCT as the implementation-defined default, yet allow all-null rows to pass the constraint, as they are not checked at all.

If the table descriptor for base table *T* includes a unique constraint descriptor indicating that the unique constraint was defined with PRIMARY KEY, then the columns of that unique constraint constitute the *primary key* of *T*. A table that has a primary key cannot have a proper supertable.

Let *T* be a base table and let *R1* and *R2* be two rows of *T*. Let *UC* be a unique constraint on *T*, and let *UNT* be its implicit or explicit <unique null treatment>. If *T* is a system-versioned table, let *R1* and *R2* be two current system rows of *T*. If *UC* includes a <without overlap specification> *WOS*, let *ATPN* be the <application time period name> contained in *WOS*; *R1* and *R2* must additionally be such that the *ATPN* period values of *R1* and *R2* overlap. Let *UR1* and *UR2* be rows containing only the columns specified by *UC* of *R1* and *R2* respectively.

UC is satisfied if and only if, for every such *UR1* and *UR2*, exactly one of the following is true:

— *UNT* is NULLS DISTINCT and *UR1* and *UR2* are unique with nulls distinct.

— *UNT* is NULLS NOT DISTINCT and *UR1* and *UR2* are unique with nulls not distinct.

In addition, if the unique constraint was defined with PRIMARY KEY, then it requires that none of the values in the specified column or columns be a null value.

4.25.3.3 Referential constraints

In addition to the components of every table constraint descriptor, a referential constraint descriptor includes:

— A list of the names of the *referencing columns* specified in the <referencing column list>.

— The *referenced table* specified in the <referenced table and columns>.

— A list of the names of the *referenced columns* specified in the <referenced table and columns>.

— If <referencing period specification> is specified, then the name of the period specified.

— The <match type> and the <referential triggered action>.

The choices for <match type> are MATCH SIMPLE, MATCH PARTIAL, and MATCH FULL; MATCH SIMPLE is the default. There is no semantic difference between these choices if there is only one referencing column (and, hence, only one referenced column). There is also no semantic difference if all referencing columns are not nullable. If there is more than one referencing column, at least one of which is nullable, and if no <referencing period specification> is specified, then the various <match type>s have the following semantics:

— MATCH SIMPLE: if at least one referencing column is null, then the row of the referencing table passes the constraint check. If all referencing columns are not null and the referenced table is not a system-versioned table, then the row passes the constraint check if and only if there is a row of the referenced table that matches all the referencing columns. If all referencing columns are not null and the referenced table is a system-versioned table, then the row passes the constraint check

if and only if there is a current system row of the referenced table that matches all the referencing columns.

— MATCH PARTIAL: if all referencing columns are null, then the row of the referencing table passes the constraint check. If at least one referencing columns is not null and the referenced table is not a system-versioned table, then the row passes the constraint check if and only if there is a row of the referenced table that matches all the non-null referencing columns. If at least one referencing columns is not null and the referenced table is a system-versioned table, then the row passes the constraint check if and only if there is a current system row of the referenced table that matches all the non-null referencing columns.

— MATCH FULL: if all referencing columns are null, then the row of the referencing table passes the constraint check. If all referencing columns are not null and the referenced table is not a system-versioned table, then the row passes the constraint check if and only if there is a row of the referenced table that matches all the referencing columns. If some referencing column is null and another referencing column is non-null, then the row of the referencing table violates the constraint check. If all referencing columns are not null and the referenced table is a system-versioned table, then the row passes the constraint check if and only if there is a current system row of the referenced table that matches all the referencing columns.

If there is more than one referencing column, at least one of which is nullable, and if <referencing period specification> is specified, then let *CATPN* be the <application time period name> contained in <referencing period specification> and let *PATPN* be the <application time period name> contained in the <referenced period specification>; the various <match type>s have the following semantics:

— MATCH SIMPLE: if at least one referencing column is null, then the row of the referencing table passes the constraint check. If all referencing columns are not null and the referenced table is not a system-versioned table, then a row *R* of the referencing table passes the constraint check if and only if there is a non-empty set *S* of rows of the referenced table such that every row in *S* matches all the referencing columns of *R* and the *CATPN* period value of *R* is a subset of the union of the *PATPN* period values of the rows in *S*. If all referencing columns are not null and the referenced table is a system-versioned table, then a row *R* of the referencing table passes the constraint check if and only if there is a non-empty set *S* of current system rows of the referenced table such that every row in *S* matches all the referencing columns of *R* and the *CATPN* period value of *R* is a subset of the union of the *PATPN* period values of the rows in *S*.

— MATCH PARTIAL: if all referencing columns are null, then the row of the referencing table passes the constraint check. If at least one referencing column is not null and the referenced table is not a system-versioned table, then a row *R* of the referencing table passes the constraint check if and only if there is a non-empty set *S* of rows of the referenced table such that every row in *S* matches all the non-null referencing columns of *R* and the *CATPN* period value of *R* is a subset of the union of the *PATPN* period values of the rows in *S*. If at least one referencing columns is not null and the referenced table is a system-versioned table, then a row *R* of the referencing table passes the constraint check if and only if there is a non-empty set *S* of current system rows of the referenced table such that every row in *S* matches all the non-null referencing columns of *R* and the *CATPN* period value of *R* is a subset of the union of the *PATPN* period values of the rows in *S*.

— MATCH FULL: if all referencing columns are null, then the row of the referencing table passes the constraint check. If all referencing columns are not null and the referenced table is not a system-versioned table, then a row *R* of the referencing table passes the constraint check if and only if there is a non-empty set *S* of rows of the referenced table such that every row in *S* matches all the referencing columns of *R* and the *CATPN* period value of *R* is a subset of the union of the *PATPN* period values of the rows in *S*. If all referencing columns are not null and the referenced table is a system-versioned table, then a row *R* of the referencing table passes the constraint check if and only if there is a non-empty set *S* of current system rows of the referenced table such that every row in *S* matches all the referencing columns of *R* and the *CATPN* period value of *R* is a subset of the union of the

PATPN period values of the rows in *S*. If some referencing column is null and another referencing column is non-null, then the row of the referencing table violates the constraint check.

NOTE 54 — In the case that <referencing period specification> is specified, even though for a given row *R* in the referencing table there can be more than one corresponding row in the referenced table, there is at most one corresponding row in the referenced table at any given point in time in the *CATPN* period value of *R*.

The ordering of the lists of referencing column names and referenced column names is implementation-defined (IS001), but shall be such that corresponding column names occupy corresponding positions in each list.

In the case that a table constraint is a referential constraint, the table is referred to as the *referencing table*. The *referenced columns* of a referential constraint shall be the *unique columns* of some unique constraint of the *referenced table*. <referencing period specification> is specified, then the unique constraint that defines the unique columns of the referenced table shall specify <without overlap specification>.

The referencing table may be the same table as the referenced table.

An <update rule> that does not contain NO ACTION specifies a *referential update action*. A <delete rule> that does not specify NO ACTION specifies a *referential delete action*. Referential update actions and referential delete actions are collectively called *referential actions*. Referential actions are carried out before, and are not part of, the checking of a referential constraint. Deferring a referential constraint defers the checking of the <search condition> of the constraint (a <match predicate>) but does not defer the referential actions of the referential constraint.

NOTE 55 — For example, if a referential update action such as ON UPDATE CASCADE is specified, then any UPDATE operation on the referenced table will be cascaded to the referencing table as part of the UPDATE operation, even if the referential constraint is deferred. Consequently, the referential constraint cannot become violated by the UPDATE statement. On the other hand, ON UPDATE SET DEFAULT could result in a violation of the referential constraint if there is no matching row after the referencing column is set to its default value. In addition, INSERT and UPDATE operations on the referencing table do not entail any automatic enforcement of the referential constraint. Any such violations of the constraint will be detected when the referential constraint is eventually checked, at or before a commit.

The options for <update rule> are:

— ON UPDATE CASCADE: any change to a referenced column in the referenced table causes the same change to the corresponding referencing column in matching rows of the referencing table.

— ON UPDATE SET NULL: any change to a referenced column in the referenced table causes the corresponding referencing column in matching rows of the referencing table to be set to null.

— ON UPDATE SET DEFAULT: any change to a referenced column in the referenced table causes the corresponding referencing column in matching rows of the referencing table to be set to its default value.

— ON UPDATE RESTRICT: any change to a referenced column in the referenced table is prohibited if there is a matching row.

— ON UPDATE NO ACTION (the default): there is no referential update action; the referential constraint only specifies a constraint check.

NOTE 56 — Even if constraint checking is not deferred, ON UPDATE RESTRICT is a stricter condition than ON UPDATE NO ACTION. ON UPDATE RESTRICT prohibits an update to a particular row if there are any matching rows; ON UPDATE NO ACTION does not perform its constraint check until the entire set of rows to be updated has been processed.

The options for <delete rule> are:

— ON DELETE CASCADE: if a row of the referenced table is deleted, then all matching rows in the referencing table are deleted.

— ON DELETE SET NULL: if a row of the referenced table is deleted, then all referencing columns in all matching rows of the referencing table to be set to null.

— ON DELETE SET DEFAULT: if a row of the referenced table is deleted, then all referencing columns in all matching rows of the referencing table to be set to the column's default value.

— ON DELETE RESTRICT: it is prohibited to delete a row of the referenced table if that row has any matching rows in the referencing table.

— ON DELETE NO ACTION (the default): there is no referential delete action; the referential constraint only specifies a constraint check.

NOTE 57 — Even if constraint checking is not deferred, ON DELETE RESTRICT is a stricter condition than ON DELETE NO ACTION. ON DELETE RESTRICT prohibits a delete of a particular row if there are any matching rows; ON DELETE NO ACTION does not perform its constraint check until the entire set of rows to be deleted has been processed.

4.25.3.4 Table check constraints

Let *T* be a base table that is not a system-versioned table. A table check constraint on *T* is satisfied if and only if the specified <search condition> evaluates to *True* or *Unknown* for every row of *T*.

Let *T* be a system-versioned table. A table check constraint on *T* is satisfied if and only if the specified <search condition> evaluates to *True* or *Unknown* for every current system row of *T*.

NOTE 58 — Consequently, an empty table satisfies every table check constraint that applies to it.

4.25.4 Domain constraints

A domain constraint is a constraint that is specified for a domain. It is applied to all columns that are based on that domain, and to all values cast to that domain.

A domain constraint is described by a domain constraint descriptor. In addition to the components of every constraint descriptor, a domain constraint descriptor includes:

— The template <search condition> for the generation of domain constraint usage <search condition>s.

— A possibly empty set of domain constraint usages.

A domain constraint usage descriptor is created implicitly by the evaluation of a <column definition> whose <data type or domain name> is a <domain name>. If *C* is such a column and *D* is the domain identified by the <domain name>, then every domain constraint *DC* defined for *D* implies a domain constraint usage, to the effect that each value in *C* satisfies *DC*.

In addition to the components of every table constraint descriptor, a domain constraint usage descriptor includes:

— The name of the applicable column.

— The applicable <search condition> that evaluates whether each value in *C* satisfies *DC*. This <search condition> is constructed from the domain constraint template <search condition>, with instances of the <general value specification> VALUE replaced by *C*.

A domain constraint is satisfied by SQL-data if and only if, for every table *T* that has a column named *C* based on that domain, the applicable <search condition> recorded in the appropriate domain constraint usage evaluates to *True* or *Unknown*.

A domain constraint is satisfied by the result of a <cast specification> if and only if the specified template <search condition>, with each occurrence of the <general value specification> VALUE replaced by that result, evaluates to *True* or *Unknown*.

4.25.5 Assertions

An assertion is a constraint whose descriptor is an independent schema component not included in any table descriptor.

4.26 Functional dependencies

This Subclause is modified by Subclause 4.5, "Functional dependencies", in ISO/IEC 9075-9.

4.26.1 Overview of functional dependency rules and notations

This Subclause defines the term *functional dependency* and specifies a minimal set of rules that a conforming SQL-implementation shall follow to determine functional dependencies and candidate keys in tables.

The rules in this Subclause may be freely augmented by implementation-defined (IE015) rules, where indicated in this Subclause.

Let *T* be any table. Let *CT* be the set comprising all the columns of *T*, and let *A* and *B* be arbitrary subsets of *CT*, not necessarily disjoint and possibly empty.

Let "*T*: *A* ↦ *B*" (read "in *T*, *A* determines *B*" or "*B* is functionally dependent on *A* in *T*") denote the functional dependency of *B* on *A* in *T*, which is true if, for any possible value of *T*, any two rows that are not distinct for every column in *A* also are not distinct for every column in *B*. When the table *T* is understood from context, the abbreviation "*A* ↦ *B*" may also be used.

If *X* ↦ *Y* is some functional dependency in some table *T*, then *X* is a *determinant* of *Y* in *T*.

Let *A* ↦ *B* and *C* ↦ *D* be any two functional dependencies in *T*. The following are also functional dependencies in *T*:

— *A* UNION (*C* DIFFERENCE *B*) ↦ *B* UNION *D*

— *C* UNION (*A* DIFFERENCE *D*) ↦ *B* UNION *D*

NOTE 59 — Here, "UNION" denotes set union and "DIFFERENCE" denotes set difference.

These two rules are called the *rules of deduction* for functional dependencies.

Every table has an associated non-empty set of functional dependencies.

The set of functional dependencies is non-empty because *X* ↦ *X* for any *X*. A functional dependency of this form is an axiomatic functional dependency, as is *X* ↦ *Y* where *Y* is a subset of *X*. *X* ↦ *Y* is a non-axiomatic functional dependency if *Y* is not a subset of *X*.

4.26.2 General rules and definitions

In the following Subclauses, let a column *C1* be a *counterpart* of a column *C2* under qualifying table *QT* if *C1* is specified by a column reference (or by a <value expression> that is a column reference) that references *C2* and *QT* is the qualifying table of *C2*. If *C1* is a counterpart of *C2* under qualifying table *QT1* and *C2* is a counterpart of *C3* under qualifying table *QT2*, then *C1* is a counterpart of *C3* under *QT2*.

The notion of counterparts naturally generalizes to sets of columns, as follows: If *S1* and *S2* are sets of columns, and there is a one-to-one correspondence between *S1* and *S2* such that each element of *S1* is a counterpart of the corresponding element of *S2*, then *S1* is a counterpart of *S2*.

The next Subclauses recursively define the notion of *known functional dependency*. This is a ternary relationship between a table and two sets of columns of that table. This relationship expresses that a functional dependency in the table is known to the SQL-implementation. All axiomatic functional dependencies are known functional dependencies. In addition, any functional dependency that can be deduced from known functional dependencies using the rules of deduction for functional dependency is a known functional dependency.

The next Subclauses also recursively define the notion of a "*BUC-set*", which is a set of columns of a table (as in "*S* is BUC-set", where *S* is a set of columns).

NOTE 60 — "BUC" is an acronym for "base table unique constraint", since the starting point of the recursion is a set of known not null columns comprising a non-deferrable unique constraint of a base table.

The notion of BUC-set is closed under the following deduction rule for BUC-sets: If *S1* and *S2* are sets of columns, *S1* is a subset of *S2*, *S1* ↦ *S2*, and *S2* is a BUC-set, then *S1* is also a BUC-set.

NOTE 61 — If a BUC-set is empty, there is at most one row in the table. This case is distinguished from a table with no BUC-set.

An SQL-implementation may define additional rules for determining BUC-sets, provided that every BUC-set *S* of columns of a table *T* shall have an associated base table *BT* such that every column of *S* has a counterpart in *BT*, and for any possible value of the columns of *S*, there is at most one row in *BT* having those values in those columns.

The next Subclauses also recursively define the notion of a "*BPK-set*", which is a set of columns of a table (as in "*S* is a BPK-set", where *S* is a set of columns). Every BPK-set is a BUC-set.

NOTE 62 — "BPK" is an acronym for "base table primary key", since the starting point of the recursion is a set of known not null columns comprising a non-deferrable primary key constraint of a base table.

The notion of BPK-set is closed under the following deduction rule for BPK-sets: If *S1* and *S2* are sets of columns, *S1* is a subset of *S2*, *S1* ↦ *S2*, and *S2* is a BPK-set, then *S1* is also a BPK-set.

NOTE 63 — Like BUC-sets, empty BPK-sets are possible.

An SQL-implementation may define additional rules for determining BPK-sets, provided that every BPK-set *S* is a BUC-set, and every member of *S* has a counterpart to a column in a primary key in the associated base table *BT*.

All applicable syntactic transformations (for example, to remove *, CUBE, or ROLLUP) shall be applied before using the rules to determine known functional dependencies, BUC-sets, and BPK-sets.

The following Subclauses use the notion of *AND-component* of a <search condition> *SC*, which is defined recursively as follows:

— If *SC* is a <boolean test> *BT*, then the only AND-component of *SC* is *BT*.

— If *SC* is a <boolean factor> *BF*, then the only AND-component of *SC* is *BF*.

— If *SC* is a <boolean term> of the form "*P* AND *Q*", then the AND-components of *SC* are the AND-components of *P* and the AND-components of *Q*.

— If *SC* is a <boolean value expression> *BVE* that specifies OR, then the only AND-component of *SC* is *BVE*.

Let *AC* be an AND-component of *SC* such that *AC* is a <comparison predicate> whose <comp op> is <equals operator>. Let *RVE1* and *RVE2* be the two <row value predicand>s that are the operands of *AC*. Suppose that both *RVE1* and *RVE2* are <row value constructor predicand>s. Let *n* be the degree of *RVE1*. Let $RVEC1_i$ and $RVEC2_i$, 1 (one) ≤ *i* ≤ *n*, be the *i*-th <common value expression>, <boolean predicand>, or <row value constructor element> of *RVE1* and *RVE2*, respectively. The <comparison predicate> "$RVEC1_i = RVEC2_i$" is called an *equality AND-component* of *SC*.

4.26.3 Known functional dependencies in a base table

Let *T* be a base table and let *CT* be the set comprising all the columns of *T*.

A set of columns *S1* of *T* is a *BPK-set* if it is the set of columns enumerated in some unique constraint *UC* of *T*, *UC* specifies PRIMARY KEY, and *UC* is non-deferrable.

A set of columns *S1* of *T* is a *BUC-set* if it is the set of columns enumerated in some unique constraint *UC* of *T*, *UC* is non-deferrable, and every member of *S1* is known not null.

If *UCL* is a set of columns of *T* such that *UCL* is a BUC-set, then *UCL* ↦ *CT* is a *known functional dependency* in *T*.

If *GC* is a generated column of *T*, then *D* ↦ *GC*, where *D* is the set of parametric columns of *GC*, is a *known functional dependency* in *T*.

Implementation-defined (IE015) rules may determine other known functional dependencies in *T*.

4.26.4 Known functional dependencies in a viewed table

Let *V* be a viewed table, and let *CT* be the set comprising all the columns of *V*.

Case:

— If *V* is a referenceable view, then let *SRC* be the self-referencing column of *V*. *V* has no BPK-set, the only BUC-set of *V* is {*SRC*}, and {*SRC*} ↦ *CT* is a known functional dependency in *V*.

NOTE 64 — This known functional dependency will remain valid even if subtables of *V* are created.

— Otherwise, let *QE* be the original <query expression> of *V*. Every counterpart in *V* of a BPK-set of *QE* is a BPK-set of *V*. Every counterpart in *V* of a BUC-set of *QE* is a BUC-set of *V*. If *S1* ↦ *S2* is a known functional dependency of *QE*, *CS1* is the counterpart of *S1* in *V* and *CS2* is the counterpart of *S2* in *V*, then *CS1* ↦ *CS2* is a known functional dependency of *V*.

4.26.5 Known functional dependencies in a transition table

Let *TT* be the transition table defined in an AFTER trigger *TR* and let *T* be the subject table of *TR*. The BPK-sets, BUC-sets, and known functional dependencies of *TT* are the same as the BPK-sets, BUC-sets, and known functional dependencies of *T*.

4.26.6 Known functional dependencies in <table value constructor>

Let *R* be the result of a <table value constructor>, and let *CR* be the set comprising all the columns of *R*.

No set of columns of *R* is a BPK-set or a BUC-set, except as determined by implementation-defined (IA203) rules.

All axiomatic functional dependencies are *known functional dependencies* of a <table value constructor>. In addition, there may be implementation-defined (IE015) known functional dependencies (for example, by examining the actual value of the <table value constructor>).

4.26.7 Known functional dependencies in a <joined table>

Let *T1* and *T2* denote the tables identified by the first and second <table reference>s of some <joined table> *JT*. Let *R* denote the table that is the result of *JT*. Let *CT* be the set of columns of the result of *JT*.

Every column of *R* has some counterpart in either *T1* or *T2*. If NATURAL is specified or the <join specification> is a <named columns join>, then some columns of *R* may have counterparts in both *T1* and *T2*.

A set of columns *S* of *R* is a *BPK-set* if *S* has some counterpart in *T1* or *T2* that is a BPK-set, every member of *S* is known not null, and *S* ↦ *CT* is a *known functional dependency* of *R*.

A set of columns *S* of *R* is a *BUC-set* if *S* has some counterpart in *T1* or *T2* that is a BUC-set, every member of *S* is known not null, and *S* ↦ *CT* is a *known functional dependency* of *R*.

NOTE 65 — The following rules for known functional dependencies in a <joined table> are not mutually exclusive. The set of known functional dependencies is the union of those dependencies generated by all applicable rules, including the rules of deduction presented earlier.

If *A* ↦ *B* is a known functional dependency in *T1*, *CA* is the counterpart of *A* in *R*, and *CB* is the counterpart of *B* in *R*, then *CA* ↦ *CB* is a known functional dependency in *R* if and only if exactly one of the following is true:

— CROSS, INNER, or LEFT is specified.

— RIGHT or FULL is specified and at least one of the following is true:

  - At least one column in *A* is known not nullable and not a join partitioning column.
  - All columns in *A* and *B* are join partitioning columns.

If *A* ↦ *B* is a known functional dependency in *T2*, *CA* is the counterpart of *A* in *R*, and *CB* is the counterpart of *B* in *R*, then *CA* ↦ *CB* is a known functional dependency in *R* when exactly one of the following is true:

— CROSS, INNER, or RIGHT is specified.

— LEFT or FULL is specified and at least one of the following is true:

  - At least one column in *A* is known not nullable and not a join partitioning column.
  - All columns in *A* and *B* are join partitioning columns.

If INNER is specified, *AP* is an equality AND-component of the <search condition>, one comparand of *AP* is a column reference *CR*, and the other comparand of *AP* is a <literal>, then let *CRC* be the counterparts of *CR* in *R*. Let {} denote the empty set. {} ↦ {*CRC*} is a *known functional dependency* in *R*.

NOTE 66 — An SQL-implementation can also choose to recognize {} -> {*CRC*} as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If INNER is specified, *AP* is an equality AND-component of the <search condition>, one comparand of *AP* is a column reference *CRA*, and the other comparand of *AP* is a column reference *CRB*, then let *CRAC* and *CRBC* be the counterparts of *CRA* and *CRB* in *R*. {*CRAC*} ↦ {*CRBC*} is a *known functional dependency* in *R*.

NOTE 67 — An SQL-implementation can also choose to recognize similar known functional dependencies of the form { CRA_1, ..., CRA_N } ↦ {*CRBC*} in case one comparand is a deterministic expression of column references CRA_1, ..., CRA_N under similar conditions.

If LEFT is specified, <join condition> is specified, and the <search condition> *SC* is deterministic, then let *DJA* be the set of <column reference>s contained in *SC* that reference columns of *T1* and let *PCB* be the set of join partitioning columns of *TRB*, if any. If *AP* is an equality AND-component of *SC*, one comparand of *AP* is a <literal> or a column reference to a column of *T1*, and the other comparand of *AP* is a column reference *CRB* to a column of *T2*, then let *CRBC* be the counterpart of *CRB* in *R*. *DJA* ∪ *PCB* ↦ {*CRBC*} is a known functional dependency in *R*.

If RIGHT is specified, <join condition> is specified, and the <search condition> *SC* is deterministic, then let *DJB* be the set of <column reference>s contained in *SC* that reference columns of *T2* and let *PCA* be the set of join partitioning columns of *TRA*, if any. If *AP* is an equality AND-component of *SC*, one comparand of *AP* is a <literal> or a column reference to a column of *T2*, and the other comparand of *AP* is a column reference *CRA* to a column of *T1*, then let *CRAC* be the counterpart of *CRA* in *R*. *DJB* ∪ *PCA* ↦ {*CRAC*} is a known functional dependency in *R*.

If NATURAL is specified, or if a <join specification> immediately containing a <named columns join> is specified, then let C_1, ..., C_N be the column names of the corresponding join columns, for i between 1 (one) and *N*. Let *SC* be the <search condition>:

```
( TN1.C1 = TN2.C1 )
AND
...
AND
( TN1.CN = TN2.CN )
```

Let *SLCC* and *SL* be the <select list>s defined in the Syntax Rules of Subclause 7.10, "<joined table>". Let *JT* be the <join type>. Let *TN1* and *TN2* be the exposed <table or query name> or <correlation name> of tables *T1* and *T2*, respectively. Let *IR* be the result of the <query expression>:

```
SELECT SLCC, TN1.*, TN2.*
FROM TN1 JT JOIN TN2
     ON SC
```

The following are recognized as additional *known functional dependencies* of *IR*:

— If INNER or LEFT is specified, then { COALESCE ($TN1.C_i$, $TN2.C_i$) } ↦ { $TN1.C_i$ }, for all *i* between 1 (one) and *N*.

— If INNER or RIGHT is specified, then { COALESCE ($TN1.C_i$, $TN2.C_i$) } ↦ { $TN2.C_i$ }, for all *i* between 1 (one) and *N*.

The *known functional dependencies* of *R* are the known functional dependencies of:

```
SELECT SL FROM IR
```

4.26.8 Known functional dependencies in a <table primary>

Let *R* be the result of some <table primary> *TP*. The BPK-sets, BUC-sets, and known functional dependencies of *R* are determined as follows.

Case:

— If *TP* simply contains a <row pattern recognition clause> *RPRC* that specifies ONE ROW PER MATCH, then *R* has no non-axiomatic known functional dependencies, no BPK-sets, and no BUC-sets.

— If *TP* simply contains a <row pattern recognition clause> *RPRC* that specifies ALL ROWS PER MATCH, then let *TP2* be the <table primary> obtained by replacing the <row pattern recognition clause> by a <correlation name>. The rules of this Subclause are applied recursively to determine the known functional dependencies of the result *R2* of *TP2*, (which is the row pattern input recognition table). If *A* ↦ *B* is a known functional dependency of *R2*, and *AC* and *BC* are columns of *R* that are counterparts of *A* and *B*, then *AC* ↦ *BC* is a known functional dependency of *R*. *R* has no BPK-sets and no BUC-sets.

— If *TP* immediately contains a <table or query name> *TQN* or an <only spec> that immediately contains a <table or query name> *TQN*, then the counterparts of the BPK-sets and BUC-sets of *TQN* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* ↦ *B* is a known functional dependency in the result of *TQN*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* ↦ *BC* is a *known functional dependency* in *R*.

— If *TP* immediately contains a <derived table> or <lateral derived table> *DT*, then the counterparts of the BPK-sets and BUC-sets of *DT* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* ↦ *B* is a known functional dependency in the result of *DT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* ↦ *BC* is a *known functional dependency* in *R*.

— If *TP* immediately contains a <collection derived table> *CDT*, and WITH ORDINALITY is specified, then let *OC* be the column name of the ordinality column and let *CT* be the set comprising all of the column names of the columns of *CDT*. {*OC*} is a BPK-set and a BUC-set, and {*OC*} ↦ *CT* is a *known functional dependency*. If WITH ORDINALITY is not specified, then these rules do not identify any BPK-set, BUC-set, or non-axiomatic known functional dependency.

— If *TP* immediately contains a <table function derived table> or a <PTF derived table>, then these rules do not identify any BPK-set, BUC-set, or non-axiomatic known functional dependency.

— If *TP* immediately contains a <data change delta table>, then let *TT* be the <target table> or <insertion target> of the <data change statement> simply contained in *TP*. The counterparts of the BPK-sets and BUC-sets of *TT* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* ↦ *B* is a known functional dependency in the result of *TT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* ↦ *BC* is a *known functional dependency* in *R*.

— If *TP* immediately contains a <parenthesized joined table>, then let *JT* be the <joined table> simply contained in *TP*. The counterparts of the BPK-sets and BUC-sets of *JT* are the BPK-sets and BUC-sets, respectively, of *R*. If *A* ↦ *B* is a known functional dependency in the result of *JT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then *AC* ↦ *BC* is a *known functional dependency* in *R*.

4.26.9 Known functional dependencies in a <table factor>

Let *R* be the result of <table factor> *TF*. Let *S* be the result of <table primary> immediately contained in *TF*. The counterparts of the BPK-sets and BUC-sets of *S* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in *S*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

4.26.10 Known functional dependencies in a <table reference>

Let *R* be the result of some <table reference> *TR*. The BPK-sets, BUC-sets, and functional dependencies of *R* are determined as follows.

Case:

— If *TR* immediately contains a <table factor> *TF*, then the counterparts of the BPK-sets and BUC-sets of *TF* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *TF*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

— If *TR* immediately contains a <joined table> *JT*, then the counterparts of the BPK-sets and BUC-sets of *JT* are the BPK-sets and BUC-sets, respectively, of *R*. If $A \mapsto B$ is a functional dependency in the result of *JT*, and *AC* and *BC* are the counterparts of *A* and *B*, respectively, then $AC \mapsto BC$ is a *known functional dependency* in *R*.

4.26.11 Known functional dependencies in the result of a <from clause>

Let *R* be the result of some <from clause> *FC*.

If there is only one <table reference> *TR* in *FC*, then the counterparts of the BPK-sets of *TR* and the counterparts of the BUC-sets of *TR* are the BPK-sets and BUC-sets of *FC*, respectively. Otherwise, these rules do not identify any BPK-sets or BUC-sets in the result of *FC*.

If *T* is a <table reference> immediately contained in the <table reference list> of *FC*, then all known functional dependencies in *T* are *known functional dependencies* in *R*.

4.26.12 Known functional dependencies in the result of a <where clause>

Let *T* be the table that is the operand of the <where clause>. Let *R* be the result of the <where clause>. A set of columns *S* in *R* is a *BUC-set* if there is a <table reference> *TR* such that every member of *S* has a counterpart in *TR*, the counterpart of *S* in *TR* is a BUC-set, and $S \mapsto CR$, where *CR* is the set of all columns of *R*. If, in addition, the counterpart of *S* is a BPK-set, then *S* is a *BPK-set*.

If $A \mapsto B$ is a known functional dependency in *T*, then let *AC* be the set of columns of *R* whose counterparts are in *A*, and let *BC* be the set of columns of *R* whose counterparts are in *B*. $AC \mapsto BC$ is a *known functional dependency* in *R*.

If *AP* is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of *AP* is a column reference *CR*, and the other comparand of *AP* is a <literal>, then let *CRC* be the counterpart of *CR* in *R*. $\{\} \mapsto \{CRC\}$ is a *known functional dependency* in R, where {} denotes the empty set.

NOTE 68 — An SQL-implementation can also choose to recognize $\{\} \mapsto \{CRC\}$ as a known functional dependency if the other comparand is a deterministic expression containing no column references.

If *AP* is an equality AND-component of the <search condition> simply contained in the <where clause> and one comparand of *AP* is a column reference *CRA*, and the other comparand of *AP* is a column references *CRB*, then let *CRAC* and *CRBC* be the counterparts of *CRA* and *CRB* in *R*. $\{CRBC\} \mapsto \{CRAC\}$ and $\{CRAC\} \mapsto \{CRBC\}$ are *known functional dependencies* in *R*.

NOTE 69 — An SQL-implementation can also choose to recognize known functional dependencies of the form {$CRAC_1$, ..., $CRAC_N$} ↦ {*CRBC*} if one comparand is a deterministic expressions that contains column references CRA_1, ..., CRA_N and the other comparand is a column reference *CRB*.

4.26.13 Known functional dependencies in the result of a <group by clause>

Let *T1* be the table that is the operand of the <group by clause>, and let *R* be the result of the <group by clause>.

Let *G* be the set of columns specified by the <grouping column reference list> of the <group by clause>, after applying all syntactic transformations to eliminate ROLLUP, CUBE, and GROUPING SETS.

The columns of *R* are the columns of *G*, with an additional column *CI*, whose value in any particular row of *R* somehow denotes the subset of rows of *T1* that is associated with the combined value of the columns of *G* in that row.

If every element of *G* is a column reference to a known not null column, then *G* is a *BUC-set* of *R*. If *G* is a subset of a BPK-set of columns of *T1*, then *G* is a *BPK-set* of *R*.

G ↦ *CI* is a *known functional dependency* in *R*.

NOTE 70 — Any <set function specification> that is specified in conjunction with *R* is necessarily a function of *CI*. If *SFVC* denotes the column containing the results of such a <set function specification>, then *CI* ↦ *SFVC* holds true, and it follows that *G* ↦ *SFVC* is a *known functional dependency* in the table containing *SFVC*.

4.26.14 Known functional dependencies in the result of a <having clause>

Let *T1* be the table that is the operand of the <having clause>, let *SC* be the <search condition> simply contained in the <having clause>, and let *R* be the result of the <having clause>.

If *S* is a set of columns of *R* and the counterpart of *S* in *T1* is a BPK-set, then *S* is a *BPK-set*. If the counterpart of *S* in *T1* is a BUC-set, then *S* is a *BUC-set*.

Any known functional dependency in the <query expression>

```
SELECT * FROM T1 WHERE SC
```

is a *known functional dependency* in *R*.

4.26.15 Known functional dependencies in a <query specification>

Let *T* be the <table expression> simply contained in the <query specification> *QS* and let *R* be the result of the <query specification>.

Let *SL* be the <select list> of the <query specification>.

Let *T1* be *T* extended to the right with columns arising from <value expression>s contained in the <select list>, as follows: A <value expression> *VE* that is not a column reference specifies a computed column *CC* in *T1*. For every row in *T1*, the value in *CC* is the result of *VE*.

Let *S* be a set of columns of *R* such that every element of *S* arises from the use of <asterisk> in *SL* or by the specification of a column reference as a <value expression> simply contained in *SL*. *S* has counterparts in *T* and *T1*. If the counterpart of *S* in *T* is a BPK-set, then *S* is a *BPK-set*. If the counterpart of *S* in *T* is a BUC-set or a BPK-set, then *S* is a *BUC-set*.

If *A* ↦ *B* is some known functional dependency in *T*, then *A* ↦ *B* is a *known functional dependency* in *T1*.

Let *CC* be the column specified by some <value expression> *VE* in the <select list> that does not contain a potential source of non-determinism.

Let *OP1*, *OP2*, ... be the operands of *VE* that are column references whose qualifying query is *QS* and that are not contained in an aggregated argument of a <set function specification>.

If *VE* does not contain a <set function specification> whose aggregation query is *QS*, then {*OP1*, *OP2*, ...} ↦ *CC* is a *known functional dependency* in *T1*.

If *VE* contains a <set function specification> whose aggregation query is *QS*, then let {*G1*, ...} be the set of grouping columns of *T*. {*G1*, ..., *OP1*, *OP2*, ...} ↦ *CC* is a *known functional dependency* in *T1*.

Let *C* ↦ *D* be some known functional dependency in *T1*. If all the columns of *C* have counterparts in *R*, then let *DR* be the set comprising those columns of *D* that have counterparts in *R*.*C* ↦ *DR* is a *known functional dependency* in *R*.

4.26.16 Known functional dependencies in a <query expression>

If a <with clause> is specified, and RECURSIVE is not specified, then the *BPK-sets*, *BUC-sets*, and *known functional dependencies* of the table identified by a <query name> in the <with list> are the same as the BPK-sets, BUC-sets, and known functional dependencies of the corresponding <query expression>, respectively. If RECURSIVE is specified, then the BPK-sets, BUC-sets, and non-axiomatic known functional dependencies are implementation-defined (IA204).

A <query expression> that is a <query term> that is a <query primary> that is a <simple table> is covered by previous Subclauses of this Clause.

If the <query expression> specifies UNION, EXCEPT or INTERSECT, then let *T1* and *T2* be the left and right operand tables and let *R* be the result. Let *CR* be the set comprising all the columns of *R*.

Each column of *R* has a counterpart in *T1* and a counterpart in *T2*.

Case:

— If EXCEPT is specified, then a set *S* of columns of *R* is a *BPK-set* if its counterpart in *T1* is a BPK-set. *S* is a BUC-set if its counterpart in *T1* is a BUC-set.

— If UNION is specified, then there are no BPK-sets and no BUC-sets.

— If INTERSECT is specified, then a set *S* of columns of *R* is a *BPK-set* if either of its counterparts in *T1* and *T2* is a BPK-set. *S* is a *BUC-set* if either of its counterparts in *T1* and *T2* is a BUC-set.

Case:

— If UNION is specified, then no non-axiomatic functional dependency in *T1* or *T2* is a known functional dependency in *R*, apart from any functional dependencies determined by implementation-defined (IA226) rules.

— If EXCEPT is specified, then all known functional dependencies in *T1* are *known functional dependencies* in *R*.

— If INTERSECT is specified, then all known functional dependencies in *T1* and all known functional dependencies in *T2* are *known functional dependencies* in *R*.

NOTE 71 — Other known functional dependencies can be determined according to implementation-defined rules.

4.27 Candidate keys

If the functional dependency *CK* ↦ *CT* holds true in some table *T*, where *CT* consists of all columns of *T*, and there is no proper subset *CK1* of *CK* such that *CK1* ↦ *CT* holds true in *T*, then *CK* is a *candidate key* of *T*. The set of candidate keys *SCK* is non-empty because, if no proper subset of *CT* is a candidate key, then *CT* is a candidate key.

NOTE 72 — Because a candidate key is a set (of columns), *SCK* is therefore a set of sets (of columns).

A candidate key *CK* is a *strong candidate key* if *CK* is a BUC-set, or if *T* is a grouped table and *CK* is a subset of the set of grouping columns of *T*. Let *SSCK* be the set of strong candidate keys.

Let *PCK* be the set of *P* such that *P* is a member of *SCK* and *P* is a *BPK-set*.

Case:

— If *PCK* is non-empty, then the *primary key* is chosen from *PCK* as follows. If *PCK* has exactly one element, then that element is the primary key; otherwise, the left-most element of *PCK* is chosen according to the "left-most rule" below. The primary key is also the *preferred candidate key*.

— Otherwise, there is no primary key and the *preferred candidate key* is chosen as follows.

  Case:

  • If *SSCK* has exactly one element, then it is the preferred candidate key; otherwise, if *SSCK* has more than one element, then the left-most element of *SSCK* is chosen, according to the "left-most" rule below.

  • Otherwise, if *SCK* has exactly one element, then it is the preferred candidate key; otherwise, the left-most element of *SCK* is chosen, according to the "left-most" rule below.

— The "left-most" rule:

  • This rule uses the ordering of the columns of a table, as specified elsewhere in this document.

    To determine the left-most of two sets of columns of *T*, first list each set in the order of the column-numbers of its members, extending the shorter list with zeros to the length of the longer list; then, starting at the left of each ordered list, step forward until a pair of unequal column numbers, one from the same position in each list, is found. The list containing the number that is the smaller member of this pair identifies the left-most of the two sets of columns of *T*.

    To determine the left-most of more than two sets of columns of *T*, take the left-most of any two sets, then pair that with one of the remaining sets and take the left-most, and so on until there are no remaining sets.

4.28 SQL-schemas

This Subclause is modified by Subclause 4.4, "SQL-schemas", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.6, "SQL-schemas", in ISO/IEC 9075-9.

An SQL-schema is a persistent descriptor that includes:

— The name of the SQL-schema.

— The <authorization identifier> of the owner of the SQL-schema.

— The name of the default character set for the SQL-schema.

— The <schema path specification> defining the SQL-path for SQL-invoked routines for the SQL-schema.

— The descriptor of every component of the SQL-schema.

04 09 In this document, the term "schema" is used only in the sense of SQL-schema. The persistent objects described by the descriptors are said to be *owned by* or to have been *created by* the <authorization identifier> of the schema. Each component descriptor is one of:

— A domain descriptor.

— 09 A base table descriptor.

— A view descriptor.

— A constraint descriptor.

— A trigger descriptor.

— A privilege descriptor.

— A character set descriptor.

— A collation descriptor.

— A transliteration descriptor.

— A user-defined type descriptor.

— A routine descriptor.

— A sequence generator descriptor.

A schema is created initially using a <schema definition> and may be subsequently modified incrementally over time by the execution of <SQL schema statement>s. <schema name>s are unique within a catalog.

A <schema name> is explicitly or implicitly qualified by a <catalog name> that identifies a catalog.

09 Base tables and views are identified by <table name>s. A <table name> consists of a <schema name>, followed by a <period>, followed by an <identifier>. The <schema name> identifies the schema that includes the table descriptor of the persistent object identified by the <table name>. The <table name>s of persistent objects defined in different schemas can have equivalent <identifier>s.

NOTE 73 — Equivalence of <identifier>s is defined in Subclause 5.2, "<token> and <separator>".

If a reference to a <table name> does not explicitly contain a <schema name>, then a specific <schema name> is implied. The particular <schema name> associated with such a <table name> depends on the context in which the <table name> appears and is governed by the rules for <schema qualified name>.

If a reference to an SQL-invoked routine that is contained in a <routine invocation> does not explicitly contain a <schema name>, then the SQL-invoked routine is selected from the SQL-path of the schema.

The *containing schema* of an <SQL schema statement> is defined as the schema identified by the <schema name> implicitly or explicitly contained in the name of the object that is created or manipulated by that SQL-statement.

4.29 Sequence generators

4.29.1 General description of sequence generators

A *sequence generator* is a mechanism for generating successive exact numeric values, one at a time. A sequence generator is either an *external sequence generator* or an *internal sequence generator*. An external sequence generator is a named schema object while an internal sequence generator is a component of another schema object. A sequence generator has a data type, which shall be an exact numeric type with scale 0 (zero), a minimum value, a maximum value, a start value, an increment, and a cycle option.

Specification of a sequence generator can optionally include the specification of a data type, a minimum value, a maximum value, a start value, an increment, and a cycle option.

If a sequence generator is associated with a negative increment, then it is a *descending sequence generator*; otherwise, it is an *ascending sequence generator*.

A sequence generator has a time-varying *current base value*, which is a value of its data type. A sequence generator has a cycle which consists of all the possible values between the minimum value and the maximum value which are expressible as (current base value + *N* * increment), where *N* is a non-negative number.

When a sequence generator is created, its current base value is initialized to the start value. Subsequently, the current base value is set to the value of the lowest non-issued value in the cycle for an ascending sequence generator, or the highest non-issued value in the cycle for a descending sequence generator.

Any time after a sequence generator is created, its current base value can be set to an arbitrary value of its data type by an <alter sequence generator statement>.

Changes to the current base value of a sequence generator are not controlled by SQL-transactions; therefore, commits and rollbacks of SQL-transactions have no effect on the current base value of a sequence generator.

A sequence generator is described by a sequence generator descriptor. A sequence generator descriptor includes:

— The sequence generator name that is a schema-qualified sequence generator name for an external sequence generator and the zero-length character string for an internal sequence generator.

— The data type descriptor of the data type associated with the sequence generator.

— The start value of the sequence generator.

— The minimum value of the sequence generator.

— The maximum value of the sequence generator.

— The increment of the sequence generator.

— The cycle option of the sequence generator.

— The current base value of the sequence generator.

4.29.2 Operations involving sequence generators

When a <next value expression> is applied to a sequence generator *SG*, *SG* issues a value *V* taken from *SG*'s current cycle such that *V* is expressible as the current base value of *SG* plus *N* multiplied by the increment of *SG*, where *N* is a non-negative number.

Thus a sequence generator will normally issue all of the values in its cycle and these will normally be in increasing or decreasing order (depending on the sign of the increment) but within that general ordering separate subgroups of ordered values may occur.

If the sequence generator's cycle is exhausted (i.e., it cannot issue a value that meets the criteria), then a new cycle is created with the current base value set to the minimum value of *SG* (if *SG* is an ascending sequence generator) or the maximum value of *SG* (if *SG* is a descending sequence generator).

If a new cycle is created and the descriptor of *SG* includes NO CYCLE, then an exception condition is raised.

If there are multiple instances of <next value expression>s specifying the same sequence generator within a single SQL-statement, all those instances return the same value for a given row processed by that SQL-statement.

4.30 SQL-client modules

An *SQL-client module* is an SQL-environment object that can include externally-invoked procedures and certain descriptors. An SQL-client module is created and destroyed by implementation-defined (IW109) mechanisms (which can include the granting and revoking of privileges required for the use of the SQL-client module). An SQL-client module exists in the SQL-environment containing an SQL-client.

If an SQL-client module *S* is defined by an <SQL-client module definition> that contains a <module authorization identifier> *MAI*, then the owner of *S* is *MAI*; otherwise, *S* has no owner.

An SQL-client module can be specified by a <SQL-client module definition> (see Subclause 13.1, "<SQL-client module definition>").

An SQL-client module includes:

— The name, if any, of the SQL-client module.

— The name of the host language from a compilation unit of which an externally-invoked procedure included in the module can be invoked.

— The <module authorization identifier>, if any.

— An indication of whether or not the <module authorization identifier> is to apply to execution of prepared statements resulting from invocation of externally-invoked procedures in the SQL-client module that contain <prepare statement>s or <execute immediate statement>s.

— SQL-client module defaults, for use in the application of Syntax Rules to <externally-invoked procedure>s, <temporary table declaration>s, and <declare cursor>s.

  • The name of the schema for use as the default <schema name> when deriving externally-invoked procedures from <externally-invoked procedure>s, specified either by the <schema name> or, failing that, by the <module authorization identifier>.

  • The SQL-path, if any, used to qualify:

    — Unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the SQL-client module.

    — Unqualified <user-defined type name>s that are immediately contained in <path-resolved user-defined type name>s that are contained in the SQL-client module.

  • The names of zero or more *SQL-client module collations*, each specifying a collation for one or more character sets for the SQL-client module.

— The name, if specified, of the character set used to express the <SQL-client module definition>.

NOTE 74 — The <module character set specification> has no effect on the SQL language contained in the SQL-client module and exists only for compatibility with ISO/IEC 9075:1992. It is used to document the character set of the SQL-client module.

— Module contents:

  • Zero or more temporary table descriptors.

  • Zero or more cursor declaration descriptors.

  • One or more externally-invoked procedures.

A compilation unit is a segment of executable code, possibly consisting of one or more subprograms. An SQL-client module is associated with a compilation unit during its execution. A single SQL-client module may be associated with multiple compilation units and multiple SQL-client modules may be associated with a single compilation unit. The manner in which this association is specified, including the possible requirement for execution of some implementation-defined (IW110) statement, is implementation-defined (IW110). Whether a compilation unit may invoke or transfer control to other compilation units, written in the same or a different programming language, is implementation-defined (IA205).

4.31 Embedded syntax

This Subclause is modified by Subclause 4.3, "Embedded syntax", in ISO/IEC 9075-10.

10 An <embedded SQL host program> is a compilation unit that consists of host language text and SQL text. The host language text conforms to the requirements of a specific programming language. The SQL

text consists of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s, as defined in this document. This allows database applications to be expressed in a hybrid form in which SQL-statements are embedded directly in a compilation unit. Such a hybrid compilation unit is defined to be equivalent to:

— An SQL-client module, containing externally-invoked procedures and declarations.

— A host language compilation unit in which each SQL-statement has been replaced by an invocation of an externally-invoked procedure in the SQL-client module, and the declarations contained in such SQL-statements have been suitably transformed into declarations in the host language.

If an <embedded SQL host program> contains an <embedded authorization declaration>, then it shall be the first statement or declaration in the <embedded SQL host program>. The <embedded authorization declaration> is not replaced by a procedure or subroutine call of an <externally-invoked procedure>, but is removed and replaced by syntax associated with the <SQL-client module definition>'s <module authorization clause>.

An SQL-implementation may reserve portions of the namespaces in the <embedded SQL host program> for the names of host language elements and host language environment elements that are generated to replace SQL-statements. This includes the names of procedures, subroutines, classes, and other resources in the host language environment, and for program variables and branch labels that may be generated as required to support the calling of these procedures or subroutines. Whether any such reservations are made, and the details of any such reservations, if any, is implementation-defined (IA206).

4.32 Dynamic SQL concepts

4.32.1 Introduction to dynamic SQL

Dynamic SQL features may be used in at least two ways:

— To execute a statement that is constructed during the execution of an SQL-client module.

— During the execution of a polymorphic table function.

This Subclause and its subordinate Subclauses are primarily concerned with the first use. However, because polymorphic table functions use some of the same features as statement-oriented dynamic SQL, there is some mention of features particular to polymorphic table functions, especially to introduce distinctive terminology where necessary to distinguish between the two kinds of dynamic SQL.

4.32.2 Overview of dynamic SQL for constructed SQL-statements

In many cases, the SQL-statement to be executed can be coded into an <SQL-client module definition> or into a compilation unit using the embedded syntax. In other cases, the SQL-statement is not known when the program is written and will be generated during program execution.

Dynamic execution of SQL-statements can generally be accomplished in two different ways. Statements can be *prepared* for execution and then later executed one or more times; when the statement is no longer needed for execution, it can be *released* by the use of a <deallocate prepared statement>. Alternatively, a statement that is needed only once can be executed without the preparation step—it can be *executed immediately* (not all SQL-statements can be executed immediately).

Many SQL-statements can be written to use "parameters" (which are manifested in static execution of SQL-statements as host parameters in <SQL procedure statement>s contained in <externally-invoked procedure>s in <SQL-client module definition>s or as host variables in <embedded SQL statement>s contained in <embedded SQL host program>s). In SQL-statements that are executed dynamically, the parameters are called dynamic parameters (<dynamic parameter specification>s) and are represented in SQL language by a <question mark> (?).

In many situations, an application that generates an SQL-statement for dynamic execution knows in detail the required characteristics (e.g., <data type>, <length>, <precision>, <scale>) of each of the dynamic parameters used in the statement; similarly, the application may also know in detail the characteristics of the values that will be returned by execution of the statement. However, in other cases, it is possible that the application does not know this information to the required level of detail; it is possible in some cases for the application to ascertain the information from the Information Schema, but in other cases (e.g., when a returned value is derived from a computation instead of simply from a column in a table, or when dynamic parameters are supplied) this information is not generally available except in the context of preparing the statement for execution.

NOTE 75 — The Information Schema is defined in ISO/IEC 9075-11.

To provide the necessary information to applications, SQL permits an application to request the SQL-server to *describe* a prepared statement. The description of a statement identifies the number of input dynamic parameters (*describe input*) and their data type information or it identifies the number of output dynamic parameters or values to be returned (*describe output*) and their data type information. The description of a statement is placed into the SQL descriptor areas already mentioned.

Many, but not all, SQL-statements can be prepared and executed dynamically.

NOTE 76 — The complete list of statements that are dynamically preparable and executable is defined in Subclause 4.41.7, "Preparable and immediately executable SQL-statements".

Certain "set statements" (<set catalog statement>, <set schema statement>, <set names statement>, and <set path statement>) have no effect other than to set up default information (catalog name, schema name, character set, and SQL path, respectively) to be applied to other SQL-statements that are prepared or executed immediately or that are invoked directly.

Syntax errors and Access Rule violations caused by the preparation or immediate execution of <preparable statement>s are identified when the statement is prepared (by <prepare statement>) or when it is executed (by <execute statement> or <execute immediate statement>); such violations are indicated by the raising of an exception condition.

When describing input dynamic parameters, the number of input dynamic parameters is the number of <dynamic parameter specification>s in the SQL-statement prior to any syntactic transformations defined in the Syntax Rules of this document. When executing a prepared SQL-statement, all instances of an input dynamic parameter that has been effectively replicated are set to the same value, which is the value supplied for that input dynamic parameter.

4.32.3 Overview of dynamic SQL for polymorphic table functions

Polymorphic table functions are described in Subclause Subclause 4.35, "SQL-invoked routines". Polymorphic table functions use SQL descriptor areas to describe the row types of generic tables, and dynamic cursors to support table arguments. Polymorphic table functions may fetch from a dynamic cursor into an SQL descriptor area. Polymorphic table functions use a dynamic SQL statement, <pipe row statement>, to output a row of the result table. In addition, polymorphic table functions may construct SQL-statements for preparation or dynamic execution.

4.32.4 Dynamic SQL statements and descriptor areas

An <execute immediate statement> can be used for a one-time preparation and execution of an SQL-statement. A <prepare statement> is used to prepare the generated SQL-statement for subsequent execution. A <deallocate prepared statement> is used to deallocate SQL-statements that have been prepared with a <prepare statement>. A description of the input dynamic parameters for a prepared statement can be obtained by execution of a <describe input statement>. A description of the resultant columns of a <dynamic select statement> or <dynamic single row select statement> can be obtained by execution of a <describe output statement>. A description of the output dynamic parameters of a statement that is neither a <dynamic select statement> nor a <dynamic single row select statement> can be obtained by execution of a <describe output statement>.

For a statement other than a <dynamic select statement>, an <execute statement> is used to associate parameters with the prepared statement and execute it as though it had been coded when the program was written. For a <dynamic select statement>, the prepared <cursor specification> is associated with a declared dynamic cursor via a <dynamic declare cursor> or with an extended dynamic cursor via an <allocate extended dynamic cursor statement>. The dynamic cursor can be opened and dynamic parameters can be associated with the dynamic cursor with a <dynamic open statement>. Operations on an open dynamic cursor are described in Subclause 4.40.2, "Operations on and using cursors".

The interface for input dynamic parameters and output dynamic parameters for a prepared statement and for the resulting values from a <dynamic fetch statement> or the execution of a prepared <dynamic single row select statement> can be either a list of dynamic parameters or embedded variables or an SQL descriptor area. An SQL descriptor area consists of one or more item descriptor areas, together with a header that includes a count of the number of those item descriptor areas. The header of an SQL descriptor area consists of the components in Table 28, "Data types of <key word>s used in the header of SQL descriptor areas", in Subclause 20.1, "Description of SQL descriptor areas". Each item descriptor area consists of the components specified in Table 29, "Data types of <key word>s used in SQL item descriptor areas", in Subclause 20.1, "Description of SQL descriptor areas". The SQL descriptor area is allocated and maintained by the system with the following statements: <allocate descriptor statement>, <deallocate descriptor statement>, <set descriptor statement>, and <get descriptor statement>.

Two kinds of identifier are used for referencing dynamic SQL objects, *extended names* and *non-extended names*. An extended name is an <identifier> assigned to a parameter or variable and the object it identifies is referenced indirectly, by referencing that parameter or variable. A non-extended name is just an <identifier> and the object it identifies is referenced by using that <identifier> directly in an SQL-statement.

SQL descriptor areas, dynamic statements, and dynamic cursors can be identified either by non-extended names or by extended names.

Two extended names are equivalent if their values, with leading and trailing <space>s removed, are equivalent according to the rules for <identifier> comparison in Subclause 5.2, "<token> and <separator>".

The *scope* of an extended name is *global*, *local*, or the invocation of the component procedures of a polymorphic table function, and is determined by the run-time context in which the object it identifies is brought into existence.

The scope of a global extended name *GEN* is the SQL-session, meaning that, during the existence of the object *O* it identifies, *GEN* can be used to reference *O* by any SQL-statement executed in that SQL-session.

The scope of a local extended name *LEN* is the SQL-client module *M* containing the externally-invoked procedure that is being executed when the object *O* identified by *LEN* is brought into existence. This means that, during the existence of *O*, *LEN* can be used to reference *O* by any SQL-statement executed in the same SQL-session by an externally-invoked procedure in *M*.

The scope of a PTF extended name *PTFEN* is the invocation of the component procedures of a polymorphic table function *PTF*. *PTFEN* identifies an object *O* during the execution of the PTF component procedures of *PTF*. If polymorphic table function invocations are nested, then a PTF extended name references an object of the innermost active polymorphic table invocation.

The scope of a non-extended name is the <SQL-client module definition> containing the SQL-statement that defines it.

NOTE 77 — The namespace of non-extended names is different from the namespace of extended names.

Let *PRP* be the prepared statement resulting from execution of a <prepare statement> in an externally-invoked procedure, SQL-invoked routine, or triggered action *E*. In the following cases, *PRP* has no owner:

— *E* is an SQL-invoked routine whose security characteristic is INVOKER.

— *E* is an externally-invoked procedure contained in an SQL-client module that either has no owner or for which FOR STATIC ONLY was specified.

Otherwise, the owner of *PRP* is the owner of *E*.

When an <execute statement> executes *PRP*, if *PRP* has an owner, then the top cell of the authorization stack is set to contain only the authorization identifier of the owner of *PRP*.

4.33 Direct invocation of SQL

Direct invocation of SQL is a mechanism for executing direct SQL-statements, known as <direct SQL statement>s. In direct invocation of SQL, the method of invoking <direct SQL statement>s, the method of raising conditions that result from the execution of <direct SQL statement>s, the method of accessing the diagnostics information that results from the execution of <direct SQL statement>s, and the method of returning the results are implementation-defined (IW113). The method of accessing the diagnostics information shall not alter the contents of the diagnostics area.

4.34 Externally-invoked procedures

An externally-invoked procedure consists of an SQL-statement and can be invoked from a compilation unit of a host language. The host language is specified by the <language clause> of the SQL-client module that contains the externally-invoked procedure.

4.35 SQL-invoked routines

This Subclause is modified by Subclause 4.6, "SQL-invoked routines", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 4.5, "SQL-invoked routines", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.3, "SQL-invoked routines", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.6, "SQL-invoked routines", in ISO/IEC 9075-14.

4.35.1 Overview of SQL-invoked routines

This Subclause is modified by Subclause 4.5.1, "Overview of SQL-invoked routines", in ISO/IEC 9075-4.

Figure 4, "Taxonomy of SQL-invoked routines", depicts the relationships between various categories of SQL-invoked routines. Every use of the word "or" in this paragraph is exclusive. An *SQL-invoked routine* is an SQL-invoked procedure or an SQL-invoked function. An SQL-invoked function is a conventional SQL-invoked function or a polymorphic table function. A conventional SQL-invoked function is an SQL-invoked method or an SQL-invoked regular function. A conventional SQL-invoked routine is an SQL-invoked procedure, SQL-invoked method, or an SQL-invoked regular function. (Stated differently, an SQL-invoked routine that is not a polymorphic table function is a conventional SQL-invoked routine.) A conventional SQL-invoked routine has a routine body, by which it is classified as either an SQL routine or an external routine. Polymorphic table functions are not classified by routine body. Not shown in Figure 4, "Taxonomy of SQL-invoked routines", SQL-invoked methods are subdivided into the mutually exclusive categories of instance methods, constructor methods, and static methods; they can also be classified as original or overriding methods.

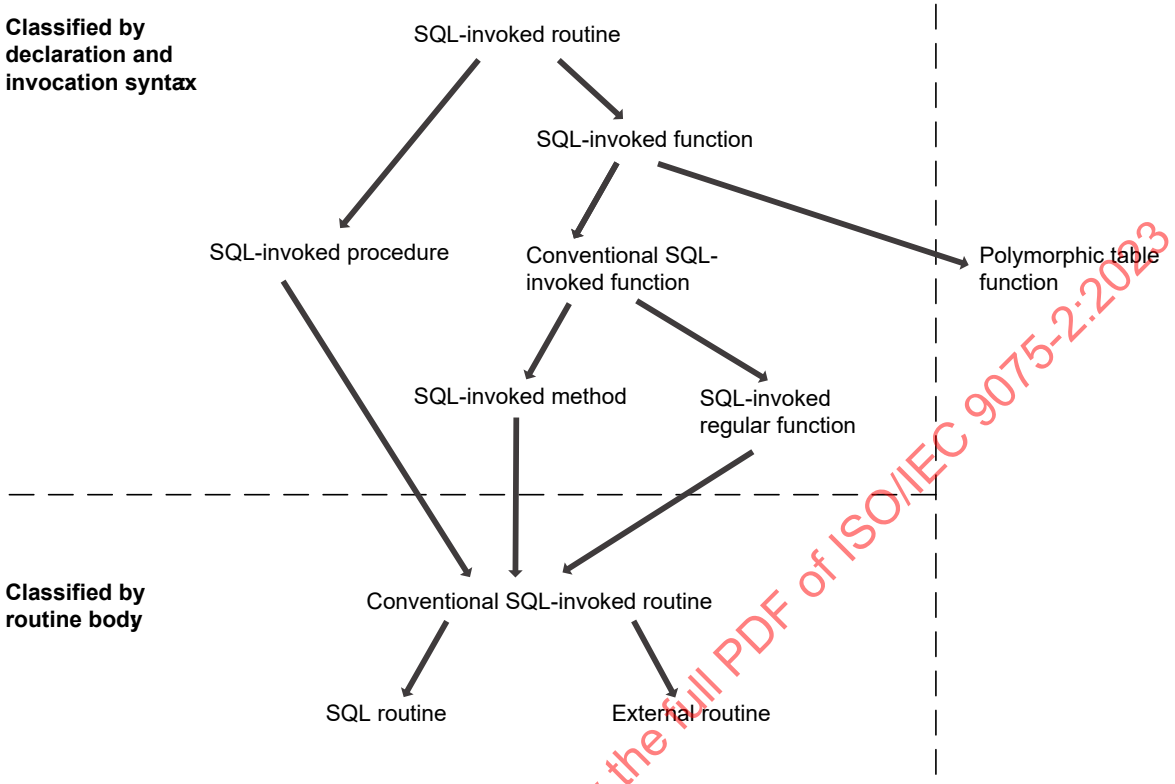

Figure 4 — Taxonomy of SQL-invoked routines

An SQL-invoked routine comprises at least a routine name, zero or more SQL parameters, and a routine body. Each parameter has a *parameter type* (also known as declared type), specified as a <data type>, <generic table parameter type>, or <descriptor parameter type>.

04 An SQL-invoked routine that is an element of an SQL-schema is called a *schema-level routine*.

An SQL-invoked routine *SR* is said to be *dependent* on a user-defined type *UDT* if *SR* is created during the execution of the <user-defined type definition> that created *UDT* or if *SR* is created during the execution of an <alter type statement> that specifies an <add attribute definition>. An SQL-invoked routine that is dependent on a user-defined type cannot be modified by an <alter routine statement> or be destroyed by a <drop routine statement>. It is destroyed implicitly by a <drop data type statement>.

An *SQL-invoked procedure* is a conventional SQL-invoked routine that is invoked from an SQL <call statement>. An SQL-invoked procedure may have input SQL parameters, output SQL parameters, and SQL parameters that are both input SQL parameters and output SQL parameters. A default value may be associated with an SQL parameter that is an input SQL parameter or both an input SQL parameter and an output SQL parameter. The format of an SQL-invoked procedure is specified by <SQL-invoked procedure> (see Subclause 11.60, "<SQL-invoked routine>").

An SQL-invoked procedure may optionally be specified to require a new savepoint level to be established when it is invoked and destroyed on return from the executed routine body. The alternative of not taking a savepoint can also be directly specified with OLD SAVEPOINT LEVEL. When an SQL-invoked function is invoked a new savepoint level is always established. Savepoint levels are described in Subclause 4.43.2, "Savepoints".

An *SQL-invoked function* is either a conventional SQL-invoked function or a polymorphic table function. A *conventional SQL-invoked function* is an SQL-invoked routine whose invocation returns a value of some

<data type>. Every parameter of a conventional SQL-invoked function is an input SQL parameter, one of which may be designated as the result SQL parameter. A default value may be associated with any SQL parameter of a conventional SQL-invoked function. The format of a conventional SQL-invoked function is specified by <SQL-invoked function> (see Subclause 11.60, "<SQL-invoked routine>"). A conventional SQL-invoked function can be a *type-preserving function*; a type-preserving function is an SQL-invoked function that has a result SQL parameter. The most specific type of a non-null result of invoking a type-preserving function shall be compatible with the most specific type of the value of the argument substituted for its result SQL parameter.

A *polymorphic table function* is an SQL-invoked routine whose invocation returns a table, and either the return type or at least one parameter type is a generic table (a table whose row type is not declared when the polymorphic table function is created). Every parameter of a polymorphic table function is an input SQL parameter. A default value may be associated with any SQL parameter of a polymorphic table function that is not a generic table parameter. The format of a polymorphic table function is specified by <SQL-invoked function> (see Subclause 11.60, "<SQL-invoked routine>").

An *SQL-invoked method* is an SQL-invoked function that is specified by <method specification designator> (see Subclause 11.60, "<SQL-invoked routine>"). There are three kinds of SQL-invoked methods: *SQL-invoked constructor methods*, *instance SQL-invoked methods* and *static SQL-invoked methods*. All SQL-invoked methods are associated with a user-defined type, also known as the *type of the method*. The <method characteristic>s of an SQL-invoked method are specified by a <method specification> contained in the <user-defined type definition> of the type of the method. For both an instance SQL-invoked method and an SQL-invoked constructor method all of the following are true:

— Its first parameter, called the *subject parameter*, has a declared type that is a user-defined type. The type of the subject parameter is the type of the method. A parameter other than the subject parameter is called an *additional parameter*.

— Its descriptor is in the same schema as the descriptor of the data type of its subject parameter.

An SQL-invoked constructor method satisfies the following additional conditions:

— Its <method name> is equivalent to the <qualified identifier> simply contained in the <user-defined type name> included in the user-defined type descriptor of the type of the method.

A static SQL-invoked method satisfies the following conditions:

— It has no subject parameter. Its first parameter, if any, is treated no differently than any other parameter.

— Its descriptor is in the same schema as the descriptor of the structured type of the method. The name of this type (or of some subtype of it) is always specified together with the name of the method when the method is to be invoked.

A conventional SQL-invoked function that is not an SQL-invoked method is an *SQL-invoked regular function*. An SQL-invoked regular function is specified by <function specification> (see Subclause 11.60, "<SQL-invoked routine>").

A *null-call function* is an SQL-invoked function that is defined to return the null value if any of its input arguments is the null value. A null-call function is an SQL-invoked function whose <null-call clause> specifies "RETURNS NULL ON NULL INPUT".

4.35.2 Characteristics of SQL-invoked routines

This Subclause is modified by Subclause 4.3.1, "Characteristics of SQL-invoked routines", in ISO/IEC 9075-13.

A conventional SQL-invoked routine can be an *SQL routine* or an *external routine*. An SQL routine is an SQL-invoked routine whose <language clause> specifies SQL. The <routine body> of an SQL routine is an <SQL procedure statement>; the <SQL procedure statement> forming the <routine body> can be any SQL-statement, including an <SQL control statement>, but excluding an <SQL connection statement> and

an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

13 An external routine is one whose <language clause> does not specify SQL. The <routine body> of an external routine is an <external body reference> whose <external routine name> identifies a program written in some programming language other than SQL. The program identified by <external routine name> shall not execute either an <SQL connection statement> or an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

The <routine body> of a polymorphic table function does not specify a single routine; instead a polymorphic table function has one or more *PTF component procedures*, each of which is an SQL-invoked procedure. The PTF component procedures of a polymorphic table function are:

— The *PTF describe component procedure* (optional).

— The *PTF start component procedure* (optional).

— The *PTF fulfill component procedure* (required).

— The *PTF finish component procedure* (optional).

PTF component procedures are SQL-invoked procedures, and consequently they have the characteristics of SQL-invoked procedures, as described in this Subclause. The role of the PTF component procedures is described in Subclause 4.35.4, "Invocation of polymorphic table functions". A polymorphic table function may optionally have one or more private parameters, which declare the private variables that are available for communication between the PTF component procedures.

An SQL-invoked routine is uniquely identified by a <specific name>, called the *specific name* of the SQL-invoked routine.

SQL-invoked routines are invoked differently depending on their form. SQL-invoked procedures are invoked by <call statement>s. SQL-invoked regular functions are invoked by <routine invocation>s that are not simply contained in a <PTF derived table>. Instance SQL-invoked methods are invoked by <method invocation>s, while SQL-invoked constructor methods are invoked by <new specification>s and static SQL-invoked methods are invoked by <static method invocation>s. A polymorphic table function is invoked by a <PTF derived table>.

An invocation of an SQL-invoked routine specifies the <routine name> of the SQL-invoked routine and supplies a sequence of argument values corresponding to the <SQL parameter declaration>s of the SQL-invoked routine. The sequence of argument values can be passed either by position or by argument name. A *subject routine* of an invocation is an SQL-invoked routine that may be invoked by a <routine invocation>. After the selection of the subject routine of a <routine invocation>, the SQL arguments are evaluated and the SQL-invoked routine that will be executed is selected.

13 The following cases apply when selecting the SQL-invoked routine to be executed:

— If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is selected from the set of overriding methods of the subject routine. (The term "set of overriding methods" is defined in the General Rules of Subclause 9.18, "Invoking an SQL-invoked routine".) The overriding method that is selected is the overriding method with a subject parameter the type designator of whose declared type precedes that of the declared type of the subject parameter of every other overriding method in the type precedence list of the most specific type of the value of the SQL argument that corresponds to the subject parameter. See the General Rules of Subclause 9.18, "Invoking an SQL-invoked routine".

— If the subject routine is not an SQL-invoked method, then the SQL-invoked routine executed is that subject routine.

After the selection of the SQL-invoked routine for execution, the values of the SQL arguments that do not specify DEFAULT are assigned to the corresponding SQL parameters of the SQL-invoked routine and default values are assigned to any SQL parameter of the SQL-invoked routine for which either no SQL argument is supplied or for which the corresponding SQL argument specifies DEFAULT.

After constructing the argument list, the SQL-invoked routine is executed. The execution of polymorphic table functions is described in Subclause 4.35.4, "Invocation of polymorphic table functions". If the SQL-invoked routine is not a polymorphic table function, then its <routine body> is executed. If the SQL-invoked routine is an SQL routine, then the <routine body> is an <SQL procedure statement> that is executed according to the General Rules of <SQL procedure statement>. If the SQL-invoked routine is an external routine, then the <routine body> identifies a program written in some programming language other than SQL that is executed according to the rules of that host language.

The <routine body> of a conventional SQL-invoked routine is always executed under the same SQL-session from which the SQL-invoked routine was invoked. Before the execution of the <routine body>, a new context for the current SQL-session is created and the values of the current context preserved. When the execution of the <routine body> completes the original context of the current SQL-session is restored.

The PTF describe component procedure of a polymorphic table function is executed under the same SQL-session from which the polymorphic table function was invoked. The other PTF component procedures of a polymorphic table function are executed on one or more virtual processors, each of which has a new context for the current SQL-session, that being a copy of the current context. As each virtual processor completes the execution of its PTF component procedures, the virtual processor is destroyed. When all virtual processors have been destroyed, the original context of the current SQL-session is restored. See Subclause 4.35.4, "Invocation of polymorphic table functions".

If the SQL-invoked routine is an external routine, then an effective SQL parameter list is constructed before the execution of the <routine body>. The effective SQL parameter list has different entries depending on the parameter passing style of the SQL-invoked routine.

The value of each entry in the effective SQL parameter list is set according to the General Rules of Subclause 9.18, "Invoking an SQL-invoked routine".

13 The values in the effective SQL parameter list are passed to the program identified by the <routine body> according to the rules of Subclause 13.5, "Data type correspondences".

After the execution of that program, if the parameter passing style of the SQL-invoked routine is SQL, then the SQL-implementation obtains the values for output parameters (if any), the value (if any) returned from the program, the value of the SQLSTATE, and the value of the message text (if any) from the values assigned by the program to the effective SQL parameter list. If the parameter passing style of the SQL-invoked routine is GENERAL, then such values are obtained in an implementation-defined (IW114) manner.

Different SQL-invoked routines can have equivalent <routine name>s. No two SQL-invoked functions in the same schema are allowed to have the same signature. No two SQL-invoked procedures in the same schema are allowed to have the same name and the same number of parameters. Subject routine determination is the process for choosing the subject routine for a given <routine invocation> given a <routine name> and an <SQL argument list>. Subject routine determination for SQL-invoked functions considers the most specific types of all of the arguments (that is, all of the arguments that are not <dynamic parameter specification>s whose types are not known at the time of subject routine determination) to the invocation of the SQL-invoked function in order from left to right. Where there is not an exact match between the most specific types of the arguments and the declared types of the parameters, type precedence lists are used to determine the closest match. See Subclause 9.6, "Subject routine determination".

If a <routine invocation> is contained in the original <query expression> of a view, in the <search condition> of a check constraint or an assertion, the <triggered action> of a trigger, or in an <SQL-invoked routine>, then the subject routine for that invocation is determined at the time the view is created, the check constraint is defined, the assertion is created, the trigger is created, or the SQL-invoked routine is

created. The PTF component procedures of a polymorphic table function are determined at the time the polymorphic table function is created. If a <routine invocation> is contained in the hierarchical <query expression> of a view *V*, then the subject routine for that invocation is determined at the time the subview *SV* of *V* whose original <query expression> contains the <routine invocation> is created. If the subject routine is an SQL-invoked procedure, an SQL-invoked regular function, a static SQL-invoked method, or a polymorphic table function, then the same SQL-invoked routine is executed whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked. If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is determined whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked, based on the most specific type of the value resulting from the evaluation of the SQL argument that correspond to the subject parameter. See the General Rules of Subclause 9.18, "Invoking an SQL-invoked routine".

All <identifier chain>s in the <routine body> of an SQL routine are resolved to identify the basis and basis referent at the time that the SQL routine is created. Thus, the same columns and SQL parameters are referenced whenever the SQL routine is invoked.

An SQL-invoked routine is either *deterministic* or *possibly non-deterministic*. An SQL-invoked function that is deterministic always returns the identical return value for a given list of SQL argument values. An SQL-invoked procedure that is deterministic always returns the identical values in its output and inout SQL parameters for a given list of SQL argument values. An SQL-invoked routine is possibly non-deterministic if it might produce non-identical results when invoked with the identical list of SQL argument values.

An external routine *does not possibly contain SQL*, *possibly contains SQL*, *possibly reads SQL-data*, or *possibly modifies SQL-data*. Only an external routine that possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data is permitted to execute SQL-statements during its invocation. Only an SQL-invoked routine that possibly reads SQL-data or possibly modifies SQL-data may read SQL-data during its invocation. Only an SQL-invoked routine that possibly modifies SQL-data may modify SQL-data during its invocation.

An SQL-invoked routine has a *routine authorization identifier*, which is (directly or indirectly) the authorization identifier of the owner of the schema that contains the SQL-invoked routine at the time that the SQL-invoked routine is created.

4.35.3 Execution of conventional SQL-invoked routines

This Subclause is modified by Subclause 4.5.2, "Execution of conventional SQL-invoked routines", in ISO/IEC 9075-4.

When an SQL-invoked routine is invoked, a copy of the current SQL-session context is pushed onto the stack and some values are modified (see the General Rules of Subclause 9.18, "Invoking an SQL-invoked routine"), before the <routine body> is executed. The treatment of the authorization stack is described in Subclause 4.42.1.2, "SQL-session authorization identifiers".

04 An SQL-invoked routine has a *routine SQL-path*, which is inherited from its containing SQL-schema, the current SQL-session, or the containing SQL-client module.

An SQL-invoked routine that is an external routine also has an *external routine SQL-path*, which is derived from the <module path specification>, if any, of the <SQL-client module definition> contained in the external program identified by the routine body of the external routine. If that <SQL-client module definition> does not specify a <module path specification>, then the external routine SQL-path is an implementation-defined (ID041) SQL-path. For both SQL and external routines, the SQL-path of the current SQL-session is used to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>. SQL routines use the routine SQL-path to determine the search order for the subject routines of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained

in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>. External routines use the external routine SQL-path to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> that is prepared in the current SQL-session or in a <direct SQL statement>.

4.35.4 Invocation of polymorphic table functions

An invocation *PTFI* of a polymorphic table function *PTF* is performed in two phases, called *compilation* and *execution*. Generally speaking, compilation is performed by Syntax Rules, and execution is performed by General Rules.

NOTE 78 — During compilation, there are certain Syntax Rules *SRS* that invoke certain General Rules *GRS*. Any exception condition that occurs during *GRS* is effectively converted to *syntax error or access rule violation (42000)* by the invoking *SRS*.

The <routine body> of *PTF* specifies one or more of the following PTF component procedures:

— The *PTF describe component procedure* (optional).

— The *PTF start component procedure* (optional).

— The *PTF fulfill component procedure* (required).

— The *PTF finish component procedure* (optional).

The PTF component procedures are conventional SQL-invoked procedures, which are invoked automatically as part of the polymorphic table function invocation. This Subclause describes the order of invocation and coordination between the PTF component procedures of *PTF*, which collectively constitutes the execution of *PTF*.

NOTE 79 — The PTF component procedures in this regard are analogous to the underlying tables of a view. A query that references a view must ultimately reference its underlying tables, yet the query itself does not mention the underlying tables, and the user might lack the privilege to know the names or columns of the underlying tables. In the same fashion, a query that invokes a polymorphic table function must ultimately invoke the PTF component procedures; however, the query does not contain explicit mention of the PTF component procedures, and the user might not have the privilege to learn the identity of the PTF component procedures.

The compilation phase of *PTFI* invokes the PTF describe component procedure.

The execution phase of *PTFI* is performed on one or more virtual processors. Each virtual processor invokes the PTF start component procedure (if any), PTF fulfill component procedure, and PTF finish component procedure (if any). Each virtual processor is assigned a list of PTF dynamic cursors, one for each input table, which may be read by the PTF fulfill component procedure. The PTF component procedures of the execution phase may generate result rows by performing a <pipe row statement>. The result of *PTFI* is the multiset union of all result rows that are generated on all virtual processors.

As a specification device solely for polymorphic table functions, limited support for SQL variables as defined in ISO/IEC 9075-4 is assumed; this does not assume or require conformance to ISO/IEC 9075-4. These SQL variables are called *PTF variables*.

The compilation phase and each virtual processor of the execution phase of *PTFI* use an associated *PTF data area*, comprising the following items:

— A PTF variable (called a *private variable*) for each private parameter of PTF. The declared type of each private variable is the declared type of the corresponding private parameter.

— For each generic table parameter *GTA* of *PTF*,

- A PTF descriptor area of the full row type of *GTA*.
- If *GTA* has set semantics, then:

  - — A PTF descriptor area of the partitioning of *GTA*.
  - — A PTF descriptor area of the ordering of *GTA*.
- A PTF descriptor area of the requested row type of *GTA*.
- A PTF dynamic cursor for *GTA*.
- A PTF descriptor area of the row type of the PTF dynamic cursor for *GTA*.

— For each descriptor argument *DA* of *PTF*, either the null value or an SQL descriptor area.

— A PTF descriptor area for the initial result row type of *PTF*.

— A PTF descriptor area of the intermediate result row type of *PTF*.

— A PTF variable of declared type CHARACTER(5) called the *status variable*.

Each PTF variable in a PTF data area has a distinct implementation-dependent (UV063) name. Each SQL descriptor area and each PTF dynamic cursor in a PTF data area area has a distinct PTF extended name. The scope of these variable names and PTF extended names is the execution of the component procedures of PTF as they are invoked in Subclause 9.27, "Invocation of a PTF component procedure", during the compilation and execution phases of *PTFI*.

Pass-through columns are a device that enables a polymorphic table function to conveniently copy columns of a table argument to a result row. A generic table parameter specifies either PASS THROUGH or NO PASS THROUGH. If a generic table parameter specifies PASS THROUGH, then the cursor for that generic table parameter is augmented with one additional column, the *pass-through input surrogate column*, of implementation-dependent (UV064) declared type and name, whose value represents all non-partitioning columns of that table argument. The intermediate result row descriptor is correspondingly augmented with one additional column, the *pass-through output surrogate column*, of the same implementation-dependent (UV064) type and name. The PTF fulfill component procedure can copy the value from the pass-through input surrogate column to the pass-through output surrogate column to indicate that all non-partitioning columns of the row of the table argument are copied to the result row. Alternatively, the PTF fulfill component procedure may set the value of the pass-through output surrogate column to the null value to indicate that these output columns are all set to the null value. These non-partitioning columns of the table argument are called the *input pass-through columns* of the table argument; the corresponding columns in the result row are called the *output pass-through columns* of the table argument.

As stated above, each generic table parameter *GTA* has three associated row type descriptors, which are used as follows. The full row type descriptor is populated by the SQL-server prior to invoking the PTF describe routine. The full row type descriptor describes every column of *GTA*. The requested row type descriptor is populated by the PTF describe component procedure to indicate which columns of *GTA* the PTF requests to receive via the PTF cursor for *GTA* during the PTF fulfill component procedure. The PTF cursor row type descriptor describes the PTF cursor for *GTA*, having one column for each column of the requested row type descriptor, plus, if GTA has pass-through columns, one additional column (the pass-through input surrogate column) of implementation-dependent (UV064) declared type and name.

A PTF data area has two result row type descriptors. The initial result row type descriptor describes the proper result columns of the polymorphic table function. There are three cases:

— If the polymorphic table function specifies a <table function column list>, then that specifies the initial result row type.

— If the polymorphic table function specifies RETURNS ONLY PASS THROUGH, then the initial result row type descriptor is empty.

— Otherwise, the initial result row type descriptor is populated by the PTF describe component procedure.

The intermediate result row type descriptor *MRRTD* is a copy of the initial result row type descriptor, plus one additional column for the pass-through output surrogate column of implementation-dependent (UV064) declared type and name for each generic table parameter that has pass-through columns. *MRRTD* is used by the PTF component procedures during the execution phase using a <pipe row statement> to pass a result row to the SQL-server. The SQL-server expands the row received in *MRRTD* to obtain a complete result row by adding the partitioning columns of any partitioned table argument and by replacing each pass-through output surrogate column *PTOSC* with the non-partitioning columns of the row of the table argument represented by the value of *PTOSC*.

The result row type of *PTFI* comprises the following columns:

— The proper result columns, qualified by the <correlation name> of *PTFI*.

— The partitioning columns of *PTFI*, qualified by the range variables of their table arguments.

— The pass-through columns, qualified by the range variables of their table arguments.

The PTF component procedures receive the PTF variables and the PTF extended names as arguments, enabling the PTF component procedures to communicate with one another. The flow of information during a polymorphic table invocation is shown schematically in Figure 5, "Flow of information during the invocation of a polymorphic table function". The diagram does not show the PTF dynamic cursors, which are only open during the execution of the PTF fulfill component procedure on each virtual processor.

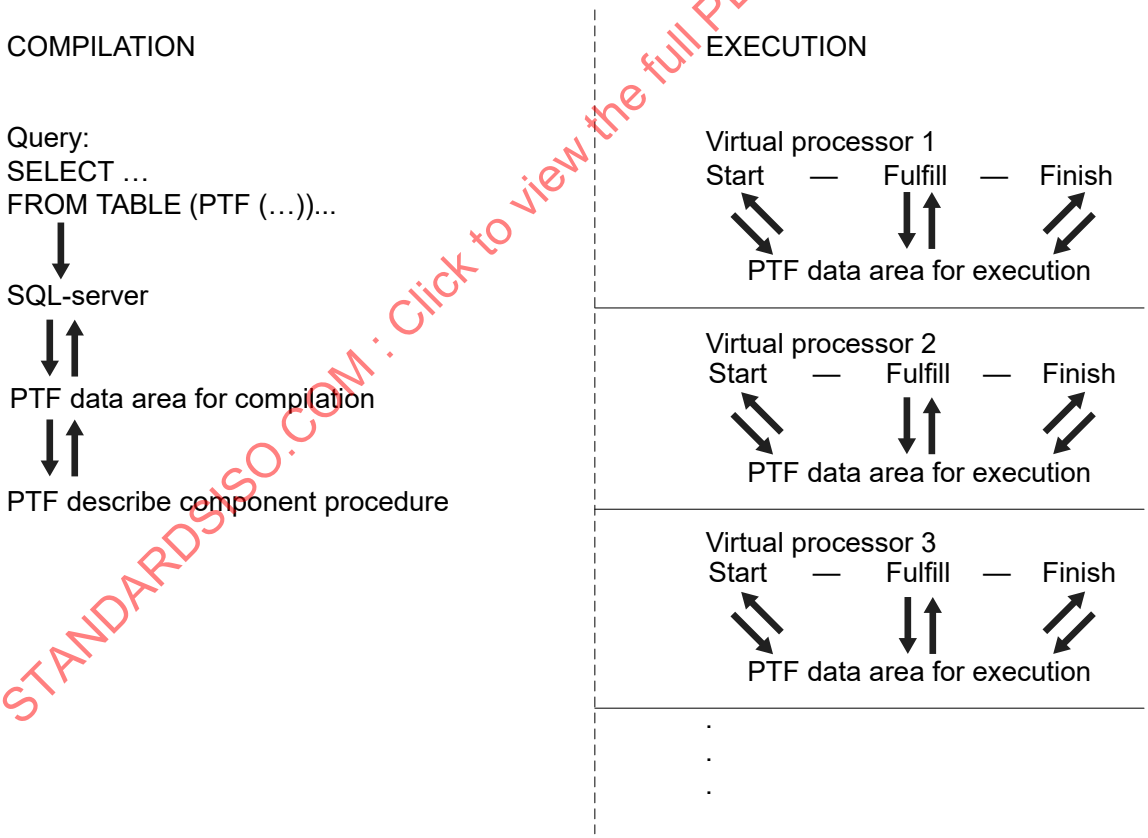

Figure 5 — Flow of information during the invocation of a polymorphic table function

Thus a PTF component procedure may receive information from a previously executed PTF component procedure, treating the private variables as input arguments, and may pass information to later PTF component procedures, treating the private variables as output arguments.

The parameter lists of the PTF component procedures are determined by Subclause 9.26, "Signatures of PTF component procedures". The argument lists to invoke them are constructed in Subclause 9.27, "Invocation of a PTF component procedure". The precise parameter lists and arguments lists of PTF component procedures are mandated if the implementation supports Feature B208, "PTF component procedure interface"; otherwise, the parameter lists and arguments lists are implementation-defined (ID042) but shall be effectively equivalent to those specified in those Subclauses.

The parameter lists of PTF component procedures as specified by the Syntax Rules of Subclause 9.26, "Signatures of PTF component procedures", are illustrated schematically in Table 10, "Schematic diagram of effective parameter lists of PTF component procedures". However, parameters of a PTF component procedure *PTFC* that are derived from parameters of *PTF* are placed in the parameter list of *PTFC* in the order of declaration in the <SQL parameter declaration list> of *PTF* rather than segregated categorically as shown in the second through eighth rows of the table.

Table 10 — Schematic diagram of effective parameter lists of PTF component procedures

	describe	**start**	**fulfill**	**finish**
Private variables	Yes (in/out)	Yes (in/out)	Yes (in/out)	Yes (in/out)
Scalar parameters	Yes (in)	Yes (in)	Yes (in)	Yes (in)
Full row descriptor of a table argument	Name of the descriptor (in)			
Requested row descriptor of a table argument	Name of the descriptor (in)			
Row descriptor of a table argument's cursor		Name of the descriptor (in)	Name of the descriptor (in)	Name of the descriptor (in)
Table argument partitioning descriptor (tables with set semantics only)	Name of the descriptor (in)	Name of the descriptor (in)	Name of the descriptor (in)	Name of the descriptor (in)
Table argument ordering descriptor (tables with set semantics only)	Name of the descriptor (in)	Name of the descriptor (in)	Name of the descriptor (in)	Name of the descriptor (in)
Input table cursor			Name of the input cursor (in)	
Initial result row type descriptor	Name of the descriptor (in)			
Intermediate result row type descriptor		Name of the descriptor (in)	Name of the descriptor (in)	Name of the descriptor (in)
Status variable	Yes (in/out)	Yes (in/out)	Yes (in/out)	Yes (in/out)

The Syntax Rules for an invocation of *PTF* execute the PTF describe component procedure of *PTF* (see the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine"). The purposes of the PTF describe component procedure are:

— To perform any syntactic checks that are particular to the polymorphic table function. (The PTF describe component procedure returns a completion condition other than *successful completion (00000)* to indicate a syntax error.)

— To request columns of each table argument. (The requested columns are made available to the PTF fulfill component procedure via a PTF extended cursor.)

— To initialize the values of the private variables.

— If the polymorphic table function does not specify a result row type and does not specify RETURNS PASS THROUGH ONLY, then to describe the initial row type of the output table of *PTF*.

The PTF describe component procedure is invoked once, with access to SQL descriptor areas that describe the input tables, but without access to the rows of those input tables.

The General Rules for an invocation of *PTF* execute the other PTF component procedures in a prescribed sequence. Execution is performed on one or more virtual processors, each having a context that is a copy of the original SQL-session context. The virtual processors are determined based on the partitioning and co-partitioning of the table arguments, as described in Subclause 9.25, "Execution of an invocation of a polymorphic table function". Each virtual processor has a cursor for each table argument; the cursor may read only a partition of the table argument. There is no communication between virtual processors; each virtual processor operates independently.

On a virtual processor *VP*, the PTF start component procedure, if specified, may perform initialization tasks such as opening files. The PTF start component procedure is optional. If specified, the PTF start component procedure is invoked with the values of the private variables of the PTF that were output by the PTF describe component procedure. The private variables are passed using input/output arguments so the PTF start component procedure may alter the values of the private values in order to communicate with the PTF fulfill component procedure and the PTF finish component procedure on virtual processor *VP*.

After the PTF start component procedure finishes executing on *VP*, a PTF dynamic cursor is opened for each table argument. For each table argument, the row set of a PTF dynamic cursor is limited to just the rows of the partition that has been assigned to *VP*.

Next the PTF fulfill component procedure is invoked on *VP*. The PTF fulfill component procedure may read from the PTF dynamic cursors, and may use the <pipe row statement> to output rows of the result table of *PTF*. The PTF fulfill component procedure receives the private variables as input, as they were output from the PTF start component procedure on *VP* (if specified) or the PTF describe component procedure. The PTF fulfill component procedure may set new values of the private variables to communicate with the PTF finish component procedure on the virtual processor *VP*.

After the PTF fulfill component procedure terminates, the PTF dynamic cursors on *VP* are closed. Finally, the PTF finish component procedure, if specified, is invoked on *VP*. The PTF finish component procedure receives the private variables as they were output by the PTF fulfill component procedure on the same virtual processor *VP*.

The result of an invocation of *PTF* is the multiset of rows produced by <pipe row statement>s that are executed on all virtual processors.

4.35.5 Routine descriptors

This Subclause is modified by Subclause 4.5.3, "Routine descriptors", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.6.1, "Routine descriptors", in ISO/IEC 9075-14.

04 14 An SQL-invoked routine is described by a *routine descriptor*. A routine descriptor includes:

— The routine name of the SQL-invoked routine.

— The specific name of the SQL-invoked routine.

— The routine authorization identifier of the SQL-invoked routine.

— The routine SQL-path of the SQL-invoked routine.

— If the routine is not a polymorphic table function, then the name of the language in which the body of the SQL-invoked routine is written.

— 14For each of the SQL-invoked routine's SQL parameters:

- The <SQL parameter name>, if it is specified.
- The <parameter type>.
- The ordinal position.
- An indication of whether the SQL parameter is an input SQL parameter, an output SQL parameter, or both an input SQL parameter and an output SQL parameter.
- An indication of whether the SQL parameter has a default value, and if so the <parameter default>.

— An indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.

— If the SQL-invoked routine is an SQL-invoked procedure, then the maximum number of returned result sets.

— An indication of whether the SQL-invoked routine is deterministic or possibly non-deterministic.

— Indications of whether the SQL-invoked routine possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

— 14If the SQL-invoked routine is an SQL-invoked function, then:

- If the <returns type> contains <returns table type>, then the <returns table type>.
- If the <returns type> contains <returns data type>, then:
  - The <returns data type> of the SQL-invoked function.
  - If the <returns data type> simply contains <locator indication>, then an indication that the return value is a locator.
- An indication of whether the SQL-invoked function is a type-preserving function or not.
- An indication of whether the SQL-invoked function is a mutator function or not.
- If the SQL-invoked function is a type-preserving function, then an indication of which parameter is the result parameter.
- An indication of whether the SQL-invoked function is a null-call function.
- An indication of whether the SQL-invoked function is an SQL-invoked method.
- An indication of whether the SQL-invoked function is a polymorphic table function.
- If the SQL-invoked function is a polymorphic table function, then:
  - For each of the polymorphic table function's private parameters:
    - The <SQL parameter name>.

- The <parameter type>.
- The ordinal position.
- An indication of whether the private parameter has a default value, and if so, the <parameter default>.

— For each of the polymorphic table function's generic table parameters:

- The <pass through option>.
- The <generic table semantics>.

— The PTF describe component procedure, if specified.

— The PTF start component procedure, if specified.

— The PTF fulfill component procedure.

— The PTF finish component procedure, if specified.

— The creation timestamp.

— The last-altered timestamp.

— If the SQL-invoked routine is an SQL routine, then:

- The SQL routine body of the SQL-invoked routine.
- The SQL security characteristic of the SQL routine.

— [14] If the SQL-invoked routine is an external routine, then:

- The external routine name of the external routine.
- The <parameter style> of the external routine.
- If the external routine specifies a <result cast>, then an indication that it specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then an indication that the <data type> specified in the <result cast> has a locator indication.
- The external security characteristic of the external routine.
- The external routine SQL-path of the external routine.
- The effective SQL parameter list of the external routine.
- For every SQL parameter that has an associated from-sql function *FSF*, the specific name of *FSF*.
- For every SQL parameter that has an associated to-sql function *TSF*, the specific name of *TSF*.
- If the SQL-invoked routine is an external function and if it has a to-sql function *TRF* associated with the result, then the specific name of *TRF*.
- For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, an indication that the SQL parameter is a locator parameter.

— The schema name of the schema that includes the SQL-invoked routine.

— If the SQL-invoked routine is an SQL-invoked method, then:

- The name of the user-defined type whose descriptor contains the corresponding method specification descriptor.

- An indication of whether STATIC was specified.

— 04 An indication of whether the routine is a schema-level routine.

— An indication of whether the SQL-invoked routine is dependent on a user-defined type.

— An indication as to whether or not the SQL-invoked routine requires a new savepoint level to be established when it is invoked.

4.35.6 Result sets returned by SQL-invoked procedures

This Subclause is modified by Subclause 4.6.1, "Result sets returned by SQL-invoked procedures", in ISO/IEC 9075-3.

An invocation of an SQL-invoked procedure *SIP1* might bring into existence a result set sequence *RSS*. *RSS* consists of the result sets of with-return cursors opened by *SIP1* and remaining open when *SIP1* terminates, placed in the order in which those result sets are created during the execution of *SIP1*.

NOTE 80 — If the same cursor is opened more than once during the execution of *SIP1*, then it is the last opening that is considered to create the result set, even if the result set in question is identical to that created by some earlier opening.

RSS is available through a received cursor, by executing an <allocate received cursor statement>, to the invoker *INV* of *SIP1*. If *SIP1* is invoked by an SQL-invoked procedure *SIP2*, then *INV* is *SIP2*. If *SIP1* is invoked by an externally-invoked procedure *EIP*, then *INV* is the SQL-client module containing *EIP*. Whether *RSS* may itself be returned to the invoker *INV* is implementation-defined (IV031).

NOTE 81 — For example, if an externally-invoked procedure *EIP* executes a <call statement> invoking an SQL-invoked procedure *SIP3* that invokes *SIP1*, then the result set sequence *RSS* returned by *SIP1* is available to *SIP3*, until either *SIP3* returns control to *EIP* or another invocation of *SIP1* by *SIP3* is given before *SIP3* returns. Whether *SIP3* returns any particular cursor *C* in *RSS* to *EIP* is implementation-defined.

The invocation of *SIP1* by *INV* destroys any existing result set sequence that might have arisen from some previous invocation of *SIP1* by *INV*. All result set sequences available to *INV* are destroyed when *INV* terminates.

A returned result set includes a sequence of rows called the *constituent rows* and an *initial cursor position*. All rows of a result set are of the same row type, called the *row type* of the result set. The *degree* of a result set is the number of columns in the result set's row type. The number of rows in the result set is its *cardinality*.

The constituent rows and initial cursor position of each returned result set *RS* in *RSS* are determined when *SIP1* returns to *INV*. If the with-return cursor *C* for *RS* is scrollable, then the constituent rows of *RS* are those of the result set of *C* as it exists when *SIP1* terminates; otherwise, the constituent rows are as for scrollable cursors, except that rows preceding the current cursor position of *C* are excluded. If *C* is scrollable, then the initial cursor position of *RS* is the position of *C* when *SIP1* terminates; otherwise, the initial cursor position of *RS* is before the first row.

NOTE 82 — The result set of *C* as it exists when *SIP1* terminates might differ from that generated when *C* was opened, if, for example, any <delete statement: positioned>s or <update statement: positioned>s are executed by *SIP1* before it terminates.

The maximum number of returned result sets that may form a result set sequence is specified by the <returned result sets characteristic> contained in the <SQL-invoked routine> defining *SIP1*. If the actual number of with-return cursors that remain open when *SIP1* returns is greater than the maximum number of returned result sets specified in the <returned result sets characteristic> clause, then a warning condition is raised. It is implementation-dependent (UA045) whether or not result sets whose positions are greater than that maximum number are returned.

03 A result set is described by a result set descriptor. A *result set descriptor* includes:

— A <cursor specification>.

— A sequence of rows.

— A position within the sequence of rows (before a row, on a row, or after the last row).

— The operational properties:

  • The operational sensitivity property (either SENSITIVE, INSENSITIVE, or ASENSITIVE).

  • The operational scrollability property (either SCROLL or NO SCROLL).

  • The operational holdability property (either WITH HOLD or WITHOUT HOLD).

  • The operational returnability property (either WITH RETURN or WITHOUT RETURN).

— If the operational holdability property is WITH HOLD and the position is on a row, then an indication whether a <fetch statement> or <dynamic fetch statement> has been issued since the last <commit statement>.

4.36 SQL-paths

This Subclause is modified by Subclause 4.6, "SQL-paths", in ISO/IEC 9075-4.

An SQL-path is a list of one or more <schema name>s that determines the search order for one of the following:

— The subject routine of a <routine invocation> whose <routine name> does not contain a <schema name>.

— The user-defined type when the <path-resolved user-defined type name> does not contain a <schema name>.

The value specified by CURRENT_PATH is the value of the SQL-path of the current SQL-session. This SQL-path is used to search for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> when the <routine invocation> is contained in <preparable statement>s that are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement>, or contained in <direct SQL statement>s that are invoked directly.

04 The definition of SQL-schemas specifies an SQL-path that is used to search for the subject routine of a <routine invocation> whose <routine name>s do not contain a <schema name> when the <routine invocation> is contained in the <schema definition>.

4.37 Host parameters

This Subclause is modified by Subclause 4.7, "Host parameters", in ISO/IEC 9075-4.

4.37.1 Overview of host parameters

A host parameter is declared in an <externally-invoked procedure> by a <host parameter declaration>. A host parameter either assumes or supplies the value of the corresponding argument in the invocation of the <externally-invoked procedure>.

A <host parameter declaration> specifies the <data type> of its value, which maps to the host language type of its corresponding argument. Host parameters cannot be null, except through the use of indicator parameters.

4.37.2 Status parameters

This Subclause is modified by Subclause 4.7.1, "Status parameters", in ISO/IEC 9075-4.

The SQLSTATE host parameter is a status parameter. It is set to status codes that indicate either that a call of the <externally-invoked procedure> completed successfully or that an exception condition was raised during execution of the <externally-invoked procedure>.

An <externally-invoked procedure> shall specify the SQLSTATE host parameter. The SQLSTATE host parameter is a character string host parameter for which exception values are defined in Clause 24, "Status codes".

If a condition is raised that causes a statement to have no effect other than that associated with raising the condition (that is, not a completion condition), then the condition is said to be an *exception condition* or *exception*. If a condition is raised that permits a statement to have an effect other than that associated with raising the condition (corresponding to an SQLSTATE class code of *successful completion (00000)*, *warning (01000)*, or *no data (02000)*), then the condition is said to be a *completion condition*.

Exception conditions or completion conditions may be raised during the execution of an <SQL procedure statement>. One of the conditions becomes the *active condition* when the <SQL procedure statement> terminates; the *active condition* is the condition returned in SQLSTATE. If the active condition is an exception condition, then it will be called the *active exception condition*. If the active condition is a completion condition, then it will be called the *active completion condition*.

The completion condition *warning (01000)* is broadly defined as completion in which the effects are correct, but there is reason to caution the user about those effects. It is raised for implementation-defined (IC011) conditions as well as conditions specified in this document. The completion condition *no data (02000)* has special significance and is used to indicate an empty result. The completion condition *successful completion (00000)* is defined to indicate a completion condition that does not correspond to *warning (01000)* or *no data (02000)*. This includes conditions in which the SQLSTATE subclass provides implementation-defined (IC012) information of a non-cautionary nature.

04 For the purpose of choosing status parameter values to be returned, exception conditions for transaction rollback have precedence over exception conditions for statement failure. Similarly, the completion condition *no data (02000)* has precedence over the completion condition *warning (01000)*, which has precedence over the completion condition *successful completion (00000)*. All exception conditions have precedence over all completion conditions. The values assigned to SQLSTATE shall obey these precedence requirements.

4.37.3 Data parameters

A data parameter is a host parameter that is used to either assume or supply the value of data exchanged between a host program and an SQL-implementation.

4.37.4 Indicator parameters

An indicator parameter is an integer host parameter that is specified immediately following another host parameter. Its primary use is to indicate whether the value that the other host parameter assumes or supplies is a null value. An indicator host parameter cannot immediately follow another indicator host parameter.

The other use for indicator parameters is to indicate whether string data truncation occurred during a transfer between a host program and an SQL-implementation in host parameters or host variables. If a non-null string value is transferred and the length of the target is sufficient to accept the entire source value, then the indicator parameter or variable is set to 0 (zero) to indicate that truncation did not occur. However, if the length of the target is insufficient, the indicator parameter or variable is set to the length (in characters or octets, as appropriate) of the source value to indicate that truncation occurred and to indicate original length in characters or octets, as appropriate, of the source.

4.37.5 Locators

A host parameter, a host variable, an SQL parameter of an external routine, or the value returned by an external function may be specified to be a *locator* by specifying AS LOCATOR. A locator is an SQL-session object, rather than SQL-data, that can be used to reference an SQL-data instance. A locator is either a large object locator, a user-defined type locator, an array locator, or a multiset locator.

A large object locator is one of the following:

— Binary large object locator, a value of which identifies a binary large object string.

— Character large object locator, a value of which identifies a large object character string.

— National character large object locator, a value of which identifies a national large object character string.

A user-defined type locator identifies a value of the user-defined type specified by the locator specification. An array locator identifies a value of the array type specified by the locator specification. A multiset locator identifies a value of the multiset type specified by the locator specification.

When the value at a site of binary large object type, character large object type, user-defined type, array type, or multiset type is to be assigned to a locator of the corresponding type, an implementation-dependent (UV065) four-octet integer value is generated and assigned to the target. A locator value uniquely identifies a value of the corresponding type.

A locator may be either *valid* or *invalid*. A host parameter or host variable specified as a locator may be further specified to be a *holdable locator*. When a locator is initially created, it is marked valid and, if applicable, not holdable. A <hold locator statement> identifying the locator shall be specifically executed before the end of the SQL-transaction in which it was created in order to make that locator holdable.

A non-holdable locator remains valid until the end of the SQL-transaction in which it was generated, unless it is explicitly made invalid. A non-holdable locator becomes invalid whenever a <free locator statement> is executed or whenever a <rollback statement> that specifies a <savepoint clause> identifying a savepoint established prior to the generation of the locator is executed. Accessing an invalid locator causes an exception condition to be raised.

A holdable locator may remain valid beyond the end of the SQL-transaction in which it is generated. A holdable locator becomes invalid whenever a <free locator statement> identifying that locator is executed or the SQL-transaction in which it is generated or any subsequent SQL-transaction is rolled back. All locators remaining valid at the end of an SQL-session are marked invalid when that SQL-session terminates.

4.38 Diagnostics area

This Subclause is modified by Subclause 4.8, "Diagnostics area", in ISO/IEC 9075-4.

A diagnostics area is a place where completion and exception condition information is stored when an SQL-statement is executed. The diagnostics areas associated with an SQL-session form the *diagnostics area stack* of that SQL-session. For definitional purposes, the diagnostics areas in this stack are considered to be numbered sequentially beginning with 1 (one). An additional diagnostics area is maintained by the SQL-client, as described in Subclause 4.4.3.2, "SQL-clients", in ISO/IEC 9075-1.

Two operations on diagnostics area stacks are specified in this document for definitional purposes only. *Pushing* a diagnostics area stack effectively creates a new first diagnostics area, incrementing the ordinal position of every existing diagnostics area in the stack by 1 (one). The content of the new first diagnostics area is initially a copy of the content of the old (now second) one. *Popping* a diagnostics area stack effectively destroys the first diagnostics area in the stack and decrements the ordinal position of every remaining diagnostics area by 1 (one). The maximum number of diagnostics areas in a diagnostics area stack is implementation-dependent (UL002).

Each diagnostics area consists of a *statement area* and a sequence of one or more *condition areas*, each of which is at any particular time either *occupied* or *vacant*. A diagnostics area is *empty* when each of its condition areas is vacant; *emptying* a diagnostics area brings about this state. A statement area consists of a collection of named *statement information items*. A condition area consists of a collection of named *condition information items*.

A statement information item gives information about the innermost SQL-statement that is being executed when a condition is raised. A condition information item gives information about the condition itself. The names and data types of statement and condition information items are given in Table 37, "Data types of <statement information item name>s", and in Table 38, "Data types of <condition information item name>s". Their meanings are given by the General Rules of Subclause 23.1, "<get diagnostics statement>".

04 At the beginning of the execution of any <SQL procedure statement> that is not an <SQL diagnostics statement>, the first diagnostics area is emptied. An SQL-implementation places information about a completion condition or an exception condition reported by SQLSTATE into a vacant condition area in this diagnostics area. If other conditions are raised, the extent to which these cause further condition areas to become occupied is implementation-defined (IA208).

An <externally-invoked procedure> containing an <SQL diagnostics statement> returns a code indicating a completion or an exception condition for that statement via SQLSTATE, but does not necessarily cause any vacant condition areas to become occupied.

The number of condition areas per diagnostics area is referred to as the *condition area limit*. An SQL-agent may set the condition area limit with the <set transaction statement>; if the SQL-agent does not specify the condition area limit, then the condition area limit is implementation-dependent (UL003), but shall be at least one condition area. An SQL-implementation may place information into this area about fewer conditions than there are condition areas. The ordering of the information about conditions placed into a diagnostics area is implementation-dependent (US030), except that the first condition area in a diagnostics area always corresponds to the condition specified by the SQLSTATE value.

The <get diagnostics statement> is used to obtain information from an occupied condition area, referenced by its ordinal position within the first diagnostics area.

4.39 Host languages

This document specifies the actions of <externally-invoked procedure>s in SQL-client modules when those <externally-invoked procedure>s are called by programs written in certain specified programming languages, called *host language*s. The term "*PLN* program", where *PLN* is the name of a host language, refers to a program that conforms to the specification for that host language as identified in Clause 2, "Normative references".

This document specifies a mechanism whereby SQL language may be embedded in programs that otherwise conform to any of the same specified programming language standards.

NOTE 83 — Interfaces between SQL and the Java programming language are defined in ISO/IEC 9075-10 and ISO/IEC 9075-13.

Although there are obvious mappings between many SQL data types and the data types of most host languages, this is not the case for all SQL data types or for all host languages.

For the purposes of interfacing with programming languages, the data types DATE, TIME, TIMESTAMP, and INTERVAL shall be converted to or from character strings in those programming languages by means of a <cast specification>. It is anticipated that future evolution of programming language standards will support data types corresponding to these four SQL data types; this document will then be revised to reflect the availability of those corresponding data types.

The data types CHARACTER, CHARACTER VARYING, and CHARACTER LARGE OBJECT are also mapped to character strings in the programming languages. However, because the facilities available in the programming languages do not provide the same capabilities as those available in SQL, there shall be agreement between the host program and SQL regarding the specific format of the character data being exchanged. Specific syntax for this agreement is provided in this document.

For standard programming languages C and COBOL, BOOLEAN values are mapped to integer variables in the host language. For standard programming languages Ada, Fortran, Pascal, and PL/I, BOOLEAN variables are directly supported.

For the purposes of interfacing with programming languages, the data type ARRAY shall be converted to a locator (see Subclause 4.37.5, "Locators").

For the purposes of interfacing with programming languages, the data type MULTISET shall be converted to a locator (see Subclause 4.37.5, "Locators").

For the purposes of interfacing with programming languages, user-defined types shall be handled with a locator (see Subclause 4.37.5, "Locators") or transformed to another SQL data type that has a defined mapping to the host language (see Subclause 4.9.6, "Transforms for user-defined types").

4.40 Cursors

This Subclause is modified by Subclause 4.7, "Cursors", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 4.10, "Cursors", in ISO/IEC 9075-4.

4.40.1 General description of cursors

This Subclause is modified by Subclause 4.7.1, "General description of cursors", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 4.10.1, "General description of cursors", in ISO/IEC 9075-4.

A cursor is a mechanism by which the rows of a table may be acted on (e.g., returned to a host programming language) one at a time.

04 A cursor is specified by a <declare cursor>, a <dynamic declare cursor>, an <allocate extended dynamic cursor statement>, an <allocate received cursor statement>, or implicitly by a <table argument> of an invocation of a polymorphic table function. A cursor specified by a <declare cursor> is a *standing cursor*. A cursor specified by a <dynamic declare cursor> is a *declared dynamic cursor*. A cursor specified by an <allocate extended dynamic cursor statement> is an *extended dynamic cursor*. An extended dynamic cursor is said to be *global* if its extended name is global; otherwise, the extended dynamic cursor is *local*. A cursor specified by an <allocate received cursor statement> is a *received cursor*. A cursor specified implicitly by a <table argument> is a *PTF dynamic cursor*. A *dynamic cursor* is either a declared dynamic cursor, an extended dynamic cursor, a received cursor, or a PTF dynamic cursor.

03 04 A *declared cursor* is either a standing cursor, a declared dynamic cursor, or a received cursor. A declared cursor has a <cursor name>. A <declare cursor>, a <dynamic declare cursor>, or an <allocate received cursor statement> is immediately contained in the <module contents> of an <SQL-client module definition>. The scope of a <cursor name> is the innermost <SQL-client module definition> *M* that contains it, with the exception of any <SQL schema statement>s contained in *M*.

Every cursor has an associated cursor declaration. For a declared cursor, the cursor declaration is specified by the cursor's <declare cursor>, <dynamic declare cursor>, or <allocate received cursor statement>. For an extended dynamic cursor, the cursor declaration is implicitly created by an <allocate extended dynamic cursor statement>. For a PTF dynamic cursor, the cursor declaration is implicitly specified by a <table argument>.

03 04 A cursor declaration is described by a cursor declaration descriptor. A *cursor declaration descriptor* includes:

— 03 The kind of cursor:
  - Standing.
  - Declared dynamic.
  - Extended dynamic.

  - Received.
  - PTF dynamic.

— 04 The provenance of the cursor:

  - 04 If the cursor is a declared cursor or a local extended dynamic cursor, then an indication of an SQL-client module.
  - If the cursor is a PTF dynamic cursor, then a <routine invocation> that invokes a polymorphic table function.
  - Otherwise, an SQL-session identifier.

— 03 The name of the cursor:

  - If the cursor is a declared cursor, then a <cursor name>.
  - 03 Otherwise, an extended name and its scope (GLOBAL, LOCAL, or PTF).

— 03 The cursor's origin:

  - If the cursor is a standing cursor, then the <cursor specification> contained in the <declare cursor>.
  - If the cursor is a declared dynamic cursor, then the <statement name> contained in the <dynamic declare cursor>.
  - If the cursor is an extended dynamic cursor, then the prepared statement specified by the extended statement name.
  - If the cursor is a PTF dynamic cursor, then the <cursor specification> of a <table argument> on an executing virtual processor.
  - 03 If the cursor is a received cursor, then the cursor's <specific routine designator>.

— The cursor's declared properties:

  - The cursor's declared sensitivity property (either SENSITIVE, INSENSITIVE, or ASENSITIVE).
  - The cursor's declared scrollability property (either SCROLL or NO SCROLL).
  - The cursor's declared holdability property (either WITH HOLD or WITHOUT HOLD).
  - The cursor's declared returnability property (either WITH RETURN or WITHOUT RETURN).

A cursor declaration descriptor is identified by the combination of its kind, its provenance, and its name.

A cursor is described by a cursor instance descriptor. A *cursor instance descriptor* includes:

— A cursor declaration descriptor.

— An SQL-session identifier.

— The cursor's state (either open or closed).

— If the cursor is open, then the cursor's result set descriptor.

A cursor instance descriptor is identified by its cursor declaration descriptor and its SQL-session identifier.

The term "cursor", unqualified by either "declaration" or "instance", refers to a cursor instance descriptor.

04 For every <declare cursor> in an <SQL-client module definition>, a cursor declaration descriptor is included in the SQL-client module, and is effectively created and destroyed at the same time as the SQL-client module. A cursor instance descriptor for the standing cursor is effectively created when an

externally-invoked procedure of that SQL-client module is invoked. The cursor instance descriptor is destroyed at the end of the SQL-session.

For every <dynamic declare cursor> in an <SQL-client module definition>, a cursor declaration descriptor is included in the SQL-client module, and is effectively created and destroyed at the same time as the SQL-client module. A cursor instance descriptor for the declared dynamic cursor is effectively created when an externally-invoked procedure of that SQL-client module is invoked, and destroyed when a <deallocate prepared statement> is executed that deallocates the prepared statement on which the declared dynamic cursor is based. The cursor instance descriptor is also destroyed at the end of the SQL-session (if not earlier).

An extended dynamic cursor is effectively created (i.e., both the cursor declaration descriptor and the cursor instance descriptor) when an <allocate extended dynamic cursor statement> is executed within an SQL-session and destroyed when that SQL-session is terminated or when a <deallocate prepared statement> is executed that deallocates the prepared statement on which the extended dynamic cursor is based.

A PTF dynamic cursor is effectively created and destroyed (i.e., both the cursor declaration descriptor and the cursor instance descriptor are created and destroyed) by the General Rules of Subclause 9.25, "Execution of an invocation of a polymorphic table function".

For every <allocate received cursor statement> in an <SQL-client module definition>, a cursor declaration descriptor is included in the SQL-client module, and is effectively created and destroyed at the same time as the SQL-client module. A cursor instance descriptor for the received cursor is effectively created when an <allocate received cursor statement> is executed within an SQL-session and destroyed when that SQL-session is terminated.

A cursor instance descriptor is in either the open state or the closed state. The initial state of a cursor is the closed state. A standing cursor is placed in the open state by an <open statement> and returned to the closed state by a <close statement> or a <rollback statement>. A declared dynamic cursor is placed in the open state by a <dynamic open statement> and returned to the closed state by a <dynamic close statement>. An extended dynamic cursor is placed in the open state by a <dynamic open statement> and returned to the closed state by a <dynamic close statement>. A PTF dynamic cursor is opened and closed by the General Rules of Subclause 9.25, "Execution of an invocation of a polymorphic table function". A received cursor is placed in the open state by an <allocate received cursor statement>, and is advanced to the next result set in the result set sequence by a <dynamic close statement>. After the last result set in the result set sequence has been processed, the received cursor is placed in the closed state by a <dynamic close statement>. A received cursor is also placed in the closed state if the associated SQL-invoked procedure is invoked again while the received cursor is open. An open cursor that was not defined as a holdable cursor is also closed by a <commit statement>. (In the case of a received cursor, if the current result set of the received cursor is not holdable, then a <commit statement> causes the received cursor to open on the next result set in the result set sequence, if any.)

A cursor in the open state identifies a result set and a position relative to the ordering of that result set. If the cursor's <cursor specification> does not simply contain an <order by clause>, or simply contains an <order by clause> that does not specify the order of the rows completely, then the rows of the result set have an order that is defined only to the extent that the <order by clause> specifies an order and is otherwise implementation-dependent (US031).

NOTE 84 — A definition of "result set" is given in Clause 3, "Terms and definitions".

When the ordering of a cursor is not defined by an <order by clause>, the relative position of two rows is implementation-dependent (US031). When the ordering of a cursor is partially determined by an <order by clause>, then the relative positions of two rows are determined only by the <order by clause>; if the two rows have equal values for the purpose of evaluating the <order by clause>, then their relative positions are implementation-dependent (US031).

A cursor whose result set's <cursor specification> simply contains an <order by clause> is an *ordered cursor*.

4.40 Cursors

A cursor is either *updatable* or *not updatable*. A PTF dynamic cursor is not updatable. A received cursor is not updatable. For other kinds of cursors, if FOR UPDATE OF is contained in the <cursor specification> of the result set descriptor of the cursor, or if the <cursor specification> *CS* identified by the cursor is a <query specification> that is simply updatable, FOR READ ONLY is not contained in *CS*, the cursor is not ordered, and the cursor is not scrollable, then the cursor is *updatable*; otherwise, the cursor is *not updatable*. The operations of update and delete are permitted for updatable cursors, subject to constraining Access Rules.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If a cursor is on a row, then that row is the current row of the cursor. A cursor may be before the first row or after the last row of a result set even though the result set's sequence of rows is empty. When a cursor is initially opened, the position of the cursor is before the first row.

A cursor declaration descriptor and a result set descriptor have four properties: the sensitivity property (either SENSITIVE, INSENSITIVE, or ASENSITIVE), the scrollability property (either SCROLL or NO SCROLL), the holdability property (either WITH HOLD or WITHOUT HOLD), and the returnability property (either WITH RETURN or WITHOUT RETURN). The four declared cursor properties are set to the explicit or implicit <cursor sensitivity>, <cursor scrollability>, <cursor holdability>, and <cursor returnability>, respectively, of the <cursor properties> contained in the <declare cursor>, <dynamic declare cursor>, or <allocate extended dynamic cursor statement>; for a PTF cursor or a received cursor, the declared cursor properties are ASENSITIVE, NO SCROLL, WITHOUT HOLD, and WITHOUT RETURN.

When a cursor is opened, the four declared cursor properties are copied to the result set descriptor's operational properties. In the case of a declared dynamic cursor or an extended dynamic cursor, the operational properties are modified as specified by the explicit <cursor attributes> of the prepared statement that is the <cursor specification> of the cursor. In the case of a received cursor, the operational sensitivity property, the operational scrollability property, and the operational holdability property are preserved from their values as established by the SQL-invoked procedure that created the result set, and the operational returnability property is implementation-defined (IV031).

A cursor whose operational sensitivity property is SENSITIVE is said to be *sensitive*; a cursor whose operational sensitivity property is INSENSITIVE is said to be *insensitive*; and a cursor whose operational sensitivity property is ASENSITIVE is said to be *asensitive*.

A cursor whose operational scrollability property is SCROLL is said to be *scrollable*.

A cursor whose operational holdability property is WITH HOLD is said to be *holdable*.

A cursor whose operational returnability property is WITH RETURN is said to be a *with-return* cursor.

A holdable cursor that has been held open retains its position when the new SQL-transaction is initiated. However, if the holdable cursor is a standing cursor, then before either an <update statement: positioned> or a <delete statement: positioned> is permitted to reference that standing cursor in the new SQL-transaction, a <fetch statement> must be issued against the standing cursor (otherwise, an exception condition is raised). If the holdable cursor is a dynamic cursor, then before a <dynamic update statement: positioned>, a <dynamic delete statement: positioned>, a <preparable dynamic update statement: positioned>, or a <preparable dynamic delete statement: positioned> is permitted to reference that dynamic cursor in the new SQL-transaction, a <dynamic fetch statement> must be issued against the dynamic cursor (otherwise an exception condition is raised).

The following aspects of a standing cursor can be determined from the <cursor specification> in its cursor declaration descriptor: whether it is ordered, whether it is updatable, the leaf underlying tables, the leaf generally underlying tables, its sensitivity property, its scrollability property, its holdability property, and its returnability property. Accordingly, terminology regarding these aspects of a standing cursor may also be applied to the cursor declaration descriptor, for example, in the Syntax Rules, Access Rules, and Conformance Rules.

4.40.2 Operations on and using cursors

A <fetch statement> positions an open standing cursor on a specified row of the standing cursor's ordering and retrieves the values of the columns of that row. An <update statement: positioned> updates the current row of the standing cursor. A <delete statement: positioned> deletes the current row of the standing cursor.

A <dynamic fetch statement> positions an open dynamic cursor on a specified row of the dynamic cursor's ordering and retrieves the values of the columns of that row. A <dynamic update statement: positioned> updates the current row of the dynamic cursor. A <dynamic delete statement: positioned> deletes the current row of the dynamic cursor. A <preparable dynamic delete statement: positioned> is used to delete rows through a dynamic cursor when the precise format of the statement is not known until runtime. A <preparable dynamic update statement: positioned> is used to update rows through a dynamic cursor when the precise format of the statement is not known until runtime.

A *positioned delete statement* is an SQL-statement that is a <delete statement: positioned>, a <dynamic delete statement: positioned>, or a <preparable dynamic delete statement: positioned>.

A *positioned update statement* is an SQL-statement that is an <update statement: positioned>, a <dynamic update statement: positioned>, or a <preparable dynamic update statement: positioned>.

If an error occurs during the execution of an SQL-statement that identifies a cursor, then, except where otherwise explicitly defined, the effect, if any, on the position or state of that cursor is implementation-dependent (UA046).

If a completion condition is raised during the execution of an SQL-statement that identifies a cursor, then the particular SQL-statement identifying that open cursor on which the completion condition is returned is implementation-dependent (UA047).

The following paragraphs define several terms used to discuss issues relating to a cursor's operational sensitivity property:

A change to SQL-data is said to be *independent* of a cursor *CR* if and only if it is not made by an <update statement: positioned> or a <delete statement: positioned> that is positioned on *CR*.

A change to SQL-data is said to be *significant* to *CR* if and only if it is independent of *CR*, and, had it been committed before *CR* was opened, would have caused the sequence of rows in the result set descriptor of *CR* to be different in any respect.

A change to SQL-data is said to be *visible* to *CR* if and only if it has an effect on the sequence of rows *SR* of the result set descriptor of *CR* by inserting a row in *SR*, deleting a row from *SR*, changing the value of a column of a row of *SR*, or reordering the rows of *SR*.

If a cursor is open, and the SQL-transaction in which the cursor was opened makes a significant change to SQL-data, then whether that change is visible through that cursor before it is closed is determined as follows:

— If the cursor is insensitive, then significant changes are not visible.

— If the cursor is sensitive, then significant changes are visible.

— If the cursor is asensitive, then the visibility of significant changes is implementation-dependent (UA048).

If a holdable cursor is open during an SQL-transaction *T* and it is held open for a subsequent SQL-transaction, then whether any significant changes made to SQL-data (by *T* or any subsequent SQL-transaction in which the cursor is held open) are visible through that cursor in the subsequent SQL-transaction before that cursor is closed is determined as follows:

— If the cursor is insensitive, then significant changes are not visible.

— If the cursor is sensitive, then the visibility of significant changes is implementation-defined (IA209).

— If the cursor is asensitive, then the visibility of significant changes is implementation-dependent (UA048).

It is implementation-dependent (UA069) whether or not a holdable cursor *HC* that is held over a COMMIT at time *CT* in one SQL-session sees changes that have been committed in another in another SQL-session between the time that *HC* was opened and *CT*.

If *CR* is a with-return cursor, then the <cursor specification> included in the result set descriptor of *CR* defines a returned result set. A with-return cursor, if declared in an SQL-invoked procedure and in the open state when the procedure returns to its invoker, yields a returned result set that can be accessed by the invoker of the procedure that generates it.

NOTE 85 — Definitions of "returned result set" and "with-return cursor" are given in Clause 3, "Terms and definitions".

4.41 SQL-statements

This Subclause is modified by Subclause 4.11, "SQL-statements", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.7, "SQL-statements", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.7, "SQL-statements", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.3, "SQL-statements", in ISO/IEC 9075-16.

4.41.1 Classes of SQL-statements

This Subclause is modified by Subclause 4.11.1, "Classes of SQL-statements", in ISO/IEC 9075-4.

An SQL-statement is a string of characters that conforms to the Format and Syntax Rules specified in the ISO/IEC 9075 series. Most SQL-statements can be prepared for execution and executed in an SQL-client module, in which case they are prepared when the SQL-client module is created and executed when the containing externally-invoked procedure is called (see Subclause 4.30, "SQL-client modules").

Most SQL-statements can be prepared for execution and executed in additional ways. These are:

— In an embedded SQL host program, in which case they are prepared when the embedded SQL host program is preprocessed (see Subclause 4.31, "Embedded syntax").

— Being prepared and executed by the use of SQL-dynamic statements (which are themselves executed in an SQL-client module or an embedded SQL host program—see Subclause 4.32, "Dynamic SQL concepts").

— Direct invocation in which case they are effectively prepared immediately prior to execution (see Subclause 4.33, "Direct invocation of SQL").

In this document, there are at least six ways of classifying SQL-statements:

— According to their effect on SQL objects, whether persistent objects, i.e., SQL-data, SQL-client modules, and schemas, or transient objects, such as SQL-sessions and other SQL-statements.

— According to whether or not they start an SQL-transaction, or can, or shall, be executed when no SQL-transaction is active.

— According to whether they possibly read SQL-data or possibly modify SQL-data.

— According to whether or not they may be embedded.

— According to whether they may be dynamically prepared and executed.

— According to whether or not they may be directly executed.

This document permits SQL-implementations to provide additional, implementation-defined (IE003), statements that may fall into any of these categories. This Subclause will not mention those statements again, as their classification is implementation-defined (IE003).

04 The main classes of SQL-statements are:

— SQL-schema statements.

— SQL-data statements.

— SQL-transaction statements.

— SQL-control statements.

— SQL-connection statements.

— SQL-session statements.

— SQL-diagnostics statements.

— SQL-dynamic statements.

— SQL embedded exception declaration.

4.41.2 SQL-statements classified by function

This Subclause is modified by Subclause 4.11.2, "SQL-statements classified by function", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.7.1, "SQL-statements classified by function", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.7.1, "SQL-statements classified by function", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.3.1, "SQL-statements classified by function", in ISO/IEC 9075-16.

4.41.2.1 SQL-schema statements

This Subclause is modified by Subclause 4.11.2.1, "SQL-schema statements", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.7.1.1, "SQL-schema statements", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.3.1.1, "SQL-schema statements", in ISO/IEC 9075-16.

04 09 16 The following are the SQL-schema statements:

— <schema definition>.

— <drop schema statement>.

— <domain definition>.

— <drop domain statement>.

— <table definition>.

— <drop table statement>.

— <view definition>.

— <drop view statement>.

— <assertion definition>.

— <drop assertion statement>.

— <alter table statement>.

— <alter domain statement>.

— <grant privilege statement>.

— <revoke statement>.
— <character set definition>.
— <drop character set statement>.
— <collation definition>.
— <drop collation statement>.
— <transliteration definition>.
— <drop transliteration statement>.
— <trigger definition>.
— <drop trigger statement>.
— <user-defined type definition>.
— <alter type statement>.
— <drop data type statement>.
— <user-defined ordering definition>.
— <drop user-defined ordering statement>.
— <user-defined cast definition>.
— <drop user-defined cast statement>.
— <transform definition>.
— <alter transform statement>.
— <drop transform statement>.
— <schema routine>.
— <alter routine statement>.
— <drop routine statement>.
— <sequence generator definition>.
— <alter sequence generator statement>.
— <drop sequence generator statement>.
— <role definition>.
— <grant role statement>.
— <drop role statement>.

4.41.2.2 SQL-data statements

The following are the SQL-data statements:

— <temporary table declaration>.
— <declare cursor> (that does not contain a <data change delta table>).
— <open statement>.

— <close statement>.
— <fetch statement>.
— <select statement: single row> (that does not contain a <data change delta table>).
— <free locator statement>.
— <hold locator statement>.
— <dynamic declare cursor>.
— <allocate extended dynamic cursor statement>.
— <allocate received cursor statement>.
— <dynamic select statement>.
— <dynamic open statement>.
— <dynamic close statement>.
— <dynamic fetch statement>.
— <direct select statement: multiple rows> (that does not contain a <data change delta table>).
— <dynamic single row select statement> (that does not contain a <data change delta table>).
— All SQL-data change statements.

4.41.2.3 SQL-data change statements

The following are the SQL-data change statements:

— <insert statement>.
— <delete statement: searched>.
— <delete statement: positioned>.
— <update statement: searched>.
— <update statement: positioned>.
— <merge statement>.
— <truncate table statement>.
— <declare cursor> (that contains a <data change delta table>).
— <select statement: single row> (that contains a <data change delta table>).
— <dynamic delete statement: positioned>.
— <preparable dynamic delete statement: positioned>.
— <dynamic update statement: positioned>.
— <preparable dynamic update statement: positioned>.
— <direct select statement: multiple rows> (that contains a <data change delta table>).
— <dynamic single row select statement> (that contains a <data change delta table>).

4.41.2.4 SQL-transaction statements

The following are the SQL-transaction statements:

— <start transaction statement>.

— <set transaction statement>.

— <set constraints mode statement>.

— <commit statement>.

— <rollback statement>.

— <savepoint statement>.

— <release savepoint statement>.

4.41.2.5 SQL-connection statements

The following are the SQL-connection statements:

— <connect statement>.

— <set connection statement>.

— <disconnect statement>.

4.41.2.6 SQL-control statements

This Subclause is modified by Subclause 4.11.2.2, "SQL-control statements", in ISO/IEC 9075-4.

04 The following are the SQL-control statements:

— <call statement>.

— <return statement>.

4.41.2.7 SQL-session statements

This Subclause is modified by Subclause 4.7.1.2, "SQL-session statements", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.7.1.1, "SQL-session statements", in ISO/IEC 9075-14.

09 14 The following are the SQL-session statements:

— <set session characteristics statement>.

— <set session user identifier statement>.

— <set role statement>.

— <set local time zone statement>.

— <set catalog statement>.

— <set schema statement>.

— <set names statement>.

— <set path statement>.

— <set transform group statement>.

— <set session collation statement>.

4.41.2.8 SQL-diagnostics statements

This Subclause is modified by Subclause 4.11.2.4, "SQL-diagnostics statements", in ISO/IEC 9075-4.

04 The following are the SQL-diagnostics statements:

— <get diagnostics statement>.

4.41.2.9 SQL-dynamic statements

The following are the SQL-dynamic statements:

— <execute immediate statement>.

— <allocate descriptor statement>.

— <deallocate descriptor statement>.

— <get descriptor statement>.

— <set descriptor statement>.

— <prepare statement>.

— <deallocate prepared statement>.

— <describe input statement>.

— <describe output statement>.

— <execute statement>.

— <copy descriptor statement>.

— <pipe row statement>.

4.41.2.10 SQL embedded exception declaration

The following is the SQL embedded exception declaration:

— <embedded exception declaration>.

4.41.3 SQL-statements and SQL-data access indication

Some SQL-statements may be classified either as SQL-statements that *possibly read SQL-data* or that *possibly modify SQL-data*. A given SQL-statement belongs to at most one such class.

The following SQL-statements possibly read SQL-data:

— SQL-data statements other than SQL-data change statements, <free locator statement>, and <hold locator statement>.

— SQL-statements that contain a <query expression> and are not SQL-statements that possibly modify SQL-data.

— SQL-statements that contain a <routine invocation> with at least one subject routine that possibly reads SQL-data and no subject routine that possibly modifies SQL-data.

The following SQL-statements possibly modify SQL-data:

— SQL-schema statements.

— SQL-data change statements.

— SQL-statements that contain a <routine invocation> with at least one subject routine that possibly modifies SQL-data.

— SQL-statements that contain a <data change delta table>.

NOTE 86 — The final item in the preceding list is redundant in this document, though not in other Parts, particularly ISO/IEC 9075-4.

4.41.4 SQL-statements and transaction states

This Subclause is modified by Subclause 4.11.3, "SQL-statements and transaction states", in ISO/IEC 9075-4.

04 The following SQL-statements are transaction-initiating SQL-statements, i.e., if there is no current SQL-transaction, and an SQL-statement of this class is executed, then an SQL-transaction is initiated:

— All SQL-schema statements.

— The following SQL-transaction statements:

- <start transaction statement>.
- <savepoint statement>.
- <commit statement>.
- <rollback statement>.

— The following SQL-data statements:

- <open statement>.
- <close statement>.
- <fetch statement>.
- <select statement: single row>.
- <insert statement>.
- <delete statement: searched>.
- <delete statement: positioned>.
- <update statement: searched>.
- <update statement: positioned>.
- <merge statement>.
- <truncate table statement>.
- <allocate extended dynamic cursor statement>.
- <allocate received cursor statement>.
- <dynamic open statement>.
- <dynamic close statement>.
- <dynamic fetch statement>.
- <direct select statement: multiple rows>.
- <dynamic single row select statement>.
- <dynamic delete statement: positioned>.

- <preparable dynamic delete statement: positioned>.
- <dynamic update statement: positioned>.
- <preparable dynamic update statement: positioned>.
- <free locator statement>.
- <hold locator statement>.

— The following SQL-dynamic statements:

- <describe input statement>.
- <describe output statement>.
- <allocate descriptor statement>.
- <deallocate descriptor statement>.
- <get descriptor statement>.
- <set descriptor statement>.
- <deallocate prepared statement>.

The <start transaction statement> explicitly initiates an SQL-transaction as the primary effect of its execution. The other transaction-initiating SQL-statements implicitly initiate an SQL-transaction when and only when no current SQL-transaction exists before the statement is executed; in that case, an SQL-transaction is implicitly initiated before execution of that SQL-statement proceeds.

Whether or not a <prepare statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <prepare statement> at the time <prepare statement> is executed. Whether or not an <execute immediate statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <execute immediate statement> at the time it is executed. Whether or not an <execute statement> starts a transaction depends on the content of the <SQL statement variable> referenced by the <prepare statement> at the time the prepared statement referenced by the <execute statement> was prepared. In each case, if the content of the <SQL statement variable> was a transaction-initiating SQL-statement other than a <start transaction statement>, then the <prepare statement>, <execute immediate statement>, or <execute statement> is treated as a transaction-initiating statement; otherwise, it is not treated as a transaction-initiating statement.

04 The following SQL-statements are not transaction-initiating SQL-statements, i.e., then executing an SQL-statement of this class does not affect the existence or absence of an SQL-transaction.

— All SQL-transaction statements except <start transaction statement>s, <savepoint statement>s, <commit statement>s, and <rollback statement>s.

— All SQL-connection statements.

— All SQL-session statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <temporary table declaration>.
- <declare cursor>.
- <dynamic declare cursor>.

- <dynamic select statement>.

04 The following SQL-statements are possibly transaction-initiating SQL-statements:

— <return statement>.

— <call statement>.

If an <SQL control statement> causes the evaluation of a <query expression> and there is no current SQL-transaction, then an SQL-transaction is initiated before evaluation of the <query expression>.

4.41.5 SQL-statement atomicity and statement execution contexts

This Subclause is modified by Subclause 4.11.4, "SQL-statement atomicity and statement execution contexts", in ISO/IEC 9075-4.

The execution of all SQL-statements other than certain SQL-control statements and certain SQL-transaction statements is atomic with respect to recovery. Such an SQL-statement is called an *atomic SQL-statement*. An SQL-statement that is not an atomic SQL-statement is called a *non-atomic SQL-statement*.

04 The following are non-atomic SQL-statements:

— <call statement>

— <execute statement>

— <execute immediate statement>

— <commit statement>

— <return statement>

— <rollback statement>

— <savepoint statement>

All other SQL-statements are atomic SQL-statements.

A statement execution context is either *atomic* or *non-atomic*.

The statement execution context brought into existence by the execution of a non-atomic SQL-statement is a *non-atomic execution context*.

The statement execution context brought into existence by the execution of an atomic SQL-statement is an *atomic execution context*.

Within one execution context, another execution context may become active. This latter execution context is said to be a *more recent execution context* than the former. If there is no execution context that is more recent than execution context *EC*, then *EC* is said to be the *most recent execution context*.

If there is no atomic execution context that is more recent than atomic execution context *AEC*, then *AEC* is the *most recent atomic execution context*.

An SQL-transaction cannot be explicitly terminated within an atomic execution context. If the execution of an atomic SQL-statement is unsuccessful, then the changes to SQL-data or schemas made by the SQL-statement are canceled.

A statement execution context includes the following:

— An indication of whether the statement execution context is atomic.

— A set of state changes.

NOTE 87 — State changes are described in Subclause 4.46.2, "Trigger execution".

— A set of old delta tables and new delta tables.

NOTE 88 — Old delta tables and new delta tables are brought into existence and subsequently destroyed by the execution of <data change delta table>.

— A set of views to be checked.

4.41.6 Embeddable SQL-statements

This Subclause is modified by Subclause 4.11.5, "Embeddable SQL-statements", in ISO/IEC 9075-4.

04 The following SQL-statements are embeddable in an embedded SQL host program, and may be the <SQL procedure statement> in an <externally-invoked procedure> in an <SQL-client module definition>:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-connection statements.

— All SQL-session statements.

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— The following SQL-data statements:

- <allocate extended dynamic cursor statement>.
- <allocate received cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <insert statement>.
- <delete statement: searched>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <update statement: searched>.
- <update statement: positioned>.
- <merge statement>.
- <truncate table statement>.
- <dynamic update statement: positioned>.
- <hold locator statement>.

- <free locator statement>.

— 04 The following SQL-control statements:

- <call statement>.
- <return statement>.

The following SQL-statements are embeddable in an embedded SQL host program, and may occur in an <SQL-client module definition>, though not in an <externally-invoked procedure>:

— <temporary table declaration>.

— <declare cursor>.

— <dynamic declare cursor>.

The following SQL-statements are embeddable in an embedded SQL host program, but shall not be specified in an <SQL-client module definition>:

— SQL embedded exception declarations.

Consequently, the following SQL-data statements are not embeddable in an embedded SQL host program, nor may they occur in an <SQL-client module definition>, nor be the <SQL procedure statement> in an <externally-invoked procedure> in an <SQL-client module definition>:

— <dynamic select statement>.

— <dynamic single row select statement>.

— <direct select statement: multiple rows>.

— <preparable dynamic delete statement: positioned>.

— <preparable dynamic update statement: positioned>.

4.41.7 Preparable and immediately executable SQL-statements

This Subclause is modified by Subclause 4.11.6, "Preparable and immediately executable SQL-statements", in ISO/IEC 9075-4.

The following SQL-statements are preparable:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-session statements.

— The following SQL-data statements:

- <delete statement: searched>.
- <dynamic select statement>.
- <dynamic single row select statement>.
- <insert statement>.
- <update statement: searched>.
- <truncate table statement>.
- <merge statement>.

- <preparable dynamic delete statement: positioned>.
- <preparable dynamic update statement: positioned>.
- <preparable implementation-defined statement>.
- <hold locator statement>.
- <free locator statement>.

— The following SQL-control statements:

- <call statement>.

04 Consequently, the following SQL-statements are not preparable:

— All SQL-connection statements.

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <allocate extended dynamic cursor statement>.
- <allocate received cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.
- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <update statement: positioned>.
- <dynamic update statement: positioned>.
- <direct select statement: multiple rows>.
- <temporary table declaration>.
- <declare cursor>.
- <dynamic declare cursor>.

— 04 The following SQL-control statements:

- <return statement>.

Any preparable SQL-statement can be executed immediately, with the exception of:

— <dynamic select statement>.

— <dynamic single row select statement>.

4.41.8 Directly executable SQL-statements

This Subclause is modified by Subclause 4.11.7, "Directly executable SQL-statements", in ISO/IEC 9075-4.

04 The following SQL-statements may be executed directly:

— All SQL-schema statements.

— All SQL-transaction statements.

— All SQL-connection statements.

— All SQL-session statements.

— The following SQL-data statements:

- <temporary table declaration>.
- <direct select statement: multiple rows>.
- <insert statement>.
- <delete statement: searched>.
- <update statement: searched>.
- <truncate table statement>.
- <merge statement>.

— 04 The following SQL-control statements:

- <call statement>.
- <return statement>.

Consequently, the following SQL-statements shall not be executed directly:

— All SQL-dynamic statements.

— All SQL-diagnostics statements.

— SQL embedded exception declarations.

— The following SQL-data statements:

- <declare cursor>.
- <dynamic declare cursor>.
- <allocate extended dynamic cursor statement>.
- <allocate received cursor statement>.
- <open statement>.
- <dynamic open statement>.
- <close statement>.
- <dynamic close statement>.

- <fetch statement>.
- <dynamic fetch statement>.
- <select statement: single row>.
- <dynamic select statement>.
- <dynamic single row select statement>.
- <delete statement: positioned>.
- <dynamic delete statement: positioned>.
- <preparable dynamic delete statement: positioned>.
- <update statement: positioned>.
- <dynamic update statement: positioned>.
- <preparable dynamic update statement: positioned>.

— <free locator statement>.

— <hold locator statement>.

4.42 Basic security model

This Subclause is modified by Subclause 4.12, "Basic security model", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.8, "Basic security model", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.11, "Basic security model", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.8, "Basic security model", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.4, "Basic security model", in ISO/IEC 9075-16.

4.42.1 Authorization identifiers

4.42.1.1 Introduction to authorization identifiers

An authorization identifier identifies a set of privileges. An authorization identifier is either a user identifier or a role name. A user identifier represents a user of the SQL-implementation. Any mapping of user identifiers to operating system users is implementation-defined (IA207). A role name represents a role.

4.42.1.2 SQL-session authorization identifiers

An SQL-session has a user identifier called the *SQL-session user identifier*. When an SQL-session is initiated, the SQL-session user identifier is determined in an implementation-defined (IW061) manner, unless the session is initiated using a <connect statement>. The value of the SQL-session user identifier can never be the null value. The SQL-session user identifier can be determined by using SESSION_USER.

An SQL-session context contains a time-varying sequence of cells, known as the *authorization stack*, each cell of which contains a user identifier, a role name, or both. This stack is maintained using a "last-in, first-out" discipline, and effectively only the top cell is visible. When an SQL-session is started, by explicit or implicit execution of a <connect statement>, the authorization stack is initialized with one cell, which contains only the user identifier known as the *SQL-session user identifier*; a role name, known as the *SQL-session role name* may be added subsequently.

Execution of a <routine invocation>, <externally-invoked procedure>, triggered action, <execute statement>, or <direct SQL statement> *X* pushes a new cell onto the top of the authorization stack. This new cell might be a copy of the existing top cell, a routine authorization identifier, or the authorization identifier of a module owner, schema owner, or prepared statement owner, as prescribed by the applicable General Rules. After execution of *X* completes, the top cell is removed.

The contents of the top cell in the authorization stack of the current SQL-session context determine the privileges for the execution of each SQL-statement. The user identifier, if any, in this cell is known as the *current user identifier*; the role name, if any, is known as the *current role name*. They may be determined using CURRENT_USER and CURRENT_ROLE, respectively.

At a given time, there may be no current user identifier or no current role name, but at least one or the other is always present.

NOTE 89 — The privileges granted to PUBLIC are available to all of the <authorization identifier>s in the SQL-environment.

The <set session user identifier statement> changes the value of the current user identifier and of the SQL-session user identifier. The <set role statement> changes the value of the current role name.

The term *current authorization identifier* denotes an authorization identifier in the top cell of the authorization stack.

4.42.1.3 SQL-client module authorization identifiers

If an <SQL-client module definition> contains a <module authorization identifier> *MAI*, then *MAI* is the owner of the corresponding SQL-client module *M* and is used as the current authorization identifier for the execution of each externally-invoked procedure in *M*. If *M* has no owner, then the current user identifier and the current role name of the SQL-session are used as the current user identifier and current role name, respectively, for the execution of each externally-invoked procedure in *M*.

4.42.1.4 SQL-schema authorization identifiers

Every schema has an owner, determined at the time of its creation from a <schema definition> *SD*. That owner is

Case:

— If *SD* simply contains a <schema authorization identifier> *SAI*, then *SAI*.

— If *SD* is simply contained in an <SQL-client module definition> that contains a <module authorization identifier> *MAI*, then *MAI*.

— Otherwise, the SQL-session user identifier.

4.42.2 Privileges

This Subclause is modified by Subclause 4.12.1, "Privileges", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.8.1, "Privileges", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.11.1, "Privileges", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 4.8.1, "Privileges", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 4.4.1, "Privileges", in ISO/IEC 9075-16.

04 09 13 14 16 A privilege authorizes a specified action to be performed by a specified <authorization identifier> on a specified object. The object in question is one of the following:

— A base table.

— A view.

— A view component.

— A column.

— A domain.

— A character set.

— A collation.

— A transliteration.

— A user-defined type.

— A table/method pair.

— An SQL-invoked routine.

— A sequence generator.

Each privilege is represented by a *privilege descriptor*. A privilege descriptor contains:

— The identification of the object on which the privilege is granted.

— The <authorization identifier> of the grantor of the privilege.

— The <authorization identifier> of the grantee of the privilege.

— Identification of the action that the privilege allows.

— An indication of whether or not the privilege is grantable.

— An indication of whether or not the privilege has the WITH HIERARCHY OPTION specified.

A privilege descriptor *P* with object *O*, grantee *G*, and action *ACT* is said to describe an *ACT privilege on O held by G*.

NOTE 90 — For example, a SELECT privilege on a table, column, or table/method pair; an INSERT privilege on a table or column; etc.

If *ACT* is SELECT and *P* has an indication that the privilege has WITH HIERARCHY OPTION, then *P* describes the SELECT WITH HIERARCHY OPTION privilege on *O* held by *G*.

A *privilege descriptor kernel* contains:

— The identification of an object.

— The identification of an action.

NOTE 91 — A privilege descriptor kernel is a subset of a privilege descriptor. A privilege descriptor kernel is used in the rules of Clause 12, "Access control", to construct privilege descriptors from <object name> and <action>, which specify, in a non-orthogonal way, the combination of object and action.

The action is one of the following:

— INSERT.

— UPDATE.

— DELETE.

— SELECT.

— REFERENCES.

— USAGE.

— UNDER.

— TRIGGER.

— EXECUTE.

NOTE 92 — Some <action>s, such as SELECT (<column name list>), both specify an action and also partially specify one or more objects.

A privilege descriptor with an object that is a table or view component *T* and an action of INSERT, UPDATE, DELETE, SELECT, TRIGGER, or REFERENCES is called a *table privilege descriptor* and identifies the existence of a privilege on *T*. If *T* is a view component, then the privilege descriptor is called a *view component table privilege descriptor*.

16 A privilege descriptor with an object that is a column *C* and an action of SELECT, INSERT, UPDATE, or REFERENCES is called a *column privilege descriptor* and identifies the existence of a privilege *C*. If *C* is a column of a view component, then the privilege descriptor is called a *view component column privilege descriptor*.

A privilege descriptor with an action of SELECT and an object that is a table/method pair is called a *table/method privilege descriptor* and identifies the existence of a privilege on the identified method of the structured type of the table identified by the privilege descriptor.

A table privilege descriptor specifies that the privilege identified by the action (unless the action is DELETE) is to be automatically granted by the grantor to the grantee on all columns subsequently added to the table. In addition, if the table is a referenceable table of structured type *ST* and the action is SELECT, then the table privilege descriptor specifies that the SELECT privilege is to be automatically granted by the grantor to the grantee on all methods subsequently added to *ST*.

09 13 14 A privilege descriptor with an action of USAGE is called a *usage privilege descriptor* and describes a privilege on an object of one of the following kinds:

— A domain.

— A user-defined type.

— A character set.

— A collation.

— A transliteration.

— A sequence generator.

A privilege descriptor with an action of UNDER is called an *under privilege descriptor* and identifies the existence of the privilege on the structured type identified by the privilege descriptor.

A privilege descriptor with an action of EXECUTE is called an *execute privilege descriptor* and identifies the existence of a privilege on the SQL-invoked routine identified by the privilege descriptor.

A grantable privilege is a privilege that may be granted by a <grant privilege statement>. The WITH GRANT OPTION clause of a <grant privilege statement> specifies whether the <authorization identifier> identifying the recipient of a privilege (acting as a grantor) may grant it to others.

Privilege descriptors that represent privileges for the owner of an object have a special grantor value, "_SYSTEM". This value is reflected in the Information Schema for all privileges that apply to the owner of the object.

NOTE 93 — "_SYSTEM" is a <delimited identifier>. Thus, the enclosing quotation marks are are the <double quote>s specified in the BNF production for <delimited identifier>.

NOTE 94 — The Information Schema is defined in ISO/IEC 9075-11.

A schema that is owned by a given schema <user identifier> or schema <role name> may contain privilege descriptors that describe privileges granted to other <authorization identifier>s (grantees). The granted privileges apply to objects defined in the current schema.

04 A *view privilege dependency descriptor* is a descriptor that includes two privilege descriptors, called the *supporting privilege descriptor* and the *dependent privilege descriptor*. A view privilege dependency descriptor is a record that an INSERT, UPDATE, or DELETE privilege of a view, or a column of a view, is directly dependent on another privilege.

4.42.3 Roles

A role, identified by a role name, is, like a user, a potential grantee and grantor of privileges and of other roles. Also like a user, a role can additionally own schemas and other objects.

A role is created by executing a <role definition> and destroyed by executing a <drop role statement>.

A role is granted to one or more authorization identifiers by executing a <grant role statement>. The granting of a role to an authorization identifier *A* is called a *role authorization* (for *A*).

If the authorization identifier *A* to which a role *R* is granted is a user identifier, then *A* is able by means of a <set role statement> to temporarily also acquire the privileges of *R*.

The privileges of a role with role name *R* are the union of the privileges whose grantee is *R* and the sets of privileges for the role names defined by the role authorizations for *R*. Cycles of role authorizations are prohibited.

The WITH ADMIN OPTION clause of the <grant role statement> for role *R* specifies that each grantee may grant *R* to others, revoke *R* from others, and destroy *R*.

Each role authorization is described by a *role authorization descriptor*. A role authorization descriptor includes:

— The role name of the role.

— The authorization identifier of the grantor.

— The authorization identifier of the grantee.

— An indication of whether or not the role authorization is grantable.

4.42.4 Security model definitions

A role *R* is *applicable for* an authorization identifier *A* if there exists a role authorization descriptor whose role name is *R* and whose grantee is PUBLIC, or *A*, or an applicable role for *A*.

A privilege *P* is *applicable for* an authorization identifier *A* if its grantee is PUBLIC, or *A*, or, if *A* is a role name, an applicable role for *A*.

NOTE 95 — "applicable for" is a persistent relationship between persistent objects. Thus, it in no way depends on any SQL-session.

An authorization identifier is *enabled* if it is the current user identifier, the current role name, or a role name that is applicable for the current role name.

A privilege *P* is *current* if *P* is applicable for an enabled authorization identifier.

NOTE 96 — "enabled" and "current" apply to (transient) elements of the current SQL-session context.

4.43 SQL-transactions

This Subclause is modified by Subclause 4.9, "SQL-transactions", in ISO/IEC 9075-9.

4.43.1 General description of SQL-transactions

An *SQL-transaction* (transaction) is a sequence of executions of SQL-statements that is atomic with respect to recovery. These operations are performed by one or more compilation units and SQL-client modules. The operations comprising an SQL-transaction may also be performed by the direct invocation of SQL.

If an SQL-implementation supports Feature T670, "Schema and data statement mixing", an SQL-transaction may cause the execution of either SQL-data and SQL-schema statements or <SQL dynamic data statement>s and dynamic SQL-schema statements. There may be additional implementation-defined (IA210)

restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then it is implementation-defined (IC013) whether an implementation-defined (IC013) exception condition is raised or a completion condition *warning (01000)* is raised.

Without Feature T670, "Schema and data statement mixing", executing either SQL-data and SQL-schema statements or executing <SQL dynamic data statement>s and dynamic SQL-schema statements in the same SQL-transaction results in raising an exception condition.

If such mixing occurs, then, in either case, the effect on any open cursor or deferred constraint is implementation-defined (IA210).

Each SQL-client module that executes an SQL-statement of an SQL-transaction is associated with that SQL-transaction. Each direct invocation of SQL that executes an SQL-statement of an SQL-transaction is associated with that SQL-transaction. An SQL-transaction is initiated when no SQL-transaction is currently active by direct invocation of SQL that results in the execution of a transaction-initiating <direct SQL statement>. An SQL-transaction is initiated when no SQL-transaction is currently active and an <externally-invoked procedure> is called that results in the execution of a *transaction-initiating* SQL-statement. An SQL-transaction is terminated by a <commit statement> or a <rollback statement>. If an SQL-transaction is terminated by successful execution of a <commit statement>, then all changes made to SQL-data or schemas by that SQL-transaction are made persistent and accessible to all concurrent and subsequent SQL-transactions. If an SQL-transaction is terminated by a <rollback statement> or unsuccessful execution of a <commit statement>, then all changes made to SQL-data or schemas by that SQL-transaction are canceled. Committed changes cannot be canceled. If execution of a <commit statement> is attempted, but certain exception conditions are raised, it is unknown whether or not the changes made to SQL-data or schemas by that SQL-transaction are canceled or made persistent.

When an SQL-transaction is terminated by a <commit statement> or a <rollback statement> that specifies AND CHAIN, a new SQL-transaction is immediately started. The new SQL-transaction will have the same characteristics as the SQL-transaction just terminated, except that the constraint mode of each integrity constraint reverts to its default mode (deferred or immediate).

4.43.2 Savepoints

An SQL-transaction may be partially rolled back by using a savepoint. The savepoint and its <savepoint name> are established within an SQL-transaction when a <savepoint statement> is executed.

An SQL-transaction has one or more *savepoint levels*, exactly one of which is the *current savepoint level*. The savepoint levels of an SQL-transaction are nested, such that when a *new savepoint level NSL is established*, the current savepoint level *CSL* ceases to be current and *NSL* becomes current. When *NSL is destroyed*, *CSL* becomes current again.

A savepoint level exists in an SQL-session *SS* even when no SQL-transaction is active, this savepoint level remaining the current one when an SQL-transaction is initiated in *SS*.

A savepoint *SP* exists at exactly one savepoint level, namely, the savepoint level that is current when *SP* is established.

If a <rollback statement> references a savepoint *SS*, then all changes made to SQL-data or schema subsequent to the establishment of *SS* are canceled, and all savepoints established since *SS* was established are destroyed.

NOTE 97 — The SQL-transaction is effectively restored to its state as it was immediately following the execution of the <savepoint statement> that established *SS*.

Savepoints existing at savepoint level *SPL* are destroyed when *SPL* is destroyed. Savepoint *SS* in the current savepoint level and all savepoints established since *SS* was established are destroyed when a <release savepoint statement> specifying the savepoint name of *SS* is executed. A savepoint may be replaced by another with the same name within a savepoint level by executing a <savepoint statement> that specifies that name.

It is implementation-defined (IA211) whether or not, or how, a <rollback statement> that references a <savepoint specifier> affects diagnostics area contents, the contents of SQL descriptor areas, and the status of prepared statements.

4.43.3 Properties of SQL-transactions

This Subclause is modified by Subclause 4.9.1, "Properties of SQL-transactions", in ISO/IEC 9075-9.

An SQL-transaction has a *constraint mode* for each integrity constraint. The constraint mode for an integrity constraint in an SQL-transaction is described in Subclause 4.25, "Integrity constraints".

An SQL-transaction has a *transaction access mode* that is either *read-only* or *read-write*. The transaction access mode may be explicitly set by a <set transaction statement> before the start of an SQL-transaction or by the use of a <start transaction statement> to start an SQL-transaction; otherwise, it is implicitly set to the default transaction access mode for the SQL-session before each SQL-transaction begins. If no <set session characteristics statement> has set the default transaction access mode for the SQL-session, then the default transaction access mode for the SQL-session is *read-write*.

09 The term *read-only* applies only to viewed tables and persistent base tables.

An SQL-transaction has a *condition area limit*, which is a positive integer that specifies the maximum number of conditions that can be placed in any diagnostics area during execution of an SQL-statement in this SQL-transaction.

An SQL-transaction has a *transaction timestamp*, a value of an implementation-defined (IV072) timestamp type that is used to set the values of system-time period start and system-time period end columns of rows, if any, modified by the execution of an SQL-data change statement in this SQL-transaction. The transaction timestamp is set by an SQL-implementation before any SQL-data change statement is executed in that transaction and, once set, remains unchanged during that SQL-transaction.

SQL-transactions initiated by different SQL-agents that access the same SQL-data or schemas and overlap in time are *concurrent SQL-transactions*.

4.43.4 Isolation levels of SQL-transactions

An SQL-transaction has a *transaction isolation level* that is READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. The transaction isolation level of an SQL-transaction defines the degree to which the operations on SQL-data or schemas in that SQL-transaction are affected by the effects of and can affect operations on SQL-data or schemas in concurrent SQL-transactions. The transaction isolation level of an SQL-transaction when any cursor is held open from the previous SQL-transaction within an SQL-session is the transaction isolation level of the previous SQL-transaction by default. If no cursor is held open, or this is the first SQL-transaction within an SQL-session, then the transaction isolation level is SERIALIZABLE by default. The transaction isolation level can be explicitly set by the <set transaction statement> before the start of an SQL-transaction or by the use of a <start transaction statement> to start an SQL-transaction. If it is not explicitly set, then the transaction isolation level is implicitly set to the default transaction isolation level for the SQL-session before each SQL-transaction begins. If no <set session characteristics statement> has set the default transaction isolation level for the SQL-session, then the default transaction isolation level for the SQL-session is SERIALIZABLE.

Execution of a <set transaction statement> is prohibited after the start of an SQL-transaction and before its termination. Execution of a <set transaction statement> before the start of an SQL-transaction sets the transaction access mode, transaction isolation level, and condition area limit for the single SQL-transaction that is started after the execution of that <set transaction statement>. If multiple <set transaction statement>s are executed before the start of an SQL-transaction, the last such statement is the one whose settings are effective for that SQL-transaction; their actions are not cumulative.

The execution of concurrent SQL-transactions at transaction isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-

transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.

The transaction isolation level specifies the kind of phenomena that can occur during the execution of concurrent SQL-transactions. The following phenomena are possible:

— *P1* ("Dirty read"): SQL-transaction *T1* modifies a row. SQL-transaction *T2* then reads that row before *T1* performs a COMMIT. If *T1* then performs a ROLLBACK, *T2* will have read a row that was never committed and that may thus be considered to have never existed.

— *P2* ("Non-repeatable read"): SQL-transaction *T1* reads a row. SQL-transaction *T2* then modifies or deletes that row and performs a COMMIT. If *T1* then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.

— *P3* ("Phantom"): SQL-transaction *T1* reads the set of rows *N* that satisfy some <search condition>. SQL-transaction *T2* then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction *T1*. If SQL-transaction *T1* then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

The four transaction isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that no updates will be lost. The transaction isolation levels are different with respect to phenomena *P1*, *P2*, and *P3*. Table 11, "SQL-transaction isolation levels and the three phenomena", specifies the phenomena that are possible and not possible for a given transaction isolation level.

Table 11 — SQL-transaction isolation levels and the three phenomena

Level	*P1*	*P2*	*P3*
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

NOTE 98 — The exclusion of these phenomena for SQL-transactions executing at transaction isolation level SERIALIZABLE is a consequence of the requirement that such transactions be serializable.

Changes made to SQL-data or schemas by an SQL-transaction in an SQL-session may be perceived by that SQL-transaction in that same SQL-session, and by other SQL-transactions, or by that same SQL-transaction in other SQL-sessions, at transaction isolation level READ UNCOMMITTED, but cannot be perceived by other SQL-transactions at transaction isolation level READ COMMITTED, REPEATABLE READ, or SERIALIZABLE until the former SQL-transaction terminates with a <commit statement>.

Regardless of the transaction isolation level of the SQL-transaction, phenomena *P1*, *P2*, and *P3* shall not occur during the implied reading of schema definitions performed on behalf of executing an SQL-statement, the checking of integrity constraints, and the execution of referential actions associated with referential constraints. The schema definitions that are implicitly read are implementation-dependent (UA049). This does not affect the explicit reading of rows from tables in the Information Schema, which is done at the transaction isolation level of the SQL-transaction.

NOTE 99 — The Information Schema is defined in ISO/IEC 9075-11.

4.43.5 Implicit rollbacks

The execution of a <rollback statement> may be initiated implicitly by an SQL-implementation when it detects the inability to guarantee the serializability of two or more concurrent SQL-transactions. When this error occurs, an exception condition is raised: *transaction rollback — serialization failure (40001)*.

The execution of a <rollback statement> may be initiated implicitly by an SQL-implementation when it detects unrecoverable errors. When such an error occurs, an exception condition is raised: *transaction rollback (40000)* with an implementation-defined (IC014) subclass code.

The execution of a <rollback statement> may be initiated implicitly when an SQL-implementation detects the loss of the current SQL-Connection. See Subclause 4.44, "SQL-connections".

4.43.6 Effects of SQL-statements in an SQL-transaction

The execution of an SQL-statement within an SQL-transaction has no effect on SQL-data or schemas other than the effect stated in the General Rules for that SQL-statement and in the General Rules for Subclause 13.4, "<SQL procedure statement>". Together with serializable execution, this implies that all read operations are repeatable within an SQL-transaction at transaction isolation level SERIALIZABLE, except for:

— The effects of changes to SQL-data or schemas and its contents made explicitly by the SQL-transaction itself.

— The effects of differences in SQL parameter values supplied to externally-invoked procedures.

— The effects of references to time-varying system variables such as CURRENT_DATE and CURRENT_USER.

4.43.7 Encompassing transactions

4.43.7.1 Encompassing transaction belonging to an external agent

In some environments (e.g., remote database access), an SQL-transaction can be part of an encompassing transaction that is controlled by an agent other than the SQL-agent.

In such environments, an encompassing transaction shall be terminated via that other agent, which in turn interacts with the SQL-implementation via an interface that may be different from SQL (COMMIT or ROLLBACK), in order to coordinate the orderly termination of the encompassing transaction. If the encompassed SQL-transaction is terminated by an implicitly initiated <rollback statement>, then the SQL-implementation will interact with that other agent to terminate that encompassing transaction. The specification of the interface between such agents and the SQL-implementation is beyond the scope of this document. However, it is important to note that the semantics of an SQL-transaction remain as defined in the following sense:

— When an agent that is different from the SQL-agent requests the SQL-implementation to rollback an SQL-transaction, the General Rules of Subclause 17.8, "<rollback statement>", are performed.

— When such an agent requests the SQL-implementation to commit an SQL-transaction, the General Rules of Subclause 17.7, "<commit statement>", are performed. To guarantee orderly termination of the encompassing transaction, this commit operation may be processed in several phases not visible to the application; not all the General Rules of Subclause 17.7, "<commit statement>", need to be executed in a single phase.

However, even in such environments, the SQL-agent interacts directly with the SQL-server to set characteristics (such as *read-only* or *read-write*, transaction isolation level, and constraints mode) that are specific to the SQL-transaction model.

4.43.7.2 Encompassing transaction belonging to the SQL-agent

It is implementation-defined (IA057) whether SQL-transactions that affect more than one SQL-server are supported. If such SQL-transactions are supported, then the part of each SQL-transaction that affects a single SQL-server is called a *branch transaction* or a branch of the SQL-transaction. If such SQL-transactions are supported, then they generally have all the same characteristics (transaction access mode, condition area limit, and transaction isolation level, as well as constraint mode). However, it is possible

to alter some characteristics of such an SQL-transaction at one SQL-server by the use of the SET LOCAL TRANSACTION statement; if a SET LOCAL TRANSACTION statement is executed at an SQL-server before any transaction-initiating SQL-statement, then it may set the characteristics of that *branch* of the SQL-transaction at that SQL-server.

The characteristics of a branch of an SQL-transaction are limited by the characteristics of the SQL-transaction as a whole:

— If the SQL-transaction is read-write, then the branch of the SQL-transaction may be read-write or read-only; if the SQL-transaction is read-only, then the branch of the SQL-transaction shall be read-only.

— If the SQL-transaction has a transaction isolation level of READ UNCOMMITTED, then the branch of the SQL-transaction may have a transaction isolation level of READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

  If the SQL-transaction has a transaction isolation level of READ COMMITTED, then the branch of the SQL-transaction shall have a transaction isolation level of READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

  If the SQL-transaction has a transaction isolation level of REPEATABLE READ, then the branch of the SQL-transaction shall have a transaction isolation level of REPEATABLE READ or SERIALIZABLE.

  If the SQL-transaction has a transaction isolation level of SERIALIZABLE, then the branch of the SQL-transaction shall have a transaction isolation level of SERIALIZABLE.

— The condition area limit of a branch of an SQL-transaction is always the same as the condition area limit of the SQL-transaction; SET LOCAL TRANSACTION shall not specify a condition area limit.

4.44 SQL-connections

An *SQL-connection* is an association between an SQL-client and an SQL-server. An SQL-connection may be established and named by a <connect statement>, which identifies the desired SQL-server by means of an <SQL-server name>. A <connection name> is specified as a <simple value specification> whose value is an <identifier>. Two <connection name>s identify the same SQL-connection if their values, with leading and trailing <space>s removed, are equivalent according to the rules for <identifier> comparison in Subclause 5.2, "<token> and <separator>". It is implementation-defined (IW123) how an SQL-implementation uses <SQL-server name> to determine the location, identity, and communication protocol required to access the SQL-server and create an SQL-session.

An SQL-connection is an *active SQL-connection* if any SQL-statement that initiates or requires an SQL-transaction has been executed at its SQL-server via that SQL-connection during the current SQL-transaction.

An SQL-connection is either *current* or *dormant*. If the SQL-connection established by the most recently executed implicit or explicit <connect statement> or <set connection statement> has not been terminated, then that SQL-connection is the *current SQL-connection*; otherwise, there is no current SQL-connection. An existing SQL-connection that is not the current SQL-connection is a *dormant SQL-connection*.

An SQL-implementation may detect the loss of the current SQL-connection during execution of any SQL-statement. When such a connection failure is detected, an exception condition is raised: *transaction rollback — statement completion unknown (40003)*. This exception condition indicates that the results of the actions performed in the SQL-server on behalf of the statement are unknown to the SQL-agent.

Similarly, an SQL-implementation may detect the loss of the current SQL-connection during the execution of a <commit statement>. When such a connection failure is detected, an exception condition is raised: *connection exception — transaction resolution unknown (08007)*. This exception condition indicates that the SQL-implementation cannot verify whether the SQL-transaction was committed successfully, rolled back, or left active.

A user may initiate an SQL-connection between the SQL-client associated with the SQL-agent and a specific SQL-server by executing a <connect statement>. Otherwise, an SQL-connection between the SQL-client and an implementation-defined (ID043) default SQL-server is initiated when an externally-invoked procedure is called and no SQL-connection is current. The SQL-connection associated with an implementation-defined (ID043) default SQL-server is called the *default SQL-connection*. An SQL-connection is terminated either by executing a <disconnect statement>, or following the last call to an externally-invoked procedure within the last active SQL-client module, or by the last execution of a <direct SQL statement> through the direct invocation of SQL. The mechanism and rules by which an SQL-implementation determines whether a call to an externally-invoked procedure is the last call within the last active SQL-client module and the mechanism and rules by which an SQL-implementation determines whether a direct invocation of SQL is the last execution of a <direct SQL statement> are implementation-defined (IW124).

An SQL-implementation shall support at least one SQL-connection and may require that the SQL-server be identified at the binding time chosen by the SQL-implementation. If an SQL-implementation permits more than one concurrent SQL-connection, then the SQL-agent may connect to more than one SQL-server and select the SQL-server by executing a <set connection statement>.

4.45 SQL-sessions

This Subclause is modified by Subclause 4.13, "SQL-sessions", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 4.10, "SQL-sessions", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.9, "SQL-sessions", in ISO/IEC 9075-14.

4.45.1 General description of SQL-sessions

This Subclause is modified by Subclause 4.13.1, "General description of SQL-sessions", in ISO/IEC 9075-4.

An *SQL-session* spans the execution of a sequence of consecutive SQL-statements invoked either by a single user from a single SQL-agent or by the direct invocation of SQL. At any one time during an SQL-session, exactly one of the SQL-statements in this sequence is being executed and is said to be an *executing statement*. In some cases, an executing statement *ES* causes a nested sequence of consecutive SQL-statements to be executed as a direct result of *ES*; during that time, exactly one of these is also an executing statement and it in turn might similarly involve execution of a further nested sequence, and so on, indefinitely. An executing statement *ES* such that no statement is executing as a direct result of *ES* is called the *innermost executing statement* of the SQL-session.

An SQL-session is associated with an SQL-connection. The SQL-session associated with the default SQL-connection is called the *default SQL-session*. An SQL-session is either *current* or *dormant*. The *current SQL-session* is the SQL-session associated with the current SQL-connection. A *dormant SQL-session* is an SQL-session that is associated with a dormant SQL-connection.

Within an SQL-session, SQL-client module declared local temporary tables are effectively created by <temporary table declaration>s contained in <SQL-client module definition>s. SQL-client module declared local temporary tables are accessible only to invocations of <externally-invoked procedure>s in the SQL-client module in which they are declared. The definitions of SQL-client module declared local temporary tables persist until the end of the SQL-session.

04 Within an SQL-session, locators are effectively created when a host parameter, a host variable, or an SQL parameter of an external routine that is specified as a binary large object locator, a character large object locator, a user-defined type locator, an array locator, or a multiset locator is assigned a value of binary large object type, character large object type, user-defined type, array type, or multiset type, respectively. These locators are part of the SQL-session context. A locator may be either valid or invalid. All locators remaining valid at the end of an SQL-session are marked invalid on termination of that SQL-session. A host variable that is a locator may be *holdable* or *non-holdable*.

4.45.2 SQL-session identification

An SQL-session has a unique implementation-dependent (UV068) SQL-session identifier. This SQL-session identifier is different from the SQL-session identifier of any other concurrent SQL-session. The SQL-session identifier is used to effectively define implementation-defined (IV073) schemas that contain the instances of any global temporary tables, created local temporary tables, or declared local temporary tables within the SQL-session.

An SQL-session is started as a result of successful execution of a <connect statement>, which sets the initial SQL-session user identifier to the value of the implicit or explicit <connection user name> contained in the <connect statement>.

An SQL-session initially has no SQL-session role name.

An SQL-session has an original time zone displacement and a current default time zone displacement, which are values of data type INTERVAL HOUR TO MINUTE. Both the original time zone displacement and the current default time zone displacement are initially set to the same implementation-defined (ID044) value. The current default time zone displacement can subsequently be changed by successful execution of a <set local time zone statement>. The original time zone displacement cannot be changed. It is also possible to set the current default time zone displacement to equal the value of the original time zone displacement.

An SQL-session has a default catalog name that is used to effectively qualify unqualified <schema name>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The default catalog name is initially set to an implementation-defined (ID045) value but can subsequently be changed by the successful execution of a <set catalog statement> or <set schema statement>. The default catalog name of an SQL-session may be determined by using the <general value specification> CURRENT_CATALOG.

An SQL-session has a default unqualified schema name that is used to effectively qualify unqualified <schema qualified name>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The default unqualified schema name is initially set to an implementation-defined (ID046) value but can subsequently be changed by the successful execution of a <set schema statement>. The default unqualified schema name of an SQL-session may be determined by using the <general value specification> CURRENT_SCHEMA.

4.45.3 SQL-session properties

This Subclause is modified by Subclause 4.10.1, "SQL-session properties", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 4.9.1, "SQL-session properties", in ISO/IEC 9075-14.

An SQL-session has an SQL-path that is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or are contained in <direct SQL statement>s when those statements are invoked directly. The SQL-path is initially set to an implementation-defined (ID047) value, but can subsequently be changed by the successful execution of a <set path statement>.

The text defining the SQL-path can be referenced by using the <general value specification> CURRENT_PATH.

An SQL-session has a default transform group name and one or more user-defined type name—transform group name pairs that are used to identify the group of transform functions for every user-defined type that is referenced in <preparable statement>s when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or is referenced in <direct SQL statement>s when those statements are invoked directly. The transform group name for a given

user-defined type name is initially set to an implementation-defined (ID048) value but can subsequently be changed by the successful execution of a <set transform group statement>.

The text defining the transform group names associated with the SQL-session can be referenced using two mechanisms: the <general value specification> "CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>", which evaluates to the name of the transform group associated with the specified data type, and the <general value specification> "CURRENT_DEFAULT_TRANSFORM_GROUP", which evaluates to the name of the transform group associated with all types that have no type-specific transform group specified for them.

An SQL-session has a default character set name that is used to identify the character set in which <preparable statement>s are represented when those statements are prepared in the current SQL-session by either an <execute immediate statement> or a <prepare statement> or in which <direct SQL statement>s are represented when those statements are invoked directly. The default character set name is initially set to an implementation-defined (ID049) value but can subsequently be changed by the successful execution of a <set names statement>.

14 For each character set known to the SQL-implementation, an SQL-session has at most one SQL-session collation for that character set, to be used when the rules of Subclause 9.15, "Collation determination", are applied. There are no SQL-session collations at the start of an SQL-session. The SQL-session collation for a character set can be set or changed by the successful execution of a <set session collation statement>.

An SQL-session has a *subject table restriction flag* that is initially set to *False* and a *restricted subject table name list* that is set to an empty list when the SQL-session is started. At the start of an evaluation of a <data change delta table> *DCDT* containing a <result option> that specifies FINAL, the subject table restriction flag is set to *True* and the name of the subject table of the <data change statement> immediately contained in *DCDT* is added to the restricted subject table name list. After the evaluation of *DCDT*, the name of the subject table of the <data change statement> immediately contained in *DCDT* is removed from the restricted subject table name list. If the restricted subject table name list becomes empty, the subject table restriction flag is set to *False*.

An SQL-invoked routine is *active* as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

At any time during an SQL-session, *containing SQL* is said to be *permitted* or *not permitted*. Similarly, *reading SQL-data* is said to be *permitted* or *not permitted* and *modifying SQL-data* is said to be *permitted* or *not permitted*.

An SQL-session has *enduring characteristics*. The enduring characteristics of an SQL-session are initially the same as the default values for the corresponding SQL-session characteristics. The enduring characteristics are changed by successful execution of a <set session characteristics statement> that specifies one or more enduring characteristics. Enduring characteristics that are not specified in a <set session characteristics statement> are not changed in any way by the successful execution of that statement.

An SQL-session has the following enduring characteristics:

— *enduring transaction characteristics*.

Each of the enduring characteristics can be altered at any time in an SQL-session by executing an appropriate <set session characteristics statement>.

09 14 An SQL-session has a stack of contexts that is preserved when an SQL-session is made dormant and restored when the SQL-session is made active. Each context in the stack comprises:

— The SQL-session identifier.

— The authorization stack.

— The SQL-session user identifier.

— The current transaction access mode.

— The current transaction isolation level.

— The current SQL diagnostics area stack and its contents, along with the current condition area limit.

— For each character set known to the SQL-implementation, the SQL-session collation, if any.

— The original time zone displacement.

— The current default time zone displacement.

— The value of the SQL-path for the current SQL-session.

— The text defining the SQL-path.

— The identities of all instances of temporary tables accessible in the SQL-session.

— The current constraint mode for each integrity constraint.

— The cursor instance descriptor of all open cursors accessible in the SQL-session.

— The set of values of all valid locators accessible in the SQL-session.

— The current default catalog name.

— The current default unqualified schema name.

— The current default character set name.

— All prepared statements prepared during the current SQL-session and not deallocated.

— The contents of all SQL dynamic descriptor areas.

— The text defining the default transform group name.

— The text defining the user-defined type name—transform group name pair for each user-defined type explicitly set by the user.

— Each currently available result set sequence *RSS*, along with the specific name of an SQL-invoked procedure *SIP* and the name of the invoker of *SIP* for the invocation causing *RSS* to be brought into existence.

NOTE 100 — Result set sequences are defined in Subclause 4.35.6, "Result sets returned by SQL-invoked procedures".

— The subject table restriction flag.

— The restricted subject table name list.

— An indication of whether a triggered action is executing, initially none executing.

— A *statement timestamp*: either "not set", or a datetime value of type TIMESTAMP(*n*) WITH TIME ZONE, where *n* is the implementation-defined (IL038) maximum <timestamp precision>, which is the date and time at which every <datetime value function> in the statement is effectively evaluated; initially "not set".

— A *statement execution context*.

— A *routine execution context*.

NOTE 101 — The use of the word "current" in the preceding list implies the values that are current in the SQL-session that is to be made dormant, and not the values that will become current in the SQL-session that will become the active SQL-session.

4.45.4 SQL-session context management

There is a stack of SQL-session contexts. There is one cell on this stack when the SQL-session begins. An additional SQL-session context is pushed on the stack for each <routine invocation>, and is removed when the <routine invocation> completes execution. When the new SQL-session context *NSSC* is created, some of its characteristics are copies of the corresponding characteristics in the SQL-session context immediately below *NSSC* on the stack of SQL-session contexts; other characteristics are set to implementation-defined (IV182) values.

Within each SQL-session context, there is a characteristic called the *SQL-session user identifier*, the value of which can be ascertained from an application by the use of the <value specification> SESSION_USER. If the initial SQL-session is begun with a <connect statement> that specifies the user identifier, the value of SESSION_USER is that user identifier; otherwise, the value of SESSION_USER in the first SQL-session context is implementation-defined (ID050).

4.45.5 Execution contexts

Execution contexts augment an SQL-session context to cater for certain special circumstances that might pertain from time to time during invocations of SQL-statements. An execution context is either a statement execution context or a routine execution context. There is always a *statement execution context* and a *routine execution context*. For certain SQL-statements, the statement execution context is always *atomic*; for others, it is always or sometimes non-atomic. Statement execution contexts are described in Subclause 4.41.5, "SQL-statement atomicity and statement execution contexts", and routine execution contexts in Subclause 4.45.6, "Routine execution context".

4.45.6 Routine execution context

A routine execution context consists of:

— An indication as to whether or not an SQL-invoked routine is active.

— An SQL-data access indication, which identifies what SQL-statements, if any, are allowed during the execution of an SQL-invoked routine. The SQL-data access indication is one of the following: does not possibly contain SQL, possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data.

— An identification of the SQL-invoked routine that is active.

— The routine SQL-path derived from the routine SQL-path if the SQL-invoked routine that is active is an SQL routine and from the external routine SQL-path if the SQL-invoked routine that is active is an external routine.

— An indication of whether the active SQL-invoked routine is an SQL-invoked function.

An SQL-invoked routine is active as soon as an SQL-statement executed by an SQL-agent causes invocation of an SQL-invoked routine and ceases to be active when execution of that invocation is complete.

When an SQL-agent causes the invocation of an SQL-invoked routine, a new context for the current SQL-session is created and the values of the current context are preserved. When the execution of that SQL-invoked routine completes, the original context of the current SQL-session is restored and some SQL-session characteristics are reset.

If the routine execution context of the SQL-session indicates that an SQL-invoked routine is active, then the routine SQL-path included in the routine execution context of the SQL-session is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in a <preparable statement> or in a <direct SQL statement>.

4.46 Triggers

4.46.1 General description of triggers

A trigger is a specification for a given action to take place every time a given operation takes place on a given object. The action, known as a *triggered action*, is an <SQL procedure statement> or a list of <SQL procedure statement>s. The object is either a persistent base table or a viewed table, known as the *subject table* of the trigger. The operation, known as a *trigger event*, is either deletion, insertion, or replacement of a collection of rows.

The triggered action is specified to take place either immediately before the triggering event, instead of it, or immediately after it, according to its specified *trigger action time*, BEFORE, INSTEAD OF, or AFTER. The trigger is a *BEFORE trigger*, an *INSTEAD OF trigger*, or an *AFTER trigger*, according to its trigger action time.

A trigger is either a *delete trigger*, an *insert trigger*, or an *update trigger*, according to the nature of its trigger event.

A trigger that is both a delete trigger and an INSTEAD OF trigger is a *delete INSTEAD OF trigger*. A trigger that is both an insert trigger and an INSTEAD OF trigger is an *insert INSTEAD OF trigger*. A trigger that is both an update trigger and an INSTEAD OF trigger is an *update INSTEAD OF trigger*.

Every trigger event arises as a consequence of executing some SQL-data change statement. That consequence might be direct, as for example when the SQL-data change statement is an <insert statement> operating on a base table, or indirect, as for example in the following cases:

— The SQL-data change statement is a <merge statement>.

— The SQL-data change statement operates on the referenced table of some foreign key whose referential action is CASCADE, SET NULL, or SET DEFAULT.

— The SQL-data change statement operates on a viewed table.

A triggered action is permitted to include SQL-data change statements that give rise to trigger events.

A collection of rows being deleted, inserted or replaced is known as a *transition table*. For a delete trigger there is just one transition table, known as an *old transition table*. For an insert trigger there is just one transition table, known as a *new transition table*. For an update trigger there is both an old transition table (the rows being replaced) and a new transition table (the replacement rows), these two tables having the same cardinality.

A reference to "the transition table" of a trigger is ambiguous in the case of an update trigger but whenever such a reference appears in this document it is immaterial to which of the two transition tables it applies.

The triggered action can be specified to take place either just once when the trigger event takes place, in which case the trigger is a *statement-level trigger*, or once for each row of the transition table when the trigger event takes place, in which case the trigger is a *row-level trigger*.

If the triggered action is specified to take place before the event, the trigger is a row-level trigger, and there is a new transition table, then the action can include statements whose effect is to alter the effect of the impending operation.

Special variables make the data in the transition table(s) available to the triggered action. For a statement-level trigger the variable is one whose value is a transition table. For a row-level trigger, the variable is a range variable, known as a *transition variable*. A transition variable ranges over the rows of a transition table, each row giving rise to exactly one execution of the triggered action, with the row in question assigned to the transition variable. A transition variable is either an *old transition variable* or a *new transition variable*, depending on the transition table over whose rows it ranges.

When there are two transition tables, old and new, each row in the new transition table is one that is derived by an update operation applied to exactly one row in the old transition table. Thus there is a 1:1 correspondence between the rows of the two tables. However, this correspondence is visible only to a row-level trigger, each invocation of which is able to access both the old and new transition variables, the new transition variable representing the result of applying the update operation in question to the row in the old transition variable.

A trigger is defined by a <trigger definition>, specifying the name of the trigger, its subject table, its trigger event, its trigger action time, whether it is statement-level or row-level, names as required for referencing transition tables or variables, and the triggered action.

A triggered action is a <triggered SQL statement> that is executed (either once for each affected row, in the case of a row-level trigger, or once for the whole trigger event in the case of a statement-level trigger) immediately before, instead of, or immediately after the trigger event takes place. The execution of a triggered action might cause the triggering of further triggered actions. It does so if it entails execution of an <SQL procedure statement> whose effect causes the trigger event of some trigger to take place.

A trigger is described by a trigger descriptor. A trigger descriptor includes:

— The name of the trigger.

— The name of the subject table.

— The trigger action time (BEFORE, INSTEAD OF, or AFTER).

— The trigger event (INSERT, DELETE, or UPDATE).

— Whether the trigger is a statement-level trigger or a row-level trigger.

— Any old transition variable name, new transition variable name, old transition table name, or new transition table name.

— The triggered action.

— The trigger column list (possibly empty) for the trigger event.

— The triggered action column set of the triggered action.

— The timestamp of creation of the trigger.

If the SQL-implementation does not support Feature T218, "Multiple triggers for the same event executed in the order created", then the *order of execution* of a set of triggers is implementation-defined (IS002); otherwise, the *order of execution* of a set of triggers is ascending by value of their timestamp of creation in their descriptors, such that the oldest trigger executes first. If one or more triggers have the same timestamp value, then their relative order of execution is implementation-defined (IS003).

A triggered action is always executed under the authorization of the owner of the schema that includes the trigger.

4.46.2 Trigger execution

A statement execution context includes a set of *state changes*. Within a statement execution context, each state change is uniquely identified by a trigger event, a subject table, and a *column list*. The trigger event can be DELETE, INSERT, or UPDATE.

A state change *SC* consists of:

— A *set of transitions*.

— A trigger event.

— A subject table.

— A column list.

— A set (initially empty) of statement-level triggers *considered as executed* for *SC*.

— A set of row-level triggers, each paired with the set of rows in *SC* for which it is considered as executed.

What constitutes a transition depends on the trigger event. If the trigger event is DELETE, a transition is a row in the old transition table. If the trigger event is INSERT, a transition is a row in the new transition table. If the trigger event is UPDATE, a transition is a row *OR* in the old transition table paired with a row *NR* in the new transition table, such that *NR* is the row derived by applying a specified update operation to *OR*. *OR* and *NR* are the *old row* and the *new row*, respectively, of the transition.

A statement-level trigger that is considered as executed for a state change *SC* (in a given statement execution context) is not subsequently executed for *SC*.

If a row-level trigger *RLT* is considered as executed for some row *R* in *SC*, then *RLT* is not subsequently executed for *R*.

A (possibly empty) old transition table exists if the trigger event is UPDATE or DELETE. It consists of a copy of each row that is to be updated in or deleted from the subject table. A (possibly empty) new transition table exists if the trigger event is UPDATE or INSERT. It consists of a copy of each row that results from updating a row in the subject table or is to be inserted into the subject table.

A <triggered action> may refer to the old transition table only if an <old transition table name> is specified for it in the <trigger definition>, and to the new transition table only if a <new transition table name> is specified for it in the <trigger definition>.

The <triggered action> of a row-level trigger may refer to a range variable ranging over the rows of the old transition table only if an <old transition variable name> is specified for it in the <trigger definition>. Similarly, the <triggered action> of a row-level trigger may refer to a range variable ranging over the rows of the new transition table only if a <new transition variable name> is specified for it in the <trigger definition>. The scope of a transition variable or transition table name is the <triggered action> of the <trigger definition> that specifies it, excluding any <SQL schema statement>s that are contained in that <triggered action>.

When a statement execution context *SEC* is created, the set of state changes *SSC* in *SEC* is empty. Let SC_j be a state change in *SSC*. Let *TE* be the trigger event (DELETE, INSERT, or UPDATE) of SC_j. Let *ST* be the subject table of SC_j.

If *TE* is INSERT or DELETE, then let *PSC* be a set whose only element is the empty set.

If *TE* is UPDATE, then:

— Let *CL* be the list of columns being updated by *SSC*.

— Let *OC* be the set of column names identifying the columns in *CL*.

— Let *PSC* be the set consisting of the empty set and every subset of the set of column names of *ST* that has at least one column that is in *OC*.

Let *PSCN* be the number of elements in *PSC*. A state change SC_j, for *j* varying from 1 (one) to *PSCN*, identified by *TE*, *ST*, and the *j*-th element in *PSC*, is added to *SSC*, provided that *SSC* does not already contain a state change corresponding to SC_j. Transitions are added to SC_j as specified by the General Rules of Subclause 15.8, "Effect of deleting rows from base tables", Subclause 15.11, "Effect of inserting tables into base tables", and Subclause 15.14, "Effect of replacing rows in base tables".

When a state change SC_j arises in *SSC*, one or more triggers are *activated by* SC_j. A trigger *TR* is activated by SC_j if and only if the subject table of *TR* is the subject table of SC_j, the trigger event of *TR* is the trigger

event of SC_j, and the set of column names listed in the trigger column list of *TR* is equivalent to the set of column names listed in SC_j.

NOTE 102 — The trigger column list is included in the descriptor of *TR*; it is empty if the trigger event is DELETE or INSERT. The trigger column list is also empty if the trigger event is UPDATE, but the <trigger event> of the <trigger definition> that defined *TR* does not specify a <trigger column list>.

For each state change SC_j in *SEC*, the BEFORE triggers activated by SC_j are executed before any of their triggering events take effect. When those triggering events have taken effect, any AFTER triggers activated by the state changes of *SEC* are executed.

The <triggered action> contained in a <trigger definition> for a row-level trigger can refer to columns of old transition variables and new transition variables. Such references can be specified as <column reference>s, which can be <target specification>s and <simple target specification>s when they refer to columns of the new transition variable in the triggered action of a BEFORE row-level trigger.

NOTE 103 — By using such <column reference>s as <assignment target>s (see ISO/IEC 9075-4), the triggered action of a BEFORE trigger is able to cause certain SQL-data change statements to have different effects from those specified in the statements.

When an execution of the <triggered SQL statement> *TSS* of a triggered action is not successful, then an exception condition is raised and the SQL-statement that caused *TSS* to be executed has no effect on SQL-data or schemas.

4.47 Client-server operation

This Subclause is modified by Subclause 4.8, "Client-server operation", in ISO/IEC 9075-3.

When an SQL-agent is active, it is bound in some implementation-defined (IW127) manner to a single SQL-client. That SQL-client processes the explicit or implicit <SQL connection statement> for the first call to an externally-invoked procedure by an SQL-agent. The SQL-client communicates with, either directly or possibly through other agents such as RDA, one or more SQL-servers. RDA is defined in ISO/IEC 9579. An SQL-session involves an SQL-agent, an SQL-client, and a single SQL-server.

SQL-client modules associated with the SQL-agent exist in the SQL-environment containing the SQL-client associated with the SQL-agent.

Called <externally-invoked procedure>s and <direct SQL statement>s containing an <SQL connection statement> or an <SQL diagnostics statement> are processed by the SQL-client. Following the successful execution of a <connect statement> or a <set connection statement>, the SQL-client modules associated with the SQL-agent are effectively materialized with an implementation-dependent (UV069) <SQL-client module name> in the SQL-server. Other called <externally-invoked procedure>s and <direct SQL statement>s are processed by the SQL-server.

03 A call by the SQL-agent to an <externally-invoked procedure> whose <SQL procedure statement> simply contains an <SQL diagnostics statement> fetches information from the specified diagnostics area in the diagnostics area stack associated with the SQL-client. Following the execution of an <SQL procedure statement> by an SQL-server, diagnostic information is passed in an implementation-dependent (UW010) manner into the SQL-agent's diagnostics area stack in the SQL-client. The effect on diagnostic information of incompatibilities between the character repertoires supported by the SQL-client and SQL-server is implementation-dependent (UA051).

4.48 JSON data handling in SQL

4.48.1 Introduction

JSON (an acronym for "JavaScript Object Notation") is both a notation (that is, a syntax) for representing data and a[n implied] data model. JSON is not an object-oriented data model in the classic sense; that is, it does not define sets of classes and methods, type inheritance, or data abstraction. Instead, JSON "objects"

are simple data structures, including arrays. Some sources say that JSON is a serialization of structured data. Its initial intended use was as a data transfer syntax. The syntax of JSON is specified very concisely in RFC 8259.

The first-class components of the JSON data model are JSON values. A JSON value is one of the following: JSON object, JSON array, JSON string, JSON number, or one of the JSON literals: **true**, **false**, and **null**. A JSON object is zero or more name-value pairs and is enclosed in curly braces — {...}. A JSON array is an ordered sequence of zero or more values and is enclosed in square brackets — [...].

In a JSON object, the name-value pairs are separated by commas, and the names are separated from the values by colons. The names are always strings and are enclosed in (double) quotation marks. In a JSON array, the values are also separated by commas. The values in both JSON objects and JSON arrays may be JSON strings, JSON numbers, JSON Booleans (represented by the JSON literals **true** and **false**), JSON nulls (represented by the JSON literal **null**), JSON objects, or JSON arrays. JSON arrays and JSON objects are fully nestable. That is, any JSON value is permitted to be an "atomic" value (a string, number, or literal), a JSON object, or a JSON array.

JSON is sometimes used to represent *associative arrays* — arrays whose elements are addressed by content, not by position. An associative array can be represented in JSON as a JSON object whose members are name-value pairs; the name is used as the "index" into the "array" — that is, to locate the appropriate member in the JSON object — and the value is used as the content of the appropriate member.

Part of JSON's design is that it is inherently schema-less. Any JSON object can be modified by adding new name-value pairs, even with names that were never considered when the object was initially created or designed. Similarly, any JSON array can be modified by changing the number of values in the array. One consequence of JSON's schema-less nature is that it is not possible to determine the validity of JSON data, except by application programs. However, a JSON processing system can determine by direct inspection whether or not a given bit of JSON data is well-formed (that is, whether it obeys the syntax defined in RFC 8259).

NOTE 104 — In the context of SQL, JSON objects and JSON arrays cannot be modified *in situ*. Instead, new JSON objects or JSON arrays are constructed that strongly resemble an existing JSON object or JSON array and then used to replace the existing JSON object or JSON array.

NOTE 105 — For most SQL/JSON functions, JSON data that acts either as an argument or as the result can be represented as character strings or binary strings.

4.48.2 Implied JSON data model

The implied JSON data model comprises JSON text and certain kinds of values represented as JSON text fragments.

NOTE 106 — RFC 8259 specifies the syntax used by JSON to represent data, and implies a data model derived from that syntax. However, that implied data model is insufficiently complete or precise to form the basis for standardization of JSON use in an SQL-environment. Consequently, this document specifies an SQL/JSON data model.

The components of the implied JSON data model are:

— A *JSON text* is a character string or binary string that conforms to the definition of "JSON-text" in RFC 8259.

— A *JSON text fragment* is a substring of a JSON text that conforms to any BNF non-terminal in RFC 8259.

— A *JSON literal* is a JSON text fragment that is any of the key words `true`, `false`, or `null`.

— A *JSON member* is a JSON text fragment that conforms to the definition of member in RFC 8259, section 4, "Objects". If *M* is a JSON member, then *M* matches the BNF production

```
member = string name-separator value
```

The key of *M* is the JSON fragment matching `string` in this production, and the bound value of *M* is the JSON fragment matching `value` in this production.

— A *JSON object* is a JSON text fragment that conforms to the definition of `object` in RFC 8259, section 4, "Objects".

— A *JSON array* is a JSON text fragment that conforms to the definition of `array` in RFC 8259, section 5, "Arrays".

— A *JSON number* is a JSON text fragment that conforms to the definition of `number` in RFC 8259, section 6, "Numbers".

— A *JSON string* is a JSON text fragment that conforms to the definition of `string` in RFC 8259, section 7, "Strings".

— The *value of a JSON string* is the Unicode character string enclosed in the delimiting <double quote>s of a JSON string.

NOTE 107 — Within JSON strings, certain characters are represented using an "escaped notation" that comprises a <reverse solidus> followed by the desired character. For example, a <double quote> in a JSON string is represented by the sequence <reverse solidus><double quote> (`\"`). (The complete list of Unicode characters that can be represented in a JSON string only by such an escape sequence is: ", \, /, backspace, form feed, line feed, carriage return, and tab; in addition, arbitrary Unicode characters can be included by using "`\u`" followed by four hexadecimal digits.) The value of a JSON string is determined after replacing all such escaped sequences with their equivalent Unicode values.

NOTE 108 — The implied JSON data model is specified in terms of the Unicode character strings that contain JSON text. The facilities specified in this document do not utilize the implied JSON data model except when parsing JSON text into the SQL/JSON data model or when serializing values of the SQL/JSON data model into JSON text.

— A *JSON value* is a JSON object, JSON array, JSON number, JSON string, or one of three JSON literals.

4.48.3 SQL/JSON data model

The SQL/JSON data model comprises SQL/JSON items and SQL/JSON sequences. The components of the SQL/JSON data model are:

— An *SQL/JSON item* is defined recursively as any of the following:

- An *SQL/JSON scalar*, defined as a non-null value of any of the following predefined (SQL) types: character string with character set Unicode, numeric, Boolean, or datetime.
- An *SQL/JSON null*, defined as a value that is distinct from any value of any SQL type.

  NOTE 109 — An SQL/JSON null is distinct from the SQL null value.
- An *SQL/JSON array*, defined as an ordered list of zero or more SQL/JSON items, called the *SQL/JSON elements* of the SQL/JSON array.
- An *SQL/JSON object*, defined as an unordered collection of zero or more SQL/JSON members, where an *SQL/JSON member* is a pair whose first value is a character string with character set Unicode and whose second value is an SQL/JSON item. The first value of an SQL/JSON member is called the *key* and the second value is called the *bound value*.

— An *SQL/JSON sequence* is an ordered list of zero or more SQL/JSON items.

The maximum number of SQL/JSON elements in an SQL/JSON array is implementation-defined (IL039).

The maximum number of SQL/JSON members in an SQL/JSON object is implementation-defined (IL039).

The maximum length of the key in an SQL/JSON member is implementation-defined (IL040).

If the declared type of an SQL/JSON element or of the key of an SQL/JSON member is a string type, the maximum length is implementation-defined (IL041).

Two SQL/JSON items are *equality-comparable* if one of them is the SQL/JSON null, or if both are SQL/JSON scalars whose SQL types are comparable and acceptable as operands of an equality operation according to the Syntax Rules of Subclause 9.11, "Equality operations".

Two SQL/JSON items are *order-comparable* if one of them is an SQL/JSON null, or if both are SQL/JSON scalars whose SQL types are comparable and acceptable as operands of an ordering operation according to the Syntax Rules of Subclause 9.14, "Ordering operations".

Two SQL/JSON items *SJI1* and *SJI2* are said to be *equivalent*, defined recursively as follows:

— If *SJI1* and *SJI2* are non-null values of a predefined type and are equal, then *SJI1* and *SJI2* are equivalent.

— If *SJI1* and *SJI2* are the SQL/JSON null, then *SJI1* and *SJI2* are equivalent.

— If *SJI1* and *SJI2* are SQL/JSON arrays, then *SJI1* and *SJI2* are equivalent if they are of the same length, and corresponding elements of *SJI1* and *SJI2* are equivalent.

NOTE 110 — "Corresponding elements" in two arrays are elements that have the same index position in both arrays.

— If *SJI1* and *SJI2* are SQL/JSON objects, then *SJI1* and *SJI2* are equivalent if they have the same number of members and there exists a bijection *B* from *SJI1* to *SJI2* mapping each SQL/JSON member *M* of *SJI1* to an SQL/JSON member *B*(*M*) of *SJI2* such that the key and bound value of *M* are equivalent to the key and bound value of *B*(*M*) of *SJI2*, respectively, for all members *M* of *SJI1*.

The keys of two SQL/JSON members of an SQL/JSON object are *equivalent* if (after converting any escaped characters to their equivalents) they have the same number of Unicode code points and all pairs of corresponding code points are equal.

NOTE 111 — RFC 8259, section 4, "Objects" says "The names within an object SHOULD be unique". Thus, non-unique keys are permitted, but not advised. The WITH UNIQUE KEYS clause in the <JSON predicate> provides a check for uniqueness.

4.48.4 SQL/JSON functions

All manipulation (e.g., retrieval, creation, testing) of SQL/JSON items is performed through a number of SQL/JSON functions, which can be categorized as SQL/JSON retrieval functions and SQL/JSON construction functions. The SQL/JSON retrieval functions are characterized by operating on SQL/JSON data and returning an SQL value other than an SQL/JSON value(possibly a Boolean value) or an SQL/JSON value. The SQL/JSON construction functions return SQL/JSON data created from operations on SQL data or other SQL/JSON data.

The SQL/JSON retrieval functions are:

— <JSON value function>.

— <JSON query>.

— <JSON simplified accessor>.

— <JSON table>.

— <JSON predicate>.

— <JSON exists predicate>.

— <JSON serialize>.

The SQL/JSON construction functions are:

— <JSON parse>.

— <JSON object constructor>.

— <JSON array constructor>.

— <JSON scalar>.

— <JSON object aggregate constructor>.

— <JSON array aggregate constructor>.

A *JSON-returning function* is an SQL/JSON construction function, JSON_QUERY, or any other <value expression> whose declared type is JSON.

4.48.5 Overview of SQL/JSON path language

The SQL/JSON path language is a query language used by certain SQL operators (JSON_VALUE, JSON_QUERY, JSON_TABLE, and JSON_EXISTS, collectively known as the *SQL/JSON query operators*) to query JSON text. The SQL/JSON path language is not, strictly speaking, SQL, though it is embedded in these operators within SQL. Lexically and syntactically, the SQL/JSON path language adopts many features of ECMAScript Language Specification 5.1 Edition, though it is neither a subset nor a superset of ECMAScript Language Specification 5.1 Edition. The semantics of the SQL/JSON path language are primarily SQL semantics.

The SQL/JSON path language is used by the SQL/JSON query operators in the architecture shown in Figure 6, "Architecture of SQL/JSON path language usage".

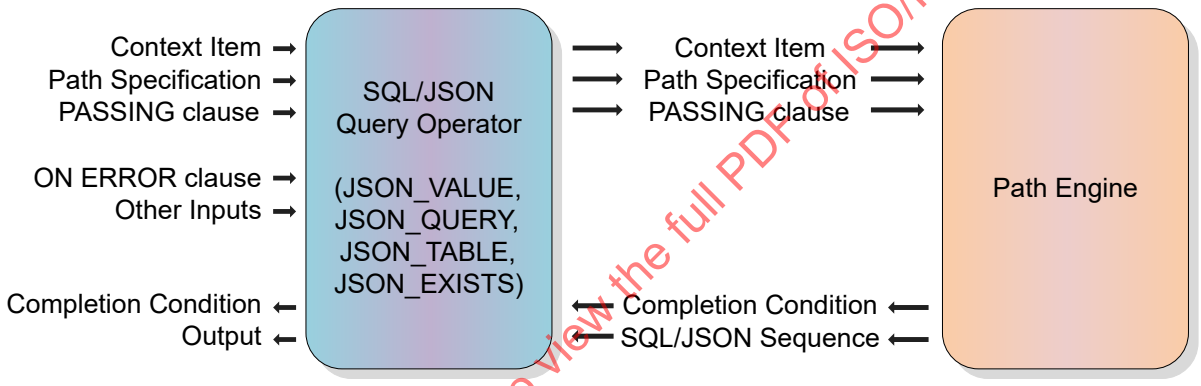

Figure 6 — Architecture of SQL/JSON path language usage

The SQL/JSON query operators share the same first three lines in the diagram, which are expressed syntactically in the <JSON API common syntax> that is used by all SQL/JSON query operators. This framework provides the following inputs to an SQL/JSON query operator:

— A context item (the JSON text to be queried).

— A path specification (the query to perform on the context item; this query is expressed in the SQL/JSON path language specified in Subclause 9.45, "SQL/JSON path language: lexical elements", and in the Format and Syntax Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics").

— A PASSING clause (SQL values to be assigned to variables in the path specification, for example, as values used in predicates within the path specification).

The SQL/JSON operators effectively pass these inputs to a "path engine" that evaluates the path specification, using the context item and the PASSING clause to specify the value of variables in the path specification. The effective behavior of the path engine is specified in the General Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics".

The result of evaluating a path specification on a context item and PASSING clause is a completion condition, and, if the completion condition is *successful completion (00000)*, an SQL/JSON sequence. The SQL/JSON query operators, in their General Rules, use the completion code and SQL/JSON sequence to complete the specific computation specified via the particular SQL/JSON query operator.

Errors can occur at the following junctures in this architecture:

— An error can occur when converting an input. For example, if the context item does not parse as JSON text, then that is an input conversion error.

— An error can occur while processing an SQL/JSON path expression. This category of errors is further subdivided as follows:

  • A structural error occurs when an SQL/JSON path expression attempts to access a non-existent element of an SQL/JSON array or a non-existent member of a JSON object.

  • A non-structural error is any other error during evaluation of an SQL/JSON path expression; for example, divide by zero.

— An error can occur when converting an output.

The SQL operators JSON_VALUE, JSON_QUERY, JSON_TABLE, and JSON_EXISTS provide the following mechanisms to handle these errors:

— The SQL/JSON path language traps any errors that occur during the evaluation of a <JSON filter expression>. Depending on the precise <JSON path predicate> contained in the <JSON filter expression>, the result may be *Unknown*, *True*, or *False*, depending on the outcome of non-error tests evaluated in the <JSON path predicate>.

— The SQL/JSON path language has two modes, strict and lax, which govern structural errors, as follows:

  • In lax mode:

    — If an operation requires an SQL/JSON array but the operand is not an SQL/JSON array, then the operand is first "wrapped" in an SQL/JSON array prior to performing the operation.

    — If an operation requires something other than an SQL/JSON array, but the operand is an SQL/JSON array, then the operand is "unwrapped" by converting its elements into an SQL/JSON sequence prior to performing the operation.

    — After applying the preceding resolutions to structural errors, if there is still a structural error, the result is an empty SQL/JSON sequence.

  • In strict mode, if the structural error occurs within a <JSON filter expression>, then the error handling of <JSON filter expression> applies. Otherwise, a structural error is an unhandled error.

— Non-structural errors outside of a <JSON path predicate> are always unhandled errors, resulting in an exception condition returned from the path engine to the SQL/JSON query operator.

— The SQL/JSON query operators provide an ON ERROR clause to specify the behavior in case of an input conversion error, an unhandled structural error, an unhandled non-structural error, or an output conversion error.

5 Lexical elements

This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-3.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-4.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-9.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-10.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-11.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-13.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-14.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-15.
This Clause is modified by Clause 5, "Lexical elements", in ISO/IEC 9075-16.

5.1 <SQL terminal character>

This Subclause is modified by Subclause 5.1, "<SQL terminal character>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 5.1, "<SQL terminal character>", in ISO/IEC 9075-16.

Function

Define the terminal symbols of the SQL language and the elements of strings.

Format

```
<SQL terminal character> ::=
  <SQL language character>

<SQL language character> ::=
    <simple Latin letter>
  | <digit>
  | <SQL special character>

<simple Latin letter> ::=
    <simple Latin upper-case letter>
  | <simple Latin lower-case letter>

<simple Latin upper-case letter> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
  | P | Q | R | S | T | U | V | W | X | Y | Z

<simple Latin lower-case letter> ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
  | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

10 16 <SQL special character> ::=
    <space>
  | <double quote>
  | <percent>
  | <ampersand>
  | <quote>
  | <left paren>
  | <right paren>
```

```
  | <asterisk>
  | <plus sign>
  | <comma>
  | <minus sign>
  | <period>
  | <solidus>
  | <colon>
  | <semicolon>
  | <left angle bracket>
  | <equals operator>
  | <right angle bracket>
  | <question mark>
  | <left bracket>
  | <right bracket>
  | <circumflex>
  | <underscore>
  | <vertical bar>
  | <left brace>
  | <right brace>
  | <dollar sign>
  | <apostrophe>

<space> ::=
  !! See the Syntax Rules.

<double quote> ::=
  " !! U+0022

<percent> ::=
  % !! U+0025

<ampersand> ::=
  & !! U+0026

<quote> ::=
  ' !! U+0027

<left paren> ::=
  ( !! U+0028

<right paren> ::=
  ) !! U+0029

<asterisk> ::=
  * !! U+002A

<plus sign> ::=
  + !! U+002B

<comma> ::=
  , !! U+002C

<minus sign> ::=
  - !! U+002D

<period> ::=
  . !! U+002E

<solidus> ::=
  / !! U+002F

<reverse solidus> ::=
  \ !! U+005C
```

```
<colon> ::=
  : !! U+003A

<semicolon> ::=
  ; !! U+003B

<left angle bracket> ::=
  < !! U+003C

<equals operator> ::=
  = !! U+003D

<right angle bracket> ::=
  > !! U+003E

<question mark> ::=
  ? !! U+003F

<left bracket or trigraph> ::=
    <left bracket>
  | <left bracket trigraph>

<right bracket or trigraph> ::=
    <right bracket>
  | <right bracket trigraph>

<left bracket> ::=
  [ !! U+005B

<left bracket trigraph> ::=
  ??( !! <U+003F, U+003F, U+0028>

<right bracket> ::=
  ] !! U+005D

<right bracket trigraph> ::=
  ??) !! <U+003F, U+003F, U+0029>

<circumflex> ::=
  ^ !! U+005E

<underscore> ::=
  _ !! U+005F

<vertical bar> ::=
  | !! U+007C

<left brace> ::=
  { !! U+007B

<right brace> ::=
  } !! U+007D

<dollar sign> ::=
  $ !! U+0024

<apostrophe> ::=
  ' !! U+0027
```

Syntax Rules

1) [10] Every character set shall contain a <space> character that is equivalent to U+0020.

Access Rules

None.

General Rules

1) There is a one-to-one correspondence between the symbols contained in <simple Latin upper-case letter> and the symbols contained in <simple Latin lower-case letter> such that, for all *i*, the symbol defined as the *i*-th alternative for <simple Latin upper-case letter> corresponds to the symbol defined as the *i*-th alternative for <simple Latin lower-case letter>.

Conformance Rules

None.

5.2 <token> and <separator>

This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 5.2, "<token> and <separator>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-11.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 5.1, "<token> and <separator>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 5.2, "<token> and <separator>", in ISO/IEC 9075-16.

Function

Specify lexical units (tokens and separators) that participate in SQL language.

Format

```
<token> ::=
    <non-delimiter token>
  | <delimiter token>

<non-delimiter token> ::=
    <regular identifier>
  | <key word>
  | <unsigned numeric literal>
  | <national character string literal>
  | <binary string literal>
  | <large object length token>
  | <Unicode delimited identifier>
  | <Unicode character string literal>
  | <SQL language identifier>

<regular identifier> ::=
  <identifier body>

<identifier body> ::=
  <identifier start> [ <identifier part>... ]

<identifier part> ::=
    <identifier start>
  | <identifier extend>

<identifier start> ::=
  !! See the Syntax Rules.

<identifier extend> ::=
  !! See the Syntax Rules.

<large object length token> ::=
  <digit>... <multiplier>

<multiplier> ::=
    K
  | M
  | G
  | T
  | P

<delimited identifier> ::=
```

```
  <double quote> <delimited identifier body> <double quote>

<delimited identifier body> ::=
  <delimited identifier part>...

<delimited identifier part> ::=
    <non-double quote character>
  | <double quote symbol>

<Unicode delimited identifier> ::=
  U <ampersand> <double quote> <Unicode delimiter body> <double quote>
      <Unicode escape specifier>

<Unicode escape specifier> ::=
  [ UESCAPE <quote> <Unicode escape character> <quote> ]

<Unicode delimiter body> ::=
  <Unicode identifier part>...

<Unicode identifier part> ::=
    <delimited identifier part>
  | <Unicode escape value>

<Unicode escape value> ::=
    <Unicode 4 digit escape value>
  | <Unicode 6 digit escape value>
  | <Unicode character escape value>

<Unicode 4 digit escape value> ::=
  <Unicode escape character> <hexit> <hexit> <hexit> <hexit>

<Unicode 6 digit escape value> ::=
  <Unicode escape character> <plus sign>
      <hexit> <hexit> <hexit> <hexit> <hexit> <hexit>

<Unicode character escape value> ::=
  <Unicode escape character> <Unicode escape character>

<Unicode escape character> ::=
  !! See the Syntax Rules.

<non-double quote character> ::=
  !! See the Syntax Rules.

<double quote symbol> ::=
  "" !! <U+0022, U+0022>

16 <delimiter token> ::=
    <character string literal>
  | <date string>
  | <time string>
  | <timestamp string>
  | <interval string>
  | <delimited identifier>
  | <SQL special character>
  | <not equals operator>
  | <less than operator>
  | <greater than operator>
  | <greater than or equals operator>
  | <less than or equals operator>
  | <concatenation operator>
  | <right arrow>
  | <left bracket trigraph>
  | <right bracket trigraph>
```

```
  | <double colon>
  | <double period>
  | <named argument assignment token>
  | <left brace minus>
  | <right minus brace>
  | <left brace backslash>
  | <right backslash brace>

<not equals operator> ::=
  <> !! <U+003C, U+003E>

<less than operator> ::=
  <left angle bracket>

<greater than operator> ::=
  <right angle bracket>

<greater than or equals operator> ::=
  >= !! <U+003E, U+003D>

<less than or equals operator> ::=
  <= !! <U+003C, U+003D>

<concatenation operator> ::=
  || !! <U+007C, U+007C>

<right arrow> ::=
  -> !! <U+002D, U+003E>

<double colon> ::=
  :: !! <U+003A, U+003A>

<double period> ::=
  .. !! <U+002E, U+002E>

<named argument assignment token> ::=
  => !! <U+003D, U+003E>

<left brace minus> ::=
  {- !! <U+007B, U+002D>

<right minus brace> ::=
  -} !! <U+002D, U+007D>

<left brace backslash> ::=
  {\ !! <U+007B, U+005C>

<right backslash brace> ::=
  \} !! <U+005C, U+007D>

<separator> ::=
  { <comment> | <whitespace> }...

<whitespace> ::=
  !! See the Syntax Rules.

<truncating whitespace> ::=
  !! See the Syntax Rules.

10 <comment> ::=
    <simple comment>
  | <bracketed comment>

<simple comment> ::=
  <simple comment introducer> [ <comment character>... ] <newline>
```

```
<simple comment introducer> ::=
  <minus sign> <minus sign>

<bracketed comment> ::=
  <bracketed comment introducer>
  [ <bracketed comment contents> ]
  <bracketed comment terminator>

<bracketed comment introducer> ::=
  /* !! <U+002F, U+002A>

<bracketed comment terminator> ::=
  */ !! <U+002A, U+002F>

<bracketed comment contents> ::=
  { <comment character> | <separator> }...!! See the Syntax Rules.

<comment character> ::=
    <non-quote character>
  | <quote>

<newline> ::=
  !! See the Syntax Rules.

<key word> ::=
    <reserved word>
  | <non-reserved word>

03 04 09 10 11 13 14 15 16 <reserved word> ::=
    ABS | ABSENT | ACOS | ALL | ALLOCATE | ALTER | AND | ANY | ANY_VALUE | ARE | ARRAY
  | ARRAY_AGG | ARRAY_MAX_CARDINALITY | AS | ASENSITIVE | ASIN | ASYMMETRIC | AT | ATAN
  | ATOMIC | AUTHORIZATION | AVG

  | BEGIN | BEGIN_FRAME | BEGIN_PARTITION | BETWEEN | BIGINT | BINARY
  | BLOB | BOOLEAN | BOTH | BTRIM | BY

  | CALL | CALLED | CARDINALITY | CASCADED | CASE | CAST | CEIL | CEILING
  | CHAR | CHAR_LENGTH | CHARACTER | CHARACTER_LENGTH | CHECK | CLASSIFIER | CLOB
  | CLOSE | COALESCE | COLLATE | COLLECT | COLUMN | COMMIT | CONDITION | CONNECT
  | CONSTRAINT | CONTAINS | CONVERT | COPY | CORR | CORRESPONDING | COS | COSH
  | COUNT | COVAR_POP | COVAR_SAMP | CREATE | CROSS | CUBE | CUME_DIST | CURRENT
  | CURRENT_CATALOG | CURRENT_DATE | CURRENT_DEFAULT_TRANSFORM_GROUP
  | CURRENT_PATH | CURRENT_ROLE | CURRENT_ROW | CURRENT_SCHEMA | CURRENT_TIME
  | CURRENT_TIMESTAMP | CURRENT_TRANSFORM_GROUP_FOR_TYPE | CURRENT_USER | CURSOR | CYCLE

  | DATE | DAY | DEALLOCATE | DEC | DECFLOAT | DECIMAL | DECLARE | DEFAULT | DEFINE
  | DELETE | DENSE_RANK | DEREF | DESCRIBE | DETERMINISTIC | DISCONNECT | DISTINCT
  | DOUBLE | DROP | DYNAMIC

  | EACH | ELEMENT | ELSE | EMPTY | END | END_FRAME | END_PARTITION | END-EXEC
  | EQUALS | ESCAPE | EVERY | EXCEPT | EXEC | EXECUTE | EXISTS | EXP
  | EXTERNAL | EXTRACT

  | FALSE | FETCH | FILTER | FIRST_VALUE | FLOAT | FLOOR | FOR | FOREIGN
  | FRAME_ROW | FREE | FROM | FULL | FUNCTION | FUSION

  | GET | GLOBAL | GRANT | GREATEST | GROUP | GROUPING | GROUPS

  | HAVING | HOLD | HOUR

  | IDENTITY | IN | INDICATOR | INITIAL | INNER | INOUT | INSENSITIVE | INSERT
  | INT | INTEGER | INTERSECT | INTERSECTION | INTERVAL | INTO | IS

  | JOIN | JSON | JSON_ARRAY | JSON_ARRAYAGG | JSON_EXISTS | JSON_OBJECT
  | JSON_OBJECTAGG | JSON_QUERY | JSON_SCALAR | JSON_SERIALIZE | JSON_TABLE
```

```
  | JSON_TABLE_PRIMITIVE | JSON_VALUE

  | LAG | LANGUAGE | LARGE | LAST_VALUE | LATERAL | LEAD | LEADING | LEAST | LEFT | LIKE
  | LIKE_REGEX | LISTAGG | LN | LOCAL | LOCALTIME | LOCALTIMESTAMP | LOG | LOG10 | LOWER
  | LPAD | LTRIM

  | MATCH | MATCH_NUMBER | MATCH_RECOGNIZE | MATCHES | MAX | MEMBER
  | MERGE | METHOD | MIN | MINUTE | MOD | MODIFIES | MODULE | MONTH | MULTISET

  | NATIONAL | NATURAL | NCHAR | NCLOB | NEW | NO | NONE | NORMALIZE | NOT
  | NTH_VALUE | NTILE | NULL | NULLIF | NUMERIC

  | OCCURRENCES_REGEX | OCTET_LENGTH | OF | OFFSET | OLD | OMIT | ON | ONE
  | ONLY | OPEN | OR | ORDER | OUT | OUTER | OVER | OVERLAPS | OVERLAY

  | PARAMETER | PARTITION | PATTERN | PER | PERCENT | PERCENT_RANK
  | PERCENTILE_CONT | PERCENTILE_DISC | PERIOD | PORTION | POSITION | POSITION_REGEX
  | POWER | PRECEDES | PRECISION | PREPARE | PRIMARY | PROCEDURE | PTF

  | RANGE | RANK | READS | REAL | RECURSIVE | REF | REFERENCES | REFERENCING
  | REGR_AVGX | REGR_AVGY | REGR_COUNT | REGR_INTERCEPT | REGR_R2 | REGR_SLOPE
  | REGR_SXX | REGR_SXY | REGR_SYY | RELEASE | RESULT | RETURN | RETURNS
  | REVOKE | RIGHT | ROLLBACK | ROLLUP | ROW | ROW_NUMBER | ROWS
  | RPAD | RTRIM | RUNNING

  | SAVEPOINT | SCOPE | SCROLL | SEARCH | SECOND | SEEK | SELECT | SENSITIVE
  | SESSION_USER | SET | SHOW | SIMILAR | SIN | SINH | SKIP | SMALLINT | SOME | SPECIFIC
  | SPECIFICTYPE | SQL | SQLEXCEPTION | SQLSTATE | SQLWARNING | SQRT | START
  | STATIC | STDDEV_POP | STDDEV_SAMP | SUBMULTISET | SUBSET | SUBSTRING
  | SUBSTRING_REGEX | SUCCEEDS | SUM | SYMMETRIC | SYSTEM | SYSTEM_TIME
  | SYSTEM_USER

  | TABLE | TABLESAMPLE | TAN | TANH | THEN | TIME | TIMESTAMP | TIMEZONE_HOUR
  | TIMEZONE_MINUTE | TO | TRAILING | TRANSLATE | TRANSLATE_REGEX | TRANSLATION | TREAT
  | TRIGGER | TRIM | TRIM_ARRAY | TRUE | TRUNCATE

  | UESCAPE | UNION | UNIQUE | UNKNOWN | UNNEST | UPDATE | UPPER | USER | USING

  | VALUE | VALUES | VALUE_OF | VAR_POP | VAR_SAMP | VARBINARY
  | VARCHAR | VARYING | VERSIONING

  | WHEN | WHENEVER | WHERE | WIDTH_BUCKET | WINDOW | WITH | WITHIN | WITHOUT

  | YEAR

03 | 04 | 09 | 10 | 11 | 13 | 14 | 15 | 16 <non-reserved word> ::=
    A | ABSOLUTE | ACTION | ADA | ADD | ADMIN | AFTER | ALWAYS | ASC
  | ASSERTION | ASSIGNMENT | ATTRIBUTE | ATTRIBUTES

  | BEFORE | BERNOULLI | BREADTH

  | C | CASCADE | CATALOG | CATALOG_NAME | CHAIN | CHAINING | CHARACTER_SET_CATALOG
  | CHARACTER_SET_NAME | CHARACTER_SET_SCHEMA | CHARACTERISTICS | CHARACTERS
  | CLASS_ORIGIN | COBOL | COLLATION | COLLATION_CATALOG | COLLATION_NAME | COLLATION_SCHEMA
  | COLUMNS | COLUMN_NAME | COMMAND_FUNCTION | COMMAND_FUNCTION_CODE | COMMITTED
  | CONDITIONAL | CONDITION_NUMBER | CONNECTION | CONNECTION_NAME | CONSTRAINT_CATALOG
  | CONSTRAINT_NAME | CONSTRAINT_SCHEMA | CONSTRAINTS | CONSTRUCTOR
  | CONTINUE | COPARTITION | CURSOR_NAME

  | DATA | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION | DEFAULTS | DEFERRABLE
  | DEFERRED | DEFINED | DEFINER | DEGREE | DEPTH | DERIVED | DESC | DESCRIPTOR
  | DIAGNOSTICS | DISPATCH | DOMAIN | DYNAMIC_FUNCTION | DYNAMIC_FUNCTION_CODE

  | ENCODING | ENFORCED | ERROR | EXCLUDE | EXCLUDING | EXPRESSION
```

```
| FINAL | FINISH | FIRST | FLAG | FOLLOWING | FORMAT | FORTRAN | FOUND | FULFILL

| G | GENERAL | GENERATED | GO | GOTO | GRANTED

| HIERARCHY

| IGNORE | IMMEDIATE | IMMEDIATELY | IMPLEMENTATION | INCLUDING | INCREMENT | INITIALLY
| INPUT | INSTANCE | INSTANTIABLE | INSTEAD | INVOKER | ISOLATION

| K | KEEP | KEY | KEYS | KEY_MEMBER | KEY_TYPE

| LAST | LENGTH | LEVEL | LOCATOR

| M | MAP | MATCHED | MAXVALUE | MEASURES | MESSAGE_LENGTH | MESSAGE_OCTET_LENGTH
| MESSAGE_TEXT | MINVALUE | MORE | MUMPS

| NAME | NAMES | NESTED | NESTING | NEXT | NFC | NFD | NFKC | NFKD
| NORMALIZED | NULL_ORDERING | NULLABLE | NULLS | NUMBER

| OBJECT | OCCURRENCE | OCTETS | OPTION | OPTIONS | ORDERING | ORDINALITY | OTHERS
| OUTPUT | OVERFLOW | OVERRIDING

| P | PAD | PARAMETER_MODE | PARAMETER_NAME | PARAMETER_ORDINAL_POSITION
| PARAMETER_SPECIFIC_CATALOG | PARAMETER_SPECIFIC_NAME | PARAMETER_SPECIFIC_SCHEMA
| PARTIAL | PASCAL | PASS | PASSING | PAST | PATH | PERMUTE | PIPE | PLACING | PLAN
| PLI | PRECEDING | PRESERVE | PREV | PRIOR | PRIVATE | PRIVILEGES | PRUNE | PUBLIC

| QUOTES

| READ | RELATIVE | REPEATABLE | RESPECT | RESTART | RESTRICT | RETURNED_CARDINALITY
| RETURNED_LENGTH | RETURNED_OCTET_LENGTH | RETURNED_SQLSTATE | RETURNING
| ROLE | ROUTINE | ROUTINE_CATALOG | ROUTINE_NAME | ROUTINE_SCHEMA | ROW_COUNT

| SCALAR | SCALE | SCHEMA | SCHEMA_NAME | SCOPE_CATALOG | SCOPE_NAME
| SCOPE_SCHEMA | SECTION | SECURITY | SELF | SEMANTICS | SEQUENCE | SERIALIZABLE
| SERVER_NAME | SESSION | SETS | SIMPLE | SIZE | SORT_DIRECTION | SOURCE | SPACE
| SPECIFIC_NAME | STATE | STATEMENT | STRING | STRUCTURE | STYLE | SUBCLASS_ORIGIN

| T | TABLE_NAME | TEMPORARY | THROUGH | TIES | TOP_LEVEL_COUNT | TRANSACTION
| TRANSACTION_ACTIVE | TRANSACTIONS_COMMITTED | TRANSACTIONS_ROLLED_BACK
| TRANSFORM | TRANSFORMS | TRIGGER_CATALOG | TRIGGER_NAME | TRIGGER_SCHEMA | TYPE

| UNBOUNDED | UNCOMMITTED | UNCONDITIONAL | UNDER | UNMATCHED | UNNAMED | USAGE
| USER_DEFINED_TYPE_CATALOG | USER_DEFINED_TYPE_CODE | USER_DEFINED_TYPE_NAME
| USER_DEFINED_TYPE_SCHEMA | UTF16 | UTF32 | UTF8

| VIEW

| WORK | WRAPPER | WRITE

| ZONE
```


Syntax Rules

1) An <identifier start> is any character in the Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, or “Nl”.

   NOTE 112 — The Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, and “Nl” are assigned to Unicode characters that are, respectively, upper-case letters, lower-case letters, title-case letters, modifier letters, other letters, and letter numbers.

2) An <identifier extend> is U+00B7, “Middle Dot”, or any character in the Unicode General Category classes “Mn”, “Mc”, “Nd”, or “Pc”.

NOTE 113 — The Unicode General Category classes "Mn", "Mc", "Nd", and "Pc", are assigned to Unicode characters that are, respectively, non-spacing marks, spacing combining marks, decimal numbers, and connector punctuations.

3) <whitespace> is any consecutive sequence of characters each of which satisfies the definition of whitespace found in Clause 3, "Terms and definitions".

4) <truncating whitespace> is an implementation-defined (IV074) subset of the characters included in <whitespace>; the subset shall always include at least <space>.

NOTE 114 — Since the Unicode definition of White_Space is subject to the addition of new characters, this definition prevents an existing conforming SQL-implementation from being made non-conforming by such a change. However, SQL-implementations are expected to align themselves with the most recent Unicode definition in a timely manner.

5) <newline> is the implementation-defined (IV075) end-of-line indicator.

NOTE 115 — <newline> is typically represented by U+000A ("Line Feed") and/or U+000D ("Carriage Return"); however, this representation is not required by the ISO/IEC 9075 series.

6) With the exception of the <space> character explicitly contained in <binary string literal>, <timestamp string>, and <interval string>, a <token>, other than a <character string literal>, a <national character string literal>, a <Unicode character string literal>, a <delimited identifier>, or a <Unicode delimited identifier> shall not contain a <space> character or other <separator>.

7) A <non-double quote character> is any character of the source language character set other than a <double quote> that is not included in the Unicode General Categories "Cc", "Cf", "Cn", "Cs", "Zl", or "Zp".

NOTE 116 — "source language character set" is defined in Subclause 4.10.2, "Source language character set", in ISO/IEC 9075-1.

8) Any <token> may be followed by a <separator>. A <non-delimiter token> shall be followed by a <delimiter token> or a <separator>.

NOTE 117 — If the Format does not allow a <non-delimiter token> to be followed by a <delimiter token>, then that <non-delimiter token> must be followed by a <separator>.

9) There shall be no <separator> separating the <minus sign>s of a <simple comment introducer>.

10) There shall be no <separator> separating any two <digit>s or separating a <digit> and <multiplier> of a <large object length token>.

11) Within a <bracketed comment contents>, any <solidus> immediately followed by an <asterisk> without any intervening <separator> shall be considered to be the <bracketed comment introducer> of a <separator> that is a <bracketed comment>.

NOTE 118 — If conforming programs place a <simple comment> within a <bracketed comment> and that <simple comment> contains the sequence of characters "*/" without a preceding "/*" in the same <simple comment>, that "*/" prematurely terminates the containing <bracketed comment>.

12) SQL text containing one or more instances of <comment> is equivalent to the same SQL text with the <comment> replaced with <newline>.

13) In a <regular identifier>, the number of <identifier part>s shall be less than 128.

14) The <delimited identifier body> of a <delimited identifier> shall not comprise more than 128 <delimited identifier part>s.

15) In a <Unicode delimited identifier>, there shall be no <separator> between the "U" and the <ampersand> nor between the <ampersand> and the <double quote>.

16) In a <Unicode delimited identifier>, the introductory "U" may be represented either in upper-case (as "U") or in lower-case (as "u").

17) <Unicode escape character> shall be a single character from the source language character set other than a <hexit>, <plus sign>, <quote>, <double quote>, or <whitespace>.

18) If the source language character set contains <reverse solidus>, then let *DEC* be <reverse solidus>; otherwise, let *DEC* be an implementation-defined (IV076) character from the source language character set that is not a <hexit>, <plus sign>, <quote>, <double quote>, or <whitespace>.

19) If a <Unicode escape specifier> does not contain <Unicode escape character>, then "UESCAPE '*DEC*'" is implicit.

20) In a <Unicode escape value> there shall be no <separator> between the <Unicode escape character> and the first <hexit>, nor between any of the <hexit>s.

21) The <Unicode delimiter body> of a <Unicode delimited identifier> shall not comprise more than 128 <Unicode identifier part>s.

22) Let *UEC* be the <Unicode escape character>. <Unicode 4 digit escape value> '*UEC*xyzw' is equivalent to the character at the Unicode code point specified by U+xyzw.

23) Let *UEC* be the <Unicode escape character>. <Unicode 6 digit escape value> '*UEC*+xyzwrs' is equivalent to the character at the Unicode code point specified by U+xyzwrs.

NOTE 119 — The 6-hexit notation is derived by taking the UCS-4 notation defined in ISO/IEC 10646:2020 and removing the leading two hexits, whose values are always 0 (zero).

24) <Unicode character escape value> is equivalent to a single instance of <Unicode escape character>.

25) For every <identifier body> *IB* there is exactly one corresponding *case-normal form CNF*. *CNF* is an <identifier body> derived from *IB* as follows.

Let *n* be the number of characters in *IB*. For *i* ranging from 1 (one) to *n*, the *i*-th character M_i of *IB* is transliterated into the corresponding character or characters of *CNF* as follows.

Case:

a) If M_i is a lower-case character or a title case character for which an equivalent upper-case sequence *U* is defined by Unicode, then let *j* be the number of characters in *U*; the next *j* characters of *CNF* are *U*.

b) Otherwise, the next character of *CNF* is M_i.

26) The case-normal form of the <identifier body> of a <regular identifier> is used for purposes such as and including determination of identifier equivalence, representation in the Definition and Information Schemas, and representation in diagnostics areas.

NOTE 120 — The Information Schema and Definition Schema are defined in ISO/IEC 9075-11.

NOTE 121 — Any lower-case letters for which there are no upper-case equivalents are left in their lower-case form.

27) The case-normal form of <regular identifier> shall not be equal, according to the comparison rules in Subclause 8.2, "<comparison predicate>", to any <reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER.

28) Two <regular identifier>s are equivalent if the case-normal forms of their <identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined (IA013) collation *IDC* that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

29) A <regular identifier> and a <delimited identifier> are equivalent if the case-normal form of the <identifier body> of the <regular identifier> and the <delimited identifier body> of the <delimited identifier> (with all occurrences of <quote> replaced by <quote symbol> and all occurrences of <double quote symbol> replaced by <double quote>), considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and *IDC*, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

30) Two <delimited identifier>s are equivalent if their <delimited identifier body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined (IA013) collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

31) Two <Unicode delimited identifier>s are equivalent if their <Unicode delimiter body>s, considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined (IA013) collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

32) A <Unicode delimited identifier> and a <delimited identifier> are equivalent if their <Unicode delimiter body> and <delimited identifier body>, respectively, each considered as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER and an implementation-defined (IA013) collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

33) A <regular identifier> and a <Unicode delimited identifier> are equivalent if the case-normal form of the <identifier body> of the <regular identifier> and the <Unicode delimiter body> of the <Unicode delimited identifier> considered as the repetition of a <character string literal>, each specifying a <character set specification> of SQL_IDENTIFIER and an implementation-defined (IA013) collation that is sensitive to case, compare equally according to the comparison rules in Subclause 8.2, "<comparison predicate>".

34) For the purposes of identifying <key word>s, any <simple Latin lower-case letter> contained in a candidate <key word> shall be effectively treated as the corresponding <simple Latin upper-case letter>.

35) [10] Neither <left brace backslash> nor <right backslash brace> shall be specified.

NOTE 122 — <left brace backslash> and <right backslash brace> are not specified in conforming SQL language. They are not used in the ISO/IEC 9075 series, nor will they be used in any future revision of the ISO/IEC 9075 series. They are intended to be used by APIs (such as JDBC) that manage connections to SQL-implementations to escape API-specific features, which are outside the scope of this document, within SQL-statements.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F391, "Long identifiers", in a <regular identifier>, the number of <identifier part>s shall be less than 18.

2) Without Feature F391, "Long identifiers", the <delimited identifier body> of a <delimited identifier> shall not comprise more than 18 <delimited identifier part>s.

NOTE 123 — Not every character set supported by a conforming SQL-implementation necessarily contains every character associated with <identifier start> and <identifier part> that is identified in the Syntax Rules of this Subclause. No conforming SQL-implementation is required to support in <identifier start> or <identifier part> any character identified in the Syntax Rules of this Subclause unless that character belongs to the character set in use for an SQL-client module or in SQL-data.

3) Without Feature T351, "Bracketed comments", conforming SQL language shall not contain a <bracketed comment>.

4) Without Feature F392, “Unicode escapes in identifiers”, conforming SQL language shall not contain a <Unicode delimited identifier>.

5) Without Feature T043, “Multiplier T”, in conforming SQL language, a <multiplier> shall not be T.

6) Without Feature T044, “Multiplier P”, in conforming SQL language, a <multiplier> shall not be P.

5.3 <literal>

This Subclause is modified by Subclause 5.2, "<literal>", in ISO/IEC 9075-14.

Function

Specify a non-null value.

Format

```
<literal> ::=
    <signed numeric literal>
  | <general literal>

<unsigned literal> ::=
    <unsigned numeric literal>
  | <general literal>

<general literal> ::=
    <character string literal>
  | <national character string literal>
  | <Unicode character string literal>
  | <binary string literal>
  | <datetime literal>
  | <interval literal>
  | <boolean literal>

<character string literal> ::=
  [ <introducer> <character set specification> ]
      <quote> [ <character representation>... ] <quote>
      [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<introducer> ::=
  <underscore>

<character representation> ::=
    <non-quote character>
  | <quote symbol>

<non-quote character> ::=
  !! See the Syntax Rules.

<quote symbol> ::=
  <quote> <quote>

<national character string literal> ::=
  N <quote> [ <character representation>... ]
      <quote> [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<Unicode character string literal> ::=
  [ <introducer> <character set specification> ]
      U <ampersand> <quote> [ <Unicode representation>... ] <quote>
      [ { <separator> <quote> [ <Unicode representation>... ] <quote> }... ]
      <Unicode escape specifier>

<Unicode representation> ::=
    <character representation>
  | <Unicode escape value>

<binary string literal> ::=
  X <quote> [ <space>... ] [ { <hexit> [ <space>... ] <hexit> [ <space>... ] }... ] <quote>
```

```
      [ { <separator> <quote> [ <space>... ] [ { <hexit> [ <space>... ]
      <hexit> [ <space>... ] }... ] <quote> }... ]

<hexit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

<octal digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<binary digit> ::=
    0 | 1

<signed numeric literal> ::=
  [ <sign> ] <unsigned numeric literal>

<unsigned numeric literal> ::=
    <exact numeric literal>
  | <approximate numeric literal>

<exact numeric literal> ::=
    <unsigned integer>
  | <unsigned decimal integer> <period> [ <unsigned decimal integer> ]
  | <period> <unsigned decimal integer>

<sign> ::=
    <plus sign>
  | <minus sign>

<approximate numeric literal> ::=
  <mantissa> E <exponent>

<mantissa> ::=
  <exact numeric literal>

<exponent> ::=
  <signed decimal integer>

<signed integer> ::=
  [ <sign> ] <unsigned integer>

<signed decimal integer> ::=
  [ <sign> ] <unsigned decimal integer>

<unsigned integer> ::=
    <unsigned decimal integer>
  | <unsigned hexadecimal integer>
  | <unsigned octal integer>
  | <unsigned binary integer>

<unsigned decimal integer> ::=
  <digit> [ { [ <underscore> ] <digit> }... ]

<unsigned hexadecimal integer> ::=
  0X { [ <underscore> ] <hexit> }...

<unsigned octal integer> ::=
  0O { [ <underscore> ] <octal digit> }...

<unsigned binary integer> ::=
  0B { [ <underscore> ] <binary digit> }...

<datetime literal> ::=
    <date literal>
  | <time literal>
  | <timestamp literal>
```

```
<date literal> ::=
  DATE <date string>

<time literal> ::=
  TIME <time string>

<timestamp literal> ::=
  TIMESTAMP <timestamp string>

<date string> ::=
  <quote> <unquoted date string> <quote>

<time string> ::=
  <quote> <unquoted time string> <quote>

<timestamp string> ::=
  <quote> <unquoted timestamp string> <quote>

<time zone interval> ::=
  <sign> <hours value> <colon> <minutes value>

<date value> ::=
  <years value> <minus sign> <months value> <minus sign> <days value>

<time value> ::=
  <hours value> <colon> <minutes value> <colon> <seconds value>

<interval literal> ::=
  INTERVAL [ <sign> ] <interval string> <interval qualifier>

<interval string> ::=
  <quote> <unquoted interval string> <quote>

<unquoted date string> ::=
  <date value>

<unquoted time string> ::=
  <time value> [ <time zone interval> ]

<unquoted timestamp string> ::=
  <unquoted date string> <space> <unquoted time string>

<unquoted interval string> ::=
  [ <sign> ] { <year-month literal> | <day-time literal> }

<year-month literal> ::=
    <years value> [ <minus sign> <months value> ]
  | <months value>

<day-time literal> ::=
    <day-time interval>
  | <time interval>

<day-time interval> ::=
  <days value> [ <space> <hours value> [ <colon> <minutes value>
      [ <colon> <seconds value> ] ] ]

<time interval> ::=
    <hours value> [ <colon> <minutes value> [ <colon> <seconds value> ] ]
  | <minutes value> [ <colon> <seconds value> ]
  | <seconds value>

<years value> ::=
  <datetime value>
```

```
<months value> ::=
  <datetime value>

<days value> ::=
  <datetime value>

<hours value> ::=
  <datetime value>

<minutes value> ::=
  <datetime value>

<seconds value> ::=
  <seconds integer value> [ <period> [ <seconds fraction> ] ]

<seconds integer value> ::=
  <unsigned decimal integer>

<seconds fraction> ::=
  <unsigned decimal integer>

<datetime value> ::=
  <unsigned decimal integer>

<boolean literal> ::=
    TRUE
  | FALSE
  | UNKNOWN
```

Syntax Rules

1) In a <character string literal> or <national character string literal>, the sequence:

```
<quote> <character representation>... <quote><separator>
<quote> <character representation>... <quote>
```

is equivalent to the sequence

```
<quote> <character representation>... <character representation>... <quote>
```

NOTE 124 — The <character representation>s in the equivalent sequence are in the same sequence and relative sequence as in the original <character string literal>.

2) In a <Unicode character string literal>, the sequence:

```
<quote> <Unicode representation>... <quote>
<separator> <quote> <Unicode representation>... <quote>
```

is equivalent to the sequence:

```
<quote> <Unicode representation>... <Unicode representation>... <quote>
```

3) In a <Unicode character string literal>, the introductory "U" may be represented either in upper-case (as "U") or in lower-case (as "u").

4) In a <binary string literal>, the sequence

```
<quote> [ <space>... ] { <hexit> [ <space>... ]
<hexit> [ <space>... ] }... <quote>
```

is equivalent to the sequence

```
<quote> { <hexit> <hexit> }... <quote>
```

NOTE 125 — The <hexit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <binary string literal>.

5) In a <binary string literal>, the sequence

```
<quote> { <hexit> <hexit> }... <quote> <separator>
<quote> { <hexit> <hexit> }... <quote>
```

is equivalent to the sequence

```
<quote> { <hexit> <hexit> }... { <hexit> <hexit> }... <quote>
```

NOTE 126 — The <hexit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <binary string literal>.

6) In a <binary string literal>, the introductory "X" may be represented either in upper-case (as "X") or in lower-case (as "x").

7) In a <character string literal>, <national character string literal>, <Unicode character string literal>, or <binary string literal>, a <separator> shall contain a <newline>.

8) A <national character string literal> is equivalent to a <character string literal> with the "N" replaced by "<introducer><character set specification>", where "<character set specification>" is an implementation-defined (IV077) <character set name>.

9) In a <national character string literal>, the introductory "N" may be represented either in upper-case (as "N") or in lower-case (as "n").

10) In a <Unicode character string literal> that specifies "<introducer><character set specification>", there shall be no <separator> between the <introducer> and the <character set specification>.

11) In a <Unicode character string literal>, there shall be no <separator> between the "U" and the <ampersand> nor between the <ampersand> and the <quote>.

12) The character set of a <Unicode character string literal> that specifies "<introducer><character set specification>" is the character set specified by the <character set specification>. The character set of a <Unicode character string literal> that does not specify "<introducer><character set specification>" is the character set of the SQL-client module that contains the <Unicode character string literal>.

13) A <Unicode character string literal> is equivalent to a <character string literal> in which every <Unicode escape value> has been replaced with the equivalent Unicode character. The set of characters contained in the <Unicode character string literal> shall be wholly contained in the character set of the <Unicode character string literal>.

NOTE 127 — The requirement for "wholly contained" applies after the replacement of <Unicode escape value>s with equivalent Unicode characters.

14) Each <character representation> is a character of the source language character set. The value of a <character string literal>, viewed as a string in the source language character set, shall be equivalent to a character string of the implicit or explicit character set of the <character string literal> or <national character string literal>.

NOTE 128 — "source language character set" is defined in Subclause 4.10.1, "Host languages", in ISO/IEC 9075-1.

15) A <non-quote character> is one of:

a) Any character of the source language character set other than a <quote>.

b) Any character other than a <quote> in the character set identified by the <character set specification> or implied by "N".

16) Case:

a) If a <character set specification> is not specified in a <character string literal>, then the set of characters contained in the <character string literal> shall be wholly contained in the character set of the <SQL-client module definition> that contains the <character string literal>.

b) Otherwise, there shall be no <separator> between the <introducer> and the <character set specification>, and the set of characters contained in the <character string literal> shall be wholly contained in the character set specified by the <character set specification>.

17) The declared type of a <character string literal> is fixed-length character string. The length of a <character string literal> is the number of <character representation>s that it contains. Each <quote symbol> contained in <character string literal> represents a single <quote> in both the value and the length of the <character string literal>. The two <quote>s contained in a <quote symbol> shall not be separated by any <separator>.

NOTE 129 — <character string literal>s are allowed to be zero-length strings (i.e., to contain no characters) even though it is not permitted to declare a <data type> that is CHARACTER with <character length> 0 (zero).

18) The character set of a <character string literal> is

Case:

a) If the <character string literal> specifies a <character set specification>, then the character set specified by that <character set specification>.

b) Otherwise, the character set of the SQL-client module that contains the <character string literal>.

19) The declared type collation of a <character string literal> is the character set collation, and the collation derivation is *implicit*.

20) It is implementation-defined (IV191) whether the declared type of a <binary string literal> is a fixed-length binary string type, a variable-length binary string type, or a binary large object string type. Each <hexit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hexit>s a, b, c, d, e, and f have respectively the same values as the <hexit>s A, B, C, D, E, and F.

21) The maximum number of <digit>s immediately contained in an <unsigned integer> is implementation-defined (IL063). The value of an <unsigned integer> is an *unsigned integer*.

22) An <unsigned decimal integer> that immediately contains <underscore>s is equivalent to the same <unsigned decimal integer> with every <underscore> removed.

23) An <unsigned hexadecimal integer> that immediately contains <underscore>s is equivalent to the same <unsigned hexadecimal integer> with every <underscore> removed.

24) An <unsigned octal integer> that immediately contains <underscore>s is equivalent to the same <unsigned octal integer> with every <underscore> removed.

25) An <unsigned binary integer> that immediately contains <underscore>s is equivalent to the same <unsigned binary integer> with every <underscore> removed.

26) In an <unsigned hexadecimal integer>, <unsigned octal integer>, or <unsigned binary integer>, the radix indicators “X”, “O”, and “B” may be represented either in upper-case (as “X”, “O”, “B”) or in lower-case (as “x”, “o”, “b”).

27) In an <unsigned hexadecimal integer>, <unsigned octal integer>, or <unsigned binary integer>, there shall be no <separator> between the radix indicators “0X”, “0O”, “0B” and the first <hexit>, <octal digit>, <binary digit>, or <underscore>.

28) An <exact numeric literal> without a <period> has an implied <period> following the last <digit>.

29) The declared type of an <exact numeric literal> *ENL* is an implementation-defined (IV048) exact numeric type whose scale is the number of <digit>s to the right of the <period>. There shall be an exact numeric type capable of representing the value of *ENL* exactly.

30) It is implementation-defined (IA090) whether the declared type of an <approximate numeric literal> *ANL* is an implementation-defined (IA090) approximate numeric type or the decimal floating-point type with an implementation-defined (IA090) precision. If the declared type of *ANL* is an approximate numeric type, then the value of *ANL* shall not be greater than the maximum value nor less than the minimum value that can be represented by the approximate numeric types. If the declared type of *ANL* is the decimal floating-point type, then the value of *ANL* shall not be greater than the maximum value nor less than the minimum value that can be represented by the decimal floating-point type.

NOTE 130 — Thus the only syntax error for an <approximate numeric literal> is what is commonly known as "overflow"; there is no syntax error for specifying more significant digits than the SQL-implementation can represent internally, nor for specifying a value that has no exact equivalent in the SQL-implementation's internal representation. ("Underflow", i.e., specifying a non-zero value so close to 0 (zero) that the closest representation in the SQL-implementation's internal representation is 0E0, is a special case of the latter condition, and is not a syntax error.)

31) In an <approximate numeric literal>, the exponent indicator "E" may be represented either in upper-case (as "E") or in lower-case (as "e").

32) A <mantissa> shall not contain an <unsigned integer> that is not an <unsigned decimal integer>.

33) The declared type of a <date literal> is DATE.

34) The declared type of a <time literal> that does not specify <time zone interval> is TIME(*P*) WITHOUT TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise. The declared type of a <time literal> that specifies <time zone interval> is TIME(*P*) WITH TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise.

35) The declared type of a <timestamp literal> that does not specify <time zone interval> is TIMESTAMP(*P*) WITHOUT TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise. The declared type of a <timestamp literal> that specifies <time zone interval> is TIMESTAMP(*P*) WITH TIME ZONE, where *P* is the number of digits in <seconds fraction>, if specified, and 0 (zero) otherwise.

36) If <time zone interval> is not specified, then the effective <time zone interval> of the datetime data type is the current default time zone displacement for the SQL-session.

37) Let *datetime component* be either <years value>, <months value>, <days value>, <hours value>, <minutes value>, or <seconds value>. The datetime components are regarded as literals for datetime fields (see Table 2, "Fields in datetime values"), year-month INTERVAL fields (see Table 4, "Fields in year-month INTERVAL values"), or day-time INTERVAL fields (see Table 4, "Fields in year-month INTERVAL values"), as stated in Table 12, "Interpretation of datetime components".

Table 12 — Interpretation of datetime components

Datetime component	Datetime field, year-month INTERVAL field, or day-time INTERVAL field
<years value>	YEAR
<months value>	MONTH
<days value>	DAY
<hours value>	HOUR
<minutes value>	MINUTE

Datetime component	Datetime field, year-month INTERVAL field, or day-time INTERVAL field
<seconds value>	SECOND

38) Let *N* be the number of <primary datetime field>s in the precision of the <interval literal>, as specified by <interval qualifier>.

The <interval literal> being defined shall contain *N* datetime components.

The declared type of <interval literal> specified with an <interval qualifier> is INTERVAL with the <interval qualifier>.

Each datetime component shall have the precision specified by the <interval qualifier>.

39) Within a <datetime literal>, the <years value> shall contain four digits. The <seconds integer value> and other datetime components, with the exception of <seconds fraction>, shall each contain two digits.

40) Within the definition of a <datetime literal>, the <datetime value>s are constrained by the rules for dates and times according to the Gregorian calendar, with each datetime component interpreted as datetime fields according to Table 12, "Interpretation of datetime components".

41) Within the definition of an <interval literal>, the <datetime value>s are constrained by the rules for intervals according to the Gregorian calendar, with each datetime component interpreted as datetime fields according to Table 12, "Interpretation of datetime components". A <sign>, if specified, applies to all interval fields.

42) Within the definition of an <interval literal> that contains a <year-month literal>, the <interval qualifier> shall not specify DAY, HOUR, MINUTE, or SECOND. Within the definition of an <interval literal> that contains a <day-time literal>, the <interval qualifier> shall not specify YEAR or MONTH.

43) Within the definition of a <datetime literal>, the value of the <time zone interval> shall be a valid time zone displacement as specified in Subclause 4.7.2, "Datetimes".

44) A <seconds integer value> shall not contain an <underscore>.

45) A <seconds fraction> shall not contain an <underscore>.

46) A <datetime value> shall not contain an <underscore>.

47) The declared type of a <boolean literal> is Boolean.

Access Rules

None.

General Rules

1) The value of a <character string literal> is the result of transliterating the sequence of <character representation>s that it contains from the source language character set to the implicit or explicit character set of the <character string literal>.

2) If the character repertoire of a <character string literal> *US* is UCS, then its value is replaced by NORMALIZE(*US*).

3) Except when it is contained in an <exact numeric literal>, the value of an <unsigned decimal integer> is the numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the string of <digit>s that constitutes the <unsigned decimal integer>.

4) The value of an <unsigned hexadecimal integer> is the numeric value determined by the application of the normal mathematical interpretation of positional hexadecimal notation to the string of <hexit>s that constitute the <unsigned hexadecimal integer>.

5) The value of an <unsigned octal integer> is the numeric value determined by the application of the normal mathematical interpretation of positional octal notation to the string of <octal digit>s that constitute the <unsigned octal integer>.

6) The value of an <unsigned binary integer> is the numeric value determined by the application of the normal mathematical interpretation of positional binary notation to the string of <binary digit>s that constitute the <unsigned binary integer>.

7) The value of an <exact numeric literal> is the numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the source characters that constitute the <exact numeric literal>.

8) Let *ANL* be an <approximate numeric literal>. Let *ANDT* be the declared type of *ANL*. Let *ANV* be the product of the exact numeric value represented by the <mantissa> of *ANL* and the number obtained by raising the number 10 to the power of the exact numeric value represented by the <exponent> of *ANL*. If *ANV* is a value of *ANDT*, then the value of *ANL* is *ANV*; otherwise, the value of *ANL* is a value of *ANDT* obtained from *ANV* by rounding or truncation. The choice of whether to round or truncate is implementation-defined (IA212).

9) The <sign> in a <signed numeric literal> or an <interval literal> is a monadic arithmetic operator. The monadic arithmetic operators + and − specify monadic plus and monadic minus, respectively. If neither monadic plus nor monadic minus are specified in a <signed numeric literal> or an <interval literal>, or if monadic plus is specified, then the literal is positive. If monadic minus is specified in a <signed numeric literal> or <interval literal>, then the literal is negative. If <sign> is specified in both possible locations in an <interval literal>, then the sign of the literal is determined by normal mathematical interpretation of multiple sign operators.

10) Let *V* be the integer value of the <unsigned integer> contained in <seconds fraction> and let *N* be the number of digits in the <seconds fraction> respectively. The resultant value of the <seconds fraction> is effectively determined as follows.

    Case:

    a) If <seconds fraction> is specified within the definition of a <datetime literal>, then the effective value of the <seconds fraction> is $V * 10^{-N}$ seconds.

    b) If <seconds fraction> is specified within the definition of an <interval literal>, then let *M* be the <interval fractional seconds precision> specified in the <interval qualifier>.

       Case:

       i) If $N < M$, then let *V1* be $V * 10^{M-N}$; the effective value of the <seconds fraction> is $V1 * 10^{-M}$ seconds.

       ii) If $N > M$, then let *V2* be the integer part of the quotient of $V/10^{N-M}$; the effective value of the <seconds fraction> is $V2 * 10^{-M}$ seconds.

       iii) Otherwise, the effective value of the <seconds fraction> is $V * 10^{-M}$ seconds.

11) The *i*-th datetime component in a <datetime literal> or <interval literal> assigns the value of the datetime component to the *i*-th <primary datetime field> in the <datetime literal> or <interval literal>.

12) If <time zone interval> is specified, then the time and timestamp values in <time literal> and <timestamp literal> represent a datetime in the specified time zone.

13) If <date value> is specified, then it is interpreted as a date in the Gregorian calendar. If <time value> is specified, then it is interpreted as a time of day. Let *DV* be the value of the <datetime literal>, disregarding <time zone interval>.

    Case:

    a) If <time zone interval> is specified, then let *TZI* be the value of the interval denoted by <time zone interval>. The value of the <datetime literal> is *DV* − *TZI*, with time zone displacement *TZI*.

    b) Otherwise, the value of the <datetime literal> is *DV*.

    NOTE 131 — If <time zone interval> is specified, then a <time literal> or <timestamp literal> is interpreted as local time with the specified time zone displacement. However, it is effectively converted to UTC while retaining the original time zone displacement.

    If <time zone interval> is not specified, then no assumption is made about time zone displacement. However, if a time zone displacement be required during subsequent processing, the current default time zone displacement of the SQL-session will be applied at that time.

14) The truth value of a <boolean literal> is *True* if TRUE is specified, is *False* if FALSE is specified, and is *Unknown* if UNKNOWN is specified.

    NOTE 132 — The null value of the Boolean data type is equivalent to the *Unknown* truth value (see Subclause 4.6, "Boolean types").

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean literal>.

2) Without Feature F555, "Enhanced seconds precision", in conforming SQL language, an <unsigned decimal integer> that is a <seconds fraction> that is contained in a <timestamp literal> shall not contain more than 6 <digit>s.

3) Without Feature F555, "Enhanced seconds precision", in conforming SQL language, a <time literal> shall not contain a <seconds fraction>.

4) Without Feature F421, "National character", conforming SQL language shall not contain a <national character string literal>.

5) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval literal>.

6) Without Feature F271, "Compound character literals", in conforming SQL language, a <character string literal> shall contain exactly one repetition of <character representation> (that is, it shall contain exactly one sequence of "<quote> [<character representation>...] <quote>").

7) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <time zone interval>.

8) Without at least one of Feature T045, "BLOB data type", or Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary string literal>.

9) Without Feature T023, “Compound binary literals”, in conforming SQL language, a <binary string literal> shall contain exactly one repetition of “<quote> [{ <hexit> <hexit> }...] <quote>”.

10) Without Feature T024, “Spaces in binary literals”, in conforming SQL language, a <binary string literal> shall not contain a <space>.

11) Without Feature F393, “Unicode escapes in literals”, conforming SQL language shall not contain a <Unicode character string literal>.

12) Without Feature T661, “Non-decimal integer literals”, conforming SQL language shall not contain an <unsigned integer> that is not an <unsigned decimal integer>.

13) Without Feature T662, “Underscores in numeric literals”, conforming SQL language shall not contain an <unsigned decimal integer> that contains an <underscore>, an <unsigned hexadecimal integer> that contains an <underscore>, an <unsigned octal integer> that contains an <underscore>, or an <unsigned binary integer> that contains an <underscore>.

5.4 Names and identifiers

This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 5.3, "Names and identifiers", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 5.2, "Names and identifiers", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 5.3, "Names and identifiers", in ISO/IEC 9075-16.

Function

Specify names.

Format

```
<identifier> ::=
  <actual identifier>

<actual identifier> ::=
    <regular identifier>
  | <delimited identifier>
  | <Unicode delimited identifier>

<SQL language identifier> ::=
  <SQL language identifier start> [ <SQL language identifier part>... ]

<SQL language identifier start> ::=
  <simple Latin letter>

<SQL language identifier part> ::=
    <simple Latin letter>
  | <digit>
  | <underscore>

<authorization identifier> ::=
    <role name>
  | <user identifier>

<table name> ::=
  <local or schema qualified name>

<domain name> ::=
  <schema qualified name>

<schema name> ::=
  [ <catalog name> <period> ] <unqualified schema name>

<unqualified schema name> ::=
  <identifier>

<catalog name> ::=
  <identifier>

<schema qualified name> ::=
  [ <schema name> <period> ] <qualified identifier>

<local or schema qualified name> ::=
  [ <local or schema qualifier> <period> ] <qualified identifier>

<local or schema qualifier> ::=
    <schema name>
```

```
  | <local qualifier>

<qualified identifier> ::=
  <identifier>

<column name> ::=
  <identifier>

<correlation name> ::=
  <identifier>

<query name> ::=
  <identifier>

<SQL-client module name> ::=
  <identifier>

<procedure name> ::=
  <identifier>

<schema qualified routine name> ::=
  <schema qualified name>

<method name> ::=
  <identifier>

<specific name> ::=
  <schema qualified name>

<cursor name> ::=
  <local qualified name>

<local qualified name> ::=
  [ <local qualifier> <period> ] <qualified identifier>

<local qualifier> ::=
  MODULE

<host parameter name> ::=
  <colon> <identifier>

<SQL parameter name> ::=
  <identifier>

<constraint name> ::=
  <schema qualified name>

<external routine name> ::=
    <identifier>
  | <character string literal>

<trigger name> ::=
  <schema qualified name>

<collation name> ::=
  <schema qualified name>

<character set name> ::=
  [ <schema name> <period> ] <SQL language identifier>

<transliteration name> ::=
  <schema qualified name>

<transcoding name> ::=
  <schema qualified name>
```

```
<schema-resolved user-defined type name> ::=
  <user-defined type name>

<user-defined type name> ::=
  [ <schema name> <period> ] <qualified identifier>

<attribute name> ::=
  <identifier>

<field name> ::=
  <identifier>

<savepoint name> ::=
  <identifier>

<sequence generator name> ::=
  <schema qualified name>

<role name> ::=
  <identifier>

<user identifier> ::=
  <identifier>

<connection name> ::=
  <simple value specification>

<SQL-server name> ::=
  <simple value specification>

<connection user name> ::=
  <simple value specification>

<SQL statement name> ::=
    <statement name>
  | <extended statement name>

<statement name> ::=
  <identifier>

<extended statement name> ::=
  [ <scope option> ] <simple value specification>

<dynamic cursor name> ::=
    <conventional dynamic cursor name>
  | <PTF cursor name>

<conventional dynamic cursor name> ::=
    <cursor name>
  | <extended cursor name>

<extended cursor name> ::=
  [ <scope option> ] <simple value specification>

<PTF cursor name> ::=
  PTF <simple value specification>

<descriptor name> ::=
    <conventional descriptor name>
  | <PTF descriptor name>

<conventional descriptor name> ::=
    <non-extended descriptor name>
  | <extended descriptor name>
```

```
<non-extended descriptor name> ::=
  <identifier>

<extended descriptor name> ::=
  [ <scope option> ] <simple value specification>

<scope option> ::=
    GLOBAL
  | LOCAL

<PTF descriptor name> ::=
  PTF <simple value specification>

<window name> ::=
  <identifier>

<row pattern variable name> ::=
  <correlation name>

<measure name> ::=
  <identifier>
```

Syntax Rules

1) In an <SQL language identifier>, the number of <SQL language identifier part>s shall be less than 128.

2) An <SQL language identifier> is equivalent to an <SQL language identifier> in which every letter that is a lower-case letter is replaced by the corresponding upper-case letter or letters. This treatment includes determination of equivalence, representation in the Information and Definition Schemas, representation in diagnostics areas, and similar uses.

   NOTE 133 — The Information Schema and Definition Schema are defined in ISO/IEC 9075-11.

3) An <SQL language identifier> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER, shall not be equal, according to the comparison rules in Subclause 8.2, "<comparison predicate>", to any <reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as the repetition of a <character string literal> that specifies a <character set specification> of SQL_IDENTIFIER.

   NOTE 134 — It is the intention that no <key word> specified in the ISO/IEC 9075 series or revisions thereto ends with an <underscore>.

4) If a <local or schema qualified name> does not contain a <local or schema qualifier>, then

   Case:

   a) 04 If the <local or schema qualified name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

   b) 04 If the <local or schema qualified name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

   c) Otherwise, the <schema name> that is specified or implicit for the SQL-client module is implicit.

5) Let *TN* be a <table name> with a <qualified identifier> *QI* and a <local or schema qualifier> *LSQ*.

Case:

a) 04 If *LSQ* is "MODULE", then *TN* shall be contained in an <SQL-client module definition> that contains, without an intervening <SQL schema statement>, a <temporary table declaration> *TT* whose <table name> has a <qualified identifier> equivalent to *QI*.

b) Otherwise, *LSQ* shall be a <schema name> that identifies a schema that contains a <table definition> or <view definition> whose <table name> has a <qualified identifier> equivalent to *QI*.

6) If a <cursor name> *CN* with a <qualified identifier> *QI* does not contain a <local qualifier>, then the <local qualifier> MODULE is implicit.

7) 04 Let *CN* be a <cursor name>. *CN* shall be contained, without an intervening <SQL schema statement>, in an <SQL-client module definition> whose <module contents> contain a <declare cursor> or <dynamic declare cursor> whose <cursor name> is *CN*.

8) If <user-defined type name> *UDTN* with a <qualified identifier> *QI* is specified, then

Case:

a) If *UDTN* is simply contained in <path-resolved user-defined type name>, then

Case:

i) If *UDTN* contains a <schema name> *SN*, then the schema identified by *SN* shall contain the descriptor of a user-defined type *UDT* such that the <qualified identifier> of *UDT* is equivalent to *QI*. *UDT* is the user-defined type identified by *UDTN*.

ii) Otherwise:

1) Case:

A) 04 If *UDTN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then let *DP* be the SQL-path of the current SQL-session.

B) 04 If *UDTN* is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

C) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains *UDTN*.

2) Let *N* be the number of <schema name>s in *DP*. Let S_i, 1 (one) $\leq i \leq N$, be the *i*-th <schema name> in *DP*.

3) Let the *set of subject types* be the set containing every user-defined type *T* in the schema identified by some S_i, 1 (one) $\leq i \leq N$, such that the <qualified identifier> of *T* is equivalent to *QI*. There shall be at least one type in the set of subject types.

4) Let *UDT* be the user-defined type contained in the set of subject types such that there is no other type *UDT2* for which the <schema name> of the schema that includes the user-defined type descriptor of *UDT2* precedes in *DP* the <schema name> identifying the schema that includes the user-defined type descriptor of *UDT*. *UDTN* identifies *UDT*.

5) The implicit <schema name> of *UDTN* is the <schema name> of the schema that includes the user-defined type descriptor of *UDT*.

b) If *UDTN* is simply contained in <schema-resolved user-defined type name> and *UDTN* does not contain a <schema name>, then

Case:

i) 04 If *UDTN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the implicit <schema name> of *UDTN* is the default <unqualified schema name> for the SQL-session.

ii) 04 If *UDTN* is contained in a <schema definition>, then the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <schema definition>.

iii) Otherwise, the implicit <schema name> of *UDTN* is the <schema name> that is specified or implicit in <SQL-client module definition>.

9) Two <user-defined type name>s are equivalent if and only if they have equivalent <qualified identifier>s and equivalent <schema name>s, regardless of whether the <schema name>s are implicit or explicit.

10) No <unqualified schema name> shall specify DEFINITION_SCHEMA.

11) If a <transcoding name> does not specify a <schema name>, then INFORMATION_SCHEMA is implicit; otherwise, INFORMATION_SCHEMA shall be specified.

12) If a <character set name> does not specify a <schema name>, then

Case:

a) If <character set name> is not immediately contained in:

i) A <character set definition>.

ii) A <drop character set statement>.

then <schema name> INFORMATION_SCHEMA is implicit.

b) Otherwise,

Case:

i) If the <character set name> is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

ii) If the <character set name> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

iii) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

13) If a <schema qualified name> *SQN* other than a <transcoding name> does not contain a <schema name>, then

Case:

a) If exactly one of the following is true:

i) *SQN* is immediately contained in a <collation name> that is not immediately contained in a <collation definition> or in a <drop collation statement>.

ii) *SQN* is immediately contained in a <transliteration name> that is not immediately contained in a <transliteration definition> or in a <drop transliteration statement>.

then <schema name> INFORMATION_SCHEMA is implicit.

b) If *SQN* is immediately contained in a <constraint name> that is contained in a <table definition> or an <alter table statement>, then the explicit or implicit <schema name> of the <table name> of the table identified by the <table definition> or <alter table statement> is implicit.

c) If *SQN* is immediately contained in a <constraint name> that is contained in a <domain definition> or an <alter domain statement>, then the explicit or implicit <schema name> of the <domain name> of the domain identified by the <domain definition> or an <alter domain statement> is implicit.

d) Otherwise,

Case:

i) 04 If *SQN* is contained, without an intervening <schema definition>, in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default <unqualified schema name> for the SQL-session is implicit.

ii) 04 If *SQN* is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

iii) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

14) If a <schema name> does not contain a <catalog name>, then

Case:

a) If the <unqualified schema name> is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then the default catalog name for the SQL-session is implicit.

b) If the <unqualified schema name> is contained in a <module authorization clause>, then an implementation-defined (ID057) <catalog name> is implicit.

c) If the <unqualified schema name> is contained in a <schema definition> other than in a <schema name clause>, then the <catalog name> that is specified or implicit in the <schema name clause> is implicit.

d) If the <unqualified schema name> is contained in a <schema name clause>, then

Case:

i) If the <schema name clause> is contained in an <SQL-client module definition>, then the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.

ii) Otherwise, an implementation-defined (ID056) <catalog name> is implicit.

e) Otherwise, the explicit or implicit <catalog name> contained in the <module authorization clause> is implicit.

15) Two <schema qualified name>s are equivalent if and only if their <qualified identifier>s are equivalent and their <schema name>s are equivalent, regardless of whether the <schema name>s are implicit or explicit.

16) Two <local or schema qualified name>s are equivalent if and only if their <qualified identifier>s are equivalent and either they both specify MODULE or they both specify or imply <schema name>s that are equivalent.

17) Two <character set name>s are equivalent if and only if their <SQL language identifier>s are equivalent and their <schema name>s are equivalent, regardless of whether the <schema name>s are implicit or explicit.

18) 13 Two <schema name>s are equivalent if and only if their <unqualified schema name>s are equivalent and their <catalog name>s are equivalent, regardless of whether the <catalog name>s are implicit or explicit.

19) An <identifier> that is a <correlation name> is associated with a table within a particular scope.

NOTE 135 — The scope of a <correlation name> is defined in the Syntax Rules of Subclause 7.6, "<table reference>", Subclause 11.49, "<trigger definition>", and elsewhere. The definition of scopes make it possible for them to be nested. In different scopes, <correlation name>s that are equivalent <identifier>s can be associated with different tables or with the same table.

20) No <authorization identifier> shall specify "PUBLIC".

21) Those <identifier>s that are valid <authorization identifier>s are implementation-defined (IA030).

22) Those <identifier>s that are valid <catalog name>s are implementation-defined (IA031).

23) The declared type of <SQL-server name>, <connection name>, and <connection user name> shall be character string with an implementation-defined (IV193) character set and shall have an octet length of 128 octets or less.

24) The <simple value specification> of <extended statement name>, <extended cursor name>, <PTF descriptor name>, or <PTF cursor name>, shall not be a <literal>.

25) If an <extended statement name> or <extended cursor name> contains an <SQL parameter reference>, then it shall also contain an explicit <scope option>.

26) The declared type of the <simple value specification> of <extended statement name>, <extended cursor name>, <extended descriptor name>, <PTF descriptor name>, or <PTF cursor name> shall be character string with an implementation-defined (IV192) character set and shall have an octet length of 128 octets or less.

27) In an <extended descriptor name>, <extended statement name>, or <extended cursor name>, if a <scope option> is not specified, then a <scope option> of LOCAL is implicit. If a <scope option> is contained in an <SQL schema statement>, then it shall not contain LOCAL.

28) 04 09 13 If a <descriptor name>contains an <extended descriptor name> that identifies an <SQL parameter reference> and does not specify a <scope option>, then that <SQL parameter reference> is used to supply the value for the <descriptor name>.

NOTE 136 — The previous rule disambiguates between an <extended descriptor name> that is an <SQL parameter reference> and a <non-extended descriptor name> that is an <identifier> and gives precedence to the <SQL parameter reference>.

Access Rules

None.

General Rules

1) A <table name> identifies a table.

2) Within its scope, a <correlation name> identifies a table.

3) Within its scope, a <query name> identifies the table defined or returned by some associated <query expression body>.

4) A <column name> identifies a column.

5) A <domain name> identifies a domain.

6) An <authorization identifier> identifies a set of privileges.

7) An <SQL-client module name> identifies an SQL-client module.

8) A <schema qualified routine name> identifies an SQL-invoked routine.

9) A <method name> identifies an SQL-invoked method *M* whose descriptor is included in the schema that includes the descriptor of the user-defined type that is the type of *M*.

10) A <specific name> identifies an SQL-invoked routine.

11) A <cursor name> identifies a standing cursor, a declared dynamic cursor, or a received cursor.

12) A <host parameter name> identifies a host parameter.

13) An <SQL parameter name> identifies an SQL parameter.

14) An <external routine name> identifies an external routine.

15) A <trigger name> identifies a trigger.

16) A <constraint name> identifies a table constraint, a domain constraint, or an assertion.

17) A <catalog name> identifies a catalog.

18) A <schema name> identifies a schema.

19) A <collation name> identifies a collation.

20) A <character set name> identifies a character set.

21) A <transliteration name> identifies a character transliteration.

22) A <transcoding name> identifies a transcoding. All <transcoding name>s are implementation-defined (IA032).

23) A <connection name> identifies an SQL-connection.

24) A <user-defined type name> identifies a user-defined type.

25) An <attribute name> identifies an attribute of a structured type.

26) A <savepoint name> identifies a savepoint. The scope of a <savepoint name> is the SQL-transaction in which it was defined.

27) A <sequence generator name> identifies a sequence generator.

28) A <field name> identifies a field.

29) A <role name> identifies a role.

30) A <user identifier> identifies a user.

31) If a prepared statement *PSX* is created in SQL-session *SS* by executing a <prepare statement> *PS1* that contains an <extended statement name> *ESN1* whose value at the time of execution is *V*, then, for as long as it exists, *PSX* can be identified by an <extended statement name> *ESN2* in an <SQL procedure statement> *PS2* executed in *SS* if the value of *ESN2* at the time of execution is *V* and the <scope option> of *ESN2* is the same as the <scope option> of *ESN1*. If the <scope option> of *ESN1*

is LOCAL, then *ESN2* identifies *PSX* only if *PS2* is contained in the same <SQL-client module definition> as *PS1*.

NOTE 137 — The "value at the time of execution" is defined in the General Rules of Subclause 20.7, "<prepare statement>".

32) A <dynamic cursor name> that is a <cursor name> is a non-extended name that identifies a declared dynamic cursor in an <SQL dynamic statement>.

33) If a <dynamic cursor name> is an <extended cursor name>, then the value of the <simple value specification> simply contained in the <extended cursor name> is an extended name that identifies an extended dynamic cursor in an <SQL dynamic statement>.

NOTE 138 — The scope of a non-extended name is defined in Subclause 4.32.4, "Dynamic SQL statements and descriptor areas".

34) If a <dynamic cursor name> is a <PTF cursor name>, then the value of the <simple value specification> simply contained in the <PTF cursor name> is a PTF extended name that identifies a PTF dynamic cursor in a <dynamic fetch statement>.

NOTE 139 — The scope of a PTF extended name is defined in Subclause 4.32.4, "Dynamic SQL statements and descriptor areas".

35) A <statement name> is a non-extended name that identifies a prepared statement created by the execution of a <prepare statement>.

36) A <non-extended descriptor name> is a non-extended name that identifies an SQL descriptor area created by the execution of an <allocate descriptor statement>.

37) If an extended dynamic cursor *CSR* is created in SQL-session *SS* by executing an <allocate extended dynamic cursor statement> *ACS* that contains an <extended statement name> *ESN1* whose value at the time of execution is *V*, then, for as long as it exists, *CSR* can be identified by an <extended statement name> *ESN2* in an <SQL procedure statement> *PS2* executed in *SS* if the value of *ESN2* at the time of execution is *V* and the <scope option> of *ESN2* is the same as the <scope option> of *ESN1*. If the <scope option> of *ESN1* is LOCAL, then *ESN2* identifies *CSR* only if *PS2* is contained in the same <SQL-client module definition> as *ACS*.

NOTE 140 — The "value at the time of execution" is defined in the General Rules of Subclause 20.17, "<allocate extended dynamic cursor statement>".

38) If a PTF dynamic cursor *PDC* is created during the execution of a polymorphic table function *PTF* on a virtual processor *VP* with PTF extended name *PEN*, then, for as long as it exists, *PDC* can be identified by a <PTF cursor name> *PCN* in a <dynamic fetch statement> *DFS* executed on *VP* if the value of the <simple value specification> contained in *PCN* at the time of execution of *DFS* is *PEN*.

39) If an SQL descriptor area *SDA* is created in SQL-session *SS* by executing an <allocate descriptor statement> *ADS* that contains an <extended descriptor name> *ESN1* whose value at the time of execution is *V*, then, for as long as it exists, *SDA* can be identified by an <extended descriptor name> *ESN2* in an <SQL procedure statement> *PS2* executed in *SS* if the value of *ESN2* at the time of execution is *V* and the <scope option> of *ESN2* is the same as the <scope option> of *ESN1*. If the <scope option> of *ESN1* is LOCAL, then *ESN2* identifies *SDA* only if *PS2* is contained in the same <SQL-client module definition> as *ADS*.

NOTE 141 — The "value at the time of execution" is defined in the General Rules of Subclause 20.2, "<allocate descriptor statement>".

40) If a PTF descriptor area *PDA* is created during the compilation of a polymorphic table function in SQL-session *SS* with PTF extended name *PEN*, or during execution of a polymorphic table function *PTF* on a virtual processor *VP* in SQL-session *SS* with PTF extended name *PEN*, then, for as long as it exists, *PDA* can be identified by a <PTF descriptor name> *PDN* in an <SQL procedure statement> *PS2* executed in *SS* if the value of the <simple value specification> contained in *PDN* at the time of execution is *PEN*.

41) A <window name> identifies a window.

42) A <row pattern variable name> is a <correlation name> used to qualify the names of columns of a row pattern input table during evaluation of a <row pattern recognition clause> or a <window definition> that specifies window row pattern recognition.

43) 04 09 13 14 15 16 A <measure name> identifies a row pattern measure column or a row pattern measure function.

Conformance Rules

1) Without at least one of Feature R010, "Row pattern recognition: FROM clause", or Feature R020, "Row pattern recognition: WINDOW clause", conforming SQL language shall not contain a <row pattern variable name>.

2) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <savepoint name>.

3) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <role name>.

4) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain a <query name>.

5) Without Feature S023, "Basic structured types", conforming SQL language shall not contain an <attribute name>.

6) Without Feature T051, "Row types", conforming SQL language shall not contain a <field name>.

7) Without Feature F651, "Catalog name qualifiers", conforming SQL language shall not contain a <catalog name>.

8) Without Feature F771, "Connection management", conforming SQL language shall not contain an explicit <connection name>.

9) Without Feature F690, "Collation support", conforming SQL language shall not contain a <collation name>.

10) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transliteration name>.

11) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transcoding name>.

12) Without Feature F821, "Local table references", conforming SQL language shall not contain a <local or schema qualifier> that contains a <local qualifier>.

13) Without Feature F251, "Domain support", conforming SQL language shall not contain a <domain name>.

14) Without Feature F491, "Constraint management", conforming SQL language shall not contain a <constraint name>.

15) Without Feature F461, "Named character sets", conforming SQL language shall not contain a <character set name>.

16) Without Feature T601, "Local cursor references", in conforming SQL language, a <cursor name> shall not contain a <local qualifier>.

17) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain an <extended statement name> or an <extended cursor name>.

18) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain a <conventional descriptor name> that is not a <literal> or a <non-extended descriptor name>.

19) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <SQL statement name>.

20) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain <conventional dynamic cursor name>.

21) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <conventional descriptor name>.

22) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window name>.

23) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <sequence generator name>.

24) Without Feature B035, "Non-extended descriptor names", conforming SQL language shall not contain a <conventional descriptor name> that is a <non-extended descriptor name>.

25) Without Feature B209, "PTF extended names", conforming SQL language shall not contain a <PTF descriptor name> or a <PTF descriptor name>.

6 Scalar expressions

This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-4.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-9.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-10.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-13.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-14.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-15.
This Clause is modified by Clause 6, "Scalar expressions", in ISO/IEC 9075-16.

6.1 <data type>

This Subclause is modified by Subclause 6.1, "<data type>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.1, "<data type>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 6.1, "<data type>", in ISO/IEC 9075-15.

Function

Specify a data type.

Format

```
<data type> ::=
    <predefined type>
  | <row type>
  | <path-resolved user-defined type name>
  | <reference type>
  | <collection type>

09 14 <predefined type> ::=
    <character string type> [ CHARACTER SET <character set specification> ]
        [ <collate clause> ]
  | <national character string type> [ <collate clause> ]
  | <binary string type>
  | <numeric type>
  | <boolean type>
  | <datetime type>
  | <interval type>
  | <JSON type>

<character string type> ::=
    CHARACTER [ <left paren> <character length> <right paren> ]
  | CHAR [ <left paren> <character length> <right paren> ]
  | CHARACTER VARYING [ <left paren> <character maximum length> <right paren> ]
  | CHAR VARYING [ <left paren> <character maximum length> <right paren> ]
  | VARCHAR [ <left paren> <character maximum length> <right paren> ]
  | <character large object type>

<character large object type> ::=
    CHARACTER LARGE OBJECT [ <left paren> <character large object length> <right paren> ]
  | CHAR LARGE OBJECT [ <left paren> <character large object length> <right paren> ]
  | CLOB [ <left paren> <character large object length> <right paren> ]

<national character string type> ::=
```



```
    NATIONAL CHARACTER [ <left paren> <character length> <right paren> ]
  | NATIONAL CHAR [ <left paren> <character length> <right paren> ]
  | NCHAR [ <left paren> <character length> <right paren> ]
  | NATIONAL CHARACTER VARYING [ <left paren> <character maximum length> <right paren> ]
  | NATIONAL CHAR VARYING [ <left paren> <character maximum length> <right paren> ]
  | NCHAR VARYING [ <left paren> <character maximum length> <right paren> ]
  | <national character large object type>

<national character large object type> ::=
    NATIONAL CHARACTER LARGE OBJECT [ <left paren> <character large object length> <right
    paren> ]
  | NCHAR LARGE OBJECT [ <left paren> <character large object length> <right paren> ]
  | NCLOB [ <left paren> <character large object length> <right paren> ]

<binary string type> ::=
    BINARY [ <left paren> <length> <right paren> ]
  | BINARY VARYING [ <left paren> <maximum length> <right paren> ]
  | VARBINARY [ <left paren> <maximum length> <right paren> ]
  | <binary large object string type>

<binary large object string type> ::=
    BINARY LARGE OBJECT [ <left paren> <large object length> <right paren> ]
  | BLOB [ <left paren> <large object length> <right paren> ]

<numeric type> ::=
    <exact numeric type>
  | <approximate numeric type>
  | <decimal floating-point type>

<exact numeric type> ::=
    NUMERIC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | DECIMAL [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | DEC [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
  | SMALLINT
  | INTEGER
  | INT
  | BIGINT

<approximate numeric type> ::=
    FLOAT [ <left paren> <precision> <right paren> ]
  | REAL
  | DOUBLE PRECISION

<decimal floating-point type> ::=
  DECFLOAT [ <left paren> <precision> <right paren> ]

<length> ::=
  <unsigned integer>

<maximum length> ::=
  <length>

<character length> ::=
  <length> [ <char length units> ]

<character maximum length> ::=
  <character length>

<large object length> ::=
    <unsigned integer> [ <multiplier> ]
  | <large object length token>

<character large object length> ::=
  <large object length> [ <char length units> ]
```

```
<char length units> ::=
    CHARACTERS
  | OCTETS

<precision> ::=
  <unsigned integer>

<scale> ::=
  <unsigned integer>

<boolean type> ::=
  BOOLEAN

<datetime type> ::=
    DATE
  | TIME [ <left paren> <time precision> <right paren> ] [ <with or without time zone> ]
  | TIMESTAMP [ <left paren> <timestamp precision> <right paren> ]
       [ <with or without time zone> ]

<with or without time zone> ::=
    WITH TIME ZONE
  | WITHOUT TIME ZONE

<time precision> ::=
  <time fractional seconds precision>

<timestamp precision> ::=
  <time fractional seconds precision>

<time fractional seconds precision> ::=
  <unsigned integer>

<interval type> ::=
  INTERVAL <interval qualifier>

<row type> ::=
  ROW <row type body>

<row type body> ::=
  <left paren> <field definition> [ { <comma> <field definition> }... ] <right paren>

<reference type> ::=
  REF <left paren> <referenced type> <right paren> [ <scope clause> ]

<scope clause> ::=
  SCOPE <table name>

<referenced type> ::=
  <path-resolved user-defined type name>

<path-resolved user-defined type name> ::=
  <user-defined type name>

15 <collection type> ::=
    <array type>
  | <multiset type>

<array type> ::=
  <data type> ARRAY
      [ <left bracket or trigraph> <maximum cardinality> <right bracket or trigraph> ]

<maximum cardinality> ::=
  <unsigned integer>

<multiset type> ::=
```

```
  <data type> MULTISET

<JSON type> ::=
  JSON
```

Syntax Rules

1) CHAR is equivalent to CHARACTER. DEC is equivalent to DECIMAL. INT is equivalent to INTEGER. VARCHAR is equivalent to CHARACTER VARYING. NCHAR is equivalent to NATIONAL CHARACTER. CLOB is equivalent to CHARACTER LARGE OBJECT. NCLOB is equivalent to NATIONAL CHARACTER LARGE OBJECT. VARBINARY is equivalent to BINARY VARYING. BLOB is equivalent to BINARY LARGE OBJECT.

2) "NATIONAL CHARACTER" is equivalent to the corresponding <character string type> with a specification of "CHARACTER SET *CSN*", where "*CSN*" is an implementation-defined (IV194) <character set name>.

3) If <character string type> is specified, then the collation derivation of the resulting character string type is *implicit*.

   Case:

   a) If <collate clause> is specified, then the collation specified by it shall be applicable to the explicit or implicit character set *CS* of the character string type. That collation is the declared type collation of the character string type.

   b) Otherwise, the character set collation of *CS* is the declared type collation of the character string type.

4) The value of a <length> shall be greater than 0 (zero).

5) If <length> is omitted, then a <length> of 1 (one) is implicit.

6) If <maximum length> is omitted, then an implementation-defined (IL007) <maximum length> is implicit.

7) If <character maximum length> is omitted, then an implementation-defined (ID069) <character maximum length> is implicit.

8) If <char length units> is specified, then the character repertoire of the explicit or implicit character set of the character string type shall be UCS.

9) If <character length> *CL* is specified and *CL* does not contain <char length units>, then CHARACTERS is implicit.

10) If <large object length> is omitted, then an implementation-defined (ID068) <large object length> is implicit.

11) The numeric value of a <large object length> *LOL* is determined as follows.

    Case:

    a) If *LOL* immediately contains <unsigned integer> and does not immediately contain <multiplier>, then the numeric value of <large object length> is the numeric value of the specified <unsigned integer>.

    b) If *LOL* immediately contains <large object length token> or immediately contains <unsigned integer> and <multiplier>, then let *D* be the value of the specified <unsigned integer> or the numeric value of the sequence of <digit>s of <large object length token> interpreted as an <unsigned integer>. The numeric value of *LOL* is the numeric value resulting from the multiplication of *D* and *MS*, where *MS* is:

i) If <multiplier> is K, then 1 024.

ii) If <multiplier> is M, then 1 048 576.

iii) If <multiplier> is G, then 1 073 741 824.

iv) If <multiplier> is T, then 1 099 511 627 776.

v) If <multiplier> is P, then 1 125 899 906 842 624.

12) *LOL* shall be greater than 0 (zero).

13) The numeric value of a <character large object length> *CLOL* is the numeric value of the <large object length> contained in *CLOL*.

14) CHARACTER specifies the data type character string.

15) Characters in a character string are numbered beginning with 1 (one).

16) Case:

a) If the explicit or implicit <char length units> is CHARACTERS, then

Case:

i) If neither VARYING nor LARGE OBJECT is specified in <character string type>, then the length in characters of the character string is fixed and is the value of <length>.

ii) If VARYING is specified in <character string type>, then the length in characters of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <length>.

iii) If LARGE OBJECT is specified in a <character string type>, then the length in characters of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

b) Otherwise:

i) If neither VARYING nor LARGE OBJECT is specified in <character string type>, then the length in octets of the character string is fixed and is the value of <length>.

ii) If VARYING is specified in <character string type>, then the length in octets of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <length>.

iii) If LARGE OBJECT is specified in a <character string type>, then the length in octets of the character string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

17) The maximum values of <length> and <large object length> are implementation-defined (IL042). Neither <length> nor <large object length> shall be greater than the corresponding maximum value.

18) If <character string type> is not contained in a <domain definition> or a <column definition> and CHARACTER SET is not specified, then an implementation-defined (ID070) <character set specification> that specifies an implementation-defined (ID070) character set that contains at least every character that is in <SQL language character> is implicit.

NOTE 142 — Subclause 11.34, "<domain definition>", and Subclause 11.4, "<column definition>", specify the result when <character string type> is contained in a <domain definition> or <column definition>, respectively.

19) BINARY specifies the data type binary string.

20) Octets in a binary string are numbered beginning with 1 (one).

21) Case:

a) If neither VARYING nor LARGE OBJECT is specified in a <binary string type>, then the length in octets of the binary string is fixed and is the value of <length>.

b) If VARYING is specified in a <binary string type>, then the length in octets of the binary string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <length>.

c) If LARGE OBJECT is specified in a <binary string type>, then the length in octets of the binary string is variable, with a minimum length of 0 (zero) and a maximum length of the value of <large object length>.

22) If a <precision> is omitted, then an implementation-defined (ID062) <precision> is implicit.

23) The value of a <precision> shall be greater than 0 (zero). If <data type> contains <decimal floating-point type>, then the possible values of <precision> are implementation-defined (IA021).

24) If a <scale> is omitted, then a <scale> of 0 (zero) is implicit.

25) If an <exact numeric type> contains <scale>, then the value of <scale> shall not be greater than the value of <precision> contained in that <exact numeric type>.

26) For each exact numeric type *ENT*, there is an implementation-defined (IV132) exact numeric type, *ENNF*(*ENT*), known as the *normal form* of *ENT* (which may be *ENT* itself), such that:

a) The name of the data type of *ENNF*(*ENT*) is neither DEC nor INT.

b) The name of the data type of *ENNF*(DEC) is the same as the name of the data type of *ENNF*(DECIMAL).

c) The name of the data type of *ENNF*(INT) is the same as the name of the data type of *ENNF*(INTEGER).

d) The precision, scale, and radix of *ENNF*(*ENT*) are the same as the precision, scale, and radix, respectively, of *ENT*.

e) *ENNF*(*ENNF*(*ENT*)) is the same as *ENNF*(*ENT*).

27) For the <exact numeric type>s DECIMAL and NUMERIC:

a) The maximum value of <precision> is implementation-defined (IL043). <precision> shall not be greater than this value.

b) The maximum value of <scale> is implementation-defined (IL043). <scale> shall not be greater than this maximum value.

28) NUMERIC specifies the data type exact numeric, with the decimal precision and scale specified by the <precision> and <scale>.

29) DECIMAL specifies the data type exact numeric, with the decimal scale specified by the <scale> and the implementation-defined (ID063) decimal precision equal to or greater than the value of the specified <precision>.

30) SMALLINT, INTEGER, and BIGINT specify the data type exact numeric, with scale of 0 (zero) and binary or decimal precision. The choice of binary versus decimal precision is implementation-defined (IV032), but the same radix shall be chosen for all three data types. The precision of SMALLINT shall be less than or equal to the precision of INTEGER, and the precision of BIGINT shall be greater than or equal to the precision of INTEGER.

31) FLOAT specifies the data type approximate numeric, with binary precision equal to or greater than the value of the specified <precision>. The maximum value of <precision> is implementation-defined (IL053). <precision> shall not be greater than this value.

32) REAL specifies the data type approximate numeric, with implementation-defined (IV129) precision.

33) DOUBLE PRECISION specifies the data type approximate numeric, with implementation-defined (IV130) precision that is greater than the implementation-defined (IV129) precision of REAL.

34) For the <approximate numeric type>s FLOAT, REAL, and DOUBLE PRECISION, the maximum and minimum values of the exponent are implementation-defined (IL054).

35) DECFLOAT specifies the decimal floating-point type, with decimal precision equal to the value of the specified <precision>.

36) For the decimal floating-point type with a given precision, the maximum and minimum values of the exponent are implementation-defined (IL054).

37) For each approximate numeric type *ANT*, there is an implementation-defined (IV078) approximate numeric type *ANNF*(*ANT*), known as the *normal form* of *ANT* (which may be *ANT* itself), such that:

    a) The precision of *ANNF*(*ANT*) is the same as the precision of *ANT*.

    b) *ANNF*(*ANNF*(*ANT*)) is the same as *ANNF*(*ANT*).

38) If <time precision> is not specified, then 0 (zero) is implicit. If <timestamp precision> is not specified, then 6 is implicit.

39) If <with or without time zone> is not specified, then WITHOUT TIME ZONE is implicit.

40) The maximum value of <time precision> and the maximum value of <timestamp precision> shall be the same implementation-defined (IL045) value that is not less than 6. The values of <time precision> and <timestamp precision> shall not be greater than that maximum value.

41) The length of a DATE is 10 positions. The length of a TIME WITHOUT TIME ZONE is 8 positions plus the <time fractional seconds precision>, plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIME WITH TIME ZONE is 14 positions plus the <time fractional seconds precision> plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIMESTAMP WITHOUT TIME ZONE is 19 positions plus the <time fractional seconds precision>, plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero). The length of a TIMESTAMP WITH TIME ZONE is 25 positions plus the <time fractional seconds precision> plus 1 (one) position if the <time fractional seconds precision> is greater than 0 (zero).

42) An <interval type> specifying an <interval qualifier> whose <start field> and <end field> are both either YEAR or MONTH or whose <single datetime field> is YEAR or MONTH is a *year-month interval* type. An <interval type> that is not a year-month interval type is a *day-time interval* type.

    NOTE 143 — The length of interval data types is specified in the General Rules of Subclause 10.1, "<interval qualifier>".

43) The *i*-th value of an interval data type corresponds to the *i*-th <primary datetime field>.

44) If <data type> is a <reference type>, then at least one of the following conditions shall be true:

    a) There exists a user-defined type descriptor whose user-defined type name is <user-defined type name> *UDTN* simply contained in <referenced type>. *UDTN* shall identify a structured type.

    b) <reference type> is contained in the <member list> of <user-defined type definition> *UDTD* and the <path-resolved user-defined type name> simply contained in <referenced type> is equivalent to the <schema-resolved user-defined type name> contained in *UDTD*.

45) The <table name> contained in a <scope clause> shall identify a referenceable table whose structured type is *UDTN*.

46) The <table name> *STN* specified in <scope clause> identifies the scope of the reference type. This scope consists of every row in the table identified by *STN*.

47) 15 An <array type> *AT* specifies an *array type*. The <data type> immediately contained in *AT* is the *element type* of the array type. The value of the <maximum cardinality> immediately contained in *AT* is the *maximum cardinality* of a site of data type *AT*. If the maximum cardinality is not specified, then an implementation-defined (ID072) maximum cardinality is implicit.

48) A <multiset type> *MT* specifies a *multiset type*. The <data type> immediately contained in *MT* is the *element type* of the multiset type.

49) <row type> specifies the row type.

50) BOOLEAN specifies the Boolean type.

51) JSON specifies the data type JSON.

52) 09 14 If <data type> *DT1* is contained in a <data type> *DT2*, then the *root data type* of *DT1* is the outermost <data type> that contains *DT1*.

Access Rules

1) If <user-defined type name>, <reference type>, <row type>, or <collection type> *TY* is specified, and *TY* is usage-dependent on a user-defined type *UDT*, then

   Case:

   a) If *TY* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include the USAGE privilege on *UDT*.

   b) Otherwise, the current privileges shall include the USAGE privilege on *UDT*.

General Rules

1) If the result of any specification or operation would be a character string value one of whose characters is not in the character set of its declared type, then an exception condition is raised: *data exception — character not in repertoire (22021)*.

2) If any specification or operation attempts to cause an item of a character string type whose character set has a character repertoire of UCS to contain a code point that is a noncharacter code point, then an exception condition is raised: *data exception — non-character in character string (22029)*.

3) If <char length units> other than CHARACTERS is specified, then the conversion of the value of <length> to characters is implementation-defined (IA227).

4) For a <datetime type>,

   Case:

   a) If DATE is specified, then the data type contains the <primary datetime field>s years, months, and days.

   b) If TIME is specified, then the data type contains the <primary datetime field>s hours, minutes, and seconds.

   c) If TIMESTAMP is specified, then the data type contains the <primary datetime field>s years, months, days, hours, minutes, and seconds.

   d) If WITH TIME ZONE is specified, then the data type contains the time zone datetime fields.

NOTE 144 — Within the non-null values of a <datetime type>, the value of the time zone interval is a valid time zone displacement as specified in Subclause 4.7.2, "Datetimes". The range for time zone intervals is larger than many readers expect because it is governed by political decisions in governmental bodies rather than by any natural law; it is subject to change at any time for the same reasons.

NOTE 145 — A <datetime type> contains no other fields than those specified by the preceding Rule.

5) For a <datetime type>, a <time fractional seconds precision> that is an explicit or implicit <time precision> or <timestamp precision> defines the number of decimal digits following the decimal point in the SECOND <primary datetime field>.

6) Table 13, "Valid values for datetime fields", specifies the constraints on the values of the <primary datetime field>s in datetime values.

Table 13 — Valid values for datetime fields

Keyword	Valid values of datetime fields
YEAR	0001 to 9999
MONTH	01 to 12
DAY	Within the range 1 (one) to 31, but further constrained by the value of MONTH and YEAR fields, according to the rules for the Gregorian calendar.
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 61.9(*N*) where "9(*N*)" indicates a sequence of *N* instances of the digit "9" and "*N*" indicates the number of digits specified by <time fractional seconds precision>.
TIMEZONE_HOUR	Ranging from an implementation-defined (IL061) negative number not greater than −12 to an implementation-defined (IL062) positive number not less than +14
TIMEZONE_MINUTE	−59 to 59

NOTE 146 — Datetime data types will allow dates in the Gregorian format to be stored in the date range 0001–01–01 CE through 9999–12–31 CE. The range for SECOND allows for as many as two "leap seconds". Interval arithmetic that involves leap seconds or discontinuities in calendars will produce implementation-defined (IA198) results.

7) If the value of any one or more of the <primary datetime field>s in a datetime value does not satisfy the constraints specified in Table 13, "Valid values for datetime fields", then an exception condition is raised: *data exception — datetime field overflow (22008)*.

8) If The values of TIMEZONE_HOUR and TIMEZONE_MINUTE have opposite signs (one negative and one positive) or the values of either or both TIMEZONE_HOUR or TIMEZONE_MINUTE do not satisfy the constraints specified in Table 13, "Valid values for datetime fields", then an exception condition is raised: *data exception — invalid time zone displacement value (22009)*.

9) An interval value can be zero, positive, or negative.

10) The values of the <primary datetime field>s within an interval data type are constrained as follows:

a) The value corresponding to the first <primary datetime field> is an integer with at most *N* digits, where *N* is the <interval leading field precision>.

b) Table 14, "Valid absolute values for interval fields", specifies the constraints for the absolute values of other <primary datetime field>s in interval values.

c) If an interval value is zero, then all fields of the interval are zero.

d) If an interval value is positive, then all fields of the interval are non-negative and at least one field is positive.

e) If an interval value is negative, then all fields of the interval are non-positive, and at least one field is negative.

Table 14 — Valid absolute values for interval fields

Keyword	Valid values of INTERVAL fields
MONTH	0 to 11
HOUR	0 to 23
MINUTE	0 to 59
SECOND	0 to 59.9(*N*) where "9(*N*)" indicates a sequence of *N* instances of the digit "9" and "*N*" indicates the number of digits specified by <interval fractional seconds precision> in the <interval qualifier>.

11) If the value of any one or more of the <primary datetime field>s within an interval data type do not satisfy the constraints specified in Table 14, "Valid absolute values for interval fields", then an exception condition is raised: *data exception — datetime field overflow (22008)*.

12) If <data type> specifies a character string type, then a character string type descriptor is created, including the following:

a) The name of the data type (either CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT).

b) The length or maximum length in characters of the character string type.

c) The catalog name, schema name, and character set name of the character set of the character string type.

d) The catalog name, schema name, and collation name of the declared type collation of the character string type.

13) If <data type> is a binary string type, then a binary string type descriptor is created, including the following:

a) The name of the data type (BINARY, BINARY VARYING, or BINARY LARGE OBJECT).

b) The length or maximum length in octets of the binary string type.

14) If <data type> *DT* specifies an <exact numeric type>, then a numeric data type descriptor is created for *DT*, including the following:

a) The name of the data type of the normal form of *DT* (NUMERIC, DECIMAL, BIGINT, INTEGER, or SMALLINT).

b) The precision of the normal form of *DT*.

c) The scale of the normal form of *DT*.

d) An indication of whether the precision and scale of the normal form of *DT* are expressed in decimal or binary terms.

e) The name of the data type specified by *DT*. If an abbreviated name was specified (INT or DEC), then the name included in the descriptor is that of corresponding full form (INTEGER or DECIMAL, respectively).

f) The explicit precision of *DT*, if specified.

g) The explicit scale of *DT*, if specified.

15) If <data type> *DT* specifies an <approximate numeric type>, then a numeric data type descriptor is created for *DT* including the following:

a) The name of the data type of the normal form of *DT* (FLOAT, REAL, or DOUBLE PRECISION).

b) The precision of the normal form of *DT*.

c) An indication that the precision is expressed in binary terms.

d) The name of the data type specified by *DT*.

e) The explicit precision of *DT*, if specified.

16) If <data type> specifies <boolean type>, then a Boolean data type descriptor is created, including the name of the Boolean type (BOOLEAN).

17) If <data type> specifies a <datetime type>, then a datetime data type descriptor is created, including the following:

a) The name of the datetime type (DATE, TIME WITHOUT TIME ZONE, TIME WITH TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE).

b) The value of the <time fractional seconds precision>, if DATE is not specified.

18) If <data type> *DT* specifies decimal floating-point type, then a numeric data type descriptor is created for *DT*, including the following:

a) The name of the data type (DECFLOAT).

b) The precision of *DT*.

c) An indication that the precision is expressed in decimal terms.

19) If <data type> specifies an <interval type>, then an interval data type descriptor is created, including the following:

a) The name of the interval data type (INTERVAL).

b) An indication of whether the interval data type is a year-month interval or a day-time interval.

c) The <interval qualifier> simply contained in the <interval type>.

20) If <data type> is a <collection type>, then a collection type descriptor is created. Let *KC* be the kind of collection specified by <collection type>. Let *ET* be the element type of <collection type>. Let *ETD* be the type designator of *ET*. The collection type descriptor includes the type designator *EDT KC*, an indication of *KC*, the descriptor of *ET*.

Case:

a) [15] If *KC* is ARRAY, then the collection type descriptor additionally includes the maximum cardinality.

21) For a <row type> *RT*, the degree of *RT* is initially set to 0 (zero). The General Rules of Subclause 6.2, "<field definition>", specify the degree of *RT* during the definition of the fields of *RT*.

22) If the <data type> is a <row type>, then a row type descriptor is created. The row type descriptor includes a field descriptor for every <field definition> of the <row type>, according to the Syntax Rules and General Rules of Subclause 6.2, "<field definition>", applied to the <field definition>s in the order in which they were specified.

23) A <reference type> identifies a reference type.

24) 14 If <data type> is a <reference type>, then a reference type descriptor is created. Let *RDTN* be the name of the <referenced type>. The reference type descriptor includes the type designator REF(*RDTN*). If a <scope clause> is specified, then the reference type descriptor includes *STN*, identifying the scope of the reference type.

NOTE 147 — The user-defined type descriptor for a user-defined type is created in the General Rules of Subclause 11.51, "<user-defined type definition>".

25) 09 If <data type> specifies <JSON type>, then a JSON data type descriptor is created, including the name of the JSON type (JSON).

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <path-resolved user-defined type name> that identifies a structured type.

2) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean type>.

3) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <time precision> that does not specify 0 (zero).

4) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <timestamp precision> that does not specify either 0 (zero) or 6.

5) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval type>.

6) Without Feature F421, "National character", conforming SQL language shall not contain a <national character string type>.

7) Without Feature F411, "Time zone specification", conforming SQL language shall not contain <with or without time zone>.

8) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <reference type>.

9) Without Feature T051, "Row types", conforming SQL language shall not contain a <row type>.

10) Without Feature S090, "Minimal array support", conforming SQL language shall not contain an <array type>.

11) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset type>.

12) Without Feature S281, "Nested collection types", conforming SQL language shall not contain a collection type that is based on a <data type> that contains a <collection type>.

13) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <scope clause> that is not simply contained in a <data type> that is simply contained in a <column definition>.

14) Without Feature S092, "Arrays of user-defined types", conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

15) Without Feature S093, "Arrays of distinct types", conforming SQL language shall not contain an <array type> that is based on a <data type> that is a distinct type.

16) Without Feature S272, "Multisets of user-defined types", conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <path-resolved user-defined type name>.

17) Without Feature S094, "Arrays of reference types", conforming SQL language shall not contain an <array type> that is based on a <data type> that contains a <reference type>.

18) Without Feature S274, "Multisets of reference types", conforming SQL language shall not contain a <multiset type> that is based on a <data type> that contains a <reference type>.

19) Without Feature S096, "Optional array bounds", conforming SQL language shall not contain an <array type> that does not immediately contain <maximum cardinality>.

20) Without Feature T045, "BLOB data type", conforming SQL language shall not contain a <binary large object string type>.

21) Without Feature T046, "CLOB data type", conforming SQL language shall not contain a <character large object type>, or a <national character large object type>.

22) Without Feature T081, "Optional string types maximum length", conforming SQL language shall not contain a <character string type> that specifies VARYING and does not immediately contain <character maximum length>.

23) Without Feature T062, "Character length units", conforming SQL language shall not contain a <char length units>.

24) Without Feature T071, "BIGINT data type", conforming SQL language shall not contain BIGINT.

25) Without Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary string type> that is not a <binary large object string type>.

26) Without Feature T081, "Optional string types maximum length", conforming SQL language shall not contain a <binary string type> that specifies VARYING and does not immediately contain <maximum length>.

27) Without Feature T076, "DECFLOAT data type", conforming SQL language shall not contain a <decimal floating-point type>.

28) 09 14 15 Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON type>.

6.2 <field definition>

This Subclause is modified by Subclause 6.2, "<field definition>", in ISO/IEC 9075-14.

Function

Define a field of a row type.

Format

```
<field definition> ::=
  <field name> <data type>
```

Syntax Rules

1) Let *RT* be the <row type> that simply contains a <field definition>.

2) The <field name> shall not be equivalent to the <field name> of any other <field definition> simply contained in *RT*.

3) The declared type of the field is <data type>.

4) Let *DT* be the <data type>.

5) If *DT* is CHARACTER or CHARACTER VARYING and does not specify a <character set specification>, then the <character set specification> specified or implicit in the <schema character set specification> is implicit.

Access Rules

None.

General Rules

1) A data type descriptor is created that describes the declared type of the field being defined.

2) The degree of the row type *RT* being defined in the simply containing <row type> is increased by 1 (one).

3) A field descriptor is created that describes the field being defined. The field descriptor includes the following:

   a) The <field name>.

   b) The data type descriptor of the declared type of the field.

   c) The ordinal position of the field.

      NOTE 148 — The ordinal position of the field is equal to the degree of *RT* at the time this <field definition> is being processed.

4) The field descriptor is included in the row type descriptor for *RT*.

Conformance Rules

1) [14]Without Feature T051, "Row types", conforming SQL language shall not contain a <field definition>.

6.3 <value expression primary>

This Subclause is modified by Subclause 6.3, "<value expression primary>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 6.2, "<value expression primary>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 6.1, "<value expression primary>", in ISO/IEC 9075-16.

Function

Specify a value that is syntactically self-delimited.

Format

```
<value expression primary> ::=
    <parenthesized value expression>
  | <non-parenthesized value expression primary>

<parenthesized value expression> ::=
  <left paren> <value expression> <right paren>

14 15 16<non-parenthesized value expression primary> ::=
    <unsigned value specification>
  | <column reference>
  | <set function specification>
  | <window function>
  | <nested window function>
  | <scalar subquery>
  | <case expression>
  | <cast specification>
  | <field reference>
  | <subtype treatment>
  | <method invocation>
  | <static method invocation>
  | <new specification>
  | <attribute or method reference>
  | <reference resolution>
  | <collection value constructor>
  | <array element reference>
  | <multiset element reference>
  | <next value expression>
  | <routine invocation>
  | <greatest or least function>
  | <row pattern navigation operation>
  | <JSON value function>
  | <JSON value constructor>
  | <JSON query>
  | <JSON simplified accessor>

15<collection value constructor> ::=
    <array value constructor>
  | <multiset value constructor>
```

Syntax Rules

1) 14 15 16The declared type of a <value expression primary> is the declared type of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <nested window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value

constructor>, <array element reference>, <multiset element reference>, <next value expression>, <greatest or least function>, <row pattern navigation operation>, <JSON value function>, <JSON value constructor>, <JSON query>, <JSON simplified accessor>, or the effective returns type of the simply contained <routine invocation>.

2) Let *NVEP* be a <non-parenthesized value expression primary> of the form "*A.B C*", where *A* satisfies the Format of <schema name>, *B* satisfies the Format of <identifier>, and *C* satisfies the Format of <SQL argument list>. If *NVEP* satisfies the Format, Syntax Rules, and Access Rules of Subclause 6.18, "<method invocation>", then *NVEP* is treated as a <method invocation>; otherwise, *NVEP* is treated as a <routine invocation>.

NOTE 149 — The formal grammar defined in the Format and Syntax Rules of Subclause 6.18, "<method invocation>", and of Subclause 9.18, "Invoking an SQL-invoked routine", does not necessarily disambiguate between a <method invocation> and the invocation of a regular function. In such cases, the preceding Syntax Rule ensures that a <non-parenthesized value expression primary> that satisfies the Format, Syntax Rules, and Access Rules of Subclause 6.18, "<method invocation>", is treated as a <method invocation>.

3) 15 The declared type of a <collection value constructor> is the declared type of the <array value constructor> or <multiset value constructor> that it immediately contains.

Access Rules

None.

General Rules

1) 14 15 16 The value of a <value expression primary> is the value of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <nested window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, <next value expression>, <greatest or least function>, <row pattern navigation operation>, <JSON value function>, <JSON value constructor>, <JSON query>, <JSON simplified accessor> or <routine invocation>.

2) 15 The value of a <collection value constructor> is the value of the <array value constructor> or <multiset value constructor> that it immediately contains.

Conformance Rules

15 *None.*

6.4 <value specification> and <target specification>

This Subclause is modified by Subclause 6.1, "<value specification> and <target specification>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 6.1, "<value specification> and <target specification>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 6.4, "<value specification> and <target specification>", in ISO/IEC 9075-14.

Function

Specify one or more values, host parameters, SQL parameters, dynamic parameters, or host variables.

Format

```
<value specification> ::=
    <literal>
  | <general value specification>

<unsigned value specification> ::=
    <unsigned literal>
  | <general value specification>

04 <general value specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <dynamic parameter specification>
  | <embedded variable specification>
  | <current collation specification>
  | CURRENT_CATALOG
  | CURRENT_DEFAULT_TRANSFORM_GROUP
  | CURRENT_PATH
  | CURRENT_ROLE
  | CURRENT_SCHEMA
  | CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>
  | CURRENT_USER
  | SESSION_USER
  | SYSTEM_USER
  | USER
  | VALUE

04 <simple value specification> ::=
    <literal>
  | <host parameter name>
  | <SQL parameter reference>
  | <embedded variable name>

04 14 <target specification> ::=
    <host parameter specification>
  | <SQL parameter reference>
  | <column reference>
  | <target array element specification>
  | <dynamic parameter specification>
  | <embedded variable specification>

04 <simple target specification> ::=
    <host parameter name>
  | <SQL parameter reference>
  | <column reference>
```

```
  | <embedded variable name>

<host parameter specification> ::=
  <host parameter name> [ <indicator parameter> ]

<dynamic parameter specification> ::=
  <question mark>

<embedded variable specification> ::=
  <embedded variable name> [ <indicator variable> ]

<indicator variable> ::=
  [ INDICATOR ] <embedded variable name>

<indicator parameter> ::=
  [ INDICATOR ] <host parameter name>

<target array element specification> ::=
  <target array reference>
      <left bracket or trigraph> <simple value specification> <right bracket or trigraph>

04 <target array reference> ::=
     <SQL parameter reference>
  | <column reference>

<current collation specification> ::=
  COLLATION FOR <left paren> <string value expression> <right paren>
```

Syntax Rules

1) The declared type of an <indicator parameter> shall be exact numeric with scale 0 (zero).

2) Each <host parameter name> shall be contained in an <SQL-client module definition>.

3) If USER is specified, then CURRENT_USER is implicit.

   NOTE 150 — In an environment where the SQL-implementation conforms to Core SQL, conforming SQL language that contains either:

   — A specified or implied <comparison predicate> that compares the <value specification> USER with a <value specification> other than USER, or

   — A specified or implied assignment in which the "value" (as defined in Subclause 9.2, "Store assignment") contains the <value specification> USER

   will become non-conforming in an environment where the SQL-implementation conforms to some SQL feature that supports character internationalization, unless the character repertoire of the implementation-defined (IA029) character set in that environment is identical to the character repertoire of SQL_IDENTIFIER.

4) The declared type of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, and CURRENT_PATH is character string. Whether the character string is fixed-length or variable-length, and its length if it is fixed-length or maximum length if it is variable-length, are implementation-defined (IV133). The character set of the character string is SQL_IDENTIFIER. The declared type collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

5) The declared type of <string value expression> simply contained in <current collation specification> shall be character string. The declared type of <current collation specification> is character string. Whether the character string is fixed-length or variable-length, and its length if fixed-length or maximum length if variable-length, are implementation-defined (IV133). The character set of the character string is SQL_IDENTIFIER. The collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

6) The <value specification> or <unsigned value specification> VALUE shall be contained in a <domain constraint>. The declared type of an instance of VALUE is the declared type of the domain to which that domain constraint belongs.

7) A <target specification>, <target array reference>, or <simple target specification> that is a <column reference> shall be a new transition variable column reference.

NOTE 151 — "new transition variable column reference" is defined in Subclause 6.6, "<identifier chain>".

8) If <target array element specification> is specified, then:

a) The declared type of the <target array reference> shall be an array type or a distinct type whose source type is an array type.

b) The declared type of a <target array element specification> is the element type of the specified <target array reference>.

c) The declared type of <simple value specification> shall be exact numeric with scale 0 (zero).

9) The declared type of an <indicator variable> shall be exact numeric with a scale of 0 (zero).

10) Each <embedded variable name> shall be contained in an <embedded SQL statement>.

11) Each <dynamic parameter specification> shall be contained in a <preparable statement> that is dynamically prepared in the current SQL-session through the execution of a <prepare statement>.

NOTE 152 — The declared type of a <dynamic parameter specification> is determined by the General Rules for Subclause 20.7, "<prepare statement>".

12) 10 14 The declared type of CURRENT_DEFAULT_TRANSFORM_GROUP and of CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is a character string. Whether the character string is fixed-length or variable-length, and its length if fixed-length or maximum-length if variable length, are implementation-defined (IV133). The character set of the character string is SQL_IDENTIFIER. The declared type collation is the character set collation of SQL_IDENTIFIER, and the collation derivation is *implicit*.

Access Rules

None.

General Rules

1) A <value specification> or <unsigned value specification> specifies a value that is not selected from a table.

2) A <host parameter specification> identifies a host parameter or a host parameter and an indicator parameter in an <SQL-client module definition>.

3) 04 A <target specification> specifies a target that is a host parameter, an output SQL parameter, a column of a new transition variable, an element of a target whose declared type is an array type or a distinct type whose source type is an array type, a parameter used in a dynamically prepared statement, or a host variable, according to whether the <target specification> is a <host parameter specification>, an <SQL parameter reference>, a <column reference>, a <target array element specification>, a <dynamic parameter specification>, or an <embedded variable specification>, respectively.

4) If a <host parameter specification> contains an <indicator parameter> and the value of the indicator parameter is negative, then the value specified by the <host parameter specification> is the null

value; otherwise, the value specified by a <host parameter specification> is the value of the host parameter identified by the <host parameter name>.

5) The value specified by a <literal> is the value represented by that <literal>.

6) The value specified by CURRENT_USER is

Case:

a) If there is a current user identifier, then the value of that current user identifier.

b) Otherwise, the null value.

7) The value specified by SESSION_USER is the value of the SQL-session user identifier.

8) The value specified by CURRENT_ROLE is

Case:

a) If there is a current role name, then the value of that current role name.

b) Otherwise, the null value.

9) The value specified by SYSTEM_USER is equal to an implementation-defined (IV079) string that represents the operating system user who executed the SQL-client module that contains the externally-invoked procedure whose execution caused the SYSTEM_USER <general value specification> to be evaluated.

10) The value specified by CURRENT_CATALOG is the character string that represents the current default catalog name.

11) The value specified by CURRENT_SCHEMA is the character string that represents the current default unqualified schema name.

12) The value specified by CURRENT_PATH is a <schema name list> where <catalog name>s are <delimited identifier>s and the <unqualified schema name>s are <delimited identifier>s. Each <schema name> is separated from the preceding <schema name> by a <comma> with no intervening <separator>s. The schemas referenced in this <schema name list> are those referenced in the SQL-path of the current SQL-session context, in the order in which they appear in that SQL-path.

13) The value specified by <current collation specification> is the name of the collation of the <string value expression>.

14) If a <simple value specification> evaluates to the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.

15) 04 A <simple target specification> specifies a target that is a host parameter, an output SQL parameter, a column of a new transition variable, or a host variable, according to whether the <simple target specification> is a <host parameter name>, an <SQL parameter reference>, a <column reference>, or an <embedded variable name>, respectively.

NOTE 153 — A <simple target specification> can never be assigned the null value.

16) If a <target specification> or <simple target specification> is assigned a value that is the zero-length character string or the zero-length binary string, then it is implementation-defined (IA213) whether an exception condition is raised: *data exception — zero-length character string (2200F)* or *data exception — zero-length binary string (2201Y)*, respectively.

17) A <dynamic parameter specification> identifies a parameter used by a dynamically prepared statement.

18) An <embedded variable specification> identifies a host variable or a host variable and an indicator variable.

19) If an <embedded variable specification> contains an <indicator variable> and the value of the indicator variable is negative, then the value specified by the <embedded variable specification> is the null value; otherwise, the value specified by an <embedded variable specification> is the value of the host variable identified by the <embedded variable name>.

20) The value specified by CURRENT_DEFAULT_TRANSFORM_GROUP is the character string that represents the default transform group name in the SQL-session context.

21) The value specified by CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is the character string that represents the transform group name associated with the data type specified by <path-resolved user-defined type name>.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <general value specification> that contains CURRENT_PATH.

2) Without Feature F251, "Domain support", conforming SQL language shall not contain a <general value specification> that contains VALUE.

3) Without Feature F321, "User authorization", conforming SQL language shall not contain a <general value specification> that contains CURRENT_USER, SYSTEM_USER, or SESSION_USER.

NOTE 154 — Although CURRENT_USER and USER are semantically the same, without Feature F321, "User authorization", CURRENT_USER must be specified as USER.

4) Without Feature T332, "Extended roles", conforming SQL language shall not contain CURRENT_ROLE.

5) Without Feature F762, "CURRENT_CATALOG", conforming SQL language shall not contain a <general value specification> that contains CURRENT_CATALOG.

6) Without Feature F763, "CURRENT_SCHEMA", conforming SQL language shall not contain a <general value specification> that contains CURRENT_SCHEMA.

7) Without Feature F611, "Indicator data types", in conforming SQL language, the declared types of <indicator parameter>s and <indicator variable>s shall be the same implementation-defined (IV080) data type.

8) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic parameter specification>.

9) Without Feature S097, "Array element assignment", conforming SQL language shall not contain a <target array element specification>.

10) Without Feature S241, "Transform functions", conforming SQL language shall not contain CURRENT_DEFAULT_TRANSFORM_GROUP.

11) Without Feature S241, "Transform functions", conforming SQL language shall not contain CURRENT_TRANSFORM_GROUP_FOR_TYPE.

12) Without Feature F693, "SQL-session and client module collations", conforming SQL language shall not contain <current collation specification>.

6.5 <contextually typed value specification>

Function

Specify a value whose data type is to be inferred from its context.

Format

```
<contextually typed value specification> ::=
    <implicitly typed value specification>
  | <default specification>

<implicitly typed value specification> ::=
    <null specification>
  | <empty specification>

<null specification> ::=
  NULL

<empty specification> ::=
    ARRAY <left bracket or trigraph> <right bracket or trigraph>
  | MULTISET <left bracket or trigraph> <right bracket or trigraph>

<default specification> ::=
  DEFAULT
```

Syntax Rules

1) If <empty specification> *ES* is specified, then let *ET* be the element type determined by the context in which *ES* appears. The declared type *DT* of *ES* is

   Case:

   a) If *ES* simply contains ARRAY, then *ET* ARRAY[0].

   b) If *ES* simply contains MULTISET, then *ET* MULTISET.

   ES is effectively replaced by `CAST ( ES AS DT )`.

   NOTE 155 — In every such context, *ES* is uniquely associated with some expression or site of declared type *DT*, which thereby becomes the declared type of *ES*.

2) The declared type *DT* of a <null specification> *NS* is determined by the context in which *NS* appears. *NS* is effectively replaced by CAST (*NS* AS *DT*).

   NOTE 156 — In every such context, *NS* is uniquely associated with some expression or site of declared type *DT*, which thereby becomes the declared type of *NS*.

3) The declared type *DT* of a <default specification> *DS* is the declared type of a <default option> *DO* included in some site descriptor, determined by the context in which *DS* appears. *DS* is effectively replaced by CAST (*DO* AS *DT*).

   NOTE 157 — In every such context, *DS* is uniquely associated with some site of declared type *DT*, which thereby becomes the declared type of *DS*.

Access Rules

None.

General Rules

1) An <empty specification> specifies a collection whose cardinality is zero.

2) A <null specification> specifies the null value.

3) A <default specification> specifies the default value of some associated item.

Conformance Rules

1) Without Feature S090, "Minimal array support", conforming SQL language shall not contain an <empty specification> that simply contains ARRAY.

2) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain an <empty specification> that simply contains MULTISET.

6.6 <identifier chain>

This Subclause is modified by Subclause 6.2, "<identifier chain>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 6.4, "<identifier chain>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 6.2, "<identifier chain>", in ISO/IEC 9075-16.

Function

Disambiguate a period-separated chain of identifiers.

Format

```
<identifier chain> ::=
  <identifier> [ { <period> <identifier> }... ]

<basic identifier chain> ::=
  <identifier chain>
```

Syntax Rules

1) Let *IC* be an <identifier chain>.

2) Let *N* be the number of <identifier>s immediately contained in *IC*.

3) Let I_i, 1 (one) ≤ i ≤ N, be the <identifier>s immediately contained in *IC*, in order from left to right.

4) Let $PIC_1 = I_1$. For each j between 2 and N, let $PIC_j = PIC_{j-1}$ <period> I_j. PIC_j is called the j-th *partial identifier chain* of *IC*.

5) Let *M* be the minimum of *N* and 4.

6) A column *C* is said to be *refinable* if the declared type of *C* is a row type, a JSON type, or a structured type.

7) 04 16 An SQL parameter *P* is said to be *refinable* if the declared type of *P* is a row type, a JSON type, or a structured type.

8) 04 For at most one j between 1 (one) and *M*, PIC_j is called the *basis* of *IC*, and j is called the *basis length* of *IC*. The *referent* of the basis is a column *C* or a period *P* of a table or an SQL parameter *SP*.

9) 16 The basis, basis length, basis scope, and basis referent of *IC* are determined as follows:

   a) If *N* = 1 (one), then

      Case:

      i) If *IC* is contained in an <order by clause> simply contained in a <query expression> *QE*, and the result of *QE* has a column *DC* whose column name is equivalent to *IC*, then PIC_1 is a candidate basis, the scope of PIC_1 is *QE*, and the referent of PIC_1 is the column *DC*.

      ii) Otherwise:

         1) *IC* shall be contained in one of the following:

            A) The scope of one or more range variables whose associated column lists include a column whose <column name> is equivalent to I_1.

B) The scope of one or more range variables whose associated period lists include a period whose period name is equivalent to I_1.

C) 04 15 The scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter whose <SQL parameter name> is equivalent to I_1.

2) Let *IS* be the innermost such scope. Let the phrase *possible scope tags* denote those range variables and <routine name>s whose scope is *IS*. The number of possible scope tags shall be 1 (one). Let *IPST* be that possible scope tag.

NOTE 158 — "range variable" is defined in Subclause 4.17.11, "Range variables". It is possible for two distinct range variables to be equivalent if their scopes are different.

Case:

A) If *IPST* is a range variable *RV*, then let *T* be the table associated with *RV*.

I) For every column *C* in the associated column list of *RV* whose <column name> is equivalent to I_1, PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *RV*, and the referent of PIC_1 is *C*.

NOTE 159 — Two or more columns with equivalent column names are distinguished by their ordinal positions within *T*.

II) For every period *P* in the associated period list of *RV* whose period name is equivalent to I_1, PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *RV*, and the referent of PIC_1 is *P*.

B) 04 15 If *IPST* is a <routine name>, then let *SP* be the SQL parameter whose <SQL parameter name> is equivalent to I_1. PIC_1 is the basis of *IC*, the basis length is 1 (one), the basis scope is the scope of *SP*, and the basis referent is *SP*.

C) If *IPST* is the universal row pattern variable *URPV* of a <row pattern recognition clause> or <window clause>, then *IC* is equivalent to $URPV.PIC_1$.

NOTE 160 — The universal row pattern variable is created by a syntactic transformation defined in Subclause 7.9, "<row pattern common syntax>".

b) If $N > 1$ (one), then the basis, basis length, basis scope, and basis referent are defined in terms of a candidate basis as follows:

i) If *IC* is contained in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_1, then PIC_1 is a candidate basis of *IC*, the scope of PIC_1 is the scope of *SP*, and the referent of PIC_1 is *SP*.

ii) If $N = 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iii) 04 If $N > 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes a refinable SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iv) 16If N >= 2 and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope.

1) 16If N = 2 and *EN* is an exposed <correlation name>, then:

NOTE 161 — The condition "*EN* is an exposed <correlation name>" is always true when applying only the Syntax Rules of this Subclause in this document. It is not always true and thus needed when also taking into account the Syntax Rules of the corresponding Subclauses in other parts of the ISO/IEC 9075 series.

A) For every column *C* in the associated column list of *EN* whose <column name> is equivalent to I_2, PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *C*.

B) For every period *P* in the associated period list of *EN* whose period name is equivalent to I_2, PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *P*.

2) 16If N > 2 and *EN* is an exposed <correlation name>, then for every refinable column *C* in the associated column list of *EN* whose <column name> is equivalent to I_2, PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *EN*, and the referent of PIC_2 is *C*.

NOTE 162 — The condition "*EN* is an exposed <correlation name>" is always true when applying only the Syntax Rules of this Subclause in this document. It is not always true and thus needed when also taking into account the Syntax Rules of the corresponding Subclauses in other parts of the ISO/IEC 9075 series.

v) If N = 2, 3, or 4, and if PIC_{N-1} is equivalent to an exposed <table or query name> that is in scope, then let *EN* be the exposed <table or query name> that is equivalent to PIC_{N-1} and has the innermost scope.

1) For every column *C* in the associated column list of *EN* whose <column name> is equivalent to I_N, PIC_N is a candidate basis of *IC*, the scope of PIC_N is the scope of *EN*, and the referent of PIC_N is *C*.

2) For every period *P* in the associated period list of *EN* whose period name is equivalent to I_N, PIC_N is a candidate basis of *IC*, the scope of PIC_N is the scope of *EN*, and the referent of PIC_N is *P*.

c) There shall be exactly one candidate basis *CB*. The basis of *IC* is *CB*. The basis length is the length of *CB*. The basis scope is the scope of *CB*. The referent of *IC* is the referent of *CB*.

10) Let *BL* be the basis length of *IC*.

11) If $BL < N$, then *IC* is equivalent to:

```
( PIC_BL ) <period>
I_BL+1 <period> ...
<period> I_N
```

04NOTE 163 — In this transformation, (PIC_{BL}) is interpreted as a <value expression primary> of the form <left paren> <value expression> <right paren>. PIC_{BL} is a <value expression> that is a <value expression primary> that is either a <non-parenthesized value expression primary> that is a <column reference> or an <unsigned value specification> that is a <general value specification> that is an <SQL parameter reference>. The identifiers I_{BL+1}, ..., I_N are parsed using the Format and Syntax Rules of Subclause 6.16, "<field reference>", Subclause 6.18, "<method invocation>" and Subclause 6.36, "<JSON simplified accessor>".

12) A <basic identifier chain> shall be an <identifier chain> whose basis is the entire identifier chain.

13) A <basic identifier chain> whose basis referent is a column is a *column reference*. If the basis length is 2, and the basis scope is a <trigger definition> whose <trigger action time> is BEFORE, and I_1 is equivalent to the <new transition variable name> of the <trigger definition>, then the column reference is a *new transition variable column reference*.

14) A column reference such that I_1 is a <row pattern variable name> is a *row pattern column reference*. I_1 is the *qualifying row pattern variable* of the column reference.

15) A <basic identifier chain> whose basis referent is a period is a *period reference*.

16) 04 16 A <basic identifier chain> whose basis referent is an SQL parameter is an *SQL parameter reference*.

17) The data type of a <basic identifier chain> *BIC* that is not a period reference is the data type of the basis referent of *BIC*.

18) If the declared type of a <basic identifier chain> *BIC* is character string, then the collation derivation of the declared type of *BIC* is

    Case:

    a) If the declared type has a declared type collation *DTC*, then *implicit*.

    b) Otherwise, *none*.

Access Rules

None.

General Rules

1) Let *BIC* be a <basic identifier chain>.

2) If *BIC* is a column reference, then *BIC* references the column *C* that is the basis referent of *BIC*.

3) If *BIC* is a period reference, then *BIC* references the period *P* that is the basis referent of *BIC*.

4) 04 15 16 If *BIC* is an SQL parameter reference, then *BIC* references the SQL parameter *SP* of a given invocation of the SQL-invoked routine that contains *SP*.

Conformance Rules

1) 04 Without Feature T325, "Qualified SQL parameter references", conforming SQL language shall not contain an SQL parameter reference whose first <identifier> is the <qualified identifier> of a <routine name>.

6.7 <column reference>

Function

Reference a column.

Format

```
<column reference> ::=
    <basic identifier chain>
  | MODULE <period> <qualified identifier> <period> <column name>
```

Syntax Rules

1) Every <column reference> has a qualifying table and a qualifying scope, as defined in succeeding Syntax Rules.

2) A <column reference> that is a <basic identifier chain> *BIC* shall be a column reference. The qualifying scope is the basis scope of *BIC* and the qualifying table is the table that contains the basis referent of *BIC*.

3) If MODULE is specified, then <qualified identifier> shall be contained in an <SQL-client module definition> *M*, and shall identify a declared local temporary table *DLTT* whose <temporary table declaration> is contained in *M*, and "MODULE <period> <qualified identifier>" shall be an exposed <table or query name> *MPQI*, and <column name> shall identify a column of *DLTT*. The qualifying table is the table identified by *MPQI*, and the qualifying scope is the scope of *MPQI*.

4) If a <column reference> *CR* is contained in a <table expression> *TE* and the qualifying scope of *CR* contains *TE*, then *CR* is an *outer reference* to the qualifying table of *CR*.

5) Let *C* be the column that is referenced by *CR*. The declared type of *CR* is

   Case:

   a) If the column descriptor of *C* includes a data type, then that data type.

   b) Otherwise, the data type identified in the domain descriptor that describes the domain that is identified by the <domain name> that is included in the column descriptor of *C*.

6) A <column reference> contained in a <query specification> is a *queried column reference*.

7) If *QCR* is a queried column reference, then:

   a) The *qualifying query* of *QCR* is

      Case:

      i) If *QCR* is a row pattern column reference, then the <row pattern measures> or <row pattern definition search condition> that contains *QCR*.

      ii) Otherwise, the <query specification> that simply contains the <from clause> that simply contains the <table reference> that defines the qualifying table of *QCR*.

   b) Let *QQ* be the qualifying query of *QCR*.

      Case:

      i) If *QQ* is a <row pattern measures> or <row pattern definition search condition>, then

Case:

1) If *QCR* is contained in a <row pattern navigation operation>, then *QCR* is a *navigated row pattern column reference*.

2) If *QCR* is contained in an aggregated argument of a <set function specification> *SFS*, and *QQ* is the aggregation query of *SFS*, then *QCR* is a *within-group-varying row pattern column reference*.

3) If *QCR* is contained in a non-aggregated argument of a <set function specification> *SFS*, and *QQ* is the aggregation query of *SFS*, then *QCR* is a *group-invariant row pattern column reference*.

4) Otherwise, *QCR* is equivalent to the following <row pattern navigation operation>:

```
LAST ( QCR )
```

ii) If *QQ* is not grouped, or if *QCR* is contained in the <from clause> or the <where clause> simply contained in *QQ*, then *QCR* is an *ordinary column reference*.

iii) If *QCR* is contained in the <having clause>, <window clause>, or <select list> simply contained in *QQ*, and *QCR* is contained in an aggregated argument of a <set function specification> *SFS*, and *QQ* is the aggregation query of *SFS*, then *QCR* is a *within-group-varying column reference*.

iv) Otherwise, *QCR* is a *group-invariant column reference*.

8) If *QCR* is a group-invariant column reference, then *QCR* shall be functionally dependent on the grouping columns of the qualifying query of *QCR*.

9) If *QCR* is a group-invariant row pattern column reference, then *QCR* shall be functionally dependent on the row pattern partitioning columns defined by the <row pattern partition by> of the <row pattern recognition clause> that contains *QCR* or the window partitioning columns defined by the <window partition clause> of the <window clause> that contains *QCR*.

Access Rules

1) Let *CR* be the <column reference>.

2) If the qualifying table of *CR* is a base table or a viewed table, then

Case:

a) If *CR* is contained in a <search condition> immediately contained in an <assertion definition> or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on the column referenced by *CR*.

b) Otherwise,

Case:

i) If *CR* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on the column referenced by *CR*.

ii) Otherwise, the current privileges shall include SELECT on the column referenced by *CR*.

3) If the qualifying table of *CR* is the result of a <data change delta table> *DCDT*, then let *ST* be the subject table of the <data change statement> simply contained in *DCDT* and let *STC* be the column of *ST* that corresponds to *CR*.

Case:

a) If *CR* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on *STC*.

b) Otherwise, the current privileges shall include SELECT on *STC*.

General Rules

1) Let *QCR* be a queried column reference. Let *QT* be the qualifying table of *QCR*, and let *C* be the column of *QT* that is referenced as the basis referent of *QCR*. The value of *QCR* is determined as follows:

a) If *QCR* is an ordinary column reference, then *QCR* denotes the value of *C* in a given row of *QT*.

b) If *QCR* is a navigated row pattern column reference, then *QCR* denotes the value of *C* in a row determined by the General Rules of Subclause 6.27, "<row pattern navigation operation>".

c) If *QCR* is a within-group-varying column reference, then *QCR* denotes the values of *C* in the rows of a given group of the qualifying query of *QCR* used to construct the argument source of a <set function specification>.

d) If *QCR* is a within-group-varying row pattern column reference, then *QCR* denotes the values of *C* in the rows that are mapped to the row pattern variable that qualifies *QCR* in a potential row pattern match of the <row pattern recognition clause> or <window clause> that contains *QCR*.

e) If *QCR* is a group-invariant column reference, then *QCR* denotes a value that is not distinct from the value of *C* in every row of a given group of the qualifying query of *QCR*. If the most specific type of *QCR* is character string, datetime with time zone, or user-defined type, then the precise value is chosen in an implementation-dependent (UV070) fashion.

Conformance Rules

1) Without Feature F821, "Local table references", conforming SQL language shall not contain a <column reference> that simply contains MODULE.

2) Without Feature T301, "Functional dependencies", in conforming SQL language, if *QCR* is a group-invariant column reference, then *QCR* shall be a reference to a grouping column of the qualifying query of *QCR*.

3) Without Feature T301, "Functional dependencies", in conforming SQL language, if *QCR* is a group-invariant row pattern column reference, then *QCR* shall reference a row pattern partitioning column defined by the <row pattern partition by> of the <row pattern recognition clause> of the <row pattern recognition clause> that contains *QCR* or the window partitioning columns defined by the <window partition clause> of the <window clause> that contains *QCR*.

6.8 <SQL parameter reference>

Function

Reference an SQL parameter.

Format

```
<SQL parameter reference> ::=
  <basic identifier chain>
```

Syntax Rules

1) An <SQL parameter reference> shall be a <basic identifier chain> that is an SQL parameter reference.

2) The declared type of an <SQL parameter reference> is the declared type of the SQL parameter that it references.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

6.9 <set function specification>

This Subclause is modified by Subclause 6.3, "<set function specification>", in ISO/IEC 9075-16.

Function

Specify a value derived by the application of a function to an argument.

Format

```
<set function specification> ::=
    [ <running or final> ] <aggregate function>
  | <grouping operation>

<running or final> ::=
    RUNNING | FINAL

<grouping operation> ::=
  GROUPING <left paren> <column reference>
      [ { <comma> <column reference> }... ] <right paren>
```

Syntax Rules

1) If <aggregate function> specifies a <general set function>, then the <value expression> simply contained in the <general set function> shall not contain a <set function specification> or a <query expression>.

2) If <aggregate function> specifies <binary set function>, then neither the <dependent variable expression> nor the <independent variable expression> simply contained in the <binary set function> shall contain a <set function specification> or a <query expression>.

3) 16 A <value expression> *VE* simply contained in a <set function specification> *SFE* is an *aggregated argument* of *SFE* if either *SFE* is a <grouping operation> or *VE* is an aggregated argument of the <aggregate function> simply contained in *SFE*; otherwise, *VE* is a *non-aggregated argument* of *SFE*.

4) 16 A column reference *CR* contained in an aggregated argument of a <set function specification> *SFS* is called an *aggregated column reference* of *SFS*.

5) If <aggregate function> specifies a <filter clause>, then the <search condition> immediately contained in <filter clause> shall not contain a <set function specification>.

6) The *aggregation query* of a <set function specification> *SFS* is determined as follows.

   Case:

   a) 16 If *SFS* has no aggregated column reference, then the aggregation query of *SFS* is the innermost <query specification>, <row pattern measures>, or <row pattern definition search condition> that contains *SFS*.

   b) Otherwise, the innermost qualifying query of the aggregated column references of *SFS* is the aggregation query of *SFS*.

7) 16 *SFS* shall have an aggregation query and shall be contained in the <having clause>, <window clause>, <select list>, <row pattern measures>, or <row pattern definition search condition> of its aggregation query.

8) Let *CR* be an aggregated column reference of *SFS* such that the qualifying query *QQ* of *CR* is not the aggregation query of *SFS*. If *QQ* is grouped and *SFS* is contained in the <having clause>, <window

clause>, or <select list> of *QQ*, then *CR* shall be functionally dependent on the grouping columns of *QQ*.

NOTE 164 — The preceding Syntax Rule follows from a more comprehensive one in Subclause 6.7, "<column reference>".

9) If <aggregate function> is specified, then the declared type of the result is the declared type of the <aggregate function>.

10) If a <grouping operation> is specified, then:

a) Let *T* be the aggregation query of <set function specification> that contains <grouping operation>. Each <column reference> shall reference a grouping column of *T*.

b) The declared type of the result is exact numeric with an implementation-defined (IV081) precision and a scale of 0 (zero).

c) If more than one <column reference> is specified, then let *N* be the number of <column reference>s and let CR_i, 1 (one) ≤ *i* ≤ *N*, be the *i*-th <column reference>.

`GROUPING ( `CR_1`, ..., `CR_{N-1}`, `CR_N` )`

is equivalent to:

`CAST ( ( 2 * GROUPING (` CR_1`, ..., `CR_{N-1} `) + GROUPING (` CR_N `) ) AS` *IDT* `)`

where *IDT* is the implementation-defined (IV081) declared type of the result.

11) Case:

a) If a <set function specification> is contained in a <row pattern measure expression> or a <row pattern definition search condition>, then:

i) <grouping operation> shall not be specified.

ii) If <running or final> is not specified, then RUNNING is implicit.

b) Otherwise, <running or final> shall not be specified.

12) If a <set function specification> is contained in a <row pattern definition search condition>, then FINAL shall not be specified.

13) An aggregated argument of a <set function specification> shall not contain a <row pattern navigation operation>.

Access Rules

None.

General Rules

1) If <aggregate function> is specified, then the result is the value of the <aggregate function>.

NOTE 165 — The value of <grouping operation> is computed by means of syntactic transformations defined in Subclause 7.13, "<group by clause>".

Conformance Rules

1) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <grouping operation>.

2) Without Feature T433, “Multi-argument GROUPING function”, conforming SQL language shall not contain a <grouping operation> that contains more than one <column reference>.

3) 16 Without Feature T301, “Functional dependencies”, in conforming SQL language, if *CR* is an aggregated column reference of *SFS* such that the qualifying query *QQ* of *CR* is not the aggregation query of *SFS*, and *QQ* is grouped and *SFS* is contained in the <having clause>, <window clause>, or <select list> of *QQ*, then *CR* shall be a reference to a grouping column of *QQ*.

6.10 <window function>

Function

Specify a window function.

Format

```
<window function> ::=
  <window function type> OVER <window name or specification>

<window function type> ::=
    <rank function type> <left paren> <right paren>
  | ROW_NUMBER <left paren> <right paren>
  | <aggregate function>
  | <ntile function>
  | <lead or lag function>
  | <first or last value function>
  | <nth value function>
  | <window row pattern measure>

<rank function type> ::=
    RANK
  | DENSE_RANK
  | PERCENT_RANK
  | CUME_DIST

<ntile function> ::=
  NTILE <left paren> <number of tiles> <right paren>

<number of tiles> ::=
    <simple value specification>
  | <dynamic parameter specification>

<lead or lag function> ::=
  <lead or lag> <left paren> <lead or lag extent>
      [ <comma> <offset> [ <comma> <default expression> ] ] <right paren>
      [ <window function null treatment> ]

<lead or lag> ::=
    LEAD | LAG

<lead or lag extent> ::=
  <value expression>

<offset> ::=
  <unsigned integer>

<default expression> ::=
  <value expression>

<window function null treatment> ::=
    RESPECT NULLS | IGNORE NULLS

<first or last value function> ::=
  <first or last value> <left paren> <value expression> <right paren> [ <window function
  null treatment> ]

<first or last value> ::=
    FIRST_VALUE | LAST_VALUE
```

```
<nth value function> ::=
  NTH_VALUE <left paren> <value expression> <comma> <nth row> <right paren>
      [ <from first or last> ] [ <window function null treatment> ]

<nth row> ::=
    <simple value specification>
  | <dynamic parameter specification>

<from first or last> ::=
    FROM FIRST
  | FROM LAST

<window name or specification> ::=
    <window name>
  | <in-line window specification>

<in-line window specification> ::=
  <window specification>

<window row pattern measure> ::=
  <measure name>
```

Syntax Rules

1) Let *OF* be the <window function>.

2) Case:

   a) If *OF* is contained in an <order by clause>, then the <order by clause> shall be simply contained in a <query expression> *QE* that is a simple table query. Let *QSS* be the <query specification> that specifies the sort table of *QE*, as specified in Subclause 7.17, "<query expression>".

      NOTE 166 — The syntactic transformation for <order by clause> specified in Subclause 7.16, "<query specification>", is applied prior to this rule.

   b) Otherwise, *OF* shall be contained in a <select list> that is immediately contained in a <query specification> *QS*. Let *QSS* be the innermost <query specification> contained in *QS* that contains *OF*.

3) Syntax Rules of Subclause 9.23, "Evaluation and transformation of <window function>", are applied with *OF* as *WINFUNC* and *QSS* as *QUERY SPEC IN*; let *QSX* be the *TRANSFORM* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

1) Case:

   a) If <window function type> is <ntile function>, then:

      i) Let *NT* be the value of <number of tiles>.

      ii) Case:

         1) If *NT* is the null value, then the result is the null value.

2) If *NT* is less than or equal to 0 (zero), then an exception condition is raised: *data exception — invalid argument for NTILE function (22014)*.

3) Otherwise:

A) Let *R* be the current row for which the <window function> is being evaluated.

B) Let *ROWNUM* be the result of evaluating ROW_NUMBER() OVER *WNS* for *R*.

C) Let *WDX1* be the window structure descriptor that describes the window defined by the <window specification>

```
( WNS RANGE BETWEEN UNBOUNDED PRECEDING
                    AND UNBOUNDED FOLLOWING )
```

D) Let *T* be the collection of rows in the window frame of *R* as defined by *WDX1*, as specified by the General Rules of Subclause 7.15, "<window clause>".

E) Let *CT* be the cardinality of *T*.

F) Case:

I) If MOD(*CT*, *NT*) = 0 (zero), then for each *i*, 1 (one) ≤ *i* ≤ *NT*, let NQ_i be (*CT* / *NT*).

II) Otherwise, for each *i*, 1 (one) ≤ *i* ≤ MOD(*CT*, *NT*), let NQ_i be CEILING(CAST(*CT* AS REAL) / *NT*), and for each *i*, MOD(*CT*, *NT*) < *i* ≤ *NT*, let NQ_i be FLOOR(CAST(*CT* AS REAL) / *NT*).

G) Let END_0 be 0 (zero). For each *i*, 1 (one) ≤ *i* ≤ *NT*, let $START_i$ be ($END_{(i-1)}$ + 1) and let END_i be ($END_{(i-1)}$ + NQ_i).

H) The value of <window function> is *i*, where $START_i \le ROWNUM \le END_i$.

b) If <window function type> is <lead or lag function>, then:

i) Let *OFFSET* be the value of *OFF* and let *DEFAULT* be the value of *VE2*.

ii) Let *WDX1* be the window structure descriptor that describes the window defined by the <window specification>

```
( WNS RANGE BETWEEN UNBOUNDED PRECEDING
                    AND UNBOUNDED FOLLOWING )
```

iii) Let *T* be the collection of rows in the window frame of *R* as defined by *WDX1*, as specified by the General Rules of Subclause 7.15, "<window clause>".

iv) If LEAD is specified, then:

1) Case:

A) If *NTREAT* is RESPECT NULLS, then let *TX* be the sequence of values that is the result of applying *VE1* to each row of *T* that follows the current row, ordered according to the window ordering of *WDX1*.

B) Otherwise, let *TX* be the sequence of values that is the result of applying *VE1* to each row of *T* that follows the current row and eliminating null values, ordered according to the window ordering of *WDX1*.

2) Let *n* be the number of values in *TX*.

3) Case:

A) If *OFFSET* > *n*, then the value of <window function> is *DEFAULT*.

B) If *OFFSET* = 0 (zero), then the value of <window function> is the value of *VE1* evaluated for the current row.

C) Otherwise, the value of <window function> is the *m*-th value of *TX*, where *m* = *OFFSET*.

v) If LAG is specified, then:

1) Case:

A) If *NTREAT* is RESPECT NULLS, then let *TX* be the sequence of values that is the result of applying *VE1* to each row of *T* that precedes the current row, ordered according to the row ordering of *WDX1*.

B) Otherwise, let *TX* be the sequence of values that is the result of applying *VE1* to each row of *T* that precedes the current row and eliminating null values, ordered according to the row ordering of *WDX1*.

2) Let *n* be the number of values in *TX*.

3) Case:

A) If *OFFSET* > *n*, then the value of <window function> is *DEFAULT*.

B) If *OFFSET* = 0 (zero), then the value of <window function> is the value of *VE1* evaluated for the current row.

C) Otherwise, the value of <window function> is the *m*-th value of *TX*, where *m* = (*n* − *OFFSET* + 1).

c) If <window function type> is <first or last value function>, then:

i) Let *T* be the collection of rows in the window frame of the current row defined by *WDX*, as specified by the General Rules of Subclause 7.15, "<window clause>".

ii) Case:

1) If RESPECT NULLS is specified or implicit, then let *TX* be the sequence of values that is the result of applying the <value expression> to each row of *T*, ordered according to the row ordering of *WDX*.

2) Otherwise, let *TX* be the sequence of values that is the result of applying the <value expression> to each row of *T* and eliminating null values, ordered according to the row ordering of *WDX*.

iii) Case:

1) If *TX* is empty, then the value of <window function> is the null value.

2) If FIRST_VALUE is specified, then the value of <window function> is the first value of *TX*.

3) Otherwise, the value of <window function> is the last value of *TX*.

d) If <window function type> is <nth value function>, then:

i) Let *RN* be the value of <nth row>.

ii) Case:

1) If *RN* is the null value, then the result is the null value.

2) If *RN* is less than or equal to 0 (zero), then an exception condition is raised: *data exception — invalid argument for NTH_VALUE function (22016)*.

3) Otherwise:

A) Let *T* be the collection of rows in the window frame of the current row defined by *WDX*, as specified by the General Rules of Subclause 7.15, "<window clause>".

B) Case:

I) If RESPECT NULLS is specified or implicit, then let *TX* be the sequence of values that is the result of applying the <value expression> to each row of *T*, ordered according to the row ordering of *WDX*.

II) Otherwise, let *TX* be the sequence of values that is the result of applying the <value expression> to each row of *T* and eliminating null values, ordered according to the row ordering of *WDX*.

C) Let *TXN* be the number of values in *TX*.

I) If *TXN* = 0 (zero) or if *TXN* < *RN*, then the value of <window function> is the null value.

II) Case:

1) If FROM LAST is specified, then the value of <window function> is the *m*-th value of *TX*, where *m* = (*TXN* − *RN* + 1).

2) Otherwise, the value of <window function> is the *m*-th value of *TX*, where *m* = *RN*.

e) If <window function type> is <window row pattern measure>, then let *R* be the current row for which <window function> is being evaluated. The value of <window function> is

Case:

i) If the skip indicator of *R* is *True*, then the null value.

NOTE 167 — The skip indicator is defined in Subclause 7.15, "<window clause>".

ii) Otherwise, the value of the <row pattern measure expression> simply contained in *RPMC*, evaluated in the designated row pattern match associated with *R*.

NOTE 168 — The designated row pattern match of *R* is defined in the General Rules of Subclause 7.15, "<window clause>".

f) Otherwise, the value of <window function> is the value of the <aggregate function>.

Conformance Rules

1) Without Feature R020, "Row pattern recognition: WINDOW clause", conforming SQL language shall not contain a <window row pattern measure>.

2) Without Feature T611, "Elementary OLAP operations", conforming SQL language shall not contain a <window function>.

3) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window name>.

4) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain PERCENT_RANK or CUME_DIST.

5) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window function> that simply contains ROW_NUMBER and immediately contains a <window name or specification> whose window structure descriptor does not contain a window ordering clause.

6) Without Feature T614, "NTILE function", conforming SQL language shall not contain <ntile function>.

7) Without Feature T615, "LEAD and LAG functions", conforming SQL language shall not contain <lead or lag function>.

8) Without Feature T616, "Null treatment option for LEAD and LAG functions", in conforming SQL language, <lead or lag function> shall not contain <window function null treatment>.

9) Without Feature T617, "FIRST_VALUE and LAST_VALUE functions", conforming SQL language shall not contain <first or last value function>.

10) Without Feature T618, "NTH_VALUE function", conforming SQL language shall not contain <nth value function>.

11) Without Feature T627, "Window framed COUNT DISTINCT", conforming SQL language shall not contain a <window function> that contains a window ordering clause or a window framing clause, and that contains a <window function type> of <aggregate function> that is a <general set function> simply containing both COUNT and DISTINCT.

6.11 <nested window function>

Function

Specify a function nested in an aggregated argument of an <aggregate function> simply contained in a <window function>.

Format

```
<nested window function> ::=
    <nested row number function>
  | <value_of expression at row>

<nested row number function> ::=
  ROW_NUMBER <left paren> <row marker> <right paren>

<value_of expression at row> ::=
  VALUE_OF <left paren> <value expression> AT <row marker expression>
      [ <comma> <value_of default value> ] <right paren>

<row marker> ::=
    BEGIN_PARTITION
  | BEGIN_FRAME
  | CURRENT_ROW
  | FRAME_ROW
  | END_FRAME
  | END_PARTITION

<row marker expression> ::=
  <row marker> [ <row marker delta> ]

<row marker delta> ::=
    <plus sign> <row marker offset>
  | <minus sign> <row marker offset>

<row marker offset> ::=
    <simple value specification>
  | <dynamic parameter specification>

<value_of default value> ::=
  <value expression>
```


Syntax Rules

1) A <nested window function> *NWF* shall be contained in an aggregated argument of an <aggregate function> *AF* immediately contained in a <window function> *WF*.

2) The declared type of <nested row number function> is an implementation-defined (IV085) exact numeric type with scale 0 (zero).

3) If <value_of expression at row> is specified, then let *DTS* be the set consisting of the declared type of the immediately contained <value expression> and, if <value_of default value> is specified, the declared type of <value_of default value>. The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *DTS* as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules. The declared type of <value_of expression at row> is *DT*.

4) The declared type of <row marker offset> shall be exact numeric with scale 0 (zero).

Access Rules

None.

General Rules

1) Let *W* be the window structure descriptor associated with *WF*. Let *R1* be a row for which the value of *NWF* is to be computed, let *P* be the window partition of *R1* as determined by *W*, let *F* be the window frame of *R1* as determined by *W*, and let *R2* be a row in the argument source of *AF* during the computation of *NWF* for row *R1*.

NOTE 169 — Given the conditions of this rule, the argument source of *AF* is the window frame *F* determined by *R1*; therefore, *R2* is in *F*.

2) Let *NP* be the number of rows in *P*. Let { ROW_1, ROW_2, ..., ROW_{NP} } be the enumeration of rows of *P* according to the ordering of rows determined by *W*. For each *n* between 1 (one) and *NP*, *n* is called the *window partition row number* of the row ROW_n.

3) If *NWF* is <nested row number function>, then:

   a) Let *RM* be the <row marker>.

   b) Case:

      i) If *RM* is BEGIN_PARTITION, then the value of *NWF* is 1 (one).

      ii) If *RM* is END_PARTITION, then the value of *NWF* is *NP*.

      iii) If *RM* is CURRENT_ROW, then the value of *NWF* is the window partition row number of *R1*.

      iv) If *RM* is BEGIN_FRAME, then the value of *NWF* is the window partition row number of the first row of *F*.

      v) If *RM* is END_FRAME, then the value of *NWF* is the window partition row number of the last row of *F*.

      vi) If *RM* is FRAME_ROW, then the value of *NWF* is the window partition row number of *R2*.

NOTE 170 — It is possible for the window frame *F* to be empty; however, in that case, there are no rows on which to evaluate *NWF* and the preceding General Rule will not be invoked. Therefore, the General Rules can assume that *F* is not empty when *RM* is BEGIN_FRAME, END_FRAME, or FRAME_ROW.

4) If *NWF* is <value_of expression at row>, then:

   a) Let *VE* be the <value expression> immediately contained in *NWF*, and let *RME* be the <row marker expression> immediately contained in *NWF*.

   b) Let *RM* be the <row marker> contained in *RME*. Let *RN* be the value of the <nested row number function> ROW_NUMBER(*RM*).

   c) Case:

      i) If *NWF* contains <value_of default value>, then let *DEF* be the value of <value_of default value>.

      ii) Otherwise, let *DEF* be the null value.

   d) Case:

i) If *RME* contains <row marker delta>, then let *OFF* be the value of the <row marker offset>. If *RME* contains <plus sign>, then let *M* be *RN*+*OFF*; otherwise, let *M* be *RN*−*OFF*.

ii) Otherwise, let *M* be *RN*.

e) Case:

i) If *M* is the null value, less than 1 (one), or greater than *NP*, then the value of *NWF* is *DEF*.

NOTE 171 — *M* can only be null if *OFF* is null. As explained in a previous informative note, *NWF* is never evaluated if *F* is empty, so the preceding rule does not need to cover the case of *F* being empty.

ii) Otherwise, the value of *NWF* is the value of *VE*, evaluated in ROW_M.

Conformance Rules

1) Without Feature T619, "Nested window functions", conforming SQL language shall not contain <nested window function>.

6.12 <case expression>

This Subclause is modified by Subclause 6.5, "<case expression>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 6.6, "<case expression>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 6.4, "<case expression>", in ISO/IEC 9075-16.

Function

Specify a conditional value.

Format

```
<case expression> ::=
    <case abbreviation>
  | <case specification>

<case abbreviation> ::=
    NULLIF <left paren> <value expression> <comma> <value expression> <right paren>
  | COALESCE <left paren> <value expression>
        { <comma> <value expression> }... <right paren>

<case specification> ::=
    <simple case>
  | <searched case>

<simple case> ::=
  CASE <case operand> <simple when clause>... [ <else clause> ] END

<searched case> ::=
  CASE <searched when clause>... [ <else clause> ] END

<simple when clause> ::=
  WHEN <when operand list> THEN <result>

15 <searched when clause> ::=
  WHEN <search condition> THEN <result>

<else clause> ::=
  ELSE <result>

16 <case operand> ::=
    <row value predicand>
  | <overlaps predicate part 1>

<when operand list> ::=
  <when operand> [ { <comma> <when operand> }... ]

14 16 <when operand> ::=
    <row value predicand>
  | <comparison predicate part 2>
  | <between predicate part 2>
  | <in predicate part 2>
  | <character like predicate part 2>
  | <octet like predicate part 2>
  | <similar predicate part 2>
  | <regex like predicate part 2>
  | <null predicate part 2>
  | <quantified comparison predicate part 2>
  | <normalized predicate part 2>
  | <match predicate part 2>
```

```
  | <overlaps predicate part 2>
  | <distinct predicate part 2>
  | <member predicate part 2>
  | <submultiset predicate part 2>
  | <set predicate part 2>
  | <type predicate part 2>

<result> ::=
    <result expression>
  | NULL

<result expression> ::=
  <value expression>
```

Syntax Rules

1) If a <case expression> specifies a <case abbreviation>, then:

   a) A <value expression> generally contained in the <case abbreviation> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

   b) A <value expression> generally contained in the <case abbreviation> shall not generally contain a <table primary> that contains a <data change delta table>.

   c) NULLIF (V_1, V_2) is equivalent to the following <case specification>:

```
CASE WHEN
V1=V2 THEN
NULL ELSE V1
END
```

      The Conformance Rules of Subclause 8.2, "<comparison predicate>", are applied to the result of this syntactic transformation.

   d) COALESCE (V_1, V_2) is equivalent to the following <case specification>:

```
CASE
  WHEN NOT V1 IS NULL THEN V1
  ELSE V2
END
```

   e) COALESCE (V_1, V_2, ..., V_n), for $n \geq 3$, is equivalent to the following <case specification>:

```
CASE
  WHEN NOT V1 IS NULL THEN V1
  ELSE COALESCE (V2, ..., Vn)
END
```

2) [15] If a <case specification> specifies a <simple case>, then let *CO* be the <case operand>.

   a) *CO* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

   b) *CO* shall not generally contain a <table primary> that contains a <data change delta table>.

   c) [16] If *CO* is <overlaps predicate part 1>, then each <when operand> shall be <overlaps predicate part 2>. If *CO* is <row value predicand>, then no <when operand> shall be an <overlaps predicate part 2>.

   d) Let *N* be the number of <simple when clause>s.

e) For each i between 1 (one) and N, let WOL_i be the <when operand list> of the i-th <simple when clause>. Let $M(i)$ be the number of <when operand>s simply contained in WOL_i. For each j between 1 and $M(i)$, let $WO_{i,j}$ be the j-th <when operand> simply contained in WOL_i.

f) For each i between 1 (one) and N, and for each j between 1 (one) and $M(i)$,

Case:

i) If $WO_{i,j}$ is a <row value predicand>, then let $EWO_{i,j}$ be

```
= WO_{i,j}
```

ii) Otherwise, let $EWO_{i,j}$ be $WO_{i,j}$.

g) Let R_i be the <result> of the i-th <simple when clause>.

h) If <else clause> is specified, then let *CEEC* be that <else clause>; otherwise, let *CEEC* be the zero-length character string.

i) The <simple case> is equivalent to a <searched case> in which the i-th <searched when clause> takes the form:

```
WHEN ( CO EWO_{i,1} ) OR
... OR
( CO EWO_{i,M(i)} )
THEN R_i
```

j) The <else clause> of the equivalent <searched case> takes the form:

```
CEEC
```

k) The Conformance Rules of the Subclauses of Clause 8, "Predicates", are applied to the result of this syntactic transformation.

NOTE 172 — The specific Subclauses of Clause 8, "Predicates", are determined by the predicates that are created as a result of the syntactic transformation.

3) At least one <result> in a <case specification> shall specify a <result expression>.

4) If an <else clause> is not specified, then ELSE NULL is implicit.

5) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the set of declared types of all <result expression>s in the <case specification> as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules. The declared type of the <case specification> is *RT*.

Access Rules

None.

General Rules

1) Case:

a) If a <result> specifies NULL, then its value is the null value.

b) If a <result> specifies a <value expression>, then its value is the value of that <value expression>.

2) Case:

   a) 15 If the value of the <search condition> of some <searched when clause> in a <case specification> is *True*, then the value of the <case expression> is the value of the <result> of the first (leftmost) <searched when clause> whose <search condition> evaluates to *True*, cast as the declared type of the <case specification>.

   b) If no <search condition> in a <case specification> evaluates to *True*, then the value of the <case expression> is the value of the <result> of the explicit or implicit <else clause>, cast as the declared type of the <case specification>.

Conformance Rules

1) Without Feature F262, "Extended CASE expression", in conforming SQL language, a <case operand> immediately contained in a <simple case> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

2) Without Feature F262, "Extended CASE expression", in conforming SQL language, a <when operand> contained in a <simple when clause> shall be a <row value predicand> that is a <row value constructor predicand> that is a single <common value expression> or <boolean predicand>.

3) 15 Without Feature F263, "Comma-separated predicates in simple CASE expression", in conforming SQL language, a <when operand list> contained in a <simple when clause> shall simply contain exactly one <when operand>.

6.13 <cast specification>

This Subclause is modified by Subclause 6.2, "<cast specification>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.6, "<cast specification>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 6.7, "<cast specification>", in ISO/IEC 9075-15.

Function

Specify a data conversion.

Format

```
14 <cast specification> ::=
  CAST <left paren>
       <cast operand> AS <cast target>
       [ FORMAT <cast template> ]
       <right paren>

<cast operand> ::=
    <value expression>
  | <implicitly typed value specification>

15 <cast target> ::=
    <domain name>
  | <data type>

<cast template> ::=
  <character string literal>
```

Syntax Rules

1) Case:

   a) 15 If a <domain name> is specified, then let *TD* be the data type of the specified domain.

   b) If a <data type> is specified, then let *TD* be the data type identified by <data type>. <data type> shall not contain a <collate clause>.

2) 09 The declared type of the result of the <cast specification> is *TD*.

3) 14 If the <cast operand> is a <value expression>, then let *SD* be the declared type of the <value expression>.

4) If <cast template> *CT* is specified, then exactly one of the following shall be true:

   a) *SD* is a datetime data type and *TD* is a character string data type. In this case:

      i) The character repertoire of *TD* and of *CT* shall be the same.

      ii) The Syntax Rules of Subclause 9.50, "Converting a datetime to a formatted character string", are applied with *SD* as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *TD* as *TARGET TYPE*.

   b) *SD* is a character string type and *TD* is a datetime data type. In this case:

      i) The character repertoire of *SD* and of *CT* shall be the same.

ii) The Syntax Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with *TD* as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *SD* as *CHARACTER STRING TYPE*.

5) Let *C* be some column and let *CO* be the <cast operand> of a <cast specification> *CS*. *C* is a *leaf column* of *CS* if *CO* consists of a single column reference that identifies *C* or of a single <cast specification> *CS1* of which *C* is a leaf column.

6) If the <cast operand>specifies an <empty specification>, then *TD* shall be a collection type or a distinct type whose source type is a collection type.

7) 09 14 If the <cast operand> is a <value expression>, then the valid combinations of *TD* and *SD* in a <cast specification> are given by the following table. "Y" indicates that the combination is syntactically valid without restriction; "M" indicates that the combination is valid subject to other Syntax Rules in this Subclause being satisfied; and "N" indicates that the combination is not valid. If the contents of a cell are unspecified, that also indicates that the combination is not valid.

SD \ *TD*	EN	AN	DF	C	D	T	TS	YM	DT	BO	UDT	B	RT	CT	RW	JS
EN	Y	Y	Y	Y	N	N	N	M	M	N	M	N	M	N	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	M	N	M	N	N	N
DF	Y	Y	Y	Y	N	N	N	N	N	N	M	N	M	N	N	N
C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	M	N	M	N	N	N
D	N	N	N	Y	Y	N	Y	N	N	N	M	N	M	N	N	N
T	N	N	N	Y	N	Y	Y	N	N	N	M	N	M	N	N	N
TS	N	N	N	Y	Y	Y	Y	N	N	N	M	N	M	N	N	N
YM	M	N	N	Y	N	N	N	Y	N	N	M	N	M	N	N	N
DT	M	N	N	Y	N	N	N	N	Y	N	M	N	M	N	N	N
BO	N	N	N	Y	N	N	N	N	N	Y	M	N	M	N	N	N
UDT	M	M	M	M	M	M	M	M	M	M	M	M	M	M	N	N
B	N	N	N	N	N	N	N	N	N	N	M	Y	M	N	N	N
RT	M	M	M	M	M	M	M	M	M	M	M	M	M	N	N	N
CT	N	N	N	N	N	N	N	N	N	N	M	N	N	M	N	N
RW	N	N	N	N	N	N	N	N	N	N	N	N	N	N	M	N
JS	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y

Where:

EN = Exact Numeric
AN = Approximate Numeric
DF - Decimal Floating-Point
C = Character (Fixed- or Variable-Length, or Character Large Object)
D = Date
T = Time
TS = Timestamp
YM = Year-Month Interval
DT = Day-Time Interval
BO = Boolean
UDT = User-Defined Type
B = Binary (Fixed- or Variable-Length or Binary Large Object)
RT = Reference type
CT = Collection type
RW = Row type
JS = JSON type

8) If *TD* is an interval and *SD* is exact numeric, then *TD* shall contain only a single <primary datetime field>.

9) If *TD* is exact numeric and *SD* is an interval, then *SD* shall contain only a single <primary datetime field>.

10) If *SD* is character string and *TD* is fixed-length, variable-length, or large object character string, then the character repertoires of *SD* and *TD* shall be the same.

11) If *TD* is a fixed-length, variable-length, or large object character string, then the declared type collation of the <cast specification> is the character set collation of the character set of *TD* and its collation derivation is *implicit*.

12) If the <cast operand> is a <value expression> and either *SD* or *TD* is a user-defined type, then either *TD* shall be a supertype of *SD* or there shall be a data type *P* such that all of the following are true:

    a) The type designator of *P* is in the type precedence list of *SD*.

    b) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.

    c) The type designator of no other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes the type designator of *P* in the type precedence list of *SD*.

13) If the <cast operand> is a <value expression> and either *SD* or *TD* is a reference type, then:

    a) Let *RTSD* and *RTTD* be the referenced types of *SD* and *TD*, respectively.

    b) If <data type> is specified and contains a <scope clause>, then let *STD* be that scope. Otherwise, let *STD*, possibly empty, be the scope included in the reference type descriptor of *SD*.

    c) Either *RSTD* and *RTTD* shall be compatible, or there shall be a data type *P* in the type precedence list of *SD* such that all of the following are true:

        i) There is a user-defined cast CF_P whose user-defined cast descriptor includes *P* as the source data type and *TD* as the target data type.

        ii) The type designator of no other data type *Q* that is included as the source data type in the user-defined cast descriptor of some user-defined cast CF_Q that has *TD* as the target data type precedes the type designator of *P* in the type precedence list of *SD*.

14) If *SD* is a collection type, then:

    a) Let *ESD* be the element type of *SD*.

    b) Let *ETD* be the element type of *TD*.

    ```
    CAST ( VALUE AS ETD )
    ```

    where *VALUE* is a <value expression> of declared type *ESD*, shall be a valid <cast specification>.

15) If *SD* is a row type, then:

    a) Let *DSD* be the degree of *SD*.

    b) Let *DTD* be the degree of *TD*.

    c) *DSD* shall be equal to *DTD*.

    d) Let FSD_i and FTD_i, 1 (one) ≤ *i* ≤ *DSD*, be the *i*-th field of *SD* and *TD*, respectively.

    e) Let $TFSD_i$ and $TFTD_i$, 1 (one) ≤ *i* ≤ *DSD*, be the declared type of FSD_i and the declared type of FTD_i, respectively.

    f) For *i* varying from 1 (one) to *DSD*, the <cast specification>:

    ```
    CAST ( VALUE_i AS TFTD_i )
    ```

    where $VALUE_i$ is an arbitrary <value expression> of declared type $TFSD_i$, shall be a valid <cast specification>.

16) [14]If <domain name> is specified, then let *D* be the domain identified by the <domain name>. The schema identified by the explicit or implicit <schema name> of the <domain name> shall include the descriptor of *D*.

Access Rules

1) If <domain name> is specified, then

   Case:

   a) If <cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on the domain identified by <domain name>.

   b) Otherwise, the current privileges shall include USAGE on the domain identified by <domain name>.

2) If the <cast operand> is a <value expression> and either *SD* or *TD* is a user-defined type or a reference type, then

   Case:

   a) If <cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include EXECUTE on CF_P.

   b) Otherwise, the current privileges shall include EXECUTE on CF_P.

General Rules

1) Let *CS* be the <cast specification>. If the <cast operand> is a <value expression> *VE*, then let *SV* be the value of *VE*.

2) Case:

   a) If the <cast operand> specifies NULL, then the result of *CS* is the null value and no further General Rules of this Subclause are applied.

   b) If the <cast operand>specifies an <empty specification>, then the result of *CS* is an empty collection of declared type *TD* and no further General Rules of this Subclause are applied.

   c) If *SV* is the null value, then the result of *CS* is the null value and no further General Rules of this Subclause are applied.

3) [14]If either *SD* or *TD* is a user-defined type, then

   Case:

   a) If *TD* is a supertype of *SD*, then *TV* is *SV*.

   b) Otherwise:

      i) Let *CP* be the cast function contained in the user-defined cast descriptor of CF_P.

      ii) Let *SAL* be a static SQL argument list that has a single SQL-argument that is <value expression>. The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *CP* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *TR* be the *VALUE* returned from the application of those General Rules

iii) Case:

1) If *TD* is a user-defined type, then *TV* is *TR*.

2) Otherwise, *TV* is the result of

```
CAST (TR AS TD)
```

4) If either *SD* or *TD* is a reference type, then

Case:

a) If *RSTD* and *RTTD* are compatible, then:

i) *TV* is *SV*.

ii) The scope in the reference type descriptor of *TV* is *STD*.

b) Otherwise:

i) Let *CP* be the cast function contained in the user-defined cast descriptor of CF_P.

ii) Let *SAL* be a static SQL argument list that has a single SQL-argument that is <value expression>. The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *CP* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *TV* be the *VALUE* returned from the application of those General Rules.

iii) The scope in the reference type descriptor of *TV* is *STD*.

5) If *SD* is an array type and *TD* is either an array type or a multiset type, then:

a) Let *SC* be the cardinality of *SV*.

b) For *i* varying from 1 (one) to *SC*, the following <cast specification> is applied:

```
CAST ( VE[i] AS ETD )
```

yielding value TVE_i.

c) If *TD* is an array type, then let *TC* be the maximum cardinality of *TD*.

Case:

i) If *SC* is greater than *TC*, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

ii) Otherwise, *TV* is the array with elements TVE_i, 1 (one) ≤ *i* ≤ *SC*.

d) If *TD* is a multiset type, then *TV* is the multiset with elements TVE_i, 1 (one) ≤ *i* ≤ *SC*.

6) If *SD* is a multiset type and *TD* is either an array type or a multiset type, then:

a) Let *SC* be the cardinality of *SV*.

b) If *TD* is an array type, then let *TC* be the maximum cardinality of *TD*.

Case:

i) If *SC* is greater than *TC*, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

ii) Otherwise, *TV* is the array resulting from the evaluation of

```
ARRAY ( ( SELECT CAST ( M.E AS ETD ) FROM UNNEST (VE) AS M(E) ) )
```

NOTE 173 — Since this cast from a multiset to an array type uses a SELECT FROM without an ORDER BY, the order of the elements in the result is implementation-dependent.

c) If *TD* is a multiset type, then *TV* is the multiset resulting from the evaluation of

```
MULTISET ( ( SELECT CAST ( M.E AS ETD ) FROM UNNEST (VE) AS M(E) ) )
```

7) If *SD* is a row type, then *TV* is the row resulting from the evaluation of

```
ROW ( CAST ( VE.FSD1 AS TFTD1 ),
      CAST ( VE.FSD2 AS TFTD2 ),
    ...
      CAST ( VE.FSDDSD AS TFTDDSD ) )
```

8) If *TD* is exact numeric, then

Case:

a) If *SD* is numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined (IA002).

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to

```
CAST ( LT AS TD )
```

c) If *SD* is an interval data type, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits, then *TV* is that representation.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

9) If *TD* is approximate numeric, then

Case:

a) If *SD* is numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding or truncating if necessary, then *TV* is that representation. The choice of whether to round or truncate is implementation-defined (IA002).

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to

```
CAST ( LT AS TD )
```

10) If *TD* is the decimal floating-point type, then

Case:

a) If *SD* is numeric, then

Case:

i) If there is a representation of *SV* in the data type *TD* that does not lose any leading significant digits after rounding if necessary, then *TV* is that representation.

ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

b) If *SD* is character string, then *SV* is replaced by *SV* with any leading or trailing <truncating whitespace> removed.

Case:

i) If *SV* does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 5.3, "<literal>", then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

ii) Otherwise, let *LT* be that <signed numeric literal>. The <cast specification> is equivalent to:

```
CAST ( LT AS TD )
```

11) If *TD* is fixed-length character string, then let *LTD* be the length in characters of *TD*.

Case:

a) If *SD* is exact numeric, then:

i) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 5.3, "<literal>", whose scale is the same as the scale of *SD*, whose interpreted value is the absolute value of *SV*, and that is not an <unsigned hexadecimal integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of '-' || *YP*.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD*−*LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

b) If *SD* is approximate numeric, then:

i) Let *YP* be a character string as follows.

Case:

1) If *SV* equals 0 (zero), then *YP* is '0E0'.

2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 5.3, "<literal>", whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0' (zero), followed by a <period> and an <unsigned integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of '-' || *YP*.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD*−*LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

c) If *SD* is the decimal floating-point type, then let *Y* be an implementation-defined (IV110) <signed numeric literal> whose interpreted value is equal to *SV*.

Case:

i) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

ii) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

iii) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD*−*LY* <space>s.

iv) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

d) If *SD* is fixed-length character string, variable-length character string, or large object character string, then

Case:

i) If the length in characters of *SV* is equal to *LTD*, then *TV* is *SV*.

ii) If the length in characters of *SV* is larger than *LTD*, then *TV* is the first *LTD* characters of *SV*. If any of the remaining characters of *SV* are non-<truncating whitespace> characters, then a completion condition is raised: *warning — string data, right truncation (01004)*.

iii) If the length in characters *M* of *SV* is smaller than *LTD*, then *TV* is *SV* extended on the right by *LTD*−*M* <space>s.

e) If *SD* is a datetime data type or an interval data type, then

i) Case:

1) If <cast template> *CT* is specified, then the General Rules of Subclause 9.50, "Converting a datetime to a formatted character string", are applied with *SV* as *DATETIME VALUE*, the value of *CT* as *TEMPLATE*, and *TD* as *TARGET TYPE*; let *Y* be the *FORMATTED CHARACTER STRING* returned from the application of those General Rules.

2) Otherwise, let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 5.3, "<literal>", such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*. If *SV* is a negative interval, then <sign> shall be specified within <unquoted interval string> in the literal *Y*.

ii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.

3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD*−*LY* <space>s.

4) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

f) If *SD* is Boolean, then

Case:

i) If *SV* is *True* and *LTD* is not less than 4, then *TV* is 'TRUE' extended on the right by *LTD*−4 <space>s.

ii) If *SV* is *False* and *LTD* is not less than 5, then *TV* is 'FALSE' extended on the right by *LTD*−5 <space>s.

iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast (22018)*.

12) If *TD* is variable-length character string or large object character string, then let *MLTD* be the maximum length in characters of *TD*.

Case:

a) If *SD* is exact numeric, then:

i) Let *YP* be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 5.3, "<literal>", whose scale is the same as the scale of *SD*, whose interpreted value is the absolute value of *SV*, and that is not an <unsigned hexadecimal integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

b) If *SD* is approximate numeric, then:

i) Let *YP* be a character string as follows.

Case:

1) If *SV* equals 0 (zero), then *YP* is '0E0'.

2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 5.3, "<literal>", whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0', followed by a <period> and an <unsigned integer>.

ii) Case:

1) If *SV* is less than 0 (zero), then let *Y* be the result of `'-' || YP`.

2) Otherwise, let *Y* be *YP*.

iii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

c) If *SD* is the decimal floating-point type, then let *Y* be an implementation-defined (IV110) <signed numeric literal> whose interpreted value is equal to *SV*.

Case:

i) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

ii) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

iii) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

d) If *SD* is fixed-length character string, variable-length character string, or large object character string, then

Case:

i) If the length in characters of *SV* is less than or equal to *MLTD*, then *TV* is *SV*.

ii) If the length in characters of *SV* is larger than *MLTD*, then *TV* is the first *MLTD* characters of *SV*. If any of the remaining characters of *SV* are non-<truncating whitespace> characters, then a completion condition is raised: *warning — string data, right truncation (01004)*.

e) If *SD* is a datetime data type or an interval data type, then:

i) Case:

1) If <cast template> *CT* is specified, then the General Rules of Subclause 9.50, "Converting a datetime to a formatted character string", are applied with *SV* as *DATETIME VALUE*, the value of *CT* as *TEMPLATE*, and *TD* as *TARGET TYPE*; let *Y* be the *FORMATTED CHARACTER STRING* returned from the application of those General Rules.

2) Otherwise, let *Y* be the shortest character string that conforms to the definition of <literal> in Subclause 5.3, "<literal>", and such that the interpreted value of *Y* is *SV* and the interpreted precision of *Y* is the precision of *SD*. If *SV* is a negative interval, then <sign> shall be specified within <unquoted interval string> in the literal *Y*.

ii) Case:

1) If *Y* contains any <SQL language character> that is not in the character repertoire of *TD*, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.

2) If the length in characters *LY* of *Y* is less than or equal to *MLTD*, then *TV* is *Y*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

f) If *SD* is Boolean, then

Case:

i) If *SV* is *True* and *MLTD* is not less than 4, then *TV* is 'TRUE'.

ii) If *SV* is *False* and *MLTD* is not less than 5, then *TV* is 'FALSE'.

iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast (22018)*.

13) If *TD* and *SD* are binary string types, then

Case:

a) If *TD* is fixed-length binary string, then let *LTD* be the length in octets of *TD*.

i) If the length in octets of *SV* is equal to *LTD*, then *TV* is *SV*.

ii) If the length in octets of *SV* is larger than *LTD*, then *TV* is the first *LTD* octets of *SV* and a completion condition is raised: *warning — string data, right truncation (01004)*.

iii) If the length in octets *M* of *SV* is smaller than *LTD*, then *TV* is *SV* extended on the right by *LTD*−*M* X'00's.

b) Otherwise, let *MLTD* be the maximum length in octets of *TD*.

Case:

i) If the length in octets of *SV* is less than or equal to *MLTD*, then *TV* is *SV*.

ii) If the length in octets of *SV* is larger than *MLTD*, then *TV* is the first *MLTD* octets of *SV* and a completion condition is raised: *warning — string data, right truncation (01004)*.

14) If *TD* is the datetime data type DATE, then

Case:

a) If *SD* is character string, then

Case:

i) If <cast template> *CT* is specified, then the General Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with DATE as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *SV* as *FORMATTED CHARACTER STRING*; let *TV* be the *DATETIME VALUE* returned from the application of those General Rules.

ii) Otherwise, *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

1) If the rules for <literal> or for <unquoted date string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

2) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.

b) If *SD* is the datetime data type DATE, then *TV* is *SV*.

c) If *SD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then *TV* is the year, month, and day <primary datetime field>s of *SV*.

d) If *SD* is the datetime data type TIMESTAMP WITH TIME ZONE, then *TV* is computed by:

```
CAST ( CAST ( VE AS TIMESTAMP WITHOUT TIME ZONE ) AS DATE )
```

15) Let *STZD* be the current default time zone displacement of the SQL-session.

16) If *TD* is the datetime data type TIME WITHOUT TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then:

i) If <cast template> *CT* is specified, then the General Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with TIME(*TSP*) WITHOUT TIME ZONE as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *SV* as *FORMATTED CHARACTER STRING*; let *TV* be the *DATETIME VALUE* returned from the application of those General Rules.

ii) Otherwise, *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

1) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

2) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITH TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIME(TSP) WITH TIME ZONE ) AS
TIME(TSP) WITHOUT TIME ZONE )
```

3) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.

b) If *SD* is TIME WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined (IA002) rounding or truncation if necessary.

c) If *SD* is TIME WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, computed modulo 24 hours, with implementation-defined (IA002) rounding or truncation if necessary.

d) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined (IA002) rounding or truncation if necessary.

e) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIME(TSP) WITHOUT TIME ZONE )
```

17) If *TD* is the datetime data type TIME WITH TIME ZONE, then let *TSP* be the <time precision> of *TD*.

Case:

a) If *SD* is character string, then:

i) If <cast template> *CT* is specified, then the General Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with TIME(*TSP*) WITH TIME ZONE as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *SV* as *FORMATTED CHARACTER STRING*; let *TV* be the *DATETIME VALUE* returned from the application of those General Rules.

ii) Otherwise, *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

1) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

2) If the rules for <literal> or for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIME(*TSP*) WITHOUT TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIME(TSP) WITHOUT TIME ZONE )
AS TIME(TSP) WITH TIME ZONE )
```

3) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.

b) If *SD* is TIME WITH TIME ZONE, then *TV* is *SV*, with implementation-defined (IA002) rounding or truncation if necessary.

c) If *SD* is TIME WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, computed modulo 24 hours, with implementation-defined (IA002) rounding or truncation if necessary, and the time zone displacement of *TV* is *STZD*.

d) If *SD* is TIMESTAMP WITH TIME ZONE, then the UTC component of *TV* is the hour, minute, and second <primary datetime field>s of *SV*, with implementation-defined (IA002) rounding or truncation if necessary, and the time zone component of *TV* is the time zone displacement of *SV*.

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITH TIME ZONE )
AS TIME(TSP) WITH TIME ZONE )
```

18) If *TD* is the datetime data type TIMESTAMP WITHOUT TIME ZONE, then let *TSP* be the <timestamp precision> of *TD*.

Case:

a) If *SD* is character string, then:

i) If <cast template> *CT* is specified, then the General Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with TIMESTAMP(*TSP*) WITHOUT TIME ZONE as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *SV* as *FORMATTED CHARACTER STRING*; let *TV* be the *DATETIME VALUE* returned from the application of those General Rules.

ii) Otherwise, *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

1) If the rules for <literal> or for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

2) If the rules for <literal> or for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITH TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITH TIME ZONE )
AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
```

3) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.

b) If *SD* is a date, then the <primary datetime field>s hour, minute, and second of *TV* are set to 0 (zero) and the <primary datetime field>s year, month, and day of *TV* are set to their respective values in *SV*.

c) If *SD* is TIME WITHOUT TIME ZONE, then the <primary datetime field>s year, month, and day of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second of *TV* are set to their respective values in *SV*, with implementation-defined (IA002) rounding or truncation if necessary.

d) If *SD* is TIME WITH TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP WITH TIME ZONE )
AS TIMESTAMP WITHOUT TIME ZONE )
```

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then *TV* is *SV*, with implementation-defined (IA002) rounding or truncation if necessary.

f) If *SD* is TIMESTAMP WITH TIME ZONE, then let *SVUTC* be the UTC component of *SV* and let *SVTZ* be the time zone displacement of *SV*. *TV* is *SVUTC* + *SVTZ*, with implementation-defined (IA002) rounding or truncation if necessary.

19) If *TD* is the datetime data type TIMESTAMP WITH TIME ZONE, then let *TSP* be the <timestamp precision> of *TD*.

Case:

a) If *SD* is character string, then:

i) If <cast template> *CT* is specified, then the General Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with TIMESTAMP(*TSP*) WITH TIME ZONE as *DATETIME TYPE*, the value of *CT* as *TEMPLATE*, and *SV* as *FORMATTED CHARACTER STRING*; let *TV* be the *DATETIME VALUE* returned from the application of those General Rules.

ii) Otherwise, *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

1) If the rules for <literal> or for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

2) If the rules for <literal> or for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type TIMESTAMP(*TSP*) WITHOUT TIME ZONE, then let *TV* be the value of:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIMESTAMP(TSP) WITH TIME ZONE )
```

3) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.

b) If *SD* is a date, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIMESTAMP(TSP) WITH TIME ZONE )
```

c) If *SD* is TIME WITHOUT TIME ZONE, then *TV* is:

```
CAST ( CAST ( VE AS TIMESTAMP(TSP) WITHOUT TIME ZONE )
AS TIMESTAMP(TSP) WITH TIME ZONE )
```

d) If *SD* is TIME WITH TIME ZONE, then the <primary datetime field>s year, month, and day of *TV* are set to their respective values in an execution of CURRENT_DATE and the <primary datetime field>s hour, minute, and second of *TV* are set to their respective values in *SV*, with implementation-defined (IA002) rounding or truncation if necessary. The time zone component of *TV* is set to the time zone displacement of *SV*.

e) If *SD* is TIMESTAMP WITHOUT TIME ZONE, then the UTC component of *TV* is *SV* – *STZD*, with a time zone displacement of *STZD*.

f) If *SD* is TIMESTAMP WITH TIME ZONE, then *TV* is *SV* with implementation-defined (IA002) rounding or truncation, if necessary.

20) If *TD* is interval, then

Case:

a) If *SD* is exact numeric, then

Case:

i) If the representation of *SV* in the data type *TD* would result in the loss of leading significant digits, then an exception condition is raised: *data exception — interval field overflow (22015)*.

ii) If the number of digits of fractional seconds precision *NDFSP* of *TD* is less than *NDSEN*, where *NDSEN* is the scale of the <exact numeric literal> formed by the <cast specification>

```
CAST (SV AS CHARACTER VARYING(max))
```

and *max* is the implementation-defined (IL006) maximum length of variable-length character strings, then it is implementation-defined (IA002) whether *TV* is determined by rounding *SV* to *NDFSP* digits of precision or by truncating *SV* to *NDFSP* digits of precision, as specified in Subclause 4.5.2, "Characteristics of numbers".

iii) Otherwise, *TV* is the representation of *SV* in the data type *TD*.

b) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> or for <unquoted interval string> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise, an exception condition is raised: *data exception — invalid interval format (22006)*.

c) If *SD* is interval and *TD* and *SD* have the same interval precision, then *TV* is *SV*.

d) If *SD* is interval and *TD* and *SD* have different interval precisions, then let *P* and *Q* be the most and least significant <primary datetime field>s of *TD*, respectively.

i) Let *Y* be the result of converting *SV* to a scalar in units *Q* (that is, observing the rules that there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year).

ii) Normalize *Y* to conform to the <interval qualifier> "*P* TO *Q*" of *TD* (again, observing the rules that there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 12 months in a year). Whether to truncate or round in the least significant field of the result is implementation-defined (IA002). If this would result in loss of precision of the leading datetime field of *Y*, then an exception condition is raised: *data exception — interval field overflow (22015)*.

iii) *TV* is the value of *Y*.

21) 09 If *TD* is Boolean, then

Case:

a) If *SD* is character string, then *SV* is replaced by

```
TRIM ( BOTH ' ' FROM VE )
```

Case:

i) If the rules for <literal> in Subclause 5.3, "<literal>", can be applied to *SV* to determine a valid value of the data type *TD*, then let *TV* be that value.

ii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast (22018)*.

b) If *SD* is Boolean, then *TV* is *SV*.

22) If *SD* is JSON, then *TV* is *SV*.

23) If the <cast specification> contains a <domain name> and that <domain name> refers to a domain that contains a <domain constraint> and if *TV* does not satisfy the <check constraint definition> simply contained in the <domain constraint>, then an exception condition is raised: *integrity constraint violation (23000)*.

24) The result of *CS* is *TV*.

Conformance Rules

1) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <cast operand> whose declared type is BINARY LARGE OBJECT or CHARACTER LARGE OBJECT.

2) Without Feature F421, "National character", conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, or NATIONAL CHARACTER LARGE OBJECT.

3) Without Feature T042, "Extended LOB data type support", conforming SQL language shall not contain a <cast operand> whose declared type is NATIONAL CHARACTER LARGE OBJECT.

4) Without Feature S043, "Enhanced reference types", in conforming SQL language, if the declared type of <cast operand> is a reference type, then <cast target> shall contain a <data type> that is a reference type.

5) 15 Without Feature T839, "Formatted cast of datetimes to/from character strings", conforming SQL language shall not contain <cast template>.

6.14 <next value expression>

This Subclause is modified by Subclause 6.3, "<next value expression>", in ISO/IEC 9075-4.

Function

Return the next value of a sequence generator.

Format

```
<next value expression> ::=
  NEXT VALUE FOR <sequence generator name>
```

Syntax Rules

1) A <next value expression> shall be directly contained in one of the following:
   a) A <select list> simply contained in a <query specification> that constitutes a <query expression> that is immediately contained in one of the following:
      i) A <cursor specification>.
      ii) A <table subquery> simply contained in an <as subquery clause> in a <table definition>.
      iii) A <from subquery>.
      iv) A <select statement: single row>.
   b) A <select list> simply contained in a <query specification> that is immediately contained in a <dynamic single row select statement>.
   c) A <from constructor>.
   d) A <merge insert value list>.
   e) 04 An <update source>.
2) <next value expression> shall not be contained in a <case expression>, a <search condition>, an <order by clause>, an <aggregate function>, a <window function>, a grouped query, or in a <query specification> that simply contains the <set quantifier> DISTINCT.
3) 04 The declared type of <next value expression> is the data type described by the data type descriptor included in the sequence generator descriptor identified by <sequence generator name>.

Access Rules

1) Case:
   a) If <next value expression> is contained in a <schema definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include USAGE privilege on the sequence generator identified by <sequence generator name>.
   b) Otherwise, the current privileges shall include USAGE privilege on the sequence generator identified by <sequence generator name>.

General Rules

1) If <next value expression> *NVE* is specified, then let *SEQ* be the sequence generator descriptor identified by the <sequence generator name> contained in *NVE*.

   Case:

   a) If *NVE* is directly contained in a <query specification> *QS*, then, for each row *RQS* in the result of *QS*, the General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are applied with *SEQ* as *SEQUENCE*; let the value for *RQS* be the *RESULT* returned from the application of those General Rules.

   b) If *NVE* is directly contained in a <contextually typed table value constructor> *TVC*, then, for each <contextually typed row value expression> *CTRVE* contained in *TVC*, the General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are applied with *SEQ* as *SEQUENCE*; let the value for *CTRVE* be the *RESULT* returned from the application of those General Rules.

   c) If *NVE* is directly contained in an <update source>, then, for each row *USR* to be updated by the <update statement: searched> or <update statement: positioned>, the General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are applied with *SEQ* as *SEQUENCE*; let the value for *USR* be the *RESULT* returned from the application of those General Rules.

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <next value expression>.

6.15 <greatest or least function>

Function

Specify functions yielding the greatest or least value of its arguments.

Format

```
<greatest or least function> ::=
  <greatest or least> <left paren> <value expression>
      { <comma> <value expression> }... <right paren>

<greatest or least> ::=
    GREATEST | LEAST
```

Syntax Rules

1) Let *GLF* be the <greatest or least function>.

2) Let *NV* be the number of <value expression>s immediately contained in *GLF*.

3) Let *VV* be the set of <value expression>s immediately contained in *GLF* and let V_i, 1 (one) $\leq i \leq NV$, be the *i*-th <value expression> immediately contained in *GLF*.

4) For every pair V_i, 1 (one) $\leq i \leq NV$, and V_j, 1 (one) $\leq j \leq NV$, in *VV*, V_i and V_j shall be comparable.

5) Each V_i, 1 (one) $\leq i \leq NV$, is an operand of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 9.14, "Ordering operations", apply.

6) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the set of declared types of all <value expression>s simply contained in the <greatest or least function> as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules. The declared type of the <greatest or least function> is *RT*.

7) Let C_i, 1 (one) $\leq i \leq NV$, be

   ```
   CAST (Vi AS RT)
   ```

Access Rules

None.

General Rules

1) If the value of any V_i, 1 (one) $\leq i \leq NV$, is the null value, then the result of *GLF* is the null value and no further General Rules of this Subclause are applied.

2) If GREATEST is specified, then let *OP* be >=. If LEAST is specified, then let *OP* be <=.

3) Let *R* be the value of some C_x, 1 (one) $\leq x \leq NV$, for *x* such that the result of the implied <comparison predicate> "C_x *OP* C_j" is *True* for all *j*, 1 (one) $\leq j \leq NV$. If there is no such *x*, then let *R* be the null value.

4) The result of *GLF* is *R*.

Conformance Rules

1) Without Feature T054, “GREATEST and LEAST”, conforming SQL language shall not contain a <greatest or least function>.

6.16 <field reference>

Function

Reference a field of a row value.

Format

```
<field reference> ::=
  <value expression primary> <period> <field name>
```

Syntax Rules

1) Let *FR* be the <field reference>, let *VEP* be the <value expression primary> immediately contained in *FR*, and let *FN* be the <field name> immediately contained in *FR*.

2) The declared type of *VEP* shall be a row type. Let *RT* be that row type.

3) *FR* is a *field reference*.

4) *FN* shall unambiguously reference a field of *RT*. Let *F* be that field.

5) The declared type of *FR* is the declared type of *F*.

Access Rules

None.

General Rules

1) Let *VR* be the value of *VEP*.

2) Case:

   a) If *VR* is the null value, then the value of *FR* is the null value.

   b) Otherwise, the value of *FR* is the value of the field *F* of *VR*.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain a <field reference>.

6.17 <subtype treatment>

Function

Modify the declared type of an expression.

Format

```
<subtype treatment> ::=
  TREAT <left paren> <subtype operand> AS <target subtype> <right paren>

<subtype operand> ::=
  <value expression>

<target subtype> ::=
    <path-resolved user-defined type name>
  | <reference type>
```

Syntax Rules

1) The declared type *VT* of the <value expression> shall be a structured type or a reference type.

2) Case:

   a) If *VT* is a structured type, then:

      i) <target subtype> shall specify a <path-resolved user-defined type name>.

      ii) Let *DT* be the structured type identified by the <user-defined type name> simply contained in <path-resolved user-defined type name>.

   b) Otherwise:

      i) <target subtype> shall specify a <reference type>.

      ii) Let *DT* be the reference type identified by <reference type>.

3) *VT* shall be a supertype of *DT*.

4) The declared type of the result of the <subtype treatment> is *DT*.

Access Rules

None.

General Rules

1) Let *V* be the value of the <value expression>.

2) Case:

   a) If *V* is the null value, then the value of the <subtype treatment> is the null value.

   b) Otherwise:

      i) If the most specific type of *V* is not a subtype of *DT*, then an exception condition is raised: *invalid target type specification (0D000)*.

NOTE 174 — "most specific type" is defined in Subclause 4.9.3.4, "Subtypes and supertypes".

ii) The value of the <subtype treatment> is *V*.

Conformance Rules

1) Without Feature S161, "Subtype treatment", conforming SQL Language shall not contain a <subtype treatment>.

2) Without Feature S162, "Subtype treatment for references", conforming SQL language shall not contain a <target subtype> that contains a <reference type>.

6.18 <method invocation>

This Subclause is modified by Subclause 6.1, "<method invocation>", in ISO/IEC 9075-13.

Function

Reference an SQL-invoked method of a user-defined type value.

Format

```
<method invocation> ::=
    <direct invocation>
  | <generalized invocation>

<direct invocation> ::=
  <value expression primary> <period> <method name> [ <SQL argument list> ]

<generalized invocation> ::=
  <left paren> <value expression primary> AS <data type> <right paren>
      <period> <method name> [ <SQL argument list> ]

<method selection> ::=
  <routine invocation>

<constructor method selection> ::=
  <routine invocation>
```

Syntax Rules

1) Let *OR* be the <method invocation>.

2) The Syntax Rules of Subclause 9.19, "Processing a method invocation", are applied with *OR* as *METHOD INVOCATION*; let *RI* be the *ROUTINE INVOCATION*, let *GENORCONS* be the *GENERALIZED OR CONSTRUCTOR*, and let *TP* be the *SQLPATH* returned from the application of those Syntax Rules.

3) Case:

   a) If *GENORCONS* is 'M', then let *MS* be the <method selection>:

      RI

   b) Otherwise, let *CMS* be the <constructor method selection>:

      RI

   NOTE 175 — These subrules set up immediate containment context used by Subclause 9.18, "Invoking an SQL-invoked routine", to determine the proper call type.

4) The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *RI* as *ROUTINE INVOCATION*, *TP* as *SQLPATH*, and the null value as *UDT*; let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

1) The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *V* be the *VALUE* returned from the application of those General Rules.

2) The value of <method invocation> is *V*.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <method invocation>.

6.19 <static method invocation>

Function

Invoke a static method.

Format

```
<static method invocation> ::=
  <path-resolved user-defined type name> <double colon> <method name>
      [ <SQL argument list> ]

<static method selection> ::=
  <routine invocation>
```

Syntax Rules

1) Let *TN* be the <user-defined type name> immediately contained in <path-resolved user-defined type name> and let *T* be the user-defined type identified by *TN*.

2) Let *MN* be the <method name> immediately contained in <static method invocation>.

3) Case:

   a) If <SQL argument list> is specified, then let *AL* be that <SQL argument list>.

   b) Otherwise, let *AL* be <left paren> <right paren>.

4) Let *TP* be an SQL-path containing only the <schema name> of every schema that includes a descriptor of a supertype of *T*.

5) Let *RI* be the following <routine invocation>:

   ```
   MN AL
   ```

6) Let *SMS* be the following <static method selection>:

   ```
   RI
   ```

   NOTE 176 — This sets up immediate containment context used by Subclause 9.18, "Invoking an SQL-invoked routine", to determine the proper call type.

7) The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *RI* as *ROUTINE INVOCATION*, *TP* as *SQLPATH*, and *T* as *UDT*; let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

1) The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *V* be the *VALUE* returned from the application of those General Rules.

2) The value of <static method invocation> is *V*.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <static method invocation>.

6.20 <new specification>

This Subclause is modified by Subclause 6.2, "<new specification>", in ISO/IEC 9075-13.

Function

Invoke a method on a structured type.

Format

```
<new specification> ::=
  NEW <path-resolved user-defined type name> <SQL argument list>

<new invocation> ::=
    <method invocation>
  | <routine invocation>
```

Syntax Rules

1) Let *UDTN* be the <path-resolved user-defined type name> immediately contained in the <new specification>. Let *MN* be the <qualified identifier> immediately contained in *UDTN*.

2) Let *UDT* be the user-defined type identified by *UDTN*. *UDT* shall be instantiable. Let *SN* be the implicit or explicit <schema name> of *UDTN*. Let *S* be the schema identified by *SN*. Let *RN* be *SN*.*MN*.

3) Case:

   a) If the <new specification> is of the form

      NEW *UDTN*()

      then

      Case:

      i) If *S* does not include the descriptor of an SQL-invoked constructor method whose method name is equivalent to *MN* and whose unaugmented SQL parameter declaration list is empty, then the <new specification> is equivalent to the <new invocation>

         RN()

      ii) Otherwise, the <new specification> is equivalent to the <new invocation>

         RN().*MN*()

   b) Otherwise, the <new specification>

      NEW *UDTN*(a_1, a_2, ..., a_n)

      is equivalent to the <new invocation>

      RN().*MN*(a_1, a_2, ..., a_n)

Access Rules

None.

NOTE 177 — The applicable privileges or current privileges (as appropriate) include EXECUTE privilege on the constructor function, and also on the indicated constructor method, according to the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine".

General Rules

13 *None.*

Conformance Rules

1) 13 Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <new specification>.

6.21 <attribute or method reference>

Function

Return a value acquired by accessing a column of the row identified by a value of a reference type or by invoking an SQL-invoked method.

Format

```
<attribute or method reference> ::=
  <value expression primary> <dereference operator> <qualified identifier>
      [ <SQL argument list> ]

<dereference operator> ::=
  <right arrow>
```

Syntax Rules

1) The declared type of the <value expression primary> *VEP* shall be a reference type and the scope included in its reference type descriptor shall not be empty. Let *RT* be the referenced type of *VEP*.

2) Let *QI* be the <qualified identifier>. If <SQL argument list> is specified, then let *SAL* be <SQL argument list>; otherwise, let *SAL* be the zero-length character string.

3) Case:

   a) If *QI* is equivalent to the attribute name of an attribute of *RT* and *SAL* is the zero-length character string, then <attribute or method reference> is effectively replaced by a <dereference operation> *AMR* of the form:

      ```
      VEP -> QI
      ```

   b) Otherwise, <attribute or method reference> is effectively replaced by a <method reference> *AMR* of the form:

      ```
      VEP -> QI SAL
      ```

4) The declared type of <attribute or method reference> is the declared type of *AMR*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S041, “Basic reference types”, conforming SQL language shall not contain an <attribute or method reference>.

6.22 <dereference operation>

Function

Access a column of the row identified by a value of a reference type.

Format

```
<dereference operation> ::=
  <reference value expression> <dereference operator> <attribute name>
```

Syntax Rules

1) Let *RVE* be the <reference value expression>. The reference type descriptor of *RVE* shall include a scope. Let *RT* be the referenced type of *RVE*.

2) Let *AN* be the <attribute name>. *AN* shall identify an attribute *AT* of *RT*.

3) The declared type of the <dereference operation> is the declared type of *AT*.

4) Let *S* be the name of the referenceable table in the scope of the reference type of *RVE*.

5) Let *OID* be the name of the self-referencing column of *S*.

6) <dereference operation> is equivalent to a <scalar subquery> of the form:

```
( SELECT AN
  FROM S
  WHERE S.OID = RVE )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <dereference operation>.

6.23 <method reference>

Function

Return a value acquired from invoking an SQL-invoked routine that is a method.

Format

```
<method reference> ::=
  <value expression primary> <dereference operator> <method name> <SQL argument list>
```

Syntax Rules

1) The declared type of the <value expression primary> *VEP* shall be a reference type and the scope included in its reference type descriptor shall not be empty.

2) Let *MN* be the method name specified by <method name>. Let *MRAL* be the <SQL argument list>.

3) The Syntax Rules of Subclause 9.19, "Processing a method invocation", are applied with `DEREF (`*VEP*`).`*MN* *MRAL* as *METHOD INVOCATION*; let *RI* be the *ROUTINE INVOCATION*, let *GENORCONS* be the *GENERALIZED OR CONSTRUCTOR*, and let *TP* be the *SQLPATH* returned from the application of those Syntax Rules.

4) Case:

   a) If *GENORCONS* is 'M', then let *MS* be the <method selection>:

      RI

   b) Otherwise, let *CMS* be the <constructor method selection>:

      RI

   NOTE 178 — These subrules set up immediate containment context used by Subclause 9.18, "Invoking an SQL-invoked routine", to determine the proper call type.

5) The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *RI* as *ROUTINE INVOCATION*, *TP* as *SQLPATH*, and the null value as *UDT*; let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* returned from the application of those Syntax Rules.

6) The declared type of <method reference> is the data type of the expression:

   `DEREF (`*VEP*`).`*MN* *MRAL*

Access Rules

1) Let *SCOPE* be the table that is the scope of *VEP*.

   Case:

   a) If <method reference> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include the table/method privilege for table *SCOPE* and method *SR*.

   b) Otherwise, the current privileges shall include the table/method privilege for table *SCOPE* and method *SR*.

General Rules

1) The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *V* be the *VALUE* returned from the application of those General Rules.

2) The value of <method reference> is *V*.

Conformance Rules

1) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <method reference>.

6.24 <reference resolution>

Function

Obtain the value referenced by a REF value.

Format

```
<reference resolution> ::=
  DEREF <left paren> <reference value expression> <right paren>
```

Syntax Rules

1) Let *RR* be the <reference resolution> and let *RVE* be the <reference value expression>. The reference type descriptor of *RVE* shall include a scope.

2) The declared type of *RR* is the structured type that is referenced by the declared type of *RVE*.

3) Let *SCOPE* be the table identified by the table name included in the reference type descriptor of *RVE*. *SCOPE* is the scoped table of *RR*.

NOTE 179 — The term "scoped table" is defined in Subclause 4.11, "Reference types".

Access Rules

1) Case:

   a) If <reference resolution> is contained in a <schema definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *SCOPE*.

   b) Otherwise, the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *SCOPE*.

General Rules

1) Let m be the number of subtables of *SCOPE*. Let S_i, 1 (one) $\le i \le m$, be the subtables, arbitrarily ordered, of *SCOPE*.

2) For each S_i, 1 (one) $\le i \le m$, let SN_i be the name of S_i, let STN_i be the name included in the descriptor of S_i of the structured type ST_i associated with S_i, let $REFCOL_i$ be the name of the self-referencing column of S_i, let N_i be the number of attributes of ST_i, and let $A_{i,j}$, 1 (one) $\le j \le N_i$, be the names of the attributes of ST_i, therefore also the names of the columns of S_i.

3) The value of <reference resolution> is the value of:

```
(
  SELECT A1,1 ( ... A1,N1
         ( STN1(), A1,N1 ), ... A1,1 )
  FROM ONLY SN1
  WHERE S1.REFCOL1 = RVE
 UNION
  SELECT A2,1 ( ... A2,N2
         ( STN2(), A2,N2 ), ... A2,1 )
```

```
  FROM ONLY SN2
  WHERE S2.REFCOL2 = RVE
UNION
...
UNION
  SELECT Am,1 ( ... Am,Nm
         ( STNm(), Am,Nm ), ... Am,1 )
  FROM ONLY SNm
  WHERE Sm.REFCOLm = RVE
)
```

NOTE 180 — The evaluation of this General Rule is effectively performed without further Access Rule checking.

Conformance Rules

1) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <reference resolution>.

6.25 <array element reference>

Function

Return an element of an array.

Format

```
<array element reference> ::=
  <array value expression>
      <left bracket or trigraph> <numeric value expression> <right bracket or trigraph>
```

Syntax Rules

1) The declared type of an <array element reference> is the element type of the specified <array value expression>.

2) The declared type of <numeric value expression> shall be exact numeric with scale 0 (zero).

Access Rules

None.

General Rules

1) If the value of <array value expression> or <numeric value expression> is the null value, then the result of <array element reference> is the null value.

2) Let *i* be the value of <numeric value expression>.

   Case:

   a) If *i* is greater than zero and less than or equal to the cardinality of <array value expression>, then the result of <array element reference> is the value of the *i*-th element of the value of <array value expression>.

   b) Otherwise, an exception condition is raised: *data exception — array element error (2202E)*.

Conformance Rules

1) Without Feature S090, "Minimal array support", conforming SQL language shall not contain an <array element reference>.

6.26 <multiset element reference>

Function

Return the sole element of a multiset of one element.

Format

```
<multiset element reference> ::=
  ELEMENT <left paren> <multiset value expression> <right paren>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression>. The <multiset element reference> is equivalent to the <scalar subquery>

```
( SELECT M.E
  FROM UNNEST (MVE) AS M(E) )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset element reference>.

6.27 <row pattern navigation operation>

Function

Return the value of a value expression evaluated in a row determined by navigation within the row pattern partition using logical and physical offsets from a row mapped to a row pattern variable.

Format

```
<row pattern navigation operation> ::=
    <row pattern navigation: logical>
  | <row pattern navigation: physical>
  | <row pattern navigation: compound>

<row pattern navigation: logical> ::=
  [ <running or final> ] <first or last>
      <left paren> <value expression> [ <comma> <logical offset> ] <right paren>

<row pattern navigation: physical> ::=
  <prev or next>
      <left paren> <value expression> [ <comma> <physical offset> ] <right paren>

<row pattern navigation: compound> ::=
  <prev or next> <left paren> [ <running or final> ] <first or last>
      <left paren> <value expression> [ <comma> <logical offset> ] <right paren>
      [ <comma> <physical offset> ] <right paren>

<first or last> ::=
    FIRST | LAST

<prev or next> ::=
    PREV | NEXT

<logical offset> ::=
    <simple value specification>
  | <dynamic parameter specification>

<physical offset> ::=
    <simple value specification>
  | <dynamic parameter specification>
```

Syntax Rules

1) Let *RPNO* be the <row pattern navigation operation>. *RPNO* shall be contained in a <row pattern measures> or a <row pattern definition search condition>.

2) The <value expression> *VE* simply contained in *RPNO* shall not contain a <set function specification>, <window function>, or <row pattern navigation operation>.

3) *VE* shall contain at least one row pattern column reference or <classifier function>.

4) There shall be exactly one row pattern variable *RPV* such that *RPV* qualifies every column reference contained in *VE* and such that the explicit or implicit argument of every <classifier function> is *RPV*.

5) If <row pattern navigation: logical> or <row pattern navigation: compound> is specified, then:

   a) If <running or final> is not specified, then RUNNING is implicit.

b) If FINAL is specified, then *RPNO* shall not be contained in a <row pattern definition search condition>.

c) Case:

i) If <logical offset> *LO* is specified, then the declared type of *LO* shall be exact numeric with scale 0 (zero). If *LO* is a <literal>, then the value of *LO* shall be non-negative.

ii) Otherwise, the <literal> 0 (zero) is implicit.

6) If <row pattern navigation: physical> or <row pattern navigation: compound> is specified, then

Case:

a) If <physical offset> *PO* is specified, then the declared type of *PO* shall be exact numeric with scale 0 (zero). If *PO* is a <literal>, then the value of *PO* shall be non-negative.

b) Otherwise, the <literal> 1 (one) is implicit.

7) If <row pattern navigation: physical> is specified, then let *PON* be the <prev or next>, and let *POX* be the specified or implicit <physical offset>. *RPNO* is equivalent to the <row pattern navigation: compound>

```
PON (RUNNING LAST (VE, 0), POX)
```

8) If <row pattern navigation: logical> is specified, then let *ROF* be the specified or implicit <running or final>, let *FOL* be the <first or last>, and let *LOX* be the specified or implicit <logical offset>. *RPNO* is equivalent to the <row pattern navigation: compound>

```
PREV (ROF FOL (VE, LOX), 0)
```

Access Rules

None.

General Rules

1) If <row pattern navigation: compound> *RPNC* is specified, then let *PON* be the <prev or next>, let *ROF* be the <running or final>, let *FOL* be the <first or last>, let *LOV* be the value of the <logical offset>, and let *POV* be the value of the <physical offset>.

2) If either *LOV* or *POV* is not 0 (zero) or positive, then an exception condition is raised: *data exception — invalid argument for row pattern navigation function (2202J)*.

3) Case:

a) If *RPNC* is contained in <row pattern recognition clause> *RPRC*, then let *RPOB* be the <row pattern order by> simply contained in *RPRC*.

b) Otherwise, let *RPOB* be the <window order clause> of the window structure descriptor of the <window clause> that contains *RPNC*.

4) Case:

a) If *RPNC* is contained in a <row pattern definition search condition>, then let (*STR*, *RS*, *k*) be the current potential row pattern match.

b) If *RPNC* is contained in a <row pattern recognition clause>, then let (*STR*, *RS*, *k*) be the current retained row pattern match.

c) Otherwise, let (*STR*, *RS*, *k*) be the current designated row pattern match.

NOTE 181 — The definitions of potential, retained, and designated row pattern matches are expressed as "(*STR*, *RS*, *k*)", as specified in Subclause 9.41, "Row pattern recognition in a sequence of rows".

5) Let *SR1* be the set of rows that are mapped to *RPV* by (*STR*, *RS*, *k*)

6) Case:

a) If *ROF* is RUNNING, then let *SR2* be the set of rows in *SR1* that are the same as the current row, or that precede the current row in the ordering of rows according to *RPOB*.

b) Otherwise, let *SR2* be *SR1*.

7) Let *RP* be the row pattern partition that contains the current row.

NOTE 182 — If *RP* is a window partition, then *RS* is typically a proper subset of *RP*.

8) Let *NP* be the number of rows in *RP*. Let { R_1, ... R_{NP} } be the enumeration of the rows of *RP* in the ordering specified by *RPOB*.

9) Let *LN* be the number of rows in *SR2*. Let the rows of *SR2* be MR_1, ..., MR_{LN}, in the order of rows according to *RPOB*.

10) Let *RV* be an implementation-dependent (UV072) range variable that is distinct from every range variable whose scope includes *RPNC*. Let *VX* be the <value expression> obtained from *VE* by replacing every occurrence of *RPV* by *RV*.

11) Case:

a) If $LOV \geq LN$, then the result of *RPNC* is the null value.

b) Otherwise:

i) Case:

1) If *FOL* is FIRST, then let *LR* be MR_{LOV+1}.

2) Otherwise, let *LR* be MR_{LN-LOV}.

ii) Let *n* be the row number of *LR* in *RP*, so that *LR* is R_n.

iii) Case:

1) If *PON* is PREV, then

Case:

A) If $POV \geq n$, then the result of *RPNC* is the null value.

B) Otherwise, the result of *RPNC* is the value of *VX* evaluated with range variable *RV* identifying row R_{n-POV}.

2) Otherwise:

A) If $n+POV > NP$, then the result of *RPNC* is the null value.

B) Otherwise, the result of *RPNC* is the value of *VX* evaluated with range variable *RV* identifying row R_{n+POV}.

Conformance Rules

1) Without at least one of Feature R010, "Row pattern recognition: FROM clause", or Feature R020, "Row pattern recognition: WINDOW clause", conforming SQL language shall not contain <row pattern navigation operation>.

6.28 <JSON value function>

Function

Extract an SQL value of a predefined type from a JSON value.

Format

```
<JSON value function> ::=
  JSON_VALUE <left paren>
      <JSON API common syntax>
      [ <JSON returning clause> ]
      [ <JSON value empty behavior> ON EMPTY ]
      [ <JSON value error behavior> ON ERROR ]
  <right paren>

<JSON returning clause> ::=
  RETURNING <data type>

<JSON value empty behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>

<JSON value error behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>
```

Syntax Rules

1) If <JSON returning clause> is not specified, then an implementation-defined (ID074) character string type is implicit.

2) The <data type> *DT* contained in the explicit or implicit <JSON returning clause> shall be a <predefined type> that identifies a character string data type, numeric data type, Boolean data type, or datetime data type.

3) The declared type of <JSON value function> is the type specified by *DT*.

4) If <JSON value empty behavior> is not specified, then NULL ON EMPTY is implicit.

5) If <JSON value error behavior> is not specified, then NULL ON ERROR is implicit.

Access Rules

None.

General Rules

1) Let *JACS* be the <JSON API common syntax>.

2) If the value of the <JSON context item> simply contained in *JACS* is the null value, then the result of <JSON value function> is the null value and no further General Rules of this Subclause are applied.

3) The General Rules of Subclause 9.47, "Processing <JSON API common syntax>", are applied with *JACS* as *JSON API COMMON SYNTAX*; let *ST1* be the *STATUS* and let *SEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

4) Let *ZB* be the explicit or implicit <JSON value empty behavior>, and let *EB* be the explicit or implicit <JSON value error behavior> contained in the <JSON value function>.

5) The General Rules of Subclause 9.48, "Casting an SQL/JSON sequence to an SQL type", are applied with *ST1* as *STATUS IN*, *SEQ* as *SQL/JSON SEQUENCE*, *ZB* as *EMPTY BEHAVIOR*, *EB* as *ERROR BEHAVIOR*, and *DT* as *DATA TYPE*; let *SR2* be the *STATUS OUT* and let *V* be the *VALUE* returned from the application of those General Rules.

6) If *ST2* is an exception condition, then the exception condition *ST2* is raised. Otherwise, *V* is the result of the <JSON value function>.

Conformance Rules

1) Without Feature T821, "Basic SQL/JSON query operators", conforming SQL language shall not contain <JSON value function>.

2) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON value function> shall not contain <JSON value empty behavior>.

3) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON value function> shall not contain <JSON value error behavior>.

4) Without Feature T826, "General value expression in ON ERROR or ON EMPTY clauses", the <value expression> contained in <JSON value empty behavior> or <JSON value error behavior> shall be a <literal> that can be cast to the data type specified by the explicit or implicit <JSON returning clause> without raising an exception condition according to the General Rules of Subclause 6.13, "<cast specification>".

6.29 <value expression>

This Subclause is modified by Subclause 6.3, "<value expression>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.8, "<value expression>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 6.8, "<value expression>", in ISO/IEC 9075-15.

Function

Specify a value.

Format

```
<value expression> ::=
    <common value expression>
  | <boolean value expression>
  | <row value expression>

09 14 <common value expression> ::=
    <numeric value expression>
  | <string value expression>
  | <datetime value expression>
  | <interval value expression>
  | <user-defined type value expression>
  | <reference value expression>
  | <collection value expression>
  | <JSON value expression>

<user-defined type value expression> ::=
  <value expression primary>

<reference value expression> ::=
  <value expression primary>

15 <collection value expression> ::=
    <array value expression>
  | <multiset value expression>
```

Syntax Rules

1) The declared type of a <value expression> is the declared type of the immediately contained <common value expression>, <boolean value expression>, or <row value expression>.

2) 09 14 The declared type of a <common value expression> is the declared type of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <user-defined type value expression>, <collection value expression>, <JSON value expression>, or <reference value expression>, respectively.

3) The declared type of a <user-defined type value expression> is the declared type of the immediately contained <value expression primary>, which shall be a user-defined type.

4) The declared type of a <reference value expression> is the declared type of the immediately contained <value expression primary>, which shall be a reference type.

5) 15 The declared type of a <collection value expression> is the declared type of the immediately contained <array value expression> or <multiset value expression>.

6) Let *C* be some column. Let *VE* be the <value expression>. *C* is an *underlying column* of *VE* if *C* is identified by some column reference contained in *VE*. *C* is a *generally underlying column* of *VE* if *C* is an underlying column of *VE* or *C* is a generally underlying column of an underlying column of *VE*.

Access Rules

None.

General Rules

1) The value of a <value expression> is the value of the simply contained <common value expression>, <boolean value expression>, or <row value expression>.

2) [14] The value of a <common value expression> is the value of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <interval value expression>, <user-defined type value expression>, <collection value expression>, <JSON value expression>, or <reference value expression>.

3) When a <value expression> *V* is evaluated for a row *R* of a table, each reference to a column of that table by a column reference *CR* directly contained in *V* is the value of that column in that row.

4) [15] The value of a <collection value expression> is the value of its immediately contained <array value expression> or <multiset value expression>.

5) The value of a <reference value expression> *RVE* is the value of the <value expression primary> immediately contained in *RVE*.

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <value expression> that is a <boolean value expression>.

2) [15] Without Feature S041, "Basic reference types", conforming SQL language shall not contain a <reference value expression>.

6.30 <numeric value expression>

Function

Specify a numeric value.

Format

```
<numeric value expression> ::=
    <term>
  | <numeric value expression> <plus sign> <term>
  | <numeric value expression> <minus sign> <term>

<term> ::=
    <factor>
  | <term> <asterisk> <factor>
  | <term> <solidus> <factor>

<factor> ::=
  [ <sign> ] <numeric primary>

<numeric primary> ::=
    <value expression primary>
  | <numeric value function>
```

Syntax Rules

1) Case:

   a) If the declared type of either operand of a dyadic arithmetic operator is the decimal floating-point type, then the declared type of the result is the decimal floating-point type with an implementation-defined (IV134) precision.

   b) If the declared type of either operand of a dyadic arithmetic operator is approximate numeric, then the declared type of the result is an implementation-defined (IV135) approximate numeric type.

   c) Otherwise, the declared type of both operands of a dyadic arithmetic operator is exact numeric and the declared type of the result is an implementation-defined (IV136) exact numeric type, with precision and scale determined as follows:

      i) Let *S1* and *S2* be the scale of the first and second operands respectively.

      ii) The precision of the result of addition and subtraction is implementation-defined (IV136), and the scale is the maximum of *S1* and *S2*.

      iii) The precision of the result of multiplication is implementation-defined (IV136), and the scale is *S1* + *S2*.

      iv) The precision and scale of the result of division are implementation-defined (IV136).

2) The declared type of a <factor> is that of the immediately contained <numeric primary>.

3) The declared type of a <numeric primary> shall be numeric.

4) If a <numeric value expression> immediately contains a <minus sign> *NMS* and immediately contains a <term> that is a <factor> that immediately contains a <sign> that is a <minus sign> *FMS*, then there shall be a <separator> between *NMS* and *FMS*.

Access Rules

None.

General Rules

1) If the value of any <numeric primary> simply contained in a <numeric value expression> is the null value, then the result of the <numeric value expression> is the null value.

2) If the <numeric value expression> contains only a <numeric primary>, then the result of the <numeric value expression> is the value of the specified <numeric primary>.

3) The monadic arithmetic operators <plus sign> and <minus sign> (+ and −, respectively) specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand.

4) The dyadic arithmetic operators <plus sign>, <minus sign>, <asterisk>, and <solidus> (+, −, *, and /, respectively) specify addition, subtraction, multiplication, and division, respectively. If the value of a divisor is zero, then an exception condition is raised: *data exception — division by zero (22012)*.

5) If the most specific type of the result of an arithmetic operation is exact numeric, then

   Case:

   a) If the operator is not division and the mathematical result of the operation is not exactly representable with the precision and scale of the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

   b) If the operator is division and the approximate mathematical result of the operation represented with the precision and scale of the declared type of the result loses one or more leading significant digits after rounding or truncating if necessary, then an exception condition is raised: *data exception — numeric value out of range (22003)*. The choice of whether to round or truncate is implementation-defined (IA002).

6) If the most specific type of the result of an arithmetic operation is approximate numeric and the exponent of the approximate mathematical result of the operation is not within the implementation-defined (IL054) exponent range for the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

7) If the most specific type *MST* of the result of an arithmetic operation is the decimal floating point type, then the result is an implementation-defined (IV086) value *IV* of *MST* such that no other value of *MST* is strictly between *IV* and the mathematical result *MR* of the operation. If the exponent of *IV* is not within the implementation-defined (IL054) exponent range for *MST*, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

Conformance Rules

None.

6.31 <numeric value function>

This Subclause is modified by Subclause 6.9, "<numeric value function>", in ISO/IEC 9075-15.

Function

Specify a function yielding a value of type numeric.

Format

```
15 <numeric value function> ::=
     <position expression>
   | <regex occurrences function>
   | <regex position expression>
   | <extract expression>
   | <length expression>
   | <cardinality expression>
   | <max cardinality expression>
   | <absolute value expression>
   | <modulus expression>
   | <trigonometric function>
   | <general logarithm function>
   | <common logarithm>
   | <natural logarithm>
   | <exponential function>
   | <power function>
   | <square root>
   | <floor function>
   | <ceiling function>
   | <width bucket function>
   | <match number function>

<position expression> ::=
     <character position expression>
   | <binary position expression>

<regex occurrences function> ::=
   OCCURRENCES_REGEX <left paren>
       <XQuery pattern> [ FLAG <XQuery option flag> ]
       IN <regex subject string>
       [ FROM <start position> ]
       [ USING <char length units> ]
       <right paren>

<XQuery pattern> ::=
   <character value expression>

<XQuery option flag> ::=
   <character value expression>

<regex subject string> ::=
   <character value expression>

<regex position expression> ::=
   POSITION_REGEX <left paren>
       [ <regex position start or after> ]
       <XQuery pattern> [ FLAG <XQuery option flag> ]
       IN <regex subject string>
       [ FROM <start position> ]
       [ USING <char length units> ]
```

```
      [ OCCURRENCE <regex occurrence> ]
      [ GROUP <regex capture group> ]
      <right paren>

<regex position start or after> ::=
    START
  | AFTER

<regex occurrence> ::=
  <numeric value expression>

<regex capture group> ::=
  <numeric value expression>

<character position expression> ::=
  POSITION <left paren> <character value expression 1> IN <character value expression 2>
      [ USING <char length units> ] <right paren>

<character value expression 1> ::=
  <character value expression>

<character value expression 2> ::=
  <character value expression>

<binary position expression> ::=
  POSITION <left paren> <binary value expression> IN <binary value expression> <right paren>

<length expression> ::=
    <char length expression>
  | <octet length expression>

<char length expression> ::=
  { CHAR_LENGTH | CHARACTER_LENGTH } <left paren> <character value expression>
      [ USING <char length units> ] <right paren>

<octet length expression> ::=
  OCTET_LENGTH <left paren> <string value expression> <right paren>

<extract expression> ::=
  EXTRACT <left paren> <extract field> FROM <extract source> <right paren>

<extract field> ::=
    <primary datetime field>
  | <time zone field>

<time zone field> ::=
    TIMEZONE_HOUR
  | TIMEZONE_MINUTE

<extract source> ::=
    <datetime value expression>
  | <interval value expression>

<cardinality expression> ::=
  CARDINALITY <left paren> <collection value expression> <right paren>

<max cardinality expression> ::=
  ARRAY_MAX_CARDINALITY <left paren> <array value expression> <right paren>

<absolute value expression> ::=
  ABS <left paren> <numeric value expression> <right paren>

<modulus expression> ::=
  MOD <left paren> <numeric value expression dividend> <comma>
      <numeric value expression divisor> <right paren>
```

6.31 <numeric value function>

```
<numeric value expression dividend> ::=
  <numeric value expression>

<numeric value expression divisor> ::=
  <numeric value expression>

<trigonometric function> ::=
  <trigonometric function name> <left paren> <numeric value expression> <right paren>

<trigonometric function name> ::=
    SIN | COS | TAN | SINH | COSH | TANH | ASIN | ACOS | ATAN

<general logarithm function> ::=
  LOG <left paren> <general logarithm base> <comma>
      <general logarithm argument> <right paren>

<general logarithm base> ::=
  <numeric value expression>

<general logarithm argument> ::=
  <numeric value expression>

<common logarithm> ::=
  LOG10 <left paren> <numeric value expression> <right paren>

<natural logarithm> ::=
  LN <left paren> <numeric value expression> <right paren>

<exponential function> ::=
  EXP <left paren> <numeric value expression> <right paren>

<power function> ::=
  POWER <left paren> <numeric value expression base> <comma>
      <numeric value expression exponent> <right paren>

<numeric value expression base> ::=
  <numeric value expression>

<numeric value expression exponent> ::=
  <numeric value expression>

<square root> ::=
  SQRT <left paren> <numeric value expression> <right paren>

<floor function> ::=
  FLOOR <left paren> <numeric value expression> <right paren>

<ceiling function> ::=
  { CEIL | CEILING } <left paren> <numeric value expression> <right paren>

<width bucket function> ::=
  WIDTH_BUCKET <left paren> <width bucket operand> <comma> <width bucket bound 1> <comma>
      <width bucket bound 2> <comma> <width bucket count> <right paren>

<width bucket operand> ::=
  <numeric value expression>

<width bucket bound 1> ::=
  <numeric value expression>

<width bucket bound 2> ::=
  <numeric value expression>

<width bucket count> ::=
  <numeric value expression>
```

```
<match number function> ::=
  MATCH_NUMBER <left paren> <right paren>
```

Syntax Rules

1) If <position expression>, <regex occurrences function>, or <regex position expression> is specified, then the declared type of the result is an implementation-defined (IV137) exact numeric type with scale 0 (zero).

2) If <character position expression> is specified, then <character value expression 1> and <character value expression 2> shall be comparable.

3) If <regex occurrences function> is specified, then:

   a) <XQuery pattern> and <regex subject string> shall be comparable.

   b) The declared type of <start position> shall be exact numeric with scale 0 (zero).

   c) If <start position> is not specified, then 1 (one) is implicit.

   d) Case:

      i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

      ii) Otherwise, CHARACTERS is implicit.

   e) If <XQuery option flag> is not specified, then the zero-length character string is implicit.

4) If <regex position expression> is specified, then:

   a) <XQuery pattern> and <regex subject string> shall be comparable.

   b) If <regex position start or after> is not specified, then START is implicit.

   c) The declared type of <start position>, <regex occurrence>, and <regex capture group> shall be exact numeric with scale 0 (zero).

   d) If <start position> is not specified, then 1 (one) is implicit.

   e) Case:

      i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

      ii) Otherwise, CHARACTERS is implicit.

   f) If <regex occurrence> is not specified, then 1 (one) is implicit.

   g) If <regex capture group> is not specified, then 0 (zero) is implicit.

   h) If <XQuery option flag> is not specified, then the zero-length character string is implicit.

5) Case:

   a) If the character encoding form of <string value expression> is not UTF8, UTF16, or UTF32, then <char length units> shall not be specified.

   b) Otherwise, if <char length units> is not specified, then CHARACTERS is implicit.

6) If <extract expression> is specified, then

   Case:

a) If <extract field> is a <primary datetime field>, then it shall identify a <primary datetime field> of the <interval value expression> or <datetime value expression> immediately contained in <extract source>.

b) If <extract field> is a <time zone field>, then the declared type of the <extract source> shall be TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE.

7) If <extract expression> is specified, then

Case:

a) If <extract field> is a <primary datetime field> that does not specify SECOND or <extract field> is not a <primary datetime field>, then the declared type of the result is an implementation-defined (IV138) exact numeric type with scale 0 (zero).

b) Otherwise, the declared type of the result is an implementation-defined (IV138) exact numeric type with scale not less than the specified or implied <time fractional seconds precision> or <interval fractional seconds precision>, as appropriate, of the SECOND <primary datetime field> of the <extract source>.

8) If a <length expression> is specified, then the declared type of the result is an implementation-defined (IV139) exact numeric type with scale 0 (zero).

9) If <cardinality expression> is specified, then the declared type of the result is an implementation-defined (IV152) exact numeric type with scale 0 (zero).

10) If <max cardinality expression> is specified, then the declared type of the result is an implementation-defined (IV153) exact numeric type with scale 0 (zero).

11) If <absolute value expression> is specified, then the declared type of the result is the declared type of the immediately contained <numeric value expression>.

12) If <modulus expression> is specified, then the declared type of each <numeric value expression> shall be either exact numeric with scale 0 (zero) or the decimal floating-point type. The declared type of the result is

Case:

a) If the declared type of either <numeric value expression> is the decimal floating-point type, then the decimal floating-point type with implementation-defined (IV140) precision.

b) Otherwise, the declared type of the immediately contained <numeric value expression divisor>.

13) If <trigonometric function> is specified, then

Case:

a) If the declared type of <numeric value expression> is the decimal floating-point type, then the declared type of the result is decimal floating-point type with implementation-defined (IV143) precision.

b) Otherwise, the declared type of the result is an implementation-defined (IV144) approximate numeric type.

14) The declared type of the result of <general logarithm function> is

Case:

a) If the declared type of either <numeric value expression> is the decimal floating-point type, then the decimal floating-point type with implementation-defined (IV141) precision.

b) Otherwise, an implementation-defined (IV142) approximate numeric type.

15) If <common logarithm> is specified, then let *NVE* be the simply contained <numeric value expression>. The <common logarithm> is equivalent to

```
LOG (10, NVE)
```

16) The declared type of the result of <natural logarithm> is

Case:

a) If the declared type of <numeric value expression> is the decimal floating-point type, then the decimal floating-point type with implementation-defined (IV145) precision.

b) Otherwise, an implementation-defined (IV146) approximate numeric type.

17) The declared type of the result of <exponential function> is

Case:

a) If the declared type of <numeric value expression> is the decimal floating-point type, then the decimal floating-point type with implementation-defined (IV147) precision.

b) Otherwise, an implementation-defined (IV148) approximate numeric type.

18) The declared type of the result of <power function> is

Case:

a) If the declared type of either <numeric value expression> is the decimal floating-point type, then the decimal floating-point type with implementation-defined (IV149) precision.

b) Otherwise, an implementation-defined (IV150) approximate numeric type.

19) If <square root> is specified, then let *NVE* be the simply contained <numeric value expression>. The <square root> is equivalent to

```
POWER (NVE, 0.5)
```

20) If <floor function> or <ceiling function> is specified, then

Case:

a) If the declared type of the simply contained <numeric value expression> *NVE* is exact numeric, then the declared type of the result is exact numeric with implementation-defined (IV151) precision, with the radix of *NVE*, and with scale 0 (zero).

b) If the declared type of the simply contained <numeric value expression> *NVE* is approximate numeric, then the declared type of the result is approximate numeric with implementation-defined (IV151) precision.

c) Otherwise, the declared type of the result is the decimal floating-point with implementation-defined (IV151) precision.

21) If <width bucket function> is specified, then the declared type of <width bucket count> shall be exact numeric with scale 0 (zero). The declared type of the result of <width bucket function> is the declared type of <width bucket count>.

22) [15]If <match number function> *MNF* is specified, then:

a) *MNF* shall be contained in a <row pattern recognition clause>.

b) The declared type of the result of *MNF* is exact numeric with implementation-defined (IV154) precision and scale 0 (zero).

Access Rules

None.

General Rules

1) Case:

   a) If <max cardinality expression> *MCE* is specified, then the result of *MCE* is the maximum cardinality of the declared type of the <array value expression> simply contained in *MCE*.

   b) Otherwise, if the value of one or more <string value expression>s, <datetime value expression>s, <interval value expression>s, and <collection value expression>s that are simply contained in a <numeric value function> is the null value, then the result of the <numeric value function> is the null value and no further General Rules of this Subclause are applied.

2) If <character position expression> is specified, then let *CVE1* be the value of <character value expression 1> and let *CVE2* be the value of <character value expression 2>.

   Case:

   a) If CHAR_LENGTH(*CVE1*) is 0 (zero), then the result is 1 (one).

   b) If <char length units> is specified, then let *CLU* be <char length units>; otherwise, let *CLU* be CHARACTERS. If there is at least one value *P* such that

      ```
      CVE1 = SUBSTRING ( CVE2 FROM P FOR CHAR_LENGTH (CVE1 USING CLU ) USING CLU )
      ```

      then the result is the least such *P*.

      NOTE 183 — The collation used is determined in the normal way.

   c) Otherwise, the result is 0 (zero).

3) If <binary position expression> is specified, then

   Case:

   a) If the first <binary value expression> has a length of 0 (zero), then the result is 1 (one).

   b) If the value of the first <binary value expression> is equal to an identical-length substring of contiguous octets from the value of the second <binary value expression>, then the result is 1 (one) greater than the number of octets within the value of the second <binary value expression> preceding the start of the first such substring.

   c) Otherwise, the result is 0 (zero).

4) If <extract expression> is specified, then

   Case:

   a) If <extract field> is a <primary datetime field>, then the result is the value of the datetime field identified by that <primary datetime field> and has the same sign as the <extract source>.

      NOTE 184 — If the value of the identified <primary datetime field> is zero or if <extract source> is not an <interval value expression>, then the sign is irrelevant.

   b) Otherwise, let *TZ* be the interval value of the implicit or explicit time zone displacement associated with the <datetime value expression>.

      Case:

i) If <extract field> is TIMEZONE_HOUR, then the result is calculated as `EXTRACT (HOUR FROM TZ)`.

ii) Otherwise, the result is calculated as `EXTRACT (MINUTE FROM TZ)`

5) If a <char length expression> is specified, then

Case:

a) If the character encoding form of <character value expression> is not UTF8, UTF16, or UTF32, then let *S* be the <string value expression>.

Case:

i) If the most specific type of *S* is character string, then the result is the number of characters in the value of *S*.

NOTE 185 — The number of characters in a character string is determined according to the semantics of the character set of that character string.

ii) Otherwise, the result is OCTET_LENGTH(*S*).

b) Otherwise, the result is the number of explicit or implicit <char length units> in <char length expression>, counted in accordance with the definition of those units in the relevant normatively referenced document.

6) If an <octet length expression> is specified, then let *S* be the <string value expression>. Let *BL* be the number of bits (binary digits) in the value of *S*. The result of the <octet length expression> is the smallest integer not less than the quotient of the division (*BL*/8).

7) The result of <cardinality expression> is the number of elements of the result of the <collection value expression>.

8) If <absolute value expression> is specified, then let *N* be the value of the immediately contained <numeric value expression>.

Case:

a) If *N* is the null value, then the result is the null value.

b) If $N \geq 0$, then the result is *N*.

c) Otherwise, the result is $-1 * N$. If $-1 * N$ is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

9) If <modulus expression> is specified, then let *N* be the value of the immediately contained <numeric value expression dividend> and let *M* be the value of the immediately contained <numeric value expression divisor>.

Case:

a) If at least one of *N* and *M* is the null value, then the result is the null value.

b) If *M* is zero, then an exception condition is raised: *data exception — division by zero (22012)*.

c) Otherwise, the result is the unique exact numeric value *R* with scale 0 (zero) such that all of the following are true:

i) *R* has the same sign as *N*.

ii) The absolute value of *R* is less than the absolute value of *M*.

iii) $N = M * K + R$ for some exact numeric value *K* with scale 0 (zero).

10) If <trigonometric function> is specified, then let *V* be the value of the simply contained <numeric value expression>, which represents an angle expressed in radians.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise, let *OP* be the <trigonometric function name>.

Case:

i) If *OP* is ACOS, then

Case:

1) If *V* is less than −1 (negative one) or *V* is greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.

2) Otherwise, the result of the <trigonometric function> is the inverse cosine of *V*.

ii) If *OP* is ASIN, then

Case:

1) If *V* is less than −1 (negative one) or *V* is greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.

2) Otherwise, the result of the <trigonometric function> is the inverse sine of *V*.

iii) If *OP* is ATAN, then the result is the inverse tangent of *V*.

iv) If *OP* is COS, then the result is the cosine of *V*.

v) If *OP* is COSH, then the result is the hyperbolic cosine of *V*.

vi) If *OP* is SIN, then the result is the sine of *V*.

vii) If *OP* is SINH, then the result is the hyperbolic sine of *V*.

viii) If *OP* is TAN, then the result is the tangent of *V*.

ix) If *OP* is TANH, then the result is the hyperbolic tangent of *V*.

11) If <general logarithm function> is specified, then let *VB* be the value of the <general logarithm base> and let *VA* be the value of the <general logarithm argument>.

Case:

a) If at least one of *VA* and *VB* is the null value, then the result is the null value.

b) If *VA* is negative or 0 (zero), then an exception condition is raised: *data exception — numeric value out of range (22003)*.

c) If *VB* is negative, 0 (zero), or 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.

d) Otherwise, the result is the logarithm with base *VB* of *VA*.

12) If <natural logarithm> is specified, then let *V* be the value of the simply contained <numeric value expression>.

Case:

a) If *V* is the null value, then the result is the null value.

b) If *V* is 0 (zero) or negative, then an exception condition is raised: *data exception — invalid argument for natural logarithm (2201E)*.

c) Otherwise, the result is the natural logarithm of *V*.

13) If <exponential function> is specified, then let *V* be the value of the simply contained <numeric value expression>.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise, the result is *e* (the base of natural logarithms) raised to the power *V*. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

14) If <power function> is specified, then let *NVEB* be the <numeric value expression base>, then let *VB* be the value of *NVEB*, let *NVEE* be the <numeric value expression exponent>, and let *VE* be the value of *NVEE*.

Case:

a) If at least one of *VB* and *VE* is the null value, then the result is the null value.

b) If *VB* is 0 (zero) and *VE* is negative, then an exception condition is raised: *data exception — invalid argument for power function (2201F)*.

c) If *VB* is 0 (zero) and *VE* is 0 (zero), then the result is 1 (one).

d) If *VB* is 0 (zero) and *VE* is positive, then the result is 0 (zero).

e) If *VB* is negative and *VE* is not equal to an exact numeric value with scale 0 (zero), then an exception condition is raised: *data exception — invalid argument for power function (2201F)*.

f) If *VB* is negative and *VE* is equal to an exact numeric value with scale 0 (zero) that is an even number, then the result is the result of

```
EXP(NVEE*LN(-NVEB))
```

g) If *VB* is negative and *VE* is equal to an exact numeric value with scale 0 (zero) that is an odd number, then the result is the result of

```
-EXP(NVEE*LN(-NVEB))
```

h) Otherwise, the result is the result of

```
EXP(NVEE*LN(NVEB))
```

15) If <floor function> is specified, then let *V* be the value of the simply contained <numeric value expression> *NVE*.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise,

Case:

i) If the most specific type of *NVE* is exact numeric, then the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

ii) Otherwise, the result is the greatest whole number that is less than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

16) If <ceiling function> is specified, then let *V* be the value of the simply contained <numeric value expression> *NVE*.

Case:

a) If *V* is the null value, then the result is the null value.

b) Otherwise,

Case:

i) If the most specific type of *NVE* is exact numeric, then the result is the least exact numeric value with scale 0 (zero) that is greater than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

ii) Otherwise, the result is the least whole number that is greater than or equal to *V*. If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

17) If <width bucket function> is specified, then let *WBO* be the value of <width bucket operand>, let *WBB1* be the value of <width bucket bound 1>, let *WBB2* be the value of <width bucket bound 2>, and let *WBC* be the value of <width bucket count>.

Case:

a) If at least one of *WBO*, *WBB1*, *WBB2*, and *WBC* is the null value, then the result is the null value.

b) If *WBC* is less than or equal to 0 (zero), then an exception condition is raised: *data exception — invalid argument for width bucket function (2201G)*.

c) If *WBB1* equals *WBB2*, then an exception condition is raised: *data exception — invalid argument for width bucket function (2201G)*.

d) If *WBB1* is less than *WBB2*, then

Case:

i) If *WBO* is less than *WBB1*, then the result is 0 (zero).

ii) If *WBO* is greater than or equal to *WBB2*, then the result is *WBC*+1. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

iii) Otherwise, the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to `((WBC * (WBO - WBB1) / (WBB2 - WBB1)) + 1)`

e) If *WBB1* is greater than *WBB2*, then

Case:

i) If *WBO* is greater than *WBB1*, then the result is 0 (zero).

ii) If *WBO* is less than or equal to *WBB2*, then the result is *WBC*+1. If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

iii) Otherwise, the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to `((WBC * (WBB1 - WBO) / (WBB1 - WBB2)) + 1)`

18) If <regex occurrences function> is specified, then:

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If at least one of *STR*, *PAT*, and *FL* is the null value, then the result of the <regex occurrences function> is the null value.

ii) If *SP* is less than 1 (one), or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex occurrences function> is −1 (negative one).

iii) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex occurrences function> is implementation-dependent (UV074).

iv) Otherwise, the General Rules of Subclause 9.28, "XQuery regular expression matching", are applied with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, and *FL* as *FLAG*; let *LOMV* be the *LIST* returned from the application of those General Rules. The result of <regex occurrences function> is the number of match vectors in *LOMV*.

19) If <regex position expression> is specified, then:

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, let *OCC* be the value of <regex occurrence>, let *CAP* be the <regex capture group> and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If at least one of *STR*, *PAT*, *OCC*, *CAP*, and *FL* is the null value, then the result of the <regex position expression> is the null value.

ii) If *OCC* is less than 1 (one), then the result of <regex position expression> is 0 (zero).

iii) If *CAP* is less than 0 (zero), or greater than the number of XQuery regular expression parenthesized subexpressions of *PAT*, then the result of <regex position expression> is 0 (zero).

iv) If *SP* is less than 1 (one), or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex position expression> is 0 (zero).

v) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex position expression> is implementation-dependent (UV074).

vi) Otherwise, the General Rules of Subclause 9.28, "XQuery regular expression matching", are applied with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, and *FL* as *FLAG*; let *LOMV* be the *LIST* returned from the application of those General Rules.

Case:

1) If there are at least *OCC* match vectors in *LOMV*, then let *MV* be the *OCC*-th match vector in *LOMV*. Let *PL* be *MV*[*CAP*], where *MV*[0] is the first position/length in *MV*. Let *P* be the position of *PL*, and let *L* be the length of *PL*. The result of the <regex position expression> is

Case:

A) If *P* is 0 (zero), then 0 (zero).

B) If <regex position start or after> is START, then *P*.

C) Otherwise, *P* + *L*.

2) Otherwise, the result of the <regex position expression> is 0 (zero).

20) [15]If <match number function> *MNF* is specified, then the result is 1 (one) plus the number of retained row pattern matches whose starting row is prior to the current row in the ordering of the row pattern partition containing the current row.

NOTE 186 — When <match number function> is used in a <row pattern definition search condition>, it is not yet known whether the current potential row pattern match will be accepted as a retained row pattern match; however, it is known how many row pattern matches have already been retained earlier in the row pattern partition. Adding one to the latter number yields the sequential number of the row pattern match, if it is ultimately retained.

Conformance Rules

1) Without Feature R010, "Row pattern recognition: FROM clause", conforming SQL language shall not contain <match number function>.

2) Without at least one of Feature S090, "Minimal array support", or Feature S271, "Basic multiset support", conforming SQL language shall not contain a <cardinality expression>.

3) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <extract expression>.

4) Feature F411, "Time zone specification", conforming SQL language shall not contain an <extract expression> that specifies a <time zone field>.

5) Without Feature F421, "National character", conforming SQL language shall not contain a <length expression> that simply contains a <string value expression> that has a declared type of NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, or NATIONAL CHARACTER LARGE OBJECT.

6) Without at least one of Feature T047, "POSITION, OCTET_LENGTH, TRIM, and SUBSTRING for BLOBs" or Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary position expression>.

7) Without Feature T050, "POSITION, CHAR_LENGTH, OCTET_LENGTH, LOWER, TRIM, UPPER, and SUBSTRING for CLOBs", conforming SQL language shall not contain a <character position expression>, a <char length expression>, or an <octet length expression> that contains a <character value expression> whose declared type is character large object type.

8) Without Feature T047, "POSITION, OCTET_LENGTH, TRIM, and SUBSTRING for BLOBs", conforming SQL language shall not contain an <octet length expression> that contains a <string value expression> whose declared type is a character large object type or a binary large object type.

9) Without Feature T441, "ABS and MOD functions", conforming SQL language shall not contain an <absolute value expression>.

10) Without Feature T441, "ABS and MOD functions", conforming SQL language shall not contain a <modulus expression>.

11) Without Feature T622, "Trigonometric functions", conforming SQL language shall not contain a <trigonometric function>.

12) Without Feature T623, "General logarithm functions", conforming SQL language shall not contain a <general logarithm function>.

13) Without Feature T624, "Common logarithm functions", conforming SQL language shall not contain a <common logarithm>.

14) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <natural logarithm>.

15) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain an <exponential function>.

16) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <power function>.

17) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <square root>.

18) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <floor function>.

19) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <ceiling function>.

20) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <width bucket function>.

21) Without Feature F842, "OCCURRENCES_REGEX function", conforming SQL language shall not contain <regex occurrences function>.

22) Without Feature F846, "Octet support in regular expression operators", in conforming SQL language, <regex occurrences function> shall not contain <char length units>.

23) Without Feature F843, "POSITION_REGEX function", conforming SQL language shall not contain <regex position expression>.

24) Without Feature F846, "Octet support in regular expression operators", in conforming SQL language, <regex position expression> shall not contain <char length units>.

25) Without Feature F847, "Non-constant regular expression", in conforming SQL language, <XQuery pattern> and <XQuery option flag> shall be <value specification>s.

26) [15] Without Feature S403, "ARRAY_MAX_CARDINALITY", conforming SQL language shall not contain <max cardinality expression>.

6.32 <string value expression>

Function

Specify a character string value or a binary string value.

Format

```
<string value expression> ::=
    <character value expression>
  | <binary value expression>

<character value expression> ::=
    <concatenation>
  | <character factor>

<concatenation> ::=
  <character value expression> <concatenation operator> <character factor>

<character factor> ::=
  <character primary> [ <collate clause> ]

<character primary> ::=
    <value expression primary>
  | <string value function>

<binary value expression> ::=
    <binary concatenation>
  | <binary factor>

<binary factor> ::=
  <binary primary>

<binary primary> ::=
    <value expression primary>
  | <string value function>

<binary concatenation> ::=
  <binary value expression> <concatenation operator> <binary factor>
```

Syntax Rules

1) The declared type of a <character primary> shall be character string.

2) The declared type of a <character value expression> is the declared type of the simply contained <concatenation> or <character factor>.

3) If <concatenation> is specified, then:

   a) Let *D1* be the declared type of the <character value expression> and let *D2* be the declared type of the <character factor> simply contained in the <concatenation>. The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *D1* and *D2* as *DTSET*; let *D* be the *RESTYPE* returned from the application of those Syntax Rules. Let *CS* be the character set of *D*, let *CO* be the collation of *D*, and let *CD* be the collation derivation of *D*.

   b) Let *M* be the length in characters of *D1* plus the length in characters of *D2*. Let *VL* be the implementation-defined (IL006) maximum length of variable-length character strings, let *LOL* be the implementation-defined (IL002) maximum length of large object character strings,

and let *FL* be the implementation-defined (IL001) maximum length of fixed-length character strings.

Case:

i) If the declared type of the <character value expression> or <character factor> is a character large object type, then the declared type of the <concatenation> is a character large object type with character set *CS*, collation *CO*, collation derivation *CD* and maximum length equal to the lesser of *M* and *LOL*.

ii) If the declared type of the <character value expression> or <character factor> is variable-length character string, then the declared type of the <concatenation> is variable-length character string with character set *CS*, collation *CO*, collation derivation *CD* and maximum length equal to the lesser of *M* and *VL*.

iii) If the declared type of the <character value expression> and <character factor> is fixed-length character string, then *M* shall not be greater than *FL* and the declared type of the <concatenation> is fixed-length character string with character set *CS*, collation *CO*, collation derivation *CD* and length *M*.

4) If <character factor> is specified, then

Case:

a) If <collate clause> is specified, then the declared type of the <character factor> is the declared type of the <character primary>, except that the declared type collation is the collation identified by <collate clause>, and its collation derivation is *explicit*.

b) Otherwise, the declared type of the <character factor> is the declared type of the <character primary>.

5) The declared type of <binary primary> shall be binary string.

6) Case:

a) If <binary concatenation> is specified, then let *B1* be the declared type of the <binary value expression> and let *B2* be the declared type of the <binary factor>. Let *M* be the length in octets of *B1* plus the length in octets of *B2*. Let *FL* be the implementation-defined (IL003) maximum length of fixed-length binary strings, let *VL* be the implementation-defined (IL007) maximum length of variable-length binary strings, let *LOL* be the implementation-defined (IL004) maximum length of binary large object strings.

The declared type of <binary concatenation> is

Case:

i) If the declared type of the <binary value expression> or <binary factor> is a binary large object type, then binary large object string with maximum length equal to the lesser of *M* and *LOL*.

ii) If the declared type of the <binary value expression> or <binary factor> is variable-length binary string, then variable-length binary string with maximum length equal to the lesser of *M* and *VL*.

iii) If the declared type of the <binary value expression> and <binary factor> is fixed-length binary string, then fixed-length binary string with length *M*, and *M* shall not be greater than *FL*.

b) Otherwise, the declared type of the <binary value expression> is the declared type of the <binary factor>.

Access Rules

None.

General Rules

1) If the value of any <character primary> or <binary primary> simply contained in a <string value expression> is the null value, then the result of the <string value expression> is the null value.

2) If <concatenation> is specified, then:

   a) If the character repertoire of <character factor> is UCS, then, in the remainder of this General Rule, the term "length" shall be taken to mean "length in characters".

   b) Let *S1* and *S2* be the result of the <character value expression> and <character factor>, respectively.

      Case:

      i) If at least one of *S1* and *S2* is the null value, then the result of the <concatenation> is the null value.

      ii) Otherwise:

         1) Let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length of *S*.

         2) If the character repertoire of <character factor> is UCS, then *S* is replaced by

            Case:

            A) If the <search condition> `S1 IS NORMALIZED AND S2 IS NORMALIZED` evaluates to *True*, then

               `NORMALIZE (S)`

            B) Otherwise, an implementation-defined (IV087) string.

         3) Case:

            A) If the most specific type of at least one of *S1* and *S2* is a character large object type, then let *LOL* be the implementation-defined (IL002) maximum length of large object character strings.

               Case:

               I) If *M* is less than or equal to *LOL*, then the result of the <concatenation> is *S* with length *M*.

               II) If *M* is greater than *LOL* and the right-most *M*–*LOL* characters of *S* are all <truncating whitespace> characters, then the result of the <concatenation> is the first *LOL* characters of *S* with length *LOL*.

               III) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

            B) If the most specific type of at least one of *S1* and *S2* is variable-length character string, then let *VL* be the implementation-defined (IL006) maximum length of variable-length character strings.

               Case:

I) If *M* is less than or equal to *VL*, then the result of the <concatenation> is *S* with length *M*.

II) If *M* is greater than *VL* and the right-most *M*−*VL* characters of *S* are all <truncating whitespace> characters, then the result of the <concatenation> is the first *VL* characters of *S* with length *VL*.

III) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

C) If the most specific types of both *S1* and *S2* are fixed-length character string, then the result of the <concatenation> is *S*.

3) If <binary concatenation> is specified, then let *S1* and *S2* be the result of the <binary value expression> and <binary factor>, respectively.

Case:

a) If at least one of *S1* and *S2* is the null value, then the result of the <binary concatenation> is the null value.

b) Otherwise, let *S* be the string consisting of *S1* followed by *S2* and let *M* be the length in octets of *S*.

Case:

i) If the most specific type of at least one of *S1* and *S2* is a binary large object type, then let *LOL* be the implementation-defined (IL004) maximum length of binary large object strings.

Case:

1) If *M* is less or equal to *LOL*, then the result of the <binary concatenation> is *S* with length *M*.

2) If *M* is greater than *LOL* and the right-most *M*−*LOL* octets of *S* are all X'00', then the result of the <binary concatenation> is the first *LOL* octets of *S* with length *LOL*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

ii) If the most specific type of at least one of *S1* and *S2* is variable-length binary string, then let *VL* be the implementation-defined (IL004) maximum length of variable-length binary strings.

Case:

1) If *M* is less than or equal to *VL*, then the result of the <binary concatenation> is *S* with length *M*.

2) If *M* is greater than *VL* and the right-most *M*−*VL* octets of *S* are all X'00', then the result of the <binary concatenation> is the first *VL* characters of *S* with length *VL*.

3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

iii) If the most specific types of both *S1* and *S2* are fixed-length binary string, then the result of the <binary concatenation> is *S*.

4) If the result of <string value expression> is the zero-length character string or the zero-length binary string, then it is implementation-defined (IA214) whether an exception condition is raised: *data*

exception — zero-length character string (2200F) or *data exception — zero-length binary string (2201Y)*, respectively.

Conformance Rules

1) Without Feature F421, "National character", conforming SQL language shall not contain a <character value expression> that has a declared type of NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, or NATIONAL CHARACTER LARGE OBJECT.

2) Without at least one of Feature T045, "BLOB data type" or Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary value expression>.

3) Without at least one of Feature T048, "Concatenation of BLOBs", or Feature T021, "BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary concatenation>.

4) Without Feature T040, "Concatenation of CLOBs", conforming SQL language shall not contain a <concatenation> whose declared type is a character large object type.

6.33 <string value function>

This Subclause is modified by Subclause 6.4, "<string value function>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 6.9, "<string value function>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 6.10, "<string value function>", in ISO/IEC 9075-15.

Function

Specify a function yielding a value of type character string or binary string.

Format

```
15 <string value function> ::=
     <character value function>
   | <binary value function>
   | <JSON serialize>

09 | 14 | 15 <character value function> ::=
     <character substring function>
   | <regular expression substring function>
   | <regex substring function>
   | <fold>
   | <transcoding>
   | <character transliteration>
   | <regex transliteration>
   | <pad function>
   | <trim function>
   | <character overlay function>
   | <normalize function>
   | <specific type method>
   | <classifier function>

<character substring function> ::=
  SUBSTRING <left paren> <character value expression> FROM <start position>
      [ FOR <string length> ] [ USING <char length units> ] <right paren>

<regular expression substring function> ::=
  SUBSTRING <left paren> <character value expression> SIMILAR <character value expression>
      ESCAPE <escape character> <right paren>

<regex substring function> ::=
  SUBSTRING_REGEX <left paren>
      <XQuery pattern> [ FLAG <XQuery option flag> ]
      IN <regex subject string>
      [ FROM <start position> ]
      [ USING <char length units> ]
      [ OCCURRENCE <regex occurrence> ]
      [ GROUP <regex capture group> ]
      <right paren>

<fold> ::=
  { UPPER | LOWER } <left paren> <character value expression> <right paren>

<transcoding> ::=
  CONVERT <left paren> <character value expression>
      USING <transcoding name> <right paren>

<character transliteration> ::=
  TRANSLATE <left paren> <character value expression>
      USING <transliteration name> <right paren>
```

```
<regex transliteration> ::=
  TRANSLATE_REGEX <left paren>
      <XQuery pattern> [ FLAG <XQuery option flag> ]
      IN <regex subject string>
      [ WITH <XQuery replacement string> ]
      [ FROM <start position> ]
      [ USING <char length units> ]
      [ OCCURRENCE <regex transliteration occurrence> ]
      <right paren>

<XQuery replacement string> ::=
  <character value expression>

<regex transliteration occurrence> ::=
    <regex occurrence>
  | ALL

<trim function> ::=
    <single-character trim function>
  | <multi-character trim function>

<single-character trim function> ::=
  TRIM <left paren> <trim operands> <right paren>

<multi-character trim function> ::=
  { BTRIM | LTRIM | RTRIM }
      <left paren> <trim source> [ <comma> <trim character> ] <right paren>

<trim operands> ::=
  [ [ <trim specification> ] [ <trim character> ] FROM ] <trim source>

<trim source> ::=
  <character value expression>

<trim specification> ::=
    LEADING
  | TRAILING
  | BOTH

<trim character> ::=
  <character value expression>

<character overlay function> ::=
  OVERLAY <left paren> <character value expression> PLACING <character value expression>
      FROM <start position> [ FOR <string length> ]
      [ USING <char length units> ] <right paren>

<pad function> ::=
  { LPAD | RPAD } <left paren> <pad operands> <right paren>

<pad operands> ::=
  <pad source> <comma> <total pad length>
  [ <comma> <padding character value expression> ]

<pad source> ::=
  <character value expression>

<total pad length> ::=
  <numeric value expression>

<padding character value expression> ::=
  <character value expression>

<normalize function> ::=
  NORMALIZE <left paren> <character value expression>
```

```
      [ <comma> <normal form> [ <comma> <normalize function result length> ] ] <right paren>

<normal form> ::=
    NFC
  | NFD
  | NFKC
  | NFKD

<normalize function result length> ::=
    <character length>
  | <character large object length>

<specific type method> ::=
  <user-defined type value expression> <period> SPECIFICTYPE
      [ <left paren> <right paren> ]

14 <binary value function> ::=
    <binary substring function>
  | <binary trim function>
  | <binary overlay function>

<binary substring function> ::=
  SUBSTRING <left paren> <binary value expression> FROM <start position>
      [ FOR <string length> ] <right paren>

<binary trim function> ::=
  TRIM <left paren> <binary trim operands> <right paren>

<binary trim operands> ::=
  [ [ <trim specification> ] [ <trim octet> ] FROM ] <binary trim source>

<binary trim source> ::=
  <binary value expression>

<trim octet> ::=
  <binary value expression>

<binary overlay function> ::=
  OVERLAY <left paren> <binary value expression> PLACING <binary value expression>
      FROM <start position> [ FOR <string length> ] <right paren>

<start position> ::=
  <numeric value expression>

<string length> ::=
  <numeric value expression>

<classifier function> ::=
  CLASSIFIER <left paren> [ <row pattern variable name> ] <right paren>
```

Syntax Rules

1) 15 The declared type of <string value function> is the declared type of the immediately contained <character value function>, or <binary value function>.

2) 09 14 15 The declared type of <character value function> is the declared type of the immediately contained <character substring function>, <regular expression substring function>, <regex substring function>, <fold>, <transcoding>, <character transliteration>, <regex transliteration>, <trim function>, <pad function>, <character overlay function>, <normalize function>, <specific type method>, or <classifier function>.

3) The declared type of a <start position>, <string length>, <regex occurrence>, or <regex capture group> shall be exact numeric with scale 0 (zero).

4) If <character substring function> *CSF* is specified, then let *DTCVE* be the declared type of the <character value expression> immediately contained in *CSF*. The maximum length, character set, and collation of the declared type *DTCSF* of *CSF* are determined as follows:

   a) Case:

      i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then *DTCSF* is a variable-length character string type with maximum length equal to the length or maximum length of *DTCVE*.

      ii) Otherwise, the *DTCSF* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

   b) The character set and collation of the <character substring function> are those of *DTCVE*.

5) If the character repertoire of <character value expression> is not UCS, then <char length units> shall not be specified.

6) If USING <char length units> is not specified, then USING CHARACTERS is implicit.

7) If <regular expression substring function> is specified, then:

   a) The declared types of the <escape character> and the <character value expression>s of the <regular expression substring function> shall be character string with the same character repertoire.

   b) Case:

      i) If the declared type of the first <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <regular expression substring function> is variable-length character string with maximum length equal to the length or maximum length of the first <character value expression>.

      ii) Otherwise, the declared type of the <regular expression substring function> is a character large object type with maximum length equal to the maximum length of the first <character value expression>.

   c) The character set and collation of the <regular expression substring function> are those of the first <character value expression>.

8) If <regex substring function> *RSF* is specified, then:

   a) If <start position> is not specified, then 1 (one) is implicit.

   b) Case:

      i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

      ii) Otherwise, CHARACTERS is implicit.

   c) If <regex occurrence> is not specified, then 1 (one) is implicit.

   d) If <regex capture group> is not specified, then 0 (zero) is implicit.

   e) If <XQuery option flag> is not specified, then the zero-length character string is implicit.

   f) Let *RSS* be the <regex subject string> immediately contained in *RSF*. Let *DTCVE* be the declared type of *RSS*. The declared type *DTRSF* of *RSF* is determined as follows:

i) Case:

1) If *DTCVE* is fixed-length character string or variable-length character string, then *DTRSF* is variable length character string with maximum length equal to the length or maximum length of *DTCVE*.

2) Otherwise, *DTRSF* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

ii) The character set and collation of *DTRSF* are those of *DTCVE*.

9) If <fold> is specified, then the declared type of the result of <fold> is that of the <character value expression>.

10) If <transcoding> is specified, then:

a) <transcoding> shall be simply contained in a <value expression> that is immediately contained in a <derived column> that is immediately contained in a <select sublist> or shall immediately contain either a <simple value specification> that is a <host parameter name> or a <value specification> that is a <host parameter specification>.

b) A <transcoding name> shall identify a transcoding.

c) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the result is variable-length character string with implementation-defined (IL046) maximum length.

ii) Otherwise, the declared type of the result is a character large object type with implementation-defined (IL046) maximum length.

d) The character set of the result is an implementation-defined (IV155) character set *CS* whose character repertoire is the same as the character repertoire of the <character value expression> and whose character encoding form is that determined by the transcoding identified by the <transcoding name>. The declared type collation of the result is the character set collation of *CS*.

11) If <character transliteration> is specified, then:

a) A <transliteration name> shall identify a character transliteration.

b) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then the declared type of the <character transliteration> is variable-length character string with implementation-defined (IL046) maximum length.

ii) Otherwise, the declared type of the <character transliteration> is a character large object type with implementation-defined (IL046) maximum length.

c) The declared type of the <character transliteration> has the character set *CS* that is the target character set of the transliteration. The declared type collation of the result is the character set collation of *CS*.

12) If <regex transliteration> *RT* is specified, then:

a) If <start position> is not specified, then 1 (one) is implicit.

b) Case:

i) If <char length units> is specified, then the character repertoire of the <regex subject string> shall be UCS.

ii) Otherwise, CHARACTERS is implicit.

c) If <regex transliteration occurrence> is not specified, then ALL is implicit.

d) If <XQuery replacement string> is not specified, then the zero-length character string is implicit.

e) If <XQuery option flag> is not specified, then the zero-length character string is implicit.

f) Let *RSS* be the <regex subject string> immediately contained in *RT*. Let *DTCVE* be the declared type of *RSS*. The declared type *DTRT* of *RT* is determined as follows:

i) Case:

1) If <XQuery replacement string> is not specified, or is the zero-length character string, then

Case:

A) If *DTCVE* is fixed-length character string or variable-length character string, then *DTRT* is variable length character string with maximum length equal to the length or maximum length of *DTCVE*.

B) Otherwise, *DTRT* is a large object character string type with maximum length equal to the maximum length of *DTCVE*.

2) Otherwise,

Case:

A) If *DTCVE* is fixed-length character string or variable-length character string, then *DTRT* is variable length character string with implementation-defined (IL047) maximum length.

B) Otherwise, *DTRT* is a large object character string type with implementation-defined (IL047) maximum length.

ii) The character set and collation of *DTRT* are those of *DTCVE*.

13) The declared type of <trim function> is the declared type of the immediately contained <single-character trim function> or <multi-character trim function>.

14) If <single-character trim function> is specified, then:

a) Case:

i) If FROM is specified, then:

1) Either <trim specification> or <trim character> or both shall be specified.

2) If <trim specification> is not specified, then BOTH is implicit.

3) If <trim character> is not specified, then <space> is implicit.

ii) Otherwise, let *SRC* be <trim source>. `TRIM ( SRC )` is equivalent to `TRIM ( BOTH ' ' FROM SRC )`.

b) Case:

i) If the declared type of <character value expression> simply contained in <trim source> is fixed-length character string or variable-length character string, then the declared

type of the <single-character trim function> is variable-length character string with maximum length equal to the length or maximum length of the <trim source>.

ii) Otherwise, the declared type of the <single-character trim function> is a character large object type with maximum length equal to the maximum length of the <trim source>.

c) If a <trim character> is specified, then <trim character> and <trim source> shall be comparable.

d) The character set and collation of the <single-character trim function> are those of the <trim source>.

15) If <multi-character trim function> *MTF* is specified, then:

a) Case:

i) If the declared type of <character value expression> simply contained in <trim source> *TS* is fixed-length character string or variable-length character string, then the declared type of *MTF* is variable-length character string with maximum length equal to the length or maximum length of *TS*.

ii) Otherwise, the declared type of *MTF* is a character large object type with maximum length equal to the maximum length of *TS*.

b) Case:

i) If a <trim character> *TC* is specified, then:

1) *TS* and *TC* shall be comparable.

2) If BTRIM is specified, then:

A) *TC* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

B) *TC* shall not generally contain a <table primary> that contains a <data change delta table>.

C) `BTRIM ( TS, TC )` is equivalent to `RTRIM ( LTRIM ( TS, TC ), TC )`

ii) Otherwise:

1) Let *TC* be <space>.

2) If BTRIM is specified, then `BTRIM ( TS )` is equivalent to `RTRIM ( LTRIM ( TS ) )`.

c) The character set and collation of *MTF* are those of *TS*.

16) If <pad function> is specified, then:

a) If <padding character value expression> is not specified, then <space> is implicit.

b) The declared type of <total pad length> shall be an exact numeric type with scale 0 (zero).

c) Case:

i) If the declared type of <character value expression> simply contained in <pad source> is fixed-length character string or variable-length character string, then the declared type of the <pad function> is variable-length character string with implementation-defined (IL046) maximum length *LR*.

ii) Otherwise, the declared type of the <pad function> is a character large object type with implementation-defined (IL046) maximum length *LR*.

d) <padding character value expression> and <pad source> shall be comparable.

e) The character set and collation of the <pad function> are those of the <pad source>.

17) If <character overlay function> is specified, then:

a) Let *CV* be the first <character value expression>, let *SP* be the <start position>, and let *RS* be the second <character value expression>.

b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be CHAR_LENGTH(*RS*).

c) The <character overlay function> is equivalent to:

```
   SUBSTRING ( CV FROM 1 FOR SP - 1 )
|| RS
|| SUBSTRING ( CV FROM SP + SL  )
```

18) If <normalize function> is specified, then:

a) The character set *CS* of the <character value expression> shall be UTF8, UTF16, or UTF32. The character set and the collation of the declared type of the <normalize function> are *CS* and the collation of the <character value expression>, respectively.

b) Case:

i) If <normal form> is specified, then let *NF* be <normal form>.

ii) Otherwise, let *NF* be NFC.

c) Case:

i) If the declared type of <character value expression> is fixed-length character string or variable-length character string, then:

1) <character large object length> shall not be specified.

2) If <normalize function result length> is specified, then let *L* be the value of the <length> contained in <normalize function result length>; otherwise, let *L* be an implementation-defined (ID097) value of exact numeric type with scale 0 (zero) that is less than or equal to the implementation-defined (IL006) maximum length of variable-length character strings.

3) The declared type of the <normalize function> is variable-length character string with maximum length equal to *L*, measured in the units of the explicit or implicit <char length units>.

ii) Otherwise:

1) <character length> shall not be specified.

2) If <normalize function result length> is specified, then let *L* be the value of the <character large object length> contained in <normalize function result length>; otherwise, let *L* be an implementation-defined (ID097) value of exact numeric type with scale 0 (zero) that is less than or equal to the implementation-defined (IL006) maximum length of large object character strings.

3) The declared type of the <normalize function> is character large object type with maximum length equal to *L*, measured in the units of the explicit or implicit <char length units>.

19) If <specific type method> is specified, then the declared type of the <specific type method> is variable-length character string with maximum length implementation-defined (IL046). The character set of the character string is SQL_IDENTIFIER.

20) 14 The declared type of <binary value function> is the declared type of the immediately contained <binary substring function>, <binary trim function>, or <binary overlay function>.

21) If <binary substring function> *BSF* is specified, then let *DTBVE* be the declared type of the <binary value expression> immediately contained in *BSF*. The declared type *DTBSF* of *BSF* and its maximum length are

Case:

a) If *DTBVE* is fixed-length binary string or variable-length binary string, then variable-length binary string type with maximum length equal to the length or maximum length of *DTBVE*.

b) Otherwise, binary large object string type with maximum length equal to the maximum length of *DTBVE*.

22) If <binary trim function> is specified, then:

a) Case:

i) If FROM is specified, then:

1) Either <trim specification> or <trim octet> or both shall be specified.

2) If <trim specification> is not specified, then BOTH is implicit.

3) If <trim octet> is not specified, then X'00' is implicit.

ii) Otherwise, let *SRC* be <binary trim source>. `TRIM ( SRC )` is equivalent to `TRIM ( BOTH X'00' FROM SRC )`.

b) Case:

i) If the declared type of <binary value expression> is fixed-length binary string or variable-length binary string, then the declared type of the <binary trim function> is variable-length binary string type with maximum length equal to the length or maximum length of the <binary trim source>.

ii) Otherwise, the declared type of the <binary trim function> is binary large object string type with maximum length equal to the maximum length of the <binary trim source>.

23) If <binary overlay function> is specified, then:

a) Let *BV* be the first <binary value expression>, let *SP* be the <start position>, and let *RS* be the second <binary value expression>.

b) If <string length> is specified, then let *SL* be <string length>; otherwise, let *SL* be OCTET_LENGTH(*RS*).

c) The <binary overlay function> is equivalent to:

```
   SUBSTRING ( BV FROM 1 FOR SP - 1 )
|| RS
|| SUBSTRING ( BV FROM SP + SL  )
```

24) 09 14 15 If <classifier function> *CF* is specified, then:

a) *CF* shall be contained in a <row pattern measures> or a <row pattern definition search condition>.

b) The declared type of the result of *CF* is character string with implementation-defined (IL048) maximum length, character set SQL_IDENTIFIER and collation SQL_IDENTIFIER.

c) If <row pattern variable name> is not specified, then the universal row pattern variable is implicit.

d) If *CF* is not simply contained in an aggregated argument of a <set function specification> or in a <row pattern navigation operation>, then *CF* is equivalent to

```
RUNNING LAST (CLASSIFIER (RPV))
```

where *RPV* is the explicit or implicit <row pattern variable name>.

Access Rules

1) Case:

a) If <string value function> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE for every transliteration identified by a <transliteration name> contained in the <string value expression>.

b) Otherwise, the current privileges shall include USAGE for every transliteration identified by a <transliteration name> contained in the <string value expression>.

General Rules

1) 15 The result of <string value function> is the result of the immediately contained <character value function> or <binary value function>.

2) 09 14 15 The result of <character value function> is the result of the immediately contained <character substring function>, <regular expression substring function>, <regex substring function>, <fold>, <transcoding>, <character transliteration>, <regex transliteration>, <trim function>, <pad function>, <character overlay function>, <normalize function>, <specific type method>, or <classifier function>.

3) If <character substring function> is specified, then:

a) If the character encoding form of <character value expression> is UTF8, UTF16, or UTF32, then, in the remainder of this General Rule, the term “character” shall be taken to mean “unit specified by <char length units>”.

b) Let *C* be the value of the <character value expression>, let *LC* be the length in characters of *C*, and let *S* be the value of the <start position>.

c) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.

d) If at least one of *C*, *S*, and *L* is the null value, then the result of the <character substring function> is the null value and no further General Rules of this Subclause are applied.

e) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error (22011)*.

f) Case:

i) If *S* is greater than *LC* or if *E* is less than 1 (one), then the result of the <character substring function> is the zero-length character string.

ii) Otherwise:

1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*−*S1*.

2) The result of the <character substring function> is a character string containing the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

4) If <regex substring function> is specified, then:

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, let *OCC* be the value of <regex occurrence>, let *CAP* be the <regex capture group>, and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If at least one of *STR*, *PAT*, *OCC*, *CAP*, and *FL* is the null value, then the result of the <regex substring function> is the null value.

ii) If *OCC* is less than 1 (one), then the result of <regex substring function> is the null value.

iii) If *CAP* is less than 0 (zero), or greater than the number of XQuery regular expression parenthesized subexpressions of *PAT*, then the result of <regex substring function> is the null value.

iv) If *SP* is less than 1 (one) or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex substring function> is the null value.

v) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex substring function> is implementation-dependent (UV074).

vi) Otherwise, the General Rules of Subclause 9.28, "XQuery regular expression matching", are applied with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, and *FL* as *FLAG*; let *LOMV* be the *LIST* returned from the application of those General Rules.

Case:

1) If there are at least *OCC* match vectors in *LOMV*, then let *MV* be the *OCC-th* match vector in *LOMV*. Let *PL* be *MV*[*CAP*], where *MV*[0] is the first position/length in *MV*. Let *P* be the position of *PL* and let *L* be the length of *PL*. The result of the <regex substring function> is

Case:

A) If *P* is 0 (zero), then the null value.

B) Otherwise, the substring of *STR* whose position is *P* and whose length is *L*.

2) Otherwise, the result of the <regex substring function> is the null value.

5) If <normalize function> is specified, then:

a) Let *S* be the value of <character value expression>.

b) If *S* is the null value, then the result of the <normalize function> is the null value.

c) Let *NR* be *S* in the normalized form specified by *NF* in accordance with ISO/IEC 10646:2020.

d) Case:

i) If the length in characters of *NR* is less than or equal to *L*, then the result of the <normalize function> is *NR*.

ii) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

6) If <regular expression substring function> is specified, then:

a) Let *C* be the result of the first <character value expression>, let *R* be the result of the second <character value expression>, and let *E* be the result of the <escape character>.

Case:

i) If at least one of *C*, *R*, and *E* is the null value, then the result of the <regular expression substring function> is the null value.

ii) If the length in characters of *E* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character (22019)*.

iii) If *R* does not contain exactly two occurrences of the two-character sequence consisting of *E*, each immediately followed by <double quote>, then an exception condition is raised: *data exception — invalid use of escape character (2200C)*.

iv) Otherwise, let *R1*, *R2*, and *R3* be the substrings of *R*, such that

```
'R' = 'R1' || 'E' || '"' || 'R2' || 'E' || '"' || 'R3'
```

is *True*.

Case:

1) If any one of *R1*, *R2*, or *R3* is not the zero-length character string and does not have the format of a <regular expression>, then an exception condition is raised: *data exception — invalid regular expression (2201B)*.

2) If the predicate

```
'C' SIMILAR TO 'R1' || 'R2' || 'R3' ESCAPE 'E'
```

is not *True*, then the result of the <regular expression substring function> is the null value.

3) Otherwise, the result *S* of the <regular expression substring function> is computed as follows:

A) Let *S1* be the shortest initial substring of *C* such that there is a substring *S23* of *C* such that the value of the following <search condition> is *True*:

```
'C' = 'S1' || 'S23' AND
'S1' SIMILAR TO 'R1' ESCAPE 'E' AND
'S23' SIMILAR TO '(R2R3)' ESCAPE 'E'
```

B) Let *S3* be the shortest final substring of *S23* such that there is a substring *S2* of *S23* such that the value of the following <search condition> is *True*:

```
'S23' = 'S2' || 'S3' AND
'S2' SIMILAR TO 'R2' ESCAPE 'E' AND
'S3' SIMILAR TO 'R3' ESCAPE 'E'
```

C) The result of the <regular expression substring function> is *S2*.

7) If <fold> is specified, then:

a) Let *S* be the value of the <character value expression>.

b) If *S* is the null value, then the result of the <fold> is the null value.

c) Let *FRML* be the length or maximum length in characters of the declared type of <fold>.

d) Case:

i) If UPPER is specified, then let *FR* be a copy of *S* in which every lower-case character that has a corresponding upper-case character or characters in the character set of *S* and every title case character that has a corresponding upper-case character or characters in the character set of *S* is replaced by that upper-case character or characters.

ii) If LOWER is specified, then let *FR* be a copy of *S* in which every upper-case character that has a corresponding lower-case character or characters in the character set of *S* and every title case character that has a corresponding lower-case character or characters in the character set of *S* is replaced by that lower-case character or characters.

e) If the character set of <character factor> is UTF8, UTF16, or UTF32, then *FR* is replaced by

Case:

i) If the <search condition> `S IS NORMALIZED` evaluated to *True*, then

```
NORMALIZE (FR)
```

ii) Otherwise, *FR*.

f) Let *FRL* be the length in characters of *FR*.

g) Case:

i) If *FRL* is less than or equal to *FRML*, then the result of the <fold> is *FR*. If the declared type of *FR* is fixed-length character string, then the result is padded on the right with (*FRML* − *FRL*) <space>s.

ii) If *FRL* is greater than *FRML*, then the result of the <fold> is the first *FRML* characters of *FR* with length *FRML*. If any of the right-most (*FRL* − *FRML*) characters of *FR* are not <truncating whitespace> characters, then a completion condition is raised: *warning — string data, right truncation (01004)*.

8) If a <character transliteration> is specified, then

Case:

a) If the value of <character value expression> is the null value, then the result of the <character transliteration> is the null value.

b) If <transliteration name> identifies a transliteration descriptor whose indication of how the transliteration is performed specifies an SQL-invoked routine *TR*, then the result of the <character transliteration> is the result of the invocation of *TR* with a single SQL argument that is the <character value expression> contained in the <character transliteration>.

c) Otherwise, the value of the <character transliteration> is the value returned by the transliteration identified by the <existing transliteration name> specified in the transliteration descriptor of the transliteration identified by <transliteration name>.

9) If <regex transliteration> is specified, then:

a) Let *RSS* be the <regex subject string>, let *STR* be the value of *RSS*, let *PAT* be the value of the <XQuery pattern>, let *SP* be the value of <start position>, let *CLU* be the <char length units>, let *REP* be the value of <XQuery replacement string>, and let *FL* be the value of <XQuery option flag>.

b) Case:

i) If at least one of *STR*, *PAT*, *REP*, and *FL* is the null value, then the result of the <regex transliteration> is the null value.

ii) If *SP* is less than 1 (one) or greater than the value of

```
CHARACTER_LENGTH ( RSS USING CLU )
```

then the result of <regex transliteration> is the null value.

iii) If *CLU* is OCTETS and the *SP*-th octet of *STR* is not the first octet of a character, then the result of <regex transliteration> is implementation-dependent (UV074).

iv) Otherwise:

1) The General Rules of Subclause 9.28, "XQuery regular expression matching", are applied with *STR* as *STRING*, *PAT* as *PATTERN*, *SP* as *POSITION*, *CLU* as *UNITS*, and *FL* as *FLAG*; let *LOMV* be the *LIST* returned from the application of those General Rules.

2) Let *RTO* be the <regex transliteration occurrence>.

Case:

A) If *RTO* is <regex occurrence>, then let *OCC* be the value of *RTO*.

Case:

I) If *OCC* is the null value, then the result of the <regex transliteration> is the null value.

II) If *OCC* is less than 1 (one) or greater than the number of match vectors in *LOMV*, then the result of the <regex transliteration> is *STR*.

III) Otherwise, let *MV* be the *OCC*-th match vector in *LOMV*. The General Rules of Subclause 9.29, "XQuery regular expression replacement", are applied with *MV* as *MATCH*, *STR* as *STRING*, *PAT* as *PATTERN*, *REP* as *REPLACEMENT*, and *FL* as *FLAG*; let the result of the <regex transliteration> be the *RESULT* returned from the application of those General Rules.

B) Otherwise, <regex transliteration occurrence> is ALL. Let *N* be the number of match vectors in *LOMV*. Let S_N be *STR*.

I) For all *i* between *N* and 1(one) in descending order, let MV_i be the *i*-th match vector in *LOMV*. The General Rules of Subclause 9.29, "XQuery regular expression replacement", are applied with MV_i as *MATCH*, S_i as *STRING*, *PAT* as *PATTERN*, *REP* as *REPLACEMENT*, and *FL* as *FLAG*; let S_{i-1} be the *RESULT* returned from the application of those General Rules.

NOTE 187 — The replacements are applied from the end of *STR* moving to the front to avoid altering the positions of matches within the string. When a replacement is performed, there are no unprocessed replacements later in the

string, and consequently the positions of other matches are not disturbed if a substring is replaced by a string of a different length.

II) The result of <regex transliteration> is S_0.

10) If a <transcoding> is specified, then

Case:

a) If the value of <character value expression> is the null value, then the result of the <transcoding> is the null value.

b) Otherwise, the value of the <transcoding> is the value of the <character value expression> after the application of the transcoding specified by <transcoding name>.

11) The result of <trim function> is the result of the immediately contained <single-character trim function> or <multi-character trim function>.

12) If <single-character trim function> is specified, then:

a) Let *S* be the value of the <trim source>.

b) Let *SC* be the value of <trim character>.

c) If at least one of *S* and *SC* is the null value, then the result of the <single-character trim function> is the null value and no further General Rules of this Subclause are applied.

d) If the length in characters of *SC* is not 1 (one), then an exception condition is raised: *data exception — trim error (22027)*.

e) Case:

i) If BOTH is specified or if no <trim specification> is specified, then the result of the <single-character trim function> is the value of *S* with every leading and trailing character equal to *SC* removed.

ii) If TRAILING is specified, then the result of the <single-character trim function> is the value of *S* with every trailing character equal to *SC* removed.

iii) If LEADING is specified, then the result of the <single-character trim function> is the value of *S* with every leading character equal to *SC* removed.

13) If <multi-character trim function> *MTF* is specified, then:

a) Let *S* be the value of *TS*.

b) Let *SC* be the value of *TC*.

c) If at least one of *S* and *SC* is the null value, then the result of *MTF* is the null value and no further General Rules of this Subclause are applied.

d) If *S* is the zero-length character string or *SC* is the zero-length character string, then the result of *MTF* is *S* and no further General Rules of this Subclause are applied.

e) Case:

i) If RTRIM is specified:

1) Let *P* be the length in characters of *S*.

2) Let *C* be the *P*-th character of *S*.

3) While *C* is contained in *SC*, do:

A) Let *P* be *P* – 1 (one).

B) Let *C* be the *P*-th character of *S*.

4) Let *LP* be the <exact numeric literal> that has the value *P*.

5) The result of *MTF* is SUBSTRING (*TS* FROM 1 FOR *LP*)

ii) If LTRIM is specified:

1) Let *P* be 1 (one).

2) Let *C* be the *P*-th character of *S*.

3) While *C* is contained in *SC*, do:

A) Let *P* be *P* + 1 (one).

B) Let *C* be the *P*-th character of *S*.

4) Let *LP* be the <exact numeric literal> that has the value *P*.

5) The result of *MTF* is SUBSTRING (*TS* FROM *LP*)

14) If <pad function> is specified, then:

a) Let *PS* be the value of <pad source>. If *PS* is the null value, then the result of <pad function> is the null value and no further General Rules of this Subclause are applied.

b) Let *TPL* be the value of <total pad length>. If *TPL* is the null value, then the result of <pad function> is the null value and no further General Rules of this Subclause are applied.

c) If *TPL* is less than 1 (one), then the result of <pad function> is the zero-length character string and no further General Rules of this Subclause are applied.

d) Let *PV* be the value of <padding character value expression>. If *PV* is the null value, then the result of <pad function> is the null value and no further General Rules of this Subclause are applied.

e) Let *LPS* be the length of *PS*.

f) Let *LPV* be the maximum of 1 (one) and the length of *PV*.

g) Let *MRL* be the minimum of *TPL* and *LR*.

h) Let *MPL* be *MRL* – *LPS*.

i) If *MPL* is 0 (zero), then the result of the <pad function> is *PS* and no further General Rules of this Subclause are applied.

j) If *MPL* is negative, then the result of the <pad function> is the first *MRL* characters of *PS* and no further General Rules of this Subclause are applied.

k) Let *PI* be the greatest integer less than or equal to $\frac{MPL}{LPV}$.

l) Let *PR* be the value of MOD(*MPL*, *LPV*).

m) Let D_0 be the zero-length character string. For *i*, 1 (one) ≤ *i* ≤ *PI*, let D_i be D_{i-1} || *PV*.

n) If *PR* is greater than 0 (zero), then let *PVR* be the first *PR* characters of *PV*; otherwise, let *PVR* be the zero-length character string.

o) The result of the <pad function> is

Case:

i) If RPAD is specified, then *PS* || D_{PI} || *PVR*.

ii) Otherwise (LPAD is specified), D_{PI} || *PVR* || *PS*.

15) If <specific type method> is specified, then:

a) Let *V* be the value of the <user-defined type value expression>.

b) Case:

i) If *V* is the null value, then let *RV* be the null value.

ii) Otherwise:

1) Let *UDT* be the most specific type of *V*.

2) Let *UDTN* be the <user-defined type name> of *UDT*.

3) Let *CN* be the <catalog name> contained in *UDTN*, let *SN* be the <unqualified schema name> contained in *UDTN*, and let *UN* be the <qualified identifier> contained in *UDTN*. Let *CND*, *SND*, and *UND* be *CN*, *SN*, and *UN*, respectively, with every occurrence of <double quote> replaced by <double quote symbol>. Let *RV* be:

"*CND*"."*SND*"."*UND*"

c) The result of <specific type method> is *RV*.

16) [14] The result of <binary value function> is the result of the simply contained <binary substring function>, <binary trim function>, or <binary overlay function>.

17) If <binary substring function> is specified, then:

a) Let *B* be the value of the <binary value expression>, let *LB* be the length in octets of *B*, and let *S* be the value of the <start position>.

b) If <string length> is specified, then let *L* be the value of <string length> and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LB*+1 and *S*.

c) If at least one of *B*, *S*, and *L* is the null value, then the result of the <binary substring function> is the null value.

d) If *E* is less than *S*, then an exception condition is raised: *data exception — substring error (22011)*.

e) Case:

i) If *S* is greater than *LB* or if *E* is less than 1 (one), then the result of the <binary substring function> is the zero-length binary string.

ii) Otherwise:

1) Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LB*+1. Let *L1* be *E1*−*S1*.

2) The result of the <binary substring function> is a binary string containing *L1* octets of *B* starting at octet number *S1* in the same order that the octets appear in *B*.

18) If <binary trim function> is specified, then

a) Let *S* be the value of the <binary trim source>.

b) Let *SO* be the value of <trim octet>.

c) If at least one of *S* and *SO* is the null value, then the result of the <binary trim function> is the null value.

d) If the length in octets of *SO* is not 1 (one), then an exception condition is raised: *data exception — trim error (22027)*.

e) Case:

i) If BOTH is specified or if no <trim specification> is specified, then the result of the <binary trim function> is the value of *S* with any leading or trailing octets equal to *SO* removed.

ii) If TRAILING is specified, then the result of the <binary trim function> is the value of *S* with any trailing octets equal to *SO* removed.

iii) If LEADING is specified, then the result of the <binary trim function> is the value of *S* with any leading octets equal to *SO* removed.

19) If the result of <string value function> is the zero-length character string or the zero-length binary string, then it is implementation-defined (IA215) whether an exception condition is raised: *data exception — zero-length character string (2200F)* or *data exception — zero-length binary string (2201Y)*, respectively.

20) 09 14 15 If <classifier function> *CF* is specified, then:

a) Case:

i) If *CF* is contained in a <row pattern definition search condition>, then let (*STR*, *RS*, *k*) be the current potential row pattern match.

ii) If *CF* is contained in a <row pattern recognition clause>, then let (*STR*, *RS*, *k*) be the current retained row pattern match.

iii) Otherwise, let (*STR*, *RS*, *k*) be the current designated row pattern match.

NOTE 188 — The definitions of potential, retained, and designated row pattern matches are expressed as "(*STR*, *RS*, *k*)", as specified in Subclause 9.41, "Row pattern recognition in a sequence of rows".

b) Let *NP* be the number of rows in *RS*. Let the rows of *RS* form the sequence R_1, R_2, ..., R_{NP}.

c) Case:

i) If *CF* is contained in <row pattern definition search condition>, then let $CR = R_c$ be the row in which the <row pattern definition search condition> is being evaluated.

ii) If *Vcount*(*STR*) = 0 (zero), then let *c* = 0 (zero).

NOTE 189 — That is, in the case of an empty match, let *c* point at a non-existent row before the beginning of the row pattern partition. This will ultimately force the result of <classifier function> to be null.

iii) If *CF* is contained in a <row pattern rows per match> that specifies ALL ROWS PER MATCH, then let $CR = R_c$ be the current row.

iv) Otherwise, let $CR = R_c$ be the last row that is mapped by (*STR*, *RS*, *k*).

d) Case:

i) If *CF* is contained in a <row pattern navigation operation>, then let *ER* = R_e be the row in which *CF* is being evaluated.

NOTE 190 — Consult the General Rules of Subclause 4.23.3, "Row pattern navigation operations", to determine this row. In some scenarios, <row pattern navigation operation> evaluates to the null value without determining a row to evaluate. In such cases, the General Rules of this Subclause are not applied, since the <value expression> of the <row pattern navigation operation> is not evaluated.

ii) If *CF* is contained in an aggregated argument of a <set function specification> *SFS*, then let *ER* = R_e be the row of the argument source of *SFS* in which *CF* is being evaluated.

NOTE 191 — Because of a syntactic transformation in the Syntax Rules, every <classifier function> is either contained in an aggregated argument of a <set function specification> or contained in a <row pattern navigation operation>.

e) The result of <classifier function> is

Case:

i) If *c* = 0 (zero), then the null value.

ii) If *CF* is contained in a <row pattern definition search condition>, and *e* > *c*, then the null value.

NOTE 192 — During evaluation of a <row pattern definition search condition>, the classifier of a future row is treated as null.

iii) If R_e is not a row that is mapped to a row pattern variable by (*STR*, *RS*, *k*), then the null value.

iv) Otherwise, a character string whose value is equivalent to the name of the primary row pattern variable to which R_e is mapped.

Conformance Rules

1) Without at least one of Feature R010, "Row pattern recognition: FROM clause", or Feature R020, "Row pattern recognition: WINDOW clause", conforming SQL language shall not contain <classifier function>.

2) Without Feature T581, "Regular expression substring function", conforming SQL language shall not contain a <regular expression substring function>.

3) Without Feature T312, "OVERLAY function", conforming SQL language shall not contain a <character overlay function>.

4) Without Feature T312, "OVERLAY function", conforming SQL language shall not contain a <binary overlay function>.

5) Without at least one of Feature T042, "Extended LOB data type support", or Feature T022, "Advanced support for BINARY and VARBINARY data types", conforming SQL language shall not contain a <binary value function>.

6) Without Feature F695, "Translation support", conforming SQL language shall not contain a <character transliteration>.

7) Without Feature F695, "Translation support", conforming SQL language shall not contain a <transcoding>.

8) Without Feature T061, "UCS support", conforming SQL language shall not contain a <normalize function>.

9) Without Feature S261, "Specific type method", conforming SQL language shall not contain a <specific type method>.

10) Without Feature F394, "Optional normal form specification", conforming SQL language shall not contain <normal form>.

11) Without Feature F421, "National character", conforming SQL language shall not contain a <character value function> that has a declared type of NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, or NATIONAL CHARACTER LARGE OBJECT.

12) Without Feature F844, "SUBSTRING_REGEX", conforming SQL language shall not contain <regex substring function>.

13) Without Feature F846, "Octet support in regular expression operators", in conforming SQL language, <regex substring function> shall not contain <char length units>.

14) Without Feature F845, "TRANSLATE_REGEX", conforming SQL language shall not contain <regex transliteration>.

15) Without Feature F846, "Octet support in regular expression operators", in conforming SQL language, <regex transliteration> shall not contain <char length units>.

16) Without Feature F847, "Non-constant regular expression", in conforming SQL language, <XQuery pattern>, <XQuery option flag>, and <XQuery replacement string> shall be <value specification>s.

17) Without Feature T050, "POSITION, CHAR_LENGTH, OCTET_LENGTH, LOWER, TRIM, UPPER, and SUBSTRING for CLOBs", conforming SQL language shall not contain a <fold>, a <trim source>, or a <character substring function> that contains a <character value expression> whose declared type is character large object type.

18) Without Feature T047, "POSITION, OCTET_LENGTH, TRIM, and SUBSTRING for BLOBs", conforming SQL language shall not contain a <binary trim source> or a <binary substring function> that contains a <binary value expression> whose declared type is binary large object type.

19) Without Feature T055, "String padding functions", conforming SQL language shall not contain a <pad function>.

20) [14] [15] Without Feature T056, "Multi-character TRIM function", conforming SQL language shall not contain a <multi-character trim function>.

6.34 <JSON value constructor>

Function

Specify construction of a JSON text fragment.

Format

```
<JSON value constructor> ::=
    <JSON object constructor>
  | <JSON array constructor>

<JSON object constructor> ::=
  JSON_OBJECT <left paren>
      [ <JSON name and value> [ { <comma> <JSON name and value> }... ] ]
      [ <JSON constructor null clause> ]
      [ <JSON key uniqueness constraint> ] ]
      [ <JSON output clause> ]
      <right paren>

<JSON name and value> ::=
    <JSON name and value 1>
  | <JSON name and value 2>

<JSON name and value 1> ::=
  [ KEY ] <JSON name> VALUE <JSON input expression>

<JSON name and value 2> ::=
  <JSON name> <colon> <JSON input expression>

<JSON name> ::=
  <character value expression>

<JSON constructor null clause> ::=
    NULL ON NULL
  | ABSENT ON NULL

<JSON array constructor> ::=
    <JSON array constructor by enumeration>
  | <JSON array constructor by query>

<JSON array constructor by enumeration> ::=
  JSON_ARRAY <left paren>
      [ <JSON input expression> [ { <comma> <JSON input expression> }... ]
      [ <JSON constructor null clause> ] ]
      [ <JSON output clause> ]
      <right paren>

<JSON array constructor by query> ::=
  JSON_ARRAY <left paren>
      <query expression>
      [ <JSON input clause> ]
      [ <JSON constructor null clause> ]
      [ <JSON output clause> ]
      <right paren>
```

Syntax Rules

1) The declared type of a <JSON value constructor> *JVC* is the declared type of its immediately contained <JSON object constructor> or <JSON array constructor>.

2) If <JSON output clause> is not specified, then

   Case:

   a) If the declared type of any <value expression> simply contained in <JSON input expression> simply contained in *JVC* is a JSON type, then RETURNING JSON is implicit.

   b) Otherwise, RETURNING *ST* FORMAT JSON is implicit, where *ST* is an implementation-defined (ID098) string type.

3) Let *JVCFDT* be the <data type> immediately contained in the explicit or implicit <JSON output clause> *JOC* and let *JVCFFO* be the explicit or implicit <JSON representation> of *JOC*.

4) If *JVCFDT* is not a JSON type, then the Syntax Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *JVCFFO* as *FORMAT OPTION* and *JVCFDT* as *TARGET TYPE*.

5) If <JSON name and value 1> *JNV1* is specified and *JNV1* does not immediately contain KEY, then the <JSON name> immediately contained in *JNV1* shall not be a <routine invocation> that immediately contains a <routine name> that is a <regular identifier> that is equivalent to KEY.

   NOTE 193 — This Syntax Rule resolves an ambiguity. In a <JSON name and value> like "KEY(x) VALUE y", KEY could be a key word or the beginning of a routine invocation. This Syntax Rule specifies that it is a key word.

6) If <JSON object constructor> is specified, then:

   a) If <JSON constructor null clause> is not specified, then NULL ON NULL is implicit.

   b) If <JSON key uniqueness constraint> is not specified, then WITHOUT UNIQUE KEYS is implicit.

   c) Let *WUK* be the explicit or implicit <JSON key uniqueness constraint>.

   d) The declared type of <JSON object constructor> is *JVCFDT*.

7) If <JSON array constructor> is specified, then:

   a) If <JSON constructor null clause> is not specified, then ABSENT ON NULL is implicit.

   b) The declared type of <JSON array constructor> is *JVCFDT*.

   c) If <JSON array constructor by query> is specified, then the <query expression> *QE* shall be of degree 1 (one).

   d) If <JSON array constructor by enumeration> is specified and immediately contains exactly 1 (one) <JSON input expression> *JVE*, then *JVE* shall not be a <scalar subquery>.

      NOTE 194 — This Syntax Rule resolves an ambiguity in which a <JSON array constructor> (e.g., JSON_ARRAY((SELECT a FROM t))) might otherwise be interpreted either as a <JSON array constructor by enumeration> or as a <JSON array constructor by query>. The ambiguity is resolved by adopting the interpretation that such a <JSON array constructor> is a <JSON array constructor by query>.

Access Rules

None.

General Rules

1) The value of a <JSON value constructor> is the value of its immediately contained <JSON object constructor> or <JSON array constructor>.

2) If <JSON object constructor> *JOC* is specified, then:

   a) If the length of the value of any <JSON name> simply contained in *JOC* exceeds its implementation-defined (IL049) maximum length, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

   b) If any <JSON input expression> simply contained in *JOC* contains a <string value expression> *SVE* and the length of the value of *SVE* exceeds its implementation-defined (IL050) maximum length, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

   c) Let *NNV* be the number of <JSON name and value>s immediately contained in *JOC*.

   d) Case:

      i) If *NNV* = 0 (zero), then let *CJO* be a JSON object with no members.

      ii) Otherwise:

         1) For each *i*, 1 (one) ≤ *i* ≤ *NNV*:

            A) Let JNV_i be the *i*-th <JSON name and value> immediately contained in *JOC*.

            B) Let JN_i be the <JSON name> simply contained in JNV_i.

            C) Let VJN_i be the value of JN_i.

            D) If VJN_i is the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.

            E) Let JVE_i be the <JSON input expression> simply contained in JNV_i and let $VJVE_i$ be the value of JVE_i.

            F) Case:

               I) If $VJVE_i$ is a null value, then let JBV_i be the SQL/JSON null.

               II) If the declared type of $VJVE_i$ is a JSON type, then let JBV_i be $VJVE_i$.

               III) If JVE_i specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.42, "Parsing JSON text", are applied with JVE_i as *JSON TEXT*, *JFO* as *FORMAT OPTION*, and *WUK* as *UNIQUENESS CONSTRAINT*; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

                  Case:

                  1) If *ST* is an exception condition, then the exception condition *ST* is raised.

                  2) Otherwise, let JBV_i be *SJI*.

               IV) Otherwise,

                  Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JBV_i be $VJVE_i$.

2) Otherwise, let JBV_i be the result of

```
CAST (VJVE_i AS SDT)
```

where *SDT* is an implementation-defined (ID100) character string type with character set Unicode.

G) Let M_i be the SQL/JSON member whose key is JVN_i and whose bound value is JBV_i.

2) If WITH UNIQUE KEYS is specified and, for any i, 1 (one) $\leq i \leq NNV$, and any j, $i < j \leq NNV$, VJN_i and VJN_j would be equivalent when interpreted as keys of an SQL/JSON object, then an exception condition is raised: *data exception — duplicate JSON object key value (22030)*.

3) Case:

A) If the implicit or explicit <JSON constructor null clause> specifies NULL ON NULL, then

Case:

I) If *NNV* is greater than the implementation-defined (IL051) maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members (2203E)*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) $\leq i \leq NNV$.

B) Otherwise,

Case:

I) If the number of M_i, 1 (one) $\leq i \leq NNV$, whose bound values are not the SQL/JSON null is greater than the implementation-defined (IL051) maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members (2203E)*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) $\leq i \leq NNV$, whose bound values are not the SQL/JSON null.

NOTE 195 — There is no implied order of the members of the constructed JSON object.

e) Let *JVCF* be *CJO*.

3) If <JSON array constructor by enumeration> *JAC* is specified, then:

a) If any <JSON input expression> simply contained in <JSON array constructor by enumeration> contains a <string value expression> *SVE* and the length of the value of *SVE* exceeds its implementation-defined (IL050) maximum length, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

b) Let *NJVE* be the number of <JSON input expression>s immediately contained in *JAC*.

c) Case:

i) If *NJVE* is 0 (zero), then let *CJA* be a JSON array with no elements.

ii) Otherwise:

1) For each i, 1 (one) ≤ i ≤ *NJVE*:

A) Let JVE_i be the i-th <JSON input expression> immediately contained in *JAC* and let $VJVE_i$ be the value of JVE_i.

B) Case:

I) If $VJVE_i$ is a null value, then let JE_i be the SQL/JSON null.

II) If the declared type of $VJVE_i$ is a JSON type, then let JE_i be $VJVE_i$.

III) If JVE_i specifies, explicitly or implicitly, <JSON input clause> *JFO*, then General Rules of Subclause 9.42, "Parsing JSON text", are applied with JVE_i as *JSON TEXT*, *JFO* as *FORMAT OPTION*, and an implementation-defined (IV156) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

Case:

1) If *ST* is an exception condition, then the exception condition *ST* is raised.

2) Otherwise, let JE_i be *SJI*.

IV) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JE_i be $VJVE_i$.

2) Otherwise, let JE_i be the result of

```
CAST (VJVE_i AS SDT)
```

where *SDT* is an implementation-defined (ID100) character string type with character set Unicode.

2) Case:

A) If the implicit or explicit <JSON constructor null clause> specifies NULL ON NULL, then

Case:

I) If *NJVE* is greater than the implementation-defined (IL052) maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements (2203D)*.

II) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ i ≤ *NJVE*.

B) Otherwise,

Case:

I) If the number of JE_i, 1 (one) ≤ i ≤ *NJVE*, that are not the SQL/JSON null is greater than the implementation-defined (IL052) maximum

number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements (2203D)*.

II) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ i ≤ *NJVE*, that are not the SQL/JSON null.

NOTE 196 — The elements of constructed JSON arrays are ordered and array element indices start with 0 (zero).

d) Let *JVCF* be *CJA*.

4) If <JSON array constructor by query> *JACQ* is specified, then:

a) *QE* is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

b) Case:

i) If *N* is 0 (zero), then let *CJAQ* be a JSON array with no elements.

ii) Otherwise:

1) For each i, 1 (one) ≤ i ≤ *N*:

A) Let JVE_i be the i-th row of *T* and let $VJVE_i$ be the value of the column of JVE_i.

B) Case:

I) If $VJVE_i$ is a null value, then let JE_i be the SQL/JSON null.

II) If the declared type of $VJVE_i$ is a JSON type, then let JE_i be $VJVE_i$.

III) If *JACQ* contains a <JSON input clause> *JFO*, then the General Rules of Subclause 9.42, "Parsing JSON text", are applied with $VJVE_i$ as *JSON TEXT*, *JFO* as *FORMAT OPTION*, and an implementation-defined (IV156) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

Case:

1) If *ST* is an exception condition, then the exception condition *ST* is raised.

2) Otherwise, let JE_i be *SJI*.

IV) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JE_i be $VJVE_i$.

2) Otherwise, let JE_i be the result of

```
CAST (VJVE_i AS SDT)
```

where where *SDT* is an implementation-defined (ID100) character string type with character set Unicode.

2) Case:

A) If the implicit or explicit <JSON constructor null clause> specifies NULL ON NULL, then

Case:

I) If *N* is greater than the implementation-defined (IL052) maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements (2203D)*.

II) Otherwise, let *CJAQ* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ i ≤ N.

B) Otherwise,

Case:

I) If the number of JE_i, 1 (one) ≤ i ≤ N, that are not the SQL/JSON null is greater than the implementation-defined (IL052) maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements (2203D)*.

II) Otherwise, let *CJAQ* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ i ≤ N, that are not the SQL/JSON null.

NOTE 197 — The elements of constructed JSON arrays are ordered and array element indices start with 0 (zero).

c) Let *JVCF* be *CJAQ*.

5) Case:

a) If *JVCFDT* is not a JSON type, then the General Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *JVCF* as *SQL/JSON ITEM*, *JVCFFO* as *FORMAT OPTION*, and *JVCFDT* as *TARGET TYPE*; let *ST* be the *STATUS* and let *CJV* be the *JSON TEXT* returned from the application of those General Rules.

Case:

i) If *ST* is an exception condition, then the exception condition *ST* is raised.

ii) Otherwise, the result of <JSON value constructor> is *CJV*.

b) Otherwise, The result of <JSON value constructor> is *JVCF*.

Conformance Rules

1) Without Feature T811, "Basic SQL/JSON constructor functions", conforming SQL language shall not contain <JSON value constructor>.

2) Without Feature T811, "Basic SQL/JSON constructor functions", conforming SQL language shall not contain <JSON object constructor>.

3) Without Feature T811, "Basic SQL/JSON constructor functions", conforming SQL language shall not contain <JSON array constructor>.

4) Without Feature T814, "Colon in JSON_OBJECT or JSON_OBJECTAGG", conforming SQL language shall not contain <JSON name and value 2>.

5) Without Feature T830, "Enforcing unique keys in SQL/JSON constructor functions", conforming SQL language shall not contain a <JSON object constructor> that specifies a <JSON key uniqueness constraint>.

6.35 <JSON query>

Function

Extract a JSON text from a JSON text using an SQL/JSON path expression.

Format

```
<JSON query> ::=
  JSON_QUERY <left paren>
  <JSON API common syntax>
      [ <JSON output clause> ]
      [ <JSON query wrapper behavior> WRAPPER ]
      [ <JSON query quotes behavior> QUOTES [ ON SCALAR STRING ] ]
      [ <JSON query empty behavior> ON EMPTY ]
      [ <JSON query error behavior> ON ERROR ]
  <right paren>

<JSON query wrapper behavior> ::=
    WITHOUT [ ARRAY ]
  | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]

<JSON query quotes behavior> ::=
    KEEP
  | OMIT

<JSON query empty behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<JSON query error behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT
```

Syntax Rules

1) If <JSON output clause> is not specified, then

   Case:

   a) If the declared type of any <value expression> simply contained in <JSON input expression> simply contained in <JSON query> is a JSON type, then RETURNING JSON is implicit.

   b) Otherwise, RETURNING *SDT* FORMAT JSON is implicit, where *SDT* is an implementation-defined (ID101) string type.

2) The declared type *DECT* of <JSON query> is the type specified by the <data type> *DT* contained in the explicit or implicit <JSON output clause>.

3) If *DECT* is a JSON type, then the implicit or explicit <JSON query quotes behavior> shall specify KEEP.

4) If <JSON query empty behavior> is not specified, then NULL ON EMPTY is implicit.

5) If <JSON query error behavior> is not specified, then NULL ON ERROR is implicit.

6) If <JSON query wrapper behavior> is not specified, then WITHOUT ARRAY is implicit

7) If <JSON query wrapper behavior> specifies WITH and neither CONDITIONAL nor UNCONDITIONAL is specified, then UNCONDITIONAL is implicit.

8) If <JSON query wrapper behavior> specifies WITH, then <JSON query quotes behavior> shall not be specified.

9) If <JSON query quotes behavior> is not specified, then KEEP is implicit.

Access Rules

None.

General Rules

1) If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of <JSON query> is the null value and no further General Rules of this Subclause are applied.

2) Let *JACS* be the <JSON API common syntax> simply contained in <JSON query>.

3) The General Rules of Subclause 9.47, "Processing <JSON API common syntax>", are applied with *JACS* as *JSON API COMMON SYNTAX*; let *ST1* be the *STATUS* and let *SEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

4) Let *WB* be the explicit or implicit <JSON query wrapper behavior>, let *QB* be the explicit or implicit <JSON query quotes behavior>, let *ZB* be the explicit or implicit <JSON query empty behavior>, let *EB* be the explicit or implicit <JSON query error behavior>, and let *FO* be the explicit or implicit <JSON representation> contained in the <JSON query>.

5) Case:

   a) If *DECT* is a JSON type, then the General Rules of Subclause 9.44, "Converting an SQL/JSON sequence to an SQL/JSON item", are applied with *ST1* as *STATUS IN*, *SEQ* as *SQL/JSON SEQUENCE*, *WB* as *WRAPPER BEHAVIOR*, *ZB* as *EMPTY BEHAVIOR*, and *EB* as *ERROR BEHAVIOR*; let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* returned from the application of those General Rules.

   b) Otherwise, the General Rules of Subclause 9.49, "Serializing an SQL/JSON sequence to an SQL string type", are applied with *ST1* as *STATUS IN*, *SEQ* as *SQL/JSON SEQUENCE*, *WB* as *WRAPPER BEHAVIOR*, *QB* as *QUOTES BEHAVIOR*, *ZB* as *EMPTY BEHAVIOR*, *EB* as *ERROR BEHAVIOR*, *DT* as *DATA TYPE*, and *FO* as *FORMAT OPTION*; let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* returned from the application of those General Rules.

6) If *ST2* is an exception condition, then the exception condition *ST2* is raised. Otherwise, *V* is the result of the <JSON query>.

Conformance Rules

1) Without Feature T828, "JSON_QUERY", conforming SQL language shall not contain <JSON query>.

2) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON query> shall not contain <JSON query empty behavior>.

3) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON query> shall not contain <JSON query error behavior>.

4) Without Feature T829, "JSON_QUERY: array wrapper options", <JSON query> shall not contain <JSON query wrapper behavior>.

6.36 <JSON simplified accessor>

Function

Extract a value from an SQL/JSON value.

Format

```
<JSON simplified accessor> ::=
  <value expression primary> <JSON simplified accessor op chain>

<JSON simplified accessor op chain> ::=
    <JSON simplified accessor op>
  | <JSON simplified accessor op chain> <JSON simplified accessor op>

<JSON simplified accessor op> ::=
    <JSON member accessor>
  | <JSON wildcard member accessor>
  | <JSON array accessor>
  | <JSON wildcard array accessor>
  | <JSON item method>
```

Syntax Rules

1) Let *JA* be the <JSON simplified accessor>, let *VEP* be the <value expression primary> immediately contained in *JA*, and let *JC* be the <JSON simplified accessor op chain> immediately contained in *JA*. Let *JO* be the <JSON simplified accessor op> immediately contained in *JC*.

2) The declared type of *VEP* shall be a JSON type.

3) Case:

   a) If *JO* is a <JSON item method> *JIM* and the <JSON method> *JM* simply contained in *JIM* specifies `bigint`, `boolean`, `date`, `decimal`, `double`, `integer`, `number`, `size`, `string`, `time`, `time_tz`, `timestamp`, `timestamp_tz`, or `type`, then:

      i) Case:

         1) If *JM* specifies `bigint`, then let *RT* be BIGINT.

         2) If *JM* specifies `boolean`, then let *RT* be BOOLEAN.

         3) If *JM* specifies `date`, then let *RT* be DATE.

         4) If *JM* specifies `decimal`, then

            Case:

            A) If <scale> *S* is specified, then let *P* be the <precision>. Let *RT* be DECIMAL(*P*,*S*).

            B) If <precision> *P* is specified, then let *RT* be DECIMAL(*P*).

            C) Otherwise, let *RT* be DECIMAL.

         5) If *JM* specifies `double`, then let *RT* be DOUBLE PRECISION.

         6) If *JM* specifies `integer`, then let *RT* be INTEGER.

7) If *JM* specifies `number` or `size`, then let *ML* be an implementation-defined (IL064) maximum length for variable-length character strings and let *RT* be CHARACTER VARYING(*ML*).

8) If *JM* specifies `string` or `type`, then let *RT* be BIGINT.

9) If *JM* specifies `time`, then

   Case:

   A) If <time precision> *P* is specified, then let *RT* be TIME(*P*).

   B) Otherwise, let *RT* be TIME.

10) If *JM* specifies `time_tz`, then

    Case:

    A) If <time precision> *P* is specified, then let *RT* be TIME(*P*) WITH TIME ZONE.

    B) Otherwise, let *RT* be TIME WITH TIME ZONE.

11) If *JM* specifies `timestamp`, then

    Case:

    A) If <timestamp precision> *P* is specified, then let *RT* be TIMESTAMP(*P*).

    B) Otherwise, let *RT* be TIMESTAMP.

12) If *JM* specifies `timestamp_tz`, then

    Case:

    A) If <timestamp precision> *P* is specified, then let *RT* be TIMESTAMP(*P*) WITH TIME ZONE.

    B) Otherwise, let *RT* be TIMESTAMP WITH TIME ZONE.

ii) *JA* is equivalent to the <JSON value function>:

```
JSON_VALUE ( VEP, 'lax $.JC' RETURNING RT
  NULL ON EMPTY NULL ON ERROR)
```

NOTE 198 — This syntactic transformation maintains the case-sensitivity of *JC* as specified. No implicit case folding is performed.

b) Otherwise, *JA* is equivalent to the <JSON query>:

```
JSON_QUERY ( VEP, 'lax $.JC' WITH CONDITIONAL ARRAY WRAPPER
  NULL ON EMPTY NULL ON ERROR)
```

NOTE 199 — This syntactic transformation maintains the case-sensitivity of *JC* as specified. No implicit case folding is performed.

4) The Conformance Rules of Subclause 6.1, "<data type>",are applied to the result of the preceding syntactic transformation.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T861, "SQL/JSON simplified accessor: case-sensitive JSON member accessor", in conforming SQL language, a <JSON path key name> simply contained in a <JSON simplified accessor op> shall conform to the Format of <delimited identifier> and shall not contain a <reverse solidus> or a <double quote symbol>.

2) Without Feature T860, "SQL/JSON simplified accessor: column reference only", conforming SQL language shall not contain a <JSON simplified accessor>.

3) Without Feature T864, "SQL/JSON simplified accessor", in conforming SQL language, the <value expression primary> immediately contained in <JSON simplified accessor> shall be a <column reference>.

4) Without Feature T862, "SQL/JSON simplified accessor: wildcard member accessor", in conforming SQL language, a <JSON simplified accessor op chain> shall not simply contain a <JSON wildcard member accessor>.

5) Without Feature T863, "SQL/JSON simplified accessor: single-quoted string literal as member accessor", in conforming SQL language, a <JSON simplified accessor op> shall not simply contain a <JSON member accessor> that immediately contains a <JSON path string literal> that starts with a <quote>.

6) Without Feature T861, "SQL/JSON simplified accessor: case-sensitive JSON member accessor", in conforming SQL language, a <JSON path string literal> *JL* that starts and ends with a <quote> immediately contained in a <JSON member accessor> simply contained in a <JSON simplified accessor op> shall not contain a <reverse solidus> or any <quote> other than the <quote>s with which *JL* starts and ends.

6.37 <JSON serialize>

Function

Serialize a value of JSON type as a string.

Format

```
<JSON serialize> ::=
  JSON_SERIALIZE <left paren> <JSON value expression>
      [ <JSON output clause> ]
  <right paren>
```

Syntax Rules

1) If <JSON output clause> is not specified, then RETURNING *SDT* FORMAT JSON is implicit, where *SDT* is an implementation-defined (ID102) string type.

2) Let *FO* be the explicit or implicit <JSON representation> simply contained in the explicit or implicit <JSON output clause> *JOC*. Let *DT* be the <data type> simply contained in *JOC*.

3) The Syntax Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *FO* as *FORMAT OPTION* and *DT* as *TARGET TYPE*.

4) The declared type of <JSON serialize> is the type specified by *DT*.

Access Rules

None.

General Rules

1) Let *JVE* be the value of the <JSON value expression>.

2) If *JVE* is the null value, then the value of <JSON serialize> is the null value and no further General Rules of this Subclause are applied.

3) The General Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *JVE* as *SQL/JSON ITEM*, *FO* as *FORMAT OPTION*, and *DT* as *TARGET TYPE*; let *ST* be the *STATUS* and let *V* be the *JSON TEXT* returned from the application of those General Rules.

   Case:

   a) If *ST* is an exception condition, then the exception condition *ST* is raised.

   b) Otherwise, *V* is the result of <JSON serialize>.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON serialize>.

2) Without Feature T851, "SQL/JSON: optional keywords for default syntax", conforming SQL language shall not contain a <JSON serialize> that simply contains FORMAT JSON.

6.38 <JSON value expression>

Function

Specify a value of type JSON.

Format

```
<JSON value expression> ::=
  <JSON primary>

<JSON primary> ::=
    <value expression primary>
  | <JSON typed value function>
```

Syntax Rules

1) The declared type of the <value expression primary> immediately contained in <JSON primary> shall be a JSON type.

2) The declared type of <JSON value expression> is the declared type of the simply contained <value expression primary> or <JSON typed value function>.

Access Rules

None.

General Rules

1) The value of <JSON value expression> is the value of the simply contained <value expression primary> or <JSON typed value function>.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON value expression>.

6.39 <JSON typed value function>

Function

Specify a function that returns a value of type JSON.

Format

```
<JSON typed value function> ::=
    <JSON parse>
  | <JSON scalar>
```

Syntax Rules

1) The declared type of the <JSON typed value function> is the declared type of the immediately contained <JSON parse> or <JSON scalar>.

Access Rules

None.

General Rules

1) The value of the <JSON typed value function> is the value of the immediately contained <JSON parse> or <JSON scalar>.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON typed value function>.

6.40 <JSON parse>

Function

Perform a non-validating parse of a string to produce an SQL/JSON value of type JSON.

Format

```
<JSON parse> ::=
  JSON <left paren> <string value expression>
      [ <JSON input clause> ]
      [ <JSON key uniqueness constraint> ]
  <right paren>
```

Syntax Rules

1) The declared type of <JSON parse> is JSON.

2) If <JSON input clause> is not specified, then FORMAT JSON is implicit.

3) Let *FO* be the explicit or implicit <JSON input clause>.

4) If <JSON key uniqueness constraint> is not specified, then WITHOUT UNIQUE KEYS is implicit.

5) Let *JPUC* be the explicit or implicit <JSON key uniqueness constraint>.

Access Rules

None.

General Rules

1) Let *SVE* be the value of the <string value expression>.

2) If *SVE* is the null value, then the value of <JSON parse> is the null value and no further General Rules of this Subclause are applied.

3) The General Rules of Subclause 9.42, "Parsing JSON text", are applied with *SVE* as *JSON TEXT*, *FO* as *FORMAT OPTION*, and *JPUC* as *UNIQUENESS CONSTRAINT*; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

   Case:

   a) If *ST* is an exception condition, then the exception condition *ST* is raised.

   b) Otherwise, *SJI* is the result of <JSON parse>.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain <JSON parse>.

2) Without Feature T802, "Enhanced JSON data type", conforming SQL language shall not contain a <JSON parse> that contains a <JSON key uniqueness constraint>.

3) Without Feature T851, "SQL/JSON: optional keywords for default syntax", conforming SQL language shall not contain a <JSON parse> that simply contains FORMAT JSON.

6.41 <JSON scalar>

Function

Generate an SQL/JSON scalar value of type JSON.

Format

```
<JSON scalar> ::=
  JSON_SCALAR <left paren> <value expression> <right paren>
```

Syntax Rules

1) The declared type of <JSON scalar> is JSON.

2) The declared type of the <value expression> *VE* shall be a character string data type, numeric data type, Boolean data type, or datetime data type.

Access Rules

None.

General Rules

1) Let *V* be the value of *VE*.

2) Case:

   a) If *V* is the null value, then the value of <JSON scalar> is the null value and no further General Rules of this Subclause are applied.

      NOTE 200 — In this case, the null value is an SQL null value. An SQL null value is distinct from an SQL/JSON null.

   b) Otherwise, the value of <JSON scalar> is an SQL/JSON scalar whose value is *V*.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON scalar>.

6.42 <datetime value expression>

Function

Specify a datetime value.

Format

```
<datetime value expression> ::=
    <datetime term>
  | <interval value expression> <plus sign> <datetime term>
  | <datetime value expression> <plus sign> <interval term>
  | <datetime value expression> <minus sign> <interval term>

<datetime term> ::=
  <datetime factor>

<datetime factor> ::=
  <datetime primary> [ <time zone> ]

<datetime primary> ::=
    <value expression primary>
  | <datetime value function>

<time zone> ::=
  AT <time zone specifier>

<time zone specifier> ::=
    LOCAL
  | TIME ZONE <interval primary>
```

Syntax Rules

1) The declared type of a <datetime primary> shall be datetime.

2) If the <datetime value expression> immediately contains neither <plus sign> nor <minus sign>, then the precision of the result of the <datetime value expression> is the precision of the <value expression primary> or <datetime value function> that it simply contains.

3) If the declared type of the <datetime primary> is DATE, then <time zone> shall not be specified.

4) Case:

   a) If <time zone> is specified and the declared type of <datetime primary> is TIMESTAMP WITHOUT TIME ZONE or TIME WITHOUT TIME ZONE, then the declared type of <datetime term> is TIMESTAMP WITH TIME ZONE or TIME WITH TIME ZONE, respectively, with the same fractional seconds precision as <datetime primary>.

   b) Otherwise, the declared type of <datetime term> is the same as the declared type of <datetime primary>.

5) If the <datetime value expression> immediately contains either <plus sign> or <minus sign>, then:

   a) The <interval value expression> or <interval term> shall contain only <primary datetime field>s that are contained within the <datetime value expression> or <datetime term>.

   b) The result of the <datetime value expression> contains the same <primary datetime field>s that are contained in the <datetime value expression> or <datetime term>, with a fractional seconds precision that is the greater of the fractional seconds precisions, if any, of either the

<datetime value expression> and <interval term>, or the <datetime term> and <interval value expression> that it simply contains.

6) The declared type of the <interval primary> immediately contained in a <time zone specifier> shall be INTERVAL HOUR TO MINUTE.

Access Rules

None.

General Rules

1) If the value of any <datetime primary>, <interval value expression>, <datetime value expression>, or <interval term> simply contained in a <datetime value expression> is the null value, then the result of the <datetime value expression> is the null value.

2) If <time zone> is specified and the <interval primary> immediately contained in <time zone specifier> is the null value, then the result of the <datetime value expression> is the null value.

3) The value of a <datetime primary> is the value of the immediately contained <value expression primary> or <datetime value function>.

4) In the following General Rules, arithmetic is performed so as to maintain the integrity of the datetime data type that is the result of the <datetime term> or <datetime value expression>. This may involve carry from or to the immediately next more significant <primary datetime field>. If the data type of the <datetime term> or <datetime value expression> is time with or without time zone, then arithmetic on the HOUR <primary datetime field> is undertaken modulo 24. If the <interval value expression> or <interval term> is a year-month interval, then the DAY field of the result is the same as the DAY field of the <datetime term> or <datetime value expression>.

5) The value of a <datetime term> is determined as follows. Let *DT* be the declared type, *DV* the UTC component of the value, and *TZD* the time zone component, if any, of the <datetime primary> *DP* simply contained in the <datetime term>, and let *STZD* be the current default time zone displacement of the SQL-session.

   Case:

   a) If <time zone> is not specified, then the value of <datetime term> is the value of *DP*.

   b) Otherwise:

      i) Case:

         1) If *DT* is datetime with time zone, then the UTC component of the <datetime term> is *DV*.

         2) Otherwise, the UTC component of the <datetime term> is *DV* – *STZD*.

      ii) Case:

         1) If LOCAL is specified, then let *TZ* be *STZD*.

         2) If TIME ZONE is specified, then

            Case:

            A) If the value of the <interval primary> immediately contained in <time zone specifier> is less than the minimum permitted negative time zone displacement or greater than the maximum permitted negative time zone displace-

ment, then an exception condition is raised: *data exception — invalid time zone displacement value (22009)*.

B) Otherwise, let *TZ* be the value of the <interval primary> simply contained in <time zone>.

iii) The time zone component of the value of the <datetime term> is *TZ*.

6) If a <datetime value expression> immediately contains the operator <plus sign> or <minus sign>, then the time zone component, if any, of the result is the same as the time zone component of the immediately contained <datetime term> or <datetime value expression>. The result (if the result type is without time zone) or the UTC component of the result (if the result type has time zone) is effectively evaluated as follows:

a) Case:

i) If <datetime value expression> immediately contains the operator <plus sign> and the <interval value expression> or <interval term> is not negative, or if <datetime value expression> immediately contains the operator <minus sign> and the <interval term> is negative, then successive <primary datetime field>s of the <interval value expression> or <interval term> are added to the corresponding fields of the <datetime value expression> or <datetime term>.

ii) Otherwise, successive <primary datetime field>s of the <interval value expression> or <interval term> are subtracted from the corresponding fields of the <datetime value expression> or <datetime term>.

b) If, after the preceding step, any <primary datetime field> of the result is outside the permissible range of values for the field or the result is invalid based on the natural rules for dates and times, then an exception condition is raised: *data exception — datetime field overflow (22008)*.

NOTE 201 — For the permissible range of values for <primary datetime field>s, see Table 13, "Valid values for datetime fields".

Conformance Rules

1) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain <datetime value expression> that immediately contains a <plus sign> or a <minus sign>.

2) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <time zone>.

6.43 <datetime value function>

Function

Specify a function yielding a value of type datetime.

Format

```
<datetime value function> ::=
    <current date value function>
  | <current time value function>
  | <current timestamp value function>
  | <current local time value function>
  | <current local timestamp value function>

<current date value function> ::=
  CURRENT_DATE

<current time value function> ::=
  CURRENT_TIME [ <left paren> <time precision> <right paren> ]

<current local time value function> ::=
  LOCALTIME [ <left paren> <time precision> <right paren> ]

<current timestamp value function> ::=
  CURRENT_TIMESTAMP [ <left paren> <timestamp precision> <right paren> ]

<current local timestamp value function> ::=
  LOCALTIMESTAMP [ <left paren> <timestamp precision> <right paren> ]
```

Syntax Rules

1) The declared type of a <current date value function> is DATE. The declared type of a <current time value function> is TIME WITH TIME ZONE. The declared type of a <current timestamp value function> is TIMESTAMP WITH TIME ZONE.

   NOTE 202 — See the Syntax Rules of Subclause 6.1, "<data type>", for rules governing <time precision> and <timestamp precision>.

2) Case:

   a) If <time precision> *TP* is specified, then LOCALTIME(*TP*) is equivalent to:

      ```
      CAST (CURRENT_TIME(TP) AS TIME(TP) WITHOUT TIME ZONE)
      ```

   b) Otherwise, LOCALTIME is equivalent to:

      ```
      CAST (CURRENT_TIME AS TIME WITHOUT TIME ZONE)
      ```

3) Case:

   a) If <timestamp precision> *TP* is specified, then LOCALTIMESTAMP(*TP*) is equivalent to:

      ```
      CAST (CURRENT_TIMESTAMP(TP) AS TIMESTAMP(TP) WITHOUT TIME ZONE)
      ```

   b) Otherwise, LOCALTIMESTAMP is equivalent to:

      ```
      CAST (CURRENT_TIMESTAMP AS TIMESTAMP WITHOUT TIME ZONE)
      ```

Access Rules

None.

General Rules

1) If the SQL-session context statement timestamp is "not set", then set it to the current date and time.

NOTE 203 — This can happen at any implementation-dependent time after the setting of the statement timestamp to "not set"; the determination of the statement timestamp need not wait for evaluation of a <datetime value function>.

2) The <datetime value function>s CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP respectively return the date, time, and timestamp from the current SQL-session context's statement timestamp; the time and timestamp values are returned with time zone displacement equal to the current default time zone displacement of the SQL-session.

3) If specified, <time precision> and <timestamp precision> respectively determine the precision of the time or timestamp value returned.

Conformance Rules

1) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <current local time value function> that contains a <time precision> that is not 0 (zero) and shall not contain a <current time value function> that contains a <time precision> that is not 0 (zero).

2) Without Feature F555, "Enhanced seconds precision", conforming SQL language shall not contain a <current local timestamp value function> that contains a <timestamp precision> that is neither 0 (zero) nor 6 and shall not contain a <current timestamp value function> that contains a <timestamp precision> that is neither 0 (zero) nor 6.

3) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <current time value function>.

4) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <current timestamp value function>.

6.44 <interval value expression>

Function

Specify an interval value.

Format

```
<interval value expression> ::=
    <interval term>
  | <interval value expression 1> <plus sign> <interval term 1>
  | <interval value expression 1> <minus sign> <interval term 1>
  | <left paren> <datetime value expression> <minus sign> <datetime term> <right paren>
       <interval qualifier>

<interval term> ::=
    <interval factor>
  | <interval term 2> <asterisk> <factor>
  | <interval term 2> <solidus> <factor>
  | <term> <asterisk> <interval factor>

<interval factor> ::=
  [ <sign> ] <interval primary>

<interval primary> ::=
    <value expression primary> [ <interval qualifier> ]
  | <interval value function>

<interval value expression 1> ::=
  <interval value expression>

<interval term 1> ::=
  <interval term>

<interval term 2> ::=
  <interval term>
```

Syntax Rules

1) The declared type of an <interval value expression> is interval. The declared type of a <value expression primary> immediately contained in an <interval primary> shall be interval.

2) Case:

   a) If the <interval value expression> simply contains an <interval qualifier> *IQ*, then the declared type of the result is INTERVAL *IQ*.

   b) If the <interval value expression> is an <interval term>, then the result of the <interval value expression> contains the same interval fields as the <interval primary>. If the <interval primary> contains a seconds field, then the result's fractional seconds precision is the same as the <interval primary>'s fractional seconds precision. The result's <interval leading field precision> is implementation-defined (IV157), but shall not be less than the <interval leading field precision> of the <interval primary>.

   c) If <interval term 1> is specified, then the result contains every interval field that is contained in the result of either <interval value expression 1> or <interval term 1>, and, if both contain a seconds field, then the fractional seconds precision of the result is the greater of the two fractional seconds precisions. The <interval leading field precision> is implementation-defined

(IV157), but shall be sufficient to represent all interval values with the interval fields and <interval leading field precision> of <interval value expression 1> as well as all interval values with the interval fields and <interval leading field precision> of <interval term 1>.

NOTE 204 — Interval fields are effectively defined by Table 4, "Fields in year-month INTERVAL values", and Table 5, "Fields in day-time INTERVAL values".

3) Case:

a) If <interval term 1> is a year-month interval, then <interval value expression 1> shall be a year-month interval.

b) If <interval term 1> is a day-time interval, then <interval value expression 1> shall be a day-time interval.

4) If <datetime value expression> is specified, then <datetime value expression> and <datetime term> shall be comparable.

5) An <interval primary> shall specify <interval qualifier> only if the <interval primary> specifies a <dynamic parameter specification>.

Access Rules

None.

General Rules

1) If an <interval term> specifies "<term> * <interval factor>", then let *T* and *F* be respectively the value of the <term> and the value of the <interval factor>. The result of the <interval term> is the result of *F* * *T*.

2) If the value of any <interval primary>, <datetime value expression>, <datetime term>, or <factor> that is simply contained in an <interval value expression> is the null value, then the result of the <interval value expression> is the null value.

3) If *IP* is an <interval primary>, then

Case:

a) If *IP* immediately contains a <value expression primary> *VEP* and an explicit <interval qualifier> *IQ*, then the value of *IP* is computed by:

```
CAST ( VEP AS INTERVAL IQ )
```

b) If *IP* immediately contains a <value expression primary> *VEP*, then the value of *IP* is the value of *VEP*.

c) If *IP* is an <interval value function> *IVF*, then the value of *IP* is the value of *IVF*.

4) If the <sign> of an <interval factor> is <minus sign>, then the value of the <interval factor> is the negative of the value of the <interval primary>; otherwise, the value of an <interval factor> is the value of the <interval primary>.

5) If <interval term 2> is specified, then:

a) Let *X* be the value of <interval term 2> and let *Y* be the value of <factor>.

b) Let *P* and *Q* be respectively the most significant and least significant <primary datetime field>s of <interval term 2>.

c) Let *E* be an exact numeric result of the operation

```
CAST ( CAST ( X AS INTERVAL Q ) AS E1 )
```

where *E1* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

d) Let *OP* be the operator * or / specified in the <interval value expression>.

e) Let *I*, the result of the <interval value expression> expressed in terms of the <primary datetime field> *Q*, be the result of

```
CAST ( (E OP Y) AS INTERVAL Q )
```

f) The result of the <interval value expression> is

```
CAST ( I AS INTERVAL W )
```

where *W* is an <interval qualifier> identifying the <primary datetime field>s *P* TO *Q*, but with <interval leading field precision> such that significant digits are not lost.

6) If <interval term 1> is specified, then let *P* and *Q* be respectively the most significant and least significant <primary datetime field>s in <interval term 1> and <interval value expression 1>, let *X* be the value of <interval value expression 1>, and let *Y* be the value of <interval term 1>.

a) Let *A* be an exact numeric result of the operation

```
CAST ( CAST ( X AS INTERVAL Q )
AS E1 )
```

where *E1* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

b) Let *B* be an exact numeric result of the operation

```
CAST ( CAST ( Y AS INTERVAL Q )
AS E2 )
```

where *E2* is an exact numeric data type of sufficient scale and precision so as to not lose significant digits.

c) Let *OP* be the operator + or − specified in the <interval value expression>.

d) Let *I*, the result of the <interval value expression> expressed in terms of the <primary datetime field> *Q*, be the result of:

```
CAST ( (A OP B) AS
INTERVAL Q )
```

e) The result of the <interval value expression> is

```
CAST ( I AS INTERVAL W )
```

where *W* is an <interval qualifier> identifying the <primary datetime field>s *P* TO *Q*, but with <interval leading field precision> such that significant digits are not lost.

7) If <datetime value expression> is specified, then let *Y* be the least significant <primary datetime field> specified by <interval qualifier>. Let *DTE* be the <datetime value expression>, let *DT* be the <datetime term>, and let *MSP* be the implementation-defined (IL045) maximum seconds precision. Evaluation of <interval value expression> proceeds as follows:

a) Case:

i) If the declared type of <datetime value expression> is TIME WITH TIME ZONE, then let *A* be the value of:

```
CAST ( DTE AT LOCAL
AS TIME(MSP) WITHOUT
TIME ZONE )
```

ii) If the declared type of <datetime value expression> is TIMESTAMP WITH TIME ZONE, then let *A* be the value of:

```
CAST ( DTE AT LOCAL
AS TIMESTAMP(MSP) WITHOUT
TIME ZONE )
```

iii) Otherwise, let *A* be the value of *DTE*.

b) Case:

i) If the declared type of <datetime term> is TIME WITH TIME ZONE, then let *B* be the value of:

```
CAST ( DT AT LOCAL
AS TIME(MSP) WITHOUT
TIME ZONE )
```

ii) If the declared type of <datetime term> is TIMESTAMP WITH TIME ZONE, then let *B* be the value of:

```
CAST ( DT AT LOCAL
AS TIMESTAMP(MSP) WITHOUT
TIME ZONE )
```

iii) Otherwise, let *B* be the value of *DT*.

c) *A* and *B* are converted to integer scalars *A2* and *B2* respectively in units *Y* as displacements from some implementation-dependent (UV075) start datetime.

d) The result is determined by effectively computing *A2*−*B2* and then converting the difference to an interval using an <interval qualifier> whose <end field> is *Y* and whose <start field> is sufficiently significant to avoid loss of significant digits. The difference of two values of type TIME (with or without time zone) is constrained to be between −24:00:00 and +24:00:00 (excluding each end point); it is implementation-defined (IA026) which of two non-zero values in this range is the result, although the computation shall be deterministic. That interval is then converted to an interval using the specified <interval qualifier>, rounding or truncating if necessary. The choice of whether to round or truncate is implementation-defined (IA002). If the required number of significant digits exceeds the implementation-defined (IL045) maximum number of significant digits, then an exception condition is raised: *data exception — interval field overflow (22015)*.

Conformance Rules

1) Without Feature F052, “Intervals and datetime arithmetic”, conforming SQL language shall not contain an <interval value expression>.

6.45 <interval value function>

Function

Specify a function yielding a value of type interval.

Format

```
<interval value function> ::=
  <interval absolute value function>

<interval absolute value function> ::=
  ABS <left paren> <interval value expression> <right paren>
```

Syntax Rules

1) If <interval absolute value function> is specified, then the declared type of the result is the declared type of the <interval value expression>.

Access Rules

None.

General Rules

1) If <interval absolute value function> is specified, then let *N* be the value of the <interval value expression>.

   Case:

   a) If *N* is the null value, then the result is the null value.

   b) If $N \geq 0$ (zero), then the result is *N*.

   c) Otherwise, the result is $-1 * N$.

Conformance Rules

1) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL shall not contain an <interval value function>.

6.46 <boolean value expression>

Function

Specify a Boolean value.

Format

```
<boolean value expression> ::=
    <boolean term>
  | <boolean value expression> OR <boolean term>

<boolean term> ::=
    <boolean factor>
  | <boolean term> AND <boolean factor>

<boolean factor> ::=
  [ NOT ] <boolean test>

<boolean test> ::=
  <boolean primary> [ IS [ NOT ] <truth value> ]

<truth value> ::=
    TRUE
  | FALSE
  | UNKNOWN

<boolean primary> ::=
    <predicate>
  | <boolean predicand>

<boolean predicand> ::=
    <parenthesized boolean value expression>
  | <non-parenthesized value expression primary>

<parenthesized boolean value expression> ::=
  <left paren> <boolean value expression> <right paren>
```

Syntax Rules

1) The declared type of a <boolean value expression> is Boolean.

2) The declared type of a <parenthesized boolean value expression> is Boolean.

3) The declared type of a <non-parenthesized value expression primary> shall be Boolean.

4) If NOT is specified in a <boolean test>, then let *BP* be the contained <boolean primary> and let *TV* be the contained <truth value>. The <boolean test> is equivalent to:

   ```
   ( NOT ( BP IS TV ) )
   ```

5) Let *X* denote either a column *C* or the <key word> VALUE. Given a <boolean value expression> *BVE* and *X*, the notion "*BVE* is a *readily-known-not-null condition* for *X*" is defined as follows.

   Case:

   a) If *BVE* is a <predicate> of the form "*RVE* IS NOT NULL", where *RVE* is a <row value predicand> that is a <row value constructor predicand> that simply contains a <common value expression>,

<boolean predicand>, or <row value constructor element> that is a <column reference> that references *C*, then *BVE* is a readily-known-not-null condition for *C*.

b) If *BVE* is the <predicate> "VALUE IS NOT NULL", then *BVE* is a readily-known-not-null condition for VALUE.

c) Otherwise, *BVE* is not a readily-known-not-null condition for *X*.

6) Let *X* denote either a column *C* or the <key word> VALUE. Given a <boolean value expression> *BVE* and *X*, the notion "*BVE* is a *known-not-null condition* for *X*" is defined recursively as follows:

a) If *BVE* is a <predicate>, then *BVE* is a known-not-null condition for *X* if *BVE* is a readily-known-not-null condition for *X*.

b) If the SQL-implementation supports Feature T101, "Enhanced nullability determination", then *BVE* is also recognized as a known-not-null condition for *X* according to the following recursive rules:

i) If *BVE* is a <parenthesized boolean value expression> and the simply contained <boolean value expression> is a known-not-null condition for *X*, then *BVE* is a known-not-null condition for *X*.

ii) If *BVE* is a <non-parenthesized value expression primary>, then *BVE* is not a known-not-null condition for *X*.

iii) If *BVE* is a <boolean test>, then let *BP* be the <boolean primary> immediately contained in *BVE*. If *BP* is a known-not-null condition for *X*, and <truth value> is not specified, then *BVE* is a known-not-null condition for *X*. Otherwise, *BVE* is not a known-not-null condition for *X*.

iv) If *BVE* is of the form "NOT *BT*", where *BT* is a <boolean test>, then

Case:

1) If *BT* is "*CR* IS NULL", where *CR* is a column reference that references column *C*, then *BVE* is a known-not-null condition for *C*.

2) If *BT* is "VALUE IS NULL", then *BVE* is a known-not-null condition for VALUE.

3) Otherwise, *BVE* is not a known-not-null condition for *X*.

NOTE 205 — For simplicity, this rule does not attempt to analyze conditions such as "NOT NOT *A* IS NULL", or "NOT (*A* IS NULL OR NOT (*B* = 2))"

v) If *BVE* is of the form "*BVE1* AND *BVE2*", then

Case:

1) If at least one of *BVE1* and *BVE2* is a known-not-null condition for *X*, then *BVE* is a known-not-null condition for *X*.

2) Otherwise, *BVE* is not a known-not-null condition for *X*.

vi) If *BVE* is of the form "*BVE1* OR *BVE2*", then *BVE* is not a known-not-null condition for *X*.

NOTE 206 — For simplicity, this rule does not detect cases such as "*A* IS NOT NULL OR *A* IS NOT NULL", which might be classified as a known-not-null condition.

c) If *BVE* conforms to an implementation-defined (IA216) rule that enables the SQL-implementation to correctly infer that, when *BVE* is *True*, then *X* cannot be null, then *BVE* is a *known-not-null condition* for *X*.

7) The notion of "retrospectively deterministic" is defined recursively as follows:

a) A <parenthesized boolean value expression> is retrospectively deterministic if the simply contained <boolean value expression> is retrospectively deterministic.

b) A <non-parenthesized value expression primary> is retrospectively deterministic if it does not contain a potential source of non-determinism.

c) A <predicate> *P* is *retrospectively deterministic* if exactly one of the following is true:

   i) *P* does not contain a potential source of non-determinism.

   ii) *P* is a <comparison predicate> of the form "*X* < *Y*", "*X* <= *Y*", "*Y* > *X*", "*Y* >= *X*", "*X* < *Y* + *Z*", "*X* <= *Y* + *Z*", "*Y* + *Z* > *X*", "*Y* + *Z* >= *X*", "*X* < *Y* − *Z*", "*X* <= *Y* − *Z*", "*Y* − *Z* > *X*", or "*Y* − *Z* >= *X*", where *Y* is CURRENT_DATE, CURRENT_TIMESTAMP or LOCALTIMESTAMP, *X* and *Z* do not contain a potential source of non-determinism, and the declared types of the left and right comparands are either both datetime with time zone or both datetime without time zone.

   iii) *P* is a <quantified comparison predicate> of the form "*Y* > <quantifier> <table subquery>", "*Y* + *Z* > <quantifier> <table subquery>", "*Y* − *Z* > <quantifier> <table subquery>", "*Y* >= <quantifier> <table subquery>", "*Y* + *Z* >= <quantifier> <table subquery>", or "*Y* - *Z* >= <quantifier> <table subquery>", where *Y* is CURRENT_DATE, CURRENT_TIMESTAMP or LOCALTIMESTAMP, *Z* does not contain a potential source of non-determinism, the <table subquery> does not contain a potential source of non-determinism, and the declared types of the left and right comparands are either both datetime with time zone or both datetime without time zone.

   iv) *P* is a <between predicate> that is transformed into a retrospectively deterministic <boolean value expression>.

d) A <boolean primary> is retrospectively deterministic if the simply contained <predicate>, <parenthesized boolean value expression> or <non-parenthesized value expression primary> is retrospectively deterministic.

e) Let *BF* be a <boolean factor>. Let *BP* be the <boolean primary> simply contained in *BF*.

   i) *BF* is called *negative* if *BF* is of any of the following forms:

```
NOT BP
BP IS FALSE
BP IS NOT TRUE
NOT BP IS NOT FALSE
NOT BP IS TRUE
```

   ii) *BF* is retrospectively deterministic if and only if exactly one of the following is true:

      1) *BF* is negative and *BF* does not contain a potential source of non-determinism.

      2) *BF* is not negative and *BP* is retrospectively deterministic.

f) A <boolean value expression> is retrospectively deterministic if every simply contained <boolean factor> is retrospectively deterministic.

Access Rules

None.

General Rules

1) The result is derived by the application of the specified Boolean operators ("AND", "OR", "NOT", and "IS") to the results derived from each <boolean primary>. If Boolean operators are not specified, then the result of the <boolean value expression> is the result of the specified <boolean primary>.

2) NOT (*True*) is *False*, NOT (*False*) is *True*, and NOT (*Unknown*) is *Unknown*.

3) Table 15, "Truth table for the AND Boolean operator", Table 16, "Truth table for the OR Boolean operator", and Table 17, "Truth table for the IS Boolean operator", specify the semantics of AND, OR, and IS, respectively.

Table 15 — Truth table for the AND Boolean operator

AND	***True***	***False***	***Unknown***
True	*True*	*False*	*Unknown*
False	*False*	*False*	*False*
Unknown	*Unknown*	*False*	*Unknown*

Table 16 — Truth table for the OR Boolean operator

OR	***True***	***False***	***Unknown***
True	*True*	*True*	*True*
False	*True*	*False*	*Unknown*
Unknown	*True*	*Unknown*	*Unknown*

Table 17 — Truth table for the IS Boolean operator

IS	TRUE	FALSE	UNKNOWN
True	*True*	*False*	*False*
False	*False*	*True*	*False*
Unknown	*False*	*False*	*True*

Conformance Rules

1) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <boolean primary> that simply contains a <non-parenthesized value expression primary>.

2) Without Feature F571, "Truth value tests", conforming SQL language shall not contain a <boolean test> that simply contains a <truth value>.

6.47 <array value expression>

Function

Specify an array value.

Format

```
<array value expression> ::=
    <array concatenation>
  | <array primary>

<array concatenation> ::=
  <array value expression 1> <concatenation operator> <array primary>

<array value expression 1> ::=
  <array value expression>

<array primary> ::=
    <array value function>
  | <value expression primary>
```

Syntax Rules

1) The declared type of the <array value expression> is the declared type of the immediately contained <array concatenation> or <array primary>.

2) The declared type of <array primary> is the declared type of the immediately contained <array value function> or <value expression primary>, which shall be an array type or a distinct type whose source type is an array type.

3) If <array concatenation> is specified, then:

   a) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of <array value expression 1> and <array primary> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules.

   b) Let *IMDC* be the implementation-defined (IL008) maximum cardinality of an array type.

   c) The declared type of the result of <array concatenation> is an array type whose element type is the element type of *DT* and whose maximum cardinality is the lesser of *IMDC* and the sum of the maximum cardinality of <array value expression 1> and the maximum cardinality of <array primary>.

Access Rules

None.

General Rules

1) The value of the result of <array value expression> is the value of the immediately contained <array concatenation> or <array primary>.

2) If <array concatenation> is specified, then let *AV1* be the value of <array value expression 1> and let *AV2* be the value of <array primary>.

Case:

a) If at least one of *AV1* and *AV2* is the null value, then the result of the <array concatenation> is the null value.

b) If the sum of the cardinality of *AV1* and the cardinality of *AV2* is greater than *IMDC*, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

c) Otherwise, the result is the array comprising every element of *AV1* followed by every element of *AV2*.

Conformance Rules

1) Without Feature S099, "Array expressions", conforming SQL language shall not contain an <array value expression>.

6.48 <array value function>

Function

Specify a function yielding a value of an array type.

Format

```
<array value function> ::=
  <trim array function>

<trim array function> ::=
  TRIM_ARRAY <left paren> <array value expression> <comma> <numeric value expression> <right
  paren>
```

Syntax Rules

1) The declared type of the <array value function> is the declared type of the immediately contained <trim array function>.

2) If <trim array function> is specified, then:

   a) The declared type of the <numeric value expression> shall be an exact numeric type with scale 0 (zero).

   b) The declared type of the <trim array function> is the declared type of the immediately contained <array value expression>.

Access Rules

None.

General Rules

1) The value of the <array value function> is the value of the immediately contained <trim array function>.

2) The result of <trim array function> is determined as follows:

   a) Let *NV* be the value of the <numeric value expression>.

   b) If *NV* is the null value, then the result is the null value and no further General Rules of this Subclause are applied.

   c) If *NV* is less than 0 (zero), then an exception condition is raised: *data exception — array element error (2202E)*.

   d) Let *AV* be the value of the <array value expression>.

   e) If *AV* is the null value, then the result is the null value and no further General Rules of this Subclause are applied.

   f) Let *AC* be the cardinality of *AV*.

   g) If *NV* is greater than *AC*, then an exception condition is raised: *data exception — array element error (2202E)*.

h) Let *N* be *AC* − *NV*.

i) Case:

i) If *N* = 0 (zero), then the result is an array whose cardinality is 0 (zero).

ii) Otherwise, the result is an array of *N* elements such that for all *i*, 1 (one) ≤ *i* ≤ *N*, the value of the *i*-th element of the result is the value of the *i*-th element of *AV*.

Conformance Rules

1) Without Feature S404, "TRIM_ARRAY", conforming SQL language shall not contain a <trim array function>.

6.49 <array value constructor>

Function

Specify construction of an array.

Format

```
<array value constructor> ::=
    <array value constructor by enumeration>
  | <array value constructor by query>

<array value constructor by enumeration> ::=
  ARRAY <left bracket or trigraph> <array element list> <right bracket or trigraph>

<array element list> ::=
  <array element> [ { <comma> <array element> }... ]

<array element> ::=
  <value expression>

<array value constructor by query> ::=
  ARRAY <table subquery>
```

Syntax Rules

1) The declared type of <array value constructor> is the declared type of the immediately contained <array value constructor by enumeration> or <array value constructor by query>.

2) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <array element>s immediately contained in the <array element list> of the <array value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules. The declared type of the <array value constructor by enumeration> is an array type with element type *DT*. The maximum cardinality of the <array value constructor by enumeration> is the number of <array element>s in the <array element list>, which shall not be greater than the implementation-defined (IL008) maximum cardinality for array types whose element type is *DT*.

3) If <array value constructor by query> is specified, then:

   a) The <query expression> *QE* simply contained in the <table subquery> shall be of degree 1 (one). Let *ET* be the declared type of the column in the result of <query expression>.

   b) The declared type of the <array value constructor by query> is array with element type *ET* and maximum cardinality equal to the implementation-defined (IL008) maximum cardinality *IMDC* for such array types.

Access Rules

None.

General Rules

1) The value of <array value constructor> is the value of the immediately contained <array value constructor by enumeration> or <array value constructor by query>.

2) The result of <array value constructor by enumeration> is an array whose *i*-th element is the value of the *i*-th <array element> immediately contained in the <array element list>, cast as the data type of *DT*.

3) The result of <array value constructor by query> is determined as follows:

   a) *QE* is evaluated, producing a table *T*. Let *N* be the number of rows in *T*.

   b) If *N* is greater than *IMDC*, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

   c) The result of <array value constructor by query> is an array of *N* elements such that for all *i*, 1 (one) ≤ *i* ≤ *N*, the value of the *i*-th element is the value of the only column in the *i*-th row of *T*.

      NOTE 207 — The ordering of the array elements is effectively determined by the General Rules of Subclause 7.17, "<query expression>".

Conformance Rules

1) Without Feature S090, "Minimal array support", conforming SQL language shall not contain an <array value constructor by enumeration>.

2) Without Feature S095, "Array constructors by query", conforming SQL language shall not contain an <array value constructor by query>.

6.50 <multiset value expression>

Function

Specify a multiset value.

Format

```
<multiset value expression> ::=
    <multiset term>
  | <multiset value expression> MULTISET UNION [ ALL | DISTINCT ] <multiset term>
  | <multiset value expression> MULTISET EXCEPT [ ALL | DISTINCT ] <multiset term>

<multiset term> ::=
    <multiset primary>
  | <multiset term> MULTISET INTERSECT [ ALL | DISTINCT ] <multiset primary>

<multiset primary> ::=
    <multiset value function>
  | <value expression primary>
```

Syntax Rules

1) The declared type of a <multiset primary> is the declared type of the immediately contained <multiset value function> or <value expression primary>, which shall be a multiset type or a distinct type whose source type is a multiset type.

2) If *MI* is a <multiset term> that immediately contains MULTISET INTERSECT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>).

   a) *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

   b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *ET1* and *ET2* as *DTSET*; let *ET* be the *REST.TYPE* returned from the application of those Syntax Rules. The result type of the MULTISET INTERSECT operation is multiset with element type *ET*.

   c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

   d) *MI* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       INTERSECT SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

3) If *MU* is a <multiset value expression> that immediately contains MULTISET UNION, then let *OP1* be the first operand (the <multiset value expression>) and let *OP2* be the second operand (the <multiset term>).

a) If DISTINCT is specified, then *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *ET1* and *ET2* as *DTSET*; let *ET* be the *RESTYPE* returned from the application of those Syntax Rules. The result type of the MULTISET UNION operation is multiset with element type *ET*.

c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

d) *MU* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       UNION SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

4) If *ME* is a <multiset value expression> that immediately contains MULTISET EXCEPT, then let *OP1* be the first operand (the <multiset value expression>) and let *OP2* be the second operand (the <multiset term>).

a) *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

b) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *ET1* and *ET2* as *DTSET*; let *ET* be the *RESTYPE* returned from the application of those Syntax Rules. The result type of the MULTISET EXCEPT operation is multiset with element type *ET*.

c) If DISTINCT is specified, then let *SQ* be DISTINCT. Otherwise, let *SQ* be ALL.

d) *ME* is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
       ELSE MULTISET ( SELECT T1.V
                         FROM UNNEST (OP1) AS T1(V)
                       EXCEPT SQ
                         SELECT T2.V
                         FROM UNNEST (OP2) AS T2(V)
                       )
  END )
```

Access Rules

None.

General Rules

1) The value of a <multiset primary> is the value of the immediately contained <multiset value function> or <value expression primary>.

2) The value of a <multiset term> that is a <multiset primary> is the value of the <multiset primary>.

3) The value of a <multiset value expression> that is a <multiset term> is the value of <multiset term>.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset value expression>.

2) Without Feature S275, "Advanced multiset support", conforming SQL language shall not contain MULTISET UNION, MULTISET INTERSECTION, or MULTISET EXCEPT.

NOTE 208 — If MULTISET UNION DISTINCT, MULTISET INTERSECTION, or MULTISET EXCEPT is specified, then the Conformance Rules of Subclause 9.13, "Multiset element grouping operations", also apply.

6.51 <multiset value function>

Function

Specify a function yielding a value of a multiset type.

Format

```
<multiset value function> ::=
  <multiset set function>

<multiset set function> ::=
  SET <left paren> <multiset value expression> <right paren>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression> simply contained in <multiset set function>. *MVE* is a multiset operand of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

2) The <multiset set function> is equivalent to

```
( CASE WHEN MVE IS NULL THEN NULL
       ELSE MULTISET ( SELECT DISTINCT M.E
                       FROM UNNEST (MVE) AS M(E) )
  END )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset value function>.

NOTE 209 — The Conformance Rules of Subclause 9.13, "Multiset element grouping operations", also apply.

6.52 <multiset value constructor>

Function

Specify construction of a multiset.

Format

```
<multiset value constructor> ::=
    <multiset value constructor by enumeration>
  | <multiset value constructor by query>
  | <table value constructor by query>

<multiset value constructor by enumeration> ::=
  MULTISET <left bracket or trigraph> <multiset element list> <right bracket or trigraph>

<multiset element list> ::=
  <multiset element> [ { <comma> <multiset element> }... ]

<multiset element> ::=
  <value expression>

<multiset value constructor by query> ::=
  MULTISET <table subquery>

<table value constructor by query> ::=
  TABLE <table subquery>
```

Syntax Rules

1) If <multiset value constructor> immediately contains a <table value constructor by query> *TVCBQ*, then:

   a) Let *QE* be the <query expression> simply contained in *TVCBQ*.

   b) *QE* shall not immediately contain an <order by clause>.

   c) Let *n* be the number of columns in the result of *QE*.

   d) Let C_1, ..., C_n be implementation-dependent (UV076) identifiers that are all distinct from one another.

   e) *TVCBQ* is equivalent to

```
MULTISET ( SELECT ROW ( C1, ..., Cn )
             FROM ( QE ) AS T ( C1, ..., Cn ) )
```

2) The declared type of <multiset value constructor> is the declared type of the immediately contained <multiset value constructor by enumeration> or <multiset value constructor by query>.

3) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <multiset element>s immediately contained in the <multiset element list> of the <multiset value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those Syntax Rules. The declared type of the <multiset value constructor by enumeration> is a multiset type with element type *DT*.

4) If <multiset value constructor by query> is specified, then:

   a) The <table subquery> *TS* shall be of degree 1 (one).

b) *TS* shall not immediately contain an <order by clause>.

c) Let *ET* be the declared type of the column in the result of *TS*.

d) The declared type of the <multiset value constructor by query> is multiset with element type *ET*.

Access Rules

None.

General Rules

1) The value of <multiset value constructor> is the value of the immediately contained <multiset value constructor by enumeration> or <multiset value constructor by query>.

2) The result of <multiset value constructor by enumeration> is a multiset whose elements are the values of the <multiset element>s immediately contained in the <multiset element list>, cast as the data type of *DT*.

3) If <multiset value constructor by query> is specified, then:

   a) Let *T* be the value of *TS*. Let *N* be the number of rows in *T*.

   b) The result of <multiset value constructor by query> is a multiset of *N* elements, with one element for each row of *T*, where the value of each element is the value of the only column in the corresponding row of *T*.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <multiset value constructor>.

2) Without Feature T326, "Table functions", in conforming SQL language, a <multiset value constructor> shall not contain a <table value constructor by query>.

7 Query expressions

This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-4.
This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-9.
This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-14.
This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-15.
This Clause is modified by Clause 7, "Query expressions", in ISO/IEC 9075-16.

7.1 <row value constructor>

Function

Specify a value or list of values to be constructed into a row.

Format

```
<row value constructor> ::=
    <common value expression>
  | <boolean value expression>
  | <explicit row value constructor>

<explicit row value constructor> ::=
    <left paren> <row value constructor element> <comma>
        <row value constructor element list> <right paren>
  | ROW <left paren> <row value constructor element list> <right paren>
  | <row subquery>

<row value constructor element list> ::=
  <row value constructor element> [ { <comma> <row value constructor element> }... ]

<row value constructor element> ::=
  <value expression>

<contextually typed row value constructor> ::=
    <common value expression>
  | <boolean value expression>
  | <contextually typed value specification>
  | <left paren> <contextually typed value specification> <right paren>
  | <left paren> <contextually typed row value constructor element> <comma>
        <contextually typed row value constructor element list> <right paren>
  | ROW <left paren> <contextually typed row value constructor element list> <right paren>

<contextually typed row value constructor element list> ::=
  <contextually typed row value constructor element>
      [ { <comma> <contextually typed row value constructor element> }... ]

<contextually typed row value constructor element> ::=
    <value expression>
  | <contextually typed value specification>

<row value constructor predicand> ::=
    <common value expression>
  | <boolean predicand>
  | <explicit row value constructor>
```

Syntax Rules

1) If a <row value constructor> is a <common value expression> or a <boolean value expression> *X*, then the <row value constructor> is equivalent to

```
ROW ( X )
```

2) If a <row value constructor predicand> is a <common value expression> or a <boolean predicand> *X*, then the <row value constructor predicand> is equivalent to

```
ROW ( X )
```

3) Let *ERVC* be an <explicit row value constructor>.

Case:

a) If *ERVC* simply contains a <row subquery>, then the declared type of *ERVC* is the declared type of that <row subquery>.

b) Otherwise, the declared type of *ERVC* is a row type described by a sequence of (<field name>, <data type>) pairs, corresponding in order to each <row value constructor element> *X* simply contained in *ERVC*. The <data type> is the declared type of *X* and the <field name> is implementation-dependent (UV077).

4) If a <row value constructor> or <row value constructor predicand> *RVC* is an <explicit row value constructor> *ERVC*, then the declared type of *RVC* is the declared type of *ERVC*.

5) Let *CTRVC* be the <contextually typed row value constructor>.

a) If *CTRVC* is a <common value expression>, <boolean value expression>, or <contextually typed value specification> *X*, then *CTRVC* is equivalent to:

```
ROW ( X )
```

b) After the syntactic transformation specified in SR 5)a) has been performed, if necessary, the declared type of *CTRVC* is a row type described by a sequence of (<field name>, <data type>) pairs, corresponding in order to each <contextually typed row value constructor element> *X* simply contained in *CTRVC*. The <data type> is the declared type of *X* and the <field name> is implementation-dependent (UV078).

6) The degree of a <row value constructor>, <contextually typed row value constructor>, or <row value constructor predicand> is the degree of its declared type.

Access Rules

None.

General Rules

1) The value of a <null specification> is the null value.

2) The value of a <default specification> is determined according to the General Rules of Subclause 11.5, "<default clause>".

3) Case:

a) If a <row value constructor>, <row value constructor predicand>, or <contextually typed row value constructor> immediately contains a <common value expression>, <boolean value

expression>, or <contextually typed row value constructor element> *X*, then the result of the <row value constructor>, <row value constructor predicand>, or <contextually typed row value constructor> is a row containing a single column whose value is the value of *X*.

b) If an <explicit row value constructor> is specified, then the result of the <row value constructor> or <row value constructor predicand> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <row value constructor element> simply contained in the <explicit row value constructor>.

c) If a <contextually typed row value constructor element list> is specified, then the result of the <contextually typed row value constructor> is a row of columns, the value of whose *i*-th column is the value of the *i*-th <contextually typed row value constructor element> in the <contextually typed row value constructor element list>.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain an <explicit row value constructor> that immediately contains ROW.

2) Without Feature T051, "Row types", conforming SQL language shall not contain a <contextually typed row value constructor> that immediately contains ROW.

3) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain an <explicit row value constructor> that is not simply contained in a <table value constructor> and that contains more than one <row value constructor element>.

4) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain an <explicit row value constructor> that is a <row subquery>.

5) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <row value constructor predicand> that immediately contains a <boolean predicand>.

6) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <contextually typed row value constructor> that is not simply contained in a <contextually typed table value constructor> and that contains more than one <row value constructor element>.

7.2 <row value expression>

Function

Specify a row value.

Format

```
<row value expression> ::=
    <row value special case>
  | <explicit row value constructor>

<table row value expression> ::=
    <row value special case>
  | <row value constructor>

<contextually typed row value expression> ::=
    <row value special case>
  | <contextually typed row value constructor>

<row value predicand> ::=
    <row value special case>
  | <row value constructor predicand>

<row value special case> ::=
  <non-parenthesized value expression primary>
```

Syntax Rules

1) The declared type of a <row value special case> shall be a row type.

2) The declared type of a <row value expression> is the declared type of the immediately contained <row value special case> or <explicit row value constructor>.

3) The declared type of a <table row value expression> is the declared type of the immediately contained <row value special case> or <row value constructor>.

4) The declared type of a <contextually typed row value expression> is the declared type of the immediately contained <row value special case> or <contextually typed row value constructor>. The declared type of a <row value predicand> is the declared type of the immediately contained <row value special case> or <row value constructor predicand>.

Access Rules

None.

General Rules

1) A <row value special case> specifies the row value denoted by the <non-parenthesized value expression primary>.

2) A <row value expression> specifies the row value denoted by the <row value special case> or <explicit row value constructor>.

3) A <table row value expression> specifies the row value denoted by the <row value special case> or <row value constructor>.

4) A <contextually typed row value expression> specifies the row value denoted by the <row value special case> or <contextually typed row value constructor>.

5) A <row value predicand> specifies the row value denoted by the <row value special case> or <row value constructor predicand>.

Conformance Rules

1) Without Feature T051, "Row types", conforming SQL language shall not contain a <row value special case>.

7.3 <table value constructor>

Function

Specify a set of <row value expression>s to be constructed into a table.

Format

```
<table value constructor> ::=
  VALUES <row value expression list>

<row value expression list> ::=
  <table row value expression> [ { <comma> <table row value expression> }... ]

<contextually typed table value constructor> ::=
  VALUES <contextually typed row value expression list>

<contextually typed row value expression list> ::=
  <contextually typed row value expression>
      [ { <comma> <contextually typed row value expression> }... ]
```

Syntax Rules

1) All <table row value expression>s immediately contained in a <row value expression list> shall be of the same degree.

2) All <contextually typed row value expression>s immediately contained in a <contextually typed row value expression list> shall be of the same degree.

3) Let *TVC* be some <table value constructor> consisting of *n* <table row value expression>s or some <contextually typed table value constructor> consisting of *n* <contextually typed row value expression>s. Let RVE_i, 1 (one) ≤ *i* ≤ *n*, denote the *i*-th <table row value expression> or the *i*-th <contextually typed row value expression>. The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the row types RVE_i, 1 (one) ≤ *i* ≤ *n* as *DTSET*; let the row type of *TVC* be the *RESTYPE* returned from the application of those Syntax Rules. The column names are implementation-dependent (UV079).

Access Rules

None.

General Rules

1) If the result of any <table row value expression> or <contextually typed row value expression> is the null value, then an exception condition is raised: *data exception — null row not permitted in table (2201C)*.

2) The result *T* of a <table value constructor> or <contextually typed table value constructor> *TVC* is a table whose cardinality is the number of <table row value expression>s or the number of <contextually typed row value expression>s in *TVC*. If *R* is the result of *n* such expressions, then *R* occurs *n* times in *T*.

Conformance Rules

1) Without Feature F641, "Row and table constructors", in conforming SQL language, the <contextually typed row value expression list> of a <contextually typed table value constructor> shall contain exactly one <contextually typed row value constructor> *RVE*. *RVE* shall be of the form "(<contextually typed row value constructor element list>)".

2) Without Feature F641, "Row and table constructors", conforming SQL language shall not contain a <table value constructor>.

7.4 <table expression>

Function

Specify a table or a grouped table.

Format

```
<table expression> ::=
  <from clause>
      [ <where clause> ]
      [ <group by clause> ]
      [ <having clause> ]
      [ <window clause> ]
```

Syntax Rules

1) The result of a <table expression> is a derived table whose row type is the row type of the result of the immediately contained <from clause>, together with the window structure descriptors defined by the <window clause>, if specified.

2) Let *C* be some column. Let *TE* be the <table expression>. *C* is an *underlying column* of *TE* if *C* is an underlying column of some column reference contained in *TE*. *C* is a *generally underlying column* of *TE* if *C* is a generally underlying column of some column reference contained in *TE*.

Access Rules

None.

General Rules

1) If all optional clauses are omitted, then the result of the <table expression> is the same as the result of the <from clause>. Otherwise, each specified clause is applied to the result of the previously specified clause and the result of the <table expression> is the result of the application of the last specified clause.

Conformance Rules

None.

7.5 <from clause>

Function

Specify a table derived from one or more tables.

Format

```
<from clause> ::=
  FROM <table reference list>

<table reference list> ::=
  <table reference> [ { <comma> <table reference> }... ]
```

Syntax Rules

1) Let *FC* be a <from clause>. Let *FCT* be the table specified by *FC*. Let *TRL* be the <table reference list> immediately contained in *FC*.

2) No element TR_i in *TRL* shall contain an outer reference to an element TR_j, where $i \leq j$.

3) If an element in *TRL* generally contains a <data change delta table>, then there shall be exactly one element in *TRL*.

4) Case:

   a) If *TRL* immediately contains a single <table reference> *TR*, then:

      i) The descriptors of the columns of *FCT* are the same as the descriptors of the columns of the table specified by *TR*.

      ii) The row type of *FCT* is the row type of the table specified by *TR*.

   b) Otherwise:

      i) The descriptors of the columns of *FCT* are the same as the descriptors of the columns of the tables specified by the <table reference>s, in the order in which the <table reference>s appear in *TRL* and in the order in which the columns are defined within each table.

      ii) The row type *RT* of *FCT* is determined by the sequence *SCD* of column descriptors of *FCT* as follows:

         1) Let *n* be the number of column descriptors in *SCD*. *RT* has *n* fields.

         2) For *i* ranging from 1 (one) to *n*, the field name of the *i*-th field descriptor in *RT* is the column name included in the *i*-th column descriptor in *SCD*.

         3) For *i* ranging from 1 (one) to *n*, the data type descriptor of the *i*-th field descriptor in *RT* is

            Case:

            A) If the *i*-th column descriptor in *SCD* includes a domain name *DN*, then the data type descriptor included in the descriptor of the domain identified by *DN*.

B) Otherwise, the data type descriptor included in the *i*-th column descriptor in *SCD*.

5) Range variable *RV1* is *in conflict with* another range variable *RV2* if any of the following are true:

a) *RV1* and *RV2* are equivalent <correlation name>s.

b) *RV1* is a <correlation name> and *RV2* is a <table name> and *RV1* is equivalent to the <qualified identifier> of *RV2*.

c) *RV1* is a <table name> and *RV2* is a <correlation name> and the <qualified identifier> of *RV1* is equivalent to *RV2*.

d) *RV1* and *RV2* are equivalent <table name>s.

6) Let *NT* be the number of <table reference>s immediately contained in *TRL*. Let NR_0 be 0 (zero).

a) For *i* ranging from 1 (one) to *NT*:

i) Let TR_i be the *i*-th <table reference> in *TRL*. Let NTR_i be the number of range variables of TR_i.

ii) For *j* ranging from 1 (one) to NTR_i, the name of the $(NR_{i-1}+j)$-th range variable of *FC* is same as the *j*-th range variable of TR_i.

iii) The associated column list of $(NR_{i-1}+j)$-th range variable of *FC* is the same as the associated column list of the *j*-th range variable of TR_i.

iv) The associated period list of $(NR_{i-1}+j)$-th range variable of *FC* is the same as the associated period list of the *j*-th range variable of TR_i.

v) Let NR_i be $(NR_{i-1} + NTR_i)$.

b) The number of range variables of *FC* is NR_{NT}.

c) Let *RV* be a range variable of *FC*. *RV* shall not be in conflict with any other range variable of *FC*.

d) The table associated with each of the range variables of *FC* is *FCT*.

e) The *scope clause SC* of *FC* is the <select statement: single row> or innermost <query specification> that contains *FC* excluding any <row pattern measures> or <row pattern definition list> contained in any <window clause>.

f) The scope of a range variable of *FC* is the <select list>, <where clause>, <group by clause>, <having clause>, and <window clause> of *SC*. If *SC* is the <query specification> that is the <query expression body> of a simple table query *STQ*, then the scope of a range variable of *FC* also includes the <order by clause> of *STQ*.

NOTE 210 — "simple table query" is defined in Subclause 7.17, "<query expression>".

g) Every range variable of *FC* is *exposed* by *FC*.

Access Rules

None.

General Rules

1) Let *TRLR* be the result of *TRL*.

   Case:

   a) If *TRL* simply contains a single <table reference> *TR*, then *TRLR* is the result of *TR*.

   b) If *TRL* simply contains *n* <table reference>s, where $n > 1$, then let *TRL-P* be the <table reference list> formed by taking the first $n-1$ elements of *TRL* in order, let *TRL-L* be the last element of *TRL*, and let *TRLR-P* be the result of *TRL-P*.

      Case:

      i) If *TRLR-P* is empty, then *TRLR* is empty.

      ii) If *TRLR-P* contains *m* rows, $m \geq 1$ (one), then for every row R_i, 1 (one) $\leq i \leq m$, let *TRLR-L*$_i$ be the corresponding evaluation of *TRL-L* under all outer references contained in *TRL-L*. Let *SUBR*$_i$ be the table containing every row formed by concatenating R_i with some row of *TRLR-L*$_i$. Every row *RR* in *SUBR*$_i$ is a row in *TRLR*, and the number of occurrences of *RR* in *TRLR* is the sum of the numbers of occurrences of *RR* in every occurrence of *SUBR*$_i$.

   The result of the <table reference list> is *TRLR* with the columns reordered according to the ordering of the descriptors of the columns of the <table reference list>.

2) The result of the <from clause> is *TRLR*.

Conformance Rules

None.

7.6 <table reference>

This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 7.1, "<table reference>", in ISO/IEC 9075-16.

Function

Reference a table.

Format

```
<table reference> ::=
    <table factor>
  | <joined table>

<table factor> ::=
  <table primary> [ <sample clause> ]

<sample clause> ::=
  TABLESAMPLE <sample method> <left paren> <sample percentage> <right paren>
      [ <repeatable clause> ]

<sample method> ::=
    BERNOULLI
  | SYSTEM

<repeatable clause> ::=
  REPEATABLE <left paren> <repeat argument> <right paren>

<sample percentage> ::=
  <numeric value expression>

<repeat argument> ::=
  <numeric value expression>

14 15 16 <table primary> ::=
    <table or query name>
        [ <query system time period specification> ]
        [ <correlation or recognition> ]
  | <derived table> <correlation or recognition>
  | <lateral derived table> <correlation or recognition>
  | <collection derived table> <correlation or recognition>
  | <table function derived table> <correlation or recognition>
  | <PTF derived table> [ <correlation or recognition> ]
  | <only spec> [ <correlation or recognition> ]
  | <data change delta table> [ <correlation or recognition> ]
  | <JSON table> <correlation or recognition>
  | <JSON table primitive> <correlation name>
  | <parenthesized joined table>

<correlation or recognition> ::=
    [ AS ] <correlation name>
        [ <parenthesized derived column list> ]
  | <row pattern recognition clause and name>

<query system time period specification> ::=
    FOR SYSTEM_TIME AS OF <point in time 1>
```

```
  | FOR SYSTEM_TIME BETWEEN [ ASYMMETRIC | SYMMETRIC ]
        <point in time 1> AND <point in time 2>
  | FOR SYSTEM_TIME FROM <point in time 1> TO <point in time 2>

<point in time 1> ::=
  <point in time>

<point in time 2> ::=
  <point in time>

<point in time> ::=
  <datetime value expression>

<only spec> ::=
  ONLY <left paren> <table or query name> <right paren>

<lateral derived table> ::=
  LATERAL <table subquery>

<collection derived table> ::=
  UNNEST <left paren> <collection value expression>
      [ { <comma> <collection value expression> }... ] <right paren>
      [ WITH ORDINALITY ]

<table function derived table> ::=
  TABLE <left paren> <collection value expression> <right paren>

<derived table> ::=
  <table subquery>

<PTF derived table> ::=
  TABLE <left paren> <routine invocation> <right paren>

<table or query name> ::=
    <table name>
  | <transition table name>
  | <query name>

<derived column list> ::=
  <column name list>

<column name list> ::=
  <column name> [ { <comma> <column name> }... ]

<data change delta table> ::=
  <result option> TABLE <left paren> <data change statement> <right paren>

<data change statement> ::=
    <delete statement: searched>
  | <insert statement>
  | <merge statement>
  | <update statement: searched>

<result option> ::=
    FINAL
  | NEW
  | OLD

<parenthesized joined table> ::=
    <left paren> <parenthesized joined table> <right paren>
  | <left paren> <joined table> <right paren>

<row pattern recognition clause and name> ::=
  [ [ AS ] <row pattern input name>
      [ <row pattern input derived column list> ] ]
```

```
      <row pattern recognition clause> [ [ AS ] <row pattern output name>
      [ <row pattern output derived column list> ] ]

<row pattern input name> ::=
  <correlation name>

<row pattern output name> ::=
  <correlation name>

<row pattern input derived column list> ::=
  <parenthesized derived column list>

<row pattern output derived column list> ::=
  <parenthesized derived column list>

<parenthesized derived column list> ::=
  <left paren> <derived column list> <right paren>
```

Syntax Rules

1) The declared type of <point in time> shall be either DATE or a timestamp type.

2) The declared type of <repeat argument> shall be an exact numeric type with scale 0 (zero).

3) Let *TR* be a <table reference>, let *TF* be the <table factor> that is immediately contained in *TR*, and let *TP* be the <table primary> that is immediately contained in *TF*. Let *TPT* be the table specified by *TP*. Let *TFT* be the table specified by *TF*. Let *TRT* be the table specified by *TR*.

4) *TF* shall not simply contain both a <sample clause> and a <row pattern recognition clause and name>.

5) 14 16 A <table primary> that simply contains any of the following *supports lateral join*: <lateral derived table>, <collection derived table>, <table function derived table>, or <JSON table>.

6) If *TP* simply contains a <table function derived table> *TFDT*, then:

   a) The <collection value expression> immediately contained in *TFDT* shall be a <routine invocation> whose subject routine is not a polymorphic table function.

   b) Let *CN* be the <correlation name> simply contained in *TP*.

   c) Let *CVE* be the <collection value expression> simply contained in *TP*.

   d) Case:

      i) If *TP* specifies a <derived column list> *DCL*, then let *TFDCL* be

         ```
         ( DCL )
         ```

      ii) Otherwise, let *TFDCL* be the zero-length character string.

   e) *TP* is equivalent to the <table primary>

      ```
      UNNEST ( CVE ) AS CN TFDCL
      ```

7) If *TP* simply contains a <PTF derived table>, then:

   a) The subject routine *SR* of the <routine invocation> *RI* simply contained in *TP* shall be a polymorphic table function.

   b) *TF* shall not simply contain a <sample clause>.

   c) Let *PTFDA* be the PTF data area of *RI*. Let *IRR* be the initial result row descriptor in *PTFDA*.

d) Case:

i) If *SR* specifies RETURNS ONLY PASS THROUGH, then *TP* shall not simply contain <correlation or recognition>.

ii) Otherwise, *TP* shall contain <correlation or recognition> *PTFCOR* that immediately contains a <correlation name>. Let *PTFCN* be that <correlation name>.

1) *PTFCN* shall not be equivalent to any range variable exposed by any <table argument> of *RI*.

2) Let *NIRR* be the number of SQL item descriptor areas in *IRR* whose LEVEL component is 0 (zero). Let $SIDA_d$, 1 (one) ≤ d ≤ *NIRR*, be an enumeration of the SQL item descriptors of *IRR* whose LEVEL component is 0 (zero). For all d, 1 (one) ≤ d ≤ *NIRR*, let $NIDA_d$ be the NAME component of $SIDA_d$ and let $DIDA_d$ be the declared type matched by $SIDA_d$.

3) Case:

A) If *PTFCOR* contains a <derived column list> *PTFDCL*, then:

I) The number of <column name>s in *PTFDCL* shall be *NIRR*.

II) For all d, 1 (one) ≤ d ≤ *NIRR*, let $NDCL_d$ be the d-th <column name> in *PTFDCL*.

III) The *external result row type* of *TP* is described by the sequence of ($NDCL_d$, $DIDA_d$) pairs, for all d, 1 (one) ≤ d ≤ *NIRR*.

B) Otherwise, the *external result row type* of *TP* is described by the sequence of ($NIDA_d$, $DIDA_d$) pairs, for all d, 1 (one) ≤ d ≤ *NIRR*.

4) There shall not be any duplicate column names in the external result row type of *TP*.

8) 13 15 If *TP* simply contains a <collection derived table> *CDT*, then:

a) Let *NCV* be the number of <collection value expression>s simply contained in *CDT*.

b) Let CVE_j, 1 (one) ≤ j ≤ *NCV*, be an enumeration of the <collection value expression>s simply contained in *CDT*, in order from left to right.

c) Let ET_j, 1 (one) ≤ j ≤ *NCV*, be the element type of the declared type of CVE_j.

d) Let *CN* be the <correlation name> simply contained in *TP*, and let *TEMP* be an <identifier> that is not equivalent to *CN* nor to any other <identifier> contained in *TP*.

e) Case:

i) If the declared type of any CVE_j, 1 (one) ≤ j ≤ *NCV*, is a multiset, then *NCV* shall be 1 (one) and WITH ORDINALITY shall not be specified. Let $IMDC_1$ be the implementation-defined (IL008) maximum cardinality of an array whose declared element type is ET_1. Let C_1 be

```
( CAST (CVE1 AS ET1 ARRAY[IMDC1]) )
```

ii) Otherwise, let C_j, 1 (one) ≤ j ≤ *NCV*, be CVE_j.

f) Let $CARD_j$, 1 (one) ≤ j ≤ *NCV*, be

```
COALESCE( CARDINALITY( Cj ), 0 )
```

g) Let $MCARD_1$ be $CARD_1$. Let $MCARD_j$, $2 \le j \le NCV$, be

```
CASE
  WHEN CARDj > MCARDj-1
    THEN CARDj
  ELSE MCARDj-1
END
```

h) Let *MAXCARD* be $MCARD_{NCV}$.

i) Let N_j, 1 (one) $\le j \le NCV$, and *NORD* be *NCV*+1 <column name>s that are not equivalent to one another nor to *CN*, *TEMP*, or any other <identifier> contained in *TP*.

j) Let $ELT1_j$, 1 (one) $\le j \le NCV$, be

```
CASE
  WHEN 1 <= CARDINALITY( Cj )
    THEN Cj[1]
  ELSE NULL
END
```

k) Let $ELTNORD_j$, 1 (one) $\le j \le NCV$, be

```
CASE
  WHEN NORD < CARDINALITY( Cj )
    THEN Cj[NORD+1]
  ELSE NULL
END
```

l) Let *RECQP* be:

```
WITH RECURSIVE TEMP(N1, ..., NNCV, NORD) AS
  (   SELECT ELT11 AS N1, ..., ELT1NCV AS NNCV, 1 AS NORD
      FROM (VALUES(1)) AS CN
      WHERE 0 < MAXCARD
    UNION
      SELECT ELTNORD1 AS N1, ..., ELTNORDNCV AS NNCV, NORD+1 AS NORD
      FROM TEMP
      WHERE NORD < MAXCARD
  )
```

m) Case:

i) If *TP* specifies a <derived column list> *DCL*, then:

1) For all *j* between 1 (one) and *NCV*,

Case:

A) If ET_j is a row type, then let DET_j be the degree of ET_j.

B) Otherwise, let DET_j be 1 (one).

2) Case:

A) If *CDT* specifies WITH ORDINALITY, then *DCL* shall contain DET_1 + ... + DET_{NCV} + 1 <column name>s.

B) Otherwise, *DCL* shall contain DET_1 + ... + DET_{NCV} <column name>s.

3) Let *PDCLP* be

```
( DCL )
```

ii) Otherwise,

Case:

1) If any ET_j, 1 (one) ≤ j ≤ *NCV*, is a row type, then:

A) For each ET_j that is a row type:

I) Let *DET*(*j*) be the degree of ET_j.

II) Let $FN_{j,i}$, 1 (one) ≤ i ≤ *DET*(*j*), be the name of the *i*-th field in ET_j.

B) For each ET_j that is not a row type, let *DET*(*j*) be 1 (one) and let $FN_{j,1}$ be an <identifier> that is not equivalent to any other <identifier> contained in *TP*.

C) Case:

I) If *CDT* specifies WITH ORDINALITY, then let *PDCLP* be:

```
( FN1,1, FN1,2, ..., FN1,DET(1),
  ...,
  FNNCV,1, FNNCV,2, ..., FNNCV,DET(NCV), NORD
)
```

II) Otherwise, let *PDCLP* be:

```
( FN1,1, FN1,2, ..., FN1,DET(1),
  ...,
  FNNCV,1, FNNCV,2, ..., FNNCV,DET(NCV)
)
```

2) Otherwise, let *PDCLP* be the zero-length character string.

n) If ET_j, 1 (one) ≤ j ≤ *NCV*, is a row type, then let SLE_j be `(`N_j`).*`; otherwise, let SLE_j be N_j.

Case:

i) If *CDT* specifies WITH ORDINALITY, then let *ELDT* be:

```
LATERAL ( RECQP SELECT SLE1, ..., SLENCV, NORD
                FROM TEMP ) AS CN PDCLP
```

The column named by *NORD* is called the *ordinality column* of *CDT*.

ii) Otherwise, let *ELDT* be:

```
LATERAL ( RECQP SELECT SLE1, ..., SLENCV
                FROM TEMP ) AS CN PDCLP
```

o) *TP* is equivalent to the <table primary> *ELDT*.

9) [15]If *TP* simply contains <table or query name> *TOQN* and *TP* does not simply contain a <PTF derived table>, then:

a) Case:

i) If *TOQN* is an <identifier> that is equivalent to a <query name> *QN*, then let *WLE* be the <with list element> simply contained in the <query expression> that contains *TP* such that the <query name> *QN1* simply contained in *WLE* is equivalent to *QN* and *QN1* is the innermost query name in scope. Let the *table specified by TOQN* be the result of *WLE*.

NOTE 211 — "query name in scope" is defined in Subclause 7.17, "<query expression>".

ii) If *TOQN* is an <identifier> that is equivalent to a <transition table name> that is in scope, then let the *table specified by TOQN* be the table identified by *TOQN*.

NOTE 212 — The scope of a <transition table name> is defined in Subclause 11.49, "<trigger definition>".

iii) Otherwise, let the *table specified by TOQN* be the table *T* identified by the <table name> *TN* simply contained in *TP*. The schema identified by the explicit or implicit qualifier of *TN* shall include the descriptor of *T*.

NOTE 213 — The preceding cases disambiguate whether *TOQN* is interpreted as a <query name>, <transition table name>, or <table name>.

b) If *TP* simply contains <only spec> *OS* and the table specified by *TOQN* is not a typed table, then *OS* is equivalent to *TOQN*.

c) If the table specified by *TOQN* is a system-versioned table and <query system time period specification> is not specified, then FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP is implicit.

d) If *TP* immediately contains <query system time period specification> *QSTPS*, then:

i) The table specified by *TOQN* shall be a system-versioned table.

ii) If BETWEEN is specified and neither SYMMETRIC nor ASYMMETRIC is specified, then ASYMMETRIC is implicit.

iii) 04 *QSTPS* shall not contain a <column reference> or an <SQL parameter reference>.

10) The table specified by a <lateral derived table> is the table specified by the simply contained <query expression>.

11) 16 If *TP* simply contains a <data change delta table> *DCDT*, then let *S* be the <data change statement> simply contained in *TP*. *S* shall not contain FOR PORTION OF. Let *TT* be the subject table of *S*.

a) Case:

i) If *S* is an <insert statement>, then the <result option> shall not specify OLD.

ii) If *S* is a <delete statement: searched>, then the <result option> shall not specify NEW or FINAL.

iii) If *S* is a <merge statement> and *S* does not contain either <merge update specification> or <merge delete specification>, then the <result option> shall not specify OLD.

iv) If *S* is a <merge statement> and *S* does not contain either <merge update specification> or <merge insert specification>, then the <result option> shall not specify NEW or FINAL.

b) If *TT* is a viewed table, then:

i) The view descriptor of *TT* shall indicate CASCADED CHECK OPTION.

ii) If FINAL is specified, then:

1) If either *S* is an <insert statement> or *S* is a <merge statement> that contains a <merge insert specification>, then *TT* shall not be trigger insertable-into and no target generally underlying table of *TT* shall be trigger insertable-into.

2) If either *S* is an <update statement: searched> or *S* is a <merge statement> that contains a <merge update specification>, then *TT* shall not be trigger updatable and no target generally underlying table of *TT* shall be trigger updatable.

3) If *S* is a <merge statement> that contains a <merge delete specification>, then *TT* shall not be trigger deletable and no target generally underlying table of *TT* shall be trigger deletable.

c) The table specified by *DCDT* is *TT*.

12) [15] [16] Let *TPTI* be the table specified by the <table or query name>, <derived table>, <lateral derived table>, <PTF derived table>, <data change delta table>, <JSON table primitive>, or the <joined table> simply contained in *TP*. The degree of *TPT* and the column descriptor of each of the columns of *TPT* are determined as follows.

Case:

a) If *TP* simply contains a <PTF derived table>, then:

i) Let *NIT* be the number of table parameters of *SR*. For *t*, 1 (one) ≤ *t* ≤ *NIT*,

Case:

1) If the *t*-th generic table parameter of *SR* specifies PASS THROUGH, then let NC_t be the number of columns of the *t*-th <table argument> of *RI*.

2) Otherwise, let NC_t be the number of partitioning columns of the *t*-th <table argument> of *RI*.

ii) Case:

1) If *SR* specifies RETURNS ONLY PASS THROUGH, then let *NPC* be 0 (zero).

2) Otherwise, let *NPC* be the number of columns in the external result row type of *TP*.

iii) The degree of *TPT* is the sum of *NPC* and the NC_t, 1 (one) ≤ *t* ≤ *NIT*.

iv) The list of column descriptors of *TPT* is formed as the concatenation of the following lists of column descriptors:

1) If *SR* does not specify RETURNS ONLY PASS THROUGH, then the column descriptors of the external result row type of *TP*. These columns are the *external result columns* of *TP*. The external result columns of *TP* are possibly nullable.

NOTE 214 — If *SR* specifies RETURNS ONLY PASS THROUGH, then there is no external result row type.

2) For each *t*, 1 (one) ≤ *t* ≤ *NIT*:

A) Case:

I) If the *t*-th generic table parameter of *SR* specifies PASS THROUGH, then the list of column descriptors of the *t*-th <table argument> of *RI*. These columns are said to be *derived* from the columns of the *t*-th <table argument> of *SR*.

II) If the *t*-th <table argument> of *RI* has at least one partitioning column, then the list of column descriptors of the partitioning columns of the *t*-th <table argument> of *RI*. These columns are said to be *derived* from the columns of the *t*-th <table argument> of *SR*.

NOTE 215 — Otherwise, the *t*-th input <table argument> of *RI* does not contribute any column descriptors to the column descriptors of *TP*.

B) A column *CX* derived from the *t*-th <table argument> of *RI* is known not null if *CX* is a known not null partitioning column and the *t*-th <table argument> is not co-partitioned; otherwise, *CX* is possibly nullable.

b) If *TP* is not a <parenthesized joined table>, then

Case:

i) If <row pattern recognition clause and name> *RPRCAN* is specified, then:

1) Let *RPRC* be the <row pattern recognition clause> simply contained in *RPRCAN*.

2) 15 16 The table specified by the <table or query name>, <derived table>, <joined table>, <lateral derived table>, <data change delta table>, or <only spec> immediately contained in *TP* is the *row pattern input table* of *RPRC*.

3) Case:

A) If <row pattern input derived column list> *RPIDCL* is specified, then:

I) The number of <column name>s in *RPIDCL* shall be the same as the degree of *TPTI*.

II) The row type of the row pattern input table of *RPRC* is described by a sequence of (<field name>, <data type>) pairs, where the <field name> in the *i*-th pair is the *i*-th <column name> in *RPIDCL* and the <data type> in the *i*-th pair is the declared type of the *i*-th column of the table specified by *TPTI*.

B) Otherwise, the row type of the row pattern input table of *RPRC* is the row type of *TPTI*.

4) Case:

A) If <row pattern input name> is specified, then let *RPIN* be that <row pattern input name>.

B) If *TP* specifies <table or query name>, then let *RPIN* be that <table or query name>.

C) If *TP* specifies <only spec>, then let *RPIN* be the <table or query name> simply contained in the <only spec>.

D) Otherwise, let *RPIN* be an implementation-dependent (UV080) <correlation name> that is distinct from all other <correlation name>s in the <SQL procedure statement> that simply contains *TP*.

5) *RPIN* is a range variable of *RPRCAN*. *RPIN* is exposed by *RPRCAN*. The scope of *RPIN* is the <row pattern partition by> and <row pattern order by> simply contained in *RPRC*. *RPIN* references the row pattern input table of *RPRC*. The associated column list of *RPIN* is the list of columns in the row type of the row pattern input table. The associated period list of *RPIN* is empty. *RPIN* is the *row pattern input name* of the row pattern input table of *RPRC*.

6) Case:

A) If <row pattern output derived column list> *RPODCL* is specified, then:

I) The number of <column name>s in *RPODCL* shall be the same as the degree of the row pattern output table of *RPRC*.

II) The row type *RT* of *TR* is described by a sequence of (<field name>, <data type>) pairs, where the <field name> in the *i*-th pair is the *i*-th <column name> in *RPODCL* and the <data type> in the *i*-th pair is the declared type of the *i*-th column of the row pattern output table of *RPRC*.

B) Otherwise, the row type *RT* of *TR* is the row type of the row pattern output table of *RPRC*.

7) If *RPRCAN* does not contain a <row pattern output name>, then an implementation-dependent (UV081) <correlation name> distinct from all other <correlation name>s in the <SQL procedure statement> that simply contains *TP* is implicit. The explicit or implicit <row pattern output name> *RPON* references the row pattern output table of *RPRC*. The associated column list of *RPON* is the list of columns in the row type of the row pattern output table. The associated period list of *RPON* is empty.

ii) Otherwise:

1) If a <derived column list> *DCL* is specified, then:

A) No <column name> shall be specified more than once in *DCL*.

B) The number of <column name>s in *DCL* shall be the same as the degree of *TPTI*.

2) The degree of *TPT* is same as the degree of *TPTI*.

3) The column name included in the descriptor of *i*-th column of *TPT* is

Case:

A) If a <derived column list> *DCL* is specified, then the *i*-th <column name> in *DCL*.

B) Otherwise, the column name included in the descriptor of *i*-th column of *TPTI*.

4) The data type included in the descriptor of the *i*-th column of *TPT* is the declared type of the *i*-th column of *TPTI*.

c) Otherwise, the degree of *TPT* and the column descriptor of each of the columns of *TPT* are same as degree and the column descriptors of each of the columns of *TPTI*.

13) The row type of <table primary> *TP* is described by a sequence of (<field name>, <data type>) pairs, where the <field name> and <data type> in the *i*-th pair are the column name and the data type, respectively, included in the descriptor of the *i*-th column of *TPT*.

14) Case:

a) If *TR* immediately contains a <table factor> *TF*, then the degree and the column descriptors of the table specified by *TF* and *TR* are the same as the degree and the column descriptors of of the table specified by the <table primary> *TP* immediately contained in *TF*. The row type of *TF* and *TR* is the same as the row type of *TP*.

b) Otherwise, the degree and the column descriptors of the table specified by *TR* are the same as the degree and the column descriptors of the table specified by <joined table> *JT*. The row type of *TR* is the same as the row type of *JT*.

15) Case:

a) If *TP* simply contains a <PTF derived table>, then the names of the range variables of *TP* are the names of the range variables of the <table argument>s of *TP* and, if *SR* does not specify RETURNS ONLY PASS THROUGH, *PTFCN*. The range variables of *TP* are *exposed* by *TP*. The table associated with each of the range variables of *TP* is *TPT*. The period list associated with each of the range variables of *TP* is empty. The column list associated with each of the range variables of *TP* is

Case:

i) If *RV* is *PTFCN*, then the external result columns of *TP*.

ii) If *RV* is a range variable of a <table argument> of *TP*, then the list of columns derived from that <table argument> that are in the associated column list of *RV* as a range variable of the <table argument>.

NOTE 216 — Each <table argument> exposes one or more range variables. In the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", each of these range variables has an associated column list *ACL*, which is used to resolve <column reference>s in the <table argument partitioning> and <table argument ordering>. If the <table argument> has pass-through columns, then every column in *ACL* is a column of *TP* that is derived from the <table argument>; otherwise, only the partitioning columns in *ACL* are columns of *TP*. Thus, it is possible that the associated column list of *RV* as a range variable of *TP* is a subset (possibly empty) of the associated column list of *RV* as a range variable of a <table argument>.

b) If *TP* is not a <parenthesized joined table>, then:

i) Case:

1) If *TP* simply contains <row pattern recognition clause and name> *RPRCAN*, then let *RV* be the <correlation name> that is the implicit or explicit <row pattern output name>.

2) If *TP* simply contains a <correlation name>, then let *RV* be that <correlation name>.

3) Otherwise, let *RV* be the <table or query name> simply contained in *TP*.

ii) *RV* is a range variable of *TP*. *RV* is *exposed* by *TP*. The table associated with *RV* is *TPT*.

iii) The associated column list of *RV* is the list consisting of every column of *TPT*.

iv) Case:

1) If *TP* immediately contains a <table or query name> that immediately contains a <table name> that identifies a table whose table descriptor includes a period descriptor and *TP* does not simply contain a <row pattern recognition clause>, then the associated period list of *RV* is the list consisting of the name of every period included in the descriptor of *TPT*.

2) Otherwise, the associated period list of *RV* is empty.

NOTE 217 — "range variable", "table associated with a range variable", "associated column list of a range variable", and "associated period list of a range variable" are defined in Subclause 4.17.11, "Range variables".

c) Otherwise, *TP* has the same number of range variables as the number of range variables of the simply contained <joined table> *JT*. The names of range variables of *TP* are the same as the names of range variables of *JT*. The table associated with each of the range variables of *TP*

is *TPT*. The associated column list of each of the range variables of *TP* is the same as the associated column list of the corresponding range variable of *JT*. The associated period list of each of the range variables of *TP* is empty. Every range variable of *TP* is *exposed* by *TP*.

16) *TF* has the same number of range variables as the number of range variables of the immediately contained <table primary> *TP*. The names of range variables of *TF* are the same as the names of range variables of *TP*. The table associated with each of the range variables of *TF* is *TFT*. The associated column list of each of the range variables of *TF* is the same as the associated column list of the corresponding range variable of *TP*. The associated period list of each of the range variables of *TF* is the same as the associated period list of the corresponding range variable of *TP*. Every range variable of *TF* is *exposed* by *TF*.

17) Case:

    a) If *TR* immediately contains a <table factor> *TF*, then *TR* has the same number of range variables as the number of range variables of *TF*. The names of range variables of *TR* are the same as the names of range variables of *TF*. The table associated with each of the range variables of *TR* is *TRT*. The associated column list of each of the range variables of *TR* is the same as the associated column list of the corresponding range variable of *TF*. The associated period list of each of the range variables of *TR* is the same as the associated period list of the corresponding range variable of *TF*. Every range variable of *TR* is *exposed* by *TR*.

    b) Otherwise, *TR* has the same number of range variables as the number of range variables of the immediately contained <joined table> *JT*. The names of range variables of *TR* are the same as the names of range variables of *JT*. The table associated with each of the range variables of *TR* is *TRT*. The associated column list of each of the range variables of *TR* is the same as the associated column list of the corresponding range variable of *JT*. The associated period list of each of the range variables of *TR* is empty. Every range variable of *TR* is *exposed* by *TR*.

18) Range variables of *TP* have no scope.

19) Case:

    a) If *TF* is simply contained in a <joined table> *JT*, then the scope of every range variable of *TF* is the <join condition> of *JT*.

    b) Otherwise, range variables of *TF* have no scope.

20) Case:

    a) If *TR* is simply contained in a <from clause> *FC*, then the scope of the range variables of *TR* is every <table primary> that supports lateral join and that is simply contained in *FC* and is preceded by *TR*, and the <join condition> of all <joined table>s contained in *FC* that contain *TR*.

    b) Otherwise, *TR* is simply contained in a <merge statement> *MS*. The scope of the range variable of *TR* is the <search condition>s, <set clause list>s, and <merge insert value list>s of *MS*, but excluding the <row pattern measures> and <row pattern definition list> contained in any <window clause>.

        NOTE 218 — Subclause 14.12, "<merge statement>", does not allow *TR* to directly contain a <joined table>.

21) Case:

    a) 14 16 If <table primary> specifies <collection derived table>, <table function derived table>, <JSON table>, <JSON table primitive>, <data change delta table>, or <row pattern recognition clause and name>, then the <table primary> is not generally updatable, not simply updatable, not effectively updatable, and not insertable-into.

    b) Otherwise:

i) A <derived table> or <lateral derived table> is a *generally updatable derived table* if the <query expression> simply contained in the <derived table> or <lateral derived table> is generally updatable.

ii) A <derived table> or <lateral derived table> is a *simply updatable derived table* if the <query expression> simply contained in the <derived table> or <lateral derived table> is simply updatable.

iii) A <derived table> or <lateral derived table> is an *effectively updatable derived table* if the <query expression> simply contained in the <derived table> or <lateral derived table> is effectively updatable.

iv) A <derived table> or <lateral derived table> is an *insertable-into derived table* if the <query expression> simply contained in the <derived table> or <lateral derived table> is insertable-into.

22) [14]If a <table reference> *TR* immediately contains a <table factor> *TF*, then

Case:

a) If *TF* simply contains a <table name> that identifies a base table, then every column of the table identified by *TF* is called an *updatable column* of *TR*.

b) If *TF* simply contains a <table name> that identifies a view, then every updatable column of the view identified by *TF* is called an *updatable column* of *TR*.

c) If *TF* simply contains a <derived table> or <lateral derived table>, then every updatable column of the table identified by the <query expression> simply contained in <derived table> or <lateral derived table> is called an *updatable column* of *TR*.

Access Rules

1) If a <table primary> *TP* simply contains a <table or query name> that simply contains a <table name> *TN*, then:

a) Let *T* be the table identified by *TN*.

b) Case:

i) If *TN* is contained in a <search condition> immediately contained in an <assertion definition> or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on at least one column of *T*.

ii) Otherwise:

1) Case:

A) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on at least one column of *T*.

B) Otherwise, the current privileges shall include SELECT on at least one column of *T*.

2) If *TP* simply contains <only spec> and *TN* identifies a typed table, then

Case:

A) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

B) Otherwise, the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

2) If the <table primary> *TP* simply contains a <data change delta table>, then let *S* be the <data change statement> simply contained in *TR*. Let *TT* be the subject table of *S*.

Case:

a) If *TP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on at least one column of *TT*.

b) Otherwise, the current privileges shall include SELECT on at least one column of *TT*.

General Rules

1) If a <table primary> *TP* simply contains a <row pattern recognition clause and name> *RPRCAN*, then:

a) Let *TP1* be the <table primary> that is obtained from *TP* by substituting

```
AS X
```

for *RPRCAN* in *TP*.

NOTE 219 — The choice of X as the <correlation name> for the row pattern input table is meaningless; the row pattern input name and the row type of the row pattern input table are determined by the Syntax Rules.

b) *TP1* is evaluated according to the remaining General Rules of this Subclause.

c) The value of *TP1* is the row pattern input table of the <row pattern recognition clause>, with row type as determined by the Syntax Rules.

d) The value of *TP* is the row pattern output table that is the value of the <row pattern recognition clause>, using the value of *TP1* as the row pattern input table.

e) The remaining General Rules of this Subclause are not evaluated for *TP*.

2) If a <table primary> *TP* simply contains a <table or query name> *TOQN*, then

Case:

a) If *TOQN* simply contains a <query name> *QN*, then the result of *TP* is the table specified by *QN*.

b) If *TOQN* simply contains a <transition table name> *TTN*, then the result of *TP* is the table specified by *TTN*.

NOTE 220 — The table identified by a <transition table name> is a transition table as defined by the General Rules of Subclause 15.8, "Effect of deleting rows from base tables", Subclause 15.11, "Effect of inserting tables into base tables", or Subclause 15.14, "Effect of replacing rows in base tables", as appropriate.

c) Otherwise, let *T* be the table specified by the <table name> simply contained in *TP*.

Case:

i) 09 If ONLY is specified, then the result of *TP* is a table that consists of every row in *T*, except those rows that have a subrow in a proper subtable of *T*.

ii) Otherwise,

Case:

1) If *T* is a system-versioned table, then:

A) Let *SVS* be the implicit or explicit <query system time period specification>, let *SSTARTCOL* be the system-time period start column of *T* and let *SENDCOL* be the system-time period end column of *T*. Let *DT* be the declared type of *SSTARTCOL*.

B) If *SVS* specifies FOR SYSTEM_TIME AS OF, then let *POTV1* be the value of <point in time 1>. Let *POT1* be the result of CAST (*POTV1* AS *DT*).

C) If *SVS* specifies FOR SYSTEM_TIME BETWEEN or FOR SYSTEM_TIME FROM, then let *POTV1* be the value of <point in time 1> and let *POTV2* be the value of <point in time 2>.

Case:

I) If SYMMETRIC is specified and *POTV1* > *POTV2*, then let *POT1* be the result of CAST (*POTV2* AS *DT*) and let *POT2* be the result of CAST (*POTV1* AS *DT*).

II) Otherwise, let *POT1* be the result of CAST (*POTV1* AS *DT*) and let *POT2* be the result of CAST (*POTV2* AS *DT*).

D) Case:

I) If *SVS* specifies FOR SYSTEM_TIME AS OF, then the result of *TP* is a table that consists of every row *R* of *T* for which the result of (*SSTARTCOL* <= *POT1* AND *SENDCOL* > *POT1*) is *True*.

II) If *SVS* specifies FOR SYSTEM_TIME BETWEEN, then the result of *TP* is a table that consists of every row *R* of *T* for which the result of (*POT1* <= *POT2* AND *SENDCOL* > *POT1* AND *SSTARTCOL* <= *POT2*) is *True*.

III) If *SVS* specifies FOR SYSTEM_TIME FROM, then the result of *TP* is a table that consists of every row *R* of *T* for which the result of (*POT1* < *POT2* AND *SENDCOL* > *POT1* AND *SSTARTCOL* < *POT2*) is *True*.

2) Otherwise, the result of *TP* is a table that consists of every row of *T*.

3) 14 If a <derived table>or <lateral derived table> *LDT* simply containing <query expression> *QE* is specified, then the result of *LDT* is the result of *QE*.

4) If a <PTF derived table> *PTFDT* is specified, then:

a) Let *PAL* be the pre-compilation argument list of *RI*.

NOTE 221 — For compilation, any arguments that were not compile-time constants were replaced by the null value in the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine". For execution it is necessary to go back to the pre-compilation argument list so that values known only during execution can be computed.

b) The General Rules of Subclause 9.25, "Execution of an invocation of a polymorphic table function", are applied with *RI* as *ROUTINE INVOCATION*, *SR* as *POLYMORPHIC TABLE FUNCTION*, *PAL* as *ARGUMENT LIST*, *PTFDA* as *PTF DATA AREA*, and the row type of *TP* as *RESULT TYPE*; let *RESTAB* be the *RESULT TABLE* returned from the application of those General Rules.

c) The result of *PTFDT* is *RESTAB*.

5) If a <data change delta table> *DCDT* is specified, then let *S* be the <data change statement> immediately contained in *DCDT* and let *TT* be the subject table of *S*.

a) If FINAL is specified, then:

i) If the subject table restriction flag of the current SQL-session context is *False*, then it is set to *True*.

ii) The <table name> of *TT* is added to the restricted subject table name list included in the current SQL-session context.

b) Case:

i) If *S* is a <delete statement: searched>, then the result of *DCDT* is the old delta table of delete operation on *TT*.

NOTE 222 — "old delta table of delete operation" is defined in Subclause 14.9, "<delete statement: searched>".

ii) If *S* is an <insert statement>, then the result of *DCDT* is the new delta table of insert operation on *TT*.

NOTE 223 — "new delta table of insert operation" is defined in Subclause 15.11, "Effect of inserting tables into base tables", and Subclause 15.13, "Effect of inserting a table into a viewed table".

iii) If *S* is a <merge statement>, then

Case:

1) If OLD is specified, then the result of *DCDT* is the old delta table of merge operation on *TT*.

NOTE 224 — "old delta table of merge operation" is defined in Subclause 14.12, "<merge statement>".

2) Otherwise, the result of *DCDT* is the new delta table of merge operation on *TT*.

NOTE 225 — "new delta table of merge operation" is defined in Subclause 14.12, "<merge statement>".

iv) If *S* is an <update statement: searched>, then

Case:

1) If OLD is specified, then the result of *DCDT* is the old delta table of update operation on *TT*.

NOTE 226 — "old delta table of update operation" is defined in Subclause 14.14, "<update statement: searched>".

2) Otherwise, the result of *DCDT* is the new delta table of update operation on *TT*.

NOTE 227 — "new delta table of update operation" is defined in Subclause 15.14, "Effect of replacing rows in base tables", and Subclause 15.16, "Effect of replacing some rows in a viewed table".

c) If FINAL is specified, then:

i) The <table name> of *TT* is removed from the restricted subject table name list included in the current SQL-session context.

ii) If the restricted subject table name list included in the current SQL-session context is empty, then the subject table restriction flag of the current SQL-session context is set to *False*.

d) All old delta tables and all new delta tables in the most recent statement execution context are destroyed.

6) 16 If *TP* immediately contains a <joined table>, then the result of *TP* is the result of that <joined table>.

7) 15 Let *TP* be the <table primary> immediately contained in a <table factor> *TF*. Let *RT* be the result of *TP*.

Case:

a) If <sample clause> is specified, then:

i) Let *N* be the number of rows in *RT* and let *S* be the value of <sample percentage>.

ii) If *S* is the null value or if *S* < 0 (zero) or if *S* > 100, then an exception condition is raised: *data exception — invalid sample size (2202H)*.

iii) If <repeatable clause> is specified, then let *RPT* be the value of <repeat argument>. If *RPT* is the null value, then an exception condition is raised: *data exception — invalid repeat argument in a sample clause (2202G)*.

iv) Case:

1) If <sample method> specifies BERNOULLI, then the result of *TF* is a table containing approximately (*N***S*/100) rows of *RT*. The probability of a row of *RT* being included in result of *TF* is *S*/100. Further, whether a given row of *RT* is included in result of *TF* is independent of whether other rows of *RT* are included in result of *TF*.

2) Otherwise, result of *TF* is a table containing approximately (*N***S*/100) rows of *RT*. The probability of a row of *RT* being included in result of *TF* is *S*/100.

v) If *TF* contains outer references, then a table with identical rows is generated every time *TF* is evaluated with a given set of values for outer references.

NOTE 228 — "outer reference" is defined in Subclause 6.7, "<column reference>".

b) Otherwise, result of *TF* is *RT*.

8) The result of a <table reference> *TR* is the result of the immediately contained <table factor> or <joined table>.

Conformance Rules

1) Without Feature R010, "Row pattern recognition: FROM clause", in conforming SQL language, a <correlation or recognition> shall not contain a <row pattern recognition clause>.

2) 15 Without at least one of:

a) Feature S090, "Minimal array support"

b) Feature S271, "Basic multiset support"

conforming SQL language shall not contain a <collection derived table>.

3) Without Feature T491, "LATERAL derived table", conforming SQL language shall not contain a <lateral derived table>.

4) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", conforming SQL language shall not contain a <query name>.

5) Without Feature S111, “ONLY in query expressions”, conforming SQL language shall not contain a <table reference> that contains an <only spec>.

6) Without Feature F591, “Derived tables”, conforming SQL language shall not contain a <derived table>.

7) Without Feature T326, “Table functions”, conforming SQL language shall not contain a <table function derived table>.

8) Without Feature T613, “Sampling”, conforming SQL language shall not contain a <sample clause>.

9) Without Feature T200, “Trigger DDL”, conforming SQL language shall not contain a <transition table name>.

10) Without Feature S301, “Enhanced UNNEST”, in conforming SQL language, a <collection derived table> shall not simply contain more than one <collection value expression>.

11) Without Feature T495, “Combined data change and retrieval”, conforming SQL language shall not contain <data change delta table>.

12) Without Feature T180, “System-versioned tables”, conforming SQL language shall not contain <query system time period specification>.

13) 14 15 16 Without Feature B200, “Polymorphic table functions”, conforming SQL language shall not contain <PTF derived table>.

7.7 <row pattern recognition clause>

Function

Match row patterns.

Format

```
<row pattern recognition clause> ::=
  MATCH_RECOGNIZE <left paren>
      [ <row pattern partition by> ]
      [ <row pattern order by> ]
      [ <row pattern measures> ]
      [ <row pattern rows per match> ]
      <row pattern common syntax>
      <right paren>

<row pattern partition by> ::=
  PARTITION BY <row pattern partition list>

<row pattern partition list> ::=
  <row pattern partition column> [ { <comma> <row pattern partition column> }... ]

<row pattern partition column> ::=
  <column reference> [ <collate clause> ]

<row pattern order by> ::=
  ORDER BY <sort specification list>

<row pattern rows per match> ::=
    ONE ROW PER MATCH
  | ALL ROWS PER MATCH [ <row pattern empty match handling> ]

<row pattern empty match handling> ::=
    SHOW EMPTY MATCHES
  | OMIT EMPTY MATCHES
  | WITH UNMATCHED ROWS
```

Syntax Rules

1) Let *RPRC* be the <row pattern recognition clause>.

2) Let *TE* be the table identified by the <table or query name>, <derived table>, <lateral derived table>, <collection derived table>, <table function derived table>, <only spec>, or <data change delta table> of the <table primary> that contains *RPRC*. *TE* is the *row pattern input table* of *RPRC*.

3) Case:

   a) If <row pattern partition by> *RPPB* is specified, then:

      i) Let *NP* be the number of <row pattern partition column>s contained in the <row pattern partition list> contained in *RPBB*.

      ii) For each <row pattern partition column> $RPPC_i$, 1 (one) ≤ i ≤ *NP*:

         1) The <column reference> $RPPCR_i$ contained in $RPPC_i$ shall unambiguously reference a column of *TE*. The column PC_i referenced in $RPPC_i$ is a *row pattern partitioning*

column of *RPRC*. Each row pattern partitioning column is an operand of a grouping operation, and the Syntax Rules of Subclause 9.12, "Grouping operations", apply.

2) Case:

A) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of PC_i shall be character string. The declared type of $RPPC_i$ is that of PC_i, except that *CS* is the declared type collation and the collation derivation is explicit.

B) Otherwise, the declared type of $RPPC_i$ is the declared type of PC_i.

iii) Let *PCS* be the ordered set of row pattern partitioning columns $RPPC_i$, 1 (one) $\leq i \leq NP$.

b) Otherwise, let *PCS* be the empty set.

4) Case:

a) If <row pattern order by> *RPOB* is specified, then:

i) Let *NS* be the number of <sort specification>s contained in the <sort specification list> contained in *RPOB*.

ii) For each <sort specification> $RPOC_i$, 1 (one) $\leq i \leq NS$, $RPOC_i$ is an operand of an ordering operation. The Syntax Rules of Subclause 9.14, "Ordering operations", apply.

iii) For all *i*, 1 (one) $\leq i \leq NS$, the <value expression> simply contained in $RPOC_i$ shall be a <column reference> that references a column of the row pattern input table.

iv) Let *OCS* be the ordered set of <sort specification>s $RPOC_i$, 1 (one) $\leq i \leq NS$.

b) Otherwise, let *OCS* be the empty set.

5) If <row pattern rows per match> is not specified, then ONE ROW PER MATCH is implicit.

6) If ALL ROWS PER MATCH is specified and <row pattern empty match handling> is not specified, then SHOW EMPTY MATCHES is implicit.

7) Case:

a) If <row pattern partition by> is specified, then let *TEM* be a table whose column descriptors are the column descriptors of the row pattern partitioning columns of *RPRC*, in the order in which they are listed in the <row pattern partition by>.

b) Otherwise, let *TEM* be a table with no column descriptors.

8) If ALL ROWS PER MATCH is specified, and <row pattern order by> is specified, then column descriptors of the columns of *TE* that are referenced by the <sort specification>s of the <row pattern order by> are appended to *TEM* in the order of specification in <row pattern order by>.

9) If <row pattern measures> *RPM* is specified, then the Syntax Rules of Subclause 7.8, "<row pattern measures>", specify the creation of row pattern measure columns. The descriptors of the row pattern measure columns are appended to the column descriptors in *TEM* in their order of specification.

10) If ALL ROWS PER MATCH is specified, then column descriptors of any columns of *TE* that are not already included in *TEM* are appended to *TEM* in the order in which they occur in *TE*.

11) *TEM* is the *row pattern output table* of *RPRC*. Let *DTEM* be the degree of *TEM*. *DTEM* shall be positive.

NOTE 229 — This implies that, if ONE ROW PER MATCH is specified or implicit, then either <row pattern partition by> or <row pattern measures> must be specified.

Access Rules

None.

General Rules

1) Case:

   a) If <row pattern partition by> *RPPB* is specified, then for each row *R* of *TE*, the *row pattern partition* of *R* is the collection of rows for which the values of *PCS* are not distinct from the values of *PCS* in *R*.

   b) Otherwise, the *row pattern partition* of *R* is is the collection of rows in *TE*.

2) Let *NRPP* be the number of distinct row pattern partitions. Let RPP_i, 1 (one) ≤ *i* ≤ *NRPP*, be an enumeration of the distinct row pattern partitions.

3) Case:

   a) If <row pattern order by> *RPOB* is specified, then the order of rows in each RPP_i, 1 (one) ≤ *i* ≤ *NRPP*, is determined by the General Rules of Subclause 10.10, "<sort specification list>", using the <sort specification list> simply contained in *RPOB*.

   b) Otherwise, the order of rows in each RPP_i, 1 (one) ≤ *i* ≤ *NRPP*, is implementation-dependent (US032).

4) Let *RPCS* be the <row pattern common syntax>. Let *RP* be the <row pattern> simply contained in *RPCS*, let *RPDL* be the <row pattern definition list> simply contained in *RPCS*, let *RPSUB* be the <row pattern subset clause> implied by *RPCS* by adding the universal row pattern variable, and let *RPST* be the <row pattern skip to> simply contained in *RPCS*.

5) For each *i*, 1 (one) ≤ *i* ≤ *NRPP*, the General Rules of Subclause 9.41, "Row pattern recognition in a sequence of rows", are applied with RPP_i as *ROW SEQUENCE*, *RP* as *PATTERN*, *RPSUB* as *SUBSETS*, and *RPDL* as *DEFINES*; let *SOM* be the *SET OF MATCHES* returned from the application of those General Rules and let $SRPM_i$ be *SOM*.

6) For each row pattern partition RPP_i, let the preferred row pattern matches be ordered by increasing initial row number, so that the preferred row pattern matches form the sequence $SRPM_i$ = { (STR_1, RPP_i, k_1), (STR_2, RPP_i, k_2), ... }, where $k_1 < k_2 <$ ...

   NOTE 230 — Two preferred row pattern matches cannot have the same initial row because preferment is a total ordering.

7) Let *Vcount* be the variable count function, as defined in Subclause 9.41, "Row pattern recognition in a sequence of rows".

8) For all *i*, 1 (one) ≤ *i* ≤ *NRPP*, the set of retained row pattern matches $RRPM_i$ is defined as follows: initially, $RRPM_i$ is empty. The following iterative steps progressively remove preferred row pattern matches from $SRPM_i$ and place some of them in $RRPM_i$, until $SRPM_i$ is empty.

   a) The first element (STR_a, RPP_i, k_a) remaining in $SRPM_i$ is removed and placed in $RRPM_i$.

   b) Case:

      i) If *RPST* specifies SKIP TO NEXT ROW, or if *Vcount*(STR_a) = 0 (zero), then no additional matches are removed from $SRPM_i$.

ii) If *RPST* specifies SKIP PAST LAST ROW, then every preferred row pattern match (STR_b, RPP_i, k_b) such that $k_b < k_a + Vcount(STR_a)$ is removed from $SRPM_i$.

NOTE 231 — These are the preferred row pattern matches that overlap the rows mapped by (STR_a, RPP_i, k_a). These preferred row pattern matches are not placed in $RRPM_i$.

iii) If *RPST* specifies SKIP TO FIRST <row pattern skip to variable name> or specifies or implies SKIP TO LAST <row pattern skip to variable name>, then let *RPV* be the row pattern variable identified by the <row pattern skip to variable name>. Let { c_1, ..., c_u } be the collection of subscripts c such that R_c is mapped to *RPV*.

Case:

1) If the collection of subscripts is empty, then an exception condition is raised: *data exception — skip to non-existent row (2202K)*.

2) If SKIP TO FIRST is specified, then let t be the minimum of c_1, ..., c_u.

Case:

A) If $t = k_a$, then an exception condition is raised: *data exception — skip to first row of match (2202L)*.

B) Otherwise, every preferred row pattern match (STR_b, RPP_i, k_b) such that $k_b < t$ is removed from $SRPM_i$.

3) Otherwise, SKIP TO LAST is specified. Let t be the maximum of c_1, ..., c_u.

Case:

A) If $t = k_a$, then an exception condition is raised: *data exception — skip to first row of match (2202L)*.

B) Otherwise, every preferred row pattern match (STR_b, RPP_i, k_b) such that $k_b < t$ is removed from $SRPM_i$.

9) For all i, 1 (one) ≤ i ≤ *NRPP*, retained row pattern matches in $RRPM_i$ are ordered in increasing order by the initial row number.

10) Let *RRPM* be the set union of all $RRPM_i$, 1 (one) ≤ i ≤ *NRPP*.

11) The row pattern output table *RPROT* is assembled as follows.

Case:

a) If ONE ROW PER MATCH is specified, then one row is created for each retained row pattern match (*STR*, RPP_i, k), as follows:

i) The value of each row pattern partitioning column of *RPROT* is a value that is not distinct from the value of the row pattern partitioning column in every row of RPP_i.

ii) The value of each row pattern measure column of *RPROT* is the value of the <value expression> that is the <row pattern measure expression> of the row pattern measure column, evaluated for (*STR*, RPP_i, k).

b) Otherwise, ALL ROWS PER MATCH is specified.

i) Each row of the row pattern input table is provided with a *matched indicator*, initially set to *False*.

ii) For each retained row pattern match (*STR*, RPP_i, *k*)

Case:

1) If *Vcount*(*STR*) = 0 (zero) and either WITH UNMATCHED ROWS is specified or SHOW EMPTY MATCHES is specified or implicit, then the matched indicator of row R_k is set to *True* and one row of *RPROT* is created as follows:

A) The value of each column *C* whose column descriptor is equivalent to a column descriptor of the row pattern input table is the value of *C* in row R_k.

B) The value of each row pattern measure column *RPMC* of *RPROT* is the value of the <value expression> that is the <row pattern measure expression> of *RPMC*, evaluated for (*STR*, RPP_i, *k*).

NOTE 232 — Any <set function specification>s will be evaluated with an empty argument source; thus COUNT is 0 (zero) and other aggregates produce a null value. Any <row pattern navigation operation>s (including any ordinary row pattern column references) evaluate to the null value. <match number function> returns the match number of (*STR*, RPP_i, *k*), and <classifier function> returns null.

NOTE 233 — If *Vcount*(*STR*) = 0 (zero) and OMIT EMPTY MATCHES is specified, then no row is created.

2) If *Vcount*(*STR*) > 0 (zero), then for each row *R* that is mapped by (*STR*, RPP_i, *k*):

A) The matched indicator of *R* is set to *True*.

B) Case:

I) If *R* is mapped to a row pattern variable that is between matching special symbols "[" and "]", then no row is created for *R*.

NOTE 234 — The special symbols "[" and "]" correspond to <left brace minus> and <right minus brace> ("{-" and "-}", respectively) in row patterns in SQL language.

II) Otherwise, a row of *RPROT* is created, as follows:

1) The value of each column *C* whose column descriptor is equivalent to a column descriptor of the row pattern input table is the value of *C* in row *R*.

2) The value of each row pattern measure column *RPMC* of *RPROT* is the value of the <value expression> that is the <row pattern measure expression> of *RPMC*, evaluated for (*STR*, RPP_i, *k*) with *R* as the current row.

iii) If WITH UNMATCHED ROWS is specified, then for each row *R* whose matched indicator is *False*, a row of *RPROT* is created, as follows:

1) The value of each column *C* whose column descriptor is equivalent to a column descriptor of the row pattern input table is the value of *C* in row *R*.

2) The value of each row pattern measure column is null.

12) The row pattern output table is the result of *RPRC*.

Conformance Rules

1) Without Feature R010, "Row pattern recognition: FROM clause", conforming SQL language shall not contain a <row pattern recognition clause>.

7.8 <row pattern measures>

Function

Specify the measure columns of a row pattern.

Format

```
<row pattern measures> ::=
  MEASURES <row pattern measure list>

<row pattern measure list> ::=
  <row pattern measure definition> [ { <comma> <row pattern measure definition> }... ]

<row pattern measure definition> ::=
  <row pattern measure expression> AS <measure name>

<row pattern measure expression> ::=
  <value expression>
```

Syntax Rules

1) <row pattern measures>shall not contain a <row pattern common syntax>.

2) <row pattern measures> shall not contain a <column reference> that is an outer reference.

3) <row pattern measures> shall not contain a <subquery> that contains a row pattern variable.

4) If <row pattern measures> is contained in <row pattern recognition clause>, then the row pattern measure columns of the row pattern output table are defined as follows. Let *NM* be the number of <row pattern measure definition>s contained in <row pattern measure list>. For each <row pattern measure definition> $RPMC_i$, 1 (one) ≤ i ≤ *NM*:

   a) Let $RPME_i$ be the <row pattern measure expression> simply contained in $RPMC_i$.

   b) Let CN_i be the <measure name> simply contained in $RPMC_i$.

   c) Let $RPMCD_i$ be the column descriptor of a column C_i, determined as follows:

      i) The declared type of C_i is the declared type of $RPME_i$.

      ii) The data type descriptor included in $RPMCD_i$ is the data type descriptor of the declared type of C_i. The column name included in $RPMCD_i$ is <measure name>.

      iii) C_i is called a *row pattern measure column*. $RPME_i$ is the definition of C_i. $RPME_i$ is said to *define* the row pattern measure column C_i. The descriptor of C_i is $RPMCD_i$.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without at least one of Feature R010, “Row pattern recognition: FROM clause”, or Feature R020, “Row pattern recognition: WINDOW clause”, conforming SQL language shall not contain a <row pattern measures>.

7.9 <row pattern common syntax>

Function

Specify syntax that is common to row patterns in <table reference> and in <window clause>.

Format

```
<row pattern common syntax> ::=
  [ AFTER MATCH <row pattern skip to> ]
      [ <row pattern initial or seek> ]
      PATTERN <left paren> <row pattern> <right paren>
      [ <row pattern subset clause> ]
      DEFINE <row pattern definition list>

<row pattern skip to> ::=
    SKIP TO NEXT ROW
  | SKIP PAST LAST ROW
  | SKIP TO FIRST <row pattern skip to variable name>
  | SKIP TO LAST <row pattern skip to variable name>
  | SKIP TO <row pattern skip to variable name>

<row pattern skip to variable name> ::=
  <row pattern variable name>

<row pattern initial or seek> ::=
    INITIAL
  | SEEK

<row pattern> ::=
    <row pattern term>
  | <row pattern alternation>

<row pattern alternation> ::=
  <row pattern> <vertical bar> <row pattern term>

<row pattern term> ::=
    <row pattern factor>
  | <row pattern term> <row pattern factor>

<row pattern factor> ::=
  <row pattern primary> [ <row pattern quantifier> ]

<row pattern quantifier> ::=
    <asterisk> [ <question mark> ]
  | <plus sign> [ <question mark> ]
  | <question mark> [ <question mark> ]
  | <left brace> [ <unsigned integer> ] <comma> [ <unsigned integer> ] <right brace>
      [ <question mark> ]
  | <left brace> <unsigned integer> <right brace>

<row pattern primary> ::=
    <row pattern primary variable name>
  | <dollar sign>
  | <circumflex>
  | <left paren> [ <row pattern> ] <right paren>
  | <left brace minus> <row pattern> <right minus brace>
  | <row pattern permute>

<row pattern primary variable name> ::=
  <row pattern variable name>
```

```
<row pattern permute> ::=
  PERMUTE <left paren> <row pattern> [ { <comma> <row pattern> }... ] <right paren>

<row pattern subset clause> ::=
  SUBSET <row pattern subset list>

<row pattern subset list> ::=
  <row pattern subset item> [ { <comma> <row pattern subset item> }... ]

<row pattern subset item> ::=
  <row pattern subset item variable name> <equals operator>
      <left paren> <row pattern subset rhs> <right paren>

<row pattern subset item variable name> ::=
  <row pattern variable name>

<row pattern subset rhs> ::=
  <row pattern subset rhs variable name>
      [ { <comma> <row pattern subset rhs variable name> }... ]

<row pattern subset rhs variable name> ::=
  <row pattern variable name>

<row pattern definition list> ::=
  <row pattern definition> [ { <comma> <row pattern definition> }... ]

<row pattern definition> ::=
  <row pattern definition variable name> AS <row pattern definition search condition>

<row pattern definition variable name> ::=
  <row pattern variable name>

<row pattern definition search condition> ::=
  <search condition>
```

Syntax Rules

1) Let *RPCS* be the <row pattern common syntax>.

2) *RPCS* shall not contain a <row pattern common syntax>.

3) Case:

   a) If *RPCS* is contained in a <row pattern recognition clause> *RPRC*, then <row pattern initial or seek> shall not be specified. Let *RPM* be the <row pattern measures> contained in *RPRC*, if any.

   b) If *RPCS* is contained in a <window clause> *WC*, then:

      i) Let *RPM* be the <row pattern measures> contained in *WC*, if any.

      ii) If <row pattern initial or seek> is not specified, then INITIAL is implicit.

4) Let *RP* be the <row pattern>. Let *RPS* be the <row pattern subset clause>. Let *NV* be the number of distinct <row pattern variable name>s simply contained in *RP* and *RPS*. These <correlation name>s are called *row pattern variables*. A row pattern variable is a range variable. Let the distinct row pattern variables be PV_i, 1 (one) $\leq i \leq NV$, in any order.

5) The scope of a row pattern variable *RPV* is the <row pattern recognition clause> or <window clause> that simply contains the <row pattern> or <row pattern subset rhs> in which *RPV* is declared.

6) The associated column list of a row pattern variable *RPV* comprises every column of the row pattern input table. The associated period list of *RPV* is empty.

7) A row pattern variable identified by a <row pattern primary variable name> is a *primary row pattern variable*.

8) A row pattern variable identified by a <row pattern subset item variable name> is a *union row pattern variable*.

9) A row pattern variable shall not be both a primary row pattern variable and a union row pattern variable.

10) Each <row pattern definition variable name> shall identify a primary row pattern variable.

11) Each <row pattern subset rhs variable name> shall identify a primary row pattern variable.

12) No two <row pattern definition variable name>s shall be equivalent.

13) A <row pattern definition search condition> shall not contain a <column reference> that is an outer reference.

14) A <row pattern definition search condition> shall not contain a <subquery> that contains a row pattern variable.

15) No two <row pattern subset item variable name>s shall be equivalent.

16) If *SRPV* is a union row pattern variable, then let *RPSI* be the <row pattern subset item> whose <row pattern subset item variable name> is equivalent to *SRPV*. Let *RHS* be the <row pattern subset rhs> of *RPSI*. Every primary row pattern variable that is equivalent to a <row pattern subset rhs variable name> is a *component* of *SRPV*.

17) If *RPV* is a primary row pattern variable that is not identified by any <row pattern definition variable name>, then the following <row pattern definition> is implicit:

```
RPV AS TRUE
```

18) Let *NPR* be the number of distinct primary row pattern variables. Let $PRPV_i$, 1 (one) $\leq i \leq NPR$, be an enumeration of the primary row pattern variables. Let *URPV* be an implementation-dependent (UV082) row pattern variable that is distinct from all range variables in the <SQL procedure statement> that simply contains *RPRC*. The following <row pattern subset item> is implicit:

```
URPV = (PRPV1, ..., PRPVNPR)
```

URPV is the *universal row pattern variable* of *RPRC*, and every primary row pattern variable is a component of *URPV*.

19) If *RP* contains a <row pattern quantifier>, then

Case:

a) <asterisk> <question mark> is equivalent to <left brace> 0 <comma> <right brace> <question mark> and the final <question mark> is not further transformed by another application of this rule.

b) <asterisk> is equivalent to <left brace> 0 <comma> <right brace>.

c) <plus sign> <question mark> is equivalent to <left brace> 1 <comma> <right brace> <question mark> and the final <question mark> is not further transformed by another application of this rule.

d) <plus sign> is equivalent to <left brace> 1 <comma> <right brace>.

e) <left brace> <comma> <right brace> <question mark> is equivalent to <left brace> 0 <comma> <right brace> <question mark> and the final <question mark> is not further transformed by another application of this rule.

f) <left brace> <comma> *UI* <right brace> <question mark>, where *UI* is an <unsigned integer>, is equivalent to <left brace> 0 <comma> *UI* <right brace> <question mark> and the final <question mark> is not further transformed by another application of this rule.

g) <left brace> <comma> <right brace> is equivalent to <left brace> 0 <comma> <right brace>.

h) <left brace> <comma> *UI* <right brace>, where *UI* is an <unsigned integer>, is equivalent to <left brace> 0 <comma> *UI* <right brace>.

i) <left brace> *UI* <right brace>, where *UI* is an <unsigned integer>, is equivalent to <left brace> *UI* <comma> *UI* <right brace>.

j) <question mark> <question mark> is equivalent to <left brace> 0 <comma> 1 <right brace> <question mark> and the final <question mark> is not further transformed by another application of this rule.

k) <question mark> is equivalent to <left brace> 0 <comma> 1 <right brace>.

20) If <left brace> <unsigned integer> <comma> <unsigned integer> <right brace> is specified, then let *VUI1* and *VUI2* be the values of the first and second <unsigned integer>s, respectively. *VUI1* shall be less than or equal to *VUI2*, and *VUI2* shall be greater than 0 (zero).

21) If *RP* contains a <row pattern permute> *RPPERM*, then let *NPERM* be the number of <row pattern>s simply contained in *RPPERM*. Let $SCRP_i$, 1 (one) ≤ *i* ≤ *NPERM*, be these <row pattern>s simply contained in *RPPERM*.

Let *Z* be the set of positive integers { 1 (one), ..., *NPERM* }. Let *PZ* be the set of permutations of *Z* (that is, the functions from *Z* to *Z* that are bijections).

NOTE 235 — A function $p:Z \rightarrow Z$ is a bijection if, for all a, b in Z, $a \neq b$, then $p(a) \neq p(b)$, and for all a in Z, there exists b in Z such that $p(b) = a$.

There are *NPERM*! (*NPERM* factorial, the product of all positive integers less than or equal to *NPERM*) different permutations in *PZ*.

PZ is ordered lexicographically as follows: for any *p*, *q* in *PZ*, *p* precedes *q* if there exists some *c* in *Z* such that for all $a < c$, $p(a) = q(a)$ and $p(c) < q(c)$.

For *h* between 1 (one) and *NPERM*!, let *q* be the *h*-th element of *PZ* in the lexicographic ordering of *PZ*, and let $PSCRP_h$ be the <row pattern>

($SCRP_{q(1)}$ $SCRP_{q(2)}$ ... $SCRP_{q(NPERM)}$)

RP is equivalent to the <row pattern> obtained by replacing *RPPERM* by

($PSCRP_1$ | $PSCRP_2$ | ... | $PSCRP_{NPERM!}$)

22) If *RPRC* specifies ALL ROWS PER MATCH WITH UNMATCHED ROWS, then a <row pattern primary> shall not contain a <left brace minus> or <right minus brace>.

23) Case:

a) If <row pattern skip to> is not specified, then AFTER MATCH SKIP PAST LAST ROW is implicit.

b) If SKIP TO <row pattern skip to variable name> is specified, then let *RPSTVN* be that <row pattern skip to variable name>; the <row pattern skip to> is equivalent to

SKIP TO LAST *RPSTVN*

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without at least one of Feature R010, "Row pattern recognition: FROM clause", or Feature R020, "Row pattern recognition: WINDOW clause", conforming SQL language shall not contain <row pattern common syntax>.

7.10 <joined table>

Function

Specify a table derived from a Cartesian product, inner join, or outer join.

Format

```
<joined table> ::=
    <cross join>
  | <qualified join>
  | <natural join>

<cross join> ::=
  <table reference> CROSS JOIN <table factor>

<qualified join> ::=
  { <table reference> | <partitioned join table> }
      [ <join type> ] JOIN
      { <table reference> | <partitioned join table> }
      <join specification>

<partitioned join table> ::=
  <table factor> PARTITION BY
      <partitioned join column reference list>

<partitioned join column reference list> ::=
  <left paren> <partitioned join column reference>
      [ { <comma> <partitioned join column reference> }... ]
      <right paren>

<partitioned join column reference> ::=
  <column reference>

<natural join> ::=
  { <table reference> | <partitioned join table> }
      NATURAL [ <join type> ] JOIN
      { <table factor> | <partitioned join table> }

<join specification> ::=
    <join condition>
  | <named columns join>

<join condition> ::=
  ON <search condition>

<named columns join> ::=
  USING <left paren> <join column list> <right paren> [ AS <join correlation name> ]

<join correlation name> ::=
  <correlation name>

<join type> ::=
    INNER
  | <outer join type> [ OUTER ]

<outer join type> ::=
    LEFT
  | RIGHT
  | FULL
```

```
<join column list> ::=
  <column name list>
```

Syntax Rules

1) Let *JT* be a <joined table>. Let *JTT* be the table specified by *JT*.

2) *JT* shall not generally contain a <data change delta table>.

3) Let *TRA* be the <table reference> or <table factor> that is the first operand of *JT*, and let *TRB* be the <table reference> or <table factor> that is the second operand of *JT*. Let *RTA* and *RTB* be the tables specified by *TRA* and *TRB*, respectively. Let *CP* be:

```
SELECT *
FROM TRA, TRB
```

4) If *TRB* contains a <lateral derived table> containing an outer reference that references *TRA*, then <join type> shall not contain RIGHT or FULL.

5) If a <qualified join> or <natural join> is specified and a <join type> is not specified, then INNER is implicit.

6) If a <qualified join> containing a <join condition> is specified and a <value expression> directly contained in the <search condition> is a <set function specification>, then *JT* shall be contained in a <having clause> or <select list>, the <set function specification> shall contain an aggregated argument *AA* that contains an outer reference, and every column reference contained in *AA* shall be an outer reference.

   NOTE 236 — "outer reference" is defined in Subclause 6.7, "<column reference>".

7) The <search condition> shall not contain a <window function> without an intervening <query expression>.

8) If neither NATURAL is specified nor a <join specification> immediately containing a <named columns join> is specified, then the degree and the descriptors of the columns of *JTT* are the same as the degree and the descriptors of the columns, respectively, of *CP*, with the possible exception of the nullability characteristics of the columns. The row type of *JT* is the same as the row type of *CP*.

9) If a <partitioned join table> *PJT* is specified, then:

   a) The <qualified join> or <natural join> shall specify an <outer join type>.

   b) Each <partitioned join column reference> shall uniquely reference a column of the table specified by the <table primary> simply contained in *PJT*. Such a column is called a *join partitioning column*.

   c) If the first operand of the <qualified join> or <natural join> is a <partitioned join table>, then the <outer join type> shall be RIGHT or FULL.

   d) If the second operand of the <qualified join> or <natural join> is a <partitioned join table>, then the <outer join type> shall be LEFT or FULL, and *TRB* shall not contain a <lateral derived table> containing an outer reference that references *TRA*.

10) If NATURAL is specified or if a <join specification> immediately containing a <named columns join> is specified, then:

    a) Case:

       i) If *JT* simply contains a <join correlation name> *JCV*, then let *RV* be *JCV*. *RV* shall not be equivalent to any range variable of the outermost <query specification> containing *JT* or to any range variable of *TRA* or *TRB*.

ii) Otherwise, let *RV* be an implementation-dependent (UV083) <correlation name> that is not equivalent to any range variable of the outermost <query specification> containing *JT* or to any range variable of *TRA* or *TRB*.

b) If NATURAL is specified, then let *common column name* be a <column name> that is equivalent to the <column name> of exactly one column of *RTA* and the <column name> of exactly one column of *RTB*. *RTA* shall not have any duplicate common column names and *RTB* shall not have any duplicate common column names. Let *corresponding join columns* refer to all columns of *RTA* and *RTB* that have common column names, if any.

c) If a <named columns join> is specified, then every <column name> in the <join column list> shall be equivalent to the <column name> of exactly one column of *RTA* and the <column name> of exactly one columns of *RTB*. Let *common column name* be the name of such a column. Let *corresponding join columns* refer to the columns identified in the <join column list>.

d) A corresponding join column shall not be a join partitioning column.

e) Let *CA* and *CB* be a pair of corresponding join columns of *RTA* and *RTB*, respectively. *CA* and *CB* shall be comparable. *CA* and *CB* are operands of an equality operation, and the Syntax Rules and Conformance Rules of Subclause 9.11, "Equality operations", apply.

f) If there is at least one corresponding join column, then:

i) Let *N* be the number of corresponding join columns.

ii) For each *i*, 1 (one) ≤ *i* ≤ *N*, let $CJCN_i$ be the *i*-th common column name, taken in order of their ordinal positions in *RTA*. Let RVA_i be the range variable of *TRA* whose associated column name list includes $CJCN_i$. Let RVB_i be the range variable of *TRB* whose associated column list includes $CJCN_i$.

iii) Let *SLCC* be a <select list> of <derived column>s of the form

```
COALESCE ( RVA_i.CJCN_i, RVB_i.CJCN_i ) AS CJCN_i
```

for every *i*, 1 (one) ≤ *i* ≤ *N*, in ascending order.

iv) For each *i*, 1 (one) ≤ *i* ≤ *N*, $CJCN_i$ is removed from the associated column name list of both RVA_i and RVB_i.

g) If *RTA* contains at least one column that is not a corresponding join column, then:

i) Let *NCA* be the number of columns of *RTA* that are not corresponding join columns.

ii) For each *j*, 1 (one) ≤ *j* ≤ *NCA*, let CA_j be the name of the *j*-th column that is not a corresponding join column, taken in order of their ordinal positions in *RTA*. Let $RVCA_j$ be the range variable of *TRA* whose associated column name list includes CA_j.

iii) Let *SLTA* be a <select list> of <derived column>s of the form

```
RVCA_j.CA_j
```

for every *j*, 1 (one) ≤ *j* ≤ *NCA*, in ascending order.

h) If *RTB* contains at least one column that is not a corresponding join column, then:

i) Let *NCB* be the number of columns of *RTB* that are not corresponding join columns.

ii) For each *k*, 1 (one) ≤ *k* ≤ *NCB*, let CB_k be the name of the *k*-th column that is not a corresponding join column, taken in order of their ordinal positions in *RTB*. Let $RVCB_k$ be the range variable of *TRB* whose associated column name list includes CB_k.

iii) Let *SLTB* be a <select list> of <derived column>s of the form

$RVCB_k$. CB_k

for every k, 1 (one) ≤ k ≤ *NCB*, in ascending order.

i) Let the <select list> *SL* be defined as

Case:

i) If all of the columns of *RTA* and *RTB* are corresponding join columns, then let *SL* be "*SLCC*".

ii) If *RTA* contains no corresponding join columns and *RTB* contains no corresponding join columns, then let *SL* be "*SLTA*, *SLTB*".

iii) If *RTA* contains no columns other than corresponding join columns, then let *SL* be "*SLCC*, *SLTB*".

iv) If *RTB* contains no columns other than corresponding join columns, then let *SL* be "*SLCC*, *SLTA*".

v) Otherwise, let *SL* be "*SLCC*, *SLTA*, *SLTB*".

j) The descriptors of the columns of the result of *JT*, with the possible exception of the nullability characteristics of the columns, are the same as the descriptors of the columns of the result of

```
SELECT SL FROM TRA, TRB
```

11) Let *NA* be the number of range variables of *TRA*. Let *NB* be the number of range variables of *TRB*.

a) Case:

i) If NATURAL is specified or if a <join specification> immediately containing a <named columns join> is specified, then:

1) The number of range variables of *JT* is the sum of *NA*, *NB*, and 1 (one).

2) The name of the i-th range variable of *JT*, 1 (one) ≤ i ≤ *NA*, is same as the i-th range variable of *TRA*. The name of the (*NA* + i)-th range variable of *JT*, 1 (one) ≤ i ≤ *NB*, is same as the i-th range variable of *TRB*. The name of the (*NA*+*NB*+1)-th range variable is *RV*.

3) Let *NC* be the number of corresponding join columns. For i, 1 (one) ≤ i ≤ *NC*, let CJC_i be the i-th column of *JT*. Let *CJCL* be the list of columns formed by CJC_i, 1 (one) ≤ i ≤ *NC*, in ascending order.

4) The associated column list of the i-th range variable of *JT*, 1 (one) ≤ i ≤ *NA*, is same as the associated column list of the i-th range variable of *TRA*. The associated column list of the (*NA* + i)-th range variable of *JT*, 1 (one) ≤ i ≤ *NB*, is the same as the associated column list of the i-th range variable of *TRB*. The column list associated with the (*NA*+*NB*+1)-th range variable is *CJCL*.

NOTE 237 — Corresponding join columns have previously been removed from the associated column lists of the range variables of both *TRA* and *TRB*.

ii) Otherwise:

1) The number of range variables of *JT* is the sum of *NA* and *NB*.

2) The name of *i*-th range variable of *JT*, 1 (one) ≤ *i* ≤ *NA*, is same as the *i*-th range variable of *TRA*. The name of the (*NA* + *i*)-th range variable of *JT*, 1 (one) ≤ *i* ≤ *NB*, is same as the *i*-th range variable of *TRB*.

3) The associated column list of the *i*-th range variable of *JT*, 1 (one) ≤ *i* ≤ *NA*, is same as the associated column list of the *i*-th range variable of *TRA*. The associated column list of the (*NA* + *i*)-th range variable of *JT*, 1 (one) ≤ *i* ≤ *NB*, is same as the associated column list of the *i*-th range variable of *TRB*.

b) The associated period list of each of the range variables of *JT* is empty.

c) The table associated with each of the range variables of *JT* is *JTT*.

d) The range variables of *JT* have no scope.

e) Every range variable of *JT* is *exposed* by *JT*.

12) The declared type of the rows of *JT* is the row type *RT* defined by the sequence of (<field name>, <data type>) pairs indicated by the sequence of column descriptors of *JT* taken in order.

13) A column *CR* of the result of the <joined table> is *readily known not null* if and only if all of the following are true:

a) The <joined table> does not simply contain an <outer join type>.

b) The <joined table> does not simply contain NATURAL.

c) The <joined table> does not simply contain a <partitioned join table>.

d) *CR* is a column reference that references a column that is readily known not null.

14) A column *CR* of the result of the <joined table> is *known not null* if and only if at least one of the following is true:

a) *CR* is readily known not null.

b) If the SQL-implementation supports Feature T101, "Enhanced nullability determination", then *CR* is not possibly nullable according to the following:

i) For every column *CR* of the result of the <joined table> that corresponds to a field *CA* of *RTA* that is not a corresponding join column or a join partitioning column, *CR* is *possibly nullable* if and only if exactly one of the following is true:

1) RIGHT or FULL is specified.

2) INNER, LEFT, or CROSS JOIN is specified or implicit and *CA* is possibly nullable.

ii) For every column *CR* of the result of the <joined table> that corresponds to a field *CB* of *RTB* that is not a corresponding join column or a join partitioning column, *CR* is *possibly nullable* if and only if exactly one of the following is true:

1) LEFT or FULL is specified.

2) INNER, RIGHT, or CROSS JOIN is specified or implicit and *CB* is possibly nullable.

iii) For every column *CR* of the result of the <joined table> that corresponds to a corresponding join column *CA* of *RTA* and a corresponding join column *CB* of *RTB*, *CR* is *possibly nullable* if and only if exactly one of the following is true:

1) LEFT or FULL is specified and *CA* is possibly nullable, or

2) RIGHT or FULL is specified and *CB* is possibly nullable.

iv) A column *CR* of the result of the <joined table> that corresponds to a join partitioning column *JPC* is *possibly nullable* if *JPC* is possibly nullable.

c) *CR* conforms to an implementation-defined (IA217) rule that correctly infers that the value of *CR* cannot be null.

Access Rules

1) If there is at least one common column name, then for each i, 1 (one) $\leq i \leq N$:

a) If the range variable RVA_i is associated with a base table or a viewed table *TA*, then

Case:

i) If the <joined table> is contained in a <search condition> immediately contained in an <assertion definition> or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on the column of *TA* that is identified by $CJCN_i$.

ii) If the <joined table> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on the column of *TA* that is identified by $CJCN_i$.

iii) Otherwise, the current privileges shall include SELECT on the column of *TA* that is identified by $CJCN_i$.

b) If the range variable RVB_i is associated with a base table or a viewed table *TB*, then

Case:

i) If the <joined table> is contained in a <search condition> immediately contained in an <assertion definition> or a <check constraint definition>, then the applicable privileges for the <authorization identifier> that owns the containing schema shall include REFERENCES on the column of *TB* that is identified by $CJCN_i$.

ii) If the <joined table> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include SELECT on the column of *TB* that is identified by $CJCN_i$.

iii) Otherwise, the current privileges shall include SELECT on the column of *TB* that is identified by $CJCN_i$.

General Rules

1) Case:

a) If a <cross join> is specified, then let *T* be *CP*.

b) If a <join condition> is specified, then let *SC* be the <search condition> and let *T* be

```
CP
WHERE SC
```

c) If NATURAL is specified or <named columns join> is specified, then

Case:

i) If there are corresponding join columns, then let *T* be

```
CP
WHERE RVA1.CJCN1 = RVB1.CJCN1
  AND ...
  AND RVAN.CJCNN = RVBN.CJCNN
```

ii) Otherwise, let *T* be *CP*.

2) Let *TR* be the result of evaluating *T*, let *DA* and *DB* be the degrees of *TRA* and *TRB*, respectively, and let *TVA* be the result of evaluating *TRA*. Let *TN* be an effective name for *T*.

3) If LEFT is specified and the second operand does not specify <partitioned join table>, then:

a) Let *PA* be the collection of rows of *TVA* for which there exists some row *R* in *TR* and some row *RA* in *TVA* such that the values of the first *DA* fields of *R* are identical to the values of the corresponding fields of *RA*.

b) Let *UA* be those rows of *TVA* that are not in *PA*.

c) Let *XA* be *UA* extended on the right with *DB* columns containing the null value.

d) Let *XNA* be an effective distinct name for *XA*.

4) If RIGHT or FULL is specified or if LEFT is specified and the second operand specifies <partitioned join table>, then:

a) Let *TVB* be the result of evaluating *TRB*.

NOTE 238 — It follows from the Syntax Rules that *TRB* does not contain a <lateral derived table> containing an outer reference that references *TRA*. This ensures that it is possible to evaluate *TRB* in isolation.

b) Case:

i) If the first operand specifies <partitioned join table>, then

Case:

1) If *TVA* is empty, then let *NA* be 0 (zero).

2) Otherwise, *TVA* is partitioned into the minimum numbers of partitions such that for each join partitioning column *JPC* of each partition, no two values of *JPC* are distinct. If the declared type of a join partitioning column is a user-defined type and the comparison of that column results in *Unknown* for two rows of *TVA*, then the assignment of those rows to partitions is implementation-dependent (UA052). Let *NA* be the number of partitions. Let GA_1, ..., GA_{NA} be an enumeration of the partitions.

ii) Otherwise, let *NA* be 1 (one), and let GA_1 be *TVA*.

c) Case:

i) If the second operand specifies <partitioned join table>, then

Case:

1) If *TVB* is empty, then let *NB* be 0 (zero).

2) Otherwise, *TVB* is partitioned into the minimum numbers of partitions such that for each join partitioning column *JPC* of each partition, no two values of *JPC* are distinct. If the declared type of a join partitioning column is a user-defined type and the comparison of that column results in *Unknown* for two rows of *TVB*, then the assignment of those rows to partitions is implementation-dependent (UA052).

Let *NB* be the number of partitions. Let GB_1, ..., GB_{NB} be an enumeration of the partitions.

ii) Otherwise, let *NB* be 1 (one), and let GB_1 be *TVB*.

d) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $PA_{i,j}$ be the collection of rows *RA* of GA_i for which there exists a row *RB* in GB_j such that the concatenation of *RA* and *RB* is in *TR*.

e) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $PB_{i,j}$ be the collection of rows *RB* of GB_j for which there exists a row *RA* in GA_i such that the concatenation of *RA* and *RB* is in *TR*.

f) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $UA_{i,j}$ be the collection of rows of GA_i that are not in $PA_{i,j}$.

g) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $UB_{i,j}$ be the collection of rows of GB_j that are not in $PB_{i,j}$.

h) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $XA_{i,j}$ be $UA_{i,j}$ extended on the right with *DB* columns, with declared types and values determined as follows. For each *k* between 1 (one) and *DB*, the declared type of the (*DA* + *k*)-th column is the declared type of the *k*-th column of *TVB*, and the value is

Case:

i) If the *k*-th column of *TVB* is a join partitioning column, then the common value of the *k*-th column of GB_j.

ii) Otherwise, the null value.

i) For each *i* between 1 (one) and *NA*, and for each *j* between 1 (one) and *NB*, let $XB_{i,j}$ be $UB_{i,j}$ extended on the left with *DA* columns, with declared types and values determined as follows. For each *k* between 1 (one) and *DA*, the declared type of the *k*-th column is the declared type of the *k*-th column of *TVA*, and the value is

Case:

i) If the *k*-th column of *TVA* is a join partitioning column, then the common value of the *k*-th column of GA_i.

ii) Otherwise, the null value.

j) Let *XA* be the collection of all rows in $XA_{i,j}$ for all *i* between 1 (one) and *NA* and all *j* between 1 (one) and *NB*.

NOTE 239 — If *NA* is 0 (zero), then *XA* is empty.

k) Let *XB* be the collection of all rows in $XB_{i,j}$, for all *i* between 1 (one) and *NA* and all *j* between 1 (one) and *NB*.

NOTE 240 — If *NB* is 0 (zero), then *XB* is empty.

l) Let *XNA* be an effective distinct name for *XA* and let *XNB* be an effective distinct name for *XB*.

5) Case:

a) If INNER or <cross join> is specified, then let *S* be *TR*.

b) If LEFT is specified, then let *S* be the result of:

```
SELECT *
FROM TN
UNION ALL
SELECT *
FROM XNA
```

c) If RIGHT is specified, then let *S* be the result of:

```
SELECT *
FROM TN
UNION ALL
SELECT *
FROM XNB
```

d) If FULL is specified, then let *S* be the result of:

```
SELECT *
FROM TN
UNION ALL
SELECT *
FROM XNA
UNION ALL
SELECT *
FROM XNB
```

6) Let *SN* be an effective name of *S*.

Case:

a) If NATURAL is specified or a <named columns join> is specified, then:

i) Let CS_i be a distinct name for the *i*-th column of *S*. Column CS_i of *S* corresponds to the *i*-th field of *RTA* if *i* is less than or equal to *DA*. Column CS_j of *S* corresponds to the (*j*–*DA*)-th field of *RTB* for *j* greater than *DA*.

ii) If there is at least one corresponding join column, then let *SLCC* be a <select list> of derived columns of the form

COALESCE (CS_i, CS_j)

for every pair of columns CS_i and CS_j, where CS_i and CS_j correspond to fields of *RTA* and *RTB* that are a pair of corresponding join columns.

iii) If *RTA* contains one or more fields that are not corresponding join columns, then let *SLTA* be a <select list> of the form:

CS_i

for every column CS_i of *S* that corresponds to a field of *RTA* that is not a corresponding join column, taken in order of their ordinal position in *S*.

iv) If *RTB* contains one or more fields that are not corresponding join columns, then let *SLTB* be a <select list> of the form:

CS_j

for every column CS_j of *S* that corresponds to a field of *RTB* that is not a corresponding join column, taken in order of their ordinal position in *S*.

v) Let the <select list> *SL* be defined as

Case:

1) If all the fields of *RTA* and *RTB* are corresponding join columns, then let *SL* be

SLCC

2) If *RTA* contains no corresponding join columns and *RTB* contains no corresponding join columns, then let *SL* be

SLTA, *SLTB*

3) If *RTA* contains no fields other than corresponding join columns, then let *SL* be

SLCC, *SLTB*

4) If *RTB* contains no fields other than corresponding join columns, then let *SL* be

SLCC, *SLTA*

5) Otherwise, let *SL* be

SLCC, *SLTA*, *SLTB*

vi) The result of the <joined table> is the result of:

```
SELECT SL
FROM SN
```

b) Otherwise, the result of the <joined table> is *S*.

Conformance Rules

1) Without Feature F407, "CROSS JOIN", conforming SQL language shall not contain a <cross join>.

2) Without Feature F405, "NATURAL JOIN", conforming SQL language shall not contain a <natural join>.

3) Without Feature F406, "FULL OUTER JOIN", conforming SQL language shall not contain an <outer join type> that immediately contains FULL.

4) Without Feature F402, "Named column joins for LOBs, arrays, and multisets", conforming SQL language shall not contain a <joined table> that simply contains either <natural join> or <named columns join> in which, if *C* is a corresponding join column, the declared type of *C* is LOB-ordered, array-ordered, or multiset-ordered.

NOTE 241 — If *C* is a corresponding join column, then the Conformance Rules of Subclause 9.11, "Equality operations", also apply.

5) Without Feature F403, "Partitioned join tables", conforming SQL language shall not contain <partitioned join table>.

6) Without Feature F404, "Range variable for common column names", conforming SQL language shall not contain a <named columns join> that simply contains a <correlation name>.

7.11 <JSON table>

Function

Query a JSON text and present it as a relational table.

Format

```
<JSON table> ::=
  JSON_TABLE <left paren>
      <JSON API common syntax>
      <JSON table columns clause>
      [ <JSON table plan clause> ]
      [ <JSON table error behavior> ON ERROR ]
  <right paren>

<JSON table columns clause> ::=
  COLUMNS <left paren>
      <JSON table column definition> [ { <comma> <JSON table column definition> }... ]
  <right paren>

<JSON table column definition> ::=
    <JSON table ordinality column definition>
  | <JSON table regular column definition>
  | <JSON table formatted column definition>
  | <JSON table nested columns>

<JSON table ordinality column definition> ::=
  <column name> FOR ORDINALITY

<JSON table regular column definition> ::=
  <column name> <data type>
      [ PATH <JSON table column path specification> ]
      [ <JSON table column empty behavior> ON EMPTY ]
      [ <JSON table column error behavior> ON ERROR ]

<JSON table column empty behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>

<JSON table column error behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>

<JSON table column path specification> ::=
  <JSON path specification>

<JSON table formatted column definition> ::=
  <column name> <data type>
      [ FORMAT <JSON representation> ]
      [ PATH <JSON table column path specification> ]
      [ <JSON table formatted column wrapper behavior> WRAPPER ]
      [ <JSON table formatted column quotes behavior> QUOTES [ ON SCALAR STRING ] ]
      [ <JSON table formatted column empty behavior> ON EMPTY ]
      [ <JSON table formatted column error behavior> ON ERROR ]

<JSON table formatted column wrapper behavior> ::=
    WITHOUT [ ARRAY ]
  | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]
```

```
<JSON table formatted column quotes behavior> ::=
    KEEP
  | OMIT

<JSON table formatted column empty behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<JSON table formatted column error behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<JSON table nested columns> ::=
  NESTED [ PATH ] <JSON table nested path specification>
      [ AS <JSON table nested path name> ]
  <JSON table columns clause>

<JSON table nested path specification> ::=
  <JSON path specification>

<JSON table nested path name> ::=
  <JSON table path name>

<JSON table path name> ::=
  <identifier>

<JSON table plan clause> ::=
    <JSON table specific plan>
  | <JSON table default plan>

<JSON table specific plan> ::=
  PLAN <left paren> <JSON table plan> <right paren>

<JSON table plan> ::=
    <JSON table path name>
  | <JSON table plan parent/child>
  | <JSON table plan sibling>

<JSON table plan parent/child> ::=
    <JSON table plan outer>
  | <JSON table plan inner>

<JSON table plan outer> ::=
  <JSON table path name> OUTER <JSON table plan primary>

<JSON table plan inner> ::=
  <JSON table path name> INNER <JSON table plan primary>

<JSON table plan sibling> ::=
    <JSON table plan union>
  | <JSON table plan cross>

<JSON table plan union> ::=
  <JSON table plan primary> UNION <JSON table plan primary>
      [ { UNION <JSON table plan primary> }... ]

<JSON table plan cross> ::=
  <JSON table plan primary> CROSS <JSON table plan primary>
      [ { CROSS <JSON table plan primary> }... ]

<JSON table plan primary> ::=
```

```
    <JSON table path name>
  | <left paren> <JSON table plan> <right paren>

<JSON table default plan> ::=
  PLAN DEFAULT <left paren>
      <JSON table default plan choices> <right paren>

<JSON table default plan choices> ::=
    <JSON table default plan inner/outer>
        [ <comma> <JSON table default plan union/cross> ]
  | <JSON table default plan union/cross>
        [ <comma> <JSON table default plan inner/outer> ]

<JSON table default plan inner/outer> ::=
    INNER | OUTER

<JSON table default plan union/cross> ::=
    UNION | CROSS

<JSON table error behavior> ::=
    ERROR
  | EMPTY

<JSON table primitive> ::=
  JSON_TABLE_PRIMITIVE <left paren>
      <JSON API common syntax>
      <JSON table primitive columns clause>
      <JSON table error behavior> ON ERROR
  <right paren>

<JSON table primitive columns clause> ::=
  COLUMNS <left paren>
      <JSON table primitive column definition>
      [ { <comma> <JSON table primitive column definition> }... ]
  <right paren>

<JSON table primitive column definition> ::=
    <JSON table ordinality column definition>
  | <JSON table regular column definition>
  | <JSON table formatted column definition>
  | <JSON table primitive chaining column>

<JSON table primitive chaining column> ::=
  <column name> FOR CHAINING
```

Syntax Rules

1) If <JSON table> *JTAB* is specified:

   a) Let *JACS* be the <JSON API common syntax> simply contained in *JTAB*.

      i) Let *JACSCI* be the <JSON context item> contained in *JACS*.

      ii) Let *JACSPATH* be the <JSON path specification> simply contained in *JACS*.

      iii) Let *JACSPN* be the explicit or implicit <JSON table path name> simply contained in *JACS*.

      iv) If *JACS* simply contains <JSON passing clause>, then let *JACSPC* be that <JSON passing clause>; otherwise, let *JACSPC* be the zero-length character string.

   b) If <JSON table error behavior> is not specified, then EMPTY ON ERROR is implicit. Let *JTEB* be the explicit or implicit <JSON table error behavior>.

c) Let *JTABCOLS* be the <JSON table columns clause> simply contained in *JTAB*.

d) Throughout *JTAB*, there is a three-way association between <JSON table path name>s, <JSON path specification>s, and <JSON table columns clause>s. At the outermost level, *JACSPN*, *JACSPATH*, and *JTABCOLS* are associated. Additional associations are defined later for each <JSON table nested columns> contained in *JTAB*. The <JSON table path name> and the <JSON table columns clause> in an association uniquely determines the other members of the association.

e) For every <JSON table regular column definition> *JTRCD* contained in *JTAB*:

i) Let *JTRCN* be the <column name> simply contained in *JTRCD*.

ii) The <data type> *JTRCDT* contained in *JTRCD* shall be a <predefined type> that identifies a character string data type, numeric data type, Boolean data type, or datetime data type.

iii) If *JTRCD* does not contain <JSON table column empty behavior>, then the implicit <JSON table column empty behavior> of *JTRCD* is NULL ON EMPTY.

iv) If *JTRCD* does not contain <JSON table column error behavior>, then

Case:

1) If *JTEB* is ERROR ON ERROR, then the implicit <JSON table column error behavior> of *JTRCD* is ERROR ON ERROR.

2) Otherwise, the implicit <JSON table column error behavior> of *JTRCD* is NULL ON ERROR.

v) Case:

1) If *JTRCD* contains <JSON table column path specification>, then let *JTRCPATH* be the <JSON table column path specification> contained in *JTRCD*.

2) Otherwise, let *JTRCPATH* be a <character string literal> consisting of the characters <dollar sign> <period>, followed by a JSON string whose value is the same characters as in *JTRCN*, in order.

vi) The Syntax Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with *JTRCPATH* as *PATH SPECIFICATION* and the zero-length character string as *PASSING CLAUSE*.

f) For every <JSON table formatted column definition> *JTQCD* contained in *JTAB*:

i) Let *JTQCN* be the <column name> simply contained in *JTQCD*.

ii) Let *JTQCDT* be the <data type> contained in *JTQCD*.

Case:

1) If *JTQCDT* does not identify a JSON type, then <JSON representation> shall be specified.

2) Otherwise, if <JSON representation> is not specified, then FORMAT JSON is implicit.

iii) Let *JTQCFO* be the explicit or implicit <JSON representation> contained in *JTQCD*.

iv) If *JTQCFO* is JSON, then *JTQCDT* shall be a string type or a JSON type.

v) If *JTQCD* does not contain <JSON table formatted column wrapper behavior>, then the implicit <JSON table formatted column wrapper behavior> is WITHOUT ARRAY.

vi) If *JTQCD* contains <JSON table formatted column wrapper behavior> that specifies WITH and neither CONDITIONAL nor UNCONDITIONAL is specified, then UNCONDITIONAL is implicit.

vii) If *JTQCD* contains <JSON table formatted column wrapper behavior> that specifies WITH, then <JSON table formatted column quotes behavior> shall not be specified.

viii) If *JTQCD* does not specify <JSON table formatted column quotes behavior>, then KEEP is implicit.

ix) If *JTQCDT* is a JSON type, then the implicit or explicit <JSON table formatted column quotes behavior> shall specify KEEP.

x) If *JTQCD* does not contain <JSON table formatted column empty behavior>, then the implicit <JSON table formatted column empty behavior> of *JTQCD* is NULL ON EMPTY.

xi) If *JTQCD* does not contain <JSON table formatted column error behavior>, then

Case:

1) If *JTEB* is ERROR ON ERROR, then the implicit <JSON table formatted column error behavior> of *JTQCD* is ERROR ON ERROR.

2) Otherwise, the implicit <JSON table formatted column error behavior> of *JTQCD* is NULL ON ERROR.

xii) Case:

1) If *JTQCD* contains <JSON table column path specification>, then let *JTQCPATH* be the <JSON table column path specification> contained in *JTQCD*.

2) Otherwise, let *JTQCPATH* be a <character string literal> consisting of the characters <dollar sign> <period>, followed by a JSON string whose value is the same characters as in *JTQCN* in order.

xiii) The Syntax Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with *JTQCPATH* as *PATH SPECIFICATION* and the zero-length character string as *PASSING CLAUSE*.

g) For every <JSON table nested columns> *JTNC* contained in *JTAB*:

i) If *JTAB* contains an explicit <JSON table plan clause>, then *JTNC* shall contain <JSON table nested path name>.

ii) If <JSON table path name> is not specified, then an implementation-dependent (UV084) <JSON table path name> is implicit.

iii) The scope of an explicit or implicit <JSON table path name> *JTNCPN* is the explicit or implicit <JSON table plan clause>.

iv) Let *JTNCPATH* be the <JSON table nested path specification> of *JTNC*.

v) The Syntax Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with *JTRCPATH* as *PATH SPECIFICATION* and the zero-length character string as *PASSING CLAUSE*.

vi) Let *JTNCCOLS* be the <JSON table columns clause> simply contained in *JTNC*.

vii) *JTNCPN*, *JTNCPATH*, and *JTNCCOLS* are associated.

h) Within *JTAB*, *JACSPN*, every <JSON table path name> contained in a <JSON table nested columns>, and every <column name> shall be distinct from one another.

NOTE 242 — In the syntactic transformation defined later, <JSON table path name>s are used as column names; hence <JSON table path name>s must be distinct from the <column name>s.

i) If <JSON table specific plan> is not specified, then an implicit <JSON table specific plan> is constructed as follows:

i) Case:

1) If <JSON table default plan choices> is specified and contains INNER, then let *JTPDEFPC* be INNER.

2) Otherwise, let *JTPDEFPC* be OUTER.

NOTE 243 — The choice of INNER or OUTER determines whether the implicit <JSON table specific plan> will perform inner or left outer joins between parent and child NESTED COLUMNS clauses.

ii) Case:

1) If <JSON table default plan choices> is specified and contains CROSS, then let *JTPDEFSIB* be CROSS.

2) Otherwise, let *JTPDEFSIB* be UNION.

NOTE 244 — The choice of CROSS or UNION determines whether the implicit <JSON table specific plan> will perform a cross product or a "union join" between sibling NESTED COLUMNS clauses. A union join is a full outer join whose match condition is always *False*, producing a list of every row of the first operand, null extended on the right, followed by a list of every row of the second operand, null extended on the left.

iii) Let *CN* be the number of <JSON table columns clause>s contained in *JTAB* without an intervening <JSON table>.

iv) Let $CNODE_i$, 1 (one) $\leq i \leq CN$, be an enumeration of the <JSON table columns clause>s contained in *JTAB* without an intervening <JSON table>, enumerated in order of the occurrence of the <key word> COLUMNS immediately contained in those <JSON table columns clause>s in *JTAB*. Let *CNODELIST* be an ordered collection of $CNODE_i$, 1 (one) $\leq i \leq CN$.

v) Let *COLTREE* be a tree whose nodes are $CNODE_i$, 1 (one) $\leq i \leq CN$, arranged according to syntactic containment.

NOTE 245 — That is, for every node *N*, the children of *N* are the <JSON table columns clause>s that are simply contained in *N*, if any.

NOTE 246 — $CNODE_i$, 1 (one) $\leq i \leq CN$, is an ordered list of nodes, *CNODELIST*. Each of those $CNODE_i$ is also a node in *COLTREE*. This permits the processing of nodes in lexicographic order (from the list) while still having access to those nodes' children (from the tree).

vi) Let CPN_c, 1 (one) $\leq c \leq CN$, be the <JSON table path name> associated with $CNODE_c$.

vii) Let $CHILDREN_c$, 1 (one) $\leq c \leq CN$, be the set of child nodes of $CNODE_c$, considered as nodes of *COLTREE*. Let $NCHILD_c$ be the number of nodes in $CHILDREN_c$.

viii) For all *c*, $CN \geq c \geq 1$ (one), in descending order,

Case:

1) If $NCHILD_c$ is 0 (zero), then let $CPLAN_c$ be CPN_c.

NOTE 247 — These are the leaf nodes of *COLTREE*.

2) If $NCHILD_c$ is 1 (one), then let *DNODE* be the only child node of $CNODE_c$, let *dsub* be the ordinal position of *DNODE* in *CNODELIST*, and let $CPLAN_c$ be

(CPN_c JTPDEFPC $CPLAN_{dsub}$)

NOTE 248 — This means that either a left outer join or an inner join will connect the evaluation of the parent <JSON table columns clause> to the evaluation of the child <JSON table columns clause>.

3) Otherwise,

A) Let *NC* be $NCHILD_c$.

B) Let $DNODE_{cn}$, 1 (one) ≤ *cn* ≤ *NC*, be the collection of nodes in $CHILDREN_c$, let *dsub*(*cn*) be the ordinal position of $DNODE_{cn}$ in *CNODELIST*, and let $CPLAN_c$ be

(CPN_c JTPDEFPC
($CPLAN_{dsub(1)}$ JTPDEFPSIB ...
JTPDEFPSIB $CPLAN_{dsub(NC)}$))

NOTE 249 — The subscripts *dsub*(1), ..., *dsub*(*NC*) are not necessarily consecutive integers.

NOTE 250 — This means that the child <JSON table columns clause>s will be connected using either a "union join" (the same as a full outer join with a condition that is never satisfied) or a cross product. Either a left outer join or an inner join will connect the parent <JSON table columns clause> to the union join or cross product of all the children.

ix) The implicit <JSON table plan clause> of *JTAB* is

PLAN ($CPLAN_1$)

NOTE 251 — For example, consider the following schematic <JSON table>:

```
JSON_TABLE ( JACSCI, 'JACSPATH' AS JACSPN
                        COLUMNS (
                                   NESTED PATH '...' AS CPN2 COLUMNS (...),
                                   NESTED PATH '...' AS CPN3 COLUMNS (...)
                                 )
           )
```

In this example, *COLTREE* can be diagrammed as shown in Figure 7, "Diagram of COLTREE", using <JSON table path name>s to label the nodes:

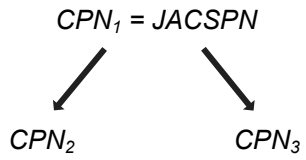

Figure 7 — Diagram of COLTREE

The implicit <JSON table specific plan> is

PLAN (JACSPN OUTER (CPN2 UNION CPN3))

j) Let *JTPLAN* be the explicit <JSON table specific plan> simply contained in *JTAB* or the implicit <JSON table specific plan> specified by the preceding Syntax Rules.

i) Every implicit or explicit <JSON table path name> contained in *JTAB* shall appear in *JTPLAN* exactly once.

ii) For every <JSON table plan parent/child> *JTPPC* contained in *JTPLAN*, let *LEFTOP* be the first <JSON table path name> contained in *JTPPC*, and let *OTHEROP* be any other

<JSON table path name> contained in *JTPPC*. Let *LEFTCOLS* be the <JSON table columns clause> that is associated with *LEFTOP*, and let *OTHERCOLS* be the <JSON table columns clause> associated with *OTHEROP*. *OTHERCOLS* shall be contained in *LEFTCOLS*.

k) The plan tree *JTTREE* of *JTAB* is a tree whose leaves are the <JSON table path name>s contained in *JTPLAN* and whose interior nodes are the BNF non-terminals contained in *JTPLAN* that contain a <JSON table path name>. The nodes of *JTTREE* are arranged according to syntactic containment, that is, for every interior node *N*, the children of *N* are the BNF non-terminals immediately contained in *N*, excluding <left paren> and <right paren>.

NOTE 252 — Continuing the prior example, given the plan

```
PLAN ( JACSPN OUTER ( CPN2 UNION CPN3 ) )
```

the plan tree can be diagrammed (using OUTER to label a <JSON table plan outer> node and UNION to label a <JSON table plan union> node) as shown in Figure 8, "Diagram of a plan tree".

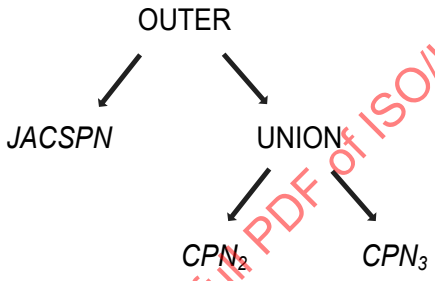

Figure 8 — Diagram of a plan tree

i) Let *NN* be the number of nodes of *JTTREE* and let the nodes of of *JTTREE* be $JTNODE_1$, ..., $JTNODE_{NN}$, enumerated in an implementation-dependent (US033) order that for all *i1*, 1 (one) ≤ *i1* ≤ *NN*, and for all *i2*, 1 (one) ≤ *i2* ≤ *NN*, if node $JTNODE_{i1}$ is contained in node $JTNODE_{i2}$, then *i1* < *i2*.

NOTE 253 — This can be done using a depth-first traversal of *JTTREE*.

ii) For all *i*, 1 (one) ≤ *i* ≤ *NN*, let $JTCORR_i$ be a <correlation name> defined as follows:

1) If $JTNODE_i$ is a <JSON table path name> *JTPN*, then let $JTCORR_i$ be *JTPN*.

2) If $JTNODE_i$ has only one child $JTNODE_c$, then let $JTCORR_i$ be equivalent to $JTCORR_c$.

NOTE 254 — This handles the cases where $JTNODE_i$ is a <JSON table plan>, <JSON table plan parent/child>, <JSON table plan sibling>, or <JSON table plan primary>.

3) Otherwise, let $JTCORR_i$ be an implementation-dependent (UV084) <correlation name> that is distinct from any <JSON table path name> and from any other <correlation name> chosen by this rule.

iii) For all *i*, 1 (one) ≤ *i* ≤ *NN*, a <select list> $JTSL_i$ and a <table primary> $JTPRIM_i$ are defined as follows.

Case:

1) If $JTNODE_i$ has exactly one child $JTNODE_c$, then let $JTSL_i$ be the same as $JTSL_c$ and let $JTPRIM_i$ be the same as $JTPRIM_c$.

2) If $JTNODE_i$ is a <JSON table plan outer>, then let $JTNODE_a$, be the <JSON table path name> that is the first operand of $JTNODE_i$ and let $JTNODE_b$, be the <JSON table plan primary> that is the second operand of $JTNODE_i$. Let $JTSL_i$ be

$JTSL_a$, $JTSL_b$

and let $JTPRIM_i$ be

```
LATERAL ( SELECT JTSL_a, JTSL_b
            FROM JTPRIM_a LEFT OUTER JOIN JTPRIM_b
              ON 0=0
          ) AS JTCORR_i
```

3) If $JTNODE_i$ is a <JSON table plan inner>, then let $JTNODE_a$, a, 1 (one) $\leq a \leq NN$, be the <JSON table path name> that is the first operand of $JTNODE_i$ and let $JTNODE_b$, b, 1 (one) $\leq b \leq NN$, be the <JSON table plan primary> that is the second operand of $JTNODE_i$. Let $JTSL_i$ be

$JTSL_a$, $JTSL_b$

and let $JTPRIM_i$ be

```
LATERAL ( SELECT JTSL_a, JTSL_b
            FROM JTPRIM_a, JTPRIM_b
          ) AS JTCORR_i
```

4) If $JTNODE_i$ is a <JSON table plan union>, then let NTP be the number of <JSON table plan primary>s that are the children of $JTNODE_i$, let $PNODE_{i,j}$, 1 (one) $\leq j \leq NPT$, be the <JSON table plan primary>s that are the children of $JTNODE_i$, and $PSUB(j)$ be the subscript that $PNODE_{i,j}$ holds as a $JTNODE$, let $JTSL_i$ be

$JTSL_{PSUB(1)}$, ..., $JTSL_{PSUB(NTP)}$

and let $JTPRIM_i$ be

```
LATERAL ( SELECT JTSL_PSUB(1), ..., JTSL_PSUB(NTP)
            FROM JTPRIM_PSUB(1) FULL OUTER JOIN
                 ...
                 JTSL_PSUB(NTP) ON 0=1
          ) AS JTCORR_i
```

NOTE 255 — For example, if NTP is 3:

```
LATERAL ( SELECT JTSL_PSUB(1), JTSL_PSUB(2), JTSL_PSUB(3)
            FROM JTPRIM_PSUB(1)
              FULL OUTER JOIN
                 JTPRIM_PSUB(2) ON 0=1
              FULL OUTER JOIN
                 JTPRIM_PSUB(3) ON 0=1
          ) AS JTCORR_i
```

5) If $JTNODE_i$ is a <JSON table plan cross>, then let NTP be the number of <JSON table plan primary>s that are the children of $JTNODE_i$, let $PNODE_{i,j}$, 1 (one) $\leq j \leq NPT$, be the <JSON table plan primary>s that are the children of $JTNODE_i$, and $PSUB(j)$ be the subscript that $PNODE_{i,j}$ holds as a $JTNODE$, let $JTSL_i$ be

$JTSL_{PSUB(1)}$, ..., $JTSL_{PSUB(NTP)}$

and let $JTPRIM_i$ be

```
LATERAL ( SELECT JTSL_PSUB(1), ..., JTSL_PSUB(NTP)
            FROM JTPRIM_PSUB(1), ..., JTSL_PSUB(NTP)
        ) AS JTCORR_i
```

6) If $JTNODE_i$ is a <JSON table path name> $JTPN_i$, then:

A) Let $JTPATH_i$ be the <JSON path specification> associated with $JTPN_i$ and let $JTCOLS_i$ be the <JSON table columns clause> associated with $JTPN_i$.

B) Case:

I) If $JTPN_i$ is *JACSPN*, then let $JTCI_i$ be *JACSCI*.

II) Otherwise, let *ANCESTOR* be the <JSON table plan inner> or <JSON table plan outer> that simply contains $JTPN_i$ in the second (<JSON table plan primary>) argument. Let *ANCESTORPN* be the <JSON table path name> that is the first operand of *ANCESTOR*. Let $JTCI_i$ be

```
ANCESTORPN.JTPN_i
```

NOTE 256 — <JSON table path name>s are used both as correlation names and column names. Here *ANCESTORPN* is the correlation name of the query and $JTPN_i$ is the column name of the column that generates the context item for $JTPRIM_i$.

C) Let *ND* be the number of <JSON table column definition>s simply contained in $JTCOLS_i$. Let $JTCD_d$, 1 (one) ≤ d ≤ *ND*, be those <JSON table column definition>s.

For all d, 1 (one) ≤ d ≤ *ND*,

Case:

I) If $JTCD_d$ is a <JSON table nested columns> *JTNC*, then let *JTCC* be the <JSON table columns clause> simply contained in *JTNC*, let $COLN_d$ be the <JSON table path name> associated with *JTCC*, and let $JTCDNEW_d$ be

```
COLN_d FOR CHAINING
```

II) If $JTCD_d$ is a <JSON table ordinality column definition>, then let $JTCDNEW_d$ be $JTCD_d$.

III) If $JTCD_d$ is a <JSON table regular column definition>, then let $COLN_d$ be the <column name> contained in $JTCD_d$ and let $JTCDNEW_d$ be $JTCD_d$, with any implicit <JSON table column empty behavior> or <JSON table column error behavior> made explicit.

IV) If $JTCD_d$ is a <JSON table formatted column definition>, then let $COLN_d$ be the <column name> contained in $JTCD_d$ and let $JTCDNEW_d$ be $JTCD_d$, with any implicit <JSON table formatted column wrapper behavior>, <JSON table formatted column quotes behavior>, <JSON table formatted column empty behavior>, or <JSON table formatted column error behavior> made explicit.

D) Let $JTSL_i$

```
COLN1, ..., COLNND
```

and let $JTPRIM_i$ be

```
LATERAL ( SELECT COLN1, ..., COLNND
            FROM JSON_TABLE_PRIMITIVE (
                   JTCIi,
                   JTPATHi AS JACSPN
                   JACSPC
                   COLUMNS ( JTCDNEW1, ..., JTCDNEWND )
                   JTEB ON ERROR ) ) AS JTPNi
```

l) Let *NC* be the number of <column name>s contained in *JTAB* and let $COLN_e$, 1 (one) ≤ e ≤ *NC*, be those <column name>s in the order in which they lexicographically appear in *JTAB*.

NOTE 257 — This includes all <column name>s in any <JSON table nested columns> at any depth of nesting, but excludes the columns whose names are <JSON table path name>s created for chaining between JSON_TABLE_PRIMITIVE invocations.

m) *JTAB* is equivalent to

```
LATERAL ( SELECT COLN1, ..., COLNNC
            FROM JTPRIMNN )
```

NOTE 258 — The <correlation or recognition> is suffixed to the preceding <lateral derived table>.

2) The degree of the table specified by <JSON table primitive> *JTP* is the number of <JSON table primitive column definition>s simply contained in *JTP*. For each <JSON table primitive column definition> *JTPCD* contained in *JTP*, a column descriptor is determined as follows:

a) The column name of the column is the <column name> simply contained in *JTPCD*.

b) Case:

i) If *JTPCD* is a <JSON table ordinality column definition>, then the declared type of the column is an implementation-defined (IV158) exact numeric type with scale 0 (zero).

ii) If *JTPCD* is a <JSON table regular column definition> or a <JSON table formatted column definition>, then the declared type of the column is the type defined by the <data type> simply contained in *JTPCD*.

iii) If *JTPCD* is a <JSON table primitive chaining column>, then the declared type of the column is an implementation-dependent (UV123) string type.

Access Rules

None.

General Rules

1) The result of *JTP* is determined as follows:

a) If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of *JTP* is an empty table and no further General Rules of this Subclause are applied.

b) Let *JACS* be the <JSON API common syntax> simply contained in *JTP*.

c) Let *JTEB* be the <JSON table error behavior> simply contained in *JTP*.

d) The General Rules of Subclause 9.47, "Processing <JSON API common syntax>", are applied with *JACS* as *JSON API COMMON SYNTAX*; let *ROWST* be the *STATUS* and let *ROWSEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

e) Case:

i) If *ROWST* is an exception condition, then

Case:

1) If *JTEB* is ERROR, then the exception condition *ROWST* is raised.

2) Otherwise, the result of *JTP* is an empty table.

ii) Otherwise, let *NI* be the number of SQL/JSON items in *ROWSEQ*, let I_j, 1 (one) ≤ j ≤ *NI*, be those SQL/JSON items in order, let *NCD* be the number of <JSON table primitive column definition>s contained in *JTP*, and let $JTCD_i$, 1 (one) ≤ i ≤ *NCD*, be those <JSON table primitive column definition>s.

For all j, 1 (one) ≤ j ≤ *NI*, and for all i, 1 (one) ≤ i ≤ *NCD*, the value of the i-th column of the j-th row in the result of *JTP* is determined as follows.

Case:

1) If $JTCD_i$ is a <JSON table ordinality column definition>, then the value of the i-th column of the j-th row is j.

2) If $JTCD_i$ is a <JSON table regular column definition>, then:

A) Let $JTCDPATH_i$ be the <JSON table column path specification>, let ZB_i be the <JSON table column empty behavior>, let EB_i be the <JSON table column error behavior> contained in $JTCD_i$, and let DT_i be the <data type> simply contained in $JTCD_i$.

B) The General Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with $JTCDPATH_i$ as *PATH SPECIFICATION*, I_j as *CONTEXT ITEM*, *True* as *ALREADY PARSED*, and the zero-length character string as *PASSING CLAUSE*; let *ST1* be the *STATUS* and let *SEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

C) The General Rules of Subclause 9.48, "Casting an SQL/JSON sequence to an SQL type", are applied with *ST1* as *STATUS IN*, *SEQ* as *SQL/JSON SEQUENCE*, ZB_i as *EMPTY BEHAVIOR*, EB_i as *ERROR BEHAVIOR*, and DT_i as *DATA TYPE*; let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* returned from the application of those General Rules.

D) If *ST2* is an exception condition, then the exception condition *ST2* is raised; otherwise, the value of the i-th column of the j-th row is *V*.

3) If $JTCD_i$ is a <JSON table formatted column definition>, then:

A) Let $JTCDPATH_i$ be the <JSON table column path specification>, let WB_i be the <JSON table formatted column wrapper behavior>, let QB_i be the <JSON table formatted column quotes behavior>, let ZB_i be the <JSON table formatted column empty behavior>, let EB_i be the <JSON table formatted column error behavior> contained in $JTCD_i$, let DT_i be the <data type>, and let FO_i be the <JSON representation> simply contained in $JTCD_i$.

B) The General Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with $JTCDPATH_i$ as *PATH SPECIFICATION*, I_j as *CONTEXT ITEM*, True as *ALREADY PARSED*, and the zero-length character string as *PASSING CLAUSE*; let *ST1* be the *STATUS* and let *SEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

C) Case:

I) If DT_i is a JSON type, then General Rules of Subclause 9.44, "Converting an SQL/JSON sequence to an SQL/JSON item", are applied with *ST1* as *STATUS IN*, *SEQ* as *SQL/JSON SEQUENCE*, WB_i as *WRAPPER BEHAVIOR*, ZB_i as *EMPTY BEHAVIOR*, and EB_i as *ERROR BEHAVIOR*; let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* returned from the application of those General Rules.

II) Otherwise, the General Rules of Subclause 9.49, "Serializing an SQL/JSON sequence to an SQL string type", are applied with *ST1* as *STATUS IN*, *SEQ* as *SQL/JSON SEQUENCE*, WB_i as *WRAPPER BEHAVIOR*, QB_i as *QUOTES BEHAVIOR*, ZB_i as *EMPTY BEHAVIOR*, EB_i as *ERROR BEHAVIOR*, DT_i as *DATA TYPE*, and FO_i as *FORMAT OPTION*; let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* returned from the application of those General Rules.

D) If *ST2* is an exception condition, then the exception condition *ST2* is raised. Otherwise, the value of the *i*-th column of the *j*-th row is *V*.

4) If $JTCD_i$ is a <JSON table primitive chaining column>, then:

A) Case:

I) If DT_i is a JSON type, then let *JT* be I_j.

II) Otherwise, the General Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with I_j as *SQL/JSON ITEM*, an implementation-dependent (UV085) <JSON representation> as *FORMAT OPTION*, and an implementation-dependent (UV085) data type as *TARGET TYPE*; let *CHAINST* be the *STATUS* and let *JT* be the *JSON TEXT* returned from the application of those General Rules.

If *CHAINST* is an exception condition, then

Case:

1) If *JTEB* is ERROR, then the exception condition *CHAINST* is raised.

2) Otherwise, let *JT* be the null value.

NOTE 259 — Because of the syntactic transformation in the Syntax Rules, this value will be the context item of a different <JSON table primitive> invocation. The null value cannot be parsed, so that query will result in an empty table (not an error, since *JTEB* is EMPTY).

B) The value of the *i*-th column of the *j*-th row of the result of *JTP* is *JT*.

Conformance Rules

1) Conforming SQL language shall not contain <JSON table primitive>.

NOTE 260 — <JSON table primitive> is a specification device used to define the semantics of nested tables in <JSON table> and is not syntax available to the user.

2) Without Feature T821, "Basic SQL/JSON query operators", conforming SQL language shall not contain <JSON table>.

3) Without Feature T824, "JSON_TABLE: specific PLAN clause", <JSON table> shall not contain <JSON table specific plan>.

4) Without Feature T838, "JSON_TABLE: PLAN DEFAULT clause", <JSON table> shall not contain <JSON table default plan>.

5) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON table> shall not contain <JSON table error behavior>.

6) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON table> shall not contain <JSON table column empty behavior>.

7) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON table> shall not contain <JSON table column error behavior>.

8) Without Feature T826, "General value expression in ON ERROR or ON EMPTY clauses", the <value expression> contained in <JSON table column empty behavior> or <JSON table column error behavior> contained in a <JSON table regular column definition> *JTRCD* shall be a <literal> that can be cast to the data type specified by the <data type> contained in *JTRCD* without raising an exception condition according to the General Rules of Subclause 6.13, "<cast specification>".

9) Without Feature T827, "JSON_TABLE: sibling NESTED COLUMNS clauses", an explicit or implicit <JSON table plan> shall not contain <JSON table plan sibling>.

10) Without Feature T851, "SQL/JSON: optional keywords for default syntax", conforming SQL language shall not contain a <JSON table formatted column definition> that simply contains both a <data type> that identifies a JSON type and FORMAT JSON.

7.12 <where clause>

Function

Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

Format

```
<where clause> ::=
  WHERE <search condition>
```

Syntax Rules

1) If a <value expression> directly contained in the <search condition> is a <set function specification>, then the <where clause> shall be contained in a <having clause> or <select list>, the <set function specification> shall contain a column reference, and every column reference contained in an aggregated argument of the <set function specification> shall be an outer reference.

   NOTE 261 — *outer reference* is defined in Subclause 6.7, "<column reference>".

2) The <search condition> shall not contain a <window function> without an intervening <query expression>.

Access Rules

None.

General Rules

1) Let *T* be the result of the preceding <from clause>.

2) The <search condition> is effectively evaluated for each row of *T*. The result of the <where clause> is a table of those rows of *T* for which the result of the <search condition> is *True*.

Conformance Rules

1) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <value expression> directly contained in a <where clause> that contains a <column reference> that references a <derived column> that generally contains a <set function specification>.

7.13 <group by clause>

Function

Specify a grouped table derived by the application of the <group by clause> to the result of the previously specified clause.

Format

```
<group by clause> ::=
  GROUP BY [ <set quantifier> ] <grouping element list>

<grouping element list> ::=
  <grouping element> [ { <comma> <grouping element> }... ]

<grouping element> ::=
    <ordinary grouping set>
  | <rollup list>
  | <cube list>
  | <grouping sets specification>
  | <empty grouping set>

<ordinary grouping set> ::=
    <grouping column reference>
  | <left paren> <grouping column reference list> <right paren>

<grouping column reference> ::=
  <column reference> [ <collate clause> ]

<grouping column reference list> ::=
  <grouping column reference> [ { <comma> <grouping column reference> }... ]

<rollup list> ::=
  ROLLUP <left paren> <ordinary grouping set list> <right paren>

<ordinary grouping set list> ::=
  <ordinary grouping set> [ { <comma> <ordinary grouping set> }... ]

<cube list> ::=
  CUBE <left paren> <ordinary grouping set list> <right paren>

<grouping sets specification> ::=
  GROUPING SETS <left paren> <grouping set list> <right paren>

<grouping set list> ::=
  <grouping set> [ { <comma> <grouping set> }... ]

<grouping set> ::=
    <ordinary grouping set>
  | <rollup list>
  | <cube list>
  | <grouping sets specification>
  | <empty grouping set>

<empty grouping set> ::=
  <left paren> <right paren>
```

Syntax Rules

1) Each <grouping column reference> shall unambiguously reference a column of the table resulting from the <from clause>. A column referenced in a <group by clause> is a *grouping column*.

NOTE 262 — "Column reference" is defined in Subclause 6.7, "<column reference>".

2) Each <grouping column reference> is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

3) For every <grouping column reference> *GC*,

Case:

a) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of the column reference shall be character string. The declared type of *GC* is that of its column reference, except that *CS* is the declared type collation and the collation derivation is *explicit*.

b) Otherwise, the declared type of *GC* is the declared type of its column reference.

4) Let *QS* be the <query specification> that simply contains the <group by clause>, and let *SL*, *FC*, *WC*, *GBC*, and *HC* be the <select list>, the <from clause>, the <where clause> if any, the <group by clause>, and the <having clause> if any, respectively, that are simply contained in *QS*.

5) Let *QSSQ* be the explicit or implicit <set quantifier> immediately contained in *QS*.

6) Let *GBSQ* be the <set quantifier> immediately contained in <group by clause>, if any; otherwise, let *GBSQ* be ALL.

7) Let *SL1* be obtained from *SL* by replacing every <asterisk> and <asterisked identifier chain> using the syntactic transformations in the Syntax Rules of Subclause 7.16, "<query specification>".

8) A <group by clause> is *primitive* if it does not contain a <rollup list>, <cube list>, <grouping sets specification>, or <grouping column reference list>, and does not contain both a <grouping column reference> and an <empty grouping set>.

9) A <group by clause> is *simple* if it does not contain a <rollup list>, <cube list> or <grouping sets specification>.

10) If *GBC* is a simple <group by clause> that is not primitive, then *GBC* is transformed into a primitive <group by clause> as follows:

a) Let *NSGB* be the number of <grouping column reference>s contained in *GBC*.

b) Case:

i) If *NSGB* is 0 (zero), then *GBC* is replaced by

```
GROUP BY ()
```

ii) Otherwise:

1) Let $SGCR_1$, ... $SGCR_{NSGB}$ be an enumeration of the <grouping column reference>s contained in *GBC*.

2) *GBC* is replaced by

```
GROUP BY SGCR1, ...
SGCRNSGB
```

NOTE 263 — That is, a simple <group by clause> that is not primitive can be transformed into a primitive <group by clause> by deleting all parentheses, and deleting extra <comma>s as necessary

for correct syntax. If there are no grouping columns at all (for example, `GROUP BY (), ()`), this is transformed to the canonical form `GROUP BY ()`.

11) If *GBC* is a primitive <group by clause>, then let *SLNEW* and *HCNEW* be obtained from *SL1* and *HC*, respectively, by replacing every <grouping operation> by the <exact numeric literal> 0 (zero). *QS* is equivalent to:

```
SELECT QSSQ SLNEW FC
WC GBC HCNEW
```

12) Let *OGSL* be an <ordinary grouping set list>. The *concatenation* of *OGSL* is defined as follows:

a) Let *NGCR* be the number of <grouping column reference>s simply contained in *OGSL* and let GCR_j, 1 (one) ≤ j ≤ *NGCR*, be an enumeration of those <grouping column reference>s, in order from left to right.

b) The concatenation of *OGSL* is the <ordinary grouping set list>

GCR_1, ..., GCR_{NGCR}

NOTE 264 — Thus, the concatenation of *OGSL* is formed by erasing all parentheses. For example, the concatenation of "(A, B), (C, D)" is "A, B, C, D".

13) Let *RL* be a <rollup list>. Let OGS_i range over the *n* <ordinary grouping set>s contained in *RL*.

a) For each *i* between 1 (one) and *n*, let $COGS_i$ be the concatenation of the <ordinary grouping set list>

OGS_1, OGS_2, ..., OGS_i

b) *RL* is equivalent to:

```
GROUPING SETS (
   ( COGS_n ),
   ( COGS_n-1 ),
   ( COGS_n-2 ),
   ...
   ( COGS_1 ),
   ( ) )
```

NOTE 265 — The result of the transform is to replace *RL* with a <grouping sets specification> that contains a <grouping set> for every initial sublist of the <ordinary grouping set list> of the <rollup list>, obtained by dropping <ordinary grouping set>s from the right, one by one, and concatenating each <ordinary grouping set list> so obtained. The <empty grouping set> is regarded as the shortest such initial sublist. For example, "ROLLUP ((A, B), (C, D))" is equivalent to "GROUPING SETS ((A, B, C, D), (A, B), ())".

14) Let *CL* be a <cube list>. Let OGS_i range over the *n* <ordinary grouping set>s contained in *CL*. *CL* is transformed as follows:

a) Let $M = 2^n - 1$ (one).

b) For each *i* between 1 (one) and *M*:

i) Let BSL_i be the binary number consisting of *n* bits (binary digits) whose value is *i*.

ii) For each *j* between 1 (one) and *n*, let $B_{i,j}$ be the *j*-th bit, counting from left to right, in BSL_i.

iii) For each *j* between 1 (one) and *n*, let $GSLCR_{i,j}$ be

Case:

1) If $B_{i,j}$ is 0 (zero), then the zero-length character string.

2) If $B_{i,j}$ is 1 (one) and $B_{i,k}$ is 0 (zero) for all $k < j$, then OGS_j.

3) Otherwise, <comma> followed by OGS_j.

iv) Let GSL_i be the concatenation of the <ordinary grouping set list>

$GSLCR_{i,1}$ $GSLCR_{i,2}$ ... $GSLCR_{i,n}$

c) *CL* is equivalent to

GROUPING SETS ((GSL_M), (GSL_{M-1}), ..., (GSL_1), ( ))

NOTE 266 — The result of the transform is to replace *CL* with a <grouping sets specification> that contains a <grouping set> for all possible subsets of the set of <ordinary grouping set>s in the <ordinary grouping set list> of the <cube list>, including <empty grouping set> as the empty subset with no <ordinary grouping set>s.

For example, `CUBE (A, B, C)` is equivalent to:

```
GROUPING SETS ( /* BSL_i */
  (A, B, C),    /* 111 */
  (A, B   ),    /* 110 */
  (A,    C),    /* 101 */
  (A      ),    /* 100 */
  (   B, C),    /* 011 */
  (   B   ),    /* 010 */
  (      C),    /* 001 */
  (       )
)
```

As another example, `CUBE ((A, B), (C, D))` is equivalent to:

```
GROUPING SETS ( /* BSL_i */
  (A, B, C, D), /* 11 */
  (A, B      ), /* 10 */
  (      C, D), /* 01 */
  (           )
)
```

15) Let *GSSA* be a <grouping sets specification>. If *GSSA* simply contains another <grouping sets specification> *GSSB*, then *GSSA* is transformed as follows:

a) Let *NA* be the number of <grouping set>s simply contained in *GSSA*, and let *NB* be the number of <grouping set>s simply contained in *GSSB*.

b) Let GSA_i be an enumeration of the <grouping set>s simply contained in *GSSA*, for 1 (one) $\leq i \leq NA$.

c) Let GSB_i be an enumeration of the <grouping set>s simply contained in *GSSB*, 1 (one) $\leq i \leq NB$.

d) Let k be the value such that $GSSB = GSA_k$.

e) *GSSA* is equivalent to

GROUPING SETS (
GSA_1, GSA_2, ... GSA_{k-1},
GSB_1, ..., GSB_{NB},
GSA_{k+1}, ..., GSA_{NA})

NOTE 267 — Thus, the nested <grouping sets specification> is removed by simply "promoting" each of its <grouping set>s to be a <grouping set> of the encompassing <grouping sets specification>.

16) Let *CGB* be a <group by clause> that is not simple. *CGB* is transformed as follows:

a) The preceding Syntax Rules are applied repeatedly to eliminate any <grouping sets specification> that is nested in another <grouping sets specification>, as well as any <rollup list> and any <cube list>.

NOTE 268 — As a result, *CGB* is either a single <grouping sets specification> or a list of two or more <grouping set>s, each of which is an <ordinary grouping set>, an <empty grouping set>, or a <grouping sets specification> that contains only <ordinary grouping set>s and <empty grouping set>s. There are no remaining <rollup list>s, <cube list>s, or nested <grouping sets specification>s.

b) Any <grouping element> *GS* that is an <ordinary grouping set> or an <empty grouping set> is replaced by the <grouping sets specification>

```
GROUPING SETS ( GS )
```

NOTE 269 — As a result, *CGB* is a list of one or more <grouping sets specification>s.

c) If *CGB* contains two or more <grouping sets specification>s, then let *GSSX* and *GSSY* be the first two <grouping sets specification>s in *CGB*. *CGB* is transformed by replacing "*GSSX* <comma> *GSSY*" as follows:

i) Let *NX* be the number of <grouping set>s in *GSSX* and let *NY* be the number of <grouping set>s in *GSSY*.

ii) Let GSX_i, 1 (one) $\le i \le NX$, be the <grouping set>s contained in *GSSX*, and let GSY_i, 1 (one) $\le i \le NY$, be the <grouping set>s contained in *GSSY*.

iii) Let $MX(i)$ be the number of <grouping column reference>s in GSX_i, and let $MY(i)$ be the number of <grouping column reference>s in GSY_i.

NOTE 270 — If GSX_i is <empty grouping set>, then $MX(i)$ is 0 (zero); and similarly for GSY_i.

iv) Let $GCRX_{i,j}$, 1 (one) $\le j \le MX(i)$ be the <grouping column reference>s contained in GSX_i, and let $GCRY_{i,j}$, 1 (one) $\le j \le MY(i)$ be the <grouping column reference>s contained in GSY_i.

NOTE 271 — If GSX_i is <empty grouping set>, then there are no $GCRX_{i,j}$; and similarly for GSY_i.

v) For each a between 1 (one) and *NX* and each b between 1 (one) and *NY*, let $GST_{a,b}$ be

$$(GCRX_{a,1}, \ldots, GCRX_{a,MX(a)}, GCRY_{b,1}, \ldots, GCRY_{b,MY(b)})$$

that is, an <ordinary grouping set> consisting of $GCRA_{a,j}$ for all j between 1 (one) and $MX(a)$, followed by $GCRY_{b,j}$ for all j between 1 (one) and $MY(b)$.

vi) *CGB* is transformed by replacing "*GSSX* <comma> *GSSY*" with

$$\begin{array}{l}\texttt{GROUPING SETS (}\\ \quad GST_{1,1}, \ldots, GST_{1,NY},\\ \quad GST_{2,1}, \ldots, GST_{2,NY},\\ \quad \ldots\\ \quad GST_{NX,1}, \ldots, GST_{NX,NY}\\ \texttt{)}\end{array}$$

NOTE 272 — Thus each <ordinary grouping set> in *GSSA* is "concatenated" with each <ordinary grouping set> in *GSSB*. For example,

```
GROUP BY GROUPING SETS ((A, B), (C)),
         GROUPING SETS ((X, Y), ())
```

is transformed to

```
GROUP BY GROUPING SETS ((A, B, X, Y), (A, B),
         (C, X, Y), (C))
```

d) The previous subrule of this Syntax Rule is applied repeatedly until *CGB* consists of a single <grouping sets specification>.

17) If <grouping element list> consists of a single <grouping sets specification> *GSS* that contains only <ordinary grouping set>s or <empty grouping set>s, then:

a) Let *m* be the number of <grouping set>s contained in *GSS*.

b) Let GS_i, $1 \le i \le m$, range over the <grouping set>s contained in *GSS*.

c) Let *p* be the number of distinct <column reference>s that are contained in *GSS*.

d) Let *PC* be an ordered list of these <column reference>s ordered according to their left-to-right occurrence in the list.

e) Let PC_k, $1 \le k \le p$, be the *k*-th <column reference> in *PC*.

f) Let $DTPC_k$ be the declared type of the column identified by PC_k.

g) Let *NDC* be the number of <derived column>s simply contained in *SL1*.

h) Let DC_q, $1 \le q \le NDC$, be an enumeration of the <derived column>s simply contained in *SL1*, in order from left to right.

i) Let DCN_q be the column name of DC_q, 1 (one) $\le q \le NDC$.

j) Let VE_q, 1 (one) $\le q \le NDC$, be the <value expression> simply contained in DC_q.

k) Let XN_k, 1 (one) $\le k \le p$, YN_k, 1 (one) $\le k \le p$, and ZN_q, 1 (one) $\le q \le NDC$, be implementation-dependent (UV132) column names that are all distinct from one another.

l) Let *SL2* be the <select list>:

```
PC1 AS XN1, GROUPING (PC1) AS YN1,
...,
PCp AS XNp, GROUPING (PCp) AS YNp,
VE1 AS ZN1, ..., VENDC AS ZNNDC
```

m) For each GS_i:

i) If GS_i is an <empty grouping set>, then let $n(i)$ be 0 (zero). If GS_i is a <grouping column reference>, then let $n(i)$ be 1 (one). Otherwise, let $n(i)$ be the number of <grouping column reference>s contained in the <grouping column reference list>.

ii) Let $GCR_{i,j}$, $1 \le j \le n(i)$, range over the <grouping column reference>s contained in GS_i.

iii) Case:

1) If GS_i is an <ordinary grouping set>, then:

A) Transform *SL2* to obtain *SL3*, and transform *HC* to obtain *HC3*, as follows.

For every PC_k, if there is no *j* such that $PC_k = GCR_{i,j}$, then make the following replacements in *SL2* and *HC*:

I) Replace each <grouping operation> in *SL2* and *HC* that contains a <column reference> that references PC_k by

```
CAST ( 1 AS IDT )
```

where *IDT* is the implementation-defined (IV183) exact numeric type with scale 0 (zero) that is the declared type of the <grouping operation>.

II) Replace each <column reference> in *SL2* and *HC* that references PC_k by

```
CAST ( NULL AS DTPC_k )
```

B) Transform *SL3* to obtain *SLNEW*, and transform *HC3* to obtain *HCNEW* by replacing each <grouping operation> that remains in *SL3* and *HC3* by

```
CAST ( 0 AS IDT )
```

where *IDT* is the implementation-defined (IV183) exact numeric type with scale 0 (zero) that is the declared type of the <grouping operation>.

NOTE 273 — Thus the value of a <grouping operation> is 0 (zero) if the grouping column referenced by the <grouping operation> is among the $GCR_{i,j}$ and 1(one) if it is not.

C) Let $GSSQL_i$ be:

```
SELECT QSSQ SLNEW
FC
WC
GROUP BY GCR_i,1, ..., GCR_i,n(i)
HCNEW
```

2) If GS_i is an <empty grouping set>, then:

A) Transform *SL2* to obtain *SLNEW*, and transform *HC* to obtain *HCNEW*, as follows.

For every k, $1 \le k \le p$:

I) Replace each <grouping operation> in *SL2* and *HC* that contains a <column reference> that references PC_k by

```
CAST ( 1 AS IDT )
```

where *IDT* is the implementation-defined (IV183) exact numeric type with scale 0 (zero) that is the declared type of the <grouping operation>.

II) Replace each <column reference> in *SL2* and *HC* that references PC_k by

```
CAST ( NULL AS DTPC_k )
```

B) Let $GSSQL_i$ be

```
SELECT QSSQ SLNEW
FC
WC
GROUP BY ()
HCNEW
```

n) Let *GU* be:

```
GSSQL_1
  UNION GBSQ
```

```
GSSQL2
  UNION GBSQ
...
  UNION GBSQ
GSSQLm
```

o) Let *COR* be an implementation-dependent (UV132) <correlation name> that is not equivalent to any other <identifier> contained in *QS*. *QS* is equivalent to

```
SELECT QSSQ ZN1 AS DC1, ..., ZNNDC AS DCNDC
FROM ( GU ) AS COR
```

Access Rules

None.

General Rules

NOTE 274 — As a result of the syntactic transformations specified in the Syntax Rules of this Subclause, only primitive <group by clause>s are left to consider.

1) If no <where clause> is specified, then let *T* be the result of the preceding <from clause>; otherwise, let *T* be the result of the preceding <where clause>.

2) Case:

   a) If there are no grouping columns, then the result of the <group by clause> is the grouped table consisting of *T* as its only group.

   b) Otherwise, the result of the <group by clause> is a partitioning of the rows of *T* into the minimum number of groups such that, for each grouping column of each group, no two values of that grouping column are distinct. If the declared type of a grouping column is a user-defined type and the comparison of that column results in *Unknown* for two rows of *T*, then the assignment of those rows to groups in the result of the <group by clause> is implementation-dependent (UA053).

3) When a <search condition> or <value expression> is applied to a group, a reference *CR* to a column that is functionally dependent on the grouping columns is understood as follows.

   Case:

   a) If *CR* is a group-invariant column reference, then it is a reference to the common value in that column of the rows in that group. If the most specific type of the column is character, datetime with time zone, or a user-defined type, then the value is an implementation-dependent (UV086) value that is not distinct from the value of the column in each row of the group.

   b) Otherwise, *CR* is a within-group-varying column reference, and as such, it is a reference to the value of the column in each row of a given group determined by the grouping columns, to be used to construct the argument source of a <set function specification>.

Conformance Rules

1) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <rollup list>.

2) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <cube list>.

3) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain a <grouping sets specification>.

4) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain an <empty grouping set>.

5) Without Feature T431, "Extended grouping capabilities", conforming SQL language shall not contain an <ordinary grouping set> that contains a <grouping column reference list>.

6) Without Feature T432, "Nested and concatenated GROUPING SETS ", conforming SQL language shall not contain a <grouping set list> that contains a <grouping sets specification>.

7) Without Feature T432, "Nested and concatenated GROUPING SETS ", conforming SQL language shall not contain a <group by clause> that simply contains a <grouping sets specification> *GSS* where *GSS* is not the only <grouping element> simply contained in the <group by clause>.

8) Without Feature T434, "GROUP BY DISTINCT ", conforming SQL language shall not contain a <group by clause> that simply contains a <set quantifier>.

NOTE 275 — The Conformance Rules of Subclause 9.12, "Grouping operations", also apply.

7.14 <having clause>

Function

Specify a grouped table derived by the elimination of groups that do not satisfy a <search condition>.

Format

```
<having clause> ::=
  HAVING <search condition>
```

Syntax Rules

1) Let *HC* be the <having clause>. Let *TE* be the <table expression> that immediately contains *HC*. If *TE* does not immediately contain a <group by clause>, then "GROUP BY ()" is implicit. Let *T* be the descriptor of the table defined by the <group by clause> *GBC* immediately contained in *TE* and let *R* be the result of *GBC*.

2) Let *G* be the set consisting of every column referenced by a <column reference> contained in *GBC*.

3) Each column reference directly contained in the <search condition> shall be one of the following:

   a) An unambiguous reference to a column that is functionally dependent on *G*.

   b) An outer reference.

      NOTE 276 — The preceding Syntax Rule follows from a more comprehensive one in Subclause 6.7, "<column reference>".

4) Each column reference contained in a <query expression> in the <search condition> that references a column of *T* shall be one of the following:

   a) An unambiguous reference to a column that is functionally dependent on *G*.

   b) Contained in an aggregated argument of a <set function specification>.

      NOTE 277 — The preceding Syntax Rule follows from a more comprehensive one in Subclause 6.7, "<column reference>".

5) The <search condition> shall not contain a <window function> without an intervening <query expression>.

6) The row type of the result of the <having clause> is the row type *RT* of *T*.

Access Rules

None.

General Rules

1) The <search condition> is evaluated for each group of *R*. The result of the <having clause> is a grouped table of those groups of *R* for which the result of the <search condition> is *True*.

Conformance Rules

1) Without Feature T301, "Functional dependencies", in conforming SQL language, each column reference directly contained in the <search condition> shall be one of the following:

   a) An unambiguous reference to a grouping column of *T*.

   b) An outer reference.

2) Without Feature T301, "Functional dependencies", in conforming SQL language, each column reference contained in a <query expression> in the <search condition> that references a column of *T* shall be one of the following:

   a) An unambiguous reference to a grouping column of *T*.

   b) Contained in an aggregated argument of a <set function specification>.

7.15 <window clause>

Function

Specify one or more window definitions.

Format

```
<window clause> ::=
  WINDOW <window definition list>

<window definition list> ::=
  <window definition> [ { <comma> <window definition> }... ]

<window definition> ::=
  <new window name> AS <window specification>

<new window name> ::=
  <window name>

<window specification> ::=
  <left paren> <window specification details> <right paren>

<window specification details> ::=
  [ <existing window name> ]
      [ <window partition clause> ]
      [ <window order clause> ]
      [ <window frame clause> ]

<existing window name> ::=
  <window name>

<window partition clause> ::=
  PARTITION BY <window partition column reference list>

<window partition column reference list> ::=
  <window partition column reference>
      [ { <comma> <window partition column reference> }... ]

<window partition column reference> ::=
  <column reference> [ <collate clause> ]

<window order clause> ::=
  ORDER BY <sort specification list>

<window frame clause> ::=
  [ <row pattern measures> ]
      <window frame units> <window frame extent>
      [ <window frame exclusion> ]
      [ <row pattern common syntax> ]

<window frame units> ::=
    ROWS
  | RANGE
  | GROUPS

<window frame extent> ::=
    <window frame start>
  | <window frame between>

<window frame start> ::=
```

```
    UNBOUNDED PRECEDING
  | <window frame preceding>
  | CURRENT ROW

<window frame preceding> ::=
  <unsigned value specification> PRECEDING

<window frame between> ::=
  BETWEEN <window frame bound 1> AND <window frame bound 2>

<window frame bound 1> ::=
  <window frame bound>

<window frame bound 2> ::=
  <window frame bound>

<window frame bound> ::=
    <window frame start>
  | UNBOUNDED FOLLOWING
  | <window frame following>

<window frame following> ::=
  <unsigned value specification> FOLLOWING

<window frame exclusion> ::=
    EXCLUDE CURRENT ROW
  | EXCLUDE GROUP
  | EXCLUDE TIES
  | EXCLUDE NO OTHERS
```

Syntax Rules

1) Let *TE* be the <table expression> that immediately contains the <window clause>.

2) <new window name> *NWN1* shall not be contained in the scope of another <new window name> *NWN2* such that *NWN1* and *NWN2* are equivalent.

3) Let *WDEF* be a <window definition>.

4) Each <column reference> contained in the <window partition clause> or <window order clause> of *WDEF* shall unambiguously reference a column of the derived table *T* that is the result of *TE*. A column referenced in a <window partition clause> is a *window partitioning column*. Each window partitioning column is an operand of a grouping operation, and the Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

   NOTE 278 — If *T* is a grouped table, then the <column reference>s contained in <window partition clause> or <window order clause> must reference columns of the grouped table obtained by performing the syntactic transformation in Subclause 7.16, "<query specification>".

5) For every <window partition column reference> *PC*,

   Case:

   a) If <collate clause> is specified, then let *CS* be the collation identified by <collation name>. The declared type of the column reference shall be character string. The declared type of *PC* is that of its column reference, except that *CS* is the declared type collation and the collation derivation is *explicit*.

   b) Otherwise, the declared type of *PC* is the declared type of its column reference.

6) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. Each column reference contained in <window clause> that references a column of *T* shall reference a column that is functionally dependent on *G* or be contained in an aggregated argument of a <set function specification>.

NOTE 279 — The preceding Syntax Rule follows from a more comprehensive one in Subclause 6.7, "<column reference>".

7) A <window clause> shall not contain a <window function> without an intervening <query expression>.

8) If *WDEF* specifies <window frame preceding>, then the <unsigned value specification> immediately contained in <window frame preceding> shall not be the <non-reserved word> UNBOUNDED.

9) If *WDEF* specifies <window frame following>, then the <unsigned value specification> immediately contained in <window frame following> shall not be the <non-reserved word> UNBOUNDED.

NOTE 280 — The preceding two rules resolve an ambiguity in which UNBOUNDED could be interpreted either as a key word or as an <unsigned value specification>. This ambiguity is resolved by requiring the interpretation as a key word.

10) If *WDEF* specifies <window frame between>, then:

a) <window frame bound 1> shall not specify UNBOUNDED FOLLOWING.

b) <window frame bound 2> shall not specify UNBOUNDED PRECEDING.

c) If <window frame bound 1> specifies CURRENT ROW, then <window frame bound 2> shall not specify <window frame preceding>.

d) If <window frame bound 1> specifies <window frame following>, then <window frame bound 2> shall not specify <window frame preceding> or CURRENT ROW.

11) If *WDEF* specifies <window frame extent>, and does not specify <window frame between>, then let *WAGS* be the <window frame start>. The <window frame extent> is equivalent to

```
BETWEEN WAGS AND CURRENT ROW
```

12) If *WDEF* specifies an <existing window name> *EWN*, then:

a) *WDEF* shall be within the scope of a <window name> that is equivalent to <existing window name>.

b) Let *WDX* be the window structure descriptor identified by *EWN*.

c) *WDEF* shall not specify <window partition clause>.

d) If *WDX* has a window ordering clause, then *WDEF* shall not specify <window order clause>.

e) *WDX* shall not have a window framing clause.

13) If *WDEF* specifies <row pattern measures>, then <row pattern common syntax> shall be specified.

14) If *WDEF* specifies <row pattern common syntax> *RPCS*, then:

a) Let *WFC* be the <window frame clause> simply contained in *WDEF*.

b) *WFC* shall specify <window frame units> that is ROWS.

c) *WFC* shall specify <window frame between> that contains a <window frame start> that is CURRENT ROW.

d) If *WFC* specifies <window frame exclusion> *WFE*, then *WFE* shall be EXCLUDE NO OTHERS.

e) *RPCS* shall not contain <circumflex> or <dollar sign>.

15) If *WDEF*'s <window frame clause> specifies <window frame preceding> or <window frame following>, then let *UVS* be the <unsigned value specification> simply contained in the <window frame preceding> or <window frame following>.

Case:

a) If RANGE is specified, then:

i) Case:

1) If *WDEF* contains <window order clause> *WOC*, then let *WDEFOC* be *WOC*.

2) Otherwise, *WDEF* shall specify an <existing window name> that identifies a window structure descriptor that includes a window ordering clause *WOGC*. Let *WDEFOC* be *WOGC*.

ii) *WDEFOC* shall contain a single <sort key> *SK*.

iii) The declared type of *SK* shall be numeric, datetime, or interval. The declared type of *UVS* shall be numeric if the declared type of *SK* is numeric; otherwise, it shall be an interval type that may be added to or subtracted from the declared type of *SK* according to the Syntax Rules of Subclause 6.42, "<datetime value expression>", and Subclause 6.44, "<interval value expression>", in this document.

b) If ROWS is specified, then the declared type of *UVS* shall be exact numeric with scale 0 (zero).

c) If GROUPS is specified, then:

i) Either *WDEF* shall contain a <window order clause>, or *WDEF* shall specify an <existing window name> that identifies a window structure descriptor that includes a window ordering clause.

ii) The declared type of *UVS* shall be exact numeric with scale 0 (zero).

16) The scope of the <new window name> simply contained in *WDEF* consists of any <window definition>s that follow *WDEF* in the <window clause>, together with the <select list> of the <query specification> that simply contains the <window clause>. If the <window clause> is simply contained in a <query specification> that is the <query expression body> of a <query expression> that is a simple table query, then the scope of <new window name> also includes the <order by clause>, if any, of that <query expression>.

17) Two window structure descriptors *WD1* and *WD2* are *order-equivalent* if and only if all of the following are true:

a) Let $WPCR1_i$, 1 (one) $\le i \le N1$, and $WPCR2_i$, 1 (one) $\le i \le N2$, be enumerations of the <window partition column reference>s contained in the window partitioning clauses of *WD1* and *WD2*, respectively, in order from left to right. $N1 = N2$, and, for all i, $WPCR1_i$ and $WPCR2_i$ are equivalent column references.

b) Let $SS1_i$, 1 (one) $\le i \le M1$, and $SS2_i$, 1 (one) $\le i \le M2$, be enumerations of the <sort specification>s contained in the window ordering clauses of *WD1* and *WD2*, respectively, in order from left to right. $M1 = M2$, and, for all i, $SS1_i$ and $SS2_i$ contain <sort key>s that are equivalent column references, specify or imply the same <ordering specification>, specify or imply the same <collate clause>, if any, and specify or imply the same <null ordering>.

Access Rules

None.

General Rules

1) Let *SL* be the <select list> of the <query specification> or <select statement: single row> that immediately contains *TE*.

Case:

a) If *SL* does not simply contain a <window function>, then the <window clause> is disregarded, and the result of *TE* is the result of the last <from clause>, <where clause>, <group by clause> or <having clause> of *TE*.

b) Otherwise, let *RTE* be the result of the last <from clause> or <where clause> simply contained in *TE*.

NOTE 281 — Although it is permissible to have a <group by clause> or a <having clause> with a <window clause>, if there are any <window function>s, then the <group by clause> and <having clause> are removed by a syntactic transformation in Subclause 7.16, "<query specification>", and so are not considered here.

i) A window structure descriptor *WDESC* is created for each <window definition> *WDEF*, as follows:

1) *WDESC*'s window name is the <new window name> simply contained in *WDEF*.

2) If <existing window name> is specified, then let *EWN* be the <existing window name> simply contained in *WDEF* and let *WDX* be the window structure descriptor identified by *EWN*.

3) If <existing window name> is specified and the window ordering clause of *WDX* is present, then the ordering window name of *WDESC* is *EWN*; otherwise, there is no ordering window name.

4) Case:

A) If *WDEF* simply contains <window partition clause> *WDEFWPC*, then *WDESC*'s window partitioning clause is *WDEFWPC*.

B) If <existing window name> is specified, then *WDESC*'s window partitioning clause is the window partitioning clause of *WDX*.

C) Otherwise, *WDESC* has no window partitioning clause.

5) Case:

A) If *WDEF* simply contains <window order clause> *WDEFWOC*, then *WDESC*'s window ordering clause is *WDEFWOC*.

B) If <existing window name> is specified, then *WDESC*'s window ordering clause is the window ordering clause of *WDX*.

C) Otherwise, *WDESC* has no window ordering clause.

6) Case:

A) If *WDEF* simply contains <window frame clause> *WDEFWFC*, then *WDESC*'s window framing clause is *WDEFWFC*.

B) Otherwise, *WDESC* has no window framing clause.

7) If *WDEF* simply contains <row pattern common syntax>, then *WDESC*'s window row pattern recognition clauses are the <row pattern measures> and <row pattern common syntax> simply contained in *WDEF*; otherwise, *WDESC* has no window row pattern recognition clauses.

ii) The result of <window clause> is *RTE*, together with the window structure descriptors defined by the <window clause>.

2) Let *WD* be a window structure descriptor.

3) *WD* defines, for each row *R* of *RTE*, the *window partition* of *R* under *WD*, consisting of the collection of rows of *RTE* that are not distinct from *R* in the window partitioning columns of *WD*. If *WD* has no window partitioning clause, then the window partition of *R* is the entire result *RTE*.

4) *WD* also defines the window ordering of the rows of each window partition defined by *WD*, according to the General Rules of Subclause 10.10, "<sort specification list>", using the <sort specification list> simply contained in *WD*'s window ordering clause. If *WD* has no window ordering clause, then the window ordering is implementation-dependent (US034), and all rows are peers. Although the window ordering of peer rows within a window partition is implementation-dependent (US035), the window ordering shall be the same for all window structure descriptors that are order-equivalent. It shall also be the same for any pair of windows *W1* and *W2* such that *W1* is the ordering window for *W2*.

5) *WD* also defines for each row *R* of *RTE* the full window frame *WF* of *R*, consisting of a collection of rows. *WF* is defined as follows.

Case:

a) If *WD* has no window framing clause, then

Case:

i) If the window ordering clause of *WD* is not present, then *WF* is the window partition of *R*.

ii) Otherwise, *WF* consists of all rows of the window partition of *R* that precede *R* or are peers of *R* in the window ordering of the window partition defined by the window ordering clause.

b) Otherwise, let *WF* initially be the window partition of *R* defined by *WD*. Let *WFC* be the window framing clause of *WD*. Let *WFB1* be the <window frame bound 1> and let *WFB2* be the <window frame bound 2> contained in *WFC*.

i) Case:

1) If RANGE is specified, then:

A) In the following subrules, when performing addition or subtraction to combine a datetime and a year-month interval, if the result would raise the exception condition *data exception — datetime field overflow (22008)* because the <primary datetime field> DAY is not valid for the computed value of the <primary datetime field>s YEAR and MONTH, then the <primary datetime field> DAY is set to the last day that is valid for the <primary datetime field>s YEAR and MONTH, and no exception condition is raised.

B) Case:

NOTE 282 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* is not permitted to be UNBOUNDED FOLLOWING.

I) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

1) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

a) If *VSK* is the null value and if NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

b) If *VSK* is not the null value, then:

i) If NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

ii) Case:

1) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK*+*V1P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

2) Otherwise, let *BOUND* be the value *VSK*–*V1P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

II) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows that are not peers of the current row and that precede the current row in the window ordering defined by *WD*.

III) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

1) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

a) If *VSK* is the null value and if NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

b) If *VSK* is not the null value, then:

i) If NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

ii) Case:

1) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK*−*V1F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

2) Otherwise, let *BOUND* be the value *VSK*+*V1F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

C) Case:

NOTE 283 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* is not permitted to be UNBOUNDED PRECEDING.

I) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

1) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

a) If *VSK* is the null value and if NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

b) If *VSK* is not the null value, then:

i) If NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

ii) Case:

1) If the <ordering specification> contained in the window ordering clause specifies DESC, then let *BOUND* be the value *VSK*+*V2P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

2) Otherwise, let *BOUND* be the value *VSK*−*V2P*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

II) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows following the current row in the ordering defined by *WD* that are not peers of the current row.

III) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

1) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, let *SK* be the only <sort key> contained in the window ordering clause of *WD*. Let *VSK* be the value of *SK* for the current row.

Case:

a) If *VSK* is the null value and if NULLS FIRST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is not the null value.

b) If *VSK* is not the null value, then:

i) If NULLS LAST is specified or implied, then remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is the null value.

ii) Case:

1) If the <ordering specification> contained in the <window order clause> specifies DESC, then let *BOUND* be the value *VSK–V2F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is less than *BOUND*.

2) Otherwise, let *BOUND* be the value *VSK+V2F*. Remove from *WF* all rows *R2* such that the value of *SK* in row *R2* is greater than *BOUND*.

2) If ROWS is specified, then:

A) Case:

NOTE 284 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* is not permitted to be UNBOUNDED FOLLOWING.

I) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

1) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows that are more than *V1P* rows preceding the current row in the window ordering defined by *WD*.

II) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows that precede the current row in the window ordering defined by *WD*.

NOTE 285 — This step removes any peers of the current row that precede it in the implementation-dependent window ordering.

III) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

1) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows that precede the current row and all rows that are less than *V1F* rows following the current row in the window ordering defined by *WD*.

NOTE 286 — If *V1F* is zero, then the current row is not removed from *WF* by this step; otherwise, the current row is removed from *WF*.

B) Case:

NOTE 287 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* is not permitted to be UNBOUNDED PRECEDING.

I) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

1) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows that follow the current row and all rows that are less than *V2P* rows preceding the current row in the window ordering defined by *WD*.

NOTE 288 — If *V2P* is zero, then the current row is not removed from *WF* by this step; otherwise, the current row is removed from *WF*.

II) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows that follow the current row in the window ordering defined by *WD*.

NOTE 289 — This step removes any peers of the current row that follow it in the implementation-dependent window ordering.

III) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

1) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows that are more than *V2F* rows following the current row in the window ordering defined by *WD*.

3) If GROUPS is specified, then:

A) The rows of the window partition are placed in *window ordering groups*, as follows: two rows are in the same window ordering group if and only if they are peers with regard to the <sort specification list>, according to the General Rules of Subclause 10.10, "<sort specification list>".

B) Window ordering groups within a window partition are ordered as follows: window ordering group *WOG1* precedes window ordering group *WOG2* if some row *R1* in *WOG1* precedes some row *R2* in *WOG2* in the window

ordering, according to the General Rules of Subclause 10.10, "<sort specification list>".

NOTE 290 — Since all rows in a window ordering group are peers with regard to the <sort specification list>, if one row of *WOG1* precedes one row of *WOG2*, then all rows of *WOG1* precede all rows of *WOG2*.

C) The *distance* between two window ordering groups *WOG1* and *WOG2* is number of window ordering groups between *WOG1* and *WOG2*, inclusive, minus 1 (one).

NOTE 291 — Thus the distance between a window ordering group *WOG* and itself is 1 (one) − 1 (one) = 0 (zero).

D) The *current window ordering group* is the window ordering group that contains the current row.

E) Case:

NOTE 292 — In the following subrules, if *WFB1* specifies UNBOUNDED PRECEDING, then no rows are removed from *WF* by this step. *WFB1* cannot be UNBOUNDED FOLLOWING.

I) If *WFB1* specifies <window frame preceding>, then let *V1P* be the value of the <unsigned value specification>.

Case:

1) If *V1P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows whose window ordering group precedes the current window ordering group by a distance greater than *V1P*.

II) If *WFB1* specifies CURRENT ROW, then remove from *WF* all rows in all window ordering groups preceding the current window ordering group.

NOTE 293 — Thus CURRENT ROW is equivalent to 0 PRECEDING.

III) If *WFB1* specifies <window frame following>, then let *V1F* be the value of the <unsigned value specification>.

Case:

1) If *V1F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows whose window ordering group precedes the current window ordering group, and all rows whose window ordering group follows the current window ordering group by a distance less than *V1F*. If *V1F* is positive, then also remove all rows of the current window ordering group from *WF*.

F) Case:

NOTE 294 — In the following subrules, if *WFB2* specifies UNBOUNDED FOLLOWING, then no rows are removed from *WF* by this step. *WFB2* cannot be UNBOUNDED PRECEDING.

I) If *WFB2* specifies <window frame preceding>, then let *V2P* be the value of the <unsigned value specification>.

Case:

1) If *V2P* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows whose window ordering group precedes the current window ordering group by a distance less than *V2P*, and all rows whose window ordering group follows the current window ordering group. If *V2P* is positive, also remove all rows of the current window ordering group from *WF*.

II) If *WFB2* specifies CURRENT ROW, then remove from *WF* all rows in all window ordering groups following the current window ordering group.

NOTE 295 — Thus CURRENT ROW is equivalent to 0 FOLLOWING.

III) If *WFB2* specifies <window frame following>, then let *V2F* be the value of the <unsigned value specification>.

Case:

1) If *V2F* is negative or the null value, then an exception condition is raised: *data exception — invalid preceding or following size in window function (22013)*.

2) Otherwise, remove from *WF* all rows whose window ordering group follows the current window ordering group by a distance greater than *V2F*.

ii) If <window frame exclusion> *WFE* is specified, then

Case:

1) If EXCLUDE CURRENT ROW is specified and the current row is still a member of *WF*, then remove the current row from *WF*.

2) If EXCLUDE GROUP is specified, then remove the current row and any peers of the current row from *WF*.

3) If EXCLUDE TIES is specified, then remove any rows other than the current row that are peers of the current row from *WF*.

NOTE 296 — If the current row is already removed from *WF*, then it remains removed from *WF*.

NOTE 297 — If EXCLUDE NO OTHERS is specified, then no additional rows are removed from *WF* by this Rule.

6) If <row pattern common syntax> is specified, then:

a) Let *NWP* be the number of distinct window partitions. Let WP_i, 1 (one) $\leq i \leq$ *NWP*, be an enumeration of the distinct window partitions.

b) For every row *R* of *RTE*, let *WF*(*R*) denote the full window frame of *R* as computed by the preceding rules.

c) Let *RPCS* be the <row pattern common syntax>. Let *RP* be the <row pattern> simply contained in *RPCS*, let *RPDL* be the <row pattern definition list> simply contained in *RPCS*, let *RPSUB* be the <row pattern subset clause> implied by *RPCS* by adding the universal row pattern variable, and let *RPST* be the <row pattern skip to> simply contained in *RPCS*.

d) For all i, 1 (one) $\leq i \leq NWP$:

i) Let NR be the number of rows in WP_i. Let $WP_i = \{ R_1, R_2, ..., R_{NR} \}$ be the enumeration of the rows of WP_i according to the window ordering.

ii) For all j, 1 (one) $\leq j \leq NR$, every row R_j of WP_i is provided a *skip indicator*, initially set to *False*.

iii) For all j, 1 (one) $\leq j \leq NR$,

Case:

1) If the skip indicator of R_j is *True*, then the window frame $WF(R_j)$ of R_j is emptied of all rows.

2) Otherwise:

A) The General Rules of Subclause 9.41, "Row pattern recognition in a sequence of rows", are applied with $WF(R_j)$ as *ROW SEQUENCE*, *RP* as *PATTERN*, *RPSUB* as *SUBSETS*, and *RPDL* as *DEFINE*; let *SOM* be the *SET OF MATCHES* returned from the application of those General Rules.

B) The *designated row pattern match* of R_j is determined as follows.

Case:

I) If *SOM* is empty, then there is no designated row pattern match of R_j.

II) If the <row pattern initial or seek> contained in *RPCS* specifies SEEK, then the designated row pattern match of R_j is the preferred row pattern match (*STR*, $WF(R_j)$, k) in *SOM* with minimal k.

NOTE 298 — The initial row number k is relative to R_j and not to R_1. The numbering begins with 1 (one), not with j. Thus, the row indicated by initial row number k is R_{j+k-1}.

III) Otherwise, the designated row pattern match of R_j is the preferred row pattern match (*STR*, $WF(R_j)$, 1) in *SOM* whose initial row number is 1 (one), if there is such; otherwise, there is no designated row pattern match of R_j.

C) Let v be the variable count of *STR*, as defined in Subclause 9.41, "Row pattern recognition in a sequence of rows".

D) Case:

I) If there is no designated row pattern match, then the window frame $WF(R_j)$ of R_j is emptied of all rows. The skip indicator of R_j is set to *True*.

II) Otherwise, the window frame $WF(R_j)$ of R_j is the set of rows $\{ R_b \mid j + k - 1 \leq b < j + k + v - 1 \}$.

NOTE 299 — These are the rows that are mapped by the designated row pattern match.

E) If there is a designated row pattern match (*STR*, *WF*(R_j), *k*), then

Case:

I) If *v* is 0 (zero) or *RPST* specifies SKIP TO NEXT ROW, then no skip indicators are set to *True*.

II) If *RPST* specifies SKIP PAST LAST ROW, then for every *b* such that $j < b < j + k + v - 1$, the skip indicator of R_b is set to *True*.

NOTE 300 — These are the rows that are mapped by (*STR*, *WF*(R_j), *k*), or are prior to such rows.

III) If *RPST* specifies SKIP TO FIRST <row pattern skip to variable name> or specifies or implies SKIP TO LAST <row pattern skip to variable name>, then let *RPV* be the row pattern variable identified by the <row pattern skip to variable name>. Let { c_1, ..., c_u } be the collection of subscripts *c* such that R_c is mapped to *RPV*.

NOTE 301 — The subscripts *c* are within the window partition WP_i, not within the window frame *WF*(R_j).

Case:

1) If the collection of subscripts { c_1 ... c_u } is empty, then an exception condition is raised: *data exception — skip to non-existent row (2202K)*.

2) If SKIP TO FIRST is specified, then let $t = \min \{ c_1, ..., c_u \}$. If $t = k+j-1$, then an exception condition is raised: *data exception — skip to first row of match (2202L)*; otherwise, for every *b* such that $j < b < t$, the skip indicator of R_b is set to *True*.

3) Otherwise, let $t = \max \{ c_1, ..., c_u \}$. If $t = k+j-1$, then an exception condition is raised: *data exception — skip to first row of match (2202L)*; otherwise, for every *b* such that $j < b < t$, the skip indicator of R_b is set to *True*.

Conformance Rules

1) Without Feature R020, "Row pattern recognition: WINDOW clause", <window specification details> shall not contain <row pattern measures> or <row pattern common syntax>.

2) Without Feature T611, "Elementary OLAP operations", conforming SQL language shall not contain a <window specification>.

3) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window clause>.

4) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain an <existing window name>.

5) Without Feature T301, "Functional dependencies", in conforming SQL language, if *T* is a grouped table, then each column reference contained in <window clause> that references a column of *T* shall be a reference to a grouping column of *T* or be contained in an aggregated argument of a <set function specification>.

6) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <window frame exclusion>.

NOTE 302 — The Conformance Rules of Subclause 9.12, "Grouping operations", also apply.

7) Without Feature T620, "WINDOW clause: GROUPS option", conforming SQL language shall not contain <window frame units> that specifies GROUPS.

7.16 <query specification>

This Subclause is modified by Subclause 7.2, "<query specification>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 7.2, "<query specification>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 7.2, "<query specification>", in ISO/IEC 9075-16.

Function

Specify a table derived from the result of a <table expression>.

Format

```
<query specification> ::=
  SELECT [ <set quantifier> ] <select list> <table expression>

<select list> ::=
    <asterisk>
  | <select sublist> [ { <comma> <select sublist> }... ]

16 <select sublist> ::=
    <derived column>
  | <qualified asterisk>

<qualified asterisk> ::=
    <asterisked identifier chain> <period> <asterisk>
  | <all fields reference>

<asterisked identifier chain> ::=
  <asterisked identifier> [ { <period> <asterisked identifier> }... ]

<asterisked identifier> ::=
  <identifier>

<derived column> ::=
  <value expression> [ <as clause> ]

<as clause> ::=
  [ AS ] <column name>

<all fields reference> ::=
  <value expression primary> <period> <asterisk>
      [ AS <left paren> <all fields column name list> <right paren> ]

<all fields column name list> ::=
  <column name list>
```


Syntax Rules

1) Let *QS* be the <query specification>.

2) Let *T* be the result of the <table expression> simply contained in *QS*.

3) Let *TQS* be the result of *QS*.

4) The degree of the table specified by a <query specification> is equal to the cardinality of the <select list>.

5) Let *C* be some column. Let DC_i, for *i* ranging from 1 (one) to the number of <derived column>s, be the *i*-th <derived column> simply contained in the <select list> of *QS*.

a) For all *i*, *C* is an *underlying column* of DC_i, and of any column reference that identifies DC_i, if *C* is an underlying column of the <value expression> of DC_i, or *C* is an underlying column of the <table expression> immediately contained in *QS*.

b) For all *i*, *C* is a *generally underlying column* of DC_i, and of any column reference that identifies DC_i, if *C* is a generally underlying column of the <value expression> of DC_i, or *C* is a generally underlying column of the <table expression> immediately contained in *QS*.

6) Each column reference contained in a <window function> shall unambiguously reference a column of *T*.

7) Case:

a) If the <select list> "*" is simply contained in a <table subquery> that is immediately contained in an <exists predicate>, then the <select list> is equivalent to a <value expression> that is an arbitrary <literal>.

b) Otherwise, the <select list> "*" is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of *T* and each column of *T* is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within *T*.

8) If a <set quantifier> DISTINCT is specified, then each column of *T* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

9) The ambiguous case of an <all fields reference> whose <value expression primary> takes the form of an <asterisked identifier chain> shall be analyzed first as an <asterisked identifier chain> to resolve the ambiguity.

10) If <asterisked identifier chain> is specified, then:

a) Let *IC* be an <asterisked identifier chain>.

b) Let *N* be the number of <asterisked identifier>s immediately contained in *IC*.

c) Let I_i, 1 (one) ≤ *i* ≤ *N*, be the <asterisked identifier>s immediately contained in *IC*, in order from left to right.

d) Let PIC_1 be I_1. For each *J* between 2 and *N*, let PIC_J be $PIC_{J-1}.I_J$. PIC_J is called the *J*-th *partial identifier chain* of *IC*.

e) Let *M* be the minimum of *N* and 3.

f) For at most one *J* between 1 and *M*, PIC_J is called the *basis* of *IC*, and *J* is called the *basis length* of *IC*. The *referent* of the basis is a table *T*, a column *C* of a table, or an SQL parameter *SP*. The *basis* and *basis scope* of *IC* are defined in terms of a *candidate basis*, according to the following rules:

i) 04 If *IC* is contained in the scope of a <routine name> whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_1, then PIC_1 is a candidate basis of *IC*, and the scope of PIC_1 is the scope of *SP*.

ii) If *N* = 2 and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes an SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iii) If $N > 2$ and PIC_1 is equivalent to the <qualified identifier> of a <routine name> *RN* whose scope contains *IC* and whose associated <SQL parameter declaration list> includes a refinable SQL parameter *SP* whose <SQL parameter name> is equivalent to I_2, then PIC_2 is a candidate basis of *IC*, the scope of PIC_2 is the scope of *SP*, and the referent of PIC_2 is *SP*.

iv) If $N = 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. If the table associated with *EN* has a column *C* of row type whose <identifier> is equivalent to I_2, then PIC_2 is a candidate basis of *IC* and the scope of PIC_2 is the scope of *EN*.

v) If $N > 2$ and PIC_1 is equivalent to an exposed <correlation name> that is in scope, then let *EN* be the exposed <correlation name> that is equivalent to PIC_1 and has innermost scope. If the table associated with *EN* has a refinable column *C* whose <identifier> is equivalent to I_2, then PIC_2 is a candidate basis of *IC* and the scope of PIC_2 is the scope of *EN*.

vi) If $N \leq 3$ and PIC_N is equivalent to an exposed <table or query name> that is in scope, then let *EN* be the exposed <table or query name> that is equivalent to PIC_N and has the innermost scope. PIC_N is a candidate basis of *IC*, and the scope of PIC_N is the scope of *EN*.

vii) There shall be exactly one candidate basis *CB* with innermost scope. The basis of *IC* is *CB*. The basis scope is the scope of *CB*.

g) Case:

i) If the basis is a <table or query name> or <correlation name>, then let *TQ* be the table associated with the basis. The <select sublist> is equivalent to a <value expression> sequence in which each <value expression> is a column reference *CR* that references a column of *TQ* that is not a common column of a <joined table>. Each column of *TQ* that is not a common column shall be referenced exactly once. The columns shall be referenced in the ascending sequence of their ordinal positions within *TQ*.

ii) Otherwise let *BL* be the length of the basis of *IC*.

Case:

1) If $BL = N$, then the <select sublist> `IC.*` is equivalent to `(IC).*`.

2) Otherwise, the <select sublist> `IC.*` is equivalent to:

$$(\ PIC_{BL}\ ).I_{BL+1}.\ldots.I_N.*$$

NOTE 303 — The equivalent syntax in either case will be analyzed as <all fields reference> ::= <value expression primary> <period> <asterisk>

11) The data type of the <value expression primary> *VEP* specified in an <all fields reference> *AFR* shall be some row type *VER*. Let *n* be the degree of *VER*. Let F_1, ..., F_n be the field names of *VER*.

Case:

a) If <all fields column name list> *AFCNL* is specified, then the number of <column name>s simply contained in *AFCNL* shall be *n*. Let $AFCN_i$, 1 (one) $\leq i \leq n$, be these <column name>s in order from left to right. *AFR* is equivalent to

$$VEP.F_1\ \texttt{AS}\ AFCN_1,$$

```
...,
VEP.Fn AS AFCNn
```

b) Otherwise, *AFR* is equivalent to:

```
VEP.F1,
...,
VEP.Fn
```

12) If and only if all of the following are true, then *QS* is a *grouped, windowed query*:

a) *T* is a grouped table.

b) Some <derived column> simply contained in *QS* simply contains a <window function>.

13) Case:

a) If *QS* is a grouped, windowed query, then the Syntax Rules of Subclause 9.20, "Transformation of query specifications", are applied with *QS* as *QUERY SPEC IN*; let *XQS* be the *QUERY SPEC OUT* returned from the application of those Syntax Rules.

b) Otherwise, let *XQS* be *QS*.

14) If <table expression> does not immediately contain a <group by clause> and <table expression> is simply contained in a <query specification> that is the aggregation query of some <set function specification>, then `GROUP BY ()` is implicit.

NOTE 304 — "aggregation query" is defined in Subclause 6.9, "<set function specification>".

15) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. In each <value expression> contained in <select list>, each column reference that references a column of *T* shall reference some column *C* that is functionally dependent on *G* or shall be contained in an aggregated argument of a <set function specification> whose aggregation query is *XQS*.

NOTE 305 — The preceding Syntax Rule follows from a more comprehensive one in Subclause 6.7, "<column reference>".

16) Each column of *TQS* has a column descriptor that includes a data type descriptor that is the same as the data type descriptor of the <value expression> simply contained in the <derived column> defining that column.

17) Case:

a) If the *i*-th <derived column> in the <select list> specifies an <as clause> that contains a <column name> *CN*, then the <column name> of the *i*-th column of the result is *CN*.

b) If the *i*-th <derived column> in the <select list> does not specify an <as clause> and the <value expression> of that <derived column> is a single column reference, then the <column name> of the *i*-th column of the result is the <column name> of the column designated by the column reference.

c) 04 If the *i*-th <derived column> in the <select list> does not specify an <as clause> and the <value expression> of that <derived column> is a single SQL parameter reference, then the <column name> of the *i*-th column of the result is the <SQL parameter name> of the SQL parameter designated by the SQL parameter reference.

d) Otherwise, the <column name> of the *i*-th column of the <query specification> is implementation-dependent (UV087).

18) A column *C* of *TQS* is *readily known not null* if *C* is defined (after performing syntactic transformations to eliminate <asterisk> and <qualified asterisk>), by a <derived column> that immediately contains

a <value expression> that is a <column reference> *CR* such that the column referenced by *CR* is a readily known not null column of the qualifying table of *CR*.

19) A column *CKNN* of *TQS* is *known not null* if and only if at least one of the following is true:

a) *CKNN* is readily known not null.

b) The SQL-implementation supports Feature T101, "Enhanced nullability determination" and *CKNN* is not defined by a <derived column> containing any of the following:

i) A column reference for a column *C* that is possibly nullable.

ii) An <indicator parameter>.

iii) An <indicator variable>.

iv) 04 A <dynamic parameter specification>.

v) An SQL parameter.

vi) A <routine invocation>, <method reference>, or <method invocation> whose subject routine is an SQL-invoked routine that either is an SQL routine or is an external routine that specifies or implies PARAMETER STYLE SQL.

vii) A <query expression>.

viii) CAST (NULL AS *X*) (where *X* represents a <data type> or a <domain name>).

ix) A <window function> whose <window function type> does not contain <rank function type>, ROW_NUMBER, or an <aggregate function> that simply contains COUNT.

x) CURRENT_USER, CURRENT_ROLE, or SYSTEM_USER.

xi) A <set function specification> that does not simply contain COUNT.

xii) A <case expression>.

xiii) A <field reference>.

xiv) 15 An <array element reference>.

xv) A <multiset element reference>.

xvi) A <dereference operation>.

xvii) A <reference resolution>.

xviii) A <comparison predicate>, <between predicate>, <in predicate>, or <quantified comparison predicate> *P* such that the declared type of a field of a <row value predicand> that is simply contained in *P* is a row type, a user-defined type, an array type, or a multiset type.

xix) A <member predicate>.

xx) A <submultiset predicate>.

c) An implementation-defined (IE016) rule by which the SQL-implementation can correctly deduce that the value of the column cannot be null.

20) After applying all relevant syntactic transformations, let *TREF* be the <table reference>s that are simply contained in the <from clause> of the <table expression>. The *simply underlying table specifications* of the <query specification> are the <table or query name>s and <query expression>s contained in *TREF* without an intervening <query expression> or <data change delta table>.

NOTE 306 — Underlying tables are only used in rules regarding views and SQL-data change statements. Since <data change delta table> is not permitted in such contexts, <data change delta table> is omitted from the analysis of underlying table specifications.

21) The terms *key-preserving* and *one-to-one* are defined as follows:

a) Let *UT* denote some simply underlying table specification of *XQS*, let *UTCOLS* be the set of columns of *UT*, let *QSCOLS* be the set of columns of *XQS*, and let *QSCN* be an exposed range variable for *UT* whose scope clause is *XQS*.

b) *XQS* is said to be *key-preserving with respect to UT* if there is some strong candidate key *CKUT* of *UT* such that every member of *CKUT* has some counterpart under *QSCN* in *QSCOLS*.

NOTE 307 — "strong candidate key" is defined in Subclause 4.27, "Candidate keys".

NOTE 308 — "Counterpart" is defined in Subclause 4.26.2, "General rules and definitions". It follows from this condition that every row in *XQS* corresponds to exactly one row in *UT*, namely that row in *UT* that has the same combined value in the columns of *CKUT* as the row in *XQS*. There can be more than one row in *XQS* that corresponds to a single row in *UT*.

c) *XQS* is said to be *one-to-one with respect to UT* if and only if *XQS* is key-preserving with respect to *UT*, *UT* is generally updatable, and there is some strong candidate key *CKQS* of *XQS* such that every member of *CKQS* is a counterpart under *UT* of some member of *UTCOLS*.

NOTE 309 — It follows from this condition that every row in *UT* corresponds to at most one row in *XQS*, namely that row in *XQS* that has the same combined value in the columns of *CKQS* as the row in *UT*.

22) A <query specification> is *potentially updatable* if and only if all of the following are true:

a) *XQS* does not immediately contain a <set quantifier> that specifies DISTINCT.

b) Of those <derived column>s in the <select list> that are column references that have a counterpart in a base table, no column of a table is referenced more than once in the <select list>.

c) The <table expression> immediately contained in *XQS* does not simply contain an explicit or implicit <group by clause> or a <having clause>.

d) The <table expression> immediately contained in *QS* does not simply contain an explicit or implicit <query system time period specification> that does not specify FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP.

e) The <table expression> immediately contained in *XQS* does not generally contain a <data change delta table>.

23) If a <query specification> *XQS* is potentially updatable, then

Case:

a) If the <from clause> of the <table expression> specifies exactly one <table reference>, then a column of *XQS* is said to be a *potentially updatable column* if it has a counterpart in *TR* that is updatable.

NOTE 310 — The notion of updatable columns of table references is defined in Subclause 7.6, "<table reference>".

b) Otherwise, a column of *XQS* is said to be a *potentially updatable column* if it has a counterpart in some updatable column of some simply underlying table specification *UT* of *XQS* such that *XQS* is one-to-one with respect to *UT*.

24) A <query specification> is *generally updatable* if it is potentially updatable and it has at least one potentially updatable column.

25) A <query specification> *XQS* is *simply updatable* if and only if all of the following are true:

a) *XQS* is generally updatable.

b) The <from clause> immediately contained in the <table expression> immediately contained in *XQS* contains exactly one <table reference>, and the table referenced by that <table reference> is simply updatable.

c) Every result column of *XQS* is potentially updatable.

d) If the <table expression> immediately contained in *XQS* immediately contains a <where clause> *WC*, then no leaf generally underlying table of *XQS* is a leaf generally underlying table of any <query expression> contained in *WC*.

26) A <query specification> *XQS* is *effectively updatable* if either *XQS* is simply updatable or if the SQL-implementation supports Feature T111, "Updatable joins, unions, and columns" and *XQS* is generally updatable.

27) Let *EQS* be an effectively updatable <query specification>, and let *SUT* be a simply underlying table specification of *EQS*. *SUT* is a *target simply underlying table specification* of *EQS* if and only if at least one of the following is true:

a) *EQS* is simply updatable.

b) *EQS* is one-to-one with *SUT*.

28) A <query specification> *IQS* is *insertable-into* if *IQS* is effectively updatable and every simply underlying table specification of *IQS* is insertable-into.

29) A column *C* of *XQS* is updatable if and only if at least one of the following is true:

a) *XQS* is simply updatable.

b) *XQS* is generally updatable, *C* is potentially updatable, and the SQL-implementation supports Feature T111, "Updatable joins, unions, and columns".

30) The row type *RT* of *TQS* is defined by the sequence of (<field name>, <data type>) pairs indicated by the sequence of column descriptors of *TQS* taken in order.

Access Rules

None.

General Rules

1) Case:

a) If *T* is not a grouped table, then each <value expression> is applied to each row of *T* yielding a table *TEMP* of *M* rows, where *M* is the cardinality of *T*. The *i*-th column of the table contains the values derived by the evaluation of the *i*-th <value expression>.

b) If *T* is a grouped table, then

Case:

i) If *T* has 0 (zero) groups, then let *TEMP* be an empty table.

ii) If *T* has one or more groups, then each <value expression> is applied to each group of *T* yielding a table *TEMP* of *M* rows, where *M* is the number of groups in *T*. The *i*-th column of *TEMP* contains the values derived by the evaluation of the *i*-th <value expression>. When a <value expression> is applied to a given group of *T*, that group is the argument source of each <set function specification> whose aggregation query is *QS*.

2) Case:

   a) If the <set quantifier> DISTINCT is not specified, then the result of the <query specification> is *TEMP*.

   b) If the <set quantifier> DISTINCT is specified, then the result of the <query specification> is the table derived from *TEMP* by the elimination of all redundant duplicate rows. If the most specific type of any column is character string, datetime with time zone, or a user-defined type, then the precise values in those columns are chosen in an implementation-dependent (UV088) fashion.

Conformance Rules

1) Without Feature F801, "Full set function", conforming SQL language shall not contain a <query specification> *QS* that contains more than one <set quantifier> containing DISTINCT, unless such <set quantifier> is contained in a <query expression> contained in *QS*.

2) Without Feature T051, "Row types", conforming SQL language shall not contain an <all fields reference>.

3) Without Feature T301, "Functional dependencies", in conforming SQL language, if *T* is a grouped table, then in each <value expression> contained in the <select list>, each <column reference> that references a column of *T* shall reference a grouping column or be specified in an aggregated argument of a <set function specification>.

4) Without Feature T325, "Qualified SQL parameter references", conforming SQL language shall not contain an <asterisked identifier chain> whose referent is an SQL parameter and whose first <identifier> is the <qualified identifier> of a <routine name>.

5) Without Feature T053, "Explicit aliases for all-fields reference", conforming SQL language shall not contain an <all fields column name list>.

   NOTE 311 — If a <set quantifier> DISTINCT is specified, then the Conformance Rules of Subclause 9.12, "Grouping operations", also apply.

6) 04 16 Without Feature T285, "Enhanced derived column names", in conforming SQL language, if any <derived column> in a <select list> does not specify an <as clause> and the <value expression> of that <derived column> is not a single column reference, then the <column name> of that column is implementation-dependent (UV087).

7.17 <query expression>

This Subclause is modified by Subclause 7.2, "<query expression>", in ISO/IEC 9075-14.

Function

Specify a table.

Format

```
<query expression> ::=
  [ <with clause> ] <query expression body>
      [ <order by clause> ] [ <result offset clause> ] [ <fetch first clause> ]

14 <with clause> ::=
  WITH [ RECURSIVE ] <with list>

<with list> ::=
  <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
  <query name> [ <left paren> <with column list> <right paren> ]
      AS <table subquery> [ <search or cycle clause> ]

<with column list> ::=
  <column name list>

<query expression body> ::=
    <query term>
  | <query expression body> UNION [ ALL | DISTINCT ]
        [ <corresponding spec> ] <query term>
  | <query expression body> EXCEPT [ ALL | DISTINCT ]
        [ <corresponding spec> ] <query term>

<query term> ::=
    <query primary>
  | <query term> INTERSECT [ ALL | DISTINCT ]
        [ <corresponding spec> ] <query primary>

<query primary> ::=
    <simple table>
  | <left paren> <query expression body>
        [ <order by clause> ] [ <result offset clause> ] [ <fetch first clause> ]
    <right paren>

<simple table> ::=
    <query specification>
  | <table value constructor>
  | <explicit table>

<explicit table> ::=
  TABLE <table or query name>

<corresponding spec> ::=
  CORRESPONDING [ BY <left paren> <corresponding column list> <right paren> ]

<corresponding column list> ::=
  <column name list>

<order by clause> ::=
  ORDER BY <sort specification list>
```

```
<result offset clause> ::=
  OFFSET <offset row count> { ROW | ROWS }

<fetch first clause> ::=
  FETCH { FIRST | NEXT } [ <fetch first quantity> ] { ROW | ROWS } { ONLY | WITH TIES }

<fetch first quantity> ::=
    <fetch first row count>
  | <fetch first percentage>

<offset row count> ::=
  <simple value specification>

<fetch first row count> ::=
  <simple value specification>

<fetch first percentage> ::=
  <simple value specification> PERCENT
```

Syntax Rules

1) Let *QE* be the <query expression> and let *QEBODY* be the <query expression body> immediately contained in *QE*. Let *T* be the table specified by *QE*.

2) If *QE* simply contains a <data change delta table>, then *QEBODY* shall be a <query specification>.

3) If <with clause> *WC* is specified, then:

   a) Let *n* be the number of <with list element>s.

   b) For *i* ranging from 1 (one) to *n*, let WLE_i be the *i*-th <with list element> of *WL*, let WQN_i be the <query name> immediately contained in WLE_i, let WQE_i be the <query expression> simply contained in WLE_i, and let WQT_i be the table defined by WQE_i.

   c) For each *i*, 1 (one) ≤ *i* < *n*, for each *j*, *i* < *j* ≤ *n*, WQN_j shall not be equivalent to WQN_i.

   d) For every *i*, 1(one) ≤ *i* ≤ *n*:

      i) If any two columns of WQT_i have equivalent names, then WLE_i shall specify a <with column list>.

      ii) If WLE_i specifies a <with column list> *WCL*, then:

         1) Equivalent <column name>s shall not be specified more than once in *WCL*.

         2) The number of <column name>s in *WCL* shall be the same as the degree of WQT_i.

      iii) Every column of a character string type in WQT_i shall have a declared type collation.

      iv) If WLE_i generally contains a <data change delta table>, then there shall be exactly one occurrence of <data change delta table> in WLE_i.

   e) If *WC* immediately contains RECURSIVE, then *WC*, its <with list>, and its <with list element>s are said to be *potentially recursive*. Otherwise, they are said to be *non-recursive*.

   f) A potentially recursive <with list element> shall not contain a <row pattern measures> or <row pattern common syntax>.

   g) A potentially recursive <with list element> shall not generally contain a <data change delta table>.

h) A potentially recursive <with list element> shall not contain a potentially recursive <with list>.

i) Case:

i) If *WC* is non-recursive, then, for all *i*, 1 (one) ≤ *i* ≤ *n*, the scope of the <query name> WQN_i is *QEBODY* and WQE_k, *i*+1 ≤ *k* ≤ *n*. A <table or query name> contained in this scope that is equivalent to WQN_i is a *query name in scope*.

ii) If *WC* is potentially recursive, then for all *i* between 1 (one) and *n*, the scope of WQN_i is the *QEBODY* and WQE_k, 1 (one) ≤ *k* ≤ *n*. A <table or query name> contained in this scope that is equivalent to WQN_i is a *query name in scope*.

j) If *WC* is potentially recursive, then:

i) Every <with list element> contained in *WC* shall contain a <with column list>.

ii) The *query name dependency graph QNDG* of *WC* is the directed graph such that:

1) The nodes of *QNDG* are the <query name>s WQN_i, 1 (one) ≤ *i* ≤ *n*.

2) Each arc from a node WQN_i to a node WQN_j represents the fact that WQN_j is referenced by a <query name> contained in WQE_i. WQN_i is said to *depend immediately* on WQN_j.

iii) *WC* is said to be *recursive* if *QNDG* contains at least one cycle.

iv) For all *i*, 1 (one) ≤ *i* ≤ *n*, if WQN_i is a member of a cycle of *QNDG*, then WLE_i, WQN_i, and WQT_i are said to be *recursive*.

v) A subset *S* of *QNDG* is *strongly connected* if either the cardinality of *S* is 1 (one) or there is a path from every node of *S* to every other node of *S*. A *stratum* is a maximal strongly connected subset of *QNDG*.

NOTE 312 — A commonly used synonym for stratum is "strongly connected component". Note that any cycle is strongly connected. If *C1* and *C2* are two strongly connected sets of nodes with non-empty intersection, then their union *C1* ∪ *C2* is also strongly connected. Hence, maximal strongly connected sets are found by forming unions of overlapping cycles. Every <query name> WQN_i belongs to exactly one stratum; thus the set of strata forms a partitioning of *QNDG*.

vi) A stratum is *recursive* if its members are recursive.

NOTE 313 — If WQN_i is not recursive, then {WQN_i} constitutes a (non-recursive) stratum. The converse is not true; a singleton stratum might be recursive.

vii) For every recursive stratum $S = \{ WQN_{s(1)}, ..., WQN_{s(k)} \}$:

NOTE 314 — The subscripts on <query name>s in *S* are not necessarily consecutive integers. For example, if a stratum consists of the second, fifth and seventh <query name>s, $S = \{WQN_2, WQN_5, WQN_7\}$, then *k* = 3, *s*(1) = 2, *s*(2) = 5, and *s*(3) = 7.

1) Among the <query expression>s $WQN_{s(1)}$, ... $WQN_{s(k)}$ corresponding to the <query name>s in *S*, there shall be at least one <query expression>, say WQE_j, such that all of the following are true:

A) WQE_j simply contains a <query expression body> that immediately contains UNION.

B) WQE_j has one operand that does not contain a <query name> equivalent to any <query name> in *S*. This operand is said to be the *non-recursive operand* of WQE_j.

2) WQE_j is said to be an *anchor expression*, and WQN_j an *anchor name*.

viii) Every cycle of *QNDG* shall contain an anchor name.

ix) If WLE_i is recursive, then:

1) Let *S* be the stratum that contains WQN_i.

2) If WQE_i does not contain a <query specification> that contains more than one <query name> referencing members of *S*, then WLE_i is *linearly recursive*.

NOTE 315 — For example, if WLE_i contains the <query specification>

```
SELECT *
FROM A AS A1, A AS A2
```

where A is a <query name> in *S*, then WLE_i is not linearly recursive. The point is that this <query specification> contains two references to members of *S*. It is irrelevant that they reference the same member of *S*.

3) WQE_i shall not immediately contain a <fetch first clause>.

4) WQE_i shall not contain a <query primary> that contains a <fetch first clause> and that contains a <query name> that references a <query name> in the stratum of WQN_i.

5) For every <query name> *WQNX* in *S*:

A) WQE_i shall not contain a <routine invocation> with an <SQL argument list> that contains one or more <SQL argument>s that immediately contain a <value expression> that contains a <query name> referencing *WQNX*.

B) WQE_i shall not contain a <query expression> *TSQ* that contains a <query name> referencing *WQNX*, unless *TSQ* is simply contained in a <derived table> that is simply contained in a <from clause> that is simply contained in a <query specification> that constitutes a <simple table> that constitutes a <query primary> that constitutes a <query term> that is immediately contained in a <query expression body> that is simply contained in WQE_i.

C) WQE_i shall not contain a <query specification> *QS* such that *QS* immediately contains a <table expression> *TE* that contains a <query name> referencing *WQNX* and at least one of the following is true:

I) *TE* immediately contains a <having clause> that contains a <set function specification>.

II) *QS* immediately contains a <select list> *SL* that contains either a <window function>, or a <set function specification>, or both.

NOTE 316 — If a <window function> is contained in an <order by clause>, then the syntactic transformation in this Subclause that moves the <window function> to a <select sublist> is effectively applied before applying this rule.

D) WQE_i shall not contain a <query expression body> that contains a <query name> referencing *WQNX* and simply contains INTERSECT ALL or EXCEPT ALL.

E) WQE_i shall not contain a <query expression body> that immediately contains EXCEPT where the right operand of EXCEPT contains a <query name> referencing *WQNX*.

F) WQE_i shall not contain a <qualified join> *QJ* in which:

I) *QJ* immediately contains a <join type> that specifies FULL and a <table reference> or <partitioned join table> that contains a <query name> referencing *WQNX*.

II) *QJ* immediately contains a <join type> that specifies LEFT and a <table reference> or <partitioned join table> following the <join type> that contains a <query name> referencing *WQNX*.

III) *QJ* immediately contains a <join type> that specifies RIGHT and a <table reference> or <partitioned join table> preceding the <join type> that contains a <query name> referencing *WQNX*.

G) WQE_i shall not contain a <natural join> *QJ* in which:

I) *QJ* immediately contains a <join type> that specifies FULL and a <table reference>, <table factor>, or <partitioned join table> that contains a <query name> referencing *WQNX*.

II) *QJ* immediately contains a <join type> that specifies LEFT and a <table factor> or <partitioned join table> following the <join type> that contains a <query name> referencing *WQNX*.

III) *QJ* immediately contains a <join type> that specifies RIGHT and a <table reference> or <partitioned join table> preceding the <join type> that contains a <query name> referencing *WQNX*.

NOTE 317 — The restrictions insure that each WLE_i, viewed as a transformation of the query names of the stratum, is monotonically increasing. According to Tarski's fixed point theorem, this insures that there is a fixed point. The General Rules use Kleene's fixed point theorem to define a sequence that converges to the minimal fixed point.

x) If WLE_i is recursive, then WLE_i shall be linearly recursive.

xi) WLE_i is said to be *expandable* if and only if all of the following are true:

1) WLE_i is linearly recursive.

2) WQE_i is a <query expression body> that immediately contains UNION or UNION ALL. Let $WQEB_i$ be the <query expression body> immediately contained in WQE_i. Let QEL_i and QTR_i be the <query expression body> and the <query term> immediately contained in $WQEB_i$. WQN_i shall not be referenced in QEL_i, and QTR_i shall be a <query specification>.

3) There is no path from WQN_i to any other <query name> in *QNDG*.

k) If a <with list element> *WLE* is not expandable, then it shall not immediately contain a <search or cycle clause>.

4) The <explicit table>

```
TABLE <table or query name>
```

is equivalent to the <query specification>

```
SELECT * FROM <table or query name>
```

5) Let *set operator* be UNION ALL, UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT.

6) If UNION, EXCEPT, or INTERSECT is specified and neither ALL nor DISTINCT is specified, then DISTINCT is implicit.

7) A <query expression> *QE1* is *simply updatable* if *QE1* is not contained in a <with clause> that specifies RECURSIVE, *QE1* does not specify a <result offset clause> or a <fetch first clause> and, for every <query expression> or <query specification> *QE2* that is simply contained in the <query expression body> of *QE1*, all of the following are true:

   a) *QE1* contains *QE2* without an intervening <query expression body> that specifies UNION ALL, UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.

   b) *QE1* contains *QE2* without an intervening <query term> that specifies INTERSECT.

   c) *QE2* is simply updatable.

8) A <query expression> *QE1* is *generally updatable* if it is not contained in a <with clause> that specifies RECURSIVE, it does not specify a <result offset clause> or a <fetch first clause> and, for every <simple table> *QE2* that is simply contained in *QE1*, all of the following are true:

   a) *QE2* is not a <table value constructor>.

   b) *QE1* contains *QE2* without an intervening <query expression body> that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.

   c) If *QE1* simply contains a <query expression body> *QEB* that specifies UNION ALL, then:

      i) *QEB* immediately contains a <query expression body> *LO* and a <query term> *RO* such that no leaf generally underlying table of *LO* is also a leaf generally underlying table of *RO*.

      ii) For every column of *QEB*, the underlying columns in the tables identified by *LO* and *RO*, respectively, are either both updatable or not updatable.

   d) *QE1* contains *QE2* without an intervening <query term> that specifies INTERSECT.

   e) *QE2* is generally updatable.

9) A <query expression> *QE* is *effectively updatable* if either *QE* is simply updatable, or if *QE* is generally updatable and the SQL-implementation supports Feature T111, “Updatable joins, unions, and columns”.

10) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *generally updatable*, *simply updatable*, or *effectively updatable* if and only if the <query expression> simply contained in *WLE* is generally updatable, simply updatable, or effectively updatable, respectively.

11) A <query expression> *QE1* is *insertable-into* if it does not specify a <result offset clause> or a <fetch first clause> and the <query expression body> of *QE1* is a <query primary> that is one of the following:

   a) An insertable-into <query specification>.

   b) An <explicit table> that contains a <table or query name> that identifies a table that is insertable-into.

   c) Of the form <left paren> <query expression body> <right paren>, where the parenthesized <query expression body> recursively satisfies this condition.

12) A table specified by a <query name> immediately contained in a <with list element> *WLE* is *insertable-into* if the <query expression> simply contained in *WLE* is insertable-into.

13) For every <simple table> *ST* contained in *QE*,

Case:

a) If *ST* is a <query specification> *QS*, then the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of *QS*.

b) If *ST* is an <explicit table> *ET*, then the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of the table identified by the <table or query name> contained in *ET*.

c) Otherwise, the column descriptor of each column of *ST* is the same as the column descriptor of the corresponding column of the <table value constructor> immediately contained in *ST*.

14) For every <query primary> *QP* contained in *QE*,

Case:

a) If *QP* is a <simple table> *ST*, then the column descriptor of each column of *QP* is the same as the column descriptor of the corresponding column of *ST*.

b) Otherwise, the column descriptor of each column of *QP* is the same as the column descriptor of the corresponding column of the <query expression body> immediately contained in *QP*.

15) If a set operator is specified in a <query term> or a <query expression body>, then:

a) Let *T1*, *T2*, and *TR* be respectively the first operand, the second operand, and the result of the <query term> or <query expression body>.

b) Let *TN1* and *TN2* be the effective names for *T1* and *T2*, respectively.

c) If the set operator is UNION DISTINCT, EXCEPT ALL, EXCEPT DISTINCT, INTERSECT ALL, or INTERSECT DISTINCT, then each column of *T1* and *T2* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

16) If a set operator is specified in a <query term> or a <query expression body>, then let *OP* be the set operator.

Case:

a) If CORRESPONDING is specified, then:

i) Within the columns of *T1*, equivalent <column name>s shall not be specified more than once and within the columns of *T2*, equivalent <column name>s shall not be specified more than once.

ii) At least one column of *T1* shall have a <column name> that is the <column name> of some column of *T2*.

iii) Case:

1) If <corresponding column list> is not specified, then let *SL* be a <select list> of those <column name>s that are <column name>s of both *T1* and *T2* in the order that those <column name>s appear in *T1*.

2) If <corresponding column list> is specified, then let *SL* be a <select list> of those <column name>s explicitly appearing in the <corresponding column list> in the order that these <column name>s appear in the <corresponding column list>.

Every <column name> in the <corresponding column list> shall be a <column name> of both *T1* and *T2*.

iv) The <query term> or <query expression body> is equivalent to:

```
( SELECT SL FROM TN1 ) OP ( SELECT SL FROM TN2 )
```

b) If CORRESPONDING is not specified, then *T1* and *T2* shall be of the same degree.

17) If a <query term> is a <query primary>, then the declared type of the <query term> is that of the <query primary>. The column descriptor of the *i*-th column of the <query term> is the same as the column descriptor of the *i*-th column of the <query primary>.

18) If a <query term> immediately contains INTERSECT, then:

a) Let *C* be the <column name> of the *i*-th column of *T1*. If the <column name> of the *i*-th column of *T2* is *C*, then the <column name> of the *i*-th column of *TR* is *C*; otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent (UV089).

b) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the set comprising the declared type of the *i*-th column of *T1* and the declared type of the *i*-th column of *T2* as *DTSET*; let the declared type of the *i*-th column of *TR* be the *RESTYPE* returned from the application of those Syntax Rules.

c) If the SQL-implementation supports Feature T101, "Enhanced nullability determination" and the *i*-th column of at least one of *T1* and *T2* is known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

19) Case:

a) If a <query expression body> is a <query term>, then the column descriptors of the <query expression body> are the same as the column descriptors of the <query term>.

b) If a <query expression body> immediately contains UNION or EXCEPT, then:

i) Let *C* be the <column name> of the *i*-th column of *T1*. If the <column name> of the *i*-th column of *T2* is *C*, then the <column name> of the *i*-th column of *TR* is *C*; otherwise, the <column name> of the *i*-th column of *TR* is implementation-dependent (UV089).

ii) Case:

1) If *TR* is not the result of an anchor expression, then the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the set comprising the declared type of the *i*-th column of *T1* and the declared type of the *i*-th column of *T2* as *DTSET*; let the declared type of the *i*-th column of *TR* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

A) If the SQL-implementation does not support Feature T101, "Enhanced nullability determination", then the *i*-th column of *TR* is possibly nullable.

B) If the <query expression body> immediately contains EXCEPT, then if the *i*-th column of *T1* is known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

C) Otherwise, if the *i*-th columns of both *T1* and *T2* are known not nullable, then the *i*-th column of *TR* is known not nullable; otherwise, the *i*-th column of *TR* is possibly nullable.

2) If *TR* is the result of an anchor expression *ARE*, then:

A) Let *SARE* be the stratum of *ARE*. Let *TSARE* be the the tables associated with the members of *SARE*. Of the operands *T1* and *T2* of *TR*, let *TNREC* be the operand that is the result of the non-recursive operand of *ARE* and let *TREC* be the other operand. The *i*-th column of *TR* is said to be *recursively derived* if there exists at least one table *TTSARE* in *TSARE* such that a column of *TTSARE* is an underlying column of the *i*-th column of *TREC*.

B) If the *i*-th column of *TR* is not recursively derived, then the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the set comprising the declared type of the *i*-th column of *T1* and the declared type of the *i*-th column of *T2* as *DTSET*; let the declared type of the *i*-th column of *TR* be the *RESTYPE* returned from the application of those Syntax Rules. If the *i*-th columns of at least one of *T1* and *T2* are known not nullable, then the *i*-th column of *TR* is *known not nullable*; otherwise, the *i*-th column of *TR* is *possibly nullable*.

C) If the *i*-th column of *TR* is recursively derived, then:

I) The *i*-th column of *TR* is *possibly nullable*.

II) Case:

1) If *T1* is *TNREC*, then the declared type of the *i*-th column of *TR* is the same as the declared type of the *i*-th column of *T1*.

2) If *T2* is *TNREC*, then the declared type of the *i*-th column of *TR* is the same as the declared type of the *i*-th column of *T2*.

20) The *simply underlying table specification* of *QE* is the <query expression body> immediately contained in *QE*.

21) After performing all relevant syntactic transformations, the *simply underlying table specifications* of a <query expression body> *QEB* are the <query specification>s and <query expression>s contained in *QEB* without an intervening <query expression>.

22) If *QE* or *QEB* is effectively updatable, then the *target simply underlying table specifications* of *QE* or *QEB* are the simply underlying table specifications of *QE* or *QEB*, respectively.

23) The *underlying columns* and *generally underlying columns* of each column of *QE* and of *QE* itself are defined as follows:

a) A column of a <table value constructor> has no underlying columns and no generally underlying columns.

b) The underlying columns and generally underlying columns of every *i*-th column of a <simple table> *ST* are the underlying columns and generally underlying columns, respectively, of the *i*-th column of the table immediately contained in *ST*.

c) If no set operator is specified, then the underlying columns and generally underlying columns of every *i*-th column of *QE* are the underlying columns and generally underlying columns, respectively, of the *i*-th column of the <simple table> simply contained in *QE*.

d) If a set operator is specified, then the underlying columns and generally underlying columns of every *i*-th column of *QE* are the underlying columns and generally underlying columns, respectively, of the *i*-th column of *T1* and those of the *i*-th column of *T2*.

e) Let *C* be some column. *C* is an underlying column of *QE* if and only if *C* is an underlying column of some column of *QE*. *C* is a generally underlying column of *QE* if and only if *C* is a generally underlying column of some column of *QE*.

24) The *updatable columns* of *QE* are defined as follows:

a) A column of a <table value constructor> is not an updatable column.

b) A column of a <simple table> *ST* is an *updatable column* of *ST* if the underlying column of *ST* is updatable.

c) If no set operator is specified, then a column of *QE* is an *updatable column* of *QE* if its underlying column is updatable.

d) If a set operator is specified, then

Case:

i) If the SQL-implementation supports Feature T111, "Updatable joins, unions, and columns", a set operator UNION ALL is specified, and both underlying columns of the *i*-th column of *QE* are updatable, then the *i*-th column of *QE* is an updatable column of *QE*.

ii) Otherwise, the *i*-th column of *QE* is not updatable.

NOTE 318 — If a set operator UNION DISTINCT, EXCEPT, or INTERSECT is specified, or if the SQL-implementation does not support Feature T111, "Updatable joins, unions, and columns", then there are no updatable columns.

25) A <query expression> *QE* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

26) If an <order by clause> is specified, then:

a) Let *OBC* be the <order by clause>. Let *NSK* be the number of <sort specification>s in *OBC*. For each *i* between 1 (one) and *NSK*, let K_i be the <sort key> contained in the *i*-th <sort specification> in *OBC*.

b) Each K_i shall contain a <column reference> and shall not contain a <query expression> or a <set function specification>.

c) If *QE* is a <query expression body> that is a <query term> that is a <query primary> that is a <simple table> that is a <query specification>, then *QE* is said to be a *simple table query*.

d) Case:

i) If <sort specification list> contains any <sort key> K_i that contains a column reference to a column that is not a column of *T*, then:

1) *QE* shall be a simple table query.

2) Let *TE* be the <table expression> immediately contained in the <query specification> *QS* simply contained in *QE*.

3) Let *SL* be the <select list> of *QS*. Let *SLT* be obtained from *SL* by replacing each <column reference> with its fully qualified equivalent.

4) Let *OBCT* be obtained from *OBC* by replacing each <column reference> that references a column of *TE* with its fully qualified equivalent.

5) For each *i* between 1 (one) and *NSK*, let KT_i be the <sort key> contained in the *i*-th <sort specification> contained in *OBCT*.

6) For each *i* between 1 (one) and *NSK*, if KT_i has the same left normal form derivation as the <value expression> immediately contained in some <derived column> *DC* of *SLT*, then:

NOTE 319 — "Left normal form derivation" is defined in Subclause 6.2, "Notation provided in the ISO/IEC 9075 series", in ISO/IEC 9075-1.

A) Case:

I) If *DC* simply contains an <as clause>, then let *CN* be the <column name> contained in the <as clause>.

II) Otherwise, let *CN* be an implementation-dependent (UV087) <column name> that is not equivalent to the explicit or implicit <column name> of any other <derived column> contained in *SLT*. Let *VE* be the <value expression> simply contained in *DC*. *DC* is replaced in *SLT* by

VE AS *CN*

B) KT_i is replaced in *OBCT* by

CN

7) Let *SCR* be the set of <column reference>s to columns of *TE* that remain in *OBCT* after the preceding transformation.

8) Let *NSCR* be the number of <column reference>s contained in *SCR*. For each *j* between 1 (one) and *NSCR*, let C_j be an enumeration of these <column reference>s.

9) Case:

A) If *NSCR* is 0 (zero), then let *SKL* be the zero-length character string.

B) Otherwise:

I) If *T* is a grouped table, then let *G* be the set of grouping columns of *T*. For each *j* between 1 (one) and *NSCR*, C_j shall reference a column that is functionally dependent on *G*.

II) *QS* shall not specify the <set quantifier> DISTINCT or directly contain one or more <set function specification>s.

III) Let *SKL* be the comma-separated list of <derived column>s:

$, C_1, C_2, \ldots, C_{NSCR}$

The columns C_j are said to be *extended sort key columns*.

10) Let *SORT* be the table specified by the <query specification>:

SELECT *SLT SKL TE*

11) Let *EOBC* be *OBCT*.

ii) Otherwise, let *SORT* be *T* and let *EOBC* be *OBC*.

e) *SORT* is said to be the *sort table* of *QE*. *EOBC* is said to be the *extended order by clause* of *SORT*.

27) The declared type of <offset row count> shall be an exact numeric with scale 0 (zero).

28) The declared type of <fetch first row count> shall be an exact numeric with scale 0 (zero).

29) The declared type of the <simple value specification> simply contained in <fetch first percentage> shall be numeric.

30) [14] If a <query expression> simply contains a <fetch first clause> that simply contains WITH TIES, then the <query expression> shall simply contain an <order by clause>. If a <query primary> simply contains a <fetch first clause> that simply contains WITH TIES, then the <query primary> shall simply contain an <order by clause>.

Access Rules

None.

General Rules

1) If a non-recursive <with clause> is specified, then:

   a) If any of the <with list element>s generally contains a <data change delta table>, the <with list element>s simply contained in the <query expression> are evaluated in the order in which they are given.

   b) For every <with list element> *WLE*, let *WQN* be the <query name> immediately contained in *WLE*. Let *WQE* be the <query expression> simply contained in *WLE*. Let *WLT* be the table resulting from evaluation of *WQE*, with each column name replaced by the corresponding element of the <with column list>, if any, immediately contained in *WLE*.

   c) Every <table reference> contained in <query expression> that specifies *WQN* identifies *WLT*.

2) If a recursive <with list> *WL* is specified, then:

   a) Let *s* be the number of strata of *QDNG*. Let the *partition dependency graph PDG* of *WL* be a directed graph such that:

      i) Each stratum of *QDNG* is a node of *PDG*.

      ii) There is an arc of *PDG* from one stratum *S1* to a different stratum *S2* if there is an arc of *QNDG* from a <query name> in *S1* to a <query name> in *S2*.

   b) Let S_1, ..., S_s be an enumeration of the strata of *PDG* such that, for all *i* and *j* between 1 (one) and *s*, if there is an arc from S_i to S_j, then $i > j$.

      NOTE 320 — There can be no cycles among the strata, because if there were a cycle, then the union of strata in that cycle would be a larger strongly connected subset of *QNDG*, contradicting the definition of a stratum. Hence such an enumeration is possible. One can first list all strata that have no outgoing arcs, thereafter listing a stratum only if all strata upon which it depends have already been listed.

   c) For all *j* between 1 (one) and *s*:

      NOTE 321 — The order of S_1, ..., S_s is non-deterministic; however, if all <query expression>s in the <with list> are deterministic, then the final result will be deterministic because the ordering of S_1, ..., S_s ensures that <query expression>s are evaluated in the order of dependency.

      Case:

      i) If S_j is not recursive, then let *SQN* be the <query name> that is the sole member of S_j. Let *SWLE* be the <with list element> that immediately contains *QN*, and let *SQE* be the <query expression> simply contained in *SWLE*. Any reference to *SQN* in its scope references the result of evaluating *SQE*.

      ii) Otherwise:

         1) Let *NQ* be the number of <query name>s in S_j. Let SQN_1, ..., SQN_{NQ} be an enumeration of the <query name>s of S_j. For all *i*, 1 (one) $\leq i \leq NQ$, let $SWLE_i$ be the <with list element> that immediately contains SQN_i, let SQE_i be the <query expression> simply contained in $SWLE_i$, let SRT_i be the row type of SQE_i, and let WT_i be an empty table of type SRT_i.

2) For all i, 1 (one) ≤ i ≤ NQ, let SQN_k reference WT_i in the <query expression>s SQE_k, 1 (one) ≤ k ≤ NQ.

3) For all i, 1 (one) ≤ i ≤ NQ, let RT_i be the result of evaluating SQE_i.

4) While there exists an i, 1 (one) ≤ i ≤ NQ, such that the cardinality of RT_i is greater than the cardinality of WT_i, do:

NOTE 322 — The iteration stops if a finite fixed point is reached. It is possible that there is no finite fixed point, in which case the following loop does not terminate. Nevertheless, if the SQL-implementation can determine that all rows required in the remainder of the query have been found, then the SQL-implementation will terminate the iteration.

A) For all i, 1 (one) ≤ i ≤ NQ, let WT_i be RT_i.

B) For all i, 1 (one) ≤ i ≤ NQ, let RT_i be the result of evaluating SQE_i.

NOTE 323 — The <query name>s SQN_i are still bound to WT_i. Thus each RT_i is recomputed by binding the <query name>s to the results in the previous iteration of the loop.

5) For all i, 1 (one) ≤ i ≤ NQ, SQN_i references RT_i in the scope of SQN_i, excluding SQE_k for all k, 1 (one) ≤ k ≤ NQ.

3) Case:

a) If no set operator is specified, then T is the result of the specified <simple table>.

b) Otherwise:

i) Let D be the degree of T.

ii) For each i, 1 (one) ≤ i ≤ D:

1) [14] Let DTC_i be the declared type of the i-th column of T.

2) Let $TCN1_i$ be the effective name for the i-th column of $T1$.

3) Let $TCN2_i$ be the effective name for the i-th column of $T2$.

4) Let $ET1$ be a <query expression> of the form

```
SELECT CAST(TCN1_1 AS DTC_1),
       CAST(TCN1_2 AS DTC_2),
       ...,
       CAST(TCN1_D AS DTC_D)
FROM TN1
```

5) Let $ET2$ be a <query expression> of the form

```
SELECT CAST(TCN2_1 AS DTC_1),
       CAST(TCN2_2 AS DTC_2),
       ...,
       CAST(TCN2_D AS DTC_D)
FROM TN2
```

iii) T contains the following rows:

1) Let R be a row that is a duplicate of some row in $ET1$ or of some row in $ET2$ or both. Let m be the number of duplicates of R in $ET1$ and let n be the number of duplicates of R in $ET2$, where $m \geq 0$ and $n \geq 0$.

2) If DISTINCT is specified or implicit, then

Case:

A) If UNION is specified, then *T* contains exactly one duplicate of *R*.

NOTE 324 — *R* exists as a result of GR 3)b)iii)1), guaranteeing that *T* always contains a duplicate of *R*.

B) If EXCEPT is specified, then

Case:

I) If $m > 0$ and $n = 0$, then *T* contains exactly one duplicate of *R*.

II) Otherwise, *T* contains no duplicate of *R*.

C) If INTERSECT is specified, then

Case:

I) If $m > 0$ and $n > 0$, then *T* contains exactly one duplicate of *R*.

II) Otherwise, *T* contains no duplicates of *R*.

3) If ALL is specified, then

Case:

A) If UNION is specified, then the number of duplicates of *R* that *T* contains is $(m + n)$.

B) If EXCEPT is specified, then the number of duplicates of *R* that *T* contains is the maximum of $(m - n)$ and 0 (zero).

C) If INTERSECT is specified, then the number of duplicates of *R* that *T* contains is the minimum of *m* and *n*.

NOTE 325 — See the General Rules of Subclause 8.2, "<comparison predicate>".

4) Case:

a) If EXCEPT is specified and a row *R* of *T* is replaced by some row *RR*, then the row of *T1* from which *R* is derived is replaced by *RR*.

b) If INTERSECT is specified, then:

i) If a row *R* is inserted into *T*, then:

1) If *T1* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T1*.

2) If *T1* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T1*.

3) If *T2* does not contain a row whose value equals the value of *R*, then *R* is inserted into *T2*.

4) If *T2* contains a row whose value equals the value of *R* and no row of *T* is derived from that row, then *R* is inserted into *T2*.

ii) If a row *R* is replaced by some row *RR*, then:

1) The row of *T1* from which *R* is derived is replaced with *RR*.

2) The row of *T2* from which *R* is derived is replaced with *RR*.

5) The rows of *T* are ordered as follows:

a) If *QE* does not immediately contain an <order by clause>, then the ordering of rows in *T* is implementation-dependent (US036).

b) If *QE* immediately contains an <order by clause>, then the ordering of rows in *T* is the same as the ordering of rows in the sort table of *QE* and its extended order by clause as determined by the General Rules of Subclause 10.10, "<sort specification list>". The table specified by *QE* is effectively the sort table of *QE* with all extended sort key columns, if any, removed.

NOTE 326 — "extended sort key column" and "extended order by clause" are defined in the Syntax Rules of this Subclause.

6) Let *OCT* be the cardinality of *T*.

a) Case:

i) If <result offset clause> is not specified, then let *RORC* be 0 (zero).

ii) Otherwise, let *RORC* be the value of <offset row count>.

b) If *RORC* is less than 0 (zero), then an exception condition is raised: *data exception — invalid row count in result offset clause (2201X)*.

c) Case:

i) If *RORC* is greater than or equal to *OCT*, then all rows are removed from *T* and the cardinality of *T* is 0 (zero).

ii) Otherwise, the first *RORC* rows in order as specified by GR 5) are removed from *T* and the cardinality of *T* is (*OCT* – *RORC*).

7) If <fetch first clause> is specified, then:

a) Let *OCT2* be the cardinality of *T*.

NOTE 327 — *OCT2* is the cardinality of *T* after the removal of any rows from *T* by the application of the prior General Rule.

b) Case:

i) If <fetch first row count> is specified, then let *FFRC* be the value of <fetch first row count>.

ii) If <fetch first percentage> is specified, then let *FFP* be the <simple value specification> simply contained in <fetch first percentage>, and let *LOCT* be a <literal> whose value is *OCT*. Let *FFRC* be the value of

```
CEILING ( FFP * LOCT / 100.0E0 )
```

NOTE 328 — The percentage is computed using the number of rows before removing the rows specified by <offset row count>.

iii) Otherwise, let *FFRC* be 1 (one).

c) If *FFRC* is less than 1 (one), then an exception condition is raised: *data exception — invalid row count in fetch first clause (2201W)*.

d) If *FFRC* is less than *OCT2*, then

Case:

i) If WITH TIES is specified, then rows other than the first *FFRC* rows in the order specified by GR 6) of this Subclause and their peers as defined in Subclause 10.10, "<sort specification list>", are removed from *T*, and the cardinality of *T* is the number of rows remaining in *T*.

ii) Otherwise, rows other than the first *FFRC* rows in order as specified by GR 6) of this Subclause are removed from *T* and the cardinality of *T* is *FFRC*.

Conformance Rules

1) Without Feature T121, "WITH (excluding RECURSIVE) in query expression", in conforming SQL language, a <query expression> shall not contain a <with clause>.

2) Without Feature T122, "WITH (excluding RECURSIVE) in subquery", in conforming SQL language, a <query expression> contained in a <query expression> shall not contain a <with clause>.

3) Without Feature T131, "Recursive query", conforming SQL language shall not contain a <query expression> that contains RECURSIVE.

4) Without Feature T132, "Recursive query in subquery", in conforming SQL language, a <query expression> contained in a <query expression> shall not contain RECURSIVE.

5) Without Feature F661, "Simple tables", conforming SQL language shall not contain a <simple table> that immediately contains a <table value constructor>.

6) Without Feature F661, "Simple tables", conforming SQL language shall not contain an <explicit table>.

7) Without Feature F303, "INTERSECT DISTINCT table operator", conforming SQL language shall not contain a <query term> that contains INTERSECT.

8) Without Feature F305, "INTERSECT ALL table operator", conforming SQL language shall not contain a <query term> that contains INTERSECT ALL.

9) Without Feature F301, "CORRESPONDING in query expressions", conforming SQL language shall not contain a <query expression> that contains CORRESPONDING.

10) Without Feature T551, "Optional key words for default syntax", conforming SQL language shall not contain UNION DISTINCT, EXCEPT DISTINCT, or INTERSECT DISTINCT.

11) Without Feature F304, "EXCEPT ALL table operator", conforming SQL language shall not contain a <query expression> that contains EXCEPT ALL.

NOTE 329 — If DISTINCT, INTERSECT or EXCEPT is specified, then the Conformance Rules of Subclause 9.12, "Grouping operations", apply.

12) Without Feature F850, "Top-level ORDER BY in query expression", in conforming SQL language, a <query expression> not immediately contained in either an <array value constructor by query> or a <cursor specification> shall not immediately contain an <order by clause>.

13) Without Feature F851, "ORDER BY in subqueries", in conforming SQL language, a <query expression> contained in another <query expression> shall not immediately contain an <order by clause>.

14) Without Feature F855, "Nested ORDER BY in query expression", in conforming SQL language, a <query primary> shall not immediately contain an <order by clause>.

15) Without Feature F856, "Nested FETCH FIRST in query expression", in conforming SQL language, a <query primary> shall not immediately contain a <fetch first clause>.

16) Without Feature F857, "Top-level FETCH FIRST in query expression", in conforming SQL language, a <query expression> shall not immediately contain a <fetch first clause>.

17) Without Feature F858, "FETCH FIRST in subqueries", in conforming SQL language, a <query expression> contained in another <query expression> shall not immediately contain a <fetch first clause>.

18) Without Feature F860, “Dynamic FETCH FIRST row count”, in conforming SQL language, a <fetch first clause> shall not contain a <fetch first row count> that is not an <unsigned integer>.

19) Without Feature F861, “Top-level OFFSET in query expression”, in conforming SQL language, a <query expression> shall not immediately contain a <result offset clause>.

20) Without Feature F862, “OFFSET in subqueries”, in conforming SQL language, a <query expression> contained in another <query expression> shall not immediately contain a <result offset clause>.

21) Without Feature F863, “Nested OFFSET in query expression”, in conforming SQL language, a <query primary> shall not immediately contain a <result offset clause>.

22) Without Feature F865, “Dynamic offset row count in OFFSET”, in conforming SQL language, a <result offset clause> shall not contain an <offset row count> that is not an <unsigned integer>.

23) Without Feature F866, “FETCH FIRST clause: PERCENT option”, in conforming SQL language, <fetch first clause> shall not contain <fetch first percentage>.

24) Without Feature F867, “FETCH FIRST clause: WITH TIES option”, in conforming SQL language, <fetch first clause> shall not contain WITH TIES.

25) [14] Without Feature F868, “ORDER BY in grouped table”, in conforming SQL language, if <sort specification list> contains any <sort key> that contains a column reference to a column that is not a column of the table *TQE* specified by <query expression>, then *TQE* shall not be a grouped table.

7.18 <search or cycle clause>

Function

Specify the generation of ordering and cycle detection information in the result of recursive query expressions.

Format

```
<search or cycle clause> ::=
    <search clause>
  | <cycle clause>
  | <search clause> <cycle clause>

<search clause> ::=
  SEARCH <recursive search order> SET <sequence column>

<recursive search order> ::=
    DEPTH FIRST BY <column name list>
  | BREADTH FIRST BY <column name list>

<sequence column> ::=
  <column name>

<cycle clause> ::=
  CYCLE <cycle column list> SET <cycle mark column> [ <cycle mark option> ]
      USING <path column>

<cycle column list> ::=
  <cycle column> [ { <comma> <cycle column> }... ]

<cycle column> ::=
  <column name>

<cycle mark column> ::=
  <column name>

<path column> ::=
  <column name>

<cycle mark option> ::=
  TO <cycle mark value> DEFAULT <non-cycle mark value>

<cycle mark value> ::=
  <value expression>

<non-cycle mark value> ::=
  <value expression>
```


Syntax Rules

1) If <cycle mark option> is not specified, then TO TRUE DEFAULT FALSE is implicit.

2) Let *WLEC* be an expandable <with list element> immediately containing a <search or cycle clause>.

3) Let *WQN* be the <query name>, *WCL* the <with column list>, and *WQE* the <query expression> simply contained in *WLEC*. Let *WQEB* be the <query expression body> immediately contained in *WQE*. Let *OP* be the set operator immediately contained in *WQEB*. Let *TLO* be the <query expression

body> that constitutes the first operand of *OP* and let *TRO* be the <query specification> that (necessarily) constitutes the second operand of *OP*.

4) Let *TROSL* be the <select list> immediately contained in *TRO*. Let *WQNTR* be the <table reference> simply contained in the <from clause> immediately contained in the <table expression> *TROTE* immediately contained in *TRO* such that *WQNTR* immediately contains *WQN*.

Case:

a) If *WQNTR* simply contains a <correlation name>, then let *WQNCRN* be that <correlation name>.

b) Otherwise, let *WQNCRN* be *WQN*.

5) If *WLEC* simply contains a <search clause> *SC*, then let *SQC* be the <sequence column> and *SO* be the <recursive search order> immediately contained in *SC*. Let *CNL* be the <column name list> immediately contained in *SO*.

a) *WCL* shall not contain a <column name> that is equivalent to *SQC*.

b) Every <column name> of *CNL* shall be equivalent to some <column name> contained in *WCL*. No <column name> shall be contained more than once in *CNL*.

c) Case:

i) If *SO* immediately contains DEPTH, then let *SCEX1* be:

```
WQNCRN.SQC
```

let *SCEX2* be:

```
SQC || ARRAY [ROW(CNL)]
```

and let *SCIN* be:

```
ARRAY [ROW(CNL)]
```

ii) If *SO* immediately contains BREADTH, then let *SCEX1* be:

```
( SELECT OC.*
  FROM ( VALUES (WQNCRN.SQC) )
  OC(LEVEL, CNL) )
```

let SCEX2 be:

```
ROW(SQC.LEVEL + 1, CNL)
```

and let *SCIN* be:

```
ROW(0, CNL)
```

6) If *WLEC* simply contains a <cycle clause> *CC*, then let *CCL* be the <cycle column list>, let *CMC* be the <cycle mark column>, let *CMX* be the explicit or implicit <cycle mark option>, and let *CPA* be the <path column> immediately contained in *CC*. Let *CMV* be the <cycle mark value> and let *CMD* be the <non-cycle mark value> immediately contained in *CMX*.

a) Every <column name> of *CCL* shall be equivalent to some <column name> contained in *WCL*. No <column name> shall be contained more than once in *CCL*.

b) *CMC* and *CPA* shall not be equivalent to each other and not equivalent to any <column name> of *WCL*.

c) The declared type of *CMV* and *CMD* shall be either character string of length 1 (one) or Boolean. *CMV* and *CMD* shall have the same declared type. *CMV* and *CMD* shall be literals and *CMV* shall not be equal to *CMD*.

d) Let *CCEX1* be:

```
WQNCRN.CMC, WQNCRN.CPA
```

Let *CCEX2* be:

```
CASE WHEN ROW(CCL) IN
  (SELECT P.* FROM UNNEST(CPA) P)

    THEN CMV ELSE CMD END,

CPA || ARRAY [ROW(CCL)]
```

Let *CCIN* be:

```
CMD, ARRAY [ROW(CCL)]
```

Let *NCCON1* be:

```
CMC <> CMV
```

7) Case:

a) If *WLEC* simply contains a <search clause> and does not simply contain a <cycle clause>, then let *EWCL* be:

```
WCL, SQC
```

Let *ETLOSL* be:

```
WCL, SCIN
```

Let *ETROSL* be:

```
WCL, SCEX2
```

Let *ETROSL1* be:

```
TROSL, SCEX1
```

Let *NCCON* be:

```
TRUE
```

b) If *WLEC* simply contains a <cycle clause> and does not simply contain a <search clause>, then let *EWCL* be:

```
WCL, CMC, CPA
```

Let *ETLOSL* be:

```
WCL, CCIN
```

Let *ETROSL* be:

```
WCL, CCEX2
```

Let *ETROSL1* be:

```
TROSL, CCEX1
```

Let *NCCON* be:

```
NCCON1
```

c) If *WLEC* simply contains both a <search clause> and a <cycle clause> *CC*, then:

i) The <column name>s *SQC*, *CMC*, and *CPA* shall not be equivalent to each other.

ii) Let *EWCL* be:

```
WCL, SQC, CMC, CPA
```

Let *ETLOSL* be:

```
WCL, SCIN, CCIN
```

Let *ETROSL* be:

```
WCL, SCEX2, CCEX2
```

Let *ETROSL1* be:

```
TROSL, SCEX1, CCEX1
```

iii) Let *NCCON* be:

```
NCCON1
```

8) *WLEC* is equivalent to the expanded <with list element>:

```
WQN(EWCL) AS
  ( SELECT ETLOSL FROM (TLO) TLOCRN(WCL)
    OP
    SELECT ETROSL
    FROM (SELECT ETROSL1 TROTE) TROCRN(EWCL)
    WHERE NCCON
  )
```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T133, "Enhanced cycle mark values", the declared types of <cycle mark value> and <non-cycle mark value> shall not be Boolean.

7.19 <subquery>

Function

Specify a scalar value, a row, or a table derived from a <query expression>.

Format

```
<scalar subquery> ::=
  <subquery>

<row subquery> ::=
  <subquery>

<table subquery> ::=
  <subquery>

<subquery> ::=
  <left paren> <query expression> <right paren>
```

Syntax Rules

1) The <subquery> immediately contained in a <table subquery> not simply contained in a <with list element>, or immediately contained in a <scalar subquery>, or immediately contained in a <row subquery> shall not generally contain a <data change delta table>.

2) The degree of a <scalar subquery> shall be 1 (one).

3) The degree of a <row subquery> shall be greater than 1 (one).

4) Let *QE* be the <query expression> simply contained in <subquery>.

5) The declared type of a <scalar subquery> is the declared type of the column of *QE*.

6) The declared type of a <row subquery> is a row type consisting of one field for each column of *QE*. The declared type and field name of each field of this row type is the declared type and column name of the corresponding column of *QE*.

7) The declared types of the columns of a <table subquery> are the declared types of the respective columns of *QE*.

Access Rules

None.

General Rules

1) If no SQL-transaction is active for the SQL-agent, then an SQL-transaction is initiated.

2) Let *RS* be a <row subquery>. Let *RRS* be the result of the <query expression> simply contained in *RS*. Let *D* be the degree of *RRS*.

   Case:

   a) If the cardinality of *RRS* is greater than 1 (one), then an exception condition is raised: *cardinality violation (21000)*.

b) If the cardinality of *RRS* is 0 (zero), then the value of the <row subquery> is a row whose degree is *D* and whose fields are all the null value.

c) Otherwise, the value of *RS* is *RRS*.

3) Let *SS* be a <scalar subquery>. Let *RSS* be the result of the <query expression> simply contained in *SS*.

Case:

a) If the cardinality of *RSS* is greater than 1 (one), then an exception condition is raised: *cardinality violation (21000)*.

b) If the cardinality of *RSS* is 0 (zero), then the value of the <scalar subquery> is the null value.

c) Otherwise, let *C* be the column of <query expression> simply contained in *SS*. The value of *SS* is the value of *C* in the unique row of the result of the <scalar subquery>.

Conformance Rules

None.

8 Predicates

This Clause is modified by Clause 7, "Predicates", in ISO/IEC 9075-13.
This Clause is modified by Clause 8, "Predicates", in ISO/IEC 9075-14.
This Clause is modified by Clause 8, "Predicates", in ISO/IEC 9075-15.
This Clause is modified by Clause 8, "Predicates", in ISO/IEC 9075-16.

8.1 <predicate>

This Subclause is modified by Subclause 8.1, "<predicate>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 8.1, "<predicate>", in ISO/IEC 9075-16.

Function

Specify a condition that can be evaluated to give a Boolean value.

Format

```
14 | 16 <predicate> ::=
      <comparison predicate>
    | <between predicate>
    | <in predicate>
    | <like predicate>
    | <similar predicate>
    | <regex like predicate>
    | <null predicate>
    | <quantified comparison predicate>
    | <exists predicate>
    | <unique predicate>
    | <normalized predicate>
    | <match predicate>
    | <overlaps predicate>
    | <distinct predicate>
    | <member predicate>
    | <submultiset predicate>
    | <set predicate>
    | <type predicate>
    | <period predicate>
    | <JSON predicate>
    | <JSON exists predicate>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) 14 16 The result of a <predicate> is the truth value of the immediately contained <comparison predicate>, <between predicate>, <in predicate>, <like predicate>, <similar predicate>, <regex like predicate>, <null predicate>, <quantified comparison predicate>, <exists predicate>, <unique predicate>, <normalized predicate>, <match predicate>, <overlaps predicate>, <distinct predicate>, <member predicate>, <submultiset predicate>, <set predicate>, <type predicate>, <period predicate>, <JSON predicate>, or <JSON exists predicate>.

Conformance Rules

None.

8.2 <comparison predicate>

This Subclause is modified by Subclause 7.1, "<comparison predicate>", in ISO/IEC 9075-13.

Function

Specify a comparison of two row values.

Format

```
<comparison predicate> ::=
  <row value predicand> <comparison predicate part 2>

<comparison predicate part 2> ::=
  <comp op> <row value predicand>

<comp op> ::=
    <equals operator>
  | <not equals operator>
  | <less than operator>
  | <greater than operator>
  | <less than or equals operator>
  | <greater than or equals operator>
```

Syntax Rules

1) The two <row value predicand>s shall be of the same degree.

2) Let *corresponding fields* be fields with the same ordinal position in the two <row value predicand>s.

3) The declared types of the corresponding fields of the two <row value predicand>s shall be comparable.

4) Let R_x and R_y respectively denote the first and second <row value predicand>s.

5) Let N be the number of fields in the declared type of R_x. Let X_i, 1 (one) $\leq i \leq N$, be the i-th field in the declared type of R_x and let Y_i be the i-th field in the declared type of R_y. For each i:

   a) Case:

      i) If <comp op> is <equals operator> or <not equals operator>, then X_i and Y_i are operands of an equality operation. The Syntax Rules and Conformance Rules of Subclause 9.11, "Equality operations", apply.

      ii) Otherwise, X_i and Y_i are operands of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 9.14, "Ordering operations", apply.

   b) Case:

      i) If the declared types of X_i and Y_i are user-defined types, then let *UDT1* and *UDT2* be respectively the declared types of X_i and Y_i. *UDT1* and *UDT2* shall be in the same subtype family. *UDT1* and *UDT2* shall have comparison types.

         NOTE 330 — "Comparison type" is defined in Subclause 4.9.5, "User-defined type comparison and assignment".

         13 NOTE 331 — The comparison form and comparison categories included in the user-defined type descriptors of both *UDT1* and *UDT2* are constrained to be the same and to be the same as those of all

their supertypes. If the comparison category is either STATE or RELATIVE, then *UDT1* and *UDT2* are constrained to have the same comparison function; if the comparison category is MAP, they are not constrained to have the same comparison function.

ii) If the declared types of X_i and Y_i are reference types, then the referenced type of the declared type of X_i and the referenced type of the declared type of Y_i shall have a common supertype.

iii) If the declared types of X_i and Y_i are collection types or distinct types whose source types are collection types in which the declared type of the elements are ET_x and ET_y, respectively, then let *RV1* and *RV2* be <value expression>s whose declared types are respectively ET_x and ET_y. The Syntax Rules of this Subclause are applied to:

```
RV1 <comp op> RV2
```

iv) If the declared types of X_i and Y_i are row types, then let *RV1* and *RV2* be <value expression>s whose declared types are respectively that of X_i and Y_i. The Syntax Rules of this Subclause are applied to:

```
RV1 <comp op> RV2
```

6) Let *CP* be the <comparison predicate> "R_x <comp op> R_y". If the declared type of R_x is a user-defined type whose comparison category is RELATIVE, then it is implementation-defined (IA059) whether the following syntactic transformations are applied; otherwise, the following syntactic transformations are applied.

Case:

a) If the <comp op> is <not equals operator>, then *CP* is equivalent to:

```
NOT( Rx = Ry )
```

b) If the <comp op> is <greater than operator>, then *CP* is equivalent to:

```
( Ry < Rx )
```

c) If the <comp op> is <less than or equals operator>, then *CP* is equivalent to:

```
( Rx < Ry
    OR
  Ry = Rx )
```

d) If the <comp op> is <greater than or equals operator>, then *CP* is equivalent to:

```
( Ry < Rx
    OR
  Ry = Rx )
```

Access Rules

None.

General Rules

1) Let *XV* and *YV* be two values represented by <value expression>s *X* and *Y*, respectively. The result of:

X <comp op> *Y*

is determined as follows.

Case:

a) If at least one of *XV* and *YV* is the null value, then

X <comp op> *Y*

is *Unknown*.

b) Otherwise,

Case:

i) If the declared types of *XV* and *YV* are row types with degree *N*, then let X_i, 1 (one) ≤ *i* ≤ *N*, denote a <value expression> whose value and declared type is that of the *i*-th field of *XV* and let Y_i denote a <value expression> whose value and declared type is that of the *i*-th field of *YV*. The result of

X <comp op> *Y*

is determined as follows:

1) *X* = *Y* is *True* if and only if $X_i = Y_i$ is *True* for all *i*.

2) *X* < *Y* is *True* if and only if $X_i = Y_i$ is *True* for all $i < n$ and $X_n < Y_n$ for some *n*.

3) *X* = *Y* is *False* if and only if NOT ($X_i = Y_i$) is *True* for some *i*.

4) *X* < *Y* is *False* if and only if *X* = *Y* is *True* or *Y* < *X* is *True*.

5) *X* <comp op> *Y* is *Unknown* if *X* <comp op> *Y* is neither *True* nor *False*.

ii) If the declared types of *XV* and *YV* are array types or distinct types whose source types are array types and the cardinalities of *XV* and *YV* are *N1* and *N2*, respectively, then let X_i, 1 (one) ≤ *i* ≤ *N1*, denote a <value expression> whose value and declared type is that of the *i*-th element of *XV* and let Y_i, 1 (one) ≤ *i* ≤ *N2*, denote a <value expression> whose value and declared type is that of the *i*-th element of *YV*. The result of

X <comp op> *Y*

is determined as follows:

1) *X* = *Y* is *True* if *N1* = 0 (zero) and *N2* = 0 (zero).

2) *X* = *Y* is *True* if *N1* = *N2* and, for all *i*, $X_i = Y_i$ is *True*.

3) *X* = *Y* is *False* if and only if exactly one of the following is true:

A) *N1* ≠ *N2*.

B) *N1* = *N2* and for some *i*, 1 (one) ≤ *i* ≤ *N1*, NOT ($X_i = Y_i$) is *True*.

4) *X* <comp op> *Y* is *Unknown* if *X* <comp op> *Y* is neither *True* nor *False*.

iii) If the declared types of *XV* and *YV* are multiset types or distinct types whose source types are multiset types and the cardinalities of *XV* and *YV* are *N1* and *N2*, respectively, then the result of

X <comp op> *Y*

is determined as follows.

Case:

1) *X* = *Y* is *True* if *N1* = 0 (zero) and *N2* = 0 (zero).

2) *X* = *Y* is *True* if *N1* = *N2*, and there exist an enumeration XVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *XV* and an enumeration YVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *YV* such that for all *j*, $XVE_j = YVE_j$.

3) *X* = *Y* is *Unknown* if *N1* = *N2*, and there exist an enumeration XVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *XV* and an enumeration YVE_j, 1 (one) ≤ *j* ≤ *N1*, of the elements of *YV* such that for all *j*, "$XVE_j = YVE_j$" is either *True* or *Unknown*.

4) Otherwise, *X* = *Y* is *False*.

iv) If the declared types of *XV* and *YV* are user-defined types, then let UDT_x and UDT_y be respectively the declared types of *XV* and *YV*. The result of

```
X <comp op> Y
```

is determined as follows:

1) If the comparison category of UDT_x is MAP, then let *HF1* be the <routine name> with explicit <schema name> of the comparison function of UDT_x and let *HF2* be the <routine name> with explicit <schema name> of the comparison function of UDT_y. If *HF1* identifies an SQL-invoked method, then let *HFX* be *X.HF1*; otherwise, let *HFX* be *HF1*(*X*). If *HF2* identifies an SQL-invoked method, then let *HFY* be *Y.HF2*; otherwise, let *HFY* be *HF2*(*Y*).

```
X <comp op> Y
```

has the same result as

```
HFX <comp op> HFY
```

2) If the comparison category of UDT_x is RELATIVE, then:

A) Let *RF* be the <routine name> with explicit <schema name> of the comparison function of UDT_x.

B) `X = Y`

has the same result as

```
RF (X, Y) = 0
```

C) `X < Y`

has the same result as

```
RF (X, Y) = -1
```

D) `X <> Y`

has the same result as

```
RF (X, Y) <> 0
```

E) It is implementation-dependent (UA068) whether `X > Y`

has the same result as

```
RF (X, Y) = 1
```

or has the same result as

```
RF (Y, X) = -1
```

F) `X <= Y`

has the same result as

```
RF (X, Y) = -1 OR RF (X, Y) = 0
```

G) It is implementation-dependent (UA068) whether `X >= Y`

has the same result as

```
RF (X, Y) = 1 OR RF (X, Y) = 0
```

or has the same result as

```
RF (Y, X) = -1 OR RF (X, Y) = 0
```

NOTE 332 — Since it is implementation-defined (IA059) whether to use the syntactic transformation to define all comparisons in terms of = and <, for portability, the application is expected ensure that *RF* (*X*, *Y*) = -*RF* (*Y*, *X*) for all *X* and Y.

3) [13]If the comparison category of UDT_X is STATE, then:

A) Let *SF* be the <routine name> of the comparison function of UDT_X.

B) `X = Y`

has the same result as

```
SF (X, Y) = TRUE
```

v) Otherwise, the result of

```
X <comp op> Y
```

is *True* or *False* as follows:

1) `X = Y`

is *True* if and only if *XV* and *YV* are equal.

2) `X < Y`

is *True* if and only if *XV* is less than *YV*.

3) `X <comp op> Y`

is *False* if and only if

```
X <comp op> Y
```

is not *True*

2) Numbers are compared with respect to their numeric values.

3) The comparison of two character strings is determined as follows:

a) The Syntax Rules of Subclause 9.15, "Collation determination", are applied with the set of declared types of the two character strings as *TYPESET*; let *CS* be the *COLL* returned from the application of those Syntax Rules.

b) If the length in characters of *X* is not equal to the length in characters of *Y*, then the shorter string is effectively replaced, for the purposes of comparison, with a copy of itself that has been extended to the length of the longer string by concatenation on the right of one or more pad characters, where the pad character is chosen based on *CS*. If *CS* has the NO PAD characteristic, then the pad character is an implementation-dependent (UV091) character different from any character in the character set of *X* and *Y* that collates less than any string under *CS*. Otherwise, the pad character is a <space>.

c) The result of the comparison of *X* and *Y* is given by the collation *CS*.

d) Depending on the collation, two strings may compare as equal even if they are of different lengths or contain different sequences of characters. When any of the operations MAX, MIN, and DISTINCT reference a grouping column, the GREATEST and LEAST functions, and UNION, EXCEPT, and INTERSECT operators refer to character strings, the specific value selected by these operations from a set of such equal values is implementation-dependent (UV090).

4) The comparison of two binary string values, *X* and *Y*, of which at least one is a binary large object string value, is determined by comparison of their octets with the same ordinal position. Let X_i and Y_i be the values of the *i*-th octets of *X* and *Y*, respectively, and let L_x be the length in octets of *X* and let L_y be the length in octets of *Y*. *X* is equal to *Y* if and only if $L_x = L_y$ and if $X_i = Y_i$ for all *i*.

5) The comparison of two binary string values *X* and *Y*, neither of which is a binary large object string value, is determined as follows:

a) Let L_x be the length in octets of *X* and let L_y be the length in octets of *Y*. Let X_i, 1 (one) $\leq i \leq L_x$, be the value of the *i*-th octet of *X*, and let Y_i, 1 (one) $\leq i \leq L_y$, be the value of the *i*-th octet of *Y*.

b) If $L_x = L_y$ and $X_i = Y_i$, 1 (one) $\leq i \leq L_x$, then *X* is equal to *Y*.

c) If $L_x < L_y$, $X_i = Y_i$ for all $i \leq L_x$, and the right-most $L_y - L_x$ octets of *Y* are all X'00's, then it is implementation-defined (JA219) whether *X* is equal to *Y* or whether *X* is less than *Y*.

d) If $L_x < L_y$, $X_i = Y_i$ for all $i \leq L_x$, and at least one of the right-most $L_y - L_x$ octets of *Y* is not X'00', then *X* is less than *Y*.

e) If $X_j < Y_j$, for some *j*, 0 (zero) $< j \leq$ minimum(L_x, L_y), and $X_i = Y_i$ for all $i < j$, then *X* is less than *Y*.

6) The comparison of two datetimes is determined according to the interval resulting from their subtraction. Let *X* and *Y* be the two values to be compared and let *H* be the least significant <primary datetime field> of *X* and *Y*, including fractional seconds precision if the data type is time or timestamp.

a) *X* is equal to *Y* if and only if

```
( X - Y ) INTERVAL H = INTERVAL '0' H
```

is *True*.

b) *X* is less than *Y* if and only if

```
( X - Y ) INTERVAL H < INTERVAL '0' H
```

is *True*.

NOTE 333 — Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.7.2, "Datetimes".

7) The comparison of two intervals is determined by the comparison of their corresponding values after conversion to integers in some common base unit. Let *X* and *Y* be the two intervals to be compared. Let *A* TO *B* be the specified or implied interval qualifier of *X* and *C* TO *D* be the specified or implied interval qualifier of *Y*. Let *T* be the least significant <primary datetime field> of *B* and *D* and let *U* be an interval qualifier of the form *T*(*N*), where *N* is an <interval leading field precision> large enough so that significance is not lost in the CAST operation.

Let *XVE* be the <value expression>

```
CAST ( X AS INTERVAL U )
```

Let *YVE* be the <value expression>

```
CAST ( Y AS INTERVAL U )
```

a) *X* is equal to *Y* if and only if

```
CAST ( XVE AS INTEGER ) = CAST ( YVE AS INTEGER )
```

is *True*.

b) *X* is less than *Y* if and only if

```
CAST ( XVE AS INTEGER ) < CAST ( YVE AS INTEGER )
```

is *True*.

8) In comparisons of Boolean values, *True* is greater than *False*

9) The result of comparing two REF values *X* and *Y* is determined by the comparison of their octets with the same ordinal position. Let L_x be the length in octets of *X* and let L_y be the length in octets of *Y*. Let X_i and Y_i, 1 (one) ≤ i ≤ L_x, be the values of the i-th octets of *X* and *Y*, respectively. *X* is equal to *Y* if and only if $L_x = L_y$ and, for all i, $X_i = Y_i$.

10) The comparison of two SQL/JSON values *X* and *Y* is determined as follows:

a) *X* is equal to *Y* if and only if *X* is equivalent to *Y*.

NOTE 334 — Subclause 4.48.3, "SQL/JSON data model", defines when two SQL/JSON items are equivalent. A non-null SQL/JSON value is an SQL/JSON item.

b) *X* is less than *Y* if and only if exactly one of the following is true:

i) *X* is the SQL/JSON null value and *Y* is not the SQL/JSON null value.

ii) *X* is an SQL/JSON scalar and *Y* is either an SQL/JSON object or an SQL/JSON array.

iii) *X* is an SQL/JSON array and *Y* is an SQL/JSON object.

iv) If *X* is an SQL/JSON scalar that is a number, character string, Boolean, or datetime, respectively and *Y* is an SQL/JSON scalar that is not a number, character string, Boolean, or datetime, respectively, then it is implementation-defined (IA228) whether *X* < *Y* is *True*.

v) *X* and *Y* are both SQL/JSON scalars that are numbers and *X* < *Y* is *True*.

vi) *X* and *Y* are both SQL/JSON scalars that are character strings and

```
X < Y COLLATE UCS_BASIC
```

is *True*.

vii) *X* and *Y* are both SQL/JSON scalars that are Booleans and *X* < *Y* is *True*.

viii) *X* and *Y* are both SQL/JSON scalars that are datetimes and exactly one of the following is true:

1) If *X* is a time value and *Y* is not a time value, then it is implementation-defined (IA228) whether *X* < *Y* is *True*.

2) *X* and *Y* are both time values and *X* < *Y* is *True*.

3) *X* and *Y* are both date values and *X* < *Y* is *True*.

4) *X* is a date value and *Y* is a timestamp value and

```
CAST(X AS TIMESTAMP) < Y
```

is *True*.

5) *X* and *Y* are both timestamp values and *X* < *Y* is *True*.

ix) *X* and *Y* are both SQL/JSON arrays. Let *CX* be the cardinality of *X* and let *CY* be the cardinality of *Y*. Let *MC* be the lesser of *CX* and *CY*. Let X_i be the *i*-th element of *X*, 1 (one) ≤ *i* ≤ *CX*, and let Y_i be the *i*-th element of *Y*, 1 (one) ≤ *i* ≤ *CY*. Exactly one of the following is true:

1) *CX* < *CY* is *True* and $X_i = Y_i$ is *True* for 1 (one) ≤ *i* < *CX*.

2) $X_i < Y_i$ is *True* for some *i* ≤ *MC* and $X_j = Y_j$ is *True* for 1 (one) ≤ *j* < *i*.

x) If *X* and *Y* are both SQL/JSON objects that are not equivalent, then it is implementation-defined (IA229) whether *X* < *Y* is *True*.

Conformance Rules

None.

NOTE 335 — If <comp op> is <equals operator> or <not equals operator>, then the Conformance Rules of Subclause 9.11, "Equality operations", apply. Otherwise, the Conformance Rules of Subclause 9.14, "Ordering operations", apply.

8.3 <between predicate>

Function

Specify a range comparison.

Format

```
<between predicate> ::=
  <row value predicand> <between predicate part 2>

<between predicate part 2> ::=
  [ NOT ] BETWEEN [ ASYMMETRIC | SYMMETRIC ]
      <row value predicand> AND <row value predicand>
```

Syntax Rules

1) If neither SYMMETRIC nor ASYMMETRIC is specified, then ASYMMETRIC is implicit.

2) Let *X*, *Y*, and *Z* be the first, second, and third <row value predicand>s, respectively.

3) "*X* NOT BETWEEN SYMMETRIC *Y* AND *Z*" is equivalent to "NOT (*X* BETWEEN SYMMETRIC *Y* AND *Z*)".

4) "*X* BETWEEN SYMMETRIC *Y* AND *Z*" is equivalent to "((*X* BETWEEN ASYMMETRIC *Y* AND *Z*) OR (*X* BETWEEN ASYMMETRIC *Z* AND *Y*))".

5) "*X* NOT BETWEEN ASYMMETRIC *Y* AND *Z*" is equivalent to "NOT (*X* BETWEEN ASYMMETRIC *Y* AND *Z*)".

6) "*X* BETWEEN ASYMMETRIC *Y* AND *Z*" is equivalent to "*X*>=*Y* AND *X*<=*Z*".

7) The Conformance Rules of Subclause 8.2, "<comparison predicate>", are applied to the result of all syntactic transformations specified in this Subclause.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T461, "Symmetric BETWEEN predicate", conforming SQL language shall not contain a <between predicate> that simply contains SYMMETRIC or ASYMMETRIC.

NOTE 336 — Since <between predicate> is an ordering operation, the Conformance Rules of Subclause 9.14, "Ordering operations", also apply.

8.4 <in predicate>

Function

Specify a quantified comparison.

Format

```
<in predicate> ::=
  <row value predicand> <in predicate part 2>

<in predicate part 2> ::=
  [ NOT ] IN <in predicate value>

<in predicate value> ::=
    <table subquery>
  | <left paren> <in value list> <right paren>

<in value list> ::=
  <row value expression> [ { <comma> <row value expression> }... ]
```

Syntax Rules

1) If <in value list> consists of a single <row value expression>, then that <row value expression> shall not be a <scalar subquery>.

   NOTE 337 — This Syntax Rule resolves an ambiguity in which <in predicate value> might be interpreted either as a <table subquery> or as a <scalar subquery>. The ambiguity is resolved by adopting the interpretation that the <in predicate value> will be interpreted as a <table subquery>.

2) Let *IVL* be an <in value list>.

   `( IVL )`

   is equivalent to the <table value constructor>:

   `( VALUES IVL )`

3) Let *RVC* be the <row value predicand> and let *IPV* be the <in predicate value>.

4) The expression

   `RVC NOT IN IPV`

   is equivalent to

   `NOT ( RVC IN IPV )`

5) The expression

   `RVC IN IPV`

   is equivalent to the <quantified comparison predicate>:

   `RVC = ANY IPV`

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F561, "Full value expressions", conforming SQL language shall not contain a <row value expression> immediately contained in an <in value list> that is not a <value specification>.

NOTE 338 — Since <in predicate> is an equality operation, the Conformance Rules of Subclause 9.11, "Equality operations", also apply.

8.5 <like predicate>

Function

Specify a pattern-match comparison.

Format

```
<like predicate> ::=
    <character like predicate>
  | <octet like predicate>

<character like predicate> ::=
  <row value predicand> <character like predicate part 2>

<character like predicate part 2> ::=
  [ NOT ] LIKE <character pattern> [ ESCAPE <escape character> ]

<character pattern> ::=
  <character value expression>

<escape character> ::=
  <character value expression>

<octet like predicate> ::=
  <row value predicand> <octet like predicate part 2>

<octet like predicate part 2> ::=
  [ NOT ] LIKE <octet pattern> [ ESCAPE <escape octet> ]

<octet pattern> ::=
  <binary value expression>

<escape octet> ::=
  <binary value expression>
```

Syntax Rules

1) The <row value predicand> immediately contained in <character like predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared types of *CVE*, <character pattern>, and <escape character> shall be character string. *CVE*, <character pattern>, and <escape character> shall be comparable.

2) The <row value predicand> immediately contained in <octet like predicate> shall be a <row value constructor predicand> that is a <common value expression> *OVE*. The declared types of *OVE*, <octet pattern>, and <escape octet> shall be binary string.

3) If <character like predicate> is specified, then:

   a) Let *MC* be the <character value expression> of *CVE*, let *PC* be the <character value expression> of the <character pattern>, and let *EC* be the <character value expression> of the <escape character> if one is specified.

   b) *MC* NOT LIKE *PC*

      is equivalent to

      NOT (*MC* LIKE *PC*)

c) *MC* NOT LIKE *PC* ESCAPE *EC*

is equivalent to

NOT (*MC* LIKE *PC* ESCAPE *EC*)

d) The Syntax Rules of Subclause 9.15, "Collation determination", are applied with set of declared types of *CVE* and *PC* as *TYPESET*; let the collation used for <like predicate> be the *COLL* returned from the application of those Syntax Rules. Let *LCN* be a collation equivalent to *COLL* with the NO PAD characteristic.

It is implementation-defined (IA060) which collations can be used as collations for the <like predicate>.

4) If <octet like predicate> is specified, then:

a) Let *MB* be the <binary value expression> of the *OVE*, let *PB* be the <binary value expression> of the <octet pattern>, and let *EB* be the <binary value expression> of the <escape octet> if one is specified.

b) *MB* NOT LIKE *PB*

is equivalent to

NOT (*MB* LIKE *PB*)

c) *MB* NOT LIKE *PB* ESCAPE *EB*

is equivalent to

NOT (*MB* LIKE *PB* ESCAPE *EB*)

Access Rules

None.

General Rules

1) Let *MCV* be the value of *MC* and let *PCV* be the value of *PC*. If *EC* is specified, then let *ECV* be its value.

2) Let *MBV* be the value of *MB* and let *PBV* be the value of *PB*. If *EB* is specified, then let *EBV* be its value.

3) If <character like predicate> is specified, then:

a) Case:

i) If ESCAPE is not specified and at least one of *MCV* and *PCV* are the null value, then the result of

MC LIKE *PC*

is *Unknown*.

ii) If ESCAPE is specified and at least one of *MCV*, *PCV*, and *ECV* are null values, then the result of

MC LIKE *PC* ESCAPE *EC*

is *Unknown*.

NOTE 339 — If none of *MCV*, *PCV*, and *ECV* (if present) are the null value, then the result is either *True* or *False*.

b) Case:

i) If an <escape character> is specified, then:

1) If the length in characters of *ECV* is not equal to 1, then an exception condition is raised: *data exception — invalid escape character (22019)*.

2) If there is not a partitioning of the string *PCV* into substrings such that each substring has length 1 (one) or 2, no substring of length 1 (one) is the escape character *ECV*, and each substring of length 2 is the escape character *ECV* followed by either the escape character *ECV*, an <underscore> character, or the <percent> character, then an exception condition is raised: *data exception — invalid escape sequence (22025)*.

If there is such a partitioning of *PCV*, then in that partitioning, each substring with length 2 represents a single occurrence of the second character of that substring and is called a *single character specifier*.

Each substring with length 1 (one) that is the <underscore> character represents an *arbitrary character specifier*. Each substring with length 1 (one) that is the <percent> character represents an *arbitrary string specifier*. Each substring with length 1 (one) that is neither the <underscore> character nor the <percent> character represents the character that it contains and is called a *single character specifier*.

ii) If an <escape character> is not specified, then each <underscore> character in *PCV* represents an arbitrary character specifier, each <percent> character in *PCV* represents an arbitrary string specifier, and each character in *PCV* that is neither the <underscore> character nor the <percent> character represents itself and is called a *single character specifier*.

c) The string *PCV* is a sequence of the minimum number of substring specifiers such that each portion of *PCV* is part of exactly one substring specifier. A *substring specifier* is an arbitrary character specifier, arbitrary string specifier, or any sequence of single character specifiers.

d) Case:

i) If the lengths of both *MCV* and *PCV* are 0 (zero), then

```
MC LIKE PC
```

is *True*.

ii) The <predicate>

```
MC LIKE PC
```

is *True* if there exists a partitioning of *MCV* into substrings such that:

1) A substring of *MCV* is a sequence of 0 (zero) or more contiguous characters of *MCV* and each character of *MCV* is part of exactly one substring.

2) If the *i*-th substring specifier PCV_i of *PCV* is an arbitrary character specifier, then the *i*-th substring MCV_i of *MCV* is any single character.

3) If the *i*-th substring specifier of *PCV* is an arbitrary string specifier, then the *i*-th substring of *MCV* is any sequence of 0 (zero) or more characters.

4) If the *i*-th substring specifier of *PCV* is sequence of single character specifiers, then the *i*-th substring of *MCV* contains 1 (one) or more characters and

```
'PCV_i' = 'MCV_i' COLLATE LCN
```

is *True*.

5) The number of substrings of *MCV* is equal to the number of substring specifiers of *PCV*.

iii) Otherwise,

```
MC LIKE PC
```

is *False*.

4) If <octet like predicate> is specified, then:

a) Case:

i) If ESCAPE is not specified and at least one of *MBV* and *PBV* are null values, then the result of

```
MB LIKE PB
```

is *Unknown*.

ii) If ESCAPE is specified and at least one of *MBV*, *PBV*, and *EBV* are null values, then the result of

```
MB LIKE PB ESCAPE EB
```

is *Unknown*.

NOTE 340 — If none of *MBV*, *PBV*, and *EBV* (if present) are the null value, then the result is either *True* or *False*.

b) <percent> in the context of an <octet like predicate> has the same bit pattern as the encoding of a <percent> in the SQL_TEXT character set.

c) <underscore> in the context of an <octet like predicate> has the same bit pattern as the encoding of an <underscore> in the SQL_TEXT character set.

d) Case:

i) If an <escape octet> is specified, then:

1) If the length in octets of *EBV* is not equal to 1, then an exception condition is raised: *data exception — invalid escape octet (2200D)*.

2) If there is not a partitioning of the string *PBV* into substrings such that each substring has length 1 (one) or 2, no substring of length 1 (one) is the escape octet *EBV*, and each substring of length 2 is the escape octet *EBV* followed by either the escape octet *EBV*, an <underscore> octet, or the <percent> octet, then an exception condition is raised: *data exception — invalid escape sequence (22025)*.

If there is such a partitioning of *PBV*, then in that partitioning, each substring with length 2 represents a single occurrence of the second octet of that substring. Each substring with length 1 (one) that is the <underscore> octet represents an *arbitrary octet specifier*. Each substring with length 1 (one) that is the <percent> octet represents an *arbitrary string specifier*. Each substring with length 1 (one) that is neither the <underscore> octet nor the <percent> octet represents the octet that it contains.

ii) If an <escape octet> is not specified, then each <underscore> octet in *PBV* represents an arbitrary octet specifier, each <percent> octet in *PBV* represents an arbitrary string specifier, and each octet in *PBV* that is neither the <underscore> octet nor the <percent> octet represents itself.

e) The string *PBV* is a sequence of the minimum number of substring specifiers such that each portion of *PBV* is part of exactly one substring specifier. A *substring specifier* is an arbitrary octet specifier, an arbitrary string specifier, or any sequence of octets other than an arbitrary octet specifier or an arbitrary string specifier.

f) Case:

i) If the lengths of both *MBV* and *PBV* are 0 (zero), then

```
MB LIKE PB
```

is *True*.

ii) The <predicate>

```
MB LIKE PB
```

is *True* if there exists a partitioning of *MBV* into substrings such that:

1) A substring of *MBV* is a sequence of 0 (zero) or more contiguous octets of *MBV* and each octet of *MBV* is part of exactly one substring.

2) If the *i*-th substring specifier of *PBV* is an arbitrary octet specifier, the *i*-th substring of *MBV* is any single octet.

3) the *i*-th substring specifier of *PBV* is an arbitrary string specifier, then the *i*-th substring of *MBV* is any sequence of 0 (zero) or more octets.

4) If the *i*-th substring specifier of *PBV* is neither an arbitrary character specifier nor an arbitrary string specifier, then the *i*-th substring of *MBV* has the same length and bit pattern as that of the substring specifier.

5) The number of substrings of *MBV* is equal to the number of substring specifiers of *PBV*.

iii) Otherwise:

```
MB LIKE PB
```

is *False*.

Conformance Rules

1) Without at least one of Feature T042, "Extended LOB data type support", or Feature T022, "Advanced support for BINARY and VARBINARY data types", conforming SQL language shall not contain an <octet like predicate>.

2) Without Feature F281, "LIKE enhancements", conforming SQL language shall not contain a <common value expression> simply contained in the <row value predicand> immediately contained in <character like predicate> that is not a column reference.

3) Without Feature F281, "LIKE enhancements", conforming SQL language shall not contain a <character pattern> that is not a <value specification>.

4) Without Feature F281, "LIKE enhancements", conforming SQL language shall not contain an <escape character> that is not a <value specification>.

5) Without Feature T042, "Extended LOB data type support", in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type CHARACTER LARGE OBJECT

6) Without both Feature F421, "National character", and Feature T042, "Extended LOB data type support", in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type NATIONAL CHARACTER LARGE OBJECT.

7) Without Feature F421, "National character", in conforming SQL language, a <character value expression> simply contained in a <like predicate> shall not be of declared type NATIONAL CHARACTER or NATIONAL CHARACTER VARYING.

8.6 <similar predicate>

Function

Specify a character string similarity by means of a regular expression.

Format

```
<similar predicate> ::=
  <row value predicand> <similar predicate part 2>

<similar predicate part 2> ::=
  [ NOT ] SIMILAR TO <similar pattern> [ ESCAPE <escape character> ]

<similar pattern> ::=
  <character value expression>

<regular expression> ::=
    <regular term>
  | <regular expression> <vertical bar> <regular term>

<regular term> ::=
    <regular factor>
  | <regular term> <regular factor>

<regular factor> ::=
    <regular primary>
  | <regular primary> <asterisk>
  | <regular primary> <plus sign>
  | <regular primary> <question mark>
  | <regular primary> <repeat factor>

<repeat factor> ::=
  <left brace> <low value> [ <upper limit> ] <right brace>

<upper limit> ::=
  <comma> [ <high value> ]

<low value> ::=
  <unsigned decimal integer>

<high value> ::=
  <unsigned decimal integer>

<regular primary> ::=
    <character specifier>
  | <percent>
  | <regular character set>
  | <left paren> <regular expression> <right paren>

<character specifier> ::=
    <non-escaped character>
  | <escaped character>

<non-escaped character> ::=
  !! See the Syntax Rules.

<escaped character> ::=
  !! See the Syntax Rules.

<regular character set> ::=
```

```
    <underscore>
  | <left bracket> <character enumeration>... <right bracket>
  | <left bracket> <circumflex> <character enumeration>... <right bracket>
  | <left bracket> <character enumeration include>...
       <circumflex> <character enumeration exclude>... <right bracket>

<character enumeration include> ::=
  <character enumeration>

<character enumeration exclude> ::=
  <character enumeration>

<character enumeration> ::=
    <character specifier>
  | <character specifier> <minus sign> <character specifier>
  | <left bracket> <colon> <regular character set identifier> <colon> <right bracket>

<regular character set identifier> ::=
  <identifier>
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared types of *CVE*, <similar pattern>, and <escape character> shall be character string. *CVE*, <similar pattern>, and <escape character> shall be comparable.

   NOTE 341 — An implication of the preceding Rule is that *CVE* immediately contains a <string value expression> that, in turn, immediately contains a <character value expression>. In other words, *CVE* is necessarily a <character value expression>.

2) Let *CM* be the <character value expression> of *CVE* and let *SP* be the <similar pattern>. If <escape character> *EC* is specified, then

   CM NOT SIMILAR TO *SP* ESCAPE *EC*

   is equivalent to

   NOT (*CM* SIMILAR TO *SP* ESCAPE *EC*)

   If <escape character> *EC* is not specified, then

   CM NOT SIMILAR TO *SP*

   is equivalent to

   NOT (*CM* SIMILAR TO *SP*)

3) The value of the <identifier> that is a <regular character set identifier> shall be either ALPHA, UPPER, LOWER, DIGIT, ALNUM, SPACE, or WHITESPACE.

4) The Syntax Rules of Subclause 9.15, "Collation determination", are applied with set of declared types of *CVE*, *SP*, and (if specified) *EC* as *TYPESET*; let the collation used for <similar predicate> be the *COLL* returned from the application of those Syntax Rules.

   It is implementation-defined (IA061) which collations can be used as collations for <similar predicate>.

5) A <non-escaped character> is any single character from the character set of the <similar pattern> that is not a <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>, or the character specified by the result of the <character value expression> of <escape character>. A <character specifier> that is a <non-escaped character> represents itself.

6) An <escaped character> is a sequence of two characters: the character specified by the result of the <character value expression> of <escape character>, followed by a second character that is a <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace>, or the character specified by the result of the <character value expression> of <escape character>. A <character specifier> that is an <escaped character> represents its second character.

7) Neither <low value> nor <high value> shall contain an <underscore>.

NOTE 342 — This Syntax Rule maintains consistency of the quantifier syntax in a <similar pattern> and an <XQuery pattern>.

8) The value of <low value> shall be a positive integer. The value of <high value> shall be greater than or equal to the value of <low value>.

Access Rules

None.

General Rules

1) Let *MCV* be the result of the <character value expression> of *CVE* and let *PCV* be the result of the <character value expression> of the <similar pattern>. If *EC* is specified, then let *ECV* be its value.

2) If the result of the <character value expression> of the <similar pattern> is not the zero-length character string and does not have the format of a <regular expression>, then an exception condition is raised: *data exception — invalid regular expression (2201B)*.

3) If an <escape character> is specified, then:

   a) If the length in characters of *ECV* is not equal to 1 (one), then an exception condition is raised: *data exception — invalid escape character (22019)*.

   b) If *ECV* is one of <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, or <left brace> and *ECV* occurs in the <regular expression> except in an <escaped character>, then an exception condition is raised: *data exception — invalid use of escape character (2200C)*.

   c) If *ECV* is a <colon> and the <regular expression> contains a <regular character set identifier>, then an exception condition is raised: *data exception — escape character conflict (2200B)*.

4) Case:

   a) If ESCAPE is not specified and at least one of *MCV* and *PCV* are the null value, then the result of

      ```
      CM SIMILAR TO SP
      ```

      is *Unknown*.

   b) If ESCAPE is specified, then if at least one of *MCV*, *PCV*, and *ECV* are the null value, then the result of

      ```
      CM SIMILAR TO SP ESCAPE EC
      ```

      is *Unknown*.

NOTE 343 — If none of *MCV*, *PCV*, and *ECV* (if present) are the null value, then the result is either *True* or *False*.

5) The set of characters in a <character enumeration> is defined as

Case:

a) If the enumeration is specified in the form "<character specifier> <minus sign> <character specifier>", then the set of all characters that collate greater than or equal to the character represented by the left <character specifier> and less than or equal to the character represented by the right <character specifier>, according to the collation of the pattern *PCV*.

b) Otherwise, the character that the <character specifier> in the <character enumeration> represents.

6) Let *LV* be the value of the <low value> contained in a <repeat factor> *RF*.

Case:

a) If *RF* does not contain an <upper limit>, then let *HV* be *LV*.

b) If *RF* contains an <upper limit> that contains a <high value>, then let *HV* be the value of <high value>.

c) Otherwise, let *HV* be the length or maximum length of *CVE*.

7) Let *R* be the result of the <character value expression> of the <similar pattern>. The regular language L(*R*) of the <similar pattern> is a (possibly infinite) set of strings. It is defined recursively for well-formed <regular expression>s *Q*, *Q1*, and *Q2* by the following rules:

a) L(*Q1* <vertical bar> *Q2*)

is the union of L(*Q1*) and L(*Q2*)

b) L(*Q* <asterisk>)

is the set of all strings that can be constructed by concatenating zero or more strings from L(*Q*).

c) L(*Q* <plus sign>)

is the set of all strings that can be constructed by concatenating one or more strings from L(*Q*).

d) L(*Q* <repeat factor>)

is the set of all strings that can be constructed by concatenating *NS*, $LV \leq NS \leq HV$, strings from L(*Q*).

e) L(<character specifier>)

is a set that contains a single string of length 1 (one) with the character that the <character specifier> represents

f) L(<percent>)

is the set of all strings of any length (zero or more) from the character set of the pattern *PCV*.

g) L(*Q* <question mark>)

is the set of all strings that can be constructed by concatenating exactly 0 (zero) or 1 (one) strings from L(*Q*).

h) L(<left paren> *Q* <right paren>)

is equal to L(*Q*)

i) L(<underscore>)

is the set of all strings of length 1 (one) from the character set of the pattern *PCV*.

j) L(<left bracket> <character enumeration> <right bracket>)

is the set of all strings of length 1 (one) from the set of characters in the <character enumeration>s.

k) L(<left bracket> <circumflex> <character enumeration> <right bracket>)

is the set of all strings of length 1 (one) with characters from the character set of the pattern *PCV* that are not contained in the set of characters in the <character enumeration>.

l) L(<left bracket> <character enumeration include> <circumflex> <character enumeration exclude> <right bracket>)

is the set of all strings of length 1 (one) taken from the set of characters in the <character enumeration include>s, except for those strings of length 1 (one) taken from the set of characters in the <character enumeration exclude>.

m) L(<left bracket> <colon> ALPHA <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin letter>s.

n) L(<left bracket> <colon> UPPER <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin upper-case letter>s.

o) L(<left bracket> <colon> LOWER <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin lower-case letter>s.

p) L(<left bracket> <colon> DIGIT <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <digit>s.

q) L(<left bracket> <colon> SPACE <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are the <space> character.

r) L(<left bracket> <colon> WHITESPACE <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are whitespace characters.

NOTE 344 — "whitespace" is defined in Clause 3, "Terms and definitions".

s) L(<left bracket> <colon> ALNUM <colon> <right bracket>)

is the set of all character strings of length 1 (one) that are <simple Latin letter>s or <digit>s.

t) `L( Q1 || Q2 )`

is the set of all strings that can be constructed by concatenating one element of L(*Q1*) and one element of L(*Q2*).

u) L(*Q*)

is the set of the zero-length character string, if *Q* is an empty regular expression.

8) The <similar predicate>

```
CM SIMILAR TO SP
```

is *True*, if there exists at least one element of L(*R*) that is equal to *MCV* according to the collation of the <similar predicate>; otherwise, it is *False*.

NOTE 345 — The <similar predicate> is defined differently from equivalent forms of the LIKE predicate. In particular, blanks at the end of a pattern and collation are handled differently.

Conformance Rules

1) Without Feature T141, "SIMILAR predicate", conforming SQL language shall not contain a <similar predicate>.

2) Without Feature T042, "Extended LOB data type support", in conforming SQL language, a <character value expression> simply contained in a <similar predicate> shall not be of declared type CHARACTER LARGE OBJECT.

8.7 <regex like predicate>

Function

Specify a pattern-match comparison using an XQuery regular expression.

Format

```
<regex like predicate> ::=
  <row value predicand> <regex like predicate part 2>

<regex like predicate part 2> ::=
  [ NOT ] LIKE_REGEX <XQuery pattern> [ FLAG <XQuery option flag> ]
```

Syntax Rules

1) The <row value predicand> immediately contained in <regex like predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE* whose declared type is character string.

2) If <XQuery option flag> is not specified, then the zero-length character string is implicit.

3) Let *PAT* be the <XQuery pattern> and let *FL* be the implicit or explicit <XQuery option flag>.

   CVE NOT LIKE_REGEX *PAT* FLAG *FL*

   is equivalent to

   NOT (*CVE* LIKE_REGEX *PAT* FLAG *FL*)

Access Rules

None.

General Rules

1) The value of the <predicate>

   CVE LIKE_REGEX *PAT* FLAG *FL*

   is

   Case:

   a) If at least one of *CVE*, *PAT*, and *FL* is the null value, then *Unknown*.

   b) Otherwise, the General Rules of Subclause 9.28, "XQuery regular expression matching", are applied with *CVE* as *STRING*, *PAT* as *PATTERN*, 1 (one) as *POSITION*, CHARACTERS as *UNITS*, and *FL* as *FLAG*; let *LOMV* be the *LIST* returned from the application of those General Rules.

      Case:

      i) If the list of match vectors in *LOMV* is non-empty, then *True*.

      ii) Otherwise, *False*.

Conformance Rules

1) Without Feature F841, “LIKE_REGEX predicate”, conforming SQL language shall not contain <regex like predicate>.

2) Without Feature F847, “Non-constant regular expression”, in conforming SQL language, <XQuery pattern> and <XQuery option flag> shall be <value specification>s.

8.8 <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::=
  <row value predicand> <null predicate part 2>

<null predicate part 2> ::=
  IS [ NOT ] NULL
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *R* be the <row value predicand> and let *V* be the value of *R*.

2) Case:

   a) If *V* is the null value, then "*R* `IS NULL`" is *True* and the value of "*R* `IS NOT NULL`" is *False*.

   b) Otherwise:

      i) The value of "*R* `IS NULL`" is

         Case:

         1) If the value of every field of *V* is the null value, then *True*.

         2) Otherwise, *False*.

      ii) The value of "*R* `IS NOT NULL`" is

         Case:

         1) If the value of no field of *V* is the null value, then *True*.

         2) Otherwise, *False*.

NOTE 346 — For all *R*, "*R* IS NOT NULL" has the same result as "NOT *R* IS NULL" if and only if *R* is of degree 1. Table 18, "<null predicate> semantics", specifies this behavior.

Table 18 — <null predicate> semantics

Expression	*R* IS NULL	*R* IS NOT NULL	NOT *R* IS NULL	NOT *R* IS NOT NULL
degree 1: null	*True*	*False*	*False*	*True*
degree 1: not null	*False*	*True*	*True*	*False*
degree > 1: all null	*True*	*False*	*False*	*True*
degree > 1: some null	*False*	*False*	*True*	*True*
degree > 1: none null	*False*	*True*	*True*	*False*

Conformance Rules

None.

8.9 <quantified comparison predicate>

Function

Specify a quantified comparison.

Format

```
<quantified comparison predicate> ::=
  <row value predicand> <quantified comparison predicate part 2>

<quantified comparison predicate part 2> ::=
  <comp op> <quantifier> <table subquery>

<quantifier> ::=
    <all>
  | <some>

<all> ::=
  ALL

<some> ::=
    SOME
  | ANY
```

Syntax Rules

1) Let *RV1* and *RV2* be <row value predicand>s whose declared types are respectively that of the <row value predicand> and the row type of the <table subquery>. The Syntax Rules and Conformance Rules of Subclause 8.2, "<comparison predicate>", are applied to:

   RV1 <comp op> *RV2*

Access Rules

None.

General Rules

1) Let *R* be the result of the <row value predicand> and let *T* be the result of the <table subquery>.

2) The result of "*R* <comp op> <quantifier> *T*" is derived by the application of the implied <comparison predicate> "*R* <comp op> *RT*" to every row *RT* in *T*.

   Case:

   a) If *T* is empty or if the implied <comparison predicate> is *True* for every row *RT* in *T*, then "*R* <comp op> <all> *T*" is *True*.

   b) If the implied <comparison predicate> is *False* for at least one row *RT* in *T*, then "*R* <comp op> <all> *T*" is *False*.

   c) If the implied <comparison predicate> is *True* for at least one row *RT* in *T*, then "*R* <comp op> <some> *T*" is *True*.

d) If *T* is empty or if the implied <comparison predicate> is *False* for every row *RT* in *T*, then "*R* <comp op> <some> *T*" is *False*.

e) If "*R* <comp op> <quantifier> *T*" is neither *True* nor *False*, then it is *Unknown*.

Conformance Rules

None.

NOTE 347 — If <equals operator> or <not equals operator> is specified, then the <quantified comparison predicate> is an equality operator and the Conformance Rules of Subclause 9.11, "Equality operations", apply. Otherwise, the <quantified comparison predicate> is an ordering operation, and the Conformance Rules of Subclause 9.14, "Ordering operations", apply.

8.10 <exists predicate>

Function

Specify a test for a non-empty set.

Format

```
<exists predicate> ::=
  EXISTS <table subquery>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *T* be the result of the <table subquery>.

2) If the cardinality of *T* is greater than 0 (zero), then the result of the <exists predicate> is *True*; otherwise, the result of the <exists predicate> is *False*.

Conformance Rules

1) Without Feature T501, "Enhanced EXISTS predicate", conforming SQL language shall not contain an <exists predicate> that simply contains a <table subquery> in which the <select list> of a <query specification> directly contained in the <table subquery> does not comprise either an <asterisk> or a single <derived column>.

8.11 <unique predicate>

Function

Specify a test for the absence of duplicate rows.

Format

```
<unique predicate> ::=
  UNIQUE [ <unique null treatment> ] <table subquery>

<unique null treatment> ::=
  NULLS [ NOT ] DISTINCT
```

Syntax Rules

1) Each column of user-defined type in the result of the <table subquery> shall have a comparison type.

2) Each column of the <table subquery> is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

3) If <unique null treatment> is not specified, then an implementation-defined (ID106) <unique null treatment> is implicit.

Access Rules

None.

General Rules

1) Let *T* be the result of the <table subquery>.

2) Let *UNT* be the implicit or explicit <unique null treatment>.

   Case:

   a) If there are zero or one rows in *T*, then the result of the <unique predicate> is *True*.

   b) If *UNT* is NULLS DISTINCT, and for every two rows *R1* and *R2* in *T*, *R1* and *R2* are unique with nulls distinct, then the result of the <unique predicate> is *True*.

   c) If *UNT* is NULLS NOT DISTINCT, and for every two rows *R1* and *R2* in *T*, *R1* and *R2* are unique with nulls not distinct, then the result of the <unique predicate> is *True*.

   d) Otherwise, the result of the <unique predicate> is *False*.

Conformance Rules

1) Without Feature F291, "UNIQUE predicate", conforming SQL language shall not contain a <unique predicate>.

   NOTE 348 — The Conformance Rules of Subclause 9.12, "Grouping operations", also apply.

2) Without Feature F292, “UNIQUE null treatment”, conforming SQL language shall not contain an explicit <unique null treatment>.

8.12 <normalized predicate>

Function

Determine whether a character string value is normalized.

Format

```
<normalized predicate> ::=
  <row value predicand> <normalized predicate part 2>

<normalized predicate part 2> ::=
  IS [ NOT ] [ <normal form> ] NORMALIZED
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a <common value expression> *CVE*. The declared type of *CVE* shall be character string and the character set of *CVE* shall be UTF8, UTF16, or UTF32.

2) Case:

   a) If <normal form> is specified, then let *NF* be <normal form>.

   b) Otherwise, let *NF* be NFC.

3) The expression

   CVE IS NOT *NF* NORMALIZED

   is equivalent to

   NOT (*CVE* IS *NF* NORMALIZED)

Access Rules

None.

General Rules

1) The result of *CVE* IS *NF* NORMALIZED is

   Case:

   a) If the value of *CVE* is the null value, then *Unknown*.

   b) If the value of *CVE* is in the normalization form specified by *NF*, as defined by ISO/IEC 10646:2020, then *True*.

   c) Otherwise, *False*.

Conformance Rules

1) Without Feature T061, "UCS support", conforming SQL language shall not contain a <normalized predicate>.

2) Without Feature F394, “Optional normal form specification”, conforming SQL language shall not contain <normal form>.

8.13 <match predicate>

Function

Specify a test for matching rows.

Format

```
<match predicate> ::=
  <row value predicand> <match predicate part 2>

<match predicate part 2> ::=
  MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ] <table subquery>
```

Syntax Rules

1) The row type of the <row value predicand> and the row type of the <table subquery> shall be comparable.

2) Each field of <row value predicand> and each column of <table subquery> is an operand of an equality operation. The Syntax Rules and Conformance Rules of Subclause 9.11, "Equality operations", apply.

3) If neither SIMPLE, PARTIAL, nor FULL is specified, then SIMPLE is implicit.

Access Rules

None.

General Rules

1) Let *R* be the <row value predicand>.

2) If SIMPLE is specified or implicit, then

   Case:

   a) If *R* is the null value, then the <match predicate> is *True*.

   b) Otherwise:

      i) If the value of some field in *R* is the null value, then the <match predicate> is *True*.

      ii) If the value of no field in *R* is the null value, then

          Case:

          1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that

             $R = RT_i$

             then the <match predicate> is *True*.

          2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that

$R = RT_i$

then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

3) If PARTIAL is specified, then

Case:

a) If R is the null value, then the <match predicate> is *True*.

b) Otherwise,

Case:

i) If the value of every field in R is the null value, then the <match predicate> is *True*.

ii) Otherwise,

Case:

1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that each non-null value of R equals its corresponding value in RT_i, then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that each non-null value of R equals its corresponding value in RT_i, then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

4) If FULL is specified, then

Case:

a) If R is the null value, then the <match predicate> is *True*.

b) Otherwise,

Case:

i) If the value of every field in R is the null value, then the <match predicate> is *True*.

ii) If the value of no field in R is the null value, then

Case:

1) If UNIQUE is not specified and there exists a row RT_i of the <table subquery> such that

$R = RT_i$

then the <match predicate> is *True*.

2) If UNIQUE is specified and there exists exactly one row RT_i in the result of evaluating the <table subquery> such that

$R = RT_i$

then the <match predicate> is *True*.

3) Otherwise, the <match predicate> is *False*.

iii) Otherwise, the <match predicate> is *False*.

Conformance Rules

1) Without Feature F741, "Referential MATCH types", conforming SQL language shall not contain a <match predicate>.

NOTE 349 — The Conformance Rules of Subclause 9.11, "Equality operations", also apply.

8.14 <overlaps predicate>

Function

Specify a test for an overlap between two datetime periods.

Format

```
<overlaps predicate> ::=
  <overlaps predicate part 1> <overlaps predicate part 2>

<overlaps predicate part 1> ::=
  <row value predicand 1>

<overlaps predicate part 2> ::=
  OVERLAPS <row value predicand 2>

<row value predicand 1> ::=
  <row value predicand>

<row value predicand 2> ::=
  <row value predicand>
```

Syntax Rules

1) The degrees of <row value predicand 1> and <row value predicand 2> shall both be 2.

2) The declared types of the first field of <row value predicand 1> and the first field of <row value predicand 2> shall both be datetime data types and these data types shall be comparable.

   NOTE 350 — Two datetimes are comparable only if they have the same <primary datetime field>s; see Subclause 4.7.2, "Datetimes".

3) The declared type of the second field of each <row value predicand> shall be a datetime data type or INTERVAL.

   Case:

   a) If the declared type is INTERVAL, then the precision of the declared type shall be such that the interval can be added to the datetime data type of the first column of the <row value predicand>.

   b) If the declared type is a datetime data type, then it shall be comparable with the datetime data type of the first column of the <row value predicand>.

Access Rules

None.

General Rules

1) If at least one of <row value predicand 1> and <row value predicand 2> is the null value, then the result of the <overlaps predicate> is *Unknown* and no further General Rules of this Subclause are applied.

2) Let *D1* be the value of the first field of <row value predicand 1> and *D2* be the value of the first field of <row value predicand 2>.

3) Case:

   a) If the most specific type of the second field of <row value predicand 1> is a datetime data type, then let *E1* be the value of the second field of <row value predicand 1>.

   b) If the most specific type of the second field of <row value predicand 1> is INTERVAL, then let *I1* be the value of the second field of <row value predicand 1>. Let *E1* = *D1* + *I1*.

4) If *D1* is the null value or if *E1* < *D1*, then let *S1* = *E1* and let *T1* = *D1*. Otherwise, let *S1* = *D1* and let *T1* = *E1*.

5) Case:

   a) If the most specific type of the second field of <row value predicand 2> is a datetime data type, then let *E2* be the value of the second field of <row value predicand 2>.

   b) If the most specific type of the second field of <row value predicand 2> is INTERVAL, then let *I2* be the value of the second field of <row value predicand 2>. Let *E2* = *D2* + *I2*.

6) If *D2* is the null value or if *E2* < *D2*, then let *S2* = *E2* and let *T2* = *D2*. Otherwise, let *S2* = *D2* and let *T2* = *E2*.

7) The result of the <overlaps predicate> is the result of the following expression:

```
( S1 > S2 AND NOT ( S1 >= T2 AND T1 >= T2 ) )
    OR
( S2 > S1 AND NOT ( S2 >= T1 AND T2 >= T1 ) )
    OR
( S1 = S2 AND ( T1 <> T2 OR T1 = T2 ) )
```

Conformance Rules

1) Without Feature F053, “OVERLAPS predicate”, conforming SQL language shall not contain an <overlaps predicate>.

8.15 <distinct predicate>

This Subclause is modified by Subclause 8.1, "<distinct predicate>", in ISO/IEC 9075-15.

Function

Specify a test of whether two row values are distinct

Format

```
<distinct predicate> ::=
  <row value predicand 3> <distinct predicate part 2>

<distinct predicate part 2> ::=
  IS [ NOT ] DISTINCT FROM <row value predicand 4>

<row value predicand 3> ::=
  <row value predicand>

<row value predicand 4> ::=
  <row value predicand>
```

Syntax Rules

1) The two <row value predicand>s shall be of the same degree.

2) Let *respective values* be values with the same ordinal position.

3) The declared types of the respective values of the two <row value predicand>s shall be comparable.

4) Let *X* be the first <row value predicand> and let *Y* be the second <row value predicand>.

5) Each field of each <row value predicand> is an operand of an equality operation. The Syntax Rules and Conformance Rules of Subclause 9.11, "Equality operations", apply.

6) If <distinct predicate part 2> immediately contains NOT, then the <distinct predicate> is equivalent to:

```
NOT ( X IS DISTINCT FROM Y )
```

Access Rules

None.

General Rules

1) Let *V1* be the value of <row value predicand 3> and let *V2* be the value of <row value predicand 4>.

   Case:

   a) If both *V1* and *V2* are the null value, then the result is *False*.

   b) If *V1* is the null value and *V2* is not the null value, or if *V1* is not the null value and *V2* is the null value, then the result is *True*.

   c) Otherwise:

i) If *V1* and *V2* are values of a predefined type or a reference type, then

Case:

1) If *V1* and *V2* are not equal, then the result is *True*.

2) Otherwise, the result is *False*.

ii) If *V1* and *V2* are values of a user-defined type whose comparison form is RELATIVE or MAP, then

Case:

1) If the result of comparing *V1* and *V2* for equality according to Subclause 8.2, "<comparison predicate>", is *Unknown*, then it is implementation-dependent (UA074) whether the result is *True* or *False*.

2) If *V1* is not equal to *V2*, then the result is *True*.

3) Otherwise, the result is *False*.

iii) If *V1* and *V2* are values of a user-defined type whose comparison form is STATE, then

Case:

1) If the most specific types of *V1* and *V2* are different, then the result is *True*.

2) If there is an attribute *A* of their common most specific type such that the value of *A* in *V1* and the value of *A* in *V2* are distinct, then the result is *True*.

3) Otherwise, the result is *False*.

iv) If *V1* and *V2* are values of row type, then

Case:

1) If at least one of their pairs of respective fields is distinct, then the result is *True*.

2) Otherwise, the result is *False*.

v) If *V1* and *V2* are values of an array type or values of a distinct type whose source type is an array type, then

Case:

1) If *V1* and *V2* do not have the same cardinality, then the result is *True*.

2) If *V1* and *V2* have the same cardinality and there exists at least one ordinal position *P* such that the array element at position *P* in *V1* is distinct from the array element at position *P* in *V2*, then the result is *True*.

3) Otherwise, the result is *False*.

vi) If *V1* and *V2* are values of a multiset type or values of a distinct type whose source type is a multiset type, then

Case:

1) If there exists a value *V* in the element type of *V1* or *V2*, including the null value, such that the number of elements in *V1* that are not distinct from *V* does not equal the number of elements in *V2* that are not distinct from *V*, then the result is *True*.

2) Otherwise, the result is *False*.

NOTE 351 — "distinct" is defined in Clause 3, "Terms and definitions", and Subclause 4.2.5, "Properties of distinct".

2) If two <row value predicand>s are not distinct, then they are said to be *duplicates*. If a number of <row value predicand>s are all duplicates of each other, then all except one are said to be *redundant duplicates*.

Conformance Rules

1) Without Feature T151, "DISTINCT predicate", conforming SQL language shall not contain a <distinct predicate>.

NOTE 352 — The Conformance Rules of Subclause 9.11, "Equality operations", also apply.

2) Without Feature T152, "DISTINCT predicate with negation", conforming SQL language shall not contain a <distinct predicate part 2> that immediately contains NOT.

8.16 <member predicate>

Function

Specify a test of whether a value is a member of a multiset.

Format

```
<member predicate> ::=
  <row value predicand> <member predicate part 2>

<member predicate part 2> ::=
  [ NOT ] MEMBER [ OF ] <multiset value expression>
```

Syntax Rules

1) Let *MVE* be the <multiset value expression> and let *ET* be the declared element type of *MVE*.

2) Case:

   a) If the <row value predicand> is a <row value constructor predicand> that is a single <common value expression> or <boolean value expression> *CVE*, then let *X* be *CVE*.

   b) Otherwise, let *X* be the <row value predicand>.

3) The declared type of *X* shall be comparable to *ET*.

4) *X* is an operand of an equality operation. The Syntax Rules and Conformance Rules of Subclause 9.11, "Equality operations", apply.

5) If <member predicate part 2> immediately contains NOT, then the <member predicate> is equivalent to

   ```
   NOT ( X MEMBER OF MVE )
   ```

Access Rules

None.

General Rules

1) Let *XV* be the value of *X*, and let *MV* be the value of *MVE*.

2) Let *N* be the result of `CARDINALITY (MVE)`.

3) The <member predicate>

   ```
   XV MEMBER OF MVE
   ```

   is evaluated as follows.

   Case:

   a) If *N* is 0 (zero), then the <member predicate> is *False*.

   b) If at least one of *XV* and *MV* is the null value, then the <member predicate> is *Unknown*.

c) Otherwise, let ME_i for 1 (one) $\leq i \leq N$ be an enumeration of the elements of *MV*.

Case:

i) If $CV = ME_i$ for some *i*, then the <member predicate> is *True*.

ii) If ME_i is the null value for some *i*, then the <member predicate> is *Unknown*.

iii) Otherwise, the <member predicate> is *False*.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <member predicate>.

NOTE 353 — The Conformance Rules of Subclause 9.11, "Equality operations", also apply.

8.17 <submultiset predicate>

Function

Specify a test of whether a multiset is a submultiset of another multiset.

Format

```
<submultiset predicate> ::=
  <row value predicand> <submultiset predicate part 2>

<submultiset predicate part 2> ::=
  [ NOT ] SUBMULTISET [ OF ] <multiset value expression>
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a single <common value expression> *CVE*. The declared type of *CVE* shall be a multiset type or a distinct type whose source type is a multiset type. Let *CVET* be the declared element type of *CVE*.

2) Let *MVE* be the <multiset value expression>. Let *MVET* be the declared element type of *MVE*.

3) *CVET* shall be comparable to *MVET*.

4) *CVE* and *MVE* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

5) If <submultiset predicate part 2> immediately contains NOT, then the <submultiset predicate> is equivalent to

   ```
   NOT ( CVE SUBMULTISET OF MVE )
   ```

Access Rules

None.

General Rules

1) Let *CV* be the value of *CVE*, and let *MV* be the value of *MVE*.

2) Let *M* be the result of CARDINALITY (*CV*), and let *N* be the result of CARDINALITY (*MV*).

3) The <submultiset predicate>

   ```
   CVE SUBMULTISET OF MVE
   ```

   is evaluated as follows.

   Case:

   a) If *M* is 0 (zero), then the <submultiset predicate> is *True*.

   b) If at least one of *CV* and *MV* is the null value, then the <submultiset predicate> is *Unknown*.

   c) Otherwise,

      Case:

i) If $M > N$, then the <submultiset predicate> is *False*.

ii) If there exists an enumeration CE_i for 1 (one) $\leq i \leq M$ of the elements of CV and an enumeration ME_j for 1 (one) $\leq j \leq N$ of the elements of MV such that for all i, 1 (one) $\leq i \leq M$, $CE_i = ME_i$, then the <submultiset predicate> is *True*.

iii) If there exist an enumeration CE_i for 1 (one) $\leq i \leq M$ of the elements of CV and an enumeration ME_j for 1 (one) $\leq i \leq N$ of the elements of MV such that for all i, 1 (one) $\leq j \leq M$, $CE_i = ME_i$ is either *True* or *Unknown*, then the <submultiset predicate> is *Unknown*.

iv) Otherwise, the <submultiset predicate> is *False*.

Conformance Rules

1) Without Feature S275, "Advanced multiset support", conforming SQL language shall not contain a <submultiset predicate>.

NOTE 354 — The Conformance Rules of Subclause 9.13, "Multiset element grouping operations", also apply.

8.18 <set predicate>

Function

Specify a test of whether a multiset is a set (that is, does not contain any duplicates).

Format

```
<set predicate> ::=
  <row value predicand> <set predicate part 2>

<set predicate part 2> ::=
  IS [ NOT ] A SET
```

Syntax Rules

1) The <row value predicand> shall be a <row value constructor predicand> that is a single <common value expression> *CVE*. The declared type of *CVE* shall be a multiset type or a distinct type whose source type is a multiset type. Let *CVET* be the element type of *CVE*.

2) *CVE* is an operand of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

3) If <set predicate part 2> immediately contains NOT, then the <set predicate> is equivalent to

   ```
   NOT ( CVE IS A SET )
   ```

4) If <set predicate part 2> does not immediately contain NOT, then the <set predicate> is equivalent to

   ```
   CARDINALITY ( CVE ) = CARDINALITY ( SET ( CVE ) )
   ```

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <set predicate>.

NOTE 355 — The Conformance Rules of Subclause 9.13, "Multiset element grouping operations", also apply.

8.19 <type predicate>

Function

Specify a type test.

Format

```
<type predicate> ::=
  <row value predicand> <type predicate part 2>

<type predicate part 2> ::=
  IS [ NOT ] OF <left paren> <type list> <right paren>

<type list> ::=
  <user-defined type specification>
      [ { <comma> <user-defined type specification> }... ]

<user-defined type specification> ::=
    <inclusive user-defined type specification>
  | <exclusive user-defined type specification>

<inclusive user-defined type specification> ::=
  <path-resolved user-defined type name>

<exclusive user-defined type specification> ::=
  ONLY <path-resolved user-defined type name>
```

Syntax Rules

1) The <row value predicand> immediately contained in <type predicate> shall be a <row value constructor predicand> that is a <common value expression> *CVE*.

2) The declared type of *CVE* shall be a user-defined type.

3) For each <user-defined type name> *UDTN* contained in a <user-defined type specification>, the schema identified by the implicit or explicit schema name of *UDTN* shall include a user-defined type descriptor whose name is equivalent to the <qualified identifier> of *UDTN*.

4) Let the term *specified type* refer to a user-defined type that is specified by a <user-defined type name> contained in a <user-defined type specification>. A type specified by an <inclusive user-defined type specification> is *inclusively specified*; a type specified by an <exclusive user-defined type specification> is *exclusively specified*.

5) Let *T* be the type specified by <inclusive user-defined type specification> or <exclusive user-defined type specification>. *T* shall be a subtype of the declared type of *CVE*.

   NOTE 356 — The term "subtype family" is defined in Subclause 4.9.3.4, "Subtypes and supertypes". If *T1* is a member of the subtype family of *T2*, then it follows that the subtype family of *T1* and the subtype family of *T2* are the same set of types.

6) Let *TL* be the <type list>.

7) A <type predicate> of the form

   ```
   CVE IS NOT
   OF (TL)
   ```

   is equivalent to

```
NOT ( CVE IS
OF (TL) )
```

Access Rules

None.

General Rules

1) Let *V* be the result of evaluating the <row value predicand>.

2) Let *ST* be the set consisting of every type that is either some exclusively specified type, or a subtype of some inclusively specified type.

3) Let *TPR* be the result of evaluating the <type predicate>.

   Case:

   a) If *V* is the null value, then *TPR* is *Unknown*.

   b) If the most specific type of *V* is a member of *ST*, then *TPR* is *True*.

   c) Otherwise, *TPR* is *False*.

Conformance Rules

1) Without Feature S151, "Type predicate", conforming SQL language shall not contain a <type predicate>.

8.20 <period predicate>

Function

Specify a test to determine the relationship between periods.

Format

```
<period predicate> ::=
    <period overlaps predicate>
  | <period equals predicate>
  | <period contains predicate>
  | <period precedes predicate>
  | <period succeeds predicate>
  | <period immediately precedes predicate>
  | <period immediately succeeds predicate>

<period overlaps predicate> ::=
  <period predicand 1> <period overlaps predicate part 2>

<period overlaps predicate part 2> ::=
  OVERLAPS <period predicand 2>

<period predicand 1> ::=
  <period predicand>

<period predicand 2> ::=
  <period predicand>

<period predicand> ::=
    <period reference>
  | PERIOD <left paren> <period start value> <comma> <period end value> <right paren>

<period reference> ::=
  <basic identifier chain>

<period start value> ::=
  <datetime value expression>

<period end value> ::=
  <datetime value expression>

<period equals predicate> ::=
  <period predicand 1> <period equals predicate part 2>

<period equals predicate part 2> ::=
  EQUALS <period predicand 2>

<period contains predicate> ::=
  <period predicand 1> <period contains predicate part 2>

<period contains predicate part 2> ::=
  CONTAINS <period or point-in-time predicand>

<period or point-in-time predicand> ::=
    <period predicand>
  | <datetime value expression>

<period precedes predicate> ::=
  <period predicand 1> <period precedes predicate part 2>

<period precedes predicate part 2> ::=
```

```
  PRECEDES <period predicand 2>

<period succeeds predicate> ::=
  <period predicand 1> <period succeeds predicate part 2>

<period succeeds predicate part 2> ::=
  SUCCEEDS <period predicand 2>

<period immediately precedes predicate> ::=
  <period predicand 1> <period immediately precedes predicate part 2>

<period immediately precedes predicate part 2> ::=
  IMMEDIATELY PRECEDES <period predicand 2>

<period immediately succeeds predicate> ::=
  <period predicand 1> <period immediately succeeds predicate part 2>

<period immediately succeeds predicate part 2> ::=
  IMMEDIATELY SUCCEEDS <period predicand 2>
```

Syntax Rules

1) For each <period predicand> that specifies PERIOD:

   a) The declared type of any <period start value> and any <period end value> shall not be TIME WITH TIME ZONE or TIME WITHOUT TIME ZONE.

   b) The declared type of the <period start value> immediately contained in such a <period predicand> shall be the same as the declared type of the <period end value> immediately contained in that same <period predicand>.

2) The declared type of any <datetime value expression> immediately contained in a <period or point-in-time predicand> shall not be TIME WITH TIME ZONE or TIME WITHOUT TIME ZONE.

Access Rules

None.

General Rules

1) The result of a <period predicate> is the truth value of the immediately contained <period overlaps predicate>, <period equals predicate>, <period contains predicate>, <period precedes predicate>, <period succeeds predicate>, <period immediately precedes predicate>, or <period immediately succeeds predicate>.

2) If one or more <period predicand>s specify PERIOD, then let *m* be the number of <period predicand>s specified. For *i*, 1 (one) ≤ *i* ≤ *m*, let PSV_i be the value of <period start value> in the *i*-th <period predicand> and let PEV_i be the value of <period end value> in the *i*-th <period predicand>.

   Case:

   a) If at least one of PSV_i and PEV_i, 1 (one) ≤ *i* ≤ *m*, is the null value, then the result of the specified <period overlaps predicate>, <period equals predicate>, <period contains predicate>, <period precedes predicate>, <period succeeds predicate>, <period immediately precedes predicate>, or <period immediately succeeds predicate> is *Unknown* and no further General Rules of this Subclause are applied.

b) If for any i, 1 (one) $\leq i \leq m$, $PSV_i \geq PEV_i$, then an exception condition is raised: *data exception — invalid period value (22020)*.

3) Let n be the implementation-defined (IL055) maximum value of <timestamp precision>.

4) Case:

a) If <period predicate> immediately contains <period contains predicate>, then:

i) Case:

1) If the <period predicand> immediately contained in <period predicand 1> specifies PERIOD, then let *PSV1* be <period start value> and let *PEV1* be <period end value>.

2) If the <period predicand> immediately contained in <period predicand 1> immediately contains a <period reference> *PR*, then let *PD* be the period descriptor of the period identified by *PR*. Let *PN* be the name of the period included in *PD*. Let *PSV1* be the name of the *PN* period start column included in *PD* and let *PEV1* be the name of the *PN* period end column included in *PD*.

ii) Let *SV1* be the result of `CAST(`*PSV1*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`. Let *EV1* be the result of `CAST(`*PEV1*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`.

iii) Case:

1) If the <period or point-in-time predicand> immediately contains a <datetime value expression>, then let *DVE* be that <datetime value expression>.

2) If the <period predicand> immediately contained in <period or point-in-time predicand> specifies PERIOD, then let *PSV2* be <period start value> and let *PEV2* be <period end value>.

3) If the <period predicand> immediately contained in <period or point-in-time predicand> immediately contains a <period reference> *PR*, then let *PD* be the period descriptor of the period identified by *PR*. Let *PN* be the name of the period included in *PD*. Let *PSV2* be the name of the *PN* period start column included in *PD* and let *PEV2* be the name of the *PN* period end column included in *PD*.

iv) Case:

1) If the <period or point-in-time predicand> immediately contains a <datetime value expression>, then let *VE* be the result of `CAST(`*DVE*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`.

2) Otherwise, let *SV2* be the result of `CAST(`*PSV2*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`. Let *EV2* be the result of `CAST(`*PEV2*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`.

v) Case:

1) If the <period or point-in-time predicand> immediately contains a <datetime value expression> and *VE* is the null value, then the result of the <period contains predicate> is *Unknown*.

2) If the <period or point-in-time predicand> immediately contains a <datetime value expression> and $SV1 \leq VE$ and $EV1 > VE$, then the result of the <period contains predicate> is *True*.

3) If the <period or point-in-time predicand> does not immediately contain a <datetime value expression> and $SV1 \leq SV2$ and $EV1 \geq EV2$, then the result of the <period contains predicate> is *True*.

4) Otherwise, the result of the <period contains predicate> is *False*.

b) Otherwise,

i) Case:

1) If the <period predicand> immediately contained in <period predicand 1> specifies PERIOD, then let *PSV1* be <period start value> and let *PEV1* be <period end value>.

2) If the <period predicand> immediately contained in <period predicand 1> immediately contains a <period reference> *PR*, then let *PD* be the period descriptor of the period identified by *PR*, let *PN* be the name of the period included in *PD*, let *PSV1* be the name of the *PN* period start column included in *PD*, and let *PEV1* be the name of the *PN* period end column included in *PD*.

ii) Case:

1) If the <period predicand> immediately contained in <period predicand 2> specifies PERIOD, then let *PSV2* be <period start value> and let *PEV2* be <period end value>.

2) If the <period predicand> immediately contained in <period predicand 2> immediately contains a <period reference> *PR*, then let *PD* be the period descriptor of the period identified by *PR*, let *PN* be the name of the period included in *PD*, let *PSV2* be the name of the *PN* period start column included in *PD*, and let *PEV2* be the name of the *PN* period end column included in *PD*.

iii) Let *SV1* be the result of `CAST(`*PSV1*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`. Let *EV1* be the result of `CAST(`*PEV1*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`. Let *SV2* be the result of `CAST(`*PSV2*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`. Let *EV2* be the result of `CAST(`*PEV2*` AS TIMESTAMP(`*n*`) WITH TIME ZONE)`.

iv) If <period predicate> immediately contains <period overlaps predicate>, then

Case:

1) If *SV1* < *EV2* and *EV1* > *SV2*, then the result of the <period overlaps predicate> is *True*.

2) Otherwise, the result of the <period overlaps predicate> is *False*.

v) If <period predicate> immediately contains <period equals predicate>, then

Case:

1) If *SV1* = *SV2* and *EV1* = *EV2*, then the result of the <period equals predicate> is *True*.

2) Otherwise, the result of the <period equals predicate> is *False*.

vi) If <period predicate> immediately contains <period precedes predicate>, then

Case:

1) If *EV1* ≤ *SV2*, then the result of the <period precedes predicate> is *True*.

2) Otherwise, the result of the <period precedes predicate> is *False*.

vii) If <period predicate> immediately contains <period succeeds predicate>, then

Case:

1) If *SV1* ≥ *EV2*, then the result of the <period succeeds predicate> is *True*.

2) Otherwise, the result of the <period succeeds predicate> is *False*.

viii) If <period predicate> immediately contains <period immediately precedes predicate>, then

Case:

1) If *EV1* = *SV2*, then the result of the <period immediately precedes predicate> is *True*.

2) Otherwise, the result of the <period immediately precedes predicate> is *False*.

ix) If <period predicate> immediately contains <period immediately succeeds predicate>, then

Case:

1) If *SV1* = *EV2*, then the result of the <period immediately succeeds predicate> is *True*.

2) Otherwise, the result of the <period immediately succeeds predicate> is *False*.

Conformance Rules

1) Without Feature T502, "Period predicates", conforming SQL language shall not contain a <period predicate>.

8.21 <search condition>

Function

Specify a condition that is *True*, *False*, or *Unknown*, depending on the value of a <boolean value expression>.

Format

```
<search condition> ::=
  <boolean value expression>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) The result of the <search condition> is the result of the <boolean value expression>.

Conformance Rules

None.

8.22 <JSON predicate>

Function

Specify a test of whether a given value is a JSON text.

Format

```
<JSON predicate> ::=
  <JSON predicate input> [ <JSON input clause> ]
      IS [ NOT ] JSON
      [ <JSON predicate type constraint> ]
      [ <JSON key uniqueness constraint> ]

<JSON predicate input> ::=
    <string value expression>
  | <JSON value expression>

<JSON predicate type constraint> ::=
    VALUE
  | ARRAY
  | OBJECT
  | SCALAR

<JSON key uniqueness constraint> ::=
    WITH UNIQUE [ KEYS ]
  | WITHOUT UNIQUE [ KEYS ]
```

Syntax Rules

1) If <JSON input clause> is not specified, then FORMAT JSON is implicit.

2) If <JSON predicate type constraint> is not specified, then VALUE is implicit.

3) If <JSON key uniqueness constraint> is not specified, then WITHOUT UNIQUE KEYS is implicit.

4) Let *JPI* be the <JSON predicate input>, *FO* be the explicit or implicit <JSON input clause>, *JTYPE* be the explicit or implicit <JSON predicate type constraint>, and *JPUC* be the explicit or implicit <JSON key uniqueness constraint>.

   JPI FO IS NOT JSON *JTYPE JPUC*

   is equivalent to

   NOT (*JPI FO* IS JSON *JTYPE JPUC*)

Access Rules

None.

General Rules

1) Let *SVE* be the value of *JPI*.

2) Case:

a) If *SVE* is the null value, then the value of the <JSON predicate>

```
SVE FO IS JSON
```

is *Unknown*, and no further General Rules of this Subclause are applied.

b) If the declared type of *JPI* is JSON, then let *SJI* be *SVE* and let *ST* be the condition *successful completion (00000)*.

c) Otherwise, the General Rules of Subclause 9.42, "Parsing JSON text", are applied with *SVE* as *JSON TEXT*, *FO* as *FORMAT OPTION*, and *JPUC* as *UNIQUENESS CONSTRAINT*; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

3) The value of the <JSON predicate>

```
SVE FO IS JSON
```

is

Case:

a) If *ST* is not *successful completion (00000)*, then *False*.

b) If *JTYPE* is ARRAY and *SJI* is not an SQL/JSON array, then *False*.

c) If *JTYPE* is OBJECT and *SJI* is not an SQL/JSON object, then *False*.

d) If *JTYPE* is SCALAR and *SJI* is either an SQL/JSON array or an SQL/JSON object, then *False*.

e) Otherwise, *True*.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON predicate> that simply contains a <JSON value expression>.

2) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON predicate> that simply contains a <string value expression>.

3) Without Feature T821, "Basic SQL/JSON query operators", conforming SQL language shall not contain <JSON predicate>.

4) Without Feature T822, "SQL/JSON: IS JSON WITH UNIQUE KEYS predicate", conforming SQL language shall not contain a <JSON predicate> that specifies a <JSON key uniqueness constraint>.

5) Without Feature T851, "SQL/JSON: optional keywords for default syntax", conforming SQL language shall not contain a <JSON predicate> that simply contains both a <JSON value expression> and FORMAT JSON.

8.23 <JSON exists predicate>

Function

Specify a test of whether a JSON path expression returns any SQL/JSON items.

Format

```
<JSON exists predicate> ::=
  JSON_EXISTS <left paren>
      <JSON API common syntax>
      [ <JSON exists error behavior> ON ERROR ]
  <right paren>

<JSON exists error behavior> ::=
    TRUE | FALSE | UNKNOWN | ERROR
```

Syntax Rules

1) If <JSON exists error behavior> is not specified, then FALSE ON ERROR is implicit.

Access Rules

None.

General Rules

1) If the value of the <JSON context item> simply contained in the <JSON API common syntax> *JACS* is the null value, then the result of <JSON exists predicate> is *Unknown* and no further General Rules of this Subclause are applied.

2) Let *JEEB* be the explicit or implicit <JSON exists error behavior>.

3) The General Rules of Subclause 9.47, "Processing <JSON API common syntax>", are applied with *JACS* as *JSON API COMMON SYNTAX*; let *ST* be the *STATUS* and let *SEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

4) The result of <JSON exists predicate> is

   Case:

   a) If *ST* is successful completion, then

      Case:

      i) If the length of *SEQ* is 0 (zero), then *False*.

      ii) Otherwise, *True*.

   b) Otherwise,

      Case:

      i) If *JEEB* is ERROR, then the exception condition *ST* is raised.

      ii) If *JEEB* is TRUE, then *True*.

iii) If *JEEB* is FALSE, then *False*.

iv) If *JEEB* is UNKNOWN, then *Unknown*.

Conformance Rules

1) Without Feature T821, "Basic SQL/JSON query operators", conforming SQL language shall not contain <JSON exists predicate>.

2) Without Feature T825, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON exists predicate> shall not contain <JSON exists error behavior>.

9 Additional common rules

This Clause is modified by Clause 8, "Additional common rules", in ISO/IEC 9075-4.
This Clause is modified by Clause 8, "Additional common rules", in ISO/IEC 9075-9.
This Clause is modified by Clause 7, "Additional common rules", in ISO/IEC 9075-10.
This Clause is modified by Clause 8, "Additional common rules", in ISO/IEC 9075-13.
This Clause is modified by Clause 10, "Additional common rules", in ISO/IEC 9075-14.
This Clause is modified by Clause 9, "Additional common rules", in ISO/IEC 9075-15.
This Clause is modified by Clause 9, "Additional common rules", in ISO/IEC 9075-16.

9.1 Retrieval assignment

This Subclause is modified by Subclause 8.1, "Retrieval assignment", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.1, "Retrieval assignment", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.1, "Retrieval assignment", in ISO/IEC 9075-15.

Function

Specify rules for assignments to targets that do not support null values or that support null values with indicator parameters (e.g., assigning SQL-data to host parameters or host variables).

Subclause Signature

```
"Retrieval assignment" [Syntax Rules] (
  Parameter: "TARGET",
  Parameter: "VALUE"
)
```

TARGET — a site that is the target of a retrieval assignment operation.

VALUE — a value that is to be assigned to TARGET.

```
"Retrieval assignment" [General Rules] (
  Parameter: "TARGET",
  Parameter: "VALUE"
)
```

TARGET — a site that is the target of a retrieval assignment operation.

VALUE — a value that is to be assigned to TARGET.

Syntax Rules

1) Let *T* be the *TARGET* and let *V* be the *VALUE* in an application of the Syntax Rules of this Subclause.

2) Let *TD* and *SD* be the declared types of *T* and *V*, respectively.

3) If *TD* is a numeric, Boolean, datetime, interval, JSON, or user-defined type, then either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

NOTE 357 — "Appropriate user-defined cast function" is defined in Subclause 4.13, "Data conversions".

4) If *TD* is binary string, then

   Case:

   a) If *T* is either a locator parameter of an external routine, a locator variable, or a host parameter that is a binary large object locator parameter, then *SD* shall be BINARY LARGE OBJECT and *SD* shall be assignable to *TD*.

   b) Otherwise, either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

5) If *TD* is character string, then

   Case:

   a) If *T* is either a locator parameter of an external routine, a locator variable, or a host parameter that is a character large object locator parameter, then *SD* shall be CHARACTER LARGE OBJECT and *SD* shall be assignable to *TD*.

   b) Otherwise, either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

6) If the declared type of *T* is a reference type, then the declared type of *V* shall be a reference type whose referenced type is a subtype of the referenced type of *T*.

7) If the declared type of *T* is a row type, then:

   a) The declared type of *V* shall be a row type.

   b) The degree of *V* shall be the same as the degree of *T*. Let *n* be that degree.

   c) Let TT_i, 1 (one) ≤ i ≤ n, be the declared type of the i-th field of *T*, let VT_i be the declared type of the i-th field of *V*, let $T1_i$ be a temporary site whose declared type is TT_i, and let $V1_i$ be an arbitrary expression whose declared type is VT_i. For each i, 1 (one) ≤ i ≤ n, the Syntax Rules of this Subclause are applied with $T1_i$ as *TARGET* and $V1_i$ as *VALUE*.

8) If the declared type of *T* is a collection type or a distinct type whose source type is a collection type, then:

   a) If the declared type of *T* is an array type or a distinct type whose source type is an array type *DTAT*, then the declared type of *V* shall be an array type or *DTAT*.

   b) 15 If the declared type of *T* is a multiset type or a distinct type whose source type is a multiset type *DTMT*, then the declared type of *V* shall be a multiset type or *DTMT*.

   c) Let *TT* be the element type of the declared type of *T*, let *VT* be the element type of the declared type of *V*, let *T1* be a temporary site whose declared type is *TT*, and let *V1* be an arbitrary expression whose declared type is *VT*. The Syntax Rules of this Subclause are applied with *T1* as *TARGET* and *V1* as *VALUE*.

9) 09 14 Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

1) 09 15 Let *T* be the *TARGET* and let *V* be the *VALUE* in an application of the General Rules of this Subclause.

2) If the declared type of *V* is not assignable to the declared type of *T*, then for the remaining General Rules of this Subclause *V* is effectively replaced by the result of evaluating the expression *UDCF*(*V*).

3) If *V* is the null value and *T* is a host parameter, then

   Case:

   a) If an indicator parameter is specified for *T*, then that indicator parameter is set to −1 (negative one).

   b) If no indicator parameter is specified for *T*, then an exception condition is raised: *data exception — null value, no indicator parameter (22002)*.

4) If *V* is the null value and *T* is a host variable, then

   Case:

   a) If an indicator variable is specified for *T*, then that indicator variable is set to −1 (negative one).

   b) If no indicator variable is specified for *T*, then an exception condition is raised: *data exception — null value, no indicator parameter (22002)*.

5) If *V* is not the null value, *T* is a host parameter, and *T* has an indicator parameter, then

   Case:

   a) If the declared type of *T* is character string or binary string and the length *M* in characters or octets, respectively, of *V* is greater than the length in characters or octets, respectively, of *T*, then the indicator parameter is set to *M*. If *M* exceeds the maximum value that the indicator parameter can contain, then an exception condition is raised: *data exception — indicator overflow (22022)*.

   b) Otherwise, the indicator parameter is set to 0 (zero).

6) If *V* is not the null value, *T* is a host variable, and *T* has an indicator variable, then

   Case:

   a) If the declared type of *T* is character string or binary string and the length in characters or octets, respectively, *M* of *V* is greater than the length in characters or octets, respectively, of *T*, then the indicator variable is set to *M*. If *M* exceeds the maximum value that the indicator variable can contain, then an exception condition is raised: *data exception — indicator overflow (22022)*.

   b) Otherwise, the indicator variable is set to 0 (zero).

7) If *V* is not the null value, then

   Case:

   a) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters of *V* is equal to *L*, then the value of *T* is set to *V*.

   b) If the declared type of *T* is fixed-length character string with length in characters *L*, and the length in characters of *V* is greater than *L*, then the value of *T* is set to the first *L* characters of *V* and a completion condition is raised: *warning — string data, right truncation (01004)*.

c) If the declared type of *T* is fixed-length character string with length in characters *L*, and the length in characters *M* of *V* is smaller than *L*, then the first *M* characters of *T* are set to *V*, and the last *L*–*M* characters of *T* are set to <space>s.

d) If the declared type of *T* is variable-length character string and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

e) If the declared type of *T* is variable-length character string and the length in characters of *V* is greater than the maximum length in characters *L* of *T*, then the value of *T* is set to the first *L* characters of *V*, then the length in characters of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation (01004)*.

f) If the declared type of *T* is a character large object type and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

g) If the declared type of *T* is a character large object type and the length in characters of *V* is greater than the maximum length in characters *L* of *T*, then the value of *T* is set to the first *L* characters of *V*, the length in characters of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation (01004)*.

h) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets of *V* is equal to *L*, then the value of *T* is set to *V*.

i) If the declared type of *T* is fixed-length binary string with length in octets *L*, and the length in octets of *V* is greater than *L*, then the value of *T* is set to the first *L* octets of *V* and a completion condition is raised: *warning — string data, right truncation (01004)*.

j) If the declared type of *T* is fixed-length binary string with length in octets *L*, and the length in octets *M* of *V* is less than *L*, then the first *M* octets of *T* are set to *V*, and the last *L*–*M* octets of *T* are set to X'00's.

k) If the declared type of *T* is variable-length binary string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

l) If the declared type of *T* is variable-length binary string and the length in octets of *V* is greater than the maximum length in octets *L* of *T*, then the value of *T* is set to the first *L* octets of *V*, then the length in octets of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation (01004)*.

m) If the declared type of *T* is binary large object string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

n) If the declared type of *T* is binary large object string and the length in octets of *V* is greater than the maximum length in octets *L* of *T*, then the value of *T* is set to the first *L* octets of *V*, the length in octets of *T* becomes *L*, and a completion condition is raised: *warning — string data, right truncation (01004)*.

o) If the declared type of *T* is numeric, then

Case:

i) If *V* is a value of the declared type of *T*, then the value of *T* is set to *V*.

ii) If a value of the declared type of *T* can be obtained from *V* by rounding or truncation, then the value of *T* is set to that value. If the declared type of *T* is exact numeric, then

it is implementation-defined (IA002) whether the approximation is obtained by rounding or by truncation.

iii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

p) If the declared type of *T* is Boolean, then the value of *T* is set to *V*.

q) If the declared type *DT* of *T* is datetime, then:

i) If only one of *DT* and the declared type of *V* is datetime with time zone, then *V* is effectively replaced by

```
CAST ( V AS DT )
```

ii) Case:

1) If *V* is a value of the declared type of *T*, then the value of *T* is set to *V*.

2) If a value of the declared type of *T* can be obtained from *V* by rounding or truncation, then the value of *T* is set to that value. It is implementation-defined (IA002) whether the approximation is obtained by rounding or truncation.

3) Otherwise, an exception condition is raised: *data exception — datetime field overflow (22008)*.

r) If the declared type of *T* is interval, then

Case:

i) If *V* is a value of the declared type of *T*, then the value of *T* is set to *V*.

ii) If a value of the declared type of *T* can be obtained from *V* by rounding or truncation, then the value of *T* is set to that value. It is implementation-defined (IA002) whether the approximation is obtained by rounding or by truncation.

iii) Otherwise, an exception condition is raised: *data exception — interval field overflow (22015)*.

s) If the declared type of *T* is JSON, then the value of *T* is set to *V*.

t) If the declared type of *T* is a row type, then:

i) Let *n* be the degree of *T*.

ii) For *i* ranging from 1 (one) to *n*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*.

u) If the declared type of *T* is a reference type, then the value of *T* is set to *V*.

v) [15] If the declared type of *T* is an array type or a distinct type whose source type is an array type, then

Case:

i) If the maximum cardinality *L* of *T* is equal to the cardinality *M* of *V*, then for *i* ranging from 1 (one) to *L*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*.

ii) If the maximum cardinality *L* of *T* is smaller than the cardinality *M* of *V*, then for *i* ranging from 1 (one) to *L*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*; a completion condition is raised: *warning — array data, right truncation (0102F)*.

iii) If the maximum cardinality *L* of *T* is greater than the cardinality *M* of *V*, then for *i* ranging from 1 (one) to *M*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*. The cardinality of the value of *T* is *M*.

NOTE 358 — The maximum cardinality *L* of *T* is unchanged.

w) If the declared type of *T* is a multiset type or a distinct type whose source type is a multiset type, then the value of *T* is set to *V*.

x) 09 14 If the declared type of *T* is a user-defined type, then the value of *T* is set to *V*.

8) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.2 Store assignment

This Subclause is modified by Subclause 8.2, "Store assignment", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.2, "Store assignment", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.2, "Store assignment", in ISO/IEC 9075-15.

Function

Specify rules for assignments where the target permits null without the use of indicator parameters or indicator variables, such as storing SQL-data or setting the value of SQL parameters.

Subclause Signature

```
"Store assignment" [Syntax Rules] (
  Parameter: "TARGET",
  Parameter: "VALUE"
)
```

TARGET — a site that is the target of a store assignment operation.

VALUE — a value that is to be assigned to TARGET.

```
"Store assignment" [General Rules] (
  Parameter: "TARGET",
  Parameter: "VALUE"
)
```

TARGET — a site that is the target of a store assignment operation.

VALUE — a value that is to be assigned to TARGET.

Syntax Rules

1) Let *T* be the *TARGET* and let *V* be the *VALUE* in an application of the Syntax Rules of this Subclause.

2) Let *TD* and *SD* be the declared types of *T* and *V*, respectively.

3) If *TD* is a character string, binary string, numeric, Boolean, datetime, interval, JSON, or user-defined type, then either *SD* shall be assignable to *TD* or there shall exist an appropriate user-defined cast function *UDCF* from *SD* to *TD*.

NOTE 359 — "Appropriate user-defined cast function" is defined in Subclause 4.13, "Data conversions".

4) If the declared type of *T* is a reference type, then the declared type of *V* shall be a reference type whose referenced type is a subtype of the referenced type of *T*.

5) If the declared type of *T* is a row type, then:

   a) The declared type of *V* shall be a row type.

   b) The degree of *V* shall be the same as the degree of *T*. Let *n* be that degree.

   c) Let TT_i, 1 (one) ≤ *i* ≤ *n*, be the declared type of the *i*-th field of *T*, let VT_i be the declared type of the *i*-th field of *V*, let $T1_i$ be a temporary site whose declared type is TT_i, and let $V1_i$ be an arbitrary expression whose declared type is VT_i. For each *i*, 1 (one) ≤ *i* ≤ *n*, the Syntax Rules of this Subclause are applied with $T1_i$ as *TARGET* and $V1_i$ as *VALUE*.

6) If the declared type of *T* is a collection type or a distinct type whose source type is a collection type, then:

   a) If the declared type of *T* is an array type or a distinct type whose source type is an array type *DTAT*, then the declared type of *V* shall be an array type or *DTAT*.

   b) 15 If the declared type of *T* is a multiset type or a distinct type whose source type is a multiset type *DTMT*, then the declared type of *V* shall be a multiset type or *DTMT*.

   c) Let *TT* be the element type of the declared type of *T*, let *VT* be the element type of the declared type of *V*, let *T1* be a temporary site whose declared type is *TT*, and let *V1* be an arbitrary expression whose declared type is *VT*. The Syntax Rules of this Subclause are applied with *T1* as *TARGET* and *V1* as *VALUE*.

7) 09 14 Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

1) 14 Let *T* be the *TARGET* and let *V* be the *VALUE* in an application of the General Rules of this Subclause.

2) If the declared type of *V* is not assignable to the declared type of *T*, then for the remaining General Rules of this Subclause *V* is effectively replaced by the result of evaluating the expression *UDCF*(*V*).

3) Case:

   a) If *V* is the null value, then

      Case:

      i) If *V* is specified using NULL, then *T* is set to the null value.

      ii) If *V* is a host parameter and contains an indicator parameter, then

         Case:

         1) If the value of the indicator parameter is equal to −1 (negative one), then *T* is set to the null value.

         2) If the value of the indicator parameter is less than −1 (negative one), then an exception condition is raised: *data exception — invalid indicator parameter value (22010)*.

      iii) If *V* is a host variable and contains an indicator variable, then

         Case:

         1) If the value of the indicator variable is equal to −1 (negative one), then *T* is set to the null value.

         2) If the value of the indicator variable is less than −1 (negative one), then an exception condition is raised: *data exception — invalid indicator parameter value (22010)*.

      iv) Otherwise, *T* is set to the null value.

   b) Otherwise,

Case:

i) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters of *V* is equal to *L*, then the value of *T* is set to *V*.

ii) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters *M* of *V* is larger than *L*, then

Case:

1) If the rightmost *M*−*L* characters of *V* are all <space>s, then the value of *T* is set to the first *L* characters of *V*.

2) If one or more of the rightmost *M*−*L* characters of *V* are not <truncating whitespace> characters, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

iii) If the declared type of *T* is fixed-length character string with length in characters *L* and the length in characters *M* of *V* is less than *L*, then the first *M* characters of *T* are set to *V* and the last *L*−*M* characters of *T* are set to <space>s.

iv) If the declared type of *T* is variable-length character string and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

v) If the declared type of *T* is variable-length character string and the length in characters *M* of *V* is greater than the maximum length in characters *L* of *T*, then

Case:

1) If the rightmost *M*−*L* characters of *V* are all <space>s, then the value of *T* is set to the first *L* characters of *V* and the length in characters of *T* is set to *L*.

2) If one or more of the rightmost *M*−*L* characters of *V* are not <truncating whitespace> characters, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

vi) If the declared type of *T* is a character large object type and the length in characters *M* of *V* is not greater than the maximum length in characters of *T*, then the value of *T* is set to *V* and the length in characters of *T* is set to *M*.

vii) If the declared type of *T* is a character large object type and the length in characters *M* of *V* is greater than the maximum length in characters *L* of *T*, then

Case:

1) If the rightmost *M*−*L* characters of *V* are all <space>s, then the value of *T* is set to the first *L* characters of *V* and the length in characters of *T* is set to *L*.

2) If one or more of the rightmost *M*−*L* characters of *V* are not <truncating whitespace> characters, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

viii) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets of *V* is equal to *L*, then the value of *T* is set to *V*.

ix) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets *M* of *V* is larger than *L*, then

Case:

1) If the rightmost *M*−*L* octets of *V* are all equal to X'00', then the value of *T* is set to the first *L* octets of *V*.

2) If one or more of the rightmost *M*−*L* octets of *V* are not equal to X'00', then an exception condition is raised: *data exception — string data, right truncation (22001)*.

x) If the declared type of *T* is fixed-length binary string with length in octets *L* and the length in octets *M* of *V* is less than *L*, then the first *M* octets of *T* are set to *V* and the last *L*−*M* octets of *T* are set to X'00's.

xi) If the declared type of *T* is variable-length binary string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

xii) If the declared type of *T* is variable-length binary string and the length in octets *M* of *V* is greater than the maximum length in octets *L* of *T*, then

Case:

1) If the rightmost *M*−*L* octets of *V* are all equal to X'00', then the value of *T* is set to the first *L* octets of *V* and the length in octets of *T* is set to *L*.

2) If one or more of the rightmost *M*−*L* octets of *V* are not equal to X'00', then an exception condition is raised: *data exception — string data, right truncation (22001)*.

xiii) If the declared type of *T* is binary large object string and the length in octets *M* of *V* is not greater than the maximum length in octets of *T*, then the value of *T* is set to *V* and the length in octets of *T* is set to *M*.

xiv) If the declared type of *T* is binary large object string and the length in octets *M* of *V* is greater than the maximum length in octets *L* of *T*, then

Case:

1) If the rightmost *M*−*L* octets of *V* are all equal to X'00', then the value of *T* is set to the first *L* octets of *V* and the length in octets of *T* is set to *L*.

2) If one or more of the rightmost *M*−*L* octets of *V* are not equal to X'00', then an exception condition is raised: *data exception — string data, right truncation (22001)*.

xv) If the declared type of *T* is numeric, then

Case:

1) If *V* is a value of the declared type of *T*, then the value of *T* is set to *V*.

2) If a value of the declared type of *T* can be obtained from *V* by rounding or truncation, then the value of *T* is set to that value. If the declared type of *T* is exact numeric, then it is implementation-defined (IA002) whether the approximation is obtained by rounding or by truncation.

3) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

xvi) If the declared type *DT* of *T* is datetime, then:

1) If only one of *DT* and the declared type of *V* is datetime with time zone, then *V* is effectively replaced by

CAST (*V* AS *DT*)

2) Case:

A) If *V* is a value of the declared type of *T*, then the value of *T* is set to *V*.

B) If a value of the declared type of *T* can be obtained from *V* by rounding or truncation, then the value of *T* is set to that value. It is implementation-defined (IA002) whether the approximation is obtained by rounding or truncation.

C) Otherwise, an exception condition is raised: *data exception — datetime field overflow (22008)*.

xvii) If the declared type of *T* is interval, then

Case:

1) If *V* is a value of the declared type of *T*, then the value of *T* is set to *V*.

2) If a value of the declared type of *T* can be obtained from *V* by rounding or truncation, then the value of *T* is set to that value. It is implementation-defined (IA002) whether the approximation is obtained by rounding or by truncation.

3) Otherwise, an exception condition is raised: *data exception — interval field overflow (22015)*.

xviii) If the declared type of *T* is Boolean, then the value of *T* is set to *V*.

xix) If the declared type of *T* is JSON, then the value of *T* is set to *V*.

xx) If the declared type of *T* is a row type, then:

1) Let *n* be the degree of *T*.

2) For *i* ranging from 1 (one) to *n*, the General Rules of this Subclause are applied with T_i as *TARGET* and V_i as *VALUE*.

xxi) If the declared type of *T* is a reference type, then the value of *T* is set to *V*.

xxii) [15] If the declared type of *T* is an array type or a distinct type whose source type is an array type, then

Case:

1) If the maximum cardinality *L* of *T* is equal to the cardinality *M* of *V*, then for *i* ranging from 1 (one) to *L*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*.

2) If the maximum cardinality *L* of *T* is smaller than the cardinality *M* of *V*, then

Case:

A) If the rightmost *M*–*L* elements of *V* are all null, then for *i* ranging from 1 (one) to *L*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*.

B) If one or more of the rightmost *M*–*L* elements of *V* are not the null value, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

3) If the maximum cardinality *L* of *T* is greater than the cardinality *M* of *V*, then for *i* ranging from 1 (one) to *M*, the General Rules of this Subclause are applied with the *i*-th element of *T* as *TARGET* and the *i*-th element of *V* as *VALUE*. The cardinality of the value of *T* is set to *M*.

NOTE 360 — The maximum cardinality *L* of *T* is unchanged.

xxiii) If the declared type of *T* is a multiset type or a distinct type whose source type is a multiset type, then the value of *T* is set to *V*.

xxiv) 09 14 If the declared type of *T* is a user-defined type, then the value of *T* is set to *V*.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.3 Passing a value from a host language to the SQL-server

This Subclause is modified by Subclause 9.3, "Passing a value from a host language to the SQL-server", in ISO/IEC 9075-15.

Function

Specify rules to pass a value from a host language to the SQL-server.

Subclause Signature

```
"Passing a value from a host language to the SQL-server" [General Rules] (
  Parameter: "LANGUAGE",
  Parameter: "SQL TYPE",
  Parameter: "HOST VALUE"
) Returns: "SQL VALUE"
```

LANGUAGE — the name of a host programming language (ADA, C, COBOL, FORTRAN, M, PASCAL, PLI).

SQL TYPE — a specification of an SQL data type, possibly including a <locator indication>.

HOST VALUE — a value of a host language type.

SQL VALUE — the SQL TYPE representation of HOST VALUE, that is the result of invoking this subclause using this signature.

Syntax Rules

None.

Access Rules

None.

General Rules

1) [15] Let *LANG* be the *LANGUAGE*, let *DT* be the *SQL TYPE*, and let *PI* be the *HOST VALUE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *SQL VALUE*.

   NOTE 361 — *DT* is either a <host parameter data type> or a <parameter type>.

2) Let *SV* be a site whose declared type is the <data type> contained in *SQL TYPE*.

3) Depending on whether *LANG* specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences table* be Table 21, "Data type correspondences for Ada", Table 22, "Data type correspondences for C", Table 23, "Data type correspondences for COBOL", Table 24, "Data type correspondences for Fortran", Table 25, "Data type correspondences for M", Table 26, "Data type correspondences for Pascal", or Table 27, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

4) Let *HDT* be

Case:

a) If *DT* contains <locator indication>, then

Case:

i) If *LANG* is ADA, then `Interfaces.SQL.INT`.

ii) If *LANG* is C, then `unsigned long`.

iii) If *LANG* is COBOL, then `PIC S9(9) USAGE IS BINARY`.

iv) If *LANG* is FORTRAN, then `INTEGER`.

v) If *LANG* is M, then character.

vi) If *LANG* is PASCAL, then `INTEGER`.

vii) If *LANG* is PLI, then `FIXED BINARY(31)`.

b) Otherwise, the host language data type listed in the host data type column of the row in the operative data type correspondences table whose value in the SQL data type column is *DT*.

NOTE 362 — The host language data type of *PI* is *HDT*.

5) Case:

a) If *DT* contains <locator indication>, then

Case:

i) If *PI* is an invalid locator, then an exception condition is raised: *locator exception — invalid specification (0F001)* and no further General Rules of this Subclause are applied.

ii) [15] Otherwise, let the value of *SV* be the binary large object string value, the large object character string value, the array value, the multiset value, or the user-defined type value corresponding to *PI*.

b) If *DT* identifies a CHARACTER(*L*) or CHARACTER VARYING(*L*) data type and *LANG* is C, then let the value of *SV* be *PI* implicitly treated as a character string type value in the character set of *DT* in which the octets of *PI* are the corresponding octets of *SV*.

Let *NC* be the implementation-defined (IV030) null character that terminates a C character string.

Case:

i) If *DT* identifies a CHARACTER(*L*) data type and some C character preceding the least significant C character of *PI* contains *NC*, then the characters of *SV* from the first occurrence of *NC* through the end of *SV* are set to <space>s.

ii) If *DT* identifies a CHARACTER VARYING(*L*) data type, then the length in characters of *SV* is set to the number of characters of *PI* that precede the first occurrence of *NC* in *PI*.

iii) If the least significant C character of the value *PI* does not contain *NC*, then an exception condition is raised: *data exception — unterminated C string (22024)*.

iv) Otherwise, that least significant C character does not correspond to any character in *PI* and is ignored.

NOTE 363 — The phrase "implementation-defined (IV030) null character that terminates a C character string" implies one or more octets all of whose bits are zero and whose number is equal to the number of octets in the largest character of the character set of *DT*. The number of such octets depends on the definition of the character set and is thus implementation-defined (IV184).

c) If exactly one of the following is true:

i) *DT* identifies a CHARACTER(*L*) data type and *LANG* is either ADA, COBOL, FORTRAN, or PASCAL.

ii) *DT* identifies a CHARACTER VARYING(*L*) data type and *LANG* is M.

iii) *DT* identifies a CHARACTER(*L*) data type or CHARACTER VARYING(*L*) data type and *LANG* is PLI.

then let the value of *SV* be *PI*, implicitly treated as a character string type value in the character set of *DT* in which the octets of *PI* are the corresponding octets of *SV*.

d) If *DT* identifies INT, DEC, or REAL and the *LANG* is M, then let *TEMP* be an SQL character string equivalent to *PI* and let the value of *SV* be the value of:

```
CAST ( TEMP AS DT )
```

e) If *DT* identifies a BOOLEAN type, then:

Case:

i) If *LANG* is ADA, then:

1) If the value of *PI* is False, then let the value of *SV* be *False*.

2) Otherwise, let the value of *SV* be *True*.

ii) If *LANG* is C, then:

1) If the value of *PI* is 0 (zero), then let the value of *SV* be *False*.

2) Otherwise, let the value of *SV* be *True*.

iii) If *LANG* is COBOL, then:

1) If the value of *PI* is 'F', then let the value of *SV* be *False*.

2) Otherwise, let the value of *SV* be *True*.

iv) If *LANG* is FORTRAN, then:

1) If the value of *PI* is `.FALSE.`, then let the value of *SV* be *False*.

2) Otherwise, let the value of *SV* be *True*.

v) If *LANG* is PASCAL, then: let the value of *SV* be *PI*.

NOTE 364 — Pascal has a Boolean-type whose values are *True* and *False*.

vi) If *LANG* is PLI, then:

1) If the value of *PI* is '0'B, then let the value of *SV* be *False*.

2) Otherwise, let the value of *SV* be *True*.

f) If *DT* identifies a CHARACTER LARGE OBJECT or BINARY LARGE OBJECT type, then

Case:

i) If *DT* identifies a CHARACTER LARGE OBJECT type, then let *CLU* be the explicit or implicit <char length units> contained in *DT*. Let *SV* be

Case:

1) If *CLU* is CHARACTERS, then a large object character string whose length in characters *LEN* is the value of the length portion of *PI* and whose value is the first *LEN* characters in the data portion of *PI*.

2) Otherwise, a large object character string whose length in octets *LEN* is the value of the length portion of *PI* and whose value is the first *LEN* octets in the data portion of *PI*.

ii) If *DT* identifies a BINARY LARGE OBJECT type, then let *SV* be a binary large object string whose length *LEN* is the value of the length portion of *PI* and whose value is the first *LEN* octets in the data portion of *PI*.

NOTE 365 — The length portion and the data portion of *PI* are defined in footnotes to the operative data type correspondences table.

g) If *DT* identifies a BINARY(*L*) or BINARY VARYING(*L*) data type and *LANG* is ADA, then

Case:

i) If *DT* identifies a BINARY(*L*) data type, then let the value of *SV* be a binary string whose octets are the corresponding octets of *PI*.

ii) If *DT* identifies a BINARY VARYING(*L*) data type, then let the value of *SV* be a binary string whose length *LEN* is the value of the length portion of *PI* and whose value is the first *LEN* octets in the data portion of *PI*.

h) If *DT* identifies a BINARY(*L*) or BINARY VARYING(*L*) data type and *LANG* is C, then

Case:

i) If *DT* identifies a BINARY(*L*) data type, then let the value of *SV* be a binary string of *L* octets whose octets are the corresponding octets of *PI*.

ii) If *DT* identifies a BINARY VARYING(*L*) data type, then let the value of *SV* be a binary string whose length *LEN* is the value of the length portion of *PI* and whose value is the first *LEN* octets in the data portion of *PI*.

i) If exactly one of the following is true:

i) *DT* identifies a BINARY(*L*) data type and *LANG* is COBOL, FORTRAN, or PASCAL.

ii) *DT* identifies a BINARY(*L*) data type or BINARY VARYING(*L*) data type and *LANG* is PLI.

then let the value of *SV* be a binary string whose octets are the corresponding octets of *PI*.

j) Otherwise, let the value of *SV* be the value *PI*.

6) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *SV* as *SQL VALUE*.

Conformance Rules

None.

9.4 Passing a value from the SQL-server to a host language

This Subclause is modified by Subclause 9.4, "Passing a value from the SQL-server to a host language", in ISO/IEC 9075-15.

Function

Specify rules to pass a value from the SQL-server to a host language.

Subclause Signature

```
"Passing a value from the SQL-server to a host language" [General Rules] (
  Parameter: "LANGUAGE",
  Parameter: "SQL TYPE",
  Parameter: "SQL VALUE"
) Returns: "HOST VALUE"
```

LANGUAGE — the name of a host programming language (ADA, C, COBOL, FORTRAN, M, PASCAL, PLI).

SQL TYPE — a specification of an SQL data type, possibly including a <locator indication>.

SQL VALUE — a value of type SQL TYPE.

HOST VALUE — a host language data type representation of SQL VALUE that is the result of invoking this subclause using this signature.

Syntax Rules

None.

Access Rules

None.

General Rules

1) [15]Let *LANG* be the *LANGUAGE*, let *DT* be the *SQL TYPE*, and let *SV* be the *SQL VALUE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *HOST VALUE*.

2) Depending on whether *LANG* specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences table* be Table 21, "Data type correspondences for Ada", Table 22, "Data type correspondences for C", Table 23, "Data type correspondences for COBOL", Table 24, "Data type correspondences for Fortran", Table 25, "Data type correspondences for M", Table 26, "Data type correspondences for Pascal", or Table 27, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

3) Let *HDT* be

Case:

a) If *DT* contains <locator indication>, then

Case:

i) If *LANG* is ADA, then `Interfaces.SQL.INT`.

ii) If *LANG* is C, then `unsigned long`.

iii) If *LANG* is COBOL, then `PIC S9(9) USAGE IS BINARY`.

iv) If *LANG* is FORTRAN, then `INTEGER`.

v) If *LANG* is M, then character.

vi) If *LANG* is PASCAL, then `INTEGER`.

vii) If *LANG* is PLI, then `FIXED BINARY(31)`.

b) Otherwise, the host language data type listed in the host data type column of the row in the operative data type correspondences table whose value in the SQL data type column is *DT*.

NOTE 366 — The host language data type of *PI* is *HDT*.

4) Let *PI* be a site of host language data type *HDT*.

5) Case:

a) [15] If *DT* contains <locator indication>, then let the value of *PI* be the binary large object locator value, the character large object locator value, the array locator value, the multiset locator value, or the user-defined type locator value that uniquely identifies *SV*.

b) If *DT* identifies CHARACTER(*L*) or CHARACTER VARYING(*L*) data types and *LANG* is C, then let *CL* be *k* greater than the maximum possible length in octets of *DT*, where *k* is the size in octets of the largest character in the character set of *DT*. Let the value of *PI* be a C character string in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL* − *k* octets, concatenated with a single implementation-defined (IV030) null character that terminates a C character string.

NOTE 367 — The phrase "implementation-defined (IV030) null character that terminates a C character string" implies *k* octets, all of whose bits are zero.

c) If *DT* identifies a CHARACTER(*L*) data type and *LANG* is either ADA, COBOL, FORTRAN, or PASCAL, then let *CL* be the maximum possible length in octets of *DT*. Let the value of *PI* be a value of host language data type *HDT* in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL* octets.

d) If *DT* identifies a CHARACTER VARYING(*L*) data type and *LANG* is M, then let *CL* be the maximum possible length in octets of *DT*. Let the value of *PI* be an M character string in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL* octets.

e) If *DT* identifies a CHARACTER(*L*) or CHARACTER VARYING(*L*) data types and *LANG* is PLI, then let *CL* be the maximum possible length in octets of *DT*.

Case:

i) If *DT* identifies CHARACTER(*L*), then let the value of *PI* be a PL/I character string in which octets of the value are the corresponding octets of *SV*.

ii) Otherwise, let the value of *PI* be a PL/I character string in which octets of the value are the corresponding octets of *SV*, padded on the right with <space>s as necessary to reach the length *CL*.

f) If *DT* identifies INT, DEC, or REAL and *LANG* is M, then let *PI* be an M character string whose octets are the corresponding octets of the value

```
CAST ( SV AS CHARACTER VARYING(ML) )
```

where *ML* is the implementation-defined (IL006) maximum length of variable-length character strings.

g) If *DT* identifies a BOOLEAN type, then

Case:

i) If *LANG* is ADA, then:

1) If *SV* is *False*, then let the value of *PI* be the value False.

2) If *SV* is *True*, then let the value of *PI* be the value True.

ii) If *LANG* is C, then:

1) If *SV* is *False*, then let the value of *PI* be the value 0 (zero).

2) If *SV* is *True*, then let the value of *PI* be the value 1 (one).

iii) If *LANG* is COBOL, then:

1) If *SV* is *False*, then let the value of *PI* be the value 'F'.

2) If *SV* is *True*, then let the value of *PI* be the value 'T'.

iv) If *LANG* is FORTRAN, then:

1) If *SV* is *False*, then let the value of *PI* be the value `.FALSE.`.

2) If *SV* is *True*, then let the value of *PI* be the value `.TRUE.`.

v) If *LANG* is PASCAL, then: let the value of *PI* be *SV*.

NOTE 368 — Pascal has a Boolean-type, whose values are *True* and *False*.

vi) If *LANG* is PLI, then:

1) If *SV* is *False*, then let the value of *PI* be the value '0'B.

2) If *SV* is *True*, then let the value of *PI* be the value '1'B.

h) If *DT* identifies a CHARACTER LARGE OBJECT or BINARY LARGE OBJECT type, then

Case:

i) If *DT* identifies a CHARACTER LARGE OBJECT type, then let *CLU* be the explicit or implicit <char length units> contained in *DT*.

Case:

1) If *CLU* is CHARACTERS, then let the length portion of *PI* be the number of characters *LEN* in *SV*, and let the first *LEN* characters of the data portion of *PI* be a character string equivalent to *SV*.

2) Otherwise, let the length portion of *PI* be the number of octets *LEN* in *SV*, and let the first *LEN* octets of the data portion of *PI* be a character string equivalent to *SV*.

ii) If *DT* identifies a BINARY LARGE OBJECT type, then let the length portion of *PI* be the number of octets *LEN* in *SV*, and let the first *LEN* octets of the data portion of *PI* be a binary string equivalent to *SV*.

NOTE 369 — The length portion and data portion of *PI* are defined in a footnote in the operative data type correspondences table.

i) If *DT* identifies a BINARY(*L*) data type, then let *PI* be a binary string of length *L* in the host language data type *HDT*, in which octets of *PI* are the corresponding octets of *SV*.

j) If *DT* identifies a BINARY VARYING(*L*) data type and *LANG* is ADA or C, then let *LEN* be the value OCTET_LENGTH(*SV*), let the length portion of *PI* be *LEN*, and let the first *LEN* octets of the data portion of *PI* be the value *SV*.

k) If *DT* identifies a BINARY VARYING(*L*) data type and *LANG* is PLI, then let *PI* be a binary string of host language data type *HDT* in which octets of the value are the corresponding octets of *SV*, padded on the right with X'00's as necessary to reach the length *L*.

l) Otherwise, let the value of *PI* be *SV*. If *LANG* is ADA and no value has been assigned to *PI*, then an implementation-dependent (UV093) value is assigned to *PI*.

6) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *PI* as *HOST VALUE*.

Conformance Rules

None.

9.5 Result of data type combinations

This Subclause is modified by Subclause 8.3, "Result of data type combinations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.3, "Result of data type combinations", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.5, "Result of data type combinations", in ISO/IEC 9075-15.

Function

Specify the declared type of the result of certain combinations of values of compatible data types, such as <case expression>s, <collection value expression>s, or a column in the result of a <query expression>.

Subclause Signature

```
"Result of data type combinations" [Syntax Rules] (
  Parameter: "DTSET"
) Returns: "RESTYPE"
```

DTSET — a set of SQL data types.

RESTYPE — an SQL data type that can be used to represent values of all SQL data types contained in DTSET.

Syntax Rules

1) Let *IDTS* be the *DTSET* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *RESTYPE*.

2) Let *DTS* be the set of data types in *IDTS* excluding any data types that are undefined. If the cardinality of *DTS* is 0 (zero), then the declared type of the result is undefined and no further Rules of this Subclause are evaluated.

NOTE 370 — The notion of "undefined data type" is defined in Subclause 20.7, "<prepare statement>".

3) Case:

a) If any of the data types in *DTS* is character string, then:

i) All data types in *DTS* shall be character string, and all of them shall have the same character repertoire. The character set of the result is the character set of the data type in *DTS* that has the character encoding form with the highest precedence.

ii) The collation derivation and declared type collation of the result are determined as follows.

Case:

1) If some data type in *DTS* has an *explicit* collation derivation and declared type collation *EC1*, then every data type in *DTS* that has an *explicit* collation derivation shall have a declared type collation that is *EC1*. The collation derivation is *explicit* and the collation is *EC1*.

2) If every data type in *DTS* has an *implicit* collation derivation, then

Case:

A) If every data type in *DTS* has the same declared type collation *IC1*, then the collation derivation is *implicit* and the declared type collation is *IC1*.

B) Otherwise, the collation derivation is *none*.

3) Otherwise, the collation derivation is *none*.

iii) Case:

1) If any of the data types in *DTS* is a character large object type, then the declared type of the result is a character large object type with maximum length in characters equal to the maximum of the lengths in characters and maximum lengths in characters of the data types in *DTS*.

2) If any of the data types in *DTS* is variable-length character string, then the declared type of the result is variable-length character string with maximum length in characters equal to the maximum of the lengths in characters and maximum lengths in characters of the data types in *DTS*.

3) Otherwise, the declared type of the result is fixed-length character string with length in characters equal to the maximum of the lengths in characters of the data types in *DTS*.

b) If any of the data types in *DTS* is binary string, then

Case:

i) If any of the data types in *DTS* is a binary large object type, then the declared type of the result is a binary large object type with maximum length in octets equal to the maximum of the lengths in octets and the maximum lengths in octets of the data types in *DTS*.

ii) If any of the data types in *DTS* is variable-length binary string, then the declared type of the result is variable-length binary string with maximum length in octets equal to the maximum of the lengths in octets and the maximum lengths in octets of the data types in *DTS*.

iii) Otherwise, the declared type of the result is fixed-length binary string with length in octets equal to the maximum of the lengths in octets of the data types in *DTS*.

c) If any data type in *DTS* is numeric, then:

i) Each data type in *DTS* shall be numeric.

ii) The declared type of the result is

Case:

1) If any data type in *DTS* is the decimal floating-point type, then the decimal floating-point type with precision equal to the maximum of the precisions of the decimal floating-point types in *DTS*.

2) If any data type in *DTS* is approximate numeric, then approximate numeric with implementation-defined (IV159) precision.

3) Otherwise, exact numeric with implementation-defined (IV160) precision and with scale equal to the maximum of the scales of the data types in *DTS*.

d) If some data type in *DTS* is a datetime data type, then every data type in *DTS* shall be a datetime data type having the same datetime fields. The declared type of the result is a datetime data type having the same datetime fields, whose fractional seconds precision is the largest of the fractional seconds precisions in *DTS*. If some data type in *DTS* has a time zone displacement value, then the result has a time zone displacement value; otherwise, the result does not have a time zone displacement value.

e) If any data type in *DTS* is interval, then each data type in *DTS* shall be interval. If the precision of any data type in *DTS* specifies YEAR or MONTH, then the precision of each data type shall specify only YEAR or MONTH. If the precision of any data type in *DTS* specifies DAY, HOUR, MINUTE, or SECOND(*N*), then the precision of no data type of *DTS* shall specify the <primary datetime field>s YEAR and MONTH. The declared type of the result is interval with precision "*S* TO *E*", where *S* and *E* are the most significant of the <start field>s and the least significant of the <end field>s of the data types in *DTS*, respectively.

f) 09 14 If any data type in *DTS* is Boolean, then each data type in *DTS* shall be Boolean. The declared type of the result is Boolean.

g) If any data type in *DTS* is a row type, then each data type in *DTS* shall be a row type with the same degree and the data type of each field in the same ordinal position of every row type shall be comparable. The declared type of the result is a row type defined by an ordered sequence of (<field name>, data type) pairs FD_i, where data type is the data type resulting from the application of this Subclause to the set of data types of fields in the same ordinal position as FD_i in every row type in *DTS* and <field name> is determined as follows.

Case:

i) If the field in the same ordinal position as FD_i in every row type in *DTS* have the same name *F*, then the <field name> in FD_i is *F*.

ii) Otherwise, the <field name> in FD_i is implementation-dependent (UV094).

h) If any data type in *DTS* is an array type or a distinct type whose source type is an array type, then every data type in *DTS* shall be an array type or a distinct type whose source type is an array type. The declared type of the result is array type with element data type *ETR*, where *ETR* is the data type resulting from the application of this Subclause to the set of element types of the array types of *DTS*, and maximum cardinality equal to the maximum of the maximum cardinalities of the data types in *DTS*.

i) 15 If any data type in *DTS* is a multiset type or a distinct type whose source type is a multiset type, then every data type in *DTS* shall be a multiset type or a distinct type whose source type is a multiset type. The declared type of the result is multiset type with element data type *ETR*, where *ETR* is the data type resulting from the application of this Subclause to the set of element types of the multiset types of *DTS*.

j) If any data type in *DTS* is a reference type, then there shall exist a subtype family *STF* such that each data type in *DTS* is a member of *STF*. Let *RT* be the minimal common supertype of each data type in *DTS*.

Case:

i) If the data type descriptor of every data type in *DTS* includes the name of a referenceable table identifying the scope of the reference type, and every such name is equivalent to some name *STN*, then the declared type of the result is:

```
RT SCOPE ( STN )
```

ii) Otherwise, the declared type of the result is *RT*.

k) If any data type in *DTS* is a JSON type, then each data type in *DTS* shall be a JSON type. The declared type of the result is JSON.

l) Otherwise, there shall exist a subtype family *STF* such that each data type in *DTS* is a member of *STF*. The declared type of the result is the minimal common supertype of each data type in *DTS*.

NOTE 371 — *Minimal common supertype* is defined in Subclause 4.9.3.4, "Subtypes and supertypes".

4) Let *DTR* be the declared type of the result.

5) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *DTR* as *RESTYPE*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.6 Subject routine determination

Function

Determine the subject routine of a given routine invocation.

Subclause Signature

```
"Subject routine determination" [Syntax Rules] (
  Parameter: "SET OF SQL-INVOKED ROUTINES",
  Parameter: "SQL ARGUMENT LIST"
) Returns: "SET OF SUBJECT ROUTINES"
```

SET OF SQL-INVOKED ROUTINES — a set of SQL-invoked routines for which appropriate subject routines are needed.

SQL ARGUMENT LIST — an SQL argument list used to identify appropriate subject routines.

SET OF SUBJECT ROUTINES — a set of subject SQL-invoked routines that is the result of invoking this subclause using this signature.

Syntax Rules

1) Let *SR* be the *SET OF SQL-INVOKED ROUTINES* and let *AL* be the *SQL ARGUMENT LIST* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *SET OF SUBJECT ROUTINES*.

2) Let n be the number of SQL-invoked routines in *SR*. Let R_i, 1 (one) $\le i \le n$, be the i-th SQL-invoked routine in *SR* in the ordering of *SR*.

3) Let m be the number of SQL arguments in *AL*. Let A_j, 1 (one) $\le j \le m$, be the j-th SQL argument in *AL*.

4) For A_j, 1 (one) $\le j \le m$, if A_j is neither a <dynamic parameter specification> nor a <contextually typed value specification>, then let $SDTA_j$ be the declared type of A_j.

5) Let $SDTP_{i,j}$ be the type designator of the declared type of the j-th SQL parameter of R_i. The type designator of the generic table parameter type is TABLE; the type designator of the descriptor parameter type is DESCRIPTOR.

6) For r varying from 1 (one) to m, if A_r is not a <dynamic parameter specification> and if there is more than one SQL-invoked routine in *SR*, then for each pair of SQL-invoked routines { R_p, R_q } in *SR*:

   a) The Syntax Rules of Subclause 9.7, "Type precedence list determination", are applied with $SDTA_r$ as *DATA TYPE*; let TPL_r be the *TYPE PRECEDENCE LIST* returned from the application of those Syntax Rules.

   b) If $SDTP_{p,r} \prec SDTP_{q,r}$ in TPL_r, then eliminate R_q from *SR*.

7) Let *RSR* be the set of SQL-invoked routines remaining in *SR*.

8) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *RSR* as *SET OF SUBJECT ROUTINES*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.7 Type precedence list determination

This Subclause is modified by Subclause 8.4, "Type precedence list determination", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.4, "Type precedence list determination", in ISO/IEC 9075-14.

Function

Determine the type precedence list of a given type.

Subclause Signature

```
"Type precedence list determination" [Syntax Rules] (
  Parameter: "DATA TYPE"
) Returns: "TYPE PRECEDENCE LIST"
```

DATA TYPE — an SQL data type.

TYPE PRECEDENCE LIST — a list of SQL data types containing every SQL data type that is included in the type precedence list of DATA TYPE.

Syntax Rules

1) Let *DT* be the *DATA TYPE* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *TYPE PRECEDENCE LIST*.

2) Let *TPL* be the list of *type designators* determined by the Syntax Rules of this Subclause.

3) Let "$A \prec B$" represent "*A* has precedence over *B*" and let "$A \simeq B$" represent "*A* has the same precedence as *B*".

4) If *DT* is a user-defined type, then:

   a) Let *ST* be the set of supertypes of *DT*. Let *n* be the number of data types in *ST*.

   b) For any two data types *TA* and *TB* in *ST*, $TA \prec TB$ if and only if *TA* is a proper subtype of *TB*.

   c) Let T_1 be *DT* and let T_{i+1}, 1 (one) $\leq i \leq n-1$, be the direct supertype of T_i.

   d) Let DTN_i, 1 (one) $\leq i \leq n$, be the data type designator of T_i.

      NOTE 372 — The type designator of a user-defined type is the type name included in its user-defined type descriptor.

   e) *TPL* is DTN_1, DTN_2, ..., DTN_n.

5) If *DT* is fixed-length character string, then *TPL* is

   ```
   CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT
   ```

6) If *DT* is variable-length character string, then *TPL* is

   ```
   CHARACTER VARYING, CHARACTER LARGE OBJECT
   ```

7) If *DT* is fixed-length binary string, then *TPL* is

   ```
   BINARY, BINARY VARYING, BINARY LARGE OBJECT
   ```

8) If *DT* is variable-length binary string, then *TPL* is

BINARY VARYING, BINARY LARGE OBJECT

9) If *DT* is numeric, then:

a) Let *NDT* be the following set of numeric types: NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT, DECFLOAT, REAL, FLOAT, and DOUBLE PRECISION. For each type *T* in *NDT*, the *effective binary precision* is defined as follows.

Case:

i) If *T* is DECIMAL or NUMERIC, then the effective binary precision is the product of $\log_2(10)$ and the implementation-defined (IL043) maximum precision of *T*.

ii) If *T* is FLOAT, then the effective binary precision is the implementation-defined (IL044) maximum precision of *T*.

iii) If the radix of *T* is decimal, then the effective binary precision is the product of $\log_2(10)$ and the implementation-defined (IV197) precision of *T*.

iv) Otherwise, the effective binary precision is the implementation-defined (IV197) precision of *T*.

b) Let *PTC* be the set of all precedence relationships determined as follows. For any two types *T1* and *T2*, not necessarily distinct, in *NDT*,

Case:

i) If *T1* and *T2* are both decimal floating-point types, then $T1 \simeq T2$.

ii) If *T1* is decimal floating-point type, and *T2* is either exact numeric or approximate numeric, then $T1 \succ T2$.

iii) If *T1* is either exact numeric or approximate numeric, and *T2* is decimal floating-point type, then $T1 \prec T2$.

iv) If *T1* is exact numeric and *T2* is approximate numeric, then $T1 \prec T2$.

v) If *T1* is approximate numeric and *T2* is exact numeric, then $T1 \succ T2$.

vi) If the effective binary precision of *T1* is greater than the effective binary precision of *T2*, then $T2 \prec T1$.

vii) If the effective binary precision of *T1* equals the effective binary precision of *T2*, then $T2 \simeq T1$.

viii) Otherwise, $T1 \prec T2$.

c) *TPL* is determined as follows:

i) *TPL* is initially empty.

ii) Let *ST* be the set of types containing *DT* and every type *T* in *NDT* for which the precedence relationship $DT \prec T$ or $DT \simeq T$ is in *PTC*.

iii) Let *n* be the number of types in *ST*.

iv) For *i* ranging from 1 (one) to *n*:

1) Let *NT* be the set of types T_k in *ST* such that there is no other type T_j in *ST* for which $T_j \prec T_k$ according to *PTC*.

2) Case:

A) If there is exactly one type T_k in *NT*, then T_k is placed next in *TPL* and all relationships of the form $T_k \prec T_r$ are removed from *PTC*, where T_r is any type in *ST*.

B) If there is more than one type T_k in *NT*, then every type T_s in *NT* is assigned the same position in *TPL* as T_k and all relationships of the forms $T_k \prec T_r$, $T_k \simeq T_r$, $T_s \prec T_r$, and $T_s \simeq T_r$ are removed from *PTC*, where T_r is any type in *ST*.

10) If *DT* specifies a year-month interval type, then *TPL* is

```
INTERVAL YEAR
```

11) If *DT* specifies a day-time interval type, then *TPL* is

```
INTERVAL DAY
```

12) If *DT* specifies DATE, then *TPL* is

```
DATE, TIMESTAMP
```

NOTE 373 — See CR 1) of this Subclause.

13) If *DT* specifies TIME, then *TPL* is

```
TIME
```

14) If *DT* specifies TIMESTAMP, then *TPL* is

```
TIMESTAMP
```

15) If *DT* specifies BOOLEAN, then *TPL* is

```
BOOLEAN
```

16) If *DT* specifies JSON, then *TPL* is

```
JSON
```

17) If *DT* is a collection type, then let *CTC* be the kind of collection specified in *DT*.

Let *n* be the number of elements in the type precedence list for the element type of *DT*. For *i* ranging from 1 (one) to *n*, let RIO_i be the *i*-th such element. *TPL* is

```
RIO1 CTC,
RIO2 CTC, ...,
RIOn CTC
```

18) If *DT* is a reference type, then let *n* be the number of elements in the type precedence list for the referenced type of *DT*. For *i* ranging from 1 (one) to *n*, let KAW_i be the *i*-th such element. *TPL* is

```
REF(KAW1),
REF(KAW2), ...,
REF(KAWn)
```

19) If *DT* is a row type, then *TPL* is

```
ROW
```

NOTE 374 — This rule is placed only to avoid the confusion that might arise if row types were not mentioned in this Subclause. As a row type cannot be used as a <parameter type>, the type precedence list of a row type is never referenced.

20) If *DT* is the generic table parameter type, then *TPL* is

    TABLE

21) If *DT* is the descriptor parameter type, then *TPL* is

    DESCRIPTOR

22) 09 14 Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *TPL* as *TYPE PRECEDENCE LIST*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F054, "TIMESTAMP in DATE type precedence list", the type precedence list of DATE is

    DATE

9.8 Host parameter mode determination

Function

Determine the parameter mode for a given host parameter.

Subclause Signature

```
"Host parameter mode determination" [Syntax Rules] (
  Parameter: "HOST PARAM DECL",
  Parameter: "SQL PROC STMT"
) Returns: "PARAMETER MODE"
```

HOST PARAM DECL — a <host parameter declaration>.

SQL PROC STMT — an <SQL procedure statement> that contains HOST PARAM DECL.

PARAMETER MODE — mode of HOST PARAM DECL (IN, OUT, INOUT, NONE).

Syntax Rules

1) Let *PD* be the *HOST PARAM DECL* and let *SPS* be the *SQL PROC STMT* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *PARAMETER MODE*.

2) Let *P* be the host parameter specified by *PD* and let *PN* be the <host parameter name> immediately contained in *PD*.

3) Whether *P* is an input host parameter, an output host parameter, or both an input host parameter and an output host parameter is determined as follows.

   Case:

   a) If *PD* is a <status parameter>, then let *PMODE* be "OUT".

   b) Otherwise,

      Case:

      i) If *PN* is contained in an <SQL argument> A_i of the <SQL argument list> of a <routine invocation> immediately contained in a <call statement> that is contained in *SPS*, then:

         1) Let *R* be the subject routine of the <routine invocation>.

         2) Let PR_i be the *i*-th SQL parameter of *R*.

         3) Case:

            A) If *PN* is contained in a <host parameter specification> that is the <target specification> that is simply contained in A_i and PR_i is an output SQL parameter, then let *PMODE* be "OUT".

            B) If *PN* is contained in a <host parameter specification> that is the <target specification> that is simply contained in A_i and PR_i is both an input SQL parameter and an output SQL parameter, then let *PMODE* be "INOUT".

            C) Otherwise, let *PMODE* be "IN".

ii) If *PN* is contained in a <value specification> or a <simple value specification> that is contained in *SPS*, and *PN* is not contained in a <target specification> or a <simple target specification> that is contained in *SPS*, then let *PMODE* be "IN".

iii) If *PN* is contained in a <target specification> or a <simple target specification> that is contained in *SPS*, and *PN* is not contained in a <value specification> or a <simple value specification> that is contained in *SPS*, then let *PMODE* "OUT".

iv) If *PN* is contained in a <value specification> or a <simple value specification> that is contained in *SPS*, and in a <target specification> or a <simple target specification> that is contained in *SPS*, then let *PMODE* be "INOUT".

v) Otherwise, let *PMODE* be "NONE".

4) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *PMODE* as *PARAMETER MODE*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.9 Type name determination

This Subclause is modified by Subclause 10.5, "Type name determination", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.6, "Type name determination", in ISO/IEC 9075-15.

Function

Determine an <identifier> given the name of a predefined or collection type.

Subclause Signature

```
"Type name determination" [Syntax Rules] (
  Parameter: "TYPE"
) Returns: "IDENTIFIER"
```

TYPE — a data type.

IDENTIFIER — an <identifier> that identifies the TYPE.

Syntax Rules

1) Let *DT* be the *TYPE* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *IDENTIFIER*.

2) *FNSDT* is defined as follows.

Case:

a) If *DT* specifies CHARACTER, then let *FNSDT* be "CHAR".

b) If *DT* specifies CHARACTER VARYING, then let *FNSDT* be "VARCHAR".

c) If *DT* specifies CHARACTER LARGE OBJECT, then let *FNSDT* be "CLOB".

d) If *DT* specifies BINARY, then let *FNSDT* be "BINARY".

e) If *DT* specifies BINARY VARYING, then let *FNSDT* be "VARBINARY".

f) If *DT* specifies BINARY LARGE OBJECT, then let *FNSDT* be "BLOB".

g) If *DT* specifies SMALLINT, then let *FNSDT* be "SMALLINT".

h) If *DT* specifies INTEGER, then let *FNSDT* be "INTEGER".

i) If *DT* specifies BIGINT, then let *FNSDT* be "BIGINT".

j) If *DT* specifies DECIMAL, then let *FNSDT* be "DECIMAL".

k) If *DT* specifies NUMERIC, then let *FNSDT* be "NUMERIC".

l) If *DT* specifies REAL, then let *FNSDT* be "REAL".

m) If *DT* specifies FLOAT, then let *FNSDT* be "FLOAT".

n) If *DT* specifies DECFLOAT, then let *FNSDT* be "DECFLOAT".

o) If *DT* specifies DOUBLE PRECISION, then let *FNSDT* be "DOUBLE".

p) If *DT* specifies DATE, then let *FNSDT* be "DATE".

q) If *DT* specifies TIME, then let *FNSDT* be "TIME".

r) If *DT* specifies TIMESTAMP, then let *FNSDT* be "TIMESTAMP".

s) If *DT* specifies INTERVAL, then let *FNSDT* be "INTERVAL".

t) If *DT* specifies BOOLEAN, then let *FNSDT* be "BOOLEAN".

u) If *DT* specifies JSON, then let *FNSDT* be "JSON".

v) If *DT* specifies ARRAY, then let *FNSDT* be "ARRAY".

w) 14 15 If *DT* specifies MULTISET, then let *FNSDT* be "MULTISET".

3) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *FNSDT* as *IDENTIFIER*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.10 Determination of identical values

This Subclause is modified by Subclause 8.5, "Determination of identical values", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.6, "Determination of identical values", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.7, "Determination of identical values", in ISO/IEC 9075-15.

Function

Determine whether two instances of values are identical, that is to say, are occurrences of the same value.

NOTE 375 — This Subclause is invoked implicitly wherever the word *identical* is used of two values.

Subclause Signature

```
"Determination of identical values" [General Rules] (
  Parameter: "FIRST VALUE",
  Parameter: "SECOND VALUE"
)
```

FIRST VALUE — the first of two values for which it is to be determined whether they are identical values.

SECOND VALUE — the second of two values for which it is to be determined whether they are identical values.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *V1* be the *FIRST VALUE* and let *V2* be the *SECOND VALUE* in an application of the General Rules of this Subclause.

2) Case:

   a) If *V1* and *V2* are both the null value, then *V1* is identical to *V2*.

   b) If *V1* is the null value and *V2* is not the null value, or if *V1* is not the null value and *V2* is the null value, then *V1* is not identical to *V2*.

   c) If *V1* and *V2* are of comparable predefined types, then

      Case:

      i) If *V1* and *V2* are character strings, then let *L* be CHARACTER_LENGTH(*V1*).

         Case:

1) If CHARACTER_LENGTH(*V2*) equals *L*, and if for all *i*, 1 (one) ≤ *i* ≤ *L*, the *i*-th character of *V1* corresponds to the same character position of ISO/IEC 10646:2020 as the *i*-th character of *V2*, then *V1* is identical to *V2*.

2) Otherwise, *V1* is not identical to *V2*.

ii) If *V1* and *V2* are TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE and are not distinct, and their time zone displacement fields are not distinct, then *V1* is identical to *V2*.

iii) Otherwise, *V1* is identical to *V2* if and only if *V1* is not distinct from *V2*.

d) 09 14 If *V1* and *V2* are of constructed types, then

Case:

i) If *V1* and *V2* are rows and their respective fields are identical, then *V1* is identical to *V2*.

ii) If *V1* and *V2* are arrays and have the same cardinality and elements in the same ordinal position in the two arrays are identical, then *V1* is identical to *V2*.

iii) 15 If *V1* and *V2* are multisets and have the same cardinality *N* and there exist enumerations $VE1_i$, 1 (one) ≤ *i* ≤ *N*, of *V1* and $VE2_i$, 1 (one) ≤ *i* ≤ *N*, of *V2* such that for all *i*, $VE1_i$ is identical to $VE2_i$, then *V1* is identical to *V2*.

iv) If *V1* and *V2* are references and *V1* is not distinct from *V2*, then *V1* is identical to *V2*.

v) Otherwise, *V1* is not identical to *V2*.

e) If *V1* and *V2* are of the same most specific type *MST* and *MST* is a user-defined type, then

Case:

i) If *MST* is a distinct type whose source type is *SDT* and the results of `SDT(V1)` and `SDT(V2)` are identical, then *V1* is identical to *V2*.

ii) If *MST* is a structured type and, for every observer function *O* defined for *MST*, the results of the invocations *O*(*V1*) and *O*(*V2*) are identical, then *V1* is identical to *V2*.

iii) Otherwise, *V1* is not identical to *V2*.

f) Otherwise, *V1* is not identical to *V2*.

3) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.11 Equality operations

This Subclause is modified by Subclause 8.6, "Equality operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.8, "Equality operations", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.8, "Equality operations", in ISO/IEC 9075-15.

Function

Specify the prohibitions and restrictions by data type on operations that involve testing for equality.

Syntax Rules

1) An *equality operation* is any of the following:
   a) A <comparison predicate> that specifies <equals operator> or <not equals operator>.
   b) A <quantified comparison predicate> that specifies <equals operator> or <not equals operator>.
   c) An <in predicate>.
   d) A <distinct predicate>.
   e) A <match predicate>.
   f) A <member predicate>.
   g) A <joined table> that specifies NATURAL or USING.
   h) A <user-defined ordering definition> that specifies MAP.
   i) A <position expression>.
2) An *operand of an equality operation* is any of the following:
   a) A field of the declared row type of a <row value predicand> that is simply contained in a <comparison predicate> that specifies <equals operator> or <not equals operator>.
   b) A field of the declared row type of a <row value predicand> that is simply contained in a <quantified comparison predicate> that specifies <equals operator> or <not equals operator>.
   c) A column of a <table subquery> that is simply contained in a <quantified comparison predicate> that specifies <equals operator> or <not equals operator>.
   d) A field of the declared row type of a <row value predicand> or <row value expression> that is simply contained in an <in predicate>.
   e) A column of a <table subquery> that is simply contained in an <in predicate>.
   f) A field of the declared row type of a <row value predicand> that is simply contained in a <distinct predicate>.
   g) A field of the declared row type of a <row value predicand> that is simply contained in a <match predicate>.
   h) A column of a <table subquery> that is simply contained in a <match predicate>.
   i) A field of the declared row type of a <row value predicand> that is simply contained in a <member predicate>.
   j) A corresponding join column of a <joined table> that specifies NATURAL or USING.

k) The <returns data type> of the SQL-invoked function identified by a <map function specification> simply contained in a <user-defined ordering definition> that specifies MAP.

l) A <string value expression> that is simply contained in a <position expression>.

m) A <binary value expression> that is simply contained in a <position expression>.

3) 15 The declared type of an operand of an equality operation shall not be UDT-NC-ordered.

4) Let *VS* be the set of declared types of the operands of an equality operation. If *VS* comprises character string types, then the Syntax Rules of Subclause 9.15, "Collation determination", are applied with *VS* as *TYPESET*; let the collation to be used in the equality operation be the *COLL* returned from the application of those Syntax Rules.

5) 09 14 If the declared type of an operand *OP* of an equality operation is a multiset type, then *OP* is a multiset operand of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared type of an operand of an equality operation shall not be ST-ordered.

2) Without Feature T042, "Extended LOB data type support", in conforming SQL language, the declared type of an operand of an equality operation shall not be LOB-ordered.

3) Without Feature S275, "Advanced multiset support", in conforming SQL language, the declared type of an operand of an equality operation shall not be multiset-ordered.

NOTE 376 — If the declared type of an operand *OP* of an equality operation is a multiset type, then *OP* is a multiset operand of a multiset element grouping operation. The Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply.

4) Without Feature T879, "JSON in equality operations", in conforming SQL language, the declared type of an operand of an equality operation shall not be JSON-ordered.

9.12 Grouping operations

This Subclause is modified by Subclause 8.7, "Grouping operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.9, "Grouping operations", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.9, "Grouping operations", in ISO/IEC 9075-15.

Function

Specify the prohibitions and restrictions by data type on operations that involve grouping of data.

Syntax Rules

1) A *grouping operation* is any of the following:
   a) A <group by clause>.
   b) A <window partition clause>.
   c) An <aggregate function> that specifies DISTINCT.
   d) A <query specification> that immediately contains DISTINCT.
   e) A <query expression body> that simply contains or implies UNION DISTINCT.
   f) A <query expression body> that simply contains EXCEPT.
   g) A <query term> that simply contains INTERSECT.
   h) A <unique predicate>.
   i) A <unique constraint definition>.
   j) A <referential constraint definition>.
   k) A <row pattern partition by>.
   l) A <table argument partitioning>.
2) An *operand of a grouping operation* is any of the following:
   a) A grouping column of a <group by clause>.
   b) A window partitioning column of a <window partition clause>.
   c) A row pattern partitioning column of a <row pattern partition by>.
   d) A <value expression> simply contained in an <aggregate function> that specifies DISTINCT.
   e) A column of the result of a <query specification> that immediately contains DISTINCT.
   f) A column of the result of a <query expression body> that simply contains or implies UNION DISTINCT.
   g) A column of the result of a <query expression body> that simply contains EXCEPT.
   h) A column of the result of a <query term> that simply contains INTERSECT.
   i) A column of the <table subquery> simply contained in a <unique predicate>.
   j) A column identified by the <unique column list> of a <unique constraint definition>.
   k) A partitioning column of a <table argument> of a <PTF derived table>.

l) A referencing column or a referenced column of a <referential constraint definition>.

3) 15 The declared type of an operand of a grouping operation shall not be LOB-ordered, array-ordered, multiset-ordered, UDT-EC-ordered, or UDT-NC-ordered.

4) 09 14 Let *VS* be the set of declared types of the operands of a grouping operation. If *VS* comprises character string types, then Syntax Rules of Subclause 9.15, "Collation determination", are applied with *VS* as *TYPESET*; let the collation to be used in the grouping operation be the *COLL* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared type of an operand of a grouping operation shall not be ST-ordered.

2) Without Feature T880, "JSON in grouping operations", in conforming SQL language, the declared type of an operand of a grouping operation shall not be JSON-ordered.

9.13 Multiset element grouping operations

This Subclause is modified by Subclause 8.8, "Multiset element grouping operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.10, "Multiset element grouping operations", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.10, "Multiset element grouping operations", in ISO/IEC 9075-15.

Function

Specify the prohibitions and restrictions by data type on the declared element type of a multiset for operations that involve grouping the elements of a multiset.

Syntax Rules

1) A *multiset element grouping operation* is any of the following:

   a) An equality operation such that the declared type of an operand of the equality operation is a multiset type.

   b) A <multiset set function>.

   c) A <multiset value expression> that specifies MULTISET UNION DISTINCT.

   d) A <multiset value expression> that specifies MULTISET EXCEPT.

   e) A <multiset term> that specifies MULTISET INTERSECT.

   f) A <submultiset predicate>.

   g) A <set predicate>.

   h) A <general set function> that specifies INTERSECTION.

2) A *multiset operand* of a multiset element grouping operation is any of the following:

   a) A <multiset value expression> simply contained in a <multiset set function>.

   b) A <multiset value expression> or a <multiset term> that is simply contained in a <multiset value expression> that simply contains MULTISET UNION DISTINCT.

   c) A <multiset value expression> or a <multiset term> that is simply contained in a <multiset value expression> that simply contains MULTISET EXCEPT.

   d) A <multiset term> or a <multiset primary> that is simply contained in a <multiset term> that simply contains MULTISET INTERSECT.

   e) An operand of an equality operation such that the declared type of the operand is a multiset type.

   f) A field of the <row value predicand> simply contained in a <submultiset predicate>.

   g) The <multiset value expression> simply contained in a <submultiset predicate>.

   h) A field of the <row value predicand> simply contained in a <set predicate>.

   i) A <value expression> simply contained in a <general set function> that specifies INTERSECTION.

3) 15 The declared element type of a multiset operand of a multiset element grouping operation shall not be LOB-ordered, array-ordered, multiset-ordered, UDT-EC-ordered, or UDT-NC-ordered.

4) 09 14 Let *VS* be the set of declared element types of the multiset operands of a multiset element grouping operation. If *VS* comprises character string types, then Syntax Rules of Subclause 9.15, "Collation determination", are applied with *VS* as *TYPESET*; let the collation to be used in the multiset element grouping operation be the *COLL* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared element type of a multiset operand of a multiset element grouping operation shall not be ST-ordered.

2) Without Feature T882, "JSON in multiset element grouping operations", in conforming SQL language, the declared type of an operand of a grouping operation shall not be JSON-ordered.

9.14 Ordering operations

This Subclause is modified by Subclause 8.9, "Ordering operations", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.11, "Ordering operations", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.11, "Ordering operations", in ISO/IEC 9075-15.

Function

Specify the prohibitions and restrictions by data type on operations that involve ordering of data.

Syntax Rules

1) An *ordering operation* is any of the following:

   a) A <comparison predicate> that does not specify <equals operator> or <not equals operator>.

   b) A <quantified comparison predicate> that does not specify <equals operator> or <not equals operator>.

   c) A <between predicate>.

   d) An <aggregate function> that specifies MAX or MIN.

   e) A <sort specification list>.

   f) A <user-defined ordering definition> that specifies ORDER FULL BY MAP.

   g) A <table argument ordering>.

   h) A <greatest or least function>.

2) An *operand of an ordering operation* is any of the following:

   a) A field of the declared row type of a <row value predicand> that is simply contained in a <comparison predicate> that does not specify <equals operator> or <not equals operator>.

   b) A field of the declared row type of a <row value predicand> that is simply contained in a <quantified comparison predicate> that does not specify <equals operator> or <not equals operator>.

   c) A column of the <table subquery> that is simply contained in a <quantified comparison predicate> that does not specify <equals operator> or <not equals operator>.

   d) A field of the declared row type of a <row value predicand> that is simply contained in a <between predicate>.

   e) A <value expression> simply contained in an <aggregate function> that specifies MAX or MIN.

   f) A <value expression> simply contained in a <sort key>.

   g) The <returns data type> of the SQL-invoked function identified by a <map function specification> simply contained in a <user-defined ordering definition> that specifies ORDER FULL BY MAP.

   h) An ordering column of a <table argument> of a <PTF derived table>.

   i) A <value expression> simply contained in a <greatest or least> function.

3) [15] The declared type of an operand of an ordering operation shall not be LOB-ordered, array-ordered, multiset-ordered, reference-ordered, UDT-EC-ordered, or UDT-NC-ordered.

NOTE 377 — In addition, by a Syntax Rule of Subclause 10.9, "<aggregate function>", an operand of MAX or MIN must not be row-ordered. This is because the ordering of non-null row-ordered values is not a total ordering (for example, the relative order of the rows (1, null) and (null, 1) is indeterminate). The General Rules of Subclause 10.10, "<sort specification list>", provide special rules to totally order rows containing null values, but these rules are not used in MAX or MIN. The other ordering operations do not require a total ordering.

4) 09 14 Let *VS* be the set of declared types of the operands of an ordering operation. If *VS* comprises character string types, then Syntax Rules of Subclause 9.15, "Collation determination", are applied with *VS* as *TYPESET*; let the collation to be used in the ordering operation be the *COLL* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", in conforming SQL language, the declared type of an operand of an ordering operation shall not be ST-ordered.

2) Without Feature T881, "JSON in ordering operations", in conforming SQL language, the declared type of an operand of an ordering operation shall not be JSON-ordered.

9.15 Collation determination

Function

Specify rules for determining the collation to be used in the comparison of character strings.

Subclause Signature

```
"Collation determination" [Syntax Rules] (
  Parameter: "TYPESET"
) Returns: "COLL"
```

TYPESET — a set of data types for which a (common) collation is desired.

COLL — the collation, if any, that is usable for all data types contained in TYPESET.

Syntax Rules

1) Let *TS* be the *TYPESET* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *COLL*.

2) At least one declared type in *TS* shall have a declared type collation.

3) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *TS* as *DTSET*; let *RDT* be the *RESTYPE* returned from the application of those Syntax Rules. Let *CCS* be the character set of *RDT*.

4) Case:

   a) If the comparison operation is a <referential constraint definition>, then, for each referencing column *RC* and its corresponding referenced column *RFC*, the declared type collation of *RC* and the declared type collation of *RFC* shall be the same collation *COLL* and the collation to be used is *COLL*.

   b) If at least one operand has an *explicit* collation derivation, then every operand whose collation derivation is *explicit* shall have the same declared type collation *EDTC* and the collation to be used is *EDTC*.

   c) If the comparison operation is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement>, or in a <direct SQL statement> that is invoked directly, and *CCS* has an SQL-session collation, then the collation to be used is that SQL-session collation.

   d) If *CCS* has an SQL-client module collation, then the collation to be used is that collation.

   e) Otherwise, every operand whose collation derivation is *implicit* shall have the same declared type collation *IDTC* and the collation to be used is *IDTC*.

5) Let *CTBU* be the collation to be used.

6) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *CBTU* as *COLL*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.16 Potential sources of non-determinism

This Subclause is modified by Subclause 8.1, "Potential sources of non-determinism", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.12, "Potential sources of non-determinism", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.12, "Potential sources of non-determinism", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 9.1, "Potential sources of non-determinism", in ISO/IEC 9075-16.

Function

Define the potential sources of non-determinism.

Syntax Rules

1) A *potential source of non-determinism* is any of the following:

NOTE 378 — The following cases are arranged in order of the subclauses in which the various potential sources of non-determinism arise.

a) A <value specification> that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH.

b) A <column reference> that is a column reference, the implicit or explicit qualifying range variable of which is a <table name> or <query name> that identifies a potential source of non-determinism.

c) A <window function> that specifies ROW_NUMBER, FIRST_VALUE, LAST_VALUE, NTH_VALUE, NTILE, LEAD, or LAG, or whose associated <window specification> specifies ROWS.

d) A <cast specification> that either is, or recursively implies through the execution of the General Rules of Subclause 6.13, "<cast specification>", one of the following:

i) A <cast specification> whose result type is datetime with time zone and whose <cast operand> has declared type that is not datetime with time zone.

ii) A <cast specification> whose result type is an array type and whose <cast operand> has a declared type that is a multiset type.

e) A <next value expression>.

f) A <greatest or least function> that simply contains a <value expression> whose declared type is based on a character string type, user-defined type, or datetime with time zone type.

NOTE 379 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

g) A <JSON value function>.

h) A <JSON query>.

i) A <datetime factor> that simply contains a <datetime primary> whose declared type is datetime without time zone and that simply contains an explicit <time zone>.

j) A <datetime value function>.

k) An <interval value expression> that computes the difference of a <datetime value expression> and a <datetime term>, such that the declared type of one operand is datetime with time zone and the other operand is datetime without time zone.

l) An <array value constructor by query>.

m) A <multiset value expression> that specifies or implies MULTISET UNION, MULTISET EXCEPT, or MULTISET INTERSECT such that the element types of the operands have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

n) A <multiset value expression> that specifies MULTISET UNION DISTINCT, MULTISET EXCEPT, or MULTISET INTERSECT and whose result type's declared element type is based on character string type, a user-defined type, or a datetime type with time zone.

o) A <multiset set function> whose declared element type is based on a character string type, a user-defined type, or a datetime type with time zone.

p) A <table name>, if the table identified by the <table name> is a viewed table, and the hierarchical <query expression> in the view descriptor identified by the <table name> contains a potential source of non-determinism.

NOTE 380 — Creation of a subtable can change a view from deterministic to possibly non-deterministic; however, this is prohibited if there is any dependency on the view that requires it not to contain a potential source of non-determinism.

q) A <query name>, if the <query expression> identified by the <query name> contains a potential source of non-determinism.

r) A <data change delta table>.

s) A <query system time period specification> that does not implicitly or explicitly specify FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP.

t) A <sample clause>.

u) A <JSON table>.

v) A <JSON serialize>.

w) A <JSON parse>.

x) A <row pattern recognition clause>.

y) A <joined table> that specifies either NATURAL or a <join specification> immediately containing a <named columns join>, and there is a common column name *CCN* such that the declared types of the two corresponding join columns identified by *CCN* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

z) A <joined table> such that the declared type of a join partitioning column is character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE.

aa) A <window specification> that contains a <row pattern common syntax>.

ab) A <query specification> *QS* that specifies the <set quantifier> DISTINCT and one of the columns of the result of *QS* has a data type of character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE.

ac) A <query specification> *QS* in which the <select list>, <having clause>, or <window clause> contains a reference to a column *C* of the result of the <table expression> simply contained in *QS* that has a data type of character string, user-defined type, TIME WITH TIME ZONE, or TIMESTAMP WITH TIME ZONE, and the functional dependency $G \mapsto C$, where *G* is the set consisting of the grouping columns of *QS*, holds in the result of *QS*.

ad) A <result offset clause>.

ae) A <fetch first clause>.

af) A <query expression> such that UNION, EXCEPT, or INTERSECT is specified and there is a column of the result such that the declared types *DT1* and *DT2* of the column in the two operands have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

ag) A <query expression> *QE* such that all of the following are true:

  i) *QE* contains a set operator UNION and ALL is not specified, or *QE* contains either of the set operators EXCEPT or INTERSECT.

  ii) At least one of the following is true:

    1) The first or second operand contains a column that has a declared type of character string.

    2) The first or second operand contains a column that has a declared type of datetime with time zone.

    3) The first or second operand contains a column that has a declared type that is a user-defined type.

ah) A <comparison predicate>, <overlaps predicate>, or <distinct predicate> simply containing <row value predicand>s *RVP1* and *RVP2* such that the declared types of *RVP1* and *RVP2* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

NOTE 381 — This includes <between predicate> because of a syntactic transformation to <comparison predicate>.

ai) A <quantified comparison predicate> or a <match predicate> simply containing a <row value predicand> *RVP* and a <table subquery> *TS* such that the declared types of *RVP* and *TS* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

NOTE 382 — This includes <in predicate> because of a syntactic transformation to <quantified comparison predicate>.

aj) A <member predicate> simply containing a <row value predicand> *RVP* and a <multiset value expression> *MVP* such that the declared type of the only field *F* of *RVP* and the element type of *MVP* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

ak) A <submultiset predicate> simply containing a <row value predicand> *RVP* and a <multiset value expression> *MVP* such that the declared type of the only field *F* of *RVP* and the declared type of *MVP* have corresponding constituents such that one constituent is datetime with time zone and the other is datetime without time zone.

al) A <JSON exists predicate>.

am) A <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic.

an) A <routine invocation> whose subject routine is a polymorphic table function having a PTF component procedure that is possibly non-deterministic.

ao) An <aggregate function> that specifies MIN or MAX and that simply contains a <value expression> whose declared type is based on a character string type, user-defined type, or datetime with time zone type.

ap) An <aggregate function> that specifies ANY_VALUE.

aq) An <aggregate function> that specifies INTERSECTION and that simply contains a <value expression> whose declared element type is based on a character string type, a user-defined type, or a datetime type with time zone.

ar) An <array aggregate function>.

as) 04 14 15 16 A <listagg set function>.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.17 Executing an <SQL procedure statement>

This Subclause is modified by Subclause 8.2, "Executing an <SQL procedure statement>", in ISO/IEC 9075-4.

Function

Execute an <SQL procedure statement>.

Subclause Signature

```
"Executing an <SQL procedure statement>" [General Rules] (
  Parameter: "EXECUTING STATEMENT"
)
```

EXECUTING STATEMENT — an <SQL procedure statement> that is to be executed.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the *EXECUTING STATEMENT* in an application of the General Rules of this Subclause.

NOTE 383 — *S* is necessarily the innermost executing statement of the SQL-session as defined in Subclause 4.45, "SQL-sessions".

NOTE 384 — *S* is not necessarily an <SQL procedure statement>.

2) Except where explicitly specified, the General Rules of this Subclause are not terminated if an exception condition is raised.

3) A *statement execution context NEWSEC* is established for the execution of *S*. Let *OLDSEC* be the most recent statement execution context. *NEWSEC* becomes the most recent statement execution context. *NEWSEC* is an atomic execution context, and therefore the most recent atomic execution context, if and only if *S* is an atomic SQL-statement.

4) In *NEWSEC*, the set of state changes *SSC* is initially empty, the set of old delta tables and new delta tables is initially empty, and the set of views to be checked is initially empty.

5) If the current SQL-session context *CSC* indicates that no triggered action is executing, and the current routine execution context of *CSC* indicates that an SQL-invoked function is not active, then the statement timestamp in *CSC* is set to "not set".

6) If *S* is immediately contained in an <externally-invoked procedure> *EP*, then:

   a) Let *LANG* be the caller language of *EP*; let *n* be the number of <host parameter declaration>s specified in *EP*; let PD_i, 1 (one) $\le i \le n$, be the *i*-th such <host parameter declaration>; let PN_i and DT_i be the <host parameter name> and <host parameter data type>, respectively, specified in PD_i; and let P_i be the host parameter corresponding to PD_i. When *EP* is called by an SQL-agent, let PI_i be the *i*-th argument in the procedure call.

b) The SQL-client module that contains *S* is associated with the SQL-agent.

7) 04 If *S* is not an <SQL diagnostics statement>, then the first diagnostics area is emptied.

8) If *S* is not an <SQL diagnostics statement> or an <SQL connection statement>, and no SQL-session is current for the SQL-agent, then

Case:

a) If the SQL-agent has not executed an <SQL connection statement> and there is no default SQL-session associated with the SQL-agent, then the following <connect statement> is effectively executed:

```
CONNECT TO DEFAULT
```

b) If the SQL-agent has not executed an <SQL connection statement> and there is a default SQL-session associated with the SQL-agent, then the following <set connection statement> is effectively executed:

```
SET CONNECTION DEFAULT
```

c) Otherwise, an exception condition is raised: *connection exception — connection does not exist (08003)*.

9) If no exception condition has been raised, then:

a) If *S* is immediately contained in an <externally-invoked procedure>, then for each *i*, 1 (one) ≤ *i* ≤ *n*, if P_i is an input host parameter or both an input host parameter and an output host parameter, then the General Rules of Subclause 9.3, "Passing a value from a host language to the SQL-server", are applied with *LANG* as *LANGUAGE*, DT_i as *SQL TYPE*, and PI_i as *HOST VALUE*; let *SV* be the *SQL VALUE* returned from the application of those General Rules. The value of P_i is set to *SV*.

b) If *S* does not conform to the Syntax Rules and Access Rules of an <SQL procedure statement>, then an exception condition is raised: *syntax error or access rule violation (42000)*.

NOTE 385 — Subclause 6.3.3.2, "Terms denoting rule requirements", and Subclause 6.3.3.3, "Rule evaluation order", in ISO/IEC 9075-1, require that the Syntax Rules and Access Rules remain satisfied during General Rule evaluation; doing so might require initiation of a transaction earlier than the General Rules otherwise suggest.

10) If no exception condition has been raised and no SQL-transaction is active for the SQL-agent and *S* is an SQL-statement that implicitly initiates SQL-transactions, then an SQL-transaction is initiated. If *S* is immediately contained in an <externally-invoked procedure>, then the SQL-client module that contains *S* is associated with the SQL-transaction.

NOTE 386 — If *S* is a <prepare statement>, a transaction can be (but is not required to be) initiated based on the contents of the <SQL statement variable>. See Subclause 4.41.4, "SQL-statements and transaction states".

11) If no exception condition has been raised and the execution of an <SQL data statement>, <SQL dynamic data statement>, <dynamic select statement>, <dynamic single row select statement>, or <direct SQL data statement> occurs within the same SQL-transaction as the execution of an SQL-schema statement, then the effects on any open cursor or deferred constraint are implementation-defined (IA210).

Case:

a) If the SQL-implementation does not support Feature T670, "Schema and data statement mixing", then an exception condition is raised: *invalid transaction state — schema and data statement mixing not supported (25007)*.

b) If any additional implementation-defined (IC013) restrictions, requirements, or conditions are violated, then it is implementation-defined (IC013) whether an implementation-defined (IC013) exception condition or a completion condition *warning (01000)* with an implementation-defined (IC013) subclass code is raised.

12) If no exception condition has been raised and *S* is an <SQL schema statement> and the transaction access mode of the current SQL-transaction is read-only, then an exception condition is raised: *invalid transaction state (25000)*.

13) If no exception condition has been raised, then:

a) The General Rules of *S* are evaluated.

b) If *S* is immediately contained in an <externally-invoked procedure>, then for each *i*, 1 (one) ≤ *i* ≤ *n*, if P_i is an output host parameter that is not the status parameter or both an input host parameter and an output host parameter, then the General Rules of Subclause 9.4, "Passing a value from the SQL-server to a host language", are applied with *LANG* as *LANGUAGE*, DT_i as *SQL TYPE*, and the value of P_i as *SQL VALUE*; let *LV* be the *HOST VALUE* returned from the application of those General Rules. The value of PI_i is set to *LV*.

14) If prior General Rules successfully initiated or resumed an SQL-session, then subsequent calls to an <externally-invoked procedure> and subsequent invocations of <direct SQL statement>s by the SQL-agent are associated with that SQL-session until the SQL-agent terminates the SQL-session or makes it dormant.

15) If no exception condition has been raised, then for every enforced referential constraint *RC*, the General Rules of Subclause 15.18, "Execution of referential actions", are applied with *RC* as *CONSTRAINT*.

16) If no exception condition has been raised, then for every state change *DSC* in *SSC* whose trigger event is DELETE, let *DSOT* be the set of transitions in *DSC* and let *DBT* be the subject table of *DSC*.

Case:

a) If *DBT* is a system-versioned table, then:

i) Let *STARTCOL* be the system-time period start column of *DBT*. Let *ENDCOL* be the system-time period end column of *DBT*. Let *DT* be the declared type of *ENDCOL*.

ii) Let *TTS* be the transaction timestamp of the current SQL-transaction. Let *CTTS* be the result of

```
CAST (TTS AS DT)
```

iii) For each row *R* marked for deletion from *DBT*, let *STARTVAL* be the value of *STARTCOL*.

Case:

1) If *CTTS* < *STARTVAL*, then an exception condition is raised: *data exception — invalid row version (2201H)*.

2) If *CTTS* = *STARTVAL*, then *R* is deleted from *DBT*.

3) Otherwise, the value of *ENDCOL* is effectively replaced by *CTTS*.

b) Otherwise, each row that is marked for deletion from *DBT* is deleted from *DBT*.

17) If no exception condition has been raised, then

Case:

a) If, for some enforced constraint *C*, the constraint mode of *C* in the current SQL-session is immediate and an applicable <search condition> included in the descriptor of *C* evaluates to *False*, then an exception condition is raised: *integrity constraint violation (23000)*.

NOTE 387 — This is needed to cater for both referential constraints specifying NO ACTION and referential constraints that are not satisfied after application of referential actions.

b) Otherwise, the General Rules of Subclause 15.20, "Execution of AFTER triggers", are applied with *SSC* as *SET OF STATE CHANGES*.

18) If no exception has been raised, then it is implementation-defined (IA230) whether tertiary effects of a data change operation that has inserted or replaced rows in a target leaf generally underlying table of a view *V* in the set of views to be checked *LV* are checked for conformity to the CHECK OPTION of *V*. If such a check is made, and a row is found not to conform to the CHECK OPTION of *V*, then an exception condition is raised: *with check option violation (44000)*.

19) If *S* is a <select statement: single row> or a <fetch statement> and a completion condition is raised: *no data (02000)*, or an exception condition is raised, then the value of each result that is assigned to a <target specification> in *S* is implementation-dependent (UV105).

NOTE 388 — If *S* is immediately contained in an <externally-invoked procedure>, then these results are the values PI_i of the output parameters PN_i. Using ISO/IEC 9075-4, the results might be the values of SQL variables. If *S* is contained in the body of an SQL routine, the results might be values of SQL parameters. Direct SQL does not support <select statement: single row> or <fetch statement>.

20) Case:

a) If *S* executed successfully, then either a completion condition is raised: *successful completion (00000)*, or a completion condition is raised: *warning (01000)*, or a completion condition is raised: *no data (02000)*, as determined by the General Rules in this and other Subclauses of the ISO/IEC 9075 series.

b) If *S* did not execute successfully, then:

i) If *S* is an atomic SQL-statement, then all changes made to SQL-data or schemas by the execution of *S* are canceled.

NOTE 389 — Atomic and non-atomic SQL-statements are defined in Subclause 4.41.5, "SQL-statement atomicity and statement execution contexts".

ii) The exception condition with which the execution of *S* completed is raised.

21) If *S* is immediately contained in an <externally-invoked procedure>, then the status parameter is set to the value specified for the condition in Clause 24, "Status codes".

22) If *S* is not an <SQL diagnostics statement>, then diagnostics information resulting from the execution of *S* is placed into the first diagnostics area, causing the first condition area in the first diagnostics area to become occupied. Whether any other condition areas become occupied is implementation-defined (IA079).

23) 04 If *NEWSEC* is atomic, then all savepoints established during its existence are destroyed.

24) *NEWSEC* ceases to exist and *OLDSEC* becomes the most recent statement execution context.

25) 04 *S* ceases to be an executing statement.

NOTE 390 — The innermost executing statement, if any, is now the one that was the innermost executing statement that caused *S* to be executed.

26) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.18 Invoking an SQL-invoked routine

This Subclause is modified by Subclause 8.4, "Invoking an SQL-invoked routine", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 7.1, "Invoking an SQL-invoked routine", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 8.1, "Invoking an SQL-invoked routine", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 10.13, "Invoking an SQL-invoked routine", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 9.13, "Invoking an SQL-invoked routine", in ISO/IEC 9075-15.

Function

Execute the invocation of an SQL-invoked routine.

Subclause Signature

```
"Invoking an SQL-invoked routine" [Syntax Rules] (
  Parameter: "ROUTINE INVOCATION",
  Parameter: "SQLPATH",
  Parameter: "UDT"
) Returns: "SUBJECT ROUTINE" and "STATIC SQL ARG LIST"
```

ROUTINE INVOCATION — a <routine invocation> to be evaluated.

SQLPATH — an optional SQL-path used to resolve user-defined type names.

UDT — a user-defined type for which ROUTINE INVOCATION may be a method.

SUBJECT ROUTINE — the particular subject routine that is determined by the invocation of this subclause using this signature.

STATIC SQL ARG LIST — the static SQL argument list that is determined by the invocation of this subclause using this signature.

```
"Invoking an SQL-invoked routine" [General Rules] (
  Parameter: "SUBJECT ROUTINE",
  Parameter: "STATIC SQL ARG LIST"
) Returns: "VALUE"
```

SUBJECT ROUTINE — the routine to be invoked.

STATIC SQL ARG LIST — a static SQL argument list for the SUBJECT ROUTINE.

VALUE — a value that is returned from the invocation of the specified SUBJECT ROUTINE when invoked with STATIC SQL ARG LIST.

Syntax Rules

1) Let *RI* be the *ROUTINE INVOCATION*, let *TP* be the *SQLPATH*, and let *UDTSM* be the *UDT* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *SUBJECT ROUTINE* and *STATIC SQL ARG LIST*.

2) Let *RN* be the <routine name> immediately contained in *RI*.

3) If *RI* is immediately contained in a <call statement>, then the <SQL argument list> of *RI* shall not contain a <generalized expression> without an intervening <routine invocation>.

4) If the <SQL argument list> contains any <SQL argument> that is a <named argument specification>, then all <SQL argument>s contained in that <SQL argument list> shall be <named argument specification>s; no two <SQL parameter name>s contained in the <SQL argument list> shall be equivalent.

5) Case:

   a) If *RI* is immediately contained in a <call statement>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* (henceforth, simply "possibly candidate routine") if *R* is an SQL-invoked procedure and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

   b) If *RI* is immediately contained in a <method selection>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an instance SQL-invoked method and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

   c) If *RI* is immediately contained in a <constructor method selection>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an SQL-invoked constructor method and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

   d) If *RI* is immediately contained in a <static method selection>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is a static SQL-invoked method and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN* and the method specification descriptor for *R* is included in a user-defined type descriptor for *UDTSM* or for some supertype of *UDTSM*.

   e) If *RI* simply contains a <table argument> or a <descriptor argument>, then an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is a polymorphic table function and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

   f) Otherwise, an SQL-invoked routine *R* is a *possibly candidate routine* for *RI* if *R* is an SQL-invoked regular function or a polymorphic table function and the <qualified identifier> of the <routine name> of *R* is equivalent to the <qualified identifier> of *RN*.

6) 04 Case:

   a) If *RI* is contained in an <SQL schema statement>, then an SQL-invoked routine *R* is an *executable routine* if and only if *R* is a possibly candidate routine and the applicable privileges for the <authorization identifier> that owns the containing schema include EXECUTE on *R*.

   b) Otherwise, an SQL-invoked routine *R* is an *executable routine* if and only if *R* is a possibly candidate routine and the current privileges include EXECUTE on *R*.

7) Case:

   a) If *RI* is immediately contained in a <method selection>, <static method selection>, or a <constructor method selection>, then let *DP* be *TP*.

   b) If the routine execution context of the current SQL-session indicates that an SQL-invoked routine is active, then let *DP* be the routine SQL-path of that routine execution context.

   c) 04 If *RI* is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

   d) If *RI* is contained in a <preparable statement> that is prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> or in a <direct SQL statement> that is invoked directly, then let *DP* be the SQL-path of the current SQL-session.

   e) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains *RI*.

8) Let *NA* be the number of <SQL argument>s in the <SQL argument list> *AL* of *RI*.

9) If *RI* is immediately contained in a <call statement>, then:

   a) *RI* shall not simply contain <table argument>, <descriptor argument>, or <co-partition clause>.

   b) An *invocable routine* is an SQL-invoked routine *SIR* that is an executable routine such that all of the following are true:

      i) *SIR* has *SIRNA* SQL parameters, with $SIRNA \geq NA$.

      ii) If the <SQL argument list> does not contain any <SQL argument> that is a <named argument specification>, then for each k, $NA < k \leq SIRNA$, the routine descriptor of *SIR* contains an indication that the k-th SQL parameter has a default value.

   c) The set of *candidate routines* of *RI* is defined as follows.

      Case:

      i) If *RN* does not contain a <schema name>, then the candidate routines of *RI* are the set union of invocable routines of all schemas whose <schema name> is in *DP*.

      ii) If *RN* contains a <schema name> *SN*, then *SN* shall be the <schema name> of a schema *S*. The candidate routines of *RI* are the invocable routines (if any) contained in *S*.

   d) For each candidate routine *PCR* of *RI*, if *AL* contains at least one <named argument specification>, then:

      i) Let *MAP* be a mapping of every <SQL parameter name> immediately contained in the <named argument specification>s to the parameter names of *PCR*, such that those names are equivalent. If *MAP* does not exist, then *PCR* is removed from the set of candidate routines.

      ii) If there exists an SQL parameter *SPWOD* for which no <named argument specification> exists in *AL* that has an equivalent <SQL parameter name> and the routine descriptor of *PCR* does not contain an indication that *SPWOD* has a default value, then *PCR* is removed from the set of candidate routines.

   e) Case:

      i) If *RN* does not contain a <schema name>, then let *S1* be a schema included in *DP* such that *S1* includes at least 1 (one) candidate routine of *RI* and there is no other schema that precedes *S1* in *DP* and contains at least 1 (one) candidate routine of *RI*. Any routine not contained in *S1* is removed from the set of candidate routines.

      ii) If *RN* contains a <schema name>, then let *S1* be the schema identified by that <schema name>. *S1* shall include at least 1 (one) candidate routine of *RI*. Any routine not contained in *S1* is removed from the set of candidate routines.

   f) The set of *candidate routines with minimal number of parameters* of *RI* comprises those candidate routines such that there is no other candidate routine that has fewer parameters.

      NOTE 391 — There is at most one routine in the set of candidate routines with minimal number of parameters.

   g) The subject routine *SR* of *RI* is the SQL-invoked routine that is a candidate routine with minimal number of parameters of *RI* such that there is no other candidate routine with minimal number of parameters.

   h) Let *SRNP* be the number of SQL parameters of *SR*. Let P_i, 1 (one) $\leq i \leq SRNP$, be the i-th SQL parameter of *SR*.

      i) Case:

1) If *AL* does not immediately contain at least one <named argument specification>, then:

   A) Let A_i, 1 (one) ≤ i ≤ *NA*, be the i-th <SQL argument> contained in *AL*.

   B) Let A_i, *NA* < i ≤ *SRNP*, be DEFAULT.

   C) Let *XAL* be an <SQL argument list> whose i-th <SQL argument> is A_i, 1 (one) ≤ i ≤ *SRNP*.

      NOTE 392 — If *SRNP* = 0 (zero), then *XAL* is an empty list of SQL arguments.

2) Otherwise:

   A) Let $NPSA_i$, 1 (one) ≤ i ≤ *SRNP*, be the <named argument SQL argument> immediately contained in the <named argument specification> in *AL* whose <SQL parameter name> is equivalent to P_i.

   B) For each SQL parameter P_i, 1 (one) ≤ i ≤ *SRNP*, for which no <named argument specification> exists in *AL* that has an equivalent <SQL parameter name>, let $NPSA_i$ be DEFAULT.

   C) Let *XAL* be an <SQL argument list> whose i-th <SQL argument> is $NPSA_i$, 1 (one) ≤ i ≤ *SRNP*.

ii) Let XA_i, 1 (one) ≤ i ≤ *SRNP*, be the i-th <SQL argument> in *XAL*.

iii) For each SQL parameter P_i, 1 (one) ≤ i ≤ *SRNP*, that is an output SQL parameter or both an input SQL parameter and an output SQL parameter:

   1) If P_i is an output SQL parameter, then XA_i shall be a <target specification>.

   2) If P_i is both an input SQL parameter and an output SQL parameter, then XA_i shall be either a <default specification> or a <target specification>.

   3) If *RI* is contained in a <triggered SQL statement> of an AFTER trigger, then XA_i shall not be a <column reference>.

   4) [10] If XA_i is an <embedded variable specification> or a <host parameter specification>, then the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with XA_i as *TARGET* and P_i as *VALUE*.

   5) [04] If XA_i is an <SQL parameter reference>, a <column reference>, or a <target array element specification>, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with XA_i as *TARGET* and P_i as *VALUE*.

      NOTE 393 — The <column reference> can only be a new transition variable column reference.

iv) For each SQL parameter P_i, 1 (one) ≤ i ≤ *SRNP*, that is an input SQL parameter but not an output SQL parameter, XA_i shall be either a <value expression> or a <contextually typed value specification>.

v) For each SQL parameter P_i, 1 (one) ≤ i ≤ *SRNP*, that is an input SQL parameter or both an input SQL parameter and an output SQL parameter, and each XA_i that is not a <contextually typed value specification>, the Syntax Rules of Subclause 9.2, "Store assignment", are applied with P_i as *TARGET* and XA_i as *VALUE*.

10) If *RI* is not immediately contained in a <call statement>, then:

a) An *invocable routine* is an SQL-invoked routine *SIR* that is an executable routine such that all of the following are true:

   i) Let the number of SQL parameters of *SIR* be *SIRNA*; $SIRNA \geq NA$.

   ii) If the <SQL argument list> does not contain any <SQL argument> that is a <named argument specification>, then for each k, $NA < k \leq SIRNA$, then the routine descriptor of *SIR* contains an indication that the k-th SQL parameter has a default value.

b) The set of *candidate routines* of *RI* is defined as follows.

   Case:

   i) If *RN* does not contain a <schema name>, then the candidate routines of *RI* are the set union of invocable routines of all schemas whose <schema name> is in *DP*.

   ii) If *RN* contains a <schema name> *SN*, then *SN* shall be the <schema name> of a schema *S*. The candidate routines of *RI* are the invocable routines (if any) contained in *S*.

c) Case:

   i) If *AL* contains at least one <named argument specification>, then:

      1) Let *MAP* be a mapping of every <SQL parameter name> immediately contained in the <named argument specification>s to the parameter names of *SIR*, such that those names are equivalent. If *MAP* does not exist, then *SIR* is removed from the set of candidate routines.

      2) The ordinal position of the SQL parameter of *SIR* to which *MAP* maps a given <named argument specification> shall be the same for all remaining candidate routines.

      3) Let A_i, 1 (one) $\leq i \leq NA$, be the <named argument SQL argument> contained in the i-th <SQL argument> in *AL* and let P_j, 1 (one) $\leq j \leq NA$, be the corresponding SQL parameter of *SIR*. Each A_i shall have exactly one corresponding P_j.

      4) If there exists an SQL parameter *SPWOD* for which no <named argument specification> exists in *AL* that has an equivalent <SQL parameter name> and the routine descriptor of *PCR* does not contain an indication that *SPWOD* has a default value, then *PCR* is removed from the set of candidate routines.

   ii) Otherwise, let A_i, 1 (one) $\leq i \leq NA$, be the i-th <SQL argument> in *AL* and let P_j, 1 (one) $\leq j \leq NA$, be the corresponding j-th SQL parameter of *SIR*.

d) For each A_i, 1 (one) $\leq i \leq NA$, that is neither a <dynamic parameter specification> nor a <contextually typed value specification> and its corresponding P_j,

   Case:

   i) If the declared type of P_j is a user-defined type, then:

      1) Let ST_i be the set of subtypes of the declared type of A_i.

      2) The Syntax Rules of Subclause 9.7, "Type precedence list determination", are applied with the data type of some type in ST_i as *DATA TYPE*; let TPL_i be the *TYPE PRECEDENCE LIST* returned from the application of those Syntax Rules.

      3) The type designator of the declared type of P_j shall be in TPL_i.

ii) If the declared type of P_j is generic table parameter type, then A_i shall be a <table argument>.

iii) If the declared type of P_j is descriptor parameter type, then A_i shall be a <descriptor argument>.

iv) Otherwise:

1) The Syntax Rules of Subclause 9.7, "Type precedence list determination", are applied with the declared type of A_i as *DATA TYPE*; let TPL_i be the *TYPE PRECEDENCE LIST* returned from the application of those Syntax Rules.

2) The type designator of the declared type of P_j shall be in TPL_i.

e) The declared type of each <value expression> immediately contained in a <generalized expression> shall be a subtype of the structured type identified by the <user-defined type name> simply contained in the <path-resolved user-defined type name> that is immediately contained in <generalized expression>.

f) For all i, 1 (one) $\leq i \leq NA$, A_i shall be neither a <target specification> nor a <named argument specification> that simply contains a <target specification>.

g) For all i, 1 (one) $\leq i \leq NA$,

Case:

i) If A_i is a <generalized expression>, then let TS_i be the data type identified by the <user-defined type name> simply contained in the <path-resolved user-defined type name> that is immediately contained in the <generalized expression>.

ii) Otherwise, if A_i is neither a <dynamic parameter specification> nor a <contextually typed value specification>, then let TS_i be the declared type of A_i.

h) The *subject routine* is defined as follows:

i) For each A_i, 1 (one) $\leq i \leq NA$,

Case:

1) If A_i is a <dynamic parameter specification> or a <contextually typed value specification>, then let V_i be that <dynamic parameter specification> or <contextually typed value specification>, respectively.

2) If A_i is a <table argument> or a <descriptor argument>, then let V_i be A_i.

3) Otherwise, let V_i be a value arbitrarily chosen whose declared type is TS_i.

ii) Let *SRAL* be an <SQL argument list> with *NA* <SQL argument>s derived from V_i, 1 (one) $\leq i \leq NA$, ordered according to the ordinal position i of the corresponding SQL parameter.

iii) The Syntax Rules of Subclause 9.6, "Subject routine determination", are applied with *SRAL* as *SQL ARGUMENT LIST* and the candidate routines of *RI* as *SET OF SQL-INVOKED ROUTINES*; let a set of candidate subject routines *CSR* be the *SET OF SUBJECT ROUTINES* returned from the application of those Syntax Rules.

iv) There shall be at least one candidate subject routine in *CSR*.

1) Case:

A) If *RN* does not contain a <schema name>, then let *S1* be a schema included in *DP* such that *S1* includes at least 1 (one) candidate subject routine of *RI* and there is no other schema that precedes *S1* in *DP* and contains at least 1 (one) candidate subject routine of *RI*. Any routine not contained in *S1* is removed from *CSR*.

B) If *RN* contains a <schema name>, then let *S1* be the schema identified by that <schema name>. *S1* shall include at least 1 (one) candidate subject routine of *RI*. Any routine not contained in *S1* is removed from *CSR*.

2) The set of *candidate subject routines with minimal number of parameters* of *RI* comprises those candidate routines such that there is no other candidate routine that has fewer parameters. Let *CSRMNP* be the set of candidate subject routines with minimal number of parameters of *RI*.

3) Case:

A) If *RI* is not immediately contained in a <static method selection>, then there shall be exactly one candidate subject routine in *CSRMNP*. Let the subject routine *SR* be the candidate subject routine in *CSRMNP*.

B) Otherwise, there shall be an SQL-invoked routine *SIRCR3* in *CSRMNP* such that there is no other candidate subject routine *R2* in *CSRMNP* for which the user-defined type described by the user-defined type descriptor that includes the routine descriptor of *R2* is a subtype of the user-defined type described by the user-defined type descriptor that includes the routine descriptor of *SIRCR3*. The subject routine *SR* is *SIRCR3*.

i) The subject routine of *RI* is *SR*.

j) Let *SRNP* be the number of SQL parameters of *SR*. Let P_i, 1 (one) ≤ i ≤ *SRNP*, be the i-th SQL parameter of *SR*.

Case:

i) If *AL* contains at least one <named argument specification>, then let $NPSA_j$, 1 (one) ≤ j ≤ *SRNP*, be the <named argument SQL argument> immediately contained in the <named argument specification> in *AL* whose <SQL parameter name> is equivalent to P_j.

1) For each SQL parameter P_i, 1 (one) < i ≤ *SRNP*, for which no <named argument specification> exists in *AL* that has an equivalent <SQL parameter name>, let $NPSA_i$ be DEFAULT.

2) Let *XAL* be an <SQL argument list> whose i-th <SQL argument> is $NPSA_i$, 1 (one) ≤ i ≤ *SRNP*.

ii) Otherwise:

1) Let $NPSA_j$, 1 (one) ≤ j ≤ *NA*, be A_j.

2) Let $NPSA_j$, *NA* ≤ j ≤ *SRNP*, be DEFAULT.

3) Let *XAL* be an <SQL argument list> whose i-th <SQL argument> is $NPSA_i$, 1 (one) ≤ i ≤ *SRNP*.

NOTE 394 — If *SRNP* = 0 (zero), then *XAL* is an empty list of SQL arguments.

k) Let XA_i, 1 (one) ≤ i ≤ *SRNP*, be the i-th <SQL argument> in *XAL*.

l) For each i, 1 (one) $\leq i \leq SRNP$,

Case:

i) If P_i is a generic table parameter, then XA_i shall be a table.

ii) If P_i is a descriptor parameter, then XA_i shall be NULL, DEFAULT, or a <descriptor value constructor>.

iii) Otherwise, if XA_i is neither a <dynamic parameter specification> nor a <contextually typed value specification>, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with P_i as *TARGET* and its corresponding XA_i as *VALUE*.

m) The *effective returns data type* of *RI* is defined as follows:

i) Case:

1) If *SR* is a type-preserving function, then let P_j be the result SQL parameter of *SR*. If the corresponding A_i contains a <generalized expression>, then let *RT* be the declared type of the <value expression> contained in the <generalized expression> of A_i; otherwise, let *RT* be the declared type of A_i.

2) Otherwise, let *RT* be the declared type of the result of *SR*.

ii) The *effective returns data type* of *RI* is *RT*.

11) If *SR* is a constructor function, then *RI* shall be simply contained in a <new invocation>.

12) If *SR* is a polymorphic table function, then:

a) *RI* shall be simply contained in a <PTF derived table> *PTFDT* or a <table function invocation>.

b) For each <table argument> *TA* simply contained in *RI*:

i) Let *TAP* be the <table argument proper> simply contained in *TA*. Let *TAPAR* be the parameter of *SR* corresponding to *TA*.

ii) *TA* has a row type *TART* and one or more range variables, to be defined in succeeding rules. The scope of each range variable of *TA* is the <table argument partitioning> and <table argument ordering> contained in *TA*. Each range variable of *TA* has an associated column list that references fields of *TART*.

Case:

1) If *TAP* is a <table function invocation> that invokes a polymorphic table function, then:

A) Let *PTFI* be the <table function invocation> simply contained in *TAP*.

B) If *TAP* simply contains a <table argument correlation name> *TACN*, then let *ASTACN* be

AS *TACN*

Otherwise, let *ASTACN* be the zero-length character string.

C) If *TAP* simply contains <table argument parenthesized derived column list>, then let *TAPDCL* be that <table argument parenthesized derived column list>; otherwise, let *TAPDCL* be the zero-length character string.

D) Let *TAPTP* be the <table primary>

TABLE (*PTFI*) *ASTACN TAPDCL*

E) The row type *TART* of *TA* is the row type of the <table primary> *TAPTP*. The range variables of *TA* are the range variables of *TAPTP*. The associated column lists of those range variables are the associated column lists determined by *TAPTP*.

2) Otherwise:

A) Case:

I) If *TAP* immediately contains a <table or query name> *TAPTOQN*, then let *TAPTAB* be the table identified by *TAPTOQN*. Let *TAPRT* be the row type of *TAPTAB*.

II) If *TAP* immediately contains a <table subquery> *TAPTSQ*, then let *TAPRT* be the row type of *TAPTSQ*.

III) Otherwise, *TAP* is a <table function invocation> that is a <routine invocation> *TAPRI*. Let *TAPSR* be the subject routine of *TAPRI*. *TAPSR* shall be a table function. Let *TAPRT* be the row type specified by the <table function column list> of *TAPSR*.

NOTE 395 — *TAPRI* must be a conventional table function because an earlier case handled the case of a nested polymorphic table function.

B) Let *TAPDEG* be the number of columns in *TAPRT*.

C) The only range variable exposed by *TA* is

Case:

I) If *TA* simply contains a <table argument correlation name> *TACN*, then *TACN*.

II) If *TAP* immediately contains a <table or query name> *TOQN*, then *TOQN*.

III) Otherwise, an implementation-dependent (UV095) range variable.

D) Case:

I) If *TA* has a <table argument parenthesized derived column list> *TAPDCL*, then:

1) The number of <column name>s contained in *TAPDCL* shall be *TAPDEG*.

2) The <column name>s contained in *TAPDCL* shall be mutually distinct.

3) The row type *TART* of *TA* is described by the sequence of (<field name>, <data type>) pairs, where the <field name> in each such pair is the <column name> in the corresponding position in *TAPDCL* and the <data type> in the that pair is the <data type> of the corresponding column of *TAPRT*.

II) Otherwise, the row type *TART* of *TA* is *TAPRT*.

E) The associated column list of the range variable exposed by *TA* is the list of fields of *TAPRT*.

iii) If *TA* contains a <table argument partitioning> *TAPBY*, then:

1) *TAPAR* shall be a table parameter of *SR* with set semantics.

2) Every <column reference> contained in *TAPBY* shall identify a column of *TART*. The columns identified by these <column reference>s are the *partitioning columns* of *TA*.

3) Each partitioning column of *TA* is an operand of a grouping operation. The Syntax Rules of Subclause 9.12, "Grouping operations", are applied.

iv) If *TA* contains a <table argument pruning>, then *TAPAR* shall be a table parameter of *SR* with set semantics that specifies KEEP WHEN EMPTY.

v) If either *TA* or *TAPAR* specifies PRUNE WHEN EMPTY, then *TA* is said to be *pruned*.

vi) If *TA* contains a <table argument ordering> *TAOBY*, then:

1) *TAPAR* shall be a table parameter of *SR* with set semantics.

2) Every <column reference> contained in *TAOBY* shall identify a column of *TART*. The columns identified by these <column reference>s are the *ordering columns* of *TA*.

3) Each ordering column of *TA* is an operand of an ordering operation. The Syntax Rules of Subclause 9.14, "Ordering operations", are applied.

c) No two range variables of the <table argument>s shall be equivalent.

d) If *RI* contains a <co-partition clause> *COCL*, then:

i) No two <range variable>s contained in *COCL* shall be equivalent.

ii) For every <co-partition specification> *COSPEC* contained in *COCL*:

1) Let *NCOSPEC* be the number of <range variable>s contained in *COSPEC*.

2) Each <range variable> contained in *COSPEC* shall reference a range variable exposed by some <table argument> of *RI*. Let $CORV_1$, ..., $CORV_{NCOSPEC}$ be those range variables, and let $CORVTA_1$, ..., $CORVTA_{NCOSPEC}$ be their <table argument>s.

3) Every <table argument> $CORVTA_i$ shall have the same number *NCORVPC* of partitioning columns. *NCORVPC* shall be positive.

4) For any pair of range variables contained in *COSPEC*, corresponding partitioning columns of the respective <table argument>s shall be comparable.

iii) No <table argument> shall be referenced in more than one <co-partition specification>.

e) For each i, 1 (one) $\leq i \leq$ *SRNP*,

Case:

i) If XA_i is DEFAULT, then

Case:

1) If P_i specifies a <parameter default> *PD*, then let PA_i be *PD*.

2) Otherwise, let PA_i be NULL.

ii) Otherwise, let PA_i be XA_i.

f) Let *PAL* be an <SQL argument list> whose *i*-th <SQL argument> is PA_i. *PAL* is the *pre-compilation argument list* of *RI*.

NOTE 396 — For conventional SQL-invoked routines, parameter defaults are supplied in the General Rules of this Subclause. For polymorphic table functions, parameter defaults are supplied here in the pre-compilation argument list. This argument list is subsequently used twice, once to create the compilation argument list (below) and again during the execution phase. In the compilation argument list, any argument that is not a compile-time constant is treated as null. The parameter defaults must be supplied before examining the arguments to decide which are compile-time constants.

g) For each *i*, 1 (one) ≤ *i* ≤ *SRNP*,

Case:

i) If P_i is a table parameter or a descriptor parameter, then let CA_i be PA_i.

ii) If PA_i is a <literal> or an <empty specification>, then let CA_i be PA_i.

iii) If PA_i is a <value expression> that conforms to an implementation-defined (IA062) rule enabling PA_i to be evaluated without accessing SQL-data, then let CA_i be PA_i.

NOTE 397 — For example, an SQL-implementation can evaluate an argument that is "1+1" during compilation prior to execution.

iv) Otherwise, let CA_i be NULL.

NOTE 398 — For compilation only, any argument that cannot be computed statically is treated as null.

h) Let *CAL* be an <SQL argument list> whose *i*-th <SQL argument> is CA_i. *CAL* is the *compilation argument list* of *RI*.

i) The General Rules of Subclause 9.24, "Compilation of an invocation of a polymorphic table function", are applied with *RI* as *ROUTINE INVOCATION*, *SR* as *POLYMORPHIC TABLE FUNCTION*, and *CAL* as *ARGUMENT LIST*; let *PTFDA* be the *PTF DATA AREA* returned from the application of those General Rules. There shall not be an exception condition raised during the execution of those General Rules.

j) The status variable in *PTFDA* shall be *successful completion (00000)*.

k) For every generic table parameter *GTA* of *SR*:

i) Let *FRT* be the PTF descriptor area of the full row type of *GTA* and let *RRT* be the PTF descriptor area of the requested row type of *GTA*.

ii) The COUNT component of the header of *RRT* shall be positive, and shall be the number *NRRT* of SQL item descriptor areas of *RRT* whose LEVEL component is 0 (zero).

iii) The NAME components of the SQL item descriptor areas of *RRT* whose LEVEL component is 0 (zero) shall each have length greater than 0 (zero), be mutually distinct, and be equal to the NAME component of a unique SQL item descriptor area of *FRT* whose LEVEL component is 0 (zero).

l) Let *IRR* be the PTF descriptor area for the initial result row type.

i) The COUNT component of the header of *IRR* shall be the number *NIRR* of SQL item descriptor areas of *IRR* whose LEVEL component is 0 (zero).

ii) The NAME components of the SQL item descriptor areas whose LEVEL is 0 (zero) shall each have length greater than 0 (zero) and be mutually distinct.

iii) Each SQL item descriptor area of *IRR* shall match a declared type.

iv) If there is no generic table parameter of *SR* that specifies PASS THROUGH, then *NIRR* shall be positive.

m) The *initial result row type* of *RI* is described by the sequence of *NIRR* (<field name>, <data type>) pairs, where the <field name> in the *i*-th pair is the value of the NAME component of the *i*-th SQL item descriptor *SIDA* whose LEVEL component is 0 (zero), and whose <data type> is matched by *SIDA*.

n) The columns of the initial result row type of *RI* are the *proper result columns* of *RI*.

13) Let the *static SQL argument list SAL* of *RI* be *XAL*.

14) [13]Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*.

Access Rules

None.

General Rules

1) Let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *VALUE*.

2) Except where explicitly specified, the General Rules of this Subclause are not terminated if an exception condition is raised.

3) Case:

a) If *SAL* is empty, then let the *dynamic SQL argument list DAL* be empty.

b) Otherwise:

i) Each SQL argument A_i, 1 (one) $\le i \le$ *SRNP*, in *SAL* that is not a <default specification> is evaluated, in an implementation-dependent (US037) order, to obtain a value V_i.

ii) For each SQL argument A_i, 1 (one) $\le i \le$ *SRNP*, in *SAL* that is a <default specification>, a value V_i is obtained in an implementation-dependent (US038) order, as follows.

Case:

1) If the descriptor of *SR* does not contain an indication that the SQL parameter P_i has a default value, then V_i is the null value.

2) Otherwise, the <parameter default> contained in the descriptor of *SR* is evaluated to obtain a value V_i.

iii) Let the *dynamic SQL argument list DAL* be the list of values V_i, 1 (one) $\le i \le$ *SRNP*, in order.

iv) If *SR* is type preserving and the null value is substituted for the result parameter, then

Case:

1) If *SR* is a mutator function, then an exception condition is raised: *data exception — null value substituted for mutator subject parameter (2202D)* and no further General Rules of this Subclause are applied.

2) Otherwise, the value of *RI* is the null value and no further General Rules of this Subclause are applied.

v) Case:

1) If *SR* is an instance SQL-invoked method, then:

A) If V_1 is the null value, then the value of *RI* is the null value and the remaining General Rules of this Subclause are not applied.

B) Let *SM* be the set of SQL-invoked methods *M* for which all of the following are true:

I) The <routine name> of *SR* and the <routine name> of *M* have equivalent <qualified identifier>s.

II) *SR* and *M* have the same number *N* of SQL parameters. Let PSR_i, 1 (one) ≤ *i* ≤ *N*, be the *i*-th SQL parameter of *SR* and PM_i, 1 (one) ≤ *i* ≤ *N*, be the *i*-th SQL parameter of *M*.

III) The declared type of the subject parameter of *M* is a subtype of the declared type of the subject parameter of *SR*.

IV) For *j* varying from 2 to *N*, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of PM_j as *TYPE1* and the declared type of PSR_j as *TYPE2*.

NOTE 399 — *SR* is an element of the set *SM*.

C) *SM* is the *set of overriding methods* of *SR* and every SQL-invoked method *M* in *SM* is an *overriding method* of *SR*.

D) Case:

I) If the first SQL argument *A1* in *SAL* contains a <generalized expression>, then let *DT1* be the data type identified by the <user-defined type name> contained in the <generalized expression> of *A1*.

II) Otherwise, let *DT1* be the most specific type of V_1.

E) Let *R* be the SQL-invoked method in *SM* such that there is no other SQL-invoked method *M1* in *SM* for which the type designator of the declared type of the subject parameter of *M1* precedes that of the declared type of the subject parameter of *R* in the type precedence list of *DT1*.

2) Otherwise, let *R* be *SR*.

4) Let *N* and *PN* be the number of values in *DAL*. For *i* ranging from 1 (one) to *PN*:

a) Let V_i be the *i*-th value in *DAL*. Let T_i be the declared type of the *i*-th SQL parameter P_i of *R*.

b) If the declared type of V_i is DATE, and T_i is either TIMESTAMP WITHOUT TIME ZONE or TIMESTAMP WITH TIME ZONE, then V_i is effectively replaced by the value obtained from

```
CAST ( Vi AS TIMESTAMP )
```

NOTE 400 — This choice is possible only if the type precedence list for DATE includes TIMESTAMP; see SR 12) of Subclause 9.7, "Type precedence list determination".

c) 14 Case:

i) [14]If P_i is an input SQL parameter or both an input SQL parameter and an output SQL parameter, then General Rules of Subclause 9.2, "Store assignment", are applied with V_i as *VALUE* and a temporary site *ST* whose declared type is T_i as *TARGET*. Let CPV_i be the value of *ST*. If the General Rules of Subclause 9.2, "Store assignment", raise any exception conditions, then no further General Rules of this Subclause are applied.

ii) Otherwise,

Case:

1) [13]If *R* is an SQL routine, then let CPV_i be the null value.

2) Otherwise, let CPV_i be an implementation-defined (IV161) value of most specific type T_i.

5) [13]If *R* is an external routine, then:

a) Let *P* be the program identified by the external name of *R*.

b) For *i* ranging from 1 (one) to *N*, let P_i be the *i*-th SQL parameter of *R* and let T_i be the declared type of P_i.

Case:

i) If P_i is an input SQL parameter or both an input SQL parameter and an output SQL parameter, then

Case:

1) [14]If P_i is a locator parameter, then CPV_i is replaced by the locator value that uniquely identifies the value of CPV_i.

2) If T_i is a user-defined type, and P_i is not a locator parameter, then:

A) Let FSF_i be the SQL-invoked routine identified by the specific name of the from-sql function associated with P_i in the routine descriptor of *R*. Let RT_i be the declared type of the result of FSF_i.

B) The General Rules of this Subclause are applied with CPV_i as *STATIC SQL ARG LIST* and FSF_i as *SUBJECT ROUTINE*; let *V* be the *VALUE* returned from the application of those General Rules.

NOTE 401 — The *V* returned is not used.

C) Let RV_i be the result of the invocation of FSF_i. CPV_i is replaced by RV_i.

ii) Otherwise,

Case:

1) [14]If P_i is a locator parameter,then CPV_i is replaced with an implementation-dependent (UV096) value of type INTEGER.

2) If T_i is a user-defined type and P_i is not a locator parameter, then:

A) Let FSF_i be the SQL-invoked routine identified by the specific name of the from-sql function associated with P_i in the routine descriptor of *R*. Let RT_i be the declared type of the result of FSF_i.

B) CPV_i is replaced by an implementation-defined (IV162) value of type RT_i.

6) Preserve the current SQL-session context *CSC* and create a new SQL-session context *RSC* derived from *CSC* as follows:

a) The current default catalog name, the current default unqualified schema name, the current default character set name, the SQL-path of the current SQL-session, the text defining the SQL-path, the current default time zone displacement of the current SQL-session, the contents of all SQL dynamic descriptor areas, the text defining the default transform group name, and the text defining the user-defined type name—transform group name pair for each user-defined type explicitly set by the user are set to implementation-defined (IV088) values.

b) The values of the current SQL-session identifier, the SQL-session user identifier, the identities of all instances of global temporary tables, the cursor instance descriptor of all open cursors accessible in the SQL-session, the current constraint mode for each integrity constraint, the current transaction access mode, the current transaction isolation level, the SQL-session collation of each character set known to the SQL-implementation, the current condition area limit, the subject table restriction flag, the restricted subject table name list, and the triggered action indicator are set to their values in *CSC*.

c) The value of the original time zone displacement is set to the value of the current time zone displacement in *CSC*.

d) The set of values of all valid locators in *CSC* is copied to *RSC*.

e) The diagnostics area stack in *CSC* is copied to *RSC* and the General Rules of Subclause 23.2, "Pushing and popping the diagnostics area stack", are applied with "PUSH" as *OPERATION* and the diagnostics area stack in *RSC* as *STACK*.

NOTE 402 — The diagnostics area pushed by the preceding rule is popped effectively when *RSC* is replaced by *CSC* on completion of the routine invocation in GR 16).

f) Case:

i) If *R* is an SQL routine, then the following are copied from *CSC* to *RSC*:

1) The identities of all global temporary tables.

2) The cursor instance descriptors of all open cursors that are global extended dynamic cursors.

3) Prepared statements that have global extended names.

4) 04 SQL descriptor areas that have global extended names.

ii) 13 Otherwise:

1) Let *M* be the <SQL-client module definition> of *P*. The following are copied from *CSC* to *RSC*:

A) The identities of all global temporary tables.

B) The cursor instance descriptors of all open cursors that are global extended dynamic cursors.

C) Prepared statements that have global extended names.

D) SQL descriptor areas that have global extended names.

2) It is implementation-defined (IA063) whether the following are copied from *CSC* to *RSC*:

A) The identities of all instances of created local temporary tables that are referenced in *M*.

B) The identities of all declared local temporary tables that are defined by <temporary table declaration>s that are contained in *M*.

C) The cursor instance descriptors of all open cursors that are not global extended dynamic cursors and whose SQL-client module is *M*.

D) Prepared statements that do not have global extended names.

E) SQL descriptor areas that do not have global extended names.

F) Every currently available result set sequence *RSS*, along with the specific name of an SQL-invoked procedure *SIP* and the name of the invoker of *SIP* for the invocation causing *RSS* to be brought into existence.

g) Indicate in the routine execution context of *RSC*:

i) That the SQL-invoked routine *R* is active.

ii) Whether *R* is an SQL-invoked function.

h) Case:

i) If the SQL-data access indication of *CSC* specifies possibly contains SQL and *R* possibly reads SQL-data or *R* possibly modifies SQL-data, then

Case:

1) If *R* is an external routine, then an exception condition is raised: *external routine exception — reading SQL-data not permitted (38004)*.

2) Otherwise, an exception condition is raised: *SQL routine exception — reading SQL-data not permitted (2F004)*.

ii) If the SQL-data access indication of *CSC* specifies possibly reads SQL-data and *R* possibly modifies SQL-data, then

Case:

1) If *R* is an external routine, then an exception condition is raised: *external routine exception — modifying SQL-data not permitted (38002)*.

2) Otherwise, an exception condition is raised: *SQL routine exception — modifying SQL-data not permitted (2F002)*.

i) Case:

i) If *R* does not possibly contain SQL, then set the SQL-data access indication in the routine execution context of *RSC* to *does not possibly contain SQL*.

ii) If *R* possibly contains SQL, then set the SQL-data access indication in the routine execution context of *RSC* to *possibly contains SQL*.

iii) If *R* possibly reads SQL-data, then set the SQL-data access indication in the routine execution context of *RSC* to *possibly reads SQL-data*.

iv) If *R* possibly modifies SQL-data, then set the SQL-data access indication in the routine execution context of *RSC* to *possibly modifies SQL-data*.

j) The authorization stack of *RSC* is set to a copy of the authorization stack of *CSC*.

k) A copy of the top cell is pushed onto the authorization stack of *RSC*.

l) Case:

i) If *R* is an external routine, then:

1) Case:

A) If the external security characteristic of *R* is IMPLEMENTATION DEFINED, then the current user identifier and the current role name of *RSC* are implementation-defined (IV089).

B) If the external security characteristic of *R* is DEFINER, then the top cell of the authorization stack of *RSC* is set to contain only the routine authorization identifier of *R*.

2) The routine SQL-path of *RSC* is set to the external routine SQL-path of *R*.

ii) Otherwise:

1) If the SQL security characteristic of *R* is DEFINER, then the top cell of the authorization stack of *RSC* is set to contain only the routine authorization identifier of *R*.

2) The routine SQL-path of *RSC* is set to the routine SQL-path of *R*.

m) If the subject routine is an SQL-invoked procedure *SIP*, then:

i) For every received cursor *RC* whose origin is the <specific routine designator> of *SIP*, the cursor declaration descriptor and the cursor instance descriptor of *RC* are destroyed.

ii) Every result set sequence associated with *SIP* is destroyed.

iii) Let *INV* be the invoker of *SIP*. An empty result set sequence *RSS*, for SQL-invoked procedure *SIP* and invoker *INV*, is added to *RSC*.

NOTE 403 — The invoker of SIP is defined in Subclause 4.35.6, "Result sets returned by SQL-invoked procedures".

n) Case:

i) If *R* is an SQL-invoked function, or if *CSC* indicates that a triggered action is executing, the statement timestamp of *RSC* is set to the statement timestamp of *CSC*.

ii) Otherwise, the statement timestamp of *RSC* is set to "not set".

o) *RSC* becomes the current SQL-session context.

7) If the descriptor of *R* includes an indication that a new savepoint level is to be established when *R* is invoked, then a new savepoint level is established.

8) If *R* is an SQL routine, then

Case:

a) If *R* is a null-call function and if any of CPV_i is the null value, then let *RV* be the null value.

b) Otherwise:

i) For *i* ranging from 1 (one) to *PN*, set the value of P_i to CPV_i.

ii) The General Rules of Subclause 9.17, "Executing an <SQL procedure statement>", are applied with the SQL routine body of *R* as *EXECUTING STATEMENT*.

iii) If, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-connection statement, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted (2F003)*.

iv) Case:

1) If the SQL-implementation does not support Feature T272, "Enhanced savepoint management", and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-transaction statement, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted (2F003)*.

2) If, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-transaction statement that is not a <savepoint statement> or a <release savepoint statement>, or is a <rollback statement> that does not specify a <savepoint clause>, then an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted (2F003)*.

v) If the SQL-implementation does not support Feature T651, "SQL-schema statements in SQL routines", and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-schema statement, an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted (2F003)*.

vi) If the SQL-implementation does not support Feature T652, "SQL-dynamic statements in SQL routines", and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-dynamic statement, an exception condition is raised: *SQL routine exception — prohibited SQL-statement attempted (2F003)*.

vii) If the SQL-data access indication of *RSC* specifies *possibly contains SQL* and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-statement that possibly reads SQL-data, or an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *SQL routine exception — reading SQL-data not permitted (2F004)*.

viii) If the SQL-data access indication of *RSC* specifies *possibly reads SQL-data* and, before the completion of the execution of the SQL routine body of *R*, an attempt is made to execute an SQL-statement that possibly modifies SQL-data then an exception condition is raised: *SQL routine exception — modifying SQL-data not permitted (2F002)*.

ix) If *R* is an SQL-invoked function, then

Case:

1) If no <return statement> is executed before completion of the execution of the SQL routine body of *R*, then an exception condition is raised: *SQL routine exception — function executed no return statement (2F005)*.

2) Otherwise, let *RV* be the returned value of the execution of the SQL routine body of *R*.

NOTE 404 — "Returned value" is defined in Subclause 16.2, "<return statement>".

x) If *R* is an SQL-invoked procedure, then for each SQL parameter of *R* that is an output SQL parameter or both an input SQL parameter and an output SQL parameter, set the value of CPV_i to the value of P_i.

9) If *R* is an external routine, then:

a) 04 The method and time of binding of *P* to the schema that includes *R* is implementation-defined (IW167).

b) If *R* specifies PARAMETER STYLE SQL, then:

i) Case:

1) If *R* is an SQL-invoked function, then the effective SQL parameter list *ESPL* of *R* is set as follows:

A) If *R* is a collection-returning external function with the element type being a row type, then let *FRN* be the degree of the element type; otherwise, let *FRN* be 1 (one).

B) For *i* ranging from 1 (one) to *PN*, the *i*-th entry in *ESPL* is set to CPV_i.

C) For *i* ranging from *PN*+1 to *PN*+*FRN*, the *i*-th entries in *ESPL* are the *result data items*.

D) For *i* ranging from (*PN*+*FRN*)+1 to (*PN*+*FRN*)+*N*, the *i*-th entry in *ESPL* is the *SQL indicator argument* corresponding to $CPV_{i-(PN+FRN)}$.

E) For *i* ranging from (*PN*+*FRN*)+*N*+1 to (*PN*+*FRN*)+*N*+*FRN*, the *i*-th entries in *ESPL* are the SQL indicator arguments corresponding to the result data items.

F) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+1, the *i*-th entry in *ESPL* is the *exception data item*.

G) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+2, the *i*-th entry in *ESPL* is the *routine name text item*.

H) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+3, the *i*-th entry in *ESPL* is the *specific name text item*.

I) For *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+4, the *i*-th entry in *ESPL* is the *message text item*.

J) If *R* is a collection-returning external function, then for *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+5, the *i*-th entry in *ESPL* is the *save area data item* and for *i* equal to (*PN*+*FRN*)+(*N*+*FRN*)+6, the *i*-th entry in *ESPL* is the *call type data item*.

K) Set the values of the SQL indicator arguments corresponding to the result data items (that is, SQL argument value list entries from (*PN*+*FRN*)+*N*+1 through (*PN*+*FRN*)+*N*+*FRN*, inclusive) to 0 (zero).

L) For *i* ranging from 1 (one) to *PN*, if CPV_i is the null value, then set entry (*PN*+*FRN*)+*i* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to −1 (negative one); otherwise, set entry (*PN*+*FRN*)+*i* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to 0 (zero).

M) If *R* is a collection-returning external function, then set the value of the save area data item (that is, SQL argument value list entry (*PN*+*FRN*)+(*N*+*FRN*)+5) to an implementation-defined (IV090) value and set the value of the call type data item (that is, SQL argument value list entry (*PN*+*FRN*)+(*N*+*FRN*)+6) to −1 (negative one).

2) Otherwise, the effective SQL parameter list *ESPL* of *R* is set as follows:

A) For *i* ranging from 1 (one) to *PN*, the *i*-th entry in *ESPL* is CPV_i.

B) For *i* ranging from *PN*+1 to *PN*+*N*, the *i*-th entry in *ESPL* is the *SQL indicator argument* corresponding to CPV_{i-PN}.

C) For *i* equal to (*PN*+*N*)+1, the *i*-th entry in *ESPL* is the *exception data item*.

D) For *i* equal to (*PN*+*N*)+2, the *i*-th entry in *ESPL* is the *routine name text item*.

E) For *i* equal to (*PN*+*N*)+3, the *i*-th entry in *ESPL* is the *specific name text item*.

F) For *i* equal to (*PN*+*N*)+4, the *i*-th entry in *ESPL* is the *message text item*.

G) For *i* ranging from 1 (one) to *PN*, if CPV_i is the null value, then set entry *PN*+*i* in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to −1 (negative one); otherwise, set entry *PN*+*i* in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) to 0 (zero).

ii) The exception data item is set to '00000'.

iii) The routine name text item is set to the <schema qualified name> of the routine name of *R*.

iv) The specific name text item is set to the <qualified identifier> of the specific name of *R*.

v) The message text item is set to the zero-length character string.

c) If *R* specifies PARAMETER STYLE GENERAL, then the effective SQL parameter list *ESPL* of *R* is set as follows:

i) If *R* is not a null-call function and, for *i* ranging from 1 (one) to *PN*, any of CPV_i is the null value, then an exception condition is raised: *external routine invocation exception — null value not allowed (39004)*.

ii) For *i* ranging from 1 (one) to *PN*, if no CPV_i is the null value, then for *j* ranging from 1 (one) to *PN*, the *j*-th entry in *ESPL* is set to CPV_j.

d) 13 If *R* specifies DETERMINISTIC and if different executions of *P* with identical SQL argument value lists do not produce identical results, then the results are implementation-dependent (UA055).

e) Let *EN* be the number of entries in *ESPL*. Let ESP_i be the *i*-th effective SQL parameter in *ESPL*.

f) For *i* ranging from 1 (one) through *EN*, let PT_i be the <data type> of ESP_i.

g) Case:

i) If *R* is a null-call function and if any of CPV_i is the null value, then *P* is assumed to have been executed.

ii) Otherwise:

1) 13 If *R* is not a collection-returning external function, then *P* is executed with a list of *EN* parameters PD_i whose parameter names are PN_i and whose values are set as follows:

A) If the language of *R* is ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, then let the *operative data type correspondences table* be Table 21, "Data type correspondences for Ada", Table 22, "Data type correspondences for C", Table 23, "Data type correspondences for COBOL", Table 24, "Data type correspondences for Fortran", Table 25, "Data type correspondences for

M", Table 26, "Data type correspondences for Pascal", or Table 27, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

B) For *i* varying from 1 (one) to *EN*, the host language data type DT_i of PD_i is the data type listed in the host data type column of the row in the operative data type correspondences table whose value in the SQL data type column corresponds to PT_i.

2) The General Rules of Subclause 9.4, "Passing a value from the SQL-server to a host language", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and the value of ESP_i as *SQL VALUE*; let PD_i be the *HOST VALUE* returned from the application of those General Rules.

3) 13 If *R* is an array-returning external function, then:

A) Let *AR* be an array whose declared type is the result data type of *R*.

B) The General Rules of Subclause 9.21, "Execution of array-returning external functions", are applied with *AR* as *ARRAY*, *ESPL* as *EFFECTIVE SQL PARAMETER LIST*, and *P* as *PROGRAM*.

4) If *R* is a multiset-returning external function, then:

A) Let *MU* be a multiset whose declared type is the result data type of *R*.

B) The General Rules of Subclause 9.22, "Execution of multiset-returning external functions", are applied with *MU* as *MULTISET*, *ESPL* as *EFFECTIVE SQL PARAMETER LIST*, and *P* as *PROGRAM*.

5) 13 15 If the SQL-data access indication of *RSC* specifies *does not possibly contain SQL* and, before the completion of any execution of *P*, an attempt is made to execute an SQL-statement, then an exception condition is raised: *external routine exception — containing SQL not permitted (38001)*.

6) If, before the completion of any execution of *P*, an attempt is made to execute an SQL-connection statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted (38003)*.

7) Case:

A) If the SQL-implementation does not support Feature T272, "Enhanced savepoint management", and, before the completion of the execution of *P*, an attempt is made to execute an SQL-transaction statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted (38003)*.

B) If, before the completion of the execution of *P*, an attempt is made to execute an SQL-transaction statement that is not <savepoint statement> or <release savepoint statement>, or is a <rollback statement> that does not specify a <savepoint clause>, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted (38003)*.

8) If the SQL-implementation does not support Feature T653, "SQL-schema statements in external routines", and, before the completion of any execution of *P*, an attempt is made to execute an SQL-schema statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted (38003)*.

9) If the SQL-implementation does not support Feature T654, "SQL-dynamic statements in external routines", and, before the completion of any execution of *P*, an attempt is made to execute an SQL-dynamic statement, then an exception condition is raised: *external routine exception — prohibited SQL-statement attempted (38003)*.

10) If the SQL-data access indication of *RSC* specifies *possibly contains SQL* and, before the completion of any execution of *P*, an attempt is made to execute an SQL-statement that possibly reads SQL-data, or an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *external routine exception — reading SQL-data not permitted (38004)*.

11) If the SQL-data access indication of *RSC* specifies *possibly reads SQL-data* and, before the completion of any execution of *P*, an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *external routine exception — modifying SQL-data not permitted (38002)*.

12) If *P* is not a conforming program according to the standard for *R*, then the results of any execution of *P* are implementation-dependent (UA054).

h) After the completion of any execution of *P*:

i) It is implementation-defined (IA066) whether:

1) Every instance of created local temporary tables and every instance of declared local temporary tables that is associated with *RSC* is destroyed.

2) For every prepared statement *PS* prepared by *P* in the current SQL-transaction that has not been deallocated by *P*:

A) Let *SSN* be the <SQL statement name> that identifies *PS*.

B) The following SQL-statement is effectively executed:

```
DEALLOCATE PREPARE SSN
```

ii) For *i* varying from 1 (one) to *EN*, the General Rules of Subclause 9.3, "Passing a value from a host language to the SQL-server", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and the value of PD_i as *HOST VALUE*; let ESP_i be the *SQL VALUE* returned from the application of those General Rules.

iii) If *R* specifies PARAMETER STYLE SQL, then

Case:

1) If the exception data item has the value '00000', then the execution of *P* was successful.

2) If the first two characters of the exception data item are equal to the SQLSTATE class code for *warning (01000)*, then a completion condition is raised: *warning (01000)*, using a subclass code equal to the final three characters of the value of the exception data item.

3) Otherwise, an exception condition is raised using a class code equal to the first two characters of the value of the exception data item and a subclass code equal to the final three characters of the value of the exception data item.

iv) If the exception data item is not '00000' and *R* specified PARAMETER STYLE SQL, then the message text item is stored in the first diagnostics area.

i) If *R* is an SQL-invoked function, then:

i) Case:

1) If *R* is an SQL-invoked method whose routine descriptor does not include a STATIC indication and if CPV_1 is the null value, then let *RDI* be the null value.

2) If *R* is a null-call function, *R* is not a collection-returning external function, and if any of CPV_i is the null value, then let *RDI* be the null value.

3) [13]If *R* is not a null-call function, *R* is not a collection-returning external function, *R* specifies PARAMETER STYLE SQL, and entry (*PN*+1)+*N*+1 in *ESPL* (that is, SQL indicator argument *N*+1 corresponding to the result data item) is negative, then let *RDI* be the null value.

4) Otherwise:

A) Case:

I) If *R* is not a collection-returning external function, *R* specifies PARAMETER STYLE SQL, and entry (*PN*+1)+*N*+1 in *ESPL* (that is, SQL indicator argument *N*+1 corresponding to the result data item) is not negative, then let *ERDI* be the value of the result data item.

II) If *R* is an array-returning external function, and *R* specifies PARAMETER STYLE SQL, then let *ERDI* be *AR*.

III) [15]If *R* is a multiset-returning external function, and *R* specifies PARAMETER STYLE SQL, then let *ERDI* be *MU*.

IV) If *R* specifies PARAMETER STYLE GENERAL, then let *HLV* be the host language value returned from *P*. Let *RDT* be the <returns data type> of *R*. If *RDT* contained <locator indication>, then let *EDT* be INTEGER; otherwise, let *EDT* be the <data type> contained in *RDT*. The General Rules of Subclause 9.3, "Passing a value from a host language to the SQL-server", are applied with the language of *R* as *LANGUAGE*, *EDT* as *SQL TYPE*, and *HLV* as *HOST VALUE*; let *ERDI* be the *SQL VALUE* returned from the application of those General Rules.

NOTE 405 — The host language value returned from *P* is passed to the SQL-implementation in an implementation-dependent manner. An argument value list entry is not used for this purpose.

B) Case:

I) If the routine descriptor of *R* indicates that the return value is a locator, then

Case:

1) If *ERDI* is an invalid locator, then an exception condition is raised: *locator exception — invalid specification (0F001)*, evaluation of GR 9) is terminated immediately, and evaluation continues with GR 12).

2) If *RT* is a binary large object type, then let *RDI* be the binary large object value corresponding to *ERDI*.

3) If *RT* is a character large object type, then let *RDI* be the large object character string corresponding to *ERDI*.

4) If *RT* is an array type, then let *RDI* be the array value corresponding to *ERDI*.

5) If *RT* is a multiset type, then let *RDI* be the multiset value corresponding to *ERDI*.

6) If *RT* is a user-defined type, then let *RDI* be the user-defined type value corresponding to *ERDI*.

II) Otherwise, if *R* specifies <result cast>, then let *CRT* be the <data type> specified in <result cast>; otherwise, let *CRT* be the <returns data type> of *R*.

Case:

1) If *R* specifies <result cast> and the routine descriptor of *R* indicates that the <result cast> has a locator indication, then

Case:

a) If *CRT* is a binary large object type, then let *RDI* be the binary large object value corresponding to *ERDI*.

b) If *CRT* is a character large object type, then let *RDI* be the large object character string corresponding to *ERDI*.

c) If *CRT* is an array type, then let *RDI* be the array value corresponding to *ERDI*.

d) If *CRT* is a multiset type, then let *RDI* be the multiset value corresponding to *ERDI*.

e) If *CRT* is a user-defined type, then let *RDI* be the user-defined type value corresponding to *ERDI*.

2) Otherwise,

Case:

a) If *CRT* is a user-defined type, then:

i) Let *TSF* be the SQL-invoked routine identified by the specific name of the to-sql function associated with the result of *R*.

ii) Case:

1) If *TSF* is an SQL-invoked method, then:

A) If *R* is a type-preserving function, then let *MAT* be the most specific type of the value of the argument substituted for the result SQL parameter of *R*; otherwise, let *MAT* be *CRT*.

B) The General Rules of this Subclause are applied with a static SQL argument list whose first element is the value returned by the invocation of `MAT()` and whose second element is *ERDI* as *STATIC SQL ARG LIST* and *TSF* as *SUBJECT ROUTINE*; let *V* be the *VALUE* returned from the application of those General Rules.

NOTE 406 — The *V* returned is not used.

2) The General Rules of this Subclause are applied with *ERDI* as *STATIC SQL ARG LIST* and *TSF* as *SUBJECT ROUTINE*; let *V* be the *VALUE* returned from the application of those General Rules.

NOTE 407 — The *V* returned is not used.

iii) Let *RDI* be the result of invocation of *TSF*.

b) [14]Otherwise, let *RDI* be *ERDI*.

ii) If *R* specified a <result cast>, then let *RT* be the <returns data type> of *R* and let *RV* be the result of:

```
CAST ( RDI AS RT )
```

Otherwise, let *RV* be *RDI*.

j) If *R* is an SQL-invoked procedure, then for each P_i, 1 (one) ≤ i ≤ *PN*, that is an output SQL parameter or both an input SQL parameter and an output SQL parameter,

Case:

i) If *R* specifies PARAMETER STYLE SQL and entry (*PN*+*i*) in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) is negative, then CPV_i is set to the null value.

ii) [13]If *R* specifies PARAMETER STYLE SQL, and entry (*PN*+*i*) in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) is not negative, and a value was not assigned to the *i*-th entry in *ESPL*, then CPV_i is set to an implementation-defined (ID111) value of type T_i.

iii) Otherwise:

NOTE 408 — In this case, either *R* specifies PARAMETER STYLE SQL and entry (*PN*+*i*) in *ESPL* (that is, the *i*-th SQL indicator argument corresponding to CPV_i) is not negative and a value was assigned to the *i*-th entry in *ESPL*, or else *R* specifies PARAMETER STYLE GENERAL.

1) Let EV_i be the *i*-th entry in *ESPL*. Let T_i be the declared type of P_i.

2) Case:

A) If P_i is a locator parameter, then

Case:

I) If *ERDI* is an invalid locator, then an exception condition is raised: *locator exception — invalid specification (0F001)*, evaluation of GR 9) is terminated immediately, and evaluation continues with GR 12).

II) If T_i is a binary large object type, then CPV_i is set to the binary large object value corresponding to EV_i.

III) If T_i is a character large object type, then CPV_i is set to the large object character string corresponding to EV_i.

IV) If T_i is an array type, then CPV_i is set to the array value corresponding to EV_i.

V) If T_i is a multiset type, then CPV_i is set to the multiset value corresponding to EV_i.

VI) If T_i is a user-defined type, then CPV_i is set to the user-defined type value corresponding to EV_i.

B) If T_i is a user-defined type, then:

I) Let TSF_i be the SQL-invoked function identified by the specific name of the to-sql function associated with P_i in the routine descriptor of R.

II) Case:

1) If TSF_i is an SQL-invoked method, then the General Rules of this Subclause are applied with a static SQL argument list whose first element is the value returned by the invocation of $T_i()$ and whose second element is EV_i as *STATIC SQL ARG LIST* and TSF_i as *SUBJECT ROUTINE*; let V be the *VALUE* returned from the application of those General Rules.

NOTE 409 — The V returned is not used.

2) Otherwise, the General Rules of this Subclause are applied with EV_i as *STATIC SQL ARG LIST* and TSF_i as *SUBJECT ROUTINE*; let V be the *VALUE* returned from the application of those General Rules.

NOTE 410 — The V returned is not used.

III) CPV_i is set to the result of an invocation of TSF_i.

C) [14] Otherwise, CPV_i is set to EV_i.

10) Case:

a) If R is an SQL-invoked function, then:

i) If R is a type-preserving function, then:

1) Let *MAT* be the most specific type of the value of the argument substituted for the result SQL parameter of R.

2) If *RV* is not the null value and the most specific type of *RV* is not compatible with *MAT*, then an exception condition is raised: *data exception — most specific type mismatch (2200G)*.

ii) Let *ERDT* be the effective returns data type of the <routine invocation>.

iii) [14] The General Rules of Subclause 9.2, "Store assignment", are applied with *RV* as *VALUE* and a temporary site *ST* whose declared type is *ERDT* as *TARGET*. Let *RES*, the result of the <routine invocation>, be the value of *ST*.

b) Otherwise, let *RES* be the null value, and, for each SQL parameter P_i of R that is an output SQL parameter or both an input SQL parameter and an output SQL parameter whose corresponding <SQL argument> A_i is not explicitly or implicitly a <default specification>, let TS_i be the <target specification> of A_i.

Case:

i) If TS_i is a <host parameter specification> or an <embedded variable specification>, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied with TS_i as *TARGET* and CPV_i as *VALUE*.

ii) 04 If TS_i is an <SQL parameter reference>, a <column reference>, or a <target array element specification>, then

NOTE 411 — The <column reference> can only be a new transition variable column reference.

Case:

1) If <target array element specification> is specified, then

Case:

A) If the value of TS_i is the null value, then an exception condition is raised: *data exception — null value in array target (2200E)*.

B) Otherwise:

I) Let *N* be the maximum cardinality of TS_i.

II) Let *M* be the cardinality of the value of TS_i.

III) Let *I* be the value of the <simple value specification> immediately contained in TS_i.

IV) Let *EDT* be the element type of TS_i.

V) Case:

1) If *I* is greater than zero and less than or equal to *M*, then the value of TS_i is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

a) For *j* varying from 1 (one) to *I*−1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in TS_i.

b) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with the *I*-th element of *A* as *TARGET* and the value of CPV_i as *VALUE*.

2) If *I* is greater than *M* and less than or equal to *N*, then the value of TS_i is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

a) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in TS_i.

b) For *j* varying from *M*+1 to *I*, the *j*-th element in *A* is the null value.

c) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with the *I*-th element of *A* as *TARGET* and the value of CPV_i as *VALUE*.

3) Otherwise, an exception condition is raised: *data exception — array element error (2202E)*.

2) Otherwise:

A) Let *PTEMP* be the null value.

B) [14]The General Rules of Subclause 9.2, "Store assignment", are applied with TS_i as *TARGET* and CPV_i as *VALUE*.

11) If *RSS* is not empty, then let *PR* be the descriptor of *SIP*.

a) Let *MAX* be the maximum number of returned result sets included in *PR*.

b) [13]Let *OPN* be the actual number of returned result sets included in *RSS*.

c) Case:

i) If *OPN* is greater than *MAX*, then:

1) Let *RTN* be *MAX*.

2) A completion condition is raised: *warning — attempt to return too many result sets (0100E)*.

ii) Otherwise, let *RTN* be *OPN*.

d) [13]For each *i*, 1 (one) ≤ *i* ≤ *RTN*, let FRC_i be the with-return cursor of the *i*-th returned result set RS_i in *RSS*.

e) For each *i*, 1 (one) ≤ *i* ≤ *RTN*,

Case:

i) If FRC_i is a scrollable cursor, then the initial cursor position of RS_i is the current cursor position of FRC_i.

ii) Otherwise,

Case:

1) If FRC_i is positioned before some row in RS_i, then let *RN* be the number of rows ordered before that row in RS_i.

2) If FRC_i is positioned on some row in RS_i, then let *RN* be the ordinal position of that row in RS_i.

3) Otherwise, let *RN* be the number of rows in RS_i.

The first *RN* rows are deleted from RS_i and the initial cursor position of RS_i is before the first row of the rows that remain.

f) [13]A completion condition is raised: *warning — result sets returned (0100C)*.

12) If *R* is an SQL routine, then for every open cursor *CR* that was opened during the execution of *R* and that is either a declared cursor or a local extended dynamic cursor, General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and SAVE as *DISPOSITION*.

13) If *R* is an external routine, then it is implementation-defined (IA067) whether, for every open cursor *CR* that was opened during the execution of *R* and that is either a declared cursor or a local extended dynamic cursor:

a) The General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and SAVE as *DISPOSITION*.

b) The cursor instance descriptor of *CR* is destroyed.

14) [13]Prepare *CSC* to become the current SQL-session context:

a) Set the value of the current constraint mode for each integrity constraint in *CSC* to the value of the current constraint mode for each integrity constraint in *RSC*.

b) The value of the current transaction access mode in *CSC* is set to the value of the current transaction access mode in *RSC*.

c) The value of the current transaction isolation level in *CSC* is set to the value of the current transaction isolation level in *RSC*.

d) The set of values of all valid locators in *CSC* is replaced by the set of values of all valid locators in *RSC*.

e) The value of the subject table restriction flag and the restricted subject table name list of *CSC* are set to the value of the subject table restriction flag and the restricted subject table name list of *RSC*.

f) Set the value of the current condition area limit in *CSC* to the value of the current condition area limit *CAL* in *RSC*.

g) If *R* is an SQL-invoked function or if *CSC* indicates that a triggered action is executing, then the value of the statement timestamp in *CSC* is set to the value of the statement timestamp in *RSC*.

h) For each occupied condition area *CA* in the first diagnostics area of *RSC*, if the value of RETURNED_SQLSTATE in *CA* does not represent *successful completion (00000)*, then

Case:

i) If the number of occupied condition areas in the first diagnostics area *DA1* in *CSC* is less than *CAL*, then *CA* is copied to the first vacant condition area in *DA1*.

NOTE 412 — This causes the first vacant condition area in *DA1* to become occupied.

ii) Otherwise, the value of MORE in the statement information area of *DA1* is set to 'Y'.

i) The identities of all instances of global temporary tables in *CSC* are replaced with the identities of the instances of global temporary tables in *RSC*.

j) The top cell is removed from the authorization stack of *RSC* and the authorization stack of *CSC* is set to a copy of the authorization stack of *RSC*.

NOTE 413 — The copying of *RSC*'s authorization stack into *CSC* is necessary in order to carry back any change in the SQL-session user identifier.

k) If the subject routine is an SQL-invoked procedure, then the result set sequence *RSS* is added to *CSC*.

NOTE 414 — *RSS* is now available for reference by an <allocate received cursor statement>.

NOTE 415 — All other items in *CSC* retain the values they had before the creation of *RSC*.

15) If *R* is an SQL-invoked function or if *R* is an SQL-invoked procedure and the descriptor of *R* includes an indication that a new savepoint level is to be established when *R* is invoked, then the current savepoint level is destroyed.

16) *CSC* becomes the current SQL-session context.

17) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *RES* as *VALUE*.

Conformance Rules

None.

9.19 Processing a method invocation

Function

Process a <method invocation>.

Subclause Signature

```
"Processing a method invocation" [Syntax Rules] (
  Parameter: "METHOD INVOCATION"
) Returns: "ROUTINE INVOCATION", "GENERALIZED OR CONSTRUCTOR", and "SQLPATH"
```

METHOD INVOCATION — a <method invocation>.

ROUTINE INVOCATION — a <routine invocation> that is derived from METHOD INVOCATION and its arguments.

GENERALIZED OR CONSTRUCTOR — 'M' or 'C', identifying whether the METHOD INVOCATION is a generalized <method invocation> or a constructor <method invocation>.

SQLPATH — an SQL-path that includes the <schema name> of the schema that contains the descriptor of the user-defined type of the <value expression primary> that is contained in the METHOD INVOCATION.

Syntax Rules

1) Let *OR* be the *METHOD INVOCATION* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *ROUTINE INVOCATION*, *GENERALIZED OR CONSTRUCTOR*, and *SQLPATH*.

2) Let *DIGI* be the <direct invocation> or <generalized invocation> immediately contained in *OR*, let *VEP* be the <value expression primary> immediately contained in *DIGI*, and let *MN* be the <method name> immediately contained in *DIGI*.

3) [13] The declared type of *VEP* shall be a user-defined type. Let *UDT* be that user-defined type.

4) Case:

   a) If *OR* contains an <SQL argument list> *ORAL*, then let *n* be the number of <SQL argument>s immediately contained in *ORAL* and let *AL* be:

      , A_1 ..., A_n

      where A_i, 1 (one) ≤ *i* ≤ *n*, are the <SQL argument>s immediately contained in *ORAL*, taken in order of their ordinal position in *ORAL*.

   b) Otherwise, let *AL* be the zero-length character string.

5) Case:

   a) If *OR* is immediately contained in <new invocation>, then let *TP* be an SQL-path containing the <schema name> of the schema that includes the descriptor of *UDT*.

   b) Otherwise, let *TP* be an SQL-path, arbitrarily defined, containing the <schema name> of every schema that includes a descriptor of a supertype or subtype of *UDT*.

6) Case:

a) If *OR* immediately contains <generalized invocation>, then let *DT* be the <data type> simply contained in the <generalized invocation>, let *GENORCONS* be 'M', and let *RI* be:

```
MN (VEP AS DT AL)
```

b) Otherwise:

i) Let *RI* be:

```
MN ( VEP AL )
```

ii) Case:

1) If *OR* is immediately contained in <new invocation>, then let *GENORCONS* be 'C'.

2) Otherwise, let *GENORCONS* be 'M'

7) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *RI* as *ROUTINE INVOCATION*, *GENORCONS* as *GENERALIZED OR CONSTRUCTOR*, and *TP* as *SQLPATH*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.20 Transformation of query specifications

This Subclause is modified by Subclause 8.5, "Transformation of query specifications", in ISO/IEC 9075-4.

Function

Transform <query specification>s to a normalized form before further processing.

Subclause Signature

```
"Transformation of query specifications" [Syntax Rules] (
  Parameter: "QUERY SPEC IN"
) Returns: "QUERY SPEC OUT"
```

QUERY SPEC IN — a <query specification> that defines a grouped, windowed query.

QUERY SPEC OUT — a normalized <query specification> that is the result of invoking this subclause using this signature.

Syntax Rules

1) Let *GWQ* be the *QUERY SPEC IN* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *QUERY SPEC OUT*.

2) *GWQ* shall be a grouped, windowed query.

3) If *GWQ* contains an <in-line window specification>, then:

   a) Let *NWF* be the number of <window function>s simply contained in *GWQ*.

   b) For each <window function> WF_i, 0 (zero) ≤ i ≤ *NWF*, simply contained in *GWQ*, the Syntax Rules of Subclause 9.23, "Evaluation and transformation of <window function>", are applied with WF_i as *WINFUNC* and *GWQ* as *QUERY SPEC IN*; let *GWQ* be the *TRANSFORM* returned from the application of those Syntax Rules.

4) If the <select list> of *GWQ* immediately contains <asterisk> or simply contains <qualified asterisk>, then Syntax Rules of this Subclause are applied with *GWQ* as *QUERY SPEC IN*; let *GWQ2* be the *QUERY SPEC OUT* returned from the application of those Syntax Rules; otherwise, let *GWQ2* be *GWQ*.

5) Let *SL*, *FC*, *WC*, *GBC*, *HC*, and *WIC* be the <select list>, <from clause>, <where clause>, <group by clause>, <having clause>, and <window clause>, respectively, of *GWQ2*. If any of <where clause>, <group by clause>, or <having clause>, are missing, then let *WC*, *GBC*, and *HC*, respectively, be the zero-length character string. Let *SQ* be the <set quantifier> immediately contained in the <query specification> of *GWQ2*, if any; otherwise, let *SQ* be the zero-length character string.

   NOTE 416 — *GWQ2* cannot lack a <window clause>, since the syntactic transformation of Subclause 9.23, "Evaluation and transformation of <window function>", will create one if there is not one in *GWQ* already.

6) Let *N1* be the number of <set function specification>s simply contained in *GWQ2*.

7) Let SFS_i, 1 (one) ≤ i ≤ *N1*, be an enumeration of the <set function specification>s simply contained in *GWQ2*.

8) Let $SFSI_i$, 1 (one) ≤ i ≤ *N1*, be a list of <identifier>s that are distinct from each other and distinct from all <identifier>s contained in *GWQ2*.

9) If *N1* = 0 (zero), then let *SFSL* be the zero-length character string; otherwise, let *SFSL* be:

```
SFS1 AS SFSI1,
SFS2 AS SFSI2,
...,
SFSN1 AS SFSIN1
```

10) Let *HCNEW* be obtained from *HC* by replacing each <set function specification> SFS_i by the corresponding <identifier> $SFSI_i$.

11) Let *N2* be the number of <column reference>s that are not outer references and are contained in *SL* or *WIC* without an intervening <query expression> or <set function specification>.

12) Let CR_j, 1 (one) ≤ *j* ≤ *N2*, be an enumeration of the <column reference>s that are not outer references and are contained in *SL* or *WIC* without an intervening <query expression> or <set function specification>.

13) Let CRI_j, 1 (one) ≤ *j* ≤ *N2*, be a list of <identifier>s that are distinct from each other, distinct from all identifiers in *GWQ2*, and distinct from all $SFSI_i$.

14) If *N2* = 0 (zero), then let *CRL* be the zero-length character string; otherwise, let *CRL* be:

```
CR1 AS CRI1, CR2 AS CRI2, ..., CRN2 AS CRIN2
```

15) Let *N3* be the number of <derived column>s simply contained in *SL* that do not specify <as clause>.

16) Let $DCOL_k$, 1 (one) ≤ *k* ≤ *N3*, be the <derived column>s simply contained in *SL* that do not specify an <as clause>. For each *k*, let $COLN_k$ be the <column name> determined as follows.

Case:

a) If $DCOL_k$ is a single column reference, then $COLN_k$ is the <column name> of the column designated by the column reference.

b) 04 If $DCOL_k$ is a single SQL parameter reference, then $COLN_k$ is the <SQL parameter name> of the SQL parameter designated by the SQL parameter reference.

c) Otherwise, $COLN_k$ is an implementation-dependent (UV122) <column name>.

17) Let *SL2* be obtained from *SL* by replacing each <derived column> $DCOL_k$ by

```
DCOLk AS COLNk
```

18) Let *GWQN* be an arbitrary <identifier>.

19) Let *SLNEW* be the <select list> obtained from *SL2* by replacing each simply contained <set function specification> SFS_i by $GWQN.SFSI_i$ and replacing each <column reference> CR_j by $GWQN.CRI_j$.

20) Let *WICNEW* be the <window clause> obtained from *WIC* by replacing each <set function specification> SFS_i by $GWQN.SFSI_i$ and by replacing each <column reference> CR_j by $GWQN.CRI_j$.

21) If either *SFSL* or *CRL* is the zero-length character string, then let *COMMA* be the zero-length character string; otherwise, let *COMMA* be "," (a <comma>).

22) *GWQ* is equivalent to the following <query specification>:

```
SELECT SLNEW
FROM ( SELECT SQ SFSL COMMA CRL
       FC
       WC
       GBC
```

```
        HC ) AS GWQN
    WICNEW
```

23) Let *XFORM* be *GWQ*.

24) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *XFORM* as *QUERY SPEC OUT*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.21 Execution of array-returning external functions

This Subclause is modified by Subclause 8.2, "Execution of array-returning functions", in ISO/IEC 9075-13.

Function

Define the execution of an external function that returns an array value.

Subclause Signature

```
"Execution of array-returning external functions" [General Rules] (
  Parameter: "ARRAY",
  Parameter: "EFFECTIVE SQL PARAMETER LIST",
  Parameter: "PROGRAM"
)
```

ARRAY — a site with a declared type that is an array type.

EFFECTIVE SQL PARAMETER LIST — a list of values to be supplied as parameters for execution of the external routine identified by PROGRAM.

PROGRAM — an external routine that returns an array value.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *AR* be the *ARRAY*, let *ESPL* be the *EFFECTIVE SQL PARAMETER LIST*, and let *P* be the *PROGRAM* in an application of the General Rules of this Subclause.

2) Let *ARC* be the cardinality of *AR*.

3) Let *EN* be the number of entries in *ESPL*.

4) Let ESP_i, 1 (one) $\leq i \leq EN$, be the *i*-th parameter in *ESPL*.

5) Let *FRN* be the number of result data items.

NOTE 417 — The number of result data items is defined in Subclause 9.18, "Invoking an SQL-invoked routine".

6) [13] Let *PN* and *N* be the number of values in the static SQL argument list of *P*.

7) [13] *P* has a list of *EN* parameters PD_i whose host language data types are determined as follows:

a) Depending on whether the language of *P* specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences table* be Table 21, "Data type correspondences for Ada", Table 22, "Data type correspondences for C", Table 23, "Data type correspondences for COBOL", Table 24, "Data type correspondences for Fortran", Table 25, "Data type correspondences for M", Table 26, "Data type correspondences for Pascal", or Table 27, "Data type

correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

b) For *i* varying from 1 (one) to *EN*, let PT_i be the <data type> of ESP_i, and let the host language data type DT_i of PD_i be the data type listed in the host data type column of the row in the operative data type correspondences table whose value in the SQL data type column is PT_i.

8) Let *E* be 0 (zero).

9) If the call type data item has a value of −1 (negative one, indicating *open call*), then:

NOTE 418 — The call type data item is defined and initialized in Subclause 9.18, "Invoking an SQL-invoked routine".

a) For *i* varying from 1 (one) through *EN*, the values of the parameters PD_i are set as follows. General Rules of Subclause 9.4, "Passing a value from the SQL-server to a host language", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and ESP_i as *SQL VALUE*; let PD_i be the *HOST VALUE* returned from the application of those General Rules.

b) *P* is executed.

c) For *i* varying from 1 (one) through *EN*, General Rules of Subclause 9.3, "Passing a value from a host language to the SQL-server", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and the value of PD_i as *HOST VALUE*; let ESP_i be the *SQL VALUE* returned from the application of those General Rules.

10) Case:

a) If the value of the exception data item is '00000' (corresponding to the completion condition *successful completion (00000)*), then set the call type data item to 0 (zero) (indicating *fetch call*).

NOTE 419 — The exception data item is defined in Subclause 9.18, "Invoking an SQL-invoked routine".

b) If the first two characters of the value of the exception data item are '01' (corresponding to the completion condition *warning (01000)* with any subcondition), then set the call type data item to 0 (zero) (indicating fetch call).

c) If the exception data item is '02000' (corresponding to the completion condition *no data (02000)*):

i) If each PD_i, for *i* ranging from (*PN*+*FRN*)+*N*+1 through (*PN*+*FRN*)+*N*+*FRN* (that is, the SQL indicator arguments corresponding to the result data items), is negative, then set *AR* to the null value.

ii) Set the call type data item to 1 (one) (indicating *close call*).

d) Otherwise, set the call type data item to 1 (one) (indicating *close call*).

11) The following steps are applied as long as the call type data item has a value 0 (zero) (corresponding to *fetch call*):

a) The values in the list of *EN* parameters PD_i are set as follows:

i) For *i* ranging from 1 (one) to *EN*−2, the General Rules of Subclause 9.4, "Passing a value from the SQL-server to a host language", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and ESP_i as *SQL VALUE*; let PD_i be the *HOST VALUE* returned from the application of those General Rules.

ii) For the save area data item, for *i* equal to *EN*−1, the value of PD_i is set to the value returned in PD_i by the prior execution of *P*.

iii) For the call type data item, for *i* equal to *EN*, the value of PD_i is set to 0 (zero).

b) *P* is executed.

c) For *i* varying from 1 (one) through *EN*, the General Rules of Subclause 9.3, "Passing a value from a host language to the SQL-server", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and the value of PD_i as *HOST VALUE*; let ESP_i be the *SQL VALUE* returned from the application of those General Rules.

d) Case:

i) If the exception data item is '00000' (corresponding to completion condition *successful completion (00000)*) or the first two characters of the exception data item are '01' (corresponding to completion condition *warning (01000)* with any subcondition), then:

1) Increment *E* by 1 (one).

2) If *E* > *ARC*, then an exception condition is raised: *data exception — array element error (2202E)*.

3) Case:

A) If each PD_i, for *i* ranging from (*PN*+*FRN*)+*N*+1 through (*PN*+*FRN*)+*N*+*FRN* (that is, the SQL indicator arguments corresponding to the result data items) is negative, then let the *E*-th element of *AR* be the null value.

B) Otherwise,

Case:

I) If *FRN* is 1 (one), then let the *E*-th element of *AR* be the value of the result data item.

II) Otherwise:

1) Let RDI_i, 1 (one) ≤ *i* ≤ *FRN*, be the value of the *i*-th result data item.

2) Let the *E*-th element of *AR* be the value of the following <row value expression>:

```
ROW ( RDI1, ..., RDIFRN )
```

ii) If the exception data item is '02000' (corresponding to completion condition *no data (02000)*), then:

1) If the value of *E* is 0 (zero), then set *AR* to an array whose cardinality is 0 (zero).

2) Set the call type data item to 1 (one) (indicating *close call*).

iii) Otherwise, set the value of the call type data item to 1 (one) (indicating *close call*).

12) If the call type data item has a value of 1 (one) (indicating *close call*), then *P* is executed with a list of *EN* parameters PD_i whose values are set as follows:

a) For *i* ranging from 1 (one) to *EN*−2, the General Rules of Subclause 9.4, "Passing a value from the SQL-server to a host language", are applied with the language of *P* as *LANGUAGE*, PT_i as *SQL TYPE*, and the value of ESP_i as *SQL VALUE*; let PD_i be the *HOST VALUE* returned from the application of those General Rules.

b) For the save area data item, for *i* equal to *EN*−1, the value of PD_i is set to the value returned in PD_i by the prior execution of *P*.

c) For the call type data item, for *i* equal to *EN*, the value of PD_i is set to 1 (one).

d) *P* is executed.

NOTE 420 — Any output arguments from the close call are ignored. The result *AR* of this Subclause has already been computed.

13) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.22 Execution of multiset-returning external functions

Function

Define the execution of an external function that returns a multiset value.

Subclause Signature

```
"Execution of multiset-returning external functions" [General Rules] (
  Parameter: "MULTISET",
  Parameter: "EFFECTIVE SQL PARAMETER LIST",
  Parameter: "PROGRAM"
)
```

MULTISET — a site with a declared type that is a multiset type.

EFFECTIVE SQL PARAMETER LIST — a list of values to be supplied as parameters for execution of the external routine identified by PROGRAM.

PROGRAM — an external routine that returns a multiset value.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *MU* be the *MULTISET*, let *ESPL* be the *EFFECTIVE SQL PARAMETER LIST*, and let *P* be the *PROGRAM* in an application of the General Rules of this Subclause.

2) Let *ET* be the element type of *MU*.

3) Let *C* be the maximum implementation-defined (IL008) cardinality of array type with element type *ET*.

4) Let *AT* be the array type *ET* ARRAY[*C*].

5) Let *AR* be an array whose declared type is *AT*.

6) The General Rules of Subclause 9.21, "Execution of array-returning external functions", are applied with *AR* as *ARRAY*, *ESPL* as *EFFECTIVE SQL PARAMETER LIST*, and *P* as *PROGRAM*.

7) Let *MU* be the result of casting *AR* to the multiset type of *MU* according to the General Rules of Subclause 6.13, "<cast specification>".

8) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.23 Evaluation and transformation of <window function>

Function

Apply syntactic restrictions to <window function>s and transform the <query specification>s in which they occur.

Subclause Signature

```
"Evaluation and transformation of <window function>" [Syntax Rules] (
  Parameter: "WINFUNC",
  Parameter: "QUERY SPEC IN"
) Returns: "TRANSFORM"
```

WINFUNC — a <window function>.

QUERY SPEC IN — a <query specification> that contains WINFUNC.

TRANSFORM — a transformed <query specification> that is the result of invoking this subclause using this signature.

Syntax Rules

1) Let *OF* be the *WINFUNC* and let *QSS* be the *QUERY SPEC IN* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *TRANSFORM*.

2) Let *TE* be the <table expression>, *SQ* be the <set quantifier>, and *SL* be the <select list> immediately contained in *QSS*. If there is no <set quantifier>, then let *SQ* be the zero-length character string.

3) *OF* shall not contain an outer reference or a <query expression>.

4) Let *WNS* be the <window name or specification>. Let *WDX* be a window structure descriptor that describes the window defined by *WNS*.

5) If <ntile function>, <lead or lag function>, <rank function type> or ROW_NUMBER is specified, then:

   a) If <ntile function>, <lead or lag function>, or <rank function type> is specified, then the window ordering clause *WOC* of *WDX* shall be present.

   b) The window framing clause of *WDX* shall not be present.

   c) Case:

      i) If *WNS* is a <window name>, then let *WNS1* be *WNS*.

      ii) Otherwise, let *WNS1* be the <window specification details> contained in *WNS*.

   d) `RANK() OVER` *WNS* is equivalent to:

```
( COUNT(*) OVER
  (WNS1 GROUPS BETWEEN UNBOUNDED PRECEDING
                   AND 1 PRECEDING) + 1 )
```

   e) If DENSE_RANK is specified, then:

      i) Let VE_1, ..., VE_N be an enumeration of the <value expression>s that are <sort key>s simply contained in *WOC*.

ii) DENSE_RANK() OVER *WNS* is equivalent to the <window function>:

```
COUNT (DISTINCT ROW ( VE1, ..., VEN ) )
OVER (WNS1 GROUPS UNBOUNDED PRECEDING)
```

f) ROW_NUMBER() OVER *WNS* is equivalent to the <window function>:

```
COUNT (*) OVER (WNS1 ROWS UNBOUNDED PRECEDING)
```

g) Let *ANT1* be an approximate numeric type with implementation-defined (IV082) precision. PERCENT_RANK( ) OVER *WNS* is equivalent to:

```
CASE
  WHEN COUNT(*) OVER (WNS1 GROUPS BETWEEN UNBOUNDED PRECEDING
                                          AND UNBOUNDED FOLLOWING) = 1
  THEN CAST (0 AS ANT1)
  ELSE
       (CAST (RANK () OVER (WNS1) AS ANT1) - 1) /
       (COUNT (*) OVER (WNS1 GROUPS BETWEEN UNBOUNDED PRECEDING
                              AND UNBOUNDED FOLLOWING) - 1)
END
```

h) Let *ANT2* be an approximate numeric type with implementation-defined (IV083) precision. CUME_DIST( ) OVER *WNS* is equivalent to:

```
 ( CAST ( COUNT (*) OVER
      ( WNS1 GROUPS UNBOUNDED PRECEDING ) AS ANT2 ) /
    COUNT(*) OVER ( WNS1 GROUPS BETWEEN UNBOUNDED PRECEDING
                                    AND UNBOUNDED FOLLOWING ) )
```

i) If <ntile function> is specified, then:

i) The declared type of <number of tiles> shall be exact numeric with scale 0 (zero).

ii) The declared type of the result is an implementation-defined (IV084) exact numeric type with scale 0 (zero).

j) If <lead or lag function> is specified, then:

i) Let *VE1* be <lead or lag extent> and let *DT* be the declared type of *VE1*.

ii) Case:

1) If <offset> is specified, then let *OFF* be <offset>.

2) Otherwise, let *OFF* be 1 (one).

iii) Case:

1) If <default expression> is specified, then let *VE2* be <default expression>. The declared type of *VE2* shall be compatible with *DT*.

2) Otherwise, let *VE2* be CAST (NULL AS *DT*).

iv) If <window function null treatment> is specified, then let *NTREAT* be the <window function null treatment>; otherwise, let *NTREAT* be RESPECT NULLS.

v) The declared type of the result is *DT*.

6) If <first or last value function> or <nth value function> is specified, then:

a) If <nth value function> is specified, then:

i) If <from first or last> is not specified, then FROM FIRST is implicit.

ii) The declared type of <nth row> shall be exact numeric with scale 0 (zero).

b) If <window function null treatment> is not specified, then RESPECT NULLS is implicit.

c) Let *DT* be the declared type of <value expression>. The declared type of the result is *DT*.

7) If <in-line window specification> is specified, then:

a) Let *WS* be the <window specification>.

b) Let *WSN* be an implementation-dependent (UV071) <window name> that is not equivalent to any other <window name> in *TE*.

c) Let *OFT* be the <window function type>.

d) Let *SLNEW* be the <select list> that is obtained from *SL* by replacing *OF* by:

```
OFT OVER WSN
```

e) Let *FC*, *WC*, *GBC*, and *HC* be <from clause>, <where clause>, <group by clause>, and <having clause>, respectively, of *TE*. If any of <where clause>, <group by clause>, or <having clause> is missing, then let *WC*, *GBC*, or *HC*, respectively, be the zero-length character string.

f) Case:

i) If there is no <window clause> immediately contained in *TE*, then let *WICNEW* be:

```
WINDOW WSN AS WS
```

ii) Otherwise, let *WIC* be the <window clause> immediately contained in *TE* and let *WICNEW* be:

```
WIC, WSN AS WS
```

g) Let *TENEW* be:

```
FC WC GBC HC WICNEW
```

h) *QSX* is equivalent to:

```
SELECT SQ SLNEW TENEW
```

8) If <window function type> is <aggregate function>, then:

a) Let *AF* be the <aggregate function>.

b) The declared type of <window function> is the declared type of *AF*.

c) *AF* shall not simply contain a <hypothetical set function>.

d) If the window ordering clause or the window framing clause of the window structure descriptor that describes the <window name or specification> is present, then:

i) *AF* shall not be an <ordered set function> or a <listagg set function>.

ii) If *AF* is not a <general set function> that simply contains the <computational operation> COUNT, then *AF* shall not simply contain DISTINCT.

9) A <measure name> that is a <window row pattern measure> shall be equivalent to the <measure name> contained in a <row pattern measure definition> *RPMC* in the <row pattern measures> of *WDX*. The declared type of the <window function> is the declared type of the <value expression> contained in *RPMC*.

10) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *QSX* as *TRANSFORM*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.24 Compilation of an invocation of a polymorphic table function

Function

Define the compilation of function invocations of polymorphic table functions.

Subclause Signature

```
"Compilation of an invocation of a polymorphic table function" [General Rules] (
  Parameter: "ROUTINE INVOCATION",
  Parameter: "POLYMORPHIC TABLE FUNCTION",
  Parameter: "ARGUMENT LIST"
) Returns: "PTF DATA AREA"
```

ROUTINE INVOCATION — a <routine invocation> that causes the invocation of a polymorphic table function.

POLYMORPHIC TABLE FUNCTION — a polymorphic table function.

ARGUMENT LIST — an <SQL argument list>.

PTF DATA AREA — a PTF data area that is populated by and returned from an invocation of this subclause using this signature.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *RI* be the *ROUTINE INVOCATION*, let *SR* be the *POLYMORPHIC TABLE FUNCTION*, and let *CAL* be the *ARGUMENT LIST* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *PTF DATA AREA*.

2) The PTF data area *PTFDA* is initialized as follows:

   a) For each private parameter *PP* of *SR*, *PTFDA* includes an SQL variable *PV* whose declared type is the declared type of *PP*, with a distinct implementation-dependent (UV131) <SQL variable name>. *PV* is the *private variable* of *PTFDA* corresponding to *PP*. The initial value of *PV* is

      Case:

      i) If *PP* has a <parameter default> *PD*, then the value of *PD*.

         NOTE 421 — As stated in a Syntax Rule of Subclause 11.60, "<SQL-invoked routine>", the <parameter default> must be deterministic and evaluable without accessing SQL-data.

      ii) Otherwise, the null value.

   b) For each generic table parameter *GTA* of *SR*:

      i) Let *NC* be the number of columns of *GTA*.

ii) The PTF descriptor *FRTDA* of the full row type of *GTA*, with a distinct implementation-dependent (UV131) PTF extended name *FRTDN*, is initialized as follows:

1) The maximum number of SQL item descriptor areas is *NC*.

2) Let *GTATN* be an effective name for the table identified by *GTA*. Let *SN* be a <statement name> distinct from any other <statement name> in the SQL-session.

3) The following <prepare statement> is executed:

```
PREPARE SN FROM 'SELECT * FROM GTATN'
```

4) The following <describe statement> is executed:

```
DESCRIBE SN USING DESCRIPTOR PTF FRTDN
```

iii) If *GTA* has set semantics, then:

1) The PTF descriptor area *PDA* of the partitioning of *GTA*, with a distinct implementation-dependent (UV131) PTF extended name, initialized as follows:

A) Let *NPC* be the number of partitioning columns of *GTA*.

B) The COUNT and TOP_LEVEL_COUNT components of the header of *PDA* are set to *NPC*.

C) The number and maximum number of SQL item descriptors of *PDA* is *NPC*.

D) For *i* between 1 (one) and *NPC*, the NAME component of the *i*-th SQL item descriptor area is the <column name> of the *i*-th partitioning column of *GTA*, and the LEVEL component is 0 (zero).

E) All other components of the header and SQL item descriptors of *PDA* are undefined.

2) The PTF descriptor area *ODA* of the ordering of *GTA*, with a distinct implementation-dependent (UV131) PTF extended name, is initialized as follows:

A) Let *NOC* be the number of ordering columns of *GTA*.

B) The COUNT and TOP_LEVEL_COUNT components of the header of *ODA* are set to *NOC*.

C) The number and maximum number of SQL item descriptors of *ODA* are set to *NOC*.

D) For *i* ranging from 1 (one) to *NOC*:

I) The LEVEL component of the *i*-th SQL item descriptor area is 0 (zero).

II) The NAME component of the *i*-th SQL item descriptor area is the <column name> of the *i*-th ordering column of *GTA*.

III) The SORT_DIRECTION component of the *i*-th SQL item descriptor area is

Case:

1) If the explicit or implicit <ordering specification> contained in the *i*-th <table argument ordering column> is ASC, then 1 (one).

2) Otherwise, −1 (negative one).

IV) The NULL_ORDERING component of the *i*-th SQL item descriptor area is

Case:

1) If the explicit or implicit <null ordering> contained in the *i*-th <table argument ordering column> is NULLS FIRST, then 1 (one).

2) Otherwise, −1 (negative one).

E) All other components of the header and SQL item descriptors of *ODA* are undefined.

iv) The PTF descriptor area for the requested row type of *GTA*, with a distinct implementation-dependent (UV131) PTF extended name, is initialized as follows:

1) The COUNT and TOP_LEVEL_COUNT components in the header are 0 (zero). All other components of the header are undefined.

2) There are no SQL item descriptor areas.

3) The maximum number of SQL item descriptor areas is *NC*.

v) The PTF descriptor area for the cursor row type of *GTA*, with a distinct implementation-dependent (UV131) PTF extended name, is initialized as follows:

1) The COUNT and TOP_LEVEL_COUNT components in the header are 0 (zero). All other components of the header are undefined.

2) There are no SQL item descriptor areas.

3) The maximum number of SQL item descriptor areas is

Case:

A) If *GTA* specifies PASS THROUGH, then *NC* + 1 (one).

B) Otherwise, *NC*.

vi) The PTF cursor cursor for *GTA* is not specified.

NOTE 422 — The PTF cursor is not used during the compilation phase.

c) For each <descriptor value constructor> *DVC* that is an <SQL argument> in *CAL*, a PTF descriptor area specified by *DVC*, with a distinct implementation-dependent (UV131) PTF extended name.

d) The PTF descriptor area *IRR* for the initial result row type, with a distinct implementation-dependent (UV131) PTF extended name, initialized as follows.

Case:

i) If the <returns data type> of *SR* contains a <table function column list> *TFCL*, then:

1) Let *NTFCL* be the number of <table function column list element>s contained in *TFCL*. The TOP_LEVEL_COUNT component in the header of *IRR* is set to *NTFCL*.

2) SQL item descriptor areas are appended to *IRR* as follows. For all *c*, 1 (one) ≤ *c* ≤ *NTFCL*:

A) Let *CD* be the *c*-th <table function column list element>.

B) Sufficient SQL item descriptors areas are appended to *IRR* to represent the <data type> contained in *CD*, as defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

C) The NAME component of the first of these appended SQL item descriptor areas is the <column name> contained in *CD*.

3) The COUNT component of *TFCL* is the number *NIRR* of SQL item descriptor areas in *IRR*.

4) The maximum number of SQL item descriptor areas of *IRR* is *NIRR*.

ii) Otherwise:

1) The COUNT and TOP_LEVEL_COUNT components in the header are 0 (zero). All other components of the header are undefined.

2) There are no SQL item descriptor areas.

3) The maximum number of SQL descriptor items is implementation-defined (IL056).

e) The PTF descriptor area for the intermediate result row type, with a distinct implementation-dependent (UV131) PTF extended name, initialized as follows:

i) The COUNT and TOP_LEVEL_COUNT components in the header are 0 (zero). All other components of the header are undefined.

ii) There are no SQL item descriptor areas.

iii) The maximum number of SQL descriptor items is implementation-defined (IL057).

f) An SQL variable of declared type CHARACTER(5) with an implementation-dependent (UV131) <SQL variable name>. This is the *status variable* of *PTFDA*. The initial value of the status variable is *successful completion (00000)*.

3) The scope of every variable name and every PTF extended name associated with *PTFDA* is the General Rules of Subclause 9.27, "Invocation of a PTF component procedure", when executing a PTF component procedure of *SR*.

4) Case:

a) If *SR* has a PTF describe component procedure, then the General Rules of Subclause 9.27, "Invocation of a PTF component procedure", are applied with *SR* as *POLYMORPHIC TABLE FUNCTION*, "describe" as *COMPONENT*, *CAL* as *ARGUMENT LIST*, and *PTFDA* as *PTF DATA AREA*.

NOTE 423 — The PTF describe component procedure is expected to populate the requested row type descriptor areas. If there is no <table function column list> and RETURNS ONLY PASS THROUGH is not specified, then the PTF describe component procedure must also populate the initial result row type descriptor area. The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", will check this.

b) Otherwise, for every generic table parameter *GTA* of *SR*, the header and all PTF item descriptors of the initial row type descriptor area of *GTA* is copied to the header and PTF item descriptors of the requested row type descriptor area of *GTA*.

NOTE 424 — When the PTF describe component procedure is absent, the initial result row type descriptor area remains unchanged.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *PTFDA* as *PTF DATA AREA*.

Conformance Rules

None.

9.25 Execution of an invocation of a polymorphic table function

Function

Execute an invocation of a polymorphic table function.

Subclause Signature

```
"Execution of an invocation of a polymorphic table function" [General Rules] (
  Parameter: "ROUTINE INVOCATION",
  Parameter: "POLYMORPHIC TABLE FUNCTION",
  Parameter: "ARGUMENT LIST",
  Parameter: "PTF DATA AREA",
  Parameter: "RESULT TYPE"
) Returns: "RESULT TABLE"
```

ROUTINE INVOCATION — a <routine invocation> that causes the invocation of a polymorphic table function.

POLYMORPHIC TABLE FUNCTION — a polymorphic table function.

ARGUMENT LIST — an <SQL argument list>.

PTF DATA AREA — a PTF data area.

RESULT TYPE — the desired row type of the result table.

RESULT TABLE — the result table created and populated by an invocation of this subclause using this signature.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *RI* be the *ROUTINE INVOCATION*, let *SR* be the *POLYMORPHIC TABLE FUNCTION*, let *PAL* be the *ARGUMENT LIST*, let *PTFDA* be the *PTF DATA AREA*, and let *RESTYPE* be the *RESULT TYPE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *RESULT TABLE*.

2) Let *RESTAB* be an empty table of type *RESTYPE*.

3) Let *NT* be the number of <table argument>s contained in *RI*. Let TA_1, ..., TA_{NT} be the <table argument>s contained in *RI*, enumerated in the order of the corresponding <generic table parameter type>s in *SR*. Let TP_1, ..., TP_{NT} be the corresponding <SQL parameter declaration>s of *SR*.

4) For all *t*, 1 (one) ≤ *t* ≤ *NT*,

   Case:

a) If TA_t has row semantics and the table specified by TA_t is empty, then *RESTAB* is returned as the *RESULT TABLE* and no further General Rules of this Subclause are applied.

b) If TA_t has set semantics, TA_t is pruned, and the table specified by TA_t is empty, then *RESTAB* is returned as the *RESULT TABLE* and no further General Rules of this Subclause are applied.

5) For all t, 1 (one) $\leq t \leq NT$,

a) Case:

i) If TA_t simply contains a <table argument correlation name> $TACN_t$, then let $ASTACN_t$ be

AS $TACN_t$

ii) Otherwise, let $ASTACN_t$ be the zero-length character string.

b) Case:

i) If TA_t simply contains <table argument parenthesized derived column list>, then let $TAPDCL_t$ be that <table argument parenthesized derived column list>.

ii) Otherwise, let $TAPDCL_t$ be the zero-length character string.

c) Let TAP_t be the <table argument proper> simply contained in TA_t.

Case:

i) If TAP_t simply contains <table or query name> $TOQN_t$, then let TR_t be

$TOQN_t$ $ASTACN_t$ $TAPDCL_t$

ii) If TAP_t simply contains <table subquery> TSQ_t, then let TR_t be

TSQ_t $ASTACN_t$ $TAPDCL_t$

iii) If TAP_t is a <table function invocation> TFI_t, then let TR_t be

TABLE (TFI_t) $ASTACN_t$ $TAPDCL_t$

6) Let *NS* be the number of <table argument>s with set semantics. Let *NP* be the number of <table argument>s with one or more partitioning columns. Let *NC* be the number of <table argument>s whose range variable is referenced in a <co-partition specification> contained in *RI*, if any.

NOTE 425 — 0 (zero) $\leq NC \leq NP \leq NS \leq NT$.

7) Let PTA_1, ..., PTA_{NT} be a permutation of the list of <table argument>s, such that:

a) If *RI* contains a <co-partition clause> *COCL*, then:

i) The <table argument>s that are referenced by <range variable>s in *COCL* are the first *NC* <table argument>s in the permuted list.

ii) All <table argument>s that are referenced in a single <co-partition specification> are adjacent in the permuted list.

b) All <table argument>s that have one or more partitioning columns are the first *NP* <table argument>s in the permuted list.

c) All <table argument>s that have set semantics are the first *NS* <table argument>s in the permuted list.

d) If there is an input table with row semantics, then it is the last one in the permuted list (PTA_{NT}).

8) Let *per* be the permutation such that, for all *t*, 1 (one) ≤ *t* ≤ *NT*, $TA_t = PTA_{per(t)}$.

9) Let RV_1, ..., RV_{NT} be distinct <correlation name>s.

10) If any generic table parameter of *SR* specifies PASS THROUGH, then let $OPQN_1$, ..., $OPQN_{NT}$ be mutually distinct implementation-dependent (UV133) <column name>s such that, for all *t*, 1 (one) ≤ *t* ≤ *NT*, $OPQN_t$ is not equivalent to the NAME component of any SQL item descriptor whose LEVEL component is 0 (zero) in the PTF descriptor area for the full row type of PTA_t in *PTFDA*.

NOTE 426 — These will be used as the column names of the pass-through input surrogate columns (in the cursors) and the pass-through output surrogate columns (in the intermediate result row type descriptor.)

11) For all *t*, 1 (one) ≤ *t* ≤ *NT*:

a) Let *REQ* be the requested row descriptor of PTA_t in *PTFDA*. Let *NREQ* be the number of SQL item descriptors in *REQ* whose LEVEL component is 0 (zero). For all *i*, 1 (one) ≤ *i* ≤ *NREQ*, let $COLN_i$ be an <identifier> whose value is the value of the NAME component of the *i*-th SQL item descriptor of *REQ* whose LEVEL component is 0 (zero). Let *SELL* be

$COLN_1$, ..., $COLN_{NREQ}$

NOTE 427 — The column names in the requested row descriptor are required to reference unique columns of the input table by a Syntax Rule in Subclause 9.18, "Invoking an SQL-invoked routine"; hence, there is no need to qualify the column names in the SELECT list with range variables. If the query needs access to columns with duplicate column names, this can be handled by renaming columns in the query.

b) Let T_t be the table specified by PTA_t.

c) Case:

i) If PTA_t specifies PASS THROUGH, then let XT_t be T_t extended with one additional column *OPQ* of implementation-dependent (UV133) declared type and with <column name> $OPQN_t$, such that, for every row *R* of XT_t, the value of *OPQ* in *R* represents the value of all columns of T_t in the corresponding row of T_t. *OPQ* is *the pass-through input surrogate column* of XT_t. Let $SELLIST_t$ be

$SELL$, $OPQN_t$

NOTE 428 — For example, for a given row *R* of T_t, *OPQ* can be the concatenation of all columns of *R* in a single BLOB, or some compression thereof, or the (compressed) concatenation of a candidate key. Using conventional column projection techniques, *OPQ* only needs to represent the columns of T_t actually required to execute the encompassing query.

ii) Otherwise, let XT_t be T_t and let $SELLIST_t$ be *SELL*.

d) Let TN_t be an effective name for XT_t.

12) The intermediate result row type descriptor *MRR* in *PTFDA* is populated as follows:

a) The initial result row type descriptor of *PTFDA* is copied to *MRR*.

b) For each generic table parameter *GTA* of *PTF* that specifies PASS THROUGH:

i) Let *PISC* be the pass-through input surrogate column for *GTA*.

ii) An SQL item descriptor area *SIDA* is appended to *MRR*, matching the declared type of *PISC* and having a NAME component equal to the value of the <column name> of *PISC*. The column described by *SIDA* is the *pass-through output surrogate column* of *GTA*.

iii) COUNT and TOP_LEVEL_COUNT in the header of *MRR* are incremented by 1 (one).

13) If *RI* contains a <co-partition clause> *COCL*, then:

a) Let *NCOSPEC* be the number of <co-partition specification>s contained in *COCL*. Let $COSPEC_1$, ..., $COSPEC_{NCOSPEC}$ be the <co-partition specification>s contained in *COCL*.

b) For all *s*, 1 (one) ≤ *s* ≤ *NCOSPEC*:

i) Let *NCOT* be the number of <range variable>s contained in $COSPEC_s$, let $CORV_1$, ..., $CORV_{NCOT}$ be those <range variable>s, let $COTA_1$, ..., $COTA_{NCOT}$ be the <table argument>s that expose those <range variable>s, and let COT_1, ..., COT_{NCOT} be the tables specified by those <table argument>s.

NOTE 429 — A <table argument> that is a nested polymorphic table function invocation can expose more than one range variable. In that case, the query can use any range variable exposed by the <table argument> to represent the <table argument> in the <co-partition specification>.

ii) Let *NCOP* be the number of partitioning columns in each of the tables COT_1, ..., COT_{NCOT}.

iii) For all *t*, 1 (one) ≤ *t* ≤ *NCOT*, let $PC_{t,1}$, ..., $PC_{t,NCOP}$ be the <column reference>s of the partitioning columns of COT_t.

iv) Let PCN_1, ..., PCN_{NCOP} be distinct <column name>s.

v) Let *X* be a <column name> that is distinct from PCN_c for all *c*, 1 (one) ≤ *c* ≤ *NCOP*.

vi) Case:

1) If, for all *t2*, 1 (one) ≤ *t2* ≤ *NCOT*, COT_{t2} is an empty table, then:

A) For all *c*, 1 (one) ≤ *c* ≤ *NCOP*, let $DTCP_c$ be the declared type of $PC_{t,c}$. Let SD_t be

```
( SELECT CAST (NULL AS DTCP_t,1) AS PCN_1,
         ...
         CAST (NULL AS DTCP_t,NCOP) AS PCN_NCOP
         1 AS X
  FROM VALUES (1) ) AS RV_t
```

B) Let FOJ_s be

```
SD_1, ..., SD_NCOT
```

NOTE 430 — Placed in a <from clause>, this is a table with one row, denoting that only one virtual processor is required to handle the empty tables COT_1, ..., COT_{NCOT}.

2) Otherwise:

A) For all *t*, 1 (one) ≤ *t* ≤ *NCOT*, let SD_t be the character string

```
( SELECT DISTINCT PC_t,1 AS PCN_1,
                  ...,
                  PC_t,NCOP AS PCN_NCOP,
                  1 AS X
  FROM TR_t ) AS RV_t
```

B) For all pairs (*t1*, *t2*) where 1 (one) ≤ *t1* < *t2* ≤ *NCOT*, let $JC_{t1,t2}$ be

```
    RVt1.PCN1 IS NOT DISTINCT FROM RVt2.PCN1
AND RVt1.PCN2 IS NOT DISTINCT FROM RVt2.PCN2
AND ...
AND RVt1.PCNNCOP IS NOT DISTINCT FROM RVt2.PCNNCOP
```

NOTE 431 — This expresses the join condition that XT_{t1} and XT_{t2} have the same values in all pairs of corresponding partitioning columns, treating a null value as the same as another null value.

C) Let FOJ_s be the <joined table>

```
( ( ... ( SD1 FULL OUTER JOIN SD2 ON JC1,2 )
                FULL OUTER JOIN SD3 ON JC1,3 AND JC2,3 )
                ...
                FULL OUTER JOIN SDNCOT ON JC1,NCOT
                                      AND JC2,NCOT
                                      AND ...
                                      AND JCNCOT−1,NCOT )
```

NOTE 432 — This is a full outer join on all tables referenced in the <co-partition specification>, performing an equijoin on all corresponding pairs of partitioning columns. Here equijoin is in the sense of regarding a null value as not distinct from another null value. Each row of this table requires a separate virtual processor.

vii) For all *t*, 1 (one) ≤ *t* ≤ *NCOT*,

Case:

1) If PTA_t is pruned, then let $JUICE_t$ be the character string

```
AND RVt.X IS NOT NULL
```

NOTE 433 — This condition will be used to exclude null-extended rows from a full outer join, thereby pruning partitions that would otherwise be created for an empty cursor.

2) Otherwise, let $JUICE_t$ be the empty character string.

viii) Let $JUICY_s$ be the concatenation:

```
JUICE1 ... JUICENCOT
```

14) For all *t*, *NC* < *t* ≤ *NP*,

NOTE 434 — There is such *t* if there are partitioned table(s) that are not co-partitioned.

a) Let *NPC* be the number of partitioning columns of T_t. Let $PC_{t,1}$, ..., $PC_{t,NPC}$ be the <column reference>s of the partitioning columns of T_t.

b) Let $PCN_{t,1}$, ..., $PCN_{t,NPC}$ be distinct <column name>s.

c) Case:

i) If PTA_t is an empty table, then for all *c*, 1 (one) ≤ *c* ≤ *NPC*, let $DTPC_{t,c}$ be the declared type of $PC_{t,c}$ and let SD_t be the character string

```
( SELECT CAST (NULL AS DTCPt,1) AS PCNt,1,
         ...,
         CAST (NULL AS DTCPt,NPC) AS PCNt,NPC
  FROM VALUES (1) ) AS RVt
```

ii) Otherwise, let SD_t be the character string

```
( SELECT DISTINCT PC_t,1 AS PCN_t,1,
                  ...,
                  PC_t,NPC AS PCN_t,NPC
  FROM TR_t ) AS RV_t
```

15) For all t, 1(one) ≤ t ≤ NT,

Case:

a) If PTA_t has a <table argument ordering list> $TAOL_t$, then:

i) Let SSL_t be the character string obtained from $TAOL_t$ by removing the initial <left paren> and the final <right paren>, if any.

ii) Let OB_t be

```
ORDER BY SSL_t
```

b) Otherwise, let OB_t be the zero-length character string.

16) Case:

a) If NP = 0 (zero), then let P be 1 (one).

NOTE 435 — P is the minimum number of virtual processors required. Since there are no partitioned tables (because NP = 0), only one virtual processor is required. However, if there is a table with row semantics, there can be more than one virtual processor, as specified in later General Rules.

b) Otherwise:

i) Case:

1) If NC = NP, then let *CROSS* be

```
SELECT RV_1.*, ..., RV_P.*
FROM FOJ_1, ..., FOJ_NCOSPEC
WHERE 1=1 JUICY_1 ... JUICY_NCOSPEC
```

NOTE 436 — This is the case in which every partitioned table is co-partitioned.

2) If 0 (zero) = NC < NP, then let *CROSS* be

```
SELECT RV_1.*, ..., RV_NP.*
FROM SD_1, ..., SD_NP
```

NOTE 437 — This is the case in which there are partitioned tables and none of them is co-partitioned.

3) Otherwise, let *CROSS* be

```
SELECT RV_1.*, ..., RV_NP.*
FROM FOJ_1, ..., FOJ_NCO,
     SD_NC+1, ..., SD_NP
WHERE 1=1 JUICY_1 ... JUICY_NCOSPEC
```

NOTE 438 — This is the case in which there are partitioned tables, some of them are co-partitioned and others are not co-partitioned.

ii) Let *PARTITIONS* be the result of evaluating *CROSS*. Let P be the number of rows in *PARTITIONS*.

NOTE 439 — P is the minimum number of virtual processors required to process the partitions of the tables. If there is no table with row semantics, P is the exact number of virtual processors. If there is a table with row semantics, the actual number of virtual processors can grow.

iii) Let R_1, ..., R_P be an enumeration of the rows of *PARTITIONS*.

For all p, 1 (one) ≤ p ≤ P:

1) Let *NMC* be the number of columns in *PARTITIONS*. Let $MCOLN_1$, ..., $MCOLN_{NMC}$ be distinct implementation-dependent (UV133) <column name>s. Let *PRINCIPLEROW* be a table whose columns names are $MCOLN_1$, ..., $MCOLN_{NMC}$ and whose only row is R_p. Let *TMR* be an effective table name for *PRINCIPLEROW*.

2) For all t, 1 (one) ≤ t ≤ *NC*:

NOTE 440 — That is, t is the index of a co-partitioned table.

A) Let *NPC* be the number of partitioning columns of T_t, let $PC_{t,1}$, ..., $PC_{t,NPC}$ be the <column reference>s of the partitioning columns of T_t, and let the column names in *PRINCIPLEROW* corresponding to the partitioning columns of T_t be $MCOLN_x$, ..., $MCOLN_{x+NPC-1}$.

B) Case:

I) If the column corresponding to $RV_t.X$ in the row R_p is null, then let $PREP_{p,t}$ be

```
0 = 1
```

NOTE 441 — R_p is caused by a null extension in the full outer join involving T_t, and signifies that there are no rows in T_t that match on the partitioning columns for any other table with which T_t is co-partitioned.

II) Otherwise, let $PRED_{p,t}$ be

```
PCt,1 IS NOT DISTINCT FROM
( SELECT MCOLNx
  FROM TMR )
AND
...
AND
PCt,NPC IS NOT DISTINCT FROM
( SELECT MCOLNx+NPC-1
  FROM TMR )
```

NOTE 442 — $RV_t.X$ is not null, signifying a row of *PARTITIONS* that is not null-extended in the columns for table T_t. The predicate $PRED_{p,t}$ selects the rows of T_t that match the partitioning columns in *PRINCIPLEROW*.

C) Let $CS_{p,t}$ be

```
SELECT SELLISTt
FROM TRt
WHERE PREDp, t
OBt
FOR READ ONLY
```

NOTE 443 — This is a <cursor specification> that will be used to define the cursor for PTA_t on a virtual processor corresponding to *PRINCIPLEROW*.

D) The *common values of the partitioning columns* of $CS_{p,t}$ are the values of $MCOLN_x$, ..., $MCOLN_{x+NPC-1}$.

3) For all t, *NC* < t ≤ *NP*

NOTE 444 — t is the index of a partitioned table that is not co-partitioned.

A) Let *NPC* be the number of partitioning columns of T_t, let $PC_{t,1}$, ..., $PC_{1,NPC}$ be the <column reference>s of the partitioning columns of T_t, and let the column names in *PRINCIPLEROW* corresponding to the partitioning columns of T_t be $MCOLN_x$, ..., $MCOLN_{x+NPC-1}$. Let $PRED_{p,t}$ be

```
    PC_t,1 IS NOT DISTINCT FROM
    ( SELECT MCOLN_x
      FROM TMR )
AND
    ...
AND
    PC_t,NPC IS NOT DISTINCT FROM
    ( SELECT MCOLN_x+NPC-1
      FROM TMR )
```

B) Let $CS_{p,t}$ be

```
SELECT SELLIST_t
FROM TR_t
WHERE PRED_t,m
OB_t
FOR READ ONLY
```

C) The *common values of the partitioning columns* $CS_{p,t}$ are the values of $MCOLN_x$, ..., $MCOLN_{x+NPC-1}$.

17) For all t, $NP < t \le NS$, and for all p, 1 (one) $\le p \le P$, let $CS_{p,t}$ be

```
SELECT SELLIST_t
FROM TR_t
OB_t
FOR READ ONLY
```

NOTE 445 — These are the tables with set semantics that are not partitioned. There is no <where clause>, so the cursor reads the entire table on each virtual processor (the table is "broadcast" into the virtual processors).

18) Case:

a) If there is an input table with row semantics, then:

i) For all m, 1 (one) $\le m \le P$,

1) Let $\{RSP_1, ..., RSP_{idp(m)}\}$ be an implementation-dependent (UA056) partition of XT_{NT}.

NOTE 446 — There can be at most one input table with row semantics, and it must be the last input table in the permuted list of input tables. XT_{NT} is not empty; otherwise, this Subclause would have already returned an empty table.

2) For all i, 1 (one) $\le i \le idp(m)$:

A) Let $QE_{m,i}$ be an implementation-dependent (UV129) <query expression> that selects the rows in $RSP_{m,i}$.

B) Let $RSCS_{m,i}$ be the <cursor specification>

```
QE_m,i
FOR READ ONLY
```

ii) Let *NVP* be the sum $idp(1) + idp(2) + ... + idp(P)$.

iii) Let *CURSORS* be the following set of *NVP* lists of <cursor specification>s, with *NT* <cursor specification>s in each list:

$$
\begin{aligned}
&\{ (CS_{1,1}, CS_{1,2}, \ldots, CS_{1,NT-1}, RSCS_{1,1}),\\
&\quad (CS_{1,1}, CS_{1,2}, \ldots, CS_{1,NT-1}, RSCS_{1,2}),\\
&\quad \ldots\\
&\quad (CS_{1,1}, CS_{1,2}, \ldots, CS_{1,NT-1}, RSCS_{1,idp(1)}),\\
&\quad (CS_{2,1}, CS_{2,2}, \ldots, CS_{2,NT-1}, RSCS_{2,1}),\\
&\quad (CS_{2,1}, CS_{2,2}, \ldots, CS_{2,NT-1}, RSCS_{2,2}),\\
&\quad \ldots\\
&\quad (CS_{2,1}, CS_{2,2}, \ldots, CS_{2,NT-1}, RSCS_{2,idp(2)}),\\
&\quad \ldots\\
&\quad (CS_{P,1}, CS_{P,2}, \ldots, CS_{P,NT-1}, RSCS_{P,1}),\\
&\quad (CS_{P,1}, CS_{P,2}, \ldots, CS_{P,NT-1}, RSCS_{P,2}),\\
&\quad \ldots\\
&\quad (CS_{P,1}, CS_{P,2}, \ldots, CS_{P,NT-1}, RSCS_{P,idp(P}) \}
\end{aligned}
$$

b) Otherwise, let *NVP* be *P* and let *CURSORS* be the following set of *NVP* (= *P*) lists of <cursor specification>s, with *NT* <cursor specification>s in each list:

$$
\begin{aligned}
&\{ (CS_{1,1}, CS_{1,2}, \ldots, CS_{1,NT}),\\
&\quad (CS_{2,1}, CS_{2,2}, \ldots, CS_{2,NT}),\\
&\quad \ldots\\
&\quad (CS_{P,1}, CS_{P,2}, \ldots, CS_{P,NT}) \}
\end{aligned}
$$

NOTE 447 — *CURSORS* contains every list of <cursor specification>s to be opened on different virtual processors. If there are no table arguments (*NT* = 0), the degenerate case is a single list with no <cursor specification>.

19) Let $LIST_1$, ... $LIST_{NVP}$ be an enumeration of the lists in *CURSORS*. For all *v*, 1 (one) ≤ *v* ≤ *NVP*, let $LIST_v$ = ($LCS_{v,1}$, ..., $LCS_{v,NT}$).

20) For every *v*, 1 (one) ≤ *v* ≤ *NVP*, let VP_v be a virtual processor. Each virtual processor has an SQL execution context that is a copy of the current SQL execution context. Each virtual processor executes independently of every other virtual processor.

For every *v*, 1 (one) ≤ *v* ≤ *NVP*, the following rules are executed on VP_v:

a) Let $REST_v$ be an empty table of row type *RESTYPE*. $REST_v$ is the *partial result table* of VP_v.

b) Let DA_v be a PTF data area that is a copy of *PTFDA*.

c) For all *t*, 1 (one) ≤ *t* ≤ *NT*,

i) The PTF cursor for the *t*-th <table argument> TA_t is initialized as follows:

1) The cursor declaration descriptor is initialized as follows:

A) The kind of cursor is PTF dynamic cursor.

B) The provenance of the cursor is *RI*.

C) The name of the is an implementation-dependent (UV130) PTF extended cursor name.

D) The cursor's origin is $LCS_{v,per(t)}$.

E) The cursor's declared properties are ASENSITIVE, NO SCROLL, WITHOUT HOLD, and WITHOUT RETURN.

2) The cursor instance descriptor is initialized as follows:

A) The cursor declaration descriptor is as specified above.

B) The SQL-session identifier is the current SQL-session identifier.

C) The cursor's state is closed.

ii) If TA_t has at least one partitioning column, then the *common values of the partitioning columns* of TA_t are the common values of the partitioning columns of $LCS_{v,per(t)}$.

iii) The PTF descriptor area for the cursor row type CRT_t of TA_t is initialized as follows:

1) Let *SN* be a <statement name> that is distinct from any other <statement name> in the SQL-session.

2) The following <prepare statement> is executed:

```
PREPARE SN FROM LCS_v,per(t)
```

3) Let *CRTDN* be the implementation-dependent (UV131) PTF extended name of CRT_t.

4) The following <describe statement> is executed:

```
DESCRIBE SN USING DESCRIPTOR PTF CRTDN
```

d) If *PTF* has a PTF start component procedure, then:

i) The General Rules of Subclause 9.27, "Invocation of a PTF component procedure", are applied with *SR* as *POLYMORPHIC TABLE FUNCTION*, "start" as *COMPONENT*, *PAL* as *ARGUMENT LIST*, and DA_v as *PTF DATA AREA*.

ii) Any rows passed by a <pipe row statement> during the execution of those General Rules are inserted in $REST_v$.

iii) If the status variable in DA_v is not *successful completion (00000)*, then the value of the status variable is raised as an exception condition.

e) For every t, 1 (one) $\leq t \leq NT$,

i) Let CUR_t be the cursor of TA_t in DA_v.

ii) The General Rules of Subclause 15.1, "Effect of opening a cursor", are applied with CUR_t as *CURSOR*.

f) The General Rules of Subclause 9.27, "Invocation of a PTF component procedure", are applied with *SR* as *POLYMORPHIC TABLE FUNCTION*, "fulfill" as *COMPONENT*, *PAL* as *ARGUMENT LIST*, and DA_v as *PTF DATA AREA*.

g) Any rows passed by a <pipe row statement> during the execution of those General Rules are inserted in $REST_v$.

h) For every t, 1 (one) $\leq t \leq NT$, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with CUR_t as *CURSOR* and DESTROY as *DISPOSITION*.

i) If the status variable in DA_v is not *successful completion (00000)*, then the value of the status variable is raised as an exception condition.

j) If *PTF* has a PTF finish component procedure, then:

i) The General Rules of Subclause 9.27, "Invocation of a PTF component procedure", are applied with *SR* as *POLYMORPHIC TABLE FUNCTION*, "finish" as *COMPONENT*, *PAL* as *ARGUMENT LIST*, and DA_v as *PTF DATA AREA*.

ii) Any rows passed by a <pipe row statement> during the execution of those General Rules are inserted in $REST_v$.

iii) If the status variable in DA_v is not *successful completion (00000)*, then the value of the status variable is raised as an exception condition.

k) The rows of $REST_v$ are inserted into *RESTAB*.

l) $REST_v$ is destroyed. DA_v and all of its components are destroyed. VP_v is destroyed.

21) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *RESTAB* as *RESULT TABLE*.

Conformance Rules

None.

9.26 Signatures of PTF component procedures

Function

Generate the signatures of the PTF component procedures of a polymorphic table function.

Subclause Signature

```
"Signatures of PTF component procedures" [Syntax Rules] (
  Parameter: "COMPONENT",
  Parameter: "PTF PARAMETER LIST",
  Parameter: "PRIVATE PARAMETER LIST"
) Returns: "COMPONENT PARAMETER LIST"
```

COMPONENT — the particular component for which a signature is to be generated ("describe", "start", "fulfill", or "finish").

PTF PARAMETER LIST — a list of parameters to be supplied as the non-private parameters of the PTF component.

PRIVATE PARAMETER LIST — a list of parameters to be supplied as the private parameters of the PTF component.

COMPONENT PARAMETER LIST — a normalized list of all parameters of the PTF component.

Syntax Rules

1) Let *COMP* be the *COMPONENT*, let *PTFPL* be the *PTF PARAMETER LIST*, and let *PRIVPL* be the *PRIVATE PARAMETER LIST* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *COMPONENT PARAMETER LIST*. .

2) Case:

   a) If the SQL-implementation supports Feature B208, "PTF component procedure interface", then:

      i) *COMPPL* is built iteratively by appending new parameters CP_c at the end of the list. Each parameter CP_c has a <parameter mode> and a <parameter type>; <SQL parameter name> and <parameter default> are not specified.

      ii) Let *ND* be the number of parameters in *PRIVPL*. Let D_d, 1 (one) ≤ d ≤ *ND* be an enumeration of the parameters in *PRIVPL*, in order of declaration.

      iii) For all d, 1 (one) ≤ d ≤ *ND*, let CP_d be a parameter in *COMPPL*, as follows:

         1) If *COMP* is "finish", then the <parameter mode> of CP_d is IN; otherwise, the <parameter mode> is INOUT.

         2) The <parameter type> of CP_d is the <parameter type> of D_d.

      iv) Let *M* be the number of parameters in *COMPPL*.

         NOTE 448 — At this point, *M* = *ND*.

      v) Let *NP* be the number of parameters in *PTFPL*. Let P_p, 1 (one) ≤ p ≤ *NP*, be an enumeration the parameters in *PTFPL*, in order of declaration.

vi) Let *L* be the implementation-defined (IL058) maximum length of a PTF extended name.

vii) For each *p*, 1 (one) ≤ *p* ≤ *NP*,

Case:

1) If the <parameter type> of P_p is <generic table parameter type>, then:

A) If *COMP* is "describe", then a parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

NOTE 449 — This is for the PTF extended name of the full row type descriptor name for the generic table.

B) If *COMP* is "fulfill", then a parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

NOTE 450 — This is for the PTF extended name of the row type descriptor of the cursor for the generic table.

C) If P_p has set semantics, then:

I) A parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

NOTE 451 — This is the partitioning descriptor of for the generic table.

II) A parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

NOTE 452 — This is the ordering descriptor of for the generic table.

D) If *COMP* is "describe", then a parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

NOTE 453 — This is for the PTF extended name of the requested row type descriptor for the cursor for P_p.

E) If *COMP* is "fulfill", then a parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

NOTE 454 — This is for the PTF extended name of the cursor of the generic input table.

2) If the <parameter type> of P_p is a descriptor type, then a parameter whose <parameter mode> is IN and whose <parameter type> is VARCHAR(*L*) is appended to *COMPPL*.

3) Otherwise, a parameter whose <parameter mode> is IN and whose <parameter type> is the <parameter type> of P_p is appended to *COMPPL*.

viii) If *COMP* is "describe", then the <returns clause> *RT* of *SR* does not specify <table function column list>, and *RT* does not specify RETURNS ONLY PASS THROUGH, then a parameter whose <parameter mode> is IN and whose <parameter type> is the <parameter type> of P_p is appended to *COMPPL*.

NOTE 455 — This is for the PTF extended name of the initial result row descriptor name.

ix) If *COMP* is not "describe", then a parameter whose <parameter mode> is IN and whose <parameter type> is the <parameter type> of P_p is appended to *COMPPL*.

NOTE 456 — This is for the PTF extended name of the intermediate result row descriptor name.

x) A parameter whose <parameter mode> is INOUT and whose <parameter type> is the CHARACTER(5) is appended to *COMPPL*.

NOTE 457 — The last parameter is for the status variable, a value of SQLSTATE.

b) Otherwise, *COMPPL* is implementation-defined (IV091).

3) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *COMPPL* as *COMPONENT PARAMETER LIST*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.27 Invocation of a PTF component procedure

Function

Invocation of a PTF component procedure.

Subclause Signature

```
"Invocation of a PTF component procedure" [General Rules] (
  Parameter: "POLYMORPHIC TABLE FUNCTION",
  Parameter: "COMPONENT",
  Parameter: "ARGUMENT LIST",
  Parameter: "PTF DATA AREA"
)
```

POLYMORPHIC TABLE FUNCTION — a polymorphic table function.

COMPONENT — the particular component to be invoked ("describe", "start", "fulfill", or "finish").

ARGUMENT LIST — an <SQL argument list>.

PTF DATA AREA — a PTF data area.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *SR* be the *POLYMORPHIC TABLE FUNCTION*, let *COMP* be the *COMPONENT*, let *AL* be the *ARGUMENT LIST*, and let *PTFDA* be the *PTF DATA AREA* in an application of the General Rules of this Subclause.

2) Case:

   a) If *COMP* is "describe", then let *COMPRT* be the PTF describe component procedure of *SR*.

   b) If *COMP* is "start", then let *COMPRT* be the PTF start component procedure of *SR*.

   c) If *COMP* is "fulfill", then let *COMPRT* be the PTF fulfill component procedure of *SR*.

   d) If *COMP* is "finish", then let *COMPRT* be the PTF finish component procedure of *SR*.

3) Case:

   a) If the SQL-implementation supports Feature B208, "PTF component procedure interface", then:

      i) The following rules construct *SAL*, a list of <SQL argument>s. *SAL* is initially an empty list of <SQL argument>s.

ii) Let *NPV* be the number of PTF private variables in *PTFDA*. Let the implementation-dependent (UV131) names of the PTF private variables be $PANAME_a$, 1 (one) ≤ a ≤ *NPV*. Let the first *NPV* elements of *SAL* be $PANAME_1$, ..., $PANAME_{NPV}$.

iii) Let *NP* be the number of parameters of *SR*. For all p, 1 (one) ≤ p ≤ *NP*, let P_p be the p-th parameter of *SR* and let A_p be the corresponding argument in *AL*.

iv) For all p, 1 (one) ≤ p ≤ *NP*,

Case:

1) If P_p is a generic table parameter, then:

A) Case:

I) If *COMP* is "describe", then a character string literal whose value is the PTF extended name of the PTF descriptor area for the row type of P_p is appended to *SAL*.

II) If *COMP* is "fulfill", then a character string literal whose value is the PTF extended name of the PTF descriptor area for the row type of the cursor for P_p is appended to *SAL*.

NOTE 458 — If *COMP* is "start" or "finish", then no row type descriptor name is appended to *SAL* at this point.

B) If P_p has set semantics, then:

I) A character string literal whose value is the PTF extended name of the PTF descriptor area for the partitioning of P_p is appended to *SAL*.

II) A character string literal whose value is the PTF extended name of the PTF descriptor area for the ordering of P_p is appended to *SAL*.

C) If *COMP* is "describe", then a character string literal whose value is the PTF extended name of the PTF extended name of the PTF descriptor area for the requested row type of P_p is appended to *SAL*.

D) If *COMP* is "fulfill", then a character string literal whose value is the PTF extended name of the PTF cursor for P_p is appended to *SAL*.

2) If P_p is a descriptor parameter, then

Case:

A) If A_p is the null value, then a null value is appended to *SAL*.

B) Otherwise, a character string literal whose value is a PTF extended name for the PTF descriptor area specified by A_p.

3) Otherwise, A_p is appended to *SAL*.

v) Case:

1) If *COMP* is "describe", the <returns clause> *RT* of *SR* does not specify <table function column list>, and *RT* does not specify RETURNS ONLY PASS THROUGH, then a character string literal whose value is the PTF extended name of the initial result row descriptor of *SR* is appended to *SAL*.

2) If *COMP* is not "describe", then a character string literal whose value is the PTF extended name of the intermediate result row descriptor of *SR* is appended to *SAL*.

vi) The PTF extended name of the status variable is appended to *SAL*.

b) Otherwise, *SAL* is implementation-defined (IV092).

4) The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *COMPRT* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *V* be the *VALUE* returned from the application of those General Rules.

NOTE 459 — The *V* returned is not used.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.28 XQuery regular expression matching

Function

Determine a list of match vectors to an XQuery regular expression in a character string.

Subclause Signature

```
"XQuery regular expression matching" [General Rules] (
  Parameter: "STRING",
  Parameter: "PATTERN",
  Parameter: "POSITION",
  Parameter: "UNITS",
  Parameter: "FLAG"
) Returns: "LIST"
```

STRING — a character string within which a regular expression match is to be sought.

PATTERN — a character string that is an XQuery regular expression.

POSITION — an integer specifying the position within STRING at which the regular expression match operation is to start.

UNITS — an indication of whether the regular expression match is performed in units of characters (CHARACTERS) or in units of octets (OCTETS).

FLAG — a character string that is an XQuery option flag.

LIST — a list of "match vectors", each of which is a pair of integers specifying the starting position and length of a regular expression match in STRING.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the *STRING*, let *PAT* be the *PATTERN*, let *SP* be the *POSITION*, let *U* be the *UNITS*, and let *FL* be the *FLAG* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *LIST*.

2) Case:

   a) If the character repertoire of *PAT* is UCS, then let *P* be *PAT*.

   b) Otherwise, let *P* be the result of an implementation-defined (IA068) transliteration of *PAT* to UCS.

3) Case:

   a) If the character repertoire of *FL* is UCS, then let *F* be *FL*.

b) Otherwise, let *F* be the result of an implementation-defined (IA068) transliteration of *FL* to UCS.

4) If *P* is not an XQuery regular expression, then an exception condition is raised: *data exception — invalid XQuery regular expression (2201S)*.

5) If *F* is not an XQuery option flag, then an exception condition is raised: *data exception — invalid XQuery option flag (2201T)*.

6) Let *LEN* be the length in characters of *S*.

7) A position/length *PL* is a pair of exact numeric values, a position *P* and a length *L*, such that either *P* = 0 (zero) and *L* = 0 (zero), or *P* is positive, *L* is non-negative, and *P* + *L* ≤ *LEN* + 1 (one). If *P* is positive, then *PL* represents the substring of *S* whose starting position is *P* and whose length is *L*. If *P* is 0 (zero), then *PL* does not represent a substring of *S*.

NOTE 460 — For example, if *S* is `'a'`, then there are four position/lengths, namely (1,0), denoting the zero-length substring at the beginning of *S*; (2,0), denoting the zero-length substring at the end of *S*; (1,1), denoting the whole string *S*; and (0,0), denoting no substring of *S*.

8) Let *G* be the number of parenthesized groups in *PAT*.

9) An *XQuery regular expression match* is defined as follows.

Case:

a) If the character repertoire of *S* is UCS, then an XQuery regular expression match is a match as defined by XQuery and XPath Functions and Operators 3.1 for the function `fn:matches()`, with *F* as the value of the parameter `$flags`, with the following modifications:

i) The metacharacters "^", "$", and " . " are interpreted according to the guidelines of RL1.6 "Line boundaries", in Unicode Technical Standard #18.

ii) The multi-character escape "\s" matches U+0020 (space), U+0009 (tab), or any single character or two-character sequence identified as a newline sequence by RL1.6 "line boundaries" in Unicode Technical Standard #18.

iii) The multi-character escape "\S" matches any single character that is not matched by a single character that matches the multi-character escape "\s".

b) Otherwise, the definition of XQuery regular expression match is implementation-defined (IV093).

10) A match vector *MV* is an array of *G* + 1 position/lengths, indexed from 0 (zero) through *G*. *MV* represents an XQuery regular expression match of *PAT*, including information about capture groups. *MV*[0] is the position/length that denotes the entire XQuery regular expression match of *PAT*. For *i* between 1 (one) and *G*, *MV*[*i*] is the position/length that denotes the *i*-th capture group within the XQuery regular expression match that is denoted by *MV*[0]. If the *i*-th parenthesized subexpression of *PAT* is not matched, then the position of *MV*[*i*] is 0 (zero).

NOTE 461 — The position of *MV*[0] is never 0 (zero), since an XQuery regular expression match to *PAT* can never be "no substring". An example of a capture group that is not matched occurs when *S* is `'a'` and *PAT* is `'(a)|(b)'`. In this example, the match vector is given by *MV*[0] = (1,1), *MV*[1] = (1,1), and *MV*[2] = (0,0), since the second parenthesized subexpression has no match.

11) The result of this Subclause is a list *LOMV* of match vectors, defined as follows:

a) Let P_0 be *SP* − 1 (one), and let L_0 be 0 (zero).

b) For $i \geq 1$ (one), P_i and L_i are defined recursively as follows. Let P_i be the position in units *U* of the first XQuery regular expression match of *PAT* in *S* such that $P_i > P_{i-1} + L_{i-1} - 1$ and $P_i > P_{i-1}$, and let L_i be the length in units *U* of this match.

NOTE 462 — The sequence P_i is strictly increasing and bounded above by *LEN* + 1 (one), so this recursive definition terminates after finding a finite number of disjoint matches.

c) Let *N* be the number of XQuery regular expression matches found.

NOTE 463 — It is possible that there are no matches, in which case *N* is 0 (zero), and *LOMV* is empty. (P_0, L_0) = (*SP* − 1, 0) is not included in the result; it is merely used as the position to start seeking the first match.

d) For *i* between 1 (one) and *N*, let MV_i be the match vector such that:

i) MV_i[0] is the position/length whose position is P_i and whose length is L_i.

ii) For *j* between 1 (one) and *G*, MV_i[*j*] is the position/length

Case:

1) If the *j*-th parenthesized subexpression of *PAT* is not matched in the *i*-th XQuery regular expression match, then position 0 (zero) and length 0 (zero).

2) Otherwise, the position and length of the *j*-th capture group within the *i*-th XQuery regular expression match.

e) Let *LOMV* be the list of *N* match vectors MV_i in order from 1 (one) to *N*.

12) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *LOMV* as *LIST*.

Conformance Rules

None.

9.29 XQuery regular expression replacement

Function

Replace a substring that matches an XQuery regular expression with a replacement character string.

Subclause Signature

```
"XQuery regular expression replacement" [General Rules] (
  Parameter: "STRING",
  Parameter: "PATTERN",
  Parameter: "MATCH",
  Parameter: "REPLACEMENT",
  Parameter: "FLAG"
) Returns: "RESULT"
```

STRING — a character string within which a regular expression match is to be sought.

PATTERN — a character string that is an XQuery regular expression.

MATCH — a "match vector" (a starting position and a length) that identifies a substring of STRING that matches the PATTERN.

REPLACEMENT — an XQuery regular expression replacement string.

FLAG — a character string that is an XQuery option flag.

RESULT — a copy of STRING in which the substring identified by MATCH has been replaced by REPLACEMENT.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *STR* be the *STRING*, let *PAT* be the *PATTERN*, let *MV* be the *MATCH*, let *REP* be the *REPLACEMENT*, and let *FL* be the *FLAG* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *RESULT*.

2) Let *PL* be *MV*[0]. Let *P* be the position of *PL* and let *L* be the length of *PL*.

3) If *L* is zero, then an exception condition is raised: *data exception — attempt to replace a zero-length string (2201U)*.

4) Case:

   a) If the character repertoire of *REP* is UCS, then let *R* be *REP*.

   b) Otherwise, let *R* be the result of an implementation-defined (IA069) conversion of *REP* to UCS.

5) If *R* is not an XQuery replacement string, then an exception condition is raised: *data exception — invalid XQuery replacement string (2201V)*.

6) Let STR_1 be the character string consisting of the first (P−1) characters of *STR*.

7) Let STR_2 be the character string consisting of the P-th through (P+L−1)-th character of *STR*.

8) Let STR_3 be the character string consisting of the (P+L)-th character through the last character of *STR*.

9) Case:

   a) If the character repertoire of *STR*, *PAT*, *REP*, and *FL* is UCS, then let *T* be the result of the XQuery function `fn:replace()`, where the value of the `$input` argument is STR_2, the value of the `$pattern` argument is *PAT*, the value of the `$replacement` argument is *R*, and the value of the `$flags` argument is *FL*.

   b) Otherwise, the value of *T* is implementation-defined (IV094).

10) Let *RETVAL* be

    STR_1 || *T* || STR_3

11) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *RETVAL* as *RESULT*.

Conformance Rules

None.

9.30 Data type identity

This Subclause is modified by Subclause 9.14, "Data type identity", in ISO/IEC 9075-15.

Function

Determine whether two data types are compatible and have the same characteristics.

Subclause Signature

```
"Data type identity" [Syntax Rules] (
  Parameter: "TYPE1",
  Parameter: "TYPE2"
)
```

TYPE1 — the first data type to be analyzed.

TYPE2 — the second data type to be analyzed.

Syntax Rules

1) Let *PM* be the *TYPE1* and let *P* be the *TYPE2* in an application of the Syntax Rules of this Subclause.

2) *PM* and *P* shall be compatible.

3) If *PM* is a fixed-length character string type, then the length of *PM* shall be equal to the length of *P*.

4) If *PM* is a variable-length character string type, then the maximum length of *PM* shall be equal to the maximum length of *P*.

5) If *PM* is an exact numeric type, then the precision and scale of *PM* shall be equal to the precision and scale of *P*, respectively.

6) If *PM* is an approximate numeric type or the decimal floating-point type, then the precision of *PM* shall be equal to the precision of *P*.

7) If *PM* is a fixed-length binary string type, then the length of *PM* shall be equal to the length of *P*.

8) If *PM* is a variable-length binary string type, then the maximum length of *PM* shall be equal to the maximum length of *P*.

9) If *PM* is a datetime data type with <time fractional seconds precision>, then the <time fractional seconds precision> of *PM* shall be equal to the <time fractional seconds precision> of *P*.

10) If *PM* is an interval type, then the <interval qualifier> of *PM* shall be equivalent to the <interval qualifier> of *P*.

11) If *PM* is a collection type, then:

    a) The kind of collection (ARRAY or MULTISET) of *PM* and the kind of collection of *P* shall be the same.

    b) [15] If *PM* is an array type, then the maximum cardinality of *PM* shall be equal to the maximum cardinality of *P*.

    c) The Syntax Rules of this Subclause are applied with *PM* as *TYPE1* and *P* as *TYPE2*.

12) If *PM* is a row type, then:

a) Let *N* be the degree of *PM*.

b) Let $DTFPM_i$ and $DTFP_i$, 1 (one) ≤ *i* ≤ *N*, be the data type of the *i*-th field of *PM* and of *P*, respectively. For *i* varying from 1 (one) to *N*, the Syntax Rules of this Subclause are applied with $DTFPM_i$ as *TYPE1* and $DTFP_i$ as *TYPE2*.

13) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.31 Determination of a from-sql function

Function

Determine the from-sql function of a user-defined type given the name of a user-defined type and the name of the group.

Subclause Signature

```
"Determination of a from-sql function" [Syntax Rules] (
  Parameter: "TYPE",
  Parameter: "GROUP"
) Returns: "FROM-SQL FUNCTION"
```

TYPE — a user-defined type.

GROUP — the name of a transform group.

FROM-SQL FUNCTION — the name of the from-sql function.

Syntax Rules

1) Let *UDT* be the *TYPE* and let *GN* be the *GROUP* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *FROM-SQL FUNCTION*.

2) Let *SSUDT* be the set of supertypes of *UDT*.

3) Let *SUDT* be the data type, if any, in *SSUDT* such that the transform descriptor included in the data type descriptor of *SUDT* includes a group descriptor *GD* that includes a group name that is equivalent to *GN*.

4) Let *FSQLF* be the the SQL-invoked function identified by the specific name of the from-sql function, if any, in *GD*.

5) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *FSQLF* as *FROM-SQL FUNCTION*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.32 Determination of a from-sql function for an overriding method

Function

Determine the from-sql function of a user-defined type given the name of an overriding method and the ordinal position of an SQL parameter.

Subclause Signature

```
"Determination of a from-sql function for an overriding method" [Syntax Rules] (
  Parameter: "ROUTINE",
  Parameter: "POSITION"
) Returns: "FROM-SQL FUNCTION"
```

ROUTINE — an <SQL-invoked routine> that is an original method associated with a user-defined type.

POSITION — an integer specifying the particular parameter of ROUTINE for which an applicable from-sql function is required.

FROM-SQL FUNCTION — the name of the from-sql function.

Syntax Rules

1) Let *R* be the *ROUTINE* and let *N* be the *POSITION* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *FROM-SQL FUNCTION*.

2) Let *OM* be original method of *R*.

3) Let *FSQLF* be the from-sql function associated with the *N*-th SQL parameter of *OM*, if any.

4) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *FSQLF* as *FROM-SQL FUNCTION*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.33 Determination of a to-sql function

Function

Determine the to-sql function of a user-defined type given the name of a user-defined type and the name of a group.

Subclause Signature

```
"Determination of a to-sql function" [Syntax Rules] (
  Parameter: "TYPE",
  Parameter: "GROUP"
) Returns: "TO-SQL FUNCTION"
```

TYPE — a user-defined type.

GROUP — the name of a transform group.

TO-SQL FUNCTION — the name of the to-sql function.

Syntax Rules

1) Let *UDT* be the *TYPE* and let *GN* be the *GROUP* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *TO-SQL FUNCTION*.

2) Let *SSUDT* be the set of supertypes of *UDT*.

3) Let *SUDT* be the data type, if any, in *SSUDT* such that the transform descriptor included in the data type descriptor of *SUDT* includes a group descriptor *GD* that includes a group name that is equivalent to *GN*.

4) Let *TSQLF* be the SQL-invoked function identified by the specific name of the to-sql function, if any, in *GD*.

5) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *TSQLF* as *TO-SQL FUNCTION*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.34 Determination of a to-sql function for an overriding method

Function

Determine the to-sql function of a user-defined type given the name of an overriding method.

Subclause Signature

```
"Determination of a to-sql function for an overriding method" [Syntax Rules] (
  Parameter: "ROUTINE"
) Returns: "TO-SQL FUNCTION"
```

ROUTINE — an <SQL-invoked routine> that is an original method associated with a user-defined type.

TO-SQL FUNCTION — the name of the to-sql function.

Syntax Rules

1) Let *R* be the *ROUTINE* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *TO-SQL FUNCTION*.

2) Let *OM* be the original method of *R*

3) Let *TSQLF* be the to-sql function associated with the result of *OM*, if any.

4) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *TSQLF* as *TO-SQL FUNCTION*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

9.35 Generation of the next value of a sequence generator

Function

Generate and return the next value of a sequence generator.

Subclause Signature

```
"Generation of the next value of a sequence generator" [General Rules] (
  Parameter: "SEQUENCE"
) Returns: "RESULT"
```

SEQUENCE — a sequence generator descriptor.

RESULT — the next available value of the sequence.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *SEQ* be the *SEQUENCE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *RESULT*.

2) Let *DT*, *CBV*, *INC*, *SMAX*, and *SMIN* be the data type, current base value, increment, maximum value and minimum value, respectively, of *SEQ*.

3) If there exists a non-negative integer *N* such that $SMIN \le CBV + N * INC \le SMAX$ and the value ($CBV + N * INC$) has not already been returned in the current cycle, then let *V1* be ($CBV + N * INC$). Otherwise,

   Case:

   a) If the cycle option of *SEQ* is NO CYCLE, then an exception condition is raised: *data exception — sequence generator limit exceeded (2200H)*.

   b) Otherwise, a new cycle is initiated.

      Case:

      i) If *SEQ* is an ascending sequence generator, then let *V1* be *SMIN*.

      ii) Otherwise, let *V1* be *SMAX*.

4) Case:

   a) If *SEQ* is an ascending sequence generator, the current base value of *SEQ* is set to the value of the lowest non-issued value in the cycle.

   b) Otherwise, the current base value of *SEQ* is set to the highest non-issued value in the cycle.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *V1* as *RESULT*.

Conformance Rules

None.

9.36 Creation of a sequence generator

Function

Complete the definition of an external or internal sequence generator.

Subclause Signature

```
"Creation of a sequence generator" [Syntax Rules] (
  Parameter: "OPTIONS",
  Parameter: "DATA TYPE"
)
```

OPTIONS — a character string representing the <common sequence generator options>.

DATA TYPE — a data type that the sequence generator returns.

```
"Creation of a sequence generator" [General Rules] (
  Parameter: "OPTIONS",
  Parameter: "DATA TYPE"
) Returns: "SEQGENDESC"
```

OPTIONS — a character string representing the <common sequence generator options>.

DATA TYPE — a data type that the sequence generator returns.

SEQGENDESC — a sequence generator descriptor that is created as a result of an invocation of this subclause using this signature.

Syntax Rules

1) Let *OPT* be the *OPTIONS* and let *DT* be the *DATA TYPE* in an application of the Syntax Rules of this Subclause.

2) *OPT* shall conform to the Format of <common sequence generator options>. The BNF non-terminal symbols used in the remainder of this Subclause refer to the contents of *OPT*.

3) Each of <sequence generator start with option>, <sequence generator increment by option>, <sequence generator max-value option>, <sequence generator min-value option>, and <sequence generator cycle option> shall be specified at most once.

4) If <sequence generator increment by option> is specified, then let *INC* be <sequence generator increment>; otherwise, let *INC* be a <signed numeric literal> whose value is 1 (one).

5) The value of *INC* shall not be 0 (zero).

6) If the value of *INC* is negative, then the sequence generator *SEQ* to be created is a *descending sequence generator*; otherwise, *SEQ* is an *ascending sequence generator*.

7) Case:

   a) If <sequence generator max-value option> is specified, then

      Case:

      i) If NO MAXVALUE is specified, then let *SMAX* be an implementation-defined (ID112) <signed numeric literal> of declared type *DT*.

ii) Otherwise, let *SMAX* be <sequence generator max value>.

b) Otherwise, let *SMAX* be an implementation-defined (ID112) <signed numeric literal> of declared type *DT*.

8) Case:

a) If <sequence generator min-value option> is specified, then

Case:

i) If NO MINVALUE is specified, then let *SMIN* be an implementation-defined (ID113) <signed numeric literal> of declared type *DT*.

ii) Otherwise, let *SMIN* be <sequence generator min value>.

b) Otherwise, let *SMIN* be an implementation-defined (ID113) <signed numeric literal> of declared type *DT*.

9) Case:

a) If <sequence generator start with option> is specified, then let *START* be <sequence generator start value>.

b) Otherwise,

Case:

i) If *SEQ* is an ascending sequence generator, then let *START* be *SMIN*.

ii) Otherwise, let *START* be *SMAX*.

10) The values of *INC*, *START*, *SMAX*, and *SMIN* shall all be exactly representable with the precision and scale of *DT*.

11) The value of *SMAX* shall be greater than the value of *SMIN*.

12) The value of *START* shall be greater than or equal to the value of *SMIN* and less than or equal to the value of *SMAX*.

13) If <sequence generator cycle option> is specified, then let *CYC* be <sequence generator cycle option>; otherwise, let *CYC* be NO CYCLE.

14) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

1) Let *OPT* be the *OPTIONS* and let *DT* be the *DATA TYPE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *SEQGENDESC*.

2) A sequence generator descriptor *SEQDS* that describes *SEQ* is created. *SEQDS* includes:

a) The sequence generator name that is the zero-length character string.

NOTE 464 — The name of an external sequence generator is later set by GR 1) of Subclause 11.72, "<sequence generator definition>"; however, internal sequence generators are anonymous.

b) The data type descriptor of *DT*.

c) The start value, set to *START*.

d) The minimum value specified by *SMIN*.

e) The maximum value specified by *SMAX*.

f) The increment specified by *INC*.

g) The cycle option specified by *CYC*.

h) The current base value, set to *START*.

3) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *SEQDS* as *SEQGENDESC*.

Conformance Rules

None.

9.37 Altering a sequence generator

Function

Complete the alteration of an internal or external sequence generator.

Subclause Signature

```
"Altering a sequence generator" [Syntax Rules] (
  Parameter: "OPTIONS",
  Parameter: "SEQUENCE"
)
```

OPTIONS — a character string representing the <alter sequence generator options>.

SEQUENCE — a sequence generator descriptor.

```
"Altering a sequence generator" [General Rules] (
  Parameter: "OPTIONS",
  Parameter: "SEQUENCE"
)
```

OPTIONS — a character string representing the <alter sequence generator options>.

SEQUENCE — a sequence generator descriptor.

Syntax Rules

1) Let *OPT* be the *OPTIONS* and let *SEQ* be the *SEQUENCE* in an application of the Syntax Rules of this Subclause.

2) *OPT* shall conform to the Format of <alter sequence generator options>. The BNF non-terminal symbols used in the remainder of this Subclause refer to the contents of *OPT*.

3) Let *DT* be the data type descriptor included in *SEQ*.

4) Each of <alter sequence generator restart option>, <sequence generator increment by option>, <sequence generator max-value option>, <sequence generator min-value option>, and <sequence generator cycle option> shall be specified at most once.

5) Case:

   a) If <sequence generator increment> is specified, then:

      i) Let *NEWIV* be <sequence generator increment>.

      ii) The value of *NEWIV* shall not be 0 (zero).

   b) Otherwise, let *NEWIV* be the increment of *SEQ*.

6) Case:

   a) If <sequence generator max-value option> is specified, then

      Case:

      i) If NO MAXVALUE is specified, then let *NEWMAX* be an implementation-defined (ID112) <signed numeric literal> of declared type *DT*.

ii) Otherwise, let *NEWMAX* be <sequence generator max value>.

b) Otherwise let *NEWMAX* be the maximum value of *SEQ*.

7) Case:

a) If <sequence generator min-value option> is specified, then

Case:

i) If NO MINVALUE is specified, then let *NEWMIN* be an implementation-defined (ID113) <signed numeric literal> of declared type *DT*.

ii) Otherwise, let *NEWMIN* be <sequence generator min value>.

b) Otherwise let *NEWMIN* be the minimum value of *SEQ*.

8) If <sequence generator cycle option> is specified, then let *NEWCYCLE* be <sequence generator cycle option>; otherwise, let *NEWCYCLE* be the cycle option of *SEQ*.

9) Case:

a) If <alter sequence generator restart option> is not specified, then let *NEWVAL* be the current base value of *SEQ*.

b) If <alter sequence generator restart option> is specified and contains a <sequence generator restart value>, then let *NEWVAL* be that <sequence generator restart value>.

c) Otherwise (<alter sequence generator restart option> is specified but does not contain a <sequence generator restart value>), let *NEWVAL* be the start value of *SEQ*.

10) The values of *NEWIV*, *NEWMAX*, *NEWMIN*, and *NEWVAL* shall all be exactly representable with the precision and scale of *DT*.

11) The value of *NEWMIN* shall be less than the value of *NEWMAX*.

12) The value of *NEWVAL* shall be greater than or equal to the value of *NEWMIN* and less than or equal to the value of *NEWMAX*.

13) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

1) Let *OPT* be the *OPTIONS* and let *SEQ* be the *SEQUENCE* in an application of the General Rules of this Subclause.

2) *SEQ* is modified as follows:

a) The increment is set to *NEWIV*.

b) The maximum value is set to *NEWMAX*.

c) The minimum value is set to *NEWMIN*.

d) The cycle option is set to *NEWCYCLE*.

e) The current base value is set to *NEWVAL*.

3) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.38 Generation of the hierarchical <query expression> of a view

Function

Generate the hierarchical <query expression> of a view.

Subclause Signature

```
"Generation of the hierarchical <query expression> of a view" [General Rules] (
  Parameter: "VIEW"
)
```

VIEW — a view.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *V* be the *VIEW* in an application of the General Rules of this Subclause.
2) Let *OQE* be the original <query expression> of *V*.
3) Case:
   a) If *V* has no proper subtables, then let *QE* be *OQE*.
   b) Otherwise:
      i) Let *N* be the number of proper subtables of *V*.
      ii) Let TN_1, ..., TN_N be an enumeration of the <table name>s of the proper subtables of *V*.
      iii) Let *QE* be

```
  OQE
UNION ALL CORRESPONDING
  SELECT * FROM ONLY (TN1)
UNION ALL CORRESPONDING
  ...
UNION ALL CORRESPONDING
  SELECT * FROM ONLY (TNN)
```

4) The hierarchical <query expression> in the view descriptor of *V* is replaced by *QE*.
5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.39 Determination of view privileges

Function

Determine the privilege descriptors of the view whose action is INSERT, UPDATE, or DELETE. In addition, determine the view privilege dependency descriptors of the view.

Subclause Signature

```
"Determination of view privileges" [General Rules] (
  Parameter: "VIEW"
)
```

VIEW — a view.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *V* be the *VIEW* in an application of the General Rules of this Subclause.

2) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <table name> of *V*.

3) The General Rules of Subclause 9.40, "Determination of view component privileges", are applied with *V* as *VIEW*; let the set of view component privilege descriptors *SVCPD* be the *DESCRIPTOR SET* returned from the application of those General Rules.

4) A view component privilege descriptor *SPD* in *SVCPD* is *simply dependent* on another privilege descriptor *PD* if *SPD* is immediately dependent on *PD*, or if there is a view component privilege descriptor *SPD2* such that *SPD* is immediately dependent on *SPD2* and *SPD2* is simply dependent on *PD*.

5) Let VC_N be the view component that is the original <query expression> included in the view descriptor of *V*.

   a) For each view component table privilege descriptor *VCTPD* in *SVCPD* whose identified object is VC_N:

      i) A privilege descriptor *PD1* is created, as follows: the identified object is *V*, the action is the same as the action of *VCTPD*, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if *VCTPD* is grantable.

      ii) For each privilege descriptor *PD* such that *VCTPD* is simply dependent on *PD*, and such that the object of *PD* is not a view component or a column of a view component, a view privilege dependency descriptor is created, as follows: the supporting privilege descriptor is *PD* and the dependent privilege descriptor is *PD1*.

b) For each view component column privilege descriptor *VCCPD* in *SVCPD* whose identified object is VC_N:

i) A privilege descriptor *PD2* is created, as follows: the identified object is the column of *V* that is the counterpart of the column identified by *VCCPD*, the action is the same as the action of *VCCPD*, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if *VCCPD* is grantable.

ii) For each privilege descriptor *PD* such that *VCCPD* is simply dependent on *PD*, and such that the object of *PD* is not a view component or a column of a view component, a view privilege dependency descriptor is created, as follows: the supporting privilege descriptor is *PD* and the dependent privilege descriptor is *PD2*.

6) If, for every column *C* of *V*, there is a column privilege descriptor *CPD* whose identified object is *C*, action is UPDATE, grantor is the special grantor value "_SYSTEM", and grantee is *A*, then it is implementation-defined (IA220) whether a table privilege descriptor *TPD* is created whose identified object is *V*, action is UPDATE, grantor is the special grantor value "_SYSTEM", grantee is *A*. If such a table privilege descriptor is created, then it is directly dependent on every such column privilege descriptor, and it has an indication that the privilege is grantable if every such column privilege descriptor has an indication that it is grantable.

If *TPD* is created, then a collection of view privilege dependency descriptors is created, one for each column *C* of *V*, in which the supporting privilege descriptor is *CPD* and the dependent privilege descriptor is *TPD*.

7) If, for every column *C* of *V*, there is a column privilege descriptor *CPD* whose identified object is *C*, action is INSERT, grantor is the special grantor value "_SYSTEM", grantee is *A*, then it is implementation-defined (IA221) whether a table privilege descriptor *TPD* is created whose identified object is *V*, action is INSERT, grantor is the special grantor value "_SYSTEM", grantee is *A*. If such a table privilege descriptor is created, then it is directly dependent on every such column privilege descriptor, and it has an indication that the privilege is grantable if every such column privilege descriptor has an indication that it is grantable.

If *TPD* is created, then a collection of view privilege dependency descriptors is created, one for each column *C* of *V*, in which the supporting privilege descriptor is *CPD* and the dependent privilege descriptor is *TPD*.

8) All view component privilege descriptors in *SVCPD* are destroyed.

9) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

9.40 Determination of view component privileges

Function

Determine view component privilege descriptors for all view components of a view.

Subclause Signature

```
"Determination of view component privileges" [General Rules] (
  Parameter: "VIEW"
) Returns: "DESCRIPTOR SET"
```

VIEW — a view.

DESCRIPTOR SET — a set of view component table privilege descriptors and view component column privilege descriptors that describe the privileges defined on VIEW.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *V* be the *VIEW* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *DESCRIPTOR SET*.

2) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <table name> of *V*.

3) Let VC_1, ..., VC_N be an enumeration of the view components of *V*. The enumeration shall have the following properties:

   a) For all *i* and *j* between 1 (one) and *N*, if VC_i is contained in VC_j, then $i < j$.

   b) For all *i* and *j* between 1 (one) and *N*, if VC_i is a <query expression> simply contained in a <with list element> that is simply contained in a <with clause> that does not specify RECURSIVE and VC_j references the table defined by VC_i, then $i < j$.

      NOTE 465 — A depth-first left-to-right traversal of the BNF of the original <query expression> *OQE* of *V* is one way to obtain such an enumeration. Necessarily, VC_N is *OQE*.

4) If *V* is effectively updatable, then the following subrules are performed to recursively create certain view component privilege descriptors. The following subrules also recursively define when a view component privilege is *immediately dependent* on another privilege descriptor.

   For each *i* between 1 (one) and *N*, in that order,

   Case:

a) If VC_i is not effectively updatable, then there are no view component privilege descriptors that identify VC_i or any column of VC_i as object.

NOTE 466 — This case includes <table value constructor>.

b) If VC_i is a <query specification>, then

Case:

i) If the <from clause> of VC_i contains exactly one <table reference> *TR*, then:

NOTE 467 — By a syntactic transformation in Subclause 7.17, "<query expression>", this case also includes <explicit table>.

1) Case:

A) If *TR* is a <table name> *TN* or an <only spec> that simply contains a <table name> *TN*, then let *S* be the set of applicable privileges for *A* on the table referenced by *TN*.

B) If *TR* is a <query name> *QN* or an <only spec> that simply contains a <query name> *QN*, then *QN* references some VC_j, where $j < i$. Let *S* be the set of view component privilege descriptors whose identified object is VC_j or a column of VC_j.

NOTE 468 — If *QN* is defined in a WITH clause that specifies RECURSIVE, then the table identified by *QN* is not effectively updatable, so the <query specification> VC_i is also not effectively updatable, and therefore VC_i has no view component privilege descriptors, by a prior rule.

C) If *TR* is a <derived table> or <lateral derived table>, then let *QE* be the <query expression> simply contained in *TR*. *TR* is some VC_j, where $j < i$. Let *S* be the set of view component privilege descriptors whose identified object is VC_j or a column of VC_j.

D) If *TR* is a <parenthesized joined table>, then let *JT* be the <joined table> simply contained in *TR*. Let *QS* be the <query specification> obtained from VC_i by replacing *TR* with *JT*. The rules of this Subclause are effectively applied as if VC_i were replaced by *QS*.

NOTE 469 — This will reclassify VC_i as a <query specification> with a <from clause> containing more than one <table reference>, considered in subsequent rules. The list of view components VC_1, ..., VC_N is not changed by this transformation.

NOTE 470 — All other cases of <table reference> are not effectively updatable, so they cannot arise here.

2) If *S* contains a table privilege descriptor *PD* whose action is DELETE, then a view component table privilege descriptor is created as follows: the identified object is VC_i, the action is DELETE, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if *PD* indicates a grantable privilege. The privilege descriptor is immediately dependent on *PD*.

3) For each updatable column *C* of VC_i, let *CC* be the counterpart of *C* in the table identified by *TR*.

A) If *S* contains a column privilege descriptor *PD* whose action is UPDATE and whose object is *CC*, then a view component column privilege descriptor is created as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The

privilege is grantable if and only if *PD* indicates a grantable privilege. The privilege descriptor is immediately dependent on *PD*.

B) If *V* is insertable-into, and *S* contains a column privilege descriptor *PD* whose action is INSERT and whose object is *CC*, then a view component column privilege descriptor is created as follows: the identified object is *C*, the action is INSERT, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if *PD* indicates a grantable privilege. The privilege descriptor is immediately dependent on *PD*.

ii) Otherwise:

1) If, for every target leaf underlying table *LUT* of VC_i, the applicable privileges for *A* include DELETE on *LUT*, then a view component table privilege descriptor is created whose identified object is VC_i, action is DELETE, grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privileges for *A* includes grantable DELETE privilege on each such *LUT*. The privilege descriptor is immediately dependent on every privilege descriptor whose identified object is such a target leaf underlying table, the action is DELETE, and grantee is *A*.

2) For each updatable column *C* of VC_i, let *LUT* be the target leaf underlying table of VC_i that has a counterpart *CC* to *C*.

A) If the applicable privileges for *A* include UPDATE privilege on *CC*, then a view component column privilege descriptor *VCCPD* is created, as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privilege for *A* includes grantable UPDATE privilege on *CC*. The privilege descriptor is immediately dependent on every privilege descriptor whose identified object is *CC*, action is *UPDATE*, and grantee is *A*.

B) If *V* is insertable-into, and the applicable privileges for *A* include INSERT privilege on *CC*, then a view component column privilege descriptor *VCCPD* is created, as follows: the identified object is *C*, the action is INSERT, the grantor is the special grantor value "_SYSTEM", and the grantee is *A*. The privilege is grantable if and only if the applicable privilege for *A* includes grantable INSERT privilege on *CC*. The privilege descriptor is immediately dependent on every privilege descriptor whose identified object is *CC*, action is INSERT, and grantee is *A*.

c) If *VC* is a <query expression>, then

Case:

i) If VC_i is a <simple table> *ST*, then there exists a $j < i$ such that *ST* is VC_j.

1) For each view component table privilege descriptor *VCTPD* of VC_j, a new view component table privilege descriptor is created, as follows: the identified object is VC_i, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the action and the indication of whether the privilege is grantable are the same as in *VCTPD*. The privilege descriptor is immediately dependent on *VCTPD*.

2) For each view component column privilege descriptor *VCCPD* of VC_j, a new view component column privilege descriptor of VC_i is created, as follows: the identified object is the column of VC_i that is the counterpart of the column of VC_j identified

by *VCCPD*, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the action and the indication of whether the privilege is grantable are the same as in *VCCPD*. The privilege descriptor is immediately dependent on *VCCPD*.

ii) Otherwise, VC_i immediately contains UNION ALL. Let VC_l and VC_r be the left and right operands of VC_i, respectively.

1) If there is a view component table privilege descriptor $VCTPD_l$ whose identified object is VC_l and whose action is DELETE, and there is a view component table privilege descriptor $VCTPD_r$ whose identified object is VC_r and whose action is DELETE, then a new view component table privilege descriptor is created as follows: the identified object is VC_i, the action is DELETE, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if both $VCTPD_l$ and $VCTPD_r$ indicate that the privilege is grantable. The privilege descriptor is immediately dependent on $VCTPD_l$ and $VCTPD_r$.

2) For each updatable column *C* of VC_i, let C_l and C_r be the counterparts of *C* in VC_l and VC_r, respectively. If there is a view component column privilege descriptor $VCCPD_l$ whose identified object is C_l and whose action is UPDATE, and there is a view component column privilege descriptor $VCCPD_r$ whose identified object is C_r and whose action is UPDATE, then a new view component column privilege descriptor is created as follows: the identified object is *C*, the action is UPDATE, the grantor is the special grantor value "_SYSTEM", the grantee is *A*, and the privilege is grantable if and only if both $VCTPD_l$ and $VCCPD_r$ indicate that the privilege is grantable. The privilege descriptor is immediately dependent on $VCCPD_l$ and $VCCPD_r$.

5) Let *SVCPD* be the set of view component table privilege descriptors and view component column privilege descriptors created by the General Rules of this Subclause.

6) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *SVCPD* as *DESCRIPTOR SET*.

Conformance Rules

None.

9.41 Row pattern recognition in a sequence of rows

Function

Find row pattern matches in a sequence of rows.

Subclause Signature

```
"Row pattern recognition in a sequence of rows" [General Rules] (
  Parameter: "ROW SEQUENCE",
  Parameter: "PATTERN",
  Parameter: "SUBSETS",
  Parameter: "DEFINES"
) Returns: "SET OF MATCHES"
```

ROW SEQUENCE — a sequence of rows (i.e., a derived table).

PATTERN — a <row pattern>.

SUBSETS — a <row pattern subset clause>.

DEFINES — a <row pattern definition list>.

SET OF MATCHES — a set of "row pattern matches", each of which comprises an identifier, a subsequence of rows that match the portion of the pattern, and the number of rows in the subsequence.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *RS* be the *ROW SEQUENCE*, let *P* be the *PATTERN*, let *S* be the *SUBSETS*, and let *D* be the *DEFINES* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *SET OF MATCHES*.

   NOTE 471 — *S*, the *SUBSETS* parameter, is not referenced explicitly in the following rules. The reason for its inclusion is that it is implicitly required in order to evaluate any <row pattern definition search condition>s that involve union row pattern variables.

2) Let *V* be the set of primary row pattern variables in *P*.

3) Let *NP* be the number of rows in *RS*. Let the rows of *RS* form the sequence R_1, R_2, ..., R_{NP}.

4) Let "(", ")", "[", "]", "$", "^", and "-" be distinct <identifier>s, known as *special symbols*, that are not equivalent to any primary row pattern variable names. In the context of these rules, an *<identifier> string* is a finite sequence of primary row pattern variables and special symbols. Various symbols from set theory, including "{", "}", ", ", "|", and "∪" are used in their set-theoretical meaning to form expressions denoting sets of <identifier> strings, and are distinct from row pattern variables and from special symbols.

5) The *variable count* of an <identifier> string *STR* is the number of row pattern variables in *STR*, counting duplicates. The variable count of *STR* is denoted *Vcount*(*STR*).

6) For each row pattern *RP*, the parenthesized language *PL*(*RP*) and a total ordering (called *preferment*, as in "STR_1 is preferred over STR_2") of *PL*(*RP*) are defined recursively as follows:

a) If *RP* is a <row pattern primary> *RPP*, then

Case:

i) If *RPP* is a primary row pattern variable, then *PL*(*RPP*) = { *RPP* }. As a singleton set, no preferment is defined in *PL*(*RPP*).

ii) If *RPP* is <dollar sign>, then *PL*(*RPP*) = { "$" }. As a singleton set, no preferment is defined in *PL*(*RPP*).

iii) If *RPP* is <circumflex>, then *PL*(*RPP*) = { "^" }. As a singleton set, no preferment is defined in *PL*(*RPP*).

iv) If *RPP* is <left paren> <right paren>, then *PL*(*RPP*) = { "(" ")" }. As a singleton set, no preferment is defined in *PL*(*RPP*).

v) If *RPP* is <left paren> <row pattern> <right paren>, then let *RP* be the simply contained <row pattern>. *PL*(*RPP*) = { "(" *STR* ")" | *STR* is in *PL*(*RP*) }. If "(" STR_1 ")" and "(" STR_2 ")" are two members of *PL*(*RPP*), then "(" STR_1 ")" is preferred over "(" STR_2 ")" if STR_1 is preferred over STR_2.

vi) If *RPP* is <left brace minus> <row pattern> <right minus brace>, then let *RP* be the simply contained <row pattern>. *PL*(*RPP*) = { "[" *STR* "]" | *STR* is in *PL*(*RP*) }. If "[" STR_1 "]" and "[" STR_2 "]" are two members of *PL*(*RPP*), then "[" STR_1 "]" is preferred over "[" STR_2 "]" if STR_1 is preferred over STR_2.

NOTE 472 — The special symbols "[" and "]" correspond to "{-" and "-}" in SQL language. The reason for using different symbols is to avoid confusion with "{" and "}" used as set-theoretic notation.

b) If *RP* is a <row pattern factor> immediately containing a <row pattern primary> *RPF* and {*n*,*m*} or {*n*,} is a <row pattern quantifier> *greedy*, then let *nn* be the maximum of 0 (zero) and *n*−1.

i) *PL*(*RPF* {*n*,*m*}) = { "(" STR_1 ... STR_p ")" | $n \le p \le m$; and for all *i*, 1 (one) $\le i \le p$, STR_i is in *PL*(*RPF*); and for all *i*, *nn* < *i* < *p*, *Vcount*(STR_i) > 0 (zero) }.

NOTE 473 — An empty match (*Vcount*(STR_i) = 0 (zero)) is permitted in the first *nn* matches (STR_1 through STR_{nn}) and also in the last match (STR_p).

ii) *PL*(*RPF* {*n*,}) = { "(" STR_1 ... STR_p ")" | $n \le p$; and for all *i*, 1 (one) $\le i \le p$, STR_i is in *PL*(*RPF*); and for all *i*, *nn* < *i* < *p*, *Vcount*(STR_i) > 0 (zero) }.

NOTE 474 — An empty match (*Vcount*(STR_i) = 0 (zero)) is permitted in the first *nn* matches (STR_1 through STR_{nn}) and also in the last match (STR_p).

iii) Given two <identifier> strings "(" $STRA_1$ ... $STRA_p$ ")" and "(" $STRB_1$ ... $STRB_q$ ")" in *PL*(*RPF greedy*), let *r* be the greatest integer such that $r \ge 0$ (zero), $r \le p$, $r \le q$, and for all *s*, 1 (one) $\le s \le r$, $STRA_s = STRB_s$. "(" $STRA_1$ ... $STRA_p$ ")" is preferred over "(" $STRB_1$ ... $STRB_q$ ")" if and only if exactly one of the following is true:

1) $r = q$ and $q < p$.

2) $r < q$, $r < p$, and $STRA_{r+1}$ is preferred over $STRB_{r+1}$.

Otherwise, "(" $STRB_1$ ... $STRB_q$ ")" is preferred over "(" $STRA_1$ ... $STRA_p$ ")".

NOTE 475 — *r* is the length of the longest initial substring of *STRA* and *STRB* on which they are equivalent. *r* might be zero, in which case *STRA* and *STRB* have no common initial substring aside from the zero-length character string. *STRA* is preferred over *STRB* if *STRB* is an initial substring of *STRA*, or if the first component of *STRA* that differs from the corresponding component of *STRB* is preferred over its corresponding component.

c) If *RP* is a <row pattern factor> immediately containing a <row pattern primary> *RPF* and {*n*,*m*}? or {*n*,}? is a <row pattern quantifier> *reluctant*, then:

i) *PL*(*RPF* {*n*,*m*}?) = *PL*(*RPF* {*n*,*m*})

ii) *PL*(*RPF* {*n*,}?) = *PL*(*RPF* {*n*,})

iii) Given two <identifier> strings "(" $STRA_1$ ... $STRA_p$ ")" and "(" $STRB_1$ ... $STRB_q$ ")" in *PL*(*RPF reluctant*), let *r* be the greatest integer such that $r \geq 0$ (zero), $r \leq p$, $r \leq q$, and for all *s*, 1 (one) $\leq s \leq r$, $STRA_s = STRB_s$. "(" $STRA_1$ ... $STRA_p$ ")" is preferred over "(" $STRB_1$ ... $STRB_q$ ")" if and only if exactly one of the following is true:

1) $r = p$ and $p < q$

2) $r < q$, $r < p$, and $STRA_{r+1}$ is preferred over $STRB_{r+1}$.

Otherwise, "(" $STRB_1$ ... $STRB_q$ ")" is preferred over "(" $STRA_1$ ... $STRA_p$ ")".

NOTE 476 — *r* is the length of the longest initial substring of *STRA* and *STRB* on which they are equivalent. *r* might be zero, in which case *STRA* and *STRB* have no common initial substring aside from the zero-length character string. *STRA* is preferred over *STRB* if *STRA* is an initial substring of *STRB*, or if the first component of *STRA* that differs from the corresponding component of *STRB* is preferred over its corresponding component.

d) If *RP* is a concatenation *RPT RPF*, where *RPT* is a <row pattern term> and *RPF* is a <row pattern factor>, then *PL*(*RPT RPF*) = { "(" STR_1 STR_2 ")" | STR_1 is in *PL*(*RPT*) and STR_2 is in *PL*(*RPF*) }. Given two <identifier> strings "(" $STRA_1$ $STRA_2$ ")" and "(" $STRB_1$ $STRB_2$ ")", "(" $STRA_1$ $STRA_2$ ")" is preferred over "(" $STRB_1$ $STRB_2$ ")" if either $STRA_1$ is preferred over $STRB_1$ or if $STRA_1$ is equivalent to $STRB_1$ and $STRA_2$ is preferred over $STRB_2$.

NOTE 477 — Consider the pattern (A|B)(C|D). In order of preferment, the <identifier> strings in the parenthesized language are:

```
"(" A C ")"
"(" A D ")"
"(" B C ")"
"(" B D ")"
```

e) If *RP* is a <row pattern alternation>, then let RP_1 be the first operand (the simply contained <row pattern>) and let RP_2 be the second operand (the simply contained <row pattern term>).

i) *PL*(*RP*) = { "(" STR_1 "-" ")" | STR_1 is in *PL*(RP_1) } ∪ { "(" "-" STR_2 ")" | STR_2 is in *PL*(RP_2) }

ii) Every member of { "(" STR_1 "-" ")" | STR_1 is in *PL*(RP_1) } is preferred over every member of { "(" "-" STR_2 ")" | STR_2 is in *PL*(RP_2) }.

iii) Given two <identifier> strings "(" *STRA* "-" ")" and "(" *STRB* "-" ")" in *PL*(*RP*), "(" *STRA* "-" ")" is preferred over "(" *STRB* "-" ")" if *STRA* is preferred over *STRB*.

iv) Given two <identifier> strings "(" "-" *STRA* ")" and "(" "-" *STRB* ")" in *PL*(*RP*), "(" "-" *STRA* ")" is preferred over "(" "-" *STRB* ")" if *STRA* is preferred over *STRB*.

7) A *potential row pattern match* (*STR*, *RS*, *k*) consists of an <identifier> string *STR* in *PL*(*RP*), a row sequence *RS* = {R_1, R_2, ..., R_{NP} }, and a positive integer *k* such that $k + Vcount(STR) - 1 \leq NP$. *k* is called the *initial row number* of the potential row pattern match.

8) Given a potential row pattern match (*STR*, *RS*, *k*), let V_j, 1 (one) ≤ *j* ≤ *Vcount*(*STR*), be an enumeration of the primary row pattern variables in *STR*, in order of occurrence in *STR*, and with duplicates. For all *j*, 1 (one) ≤ *j* ≤ *Vcount*(*STR*), row R_{k+j-1} in the row sequence *RS* is mapped to the primary row pattern variable V_j, and to each union row pattern variable that has V_j as a component.

9) A *candidate row pattern match* is a potential row pattern match (*STR*, *RS*, *k*) such that all of the following are true:

   a) For all *j*, 1 (one) ≤ *j* ≤ *Vcount*(*STR*), the value of the <row pattern definition search condition> contained in *D* that defines V_j is *True* for the row R_{k+j-1} that is mapped to V_j.

   b) If *STR* contains "^", then no row pattern variable precedes "^" in *STR*, and *k* = 1 (one).

   c) If *STR* contains "$", then no row pattern variable follows "$" in *STR*, and *k* + *Vcount*(*STR*) − 1 = *NP*.

10) A *preferred row pattern match* is a candidate row pattern match (*STR*, *RS*, *k*) such that there is no <identifier> string *STRB* that is preferred over *STR* and (*STRB*, *RS*, *k*) is a candidate row pattern match.

11) Let *SM* be the set of all preferred matches.

12) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *SM* as *SET OF MATCHES*.

Conformance Rules

None.

9.42 Parsing JSON text

Function

Convert a JSON text to an SQL/JSON item.

Subclause Signature

```
"Parsing JSON text" [General Rules] (
  Parameter: "JSON TEXT",
  Parameter: "FORMAT OPTION",
  Parameter: "UNIQUENESS CONSTRAINT"
) Returns: "STATUS" and "SQL/JSON ITEM"
```

JSON TEXT — a character string that is the JSON text to be parsed.

FORMAT OPTION — a value indicating how JSON TEXT must be parsed (JSON implies that JSON TEXT is parsed using JSON rules; other content implies using rules other than JSON rules).

UNIQUENESS CONSTRAINT — a value indicating whether or not two parsed JSON members are permitted to have equivalent key values (WITH UNIQUE KEYS specifies that no duplicate values are allowed).

STATUS — an SQLSTATE value that indicates the success or reason for failure of the attempt to parse JSON TEXT.

SQL/JSON ITEM — the SQL/JSON item produced by a successful parse of JSON TEXT.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *JV* be the *JSON TEXT*, let *FO* be the *FORMAT OPTION*, and let *UC* be the *UNIQUENESS CONSTRAINT* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS* and *SQL/JSON ITEM*.

2) Let *ST* be the condition *successful completion (00000)*.

3) Case:

   a) If *FO* is JSON, then:

      i) Case:

         1) If *JV* is a character string, then let *ENC* be the Unicode encoding of *JV*.

         2) Otherwise,

            Case:

A) If *FO* contains an explicit encoding, then let *ENC* be that encoding.

B) Otherwise, let *ENC* be an implementation-defined (IA222) encoding.

NOTE 478 — One way an SQL-implementation could determine the encoding is using the scheme specified in Section 3, "Encoding", in RFC 4627.

ii) *JV* is parsed according to the grammar of Section 2, "JSON grammar", in RFC 8259, using the encoding *ENC*.

iii) Case:

1) If *JV* is not a JSON text, then let *ST* be the exception condition: *data exception — invalid JSON text (22032)*.

2) If *UC* is WITH UNIQUE KEYS and *JV* contains a JSON object that has two JSON members whose keys are equivalent, then let *ST* be the exception condition: *data exception — non-unique keys in a JSON object (22037)*.

3) Otherwise:

A) The function *F* transforming a JSON text fragment *J* to an SQL/JSON item is defined recursively according to the grammar of Section 2, "JSON grammar", in RFC 8259, as follows:

I) If *J* is the JSON literal **false**, then *F*(*J*) is the truth value *False*.

II) If *J* is the JSON literal **true**, then *F*(*J*) is the truth value *True*.

III) If *J* is the JSON literal **null**, then *F*(*J*) is the SQL/JSON null value.

IV) If *J* is a JSON number, then *F*(*J*) is the value of the <signed numeric literal> whose characters are identical to *J*.

V) If *J* is a JSON string, then *F*(*J*) is an SQL character string whose character set is Unicode and whose characters are the ones enclosed by quotation marks in *J* after replacing any escape sequences by their unescaped equivalents.

VI) If *J* is a JSON array, then *F*(*J*) is the SQL/JSON array whose elements are obtained by applying the transform *F* to each element of *J* in turn.

VII) If *J* is a JSON member *K*:*V*, then *F*(*J*) is the SQL/JSON member whose key is *F*(*K*) and whose bound value is *F*(*V*).

VIII) If *J* is a JSON object, then:

1) Let $M_1, ..., M_m$ be the members of *J*, enumerated in an implementation-dependent (US039) order. For all *i*, 1 (one) ≤ *i* ≤ *m*, let K_i be the key and let V_i be the bound value of M_i. Let SJM_i be the SQL/JSON member whose key is $F(K_i)$ and whose bound value is $F(V_i)$.

2) It is implementation-dependent (UA057) whether members with redundant duplicate keys are removed. If the implementation-dependent (UA057) choice is to delete members with redundant duplicate keys, then for all *i*, 1 (one) ≤ *i* ≤ *m*, and all *j*, 1 (one) ≤ *j* ≤ *m*, *i* ≠ *j*, if K_i is equivalent to K_j, an implementation-dependent (UA057) choice of M_i or M_j is removed from the list of members of *J*.

3) F(*J*) is the SQL/JSON object whose members are SJM_1, ..., SJM_m.

B) Let *SJI* be *F*(*JV*).

b) Otherwise, let *SJI* be the SQL/JSON item obtained using implementation-defined (IA070) rules for parsing *JV* according to format *FO* and uniqueness constraint *UC*. If there is an error during this conversion, then let *ST* be an implementation-defined (IC015) exception condition.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *ST* as *STATUS* and *SJI* as *SQL/JSON ITEM*.

Conformance Rules

None.

9.43 Serializing an SQL/JSON item

Function

Serialize an SQL/JSON item as a JSON text.

Subclause Signature

```
"Serializing an SQL/JSON item" [Syntax Rules] (
  Parameter: "FORMAT OPTION",
  Parameter: "TARGET TYPE"
)
```

FORMAT OPTION — a value indicating how an SQL/JSON item must be serialized into a character or binary string (JSON implies serialization using JSON rules; other content implies rules other than JSON rules).

TARGET TYPE — the data type specified for the serialized result.

```
"Serializing an SQL/JSON item" [General Rules] (
  Parameter: "SQL/JSON ITEM",
  Parameter: "FORMAT OPTION",
  Parameter: "TARGET TYPE"
) Returns: "STATUS" and "JSON TEXT"
```

SQL/JSON ITEM — an SQL/JSON item to be serialized.

FORMAT OPTION — a value indicating how an SQL/JSON item must be serialized into a character or binary string (JSON implies serialization using JSON rules; other content implies using rules other than JSON rules).

TARGET TYPE — the data type specified for the serialized result.

STATUS — an SQLSTATE value that indicates the success or reason for failure of the attempt to serialize SQL/JSON ITEM.

JSON TEXT — the serialized representation of SQL/JSON ITEM.

Syntax Rules

1) Let *FO* be the *FORMAT OPTION* and let *TT* be the *TARGET TYPE* in an application of the Syntax Rules of this Subclause.

2) Case:

   a) If *FO* contains JSON, then *TT* shall be either a character string type or a binary string type. If *TT* is a character string type, then the character set of *TT* shall be a Universal Character Set.

   b) Otherwise, *TT* shall be an implementation-defined (IV163) data type appropriate to the format identified by *FO*.

Access Rules

None.

General Rules

1) Let *SJI* be the *SQL/JSON ITEM*, let *FO* be the *FORMAT OPTION*, and let *TT* be the *TARGET TYPE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS* and *JSON TEXT*.

2) Case:

   a) If *FO* contains JSON then:

      i) Case:

         1) If *TT* is a character string type, then let *ENC* be the Unicode encoding of *TT*.

         2) If *TT* is a binary string type, then let *ENC* be UTF8, UTF16, or UTF32, as specified in the <JSON representation> contained in *FO*.

      ii) Let *JV* be an implementation-dependent (UV097) value of type *TT* and encoding *ENC* such that these two conditions hold:

         1) *JV* is a JSON text.

         2) When the General Rules of Subclause 9.42, "Parsing JSON text", are applied with *JV* as *JSON TEXT*, *FO* as *FORMAT OPTION*, and WITHOUT UNIQUE KEYS as *UNIQUENESS CONSTRAINT*; let *CST* be the *STATUS* and let *CSJI* be the *SQL/JSON ITEM* returned from the application of those General Rules, *CST* is *successful completion (00000)* and *CSJI* is an SQL/JSON item that is equivalent to *SJI*.

         If there is no such *JV*, then let *ST* be the exception condition: *data exception — invalid JSON text (22032)*.

      iii) If *JV* is longer than the length or maximum length of *TT*, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

   b) Otherwise, let *JV* be an implementation-defined (IV095) value such that, when the General Rules of Subclause 9.42, "Parsing JSON text", are applied with *JV* as *JSON TEXT*, *FO* as *FORMAT OPTION*, and an implementation-defined (IV164) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*; let *CST* be the *STATUS* and let *CSJI* be the *SQL/JSON ITEM* returned from the application of those General Rules, *CST* is *successful completion (00000)* and *CSJI* is an SQL/JSON item that is equivalent to *SJI* according to an implementation-defined (IW149) definition of this equivalence. If there is no such *JV*, then let *ST* be the exception condition *data exception — invalid JSON text (22032)*.

3) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *ST* as *STATUS* and *JV* as *JSON TEXT*.

Conformance Rules

None.

9.44 Converting an SQL/JSON sequence to an SQL/JSON item

Function

Convert an SQL/JSON sequence to an SQL/JSON item.

Subclause Signature

```
"Converting an SQL/JSON sequence to an SQL/JSON item" [General Rules] (
  Parameter: "STATUS IN",
  Parameter: "SQL/JSON SEQUENCE",
  Parameter: "WRAPPER BEHAVIOR",
  Parameter: "EMPTY BEHAVIOR",
  Parameter: "ERROR BEHAVIOR"
) Returns: "STATUS OUT" and "VALUE"
```

STATUS IN — an SQLSTATE value submitted to an invocation of this subclause using this signature.

SQL/JSON SEQUENCE — an SQL/JSON sequence.

WRAPPER BEHAVIOR — an indication of whether the SQL/JSON SEQUENCE is to be "wrapped" in an SQL/JSON array (WITH UNCONDITIONAL ARRAY or WITH CONDITIONAL ARRAY) or not (WITHOUT ARRAY).

EMPTY BEHAVIOR — an indication of the manner in which to behave if the SQL/JSON SEQUENCE is empty (ERROR, EMPTY ARRAY, or EMPTY OBJECT).

ERROR BEHAVIOR — an indication of the manner in which to behave if an exception condition is raised (ERROR, NULL, EMPTY ARRAY, or EMPTY OBJECT).

STATUS OUT — an SQLSTATE value that indicates the success or reason for failure of the conversion attempt.

VALUE — the resulting SQL/JSON item, if successful.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *INST* be the *STATUS IN*, let *SEQ* be the *SQL/JSON SEQUENCE*, let *WRAPPER* be the *WRAPPER BEHAVIOR*, let *ONEMPTY* be the *EMPTY BEHAVIOR*, and let *ONERROR* be the *ERROR BEHAVIOR* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS OUT* and *VALUE*.

2) Except where explicitly specified, the General Rules of this Subclause are not terminated if an exception condition is raised.

3) Let *TEMPST* be *INST*.

4) If *TEMPST* is *successful completion (00000)*, then:

a) Case:

i) If the length of *SEQ* is 0 (zero), then let *WRAPIT* be *False*.

NOTE 479 — This ensures that the ON EMPTY behavior supersedes the WRAPPER behavior.

ii) If *WRAPPER* is WITHOUT ARRAY, then let *WRAPIT* be *False*.

iii) If *WRAPPER* is *WITH* UNCONDITIONAL ARRAY, then let *WRAPIT* be *True*.

iv) If *WRAPPER* is WITH CONDITIONAL ARRAY, then

Case:

1) If *SEQ* has a single SQL/JSON item, then let *WRAPIT* be *False*.

2) Otherwise, let *WRAPIT* be *True*.

b) Case:

i) If *WRAPIT* is *False*, then let *SEQ2* be *SEQ*.

ii) Otherwise, let *SEQ2* be an SQL/JSON sequence with a single SQL/JSON item that is an SQL/JSON array whose elements are the items of *SEQ*, preserving the order of the items in the resulting array.

c) Case:

i) If the length of SEQ2 is 0 (zero), then

Case:

1) If *ONEMPTY* is ERROR, then let *TEMPST* be the exception condition *data exception — no SQL/JSON item (22035)*.

2) If *ONEMPTY* is EMPTY ARRAY, then let *V* be an SQL/JSON array with no SQL/JSON elements.

3) If *ONEMPTY* is EMPTY OBJECT, then let *V* be an SQL/JSON object with no SQL/JSON members.

NOTE 480 — The case when *ONEMPTY* is NULL is handled later in these General Rules.

ii) If the length of *SEQ2* is 1 (one), then let *V* be the only SQL/JSON item in *SEQ2*.

iii) Otherwise, let *TEMPST* be the exception condition *data exception — more than one SQL/JSON item (22034)*.

5) Case:

a) If *TEMPST* is *successful completion (00000)*, then

Case:

i) If the length of *SEQ2* is 0 (zero) and *ONEMPTY* is NULL, then let *JV* be the null value.

ii) Otherwise, let *JV* be *V*.

b) If *TEMPST* is an exception condition, then

Case:

i) If *ONERROR* is ERROR, then let *OUTST* be *TEMPST*.

ii) Otherwise, let *OUTST* be *successful completion (00000)*.

Case:

1) If *ONERROR* is NULL, then let *JV* be the null value.

2) If *ONERROR* is EMPTY ARRAY, then let *JV* be an SQL/JSON array that has no SQL/JSON elements.

3) If *ONERROR* is EMPTY OBJECT, then let *JV* be an SQL/JSON object that has no SQL/JSON members.

6) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *OUTST* as *STATUS OUT* and *JV* as *VALUE*.

Conformance Rules

None.

9.45 SQL/JSON path language: lexical elements

Function

Specify the lexical analysis of the SQL/JSON path language.

Format

```
<SQL/JSON special symbol> ::=
    <alternative not equals operator>
  | <asterisk>
  | <at sign>
  | <comma>
  | <dollar sign>
  | <double ampersand>
  | <double equals>
  | <double vertical bar>
  | <exclamation mark>
  | <greater than operator>
  | <greater than or equals operator>
  | <left bracket>
  | <left paren>
  | <less than operator>
  | <less than or equals operator>
  | <minus sign>
  | <not equals operator>
  | <percent>
  | <period>
  | <plus sign>
  | <question mark>
  | <right bracket>
  | <right paren>
  | <solidus>

<alternative not equals operator> ::=
  != !! <U+0021, U+003D>

<at sign> ::=
  @ !! U+0040

<double ampersand> ::=
  && !! <U+0026, U+0026>

<double equals> ::=
  == !! <U+003D, U+003D>

<double vertical bar> ::=
  || !! <U+007C, U+007C>

<exclamation mark> ::=
  ! !! U+0021

<SQL/JSON key word> ::=
    abs
  | bigint
  | boolean
  | ceiling
  | date
  | datetime
  | decimal
```

```
  | double
  | exists
  | false
  | flag
  | floor
  | integer
  | is
  | keyvalue
  | last
  | lax
  | like_regex
  | null
  | number
  | size
  | starts
  | strict
  | string
  | time
  | time_tz
  | timestamp
  | timestamp_tz
  | to
  | true
  | type
  | unknown
  | with

<JSON path literal> ::=
  !! See the Syntax Rules.

<JSON path string literal> ::=
  !! See the Syntax Rules.

<JSON path numeric literal> ::=
  !! See the Syntax Rules.

<JSON path identifier> ::=
  !! See the Syntax Rules.

<JSON path key name> ::=
  !! See the Syntax Rules.
```

Syntax Rules

1) SQL/JSON path language adopts the conventions in ECMAScript Language Specification 5.1 Edition, section 6, "Source text", regarding the source text of an SQL/JSON path expression, and escape sequences used in that source text.

   NOTE 481 — Thus, an SQL/JSON path expression is assumed to be in Unicode. An SQL-implementation can provide a conversion to Unicode if the SQL/JSON path expression is written in a character string of some other character set; any such conversion is outside the scope of this document.

2) SQL/JSON path language adopts the lexical rules of ECMAScript Language Specification 5.1 Edition, section 5.1.2, "Lexical and RegExp grammars", and section 7, "Lexical conventions", with the following modifications:

   a) Support for any feature relevant to the lexical rules not defined in ECMAScript Language Specification 5.1 Edition, but defined in a later edition, is implementation-defined (IA039).

      NOTE 482 — For example, support for DecimalBigIntegerLiteral as specified in ECMAScript Language Specification 13th Edition, is implementation-defined (IA039).

b) The only goal symbol is *InputElementDiv* (modifies section 7, "Lexical conventions", paragraph 2).

c) There are no comments (modifies section 7.4, "Comments").

d) There are no reserved words (modifies section 7.6.1, "Reserved words").

NOTE 483 — Lexically, the only issue is whether a token that matches an <SQL/JSON key word> is a <JSON path key name> or in fact a key word. This can be decided by observing that a <JSON path key name> cannot be followed by a <left paren>.

e) The following are additional punctuators: @ (modifies section 7.7, "Punctuators").

f) There is no *RegularExpressionLiteral* (modifies section 7.8.5, "Regular expression literals").

NOTE 484 — SQL/JSON path language uses SQL regular expressions in the <JSON like_regex predicate>, not ECMAScript Language Specification 5.1 Edition regular expressions.

g) There is no automatic semicolon insertion (modifies section 7.9, "Automatic semicolon insertion").

NOTE 485 — It follows that SQL/JSON path language is case-sensitive in both identifiers and key words. Unlike SQL, there are no "quoted" identifiers, and there is no automatic conversion of any identifiers to upper-case.

3) SQL/JSON grammar is stated with BNF non-terminals enclosed in angle brackets "< >". The correspondences between SQL/JSON BNF non-terminals and ECMAScript Language Specification 5.1 Edition BNF non-terminals that apply are described in Table 19, "SQL/JSON and ECMAScript correspondences".

Table 19 — SQL/JSON and ECMAScript correspondences

SQL/JSON path language	ECMAScript
<JSON path literal>	*Literal*
<JSON path numeric literal>	*NumericLiteral*
<JSON path string literal>	*StringLiteral*
<JSON path identifier>	*Identifier*

4) A <JSON path identifier> is classified as follows.

Case:

a) A <JSON path identifier> that is a <dollar sign> is a <JSON path context variable>.

b) A <JSON path identifier> that begins with <dollar sign> is a <JSON path named variable>.

c) Otherwise, a <JSON path identifier> is a <JSON path key name>.

Access Rules

None.

General Rules

1) The value of a <JSON path literal> is determined as follows:

a) The value of a <JSON path numeric literal> *JPNL* is the value of the <signed numeric literal> whose characters are identical to *JPNL*.

b) The value of a <JSON path string literal> *JPSL* is an SQL character string whose character set is Unicode and whose characters are the ones enclosed by single or double quotation marks (but excluding these delimiters) in *JPSL* after replacing all escape sequences by their unescaped equivalents.

c) The value of `null` is the SQL/JSON null.

d) The value of `true` is *True*.

e) The value of `false` is *False*.

Conformance Rules

1) Without Feature T840, "Hex integer literals in SQL/JSON path language", in conforming SQL language, there is no HexIntegerLiteral in the SQL/JSON path language.

9.46 SQL/JSON path language: syntax and semantics

Function

Specify the syntax and semantics of SQL/JSON path language.

Subclause Signature

```
"SQL/JSON path language: syntax and semantics" [Syntax Rules] (
  Parameter: "PATH SPECIFICATION",
  Parameter: "PASSING CLAUSE"
)
```

PATH SPECIFICATION — a <JSON path expression>.

PASSING CLAUSE — a <JSON passing clause>.

```
"SQL/JSON path language: syntax and semantics" [General Rules] (
  Parameter: "PATH SPECIFICATION",
  Parameter: "CONTEXT ITEM",
  Parameter: "ALREADY PARSED",
  Parameter: "PASSING CLAUSE"
) Returns: "STATUS" and "SQL/JSON SEQUENCE"
```

PATH SPECIFICATION — a <JSON path expression>.

CONTEXT ITEM — an SQL/JSON context item.

ALREADY PARSED — *True* if the SQL/JSON item identified by CONTEXT ITEM has been parsed; *False* otherwise.

PASSING CLAUSE — a <JSON passing clause>.

STATUS — an SQLSTATE value that indicates the success or reason for failure of the invocation of this subclause using this signature.

SQL/JSON SEQUENCE — an SQL/JSON sequence comprising zero or more SQL/JSON items that is the result of invoking this subclause using this signature.

Format

```
<JSON path expression> ::=
  <JSON path mode> <JSON path wff>

<JSON path mode> ::=
    strict | lax

<JSON path primary> ::=
    <JSON path literal>
  | <JSON path variable>
  | <left paren> <JSON path wff> <right paren>

<JSON path variable> ::=
    <JSON path context variable>
  | <JSON path named variable>
  | <at sign>
  | <JSON last subscript>
```

```
<JSON path context variable> ::=
  <dollar sign>

<JSON path named variable> ::=
  <dollar sign> <JSON path identifier>

<JSON last subscript> ::=
  last

<JSON accessor expression> ::=
    <JSON path primary>
  | <JSON accessor expression> <JSON accessor op>

<JSON accessor op> ::=
    <JSON member accessor>
  | <JSON wildcard member accessor>
  | <JSON array accessor>
  | <JSON wildcard array accessor>
  | <JSON filter expression>
  | <JSON item method>

<JSON member accessor> ::=
    <period> <JSON path key name>
  | <period> <JSON path string literal>
```

NOTE 486 — Unlike ECMAScript Language Specification 5.1 Edition, SQL/JSON path language does not provide a member accessor using brackets that enclose a character string.

```
<JSON wildcard member accessor> ::=
  <period> <asterisk>

<JSON array accessor> ::=
  <left bracket> <JSON subscript list> <right bracket>

<JSON subscript list> ::=
  <JSON subscript> [ { <comma> <JSON subscript> }... ]

<JSON subscript> ::=
    <JSON path wff 1>
  | <JSON path wff 2> to <JSON path wff 3>

<JSON path wff 1> ::=
  <JSON path wff>

<JSON path wff 2> ::=
  <JSON path wff>

<JSON path wff 3> ::=
  <JSON path wff>

<JSON wildcard array accessor> ::=
  <left bracket> <asterisk> <right bracket>

<JSON filter expression> ::=
  <question mark> <left paren> <JSON path predicate> <right paren>
```

NOTE 487 — Unlike ECMAScript Language Specification 5.1 Edition, predicates are not expressions; instead they form a separate language that can only be invoked within a <JSON filter expression>.

```
<JSON item method> ::=
  <period> <JSON method>

<JSON method> ::=
    type <left paren> <right paren>
  | size <left paren> <right paren>
```

```
  | double <left paren> <right paren>
  | ceiling <left paren> <right paren>
  | floor <left paren> <right paren>
  | abs <left paren> <right paren>
  | datetime <left paren> [ <JSON datetime template> ] <right paren>
  | keyvalue <left paren> <right paren>
  | bigint <left paren> <right paren>
  | boolean <left paren> <right paren>
  | date <left paren> <right paren>
  | decimal <left paren> [ <precision> [ <comma> <scale> ] ] <right paren>
  | integer <left paren> <right paren>
  | number <left paren> <right paren>
  | string <left paren> <right paren>
  | time <left paren> [ <time precision> ] <right paren>
  | time_tz <left paren> [ <time precision> ] <right paren>
  | timestamp <left paren> [ <timestamp precision> ] <right paren>
  | timestamp_tz <left paren> [ <timestamp precision> ] <right paren>

<JSON datetime template> ::=
  <JSON path string literal>

<JSON unary expression> ::=
    <JSON accessor expression>
  | <plus sign> <JSON unary expression>
  | <minus sign> <JSON unary expression>

<JSON multiplicative expression> ::=
    <JSON unary expression>
  | <JSON multiplicative expression> <asterisk> <JSON unary expression>
  | <JSON multiplicative expression> <solidus> <JSON unary expression>
  | <JSON multiplicative expression> <percent> <JSON unary expression>

<JSON additive expression> ::=
    <JSON multiplicative expression>
  | <JSON additive expression> <plus sign> <JSON multiplicative expression>
  | <JSON additive expression> <minus sign> <JSON multiplicative expression>

<JSON path wff> ::=
  <JSON additive expression>
```

NOTE 488 — This concludes the main language for JSON path expressions. Next comes the language for predicates, used only in <JSON filter expression>.

```
<JSON predicate primary> ::=
    <JSON delimited predicate>
  | <JSON non-delimited predicate>

<JSON delimited predicate> ::=
    <JSON exists path predicate>
  | <left paren> <JSON path predicate> <right paren>

<JSON non-delimited predicate> ::=
    <JSON comparison predicate>
  | <JSON like_regex predicate>
  | <JSON starts with predicate>
  | <JSON unknown predicate>

<JSON exists path predicate> ::=
  exists <left paren> <JSON path wff> <right paren>

<JSON comparison predicate> ::=
  <JSON path wff> <JSON comp op> <JSON path wff>
```

NOTE 489 — Comparison operators are not left associative, unlike ECMAScript Language Specification 5.1 Edition.

```
<JSON comp op> ::=
    <double equals>
  | <JSON path not equals operator>
  | <less than operator>
  | <greater than operator>
  | <less than or equals operator>
  | <greater than or equals operator>
```

NOTE 490 — Equality operators have the same precedence as inequality comparison operators, unlike ECMAScript Language Specification 5.1 Edition.

```
<JSON path not equals operator> ::=
    <not equals operator>
  | <alternative not equals operator>

<JSON like_regex predicate> ::=
  <JSON path wff> like_regex <JSON like_regex pattern>
      [ flag <JSON like_regex flags> ]

<JSON like_regex pattern> ::=
  <JSON path string literal>

<JSON like_regex flags> ::=
  <JSON path string literal>

<JSON starts with predicate> ::=
  <JSON starts with whole> starts with <JSON starts with initial>

<JSON starts with whole> ::=
  <JSON path wff>

<JSON starts with initial> ::=
    <JSON path string literal>
  | <JSON path named variable>

<JSON unknown predicate> ::=
  <left paren> <JSON path predicate> <right paren> is unknown

<JSON boolean negation> ::=
    <JSON predicate primary>
  | <exclamation mark> <JSON delimited predicate>

<JSON boolean conjunction> ::=
    <JSON boolean negation>
  | <JSON boolean conjunction> <double ampersand> <JSON boolean negation>

<JSON boolean disjunction> ::=
    <JSON boolean conjunction>
  | <JSON boolean disjunction> <double vertical bar> <JSON boolean conjunction>

<JSON path predicate> ::=
  <JSON boolean disjunction>
```


Syntax Rules

1) Let *P* be the *PATH SPECIFICATION* and let *PC* be the *PASSING CLAUSE* in an application of the Syntax Rules of this Subclause.

2) Let *N* be the number of <JSON argument>s contained in *PC*. For *i*, 1 (one) ≤ *i* ≤ *N*, let JA_i be those <JSON argument>s, let VE_i be the <value expression> simply contained in JA_i, and let ID_i be the <identifier> immediately contained in JA_i.

3) *P* shall conform to <JSON path expression>, using the lexical rules specified in Subclause 9.45, "SQL/JSON path language: lexical elements".

4) For every <JSON path named variable> *JPNV* contained in *P*, the <JSON path identifier> contained in *JPNV* shall be equivalent to ID_i for some *i*, 1 (one) ≤ *i* ≤ *N*.

5) A <JSON path primary> that is an <at sign> shall be contained in a <JSON path predicate>.

6) A <JSON last subscript> shall be contained in a <JSON subscript>.

7) If <JSON like_regex predicate> *JLP* is specified, then the value of <JSON like_regex pattern> shall be an XQuery regular expression. If *JLP* simply contains <JSON like_regex flags> *JLF*, then the value of *JLF* shall be an XQuery option flag.

8) If <JSON starts with initial> is a <JSON path named variable> *JPNV*, then *JPNV* shall be equivalent to some ID_i for some *i*, 1 (one) ≤ *i* ≤ *N*, such that the declared type of VE_i is a character string type and JA_i does not have an implicit or explicit <JSON input clause>.

9) *P* shall not be such that an implementation-defined (IA223) syntactic analysis of *P* determines that the evaluation of *P* according to the General Rules of this Subclause would raise an exception condition.

NOTE 491 — For example, if *P* is '`strict $[last+1]`', an exception condition would certainly be raised according to the General Rules. The preceding rule allows the SQL-implementation to treat this as a violation of the Syntax Rules.

10) If <JSON datetime template> *JDT* is specified, then the value of *JDT* shall conform to the lexical grammar of a <datetime template> in the Format of Subclause 9.52, "Datetime templates".

a) If *JDT* contains <datetime template year>, <datetime template rounded year>, <datetime template month>, <datetime template day of month>, or <datetime template day of year>, then JDT is *dated*.

b) If *JDT* contains <datetime template 12-hour>, <datetime template 24-hour>, <datetime template minute>, <datetime template second of minute>, <datetime template second of day>, <datetime template fraction>, or <datetime template am/pm>, then *JDT* is *timed*.

The *fractional seconds precision FSP* of *JDT* is

Case:

i) If *JDT* contains <datetime template fraction> FF1, FF2, FF3, FF4, FF5, FF6, FF7, FF8, or FF9, then 1 (one), 2, 3, 4, 5, 6, 7, 8, or 9, respectively.

ii) Otherwise, 0 (zero).

c) If *JDT* contains <datetime template time zone hour> or <datetime template time zone minute>, then *JDT* is *zoned*.

d) If *JDT* is zoned, then *JDT* shall be timed.

e) *JDT* shall be dated or timed or both.

f) The implicit datetime data type *IDT* of *JDT* is

Case:

i) If *JDT* is dated, timed, and zoned, then TIMESTAMP (*FSP*) WITH TIME ZONE.

ii) If *JDT* is dated, timed, and not zoned, then TIMESTAMP (*FSP*) WITHOUT TIME ZONE.

iii) If *JDT* is timed and zoned, then TIME (*FSP*) WITH TIME ZONE.

iv) If *JDT* is timed and not zoned, then TIME (*FSP*) WITHOUT TIME ZONE.

v) If *JDT* is dated but not timed and not zoned, then DATE.

g) Let *L* be the maximum template length of *JDT* as defined by the Syntax Rules of Subclause 9.50, "Converting a datetime to a formatted character string". Let *CST* be a variable-length character string type of maximum length *L* and character set Unicode.

h) The Syntax Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with *IDT* as *DATETIME TYPE*, *JDT* as *TEMPLATE*, and *CST* as *CHARACTER STRING TYPE*.

11) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

1) Let *P* be the *PATH SPECIFICATION*, let *CI* be the *CONTEXT ITEM*, let *PARSED* be the *ALREADY PARSED*, and let *PC* be the *PASSING CLAUSE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS* and *SQL/JSON SEQUENCE*. .

2) Let *ST* be the completion condition *successful completion (00000)*.

3) Let *JT* be the value of the <value expression> contained in *CI* and let *FO* be the explicit or implicit <JSON input clause> contained in *CI*.

4) If the declared type of *JT* is not JSON and *PARSED* is *False*, then it is implementation-defined (IA050) whether the following rules are applied:

a) The General Rules of Subclause 9.42, "Parsing JSON text", are applied with *JT* as *JSON TEXT*, an implementation-defined (IV185) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*, and *FO* as *FORMAT OPTION*; let *ST* be the *STATUS* and let *CISJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

b) If *ST* is not *successful completion (00000)*, then *ST* is returned as the *STATUS* of this application of these General Rules, and no further General Rules of this Subclause are applied.

5) It is implementation-defined (IA054) whether the following rules are applied:

a) Let *N* be the number of <JSON argument>s contained in *PC*.

b) For all *i*, 1 (one) ≤ *i* ≤ *N*:

i) Let JA_i be the *i*-th <JSON argument> contained in *PC*.

ii) If JA_i contains an implicit or explicit <JSON input clause> $JAFO_i$, then let VE_i be the value of the <value expression> contained in JA_i. If the declared type of VE_i is not JSON, then:

1) The General Rules of Subclause 9.42, "Parsing JSON text", are applied with VE_i as *JSON TEXT*, an implementation-defined (IV185) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*, and $JAFO_i$ as *FORMAT OPTION*; let VV_i be the *STATUS* and let ST_i be the *SQL/JSON ITEM* returned from the application of those General Rules.

2) If ST_i is not *successful completion (00000)*, then ST_i is returned as the *STATUS* of these General Rules, and no further General Rules of this Subclause are applied.

6) Let *MODE* be the <JSON path mode> contained in *P*, and let *WFF* be the <JSON path wff> contained in *P*.

7) Let the function *Wrap*(*SEQ*), where *SEQ* is some SQL/JSON sequence, be defined as follows:

a) Let *n* be the number of SQL/JSON items in *SEQ*.

b) For each *j*, 1 (one) ≤ *j* ≤ *n*, let I_j be the *j*-th SQL/JSON item in *SEQ*.

Case:

i) If I_j is an SQL/JSON array, then let $I2_j$ be I_j.

ii) Otherwise, let $I2_j$ be an SQL/JSON array whose only element is I_j.

c) The result of *Wrap*(*SEQ*) is the SQL/JSON sequence $I2_1$, ..., $I2_n$.

NOTE 492 — Thus, *Wrap*() wraps any non-SQL/JSON array in an SQL/JSON array. This function is used in lax mode for certain operations that require an array as input.

8) Let the function *Unwrap*(*SEQ*), where *SEQ* is some SQL/JSON sequence, be defined as follows:

a) Let *n* be the number of SQL/JSON items in *SEQ*.

b) For each *j*, 1 (one) ≤ *j* ≤ *n*, let I_j be the *j*-th SQL/JSON item in *SEQ*.

Case:

i) If I_j is an SQL/JSON array, then let *m* be the number of elements of I_j and let E_k, 1 (one) ≤ *k* ≤ *m*, be the *k*-th element of I_j in order. Let $I2_j$ be the SQL/JSON sequence E_1, ..., E_m.

ii) Otherwise, let $I2_j$ be I_j.

c) The result of *Unwrap*(*SEQ*) is the SQL/JSON sequence $I2_1$, ..., $I2_n$.

NOTE 493 — Thus, *Unwrap*() unwraps any non-SQL/JSON array into its elements. This function is used in lax mode for certain operations that require an SQL/JSON sequence of non-SQL/JSON arrays as input.

9) The result of evaluating *P* is determined recursively by the following definitions to evaluate all BNF non-terminals contained in *P*.

10) For every BNF production in the Format of this Subclause of the form

```
<JSON BNF non-terminal 1> ::=
  <JSON BNF non-terminal 2>

<JSON BNF non-terminal 2> ::=
  ...
```

the result of evaluating <JSON BNF non-terminal 1> is the same as the result of evaluating <JSON BNF non-terminal 2>.

11) The result of evaluating a <JSON path wff> is a completion condition, and, if that completion condition is *successful completion (00000)*, then an SQL/JSON sequence. For conciseness, the result will be stated either as an exception condition or as an SQL/JSON sequence (in the latter case, the completion condition *successful completion (00000)* is implicit). Unsuccessful completion conditions are not automatically raised and do not terminate application of the General Rules in this Subclause.

a) If <JSON path context variable> *JPCV* is specified, then

Case:

i) If *PARSED* is *True*, then the result of evaluating *JPCV* is *JT*.

ii) If the declared type of *JT* is JSON, then the result of evaluating *JPCV* is *JT*.

iii) Otherwise:

1) The General Rules of Subclause 9.42, "Parsing JSON text", are applied with *JT* as *JSON TEXT*, an implementation-defined (IV185) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*, and *FO* as *FORMAT OPTION*; let *ST* be the *STATUS* and let *CISJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

2) Case:

A) If *ST* is not *successful completion (00000)*, then the result of evaluating *JPCV* is *ST*.

B) Otherwise, the result of evaluating *JPCV* is *CISJI*.

b) If <JSON path named variable> *JPNV* is specified, then let *JPI* be the <JSON path identifier> contained in *JPNV*, let *JA* be the <JSON argument> contained in *PC* that contains an <identifier> that is equivalent to *JPI*, and let *VE* be the value of the <value expression> simply contained in *JA*.

i) Case:

1) If *JA* contains an implicit or explicit <JSON input clause> *JAFO*, then

Case:

A) If *VE* is the null value, then let *ST* be the completion condition *successful completion (00000)* and let *VV* be the empty SQL/JSON sequence.

B) If the declared type of *VE* is JSON, then let *ST* be the completion condition *successful completion (00000)* and let *VV* be *VE*.

C) Otherwise, the General Rules of Subclause 9.42, "Parsing JSON text", are applied with *VE* as *JSON TEXT*, an implementation-defined (IV185) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*, and *JAFO* as *FORMAT OPTION*; let *ST* be the *STATUS* and let *VV* be the *SQL/JSON ITEM* returned from the application of those General Rules.

2) Otherwise:

A) If the declared type of *VE* is character string with character set Unicode, numeric, Boolean, or datetime type, then let *ST* be the completion condition *successful completion (00000)* and let *VV* be *VE*.

B) Otherwise, let *CDT* be an implementation-defined (IV165) character string type with character set Unicode. Let *VV* be the result of

```
CAST (VE AS CDT)
```

Let *ST* be the completion code that results from this <cast specification>.

ii) Case:

1) If *ST* is not *successful completion (00000)*, then the result of *JPNV* is *ST*.

2) If *VV* is a null value, then the result of *JPNV* is the SQL/JSON null.

3) Otherwise, the result of *JPNV* is *VV*.

c) If <at sign> *AS* is specified, then let *JFE* be the innermost <JSON filter expression> containing *AS*. The value of *AS* is the current SQL/JSON item in the evaluation of *JFE*.

d) If <JSON last subscript> *LAST* is specified, then let *JAA* be the innermost <JSON array accessor> that contains *LAST*. Let *JAE* be the <JSON accessor expression> that is the prefixed argument to *JAA*. Let *JAER* be the result of evaluating *JAE*. *JAER* is an SQL/JSON sequence of arrays (any errors are already handled by the General Rules relevant to *JAE*). Let *ARR* be the current SQL/JSON array in *JAER*. Let *N* be the number of elements in *ARR*. The result of *LAST* is *N*−1.

NOTE 494 — Subscripts are 0-relative; therefore, the last subscript of an array of size *N* is *N*−1.

e) The result of evaluating a <JSON path literal> is the value of the <JSON path literal>, as specified in Subclause 9.45, "SQL/JSON path language: lexical elements".

f) The result of evaluating <left paren> <JSON path wff> <right paren> is the result of evaluating the <JSON path wff>.

g) If a <JSON accessor expression> *JAE* that contains a <JSON accessor op> *JAOP* is specified, then let *BASE* be the result of the <JSON accessor expression> contained in *JAE*.

Case:

i) If *BASE* is an exception condition, then *BASE* is the result of evaluating *JAE*.

ii) Otherwise, let *ST* be the condition *successful completion (00000)*.

Case:

1) If *JAOP* is <JSON member accessor>, then:

A) Case:

I) If *JAOP* contains <JSON path identifier> *JPI*, then let *JPSL* be a Unicode character string whose value is the character string that composes *JPI*.

II) Otherwise, let *JPSL* be the value of <JSON path string literal>.

B) Case:

I) If *MODE* is `lax`, then let *BASE* be *Unwrap*(*BASE*).

II) If *MODE* is `strict`, and if any SQL/JSON item in *BASE* is not an SQL/JSON object that contains a member whose key is equivalent to *JPSL*, then let *ST* be the condition *data exception — SQL/JSON member not found (2203A)*.

C) If any SQL/JSON object in *BASE* has two SQL/JSON members with equivalent keys, then it is implementation-defined (IA071) whether *ST* is set to the condition *data exception — non-unique keys in a JSON object (22037)*.

D) If *ST* is successful completion, then:

I) Let *n* be the number of SQL/JSON items in *BASE*.

II) For all *j*, 1 (one) ≤ *j* ≤ *n*,

1) Let I_j be the *j*-th SQL/JSON item in *BASE*.

2) Case:

a) If I_j is an SQL/JSON object containing a member M whose key is equivalent to *JPSL*, then let V_j be the bound value of M. If there is more than one member whose key is equivalent to *JPSL*, then it is implementation-dependent (UA075) which one is chosen.

b) Otherwise, let V_j be the empty SQL/JSON sequence.

E) The result of *JAE* is

Case:

I) If *ST* is an exception condition, then *ST*.

II) Otherwise, the SQL/JSON sequence that is the concatenation of V_1, ..., V_n, in order.

2) If *JAOP* is <JSON wildcard member accessor>, then:

A) Case:

I) If *MODE* is `lax`, then let *BASE* be *Unwrap*(*BASE*).

II) If *MODE* is `strict`, and if any SQL/JSON item in *BASE* is not an SQL/JSON object, then let *ST* be the condition *data exception — SQL/JSON object not found (2203C)*.

B) If any SQL/JSON object in *BASE* has two SQL/JSON members with equivalent keys, then it is implementation-defined (IA071) whether *ST* is set to the condition *data exception — non-unique keys in a JSON object (22037)*.

C) If *ST* is *successful completion (00000)*, then:

I) Let n be the number of SQL/JSON items in *BASE*.

II) For all j, 1 (one) $\le j \le n$,

1) Let I_j be the j-th SQL/JSON item in *BASE*.

2) Case:

a) If I_j is an SQL/JSON object, then:

i) Let m be the number of SQL/JSON members of I_j.

ii) For all i, 1 (one) $\le i \le m$, let M_i be the i-th SQL/JSON member of I_j, taken in an implementation-dependent (US040) order. Let K_i be the key and let V_i be the bound value of M_i.

iii) The following rule is performed an implementation-dependent (UA058) number of times:

If any pair of keys K_a and K_b are equivalent, where $a \ne b$, then one of M_a or M_b is deleted from the list of members of I_j.

iv) Let q be the number of remaining SQL/JSON members of I_j after the duplicate elimination

described in the preceding rule. Let BV_h, 1 (one) ≤ h ≤ q, be the h-th bound value of the remaining SQL/JSON members of I_j, in an implementation-dependent (US040) order. Let V_j be the SQL/JSON sequence BV_1, ..., BV_q.

b) Otherwise, let V_j be the empty SQL/JSON sequence.

D) The result of *JAE* is

Case:

I) If *ST* is an exception condition, then ST.

II) Otherwise, the SQL/JSON sequence that is the concatenation of V_1, ..., V_n, in order.

NOTE 495 — The order of members within an SQL/JSON object is implementation-dependent, but the order of V_1, ..., V_n is not.

3) If *JAOP* is <JSON array accessor>, then:

A) If *MODE* is `lax`, then let *BASE* be *Wrap*(*BASE*).

B) If any SQL/JSON item in *BASE* is not an SQL/JSON array, then let *ST* be the condition *data exception — SQL/JSON array not found (22039)*.

C) Let *JSL* be the <JSON subscript list> simply contained in *JAOP*. Let s be the number of <JSON subscript>s simply contained in *JSL*.

D) For all i, 1 (one) ≤ i ≤ s,

I) Let JS_i be the i-th <JSON subscript> simply contained in *JSL*.

II) Case:

1) If JS_i is <JSON path wff 1>, then let $JSFROM_i$ be JS_i and let $JSTO_i$ be JS_i.

2) Otherwise, let $JSFROM_i$ be the <JSON path wff 2> contained in JS_i and let $JSTO_i$ be the <JSON path wff 3> contained in JS_i.

E) If *ST* is *successful completion (00000)*, then let n be the number of SQL/JSON arrays in *BASE*.

For all j, 1(one) ≤ j ≤ n and all i, 1(one) ≤ i ≤ s,

I) Let I_j be the j-th SQL/JSON array in *BASE*.

II) Let $RJSFROM_i$ be the result of evaluating $JSFROM_i$ and let $RJSTO_i$ be the result of evaluating $JSTO_i$ with I_j as the current SQL/JSON array.

NOTE 496 — The current SQL/JSON array determines the value of `last` (<JSON last subscript>).

III) If $TJSFROM_i$ is not a singleton numeric value or $TJSTO_i$ is not a singleton numeric value, then let *ST* be the exception condition *data exception — invalid SQL/JSON subscript (22033)*.

F) If *ST* is *successful completion (00000)*, then for all j, 1 (one) ≤ j ≤ n,

I) Let N_j be the number of elements of I_j.

II) For all i, 1 (one) ≤ i ≤ s,

1) Let $TJSFROM_i$ be the result of implementation-defined (IA073) truncation or rounding of $RJSFROM_i$ and let $TJSTO_i$ be the result of implementation-defined (IA073) truncation or rounding of $RJSTO_i$.

2) Each of the following is a *potential error*: $TJSFROM_i$ is negative, $TJSFROM_i$ is greater than or equal to N_j, $TJSTO_i$ is negative, $TJSTO_i$ is greater than or equal to N_j, or $TJSFROM_i$ is greater than $TJSTO_j$.

Case:

a) If there is a potential error and *MODE* is `strict`, then let *ST* be the condition *data exception — invalid SQL/JSON subscript (22033)*.

b) Otherwise, let RJS_i be the list of all integers between $TJSFROM_i$ and $TJSTO_i$, inclusive, whose values are between 0 (zero) and N_j, inclusive, in ascending order.

III) If *ST* is *successful completion (00000)*, then:

1) Let *RJSU* be the list of integers formed by concatenating RJS_i, 1 (one) ≤ i ≤ s, in that order.

2) Let e be the number of integers in *RJSU*. For each f, 1 (one) ≤ f ≤ e, let $RJSU_f$ be the f-th integer in *RJSU* and let E_f be the element of I_j whose subscript is $RJSU_f$, using 0-relative subscripting.

3) Let V_j be the SQL/JSON sequence E_1, ..., E_e, in order.

G) The result of *JAE* is

Case:

I) If *ST* is an exception condition, then *ST*.

II) Otherwise, the SQL/JSON sequence that is the concatenation of V_1, ..., V_n, in order.

4) If *JAOP* is <JSON wildcard array accessor>, then:

A) If *MODE* is `lax`, then let *BASE* be *Wrap*(*BASE*).

B) If any SQL/JSON item in *BASE* is not an SQL/JSON array, then let *ST* be the condition *data exception — SQL/JSON array not found (22039)*.

C) If *ST* is successful completion, then let n be the number of SQL/JSON items in *BASE*. For all j, 1 (one) ≤ j ≤ n,

I) Let I_j be the j-th SQL/JSON item in *BASE*.

II) Let t be the number of elements of I_j and let E_k, 1 (one) ≤ k ≤ t, be the k-th element of I_j, in order.

III) Let V_j be the SQL/JSON sequence E_1, ..., E_t, in order.

D) The result of *JAE* is

Case:

I) If *ST* is an exception condition, then *ST*.

II) Otherwise, the SQL/JSON sequence that is the concatenation of V_1, ..., V_n, in order.

5) If *JAOP* is <JSON filter expression>, then:

A) If *MODE* is `lax`, then let *BASE* be *Unwrap*(*BASE*).

B) Let *JPP* be the <JSON path predicate> simply contained in *JAOP*.

C) Let n be the number of SQL/JSON items in *BASE*. For all j, 1 (one) $\leq j \leq n$,

I) Let I_j be the j-th SQL/JSON item in *BASE*, in order.

II) Let *TV* be the result of evaluating *JPP* with I_j as the current SQL/JSON item of *JPP*.

NOTE 497 — The current SQL/JSON item of *JPP* determines the value of <at sign> contained in *JPP*.

III) Case:

1) If *TV* is *True*, then let V_j be I_j.

2) Otherwise, let V_j be the empty SQL/JSON sequence.

D) The result of *JAE* is the SQL/JSON sequence that is the concatenation of V_1, ..., V_n, in order.

6) If *JAOP* is <JSON item method>, then let *JM* be the <JSON method>.

A) If *MODE* is `lax` and <JSON method> is not `type` or `size`, then let *BASE* be *Unwrap*(*BASE*).

B) Let n be the number of SQL/JSON items in *BASE*. Let I_j, 1 (one) $\leq j \leq n$, be the j-th SQL/JSON item in *BASE*, in order.

C) Case:

I) If *JM* specifies `type`, then:

1) For all j, 1 (one) $\leq j \leq n$, let V_j be

Case:

a) If I_j is an SQL/JSON null, then the Unicode character string "null".

b) If I_j is numeric, then the Unicode character string "number".

c) If I_j is a character string, then the Unicode character string "string".

d) If I_j is a Boolean, then the Unicode character string "boolean".

e) If I_j is a date, then the Unicode character string "date".

f) If I_j is a time without time zone, then the Unicode character string "time without time zone".

g) If I_j is a time with time zone, then the Unicode character string "time with time zone".

h) If I_j is a timestamp without time zone, then the Unicode character string "timestamp without time zone".

i) If I_j is a timestamp with time zone, then the Unicode character string "timestamp with time zone".

j) If I_j is array, then the Unicode character string "array".

k) If I_j is object, then the Unicode character string "object".

2) The result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

II) If *JM* specifies `size`, then

Case:

1) If *MODE* is `strict` and any V_j is not an array, then the result of *JAE* is the exception condition *data exception — SQL/JSON array not found (22039)*.

2) Otherwise:

a) For all j, 1 (one) $\leq j \leq n$, let V_j be

Case:

i) If I_j is an SQL/JSON array, then the number of SQL/JSON elements of I_j.

ii) Otherwise, 1 (one).

b) The result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

III) If *JM* specifies `double`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
CAST (X AS DOUBLE PRECISION)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

IV) If *JM* specifies `ceiling`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a number, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
CEILING (X)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

V) If *JM* specifies `floor`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a number, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
FLOOR (X)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

VI) If *JM* specifies `abs`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a number, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
ABS (X)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

VII) If *JM* specifies `datetime`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a character string, then let ST_j be the exception condition *data exception — invalid argument for SQL/JSON datetime function (22031)*.

b) If *JM* specifies <JSON datetime template> *JDT*, then let *IDT* be the implicit datetime data type of *JDT*. The General Rules of Subclause 9.51, "Converting a formatted character string to a datetime", are applied with *IDT* as *DATETIME TYPE*, *JDT* as *TEMPLATE*, and I_j as *FORMATTED CHARACTER STRING*; let *DV* be the *DATETIME VALUE* returned from the application of those General Rules. If the evaluation of those General Rules would raise an exception, then that exception is not raised; instead, let ST_j be the exception condition *data exception — invalid argument for SQL/JSON datetime function (22031)*; otherwise, let ST_j be *DV*.

c) Otherwise:

i) If the rules for <unquoted date string> in Subclause 5.3, "<literal>", can be applied to I_j to determine a valid value of the data type DATE, then let V_j be that value.

ii) If the rules for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to I_j to determine a valid value of the data type TIME WITH TIME ZONE, then let V_j be that value.

iii) If the rules for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to I_j to

determine a valid value of the data type TIME WITHOUT TIME ZONE, then let V_j be that value.

iv) If the rules for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to I_j to determine a valid value of the data type TIMESTAMP WITH TIME ZONE, then let V_j be that value.

v) If the rules for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to I_j to determine a valid value of the data type TIMESTAMP WITHOUT TIME ZONE, then let V_j be that value.

vi) Otherwise, let ST_j be the exception condition *data exception — invalid argument for SQL/JSON datetime function (22031)*.

2) Case:

a) If any ST_j is not *successful completion (00000)*, then the result of *JAE* is ST_j. If more than one ST_j is an exception condition, then it is implementation-dependent (UA070) which ST_j is raised.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

VIII) If *JM* specifies `keyvalue`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not an SQL/JSON object, then let ST_j be the exception condition *data exception — SQL/JSON object not found (2203C)*.

b) If I_j has two or more SQL/JSON members with the same key, then it is implementation-defined (IA072) whether *ST* is set to the condition *data exception — non-unique keys in a JSON object (22037)*.

c) Otherwise:

i) Let ID_j be an implementation-defined exact numeric value of scale 0 (zero) that uniquely identifies I_j.

ii) Let m be the number of SQL/JSON members of I_j.

iii) For all i, 1 (one) $\leq i \leq m$, let M_i be the i-th SQL/JSON member of I_j, taken in an implementation-dependent (US040) order, let K_i be the key and let V_i be the bound value of M_i.

iv) The following rule is performed an implementation-dependent (UA058) number of times:

If any pair of keys K_a and K_b are equal, where $a \neq b$, then one of M_a or M_b is deleted from the list of members of I_j.

v) Let q be the number of remaining SQL/JSON members of I_j after the duplicate elimination described in the preceding rule.

vi) For all h, 1 (one) $\leq h \leq q$, let OBJ_h be an SQL/JSON object with three members:

1) The first member has key "key" and bound value K_i.

2) The second member has key "value" and bound value BV_i.

3) The third member has key "id" and bound value ID_j.

vii) Let V_j be the SQL/JSON sequence OBJ_1, ..., OBJ_q.

2) Case:

a) If any *ST* is not *successful completion (00000)*, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

IX) If *JM* specifies `bigint`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
CAST (X AS BIGINT)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

X) If *JM* specifies `boolean`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a Boolean value or character string, then let *ST* be *data exception — non-Boolean SQL/JSON item (2202V)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
CAST (X AS BOOLEAN)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XI) If *JM* specifies `date`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a character string, then let *ST* be *data exception — non-date SQL/JSON item (2202W)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
CAST (X AS DATE)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XII) If *JM* specifies `decimal`, then:

1) For all j, 1 (one) $\leq j \leq n$,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j.

i) Case:

1) If <precision> is not specified, then let *PS* be a zero-length string.

2) Otherwise, let *P* be <precision>.

Case:

A) If <scale> *S* is specified, then let *PS* be

`(P, S)`

B) Otherwise, let *PS* be

`(P)`

ii) Let V_j be the result of

```
CAST (X AS DECIMAL PS)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XIII) If *JM* specifies `integer`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let V_j be the result of

```
CAST (X AS INTEGER)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XIV) If *JM* specifies `number`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-numeric SQL/JSON item (22036)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let *NT* be an implementation-defined (IV186) numeric type. Let V_j be the result of

```
CAST (X AS NT)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XV) If *JM* specifies `string`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a character string, number, or Boolean value, then let *ST* be *data exception — non-string SQL/JSON item (2202X)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j. Let *ML* be an implementation-defined (IL006) maximum length of variable-length character strings. Let V_j be the result of

```
CAST (X AS CHARACTER VARYING(ML)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XVI) If *JM* specifies `time`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a character string, then let *ST* be *data exception — non-time SQL/JSON item (2202Y)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j.

i) Case:

1) If <time precision> is not specified, then let *PS* be a zero-length string.

2) Otherwise, let *P* be <time precision> and let *PS* be:

```
(P)
```

ii) Let V_j be the result of

```
CAST (X AS TIME PS)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XVII) If *JM* specifies `time_tz`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-time SQL/JSON item (2202Y)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j.

i) Case:

1) If <time precision> is not specified, then let *PS* be a zero-length string.

2) Otherwise, let *P* be <time precision> and let *PS* be:

```
(P)
```

ii) Let V_j be the result of

```
CAST (X AS TIME PS WITH TIME ZONE)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XVIII) If *JM* specifies `timestamp`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a character string, then let *ST* be *data exception — non-timestamp SQL/JSON item (2202Z)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j.

i) Case:

1) If <timestamp precision> is not specified, then let *PS* be a zero-length string.

2) Otherwise, let *P* be <timestamp precision> and let *PS* be:

```
(P)
```

ii) Let V_j be the result of

```
CAST (X AS TIMESTAMP PS)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

XIX) If *JM* specifies `timestamp_tz`, then:

1) For all *j*, 1 (one) ≤ *j* ≤ *n*,

Case:

a) If I_j is not a number or character string, then let *ST* be *data exception — non-timestamp SQL/JSON item (2202Z)*.

b) Otherwise, let *X* be an SQL variable whose value is I_j.

i) Case:

1) If <timestamp precision> is not specified, then let *PS* be a zero-length string.

2) Otherwise, let *P* be <timestamp precision> and let *PS* be:

```
(P)
```

ii) Let V_j be the result of

```
CAST (X AS TIMESTAMP PS WITH TIME ZONE)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not successful completion, then the result of *JAE* is *ST*.

b) Otherwise, the result of *JAE* is the SQL/JSON sequence V_1, ..., V_n.

h) If <JSON unary expression> *JUE* simply contains <plus sign> or <minus sign>, then let *BASE* be the result of evaluating the <JSON unary expression> simply contained in *JUE*. If *MODE* is `lax`, then let *BASE* be *Unwrap*(*BASE*).

Case:

i) If *BASE* is an exception condition, then *BASE* is the result of evaluating *JUE*.

ii) If any item in *BASE* is not a number, then the result of evaluating *JUE* is the exception condition *data exception — SQL/JSON number not found (2203B)*.

iii) Otherwise, let *ST* be the condition *successful completion (00000)*.

Case:

1) If *JUE* is <plus sign>, then the result of *JUE* is *BASE*.

2) Otherwise, let *n* be the number of SQL/JSON items contained in *BASE*

A) For all *j*, 1 (one) ≤ *j* ≤ *n*,

I) Let I_j be the *j*-th SQL/JSON item contained in *BASE*.

II) Let X_j be an SQL variable whose value is I_j. Let V_j be the result of

$-X_j$

If an exception condition is raised when evaluating $-X_j$, then let *ST* be that exception condition.

B) If *ST* is an exception condition, then the result of *JUE* is *ST*. Otherwise, the result of *JUE* is the SQL/JSON sequence V_1, ..., V_j.

i) If <JSON multiplicative expression> *JME* simply contains <asterisk>, <solidus>, or <percent>, then:

i) Let *OP1* be the result of the <JSON multiplicative expression> simply contained in *JME* and let *OP2* be the <JSON unary expression> simply contained in *JME*.

ii) If *MODE* is `lax`, then let *OP1* be *Unwrap*(*OP1*) and let *OP2* be *Unwrap*(*OP2*).

iii) Case:

1) If either *OP1* or *OP2* is not a singleton numeric value, then let *ST* be the exception condition *data exception — singleton SQL/JSON item required (22038)*.

2) Otherwise, let *X* be an SQL variable whose value is *OP1* and let *Y* be an SQL variable whose value is *OP2*.

Case:

A) If *JME* simply contains <asterisk>, then let *Z* be the result of the <numeric value expression>

X * *Y*

If this calculation raises an exception condition, then let *ST* be that exception condition.

B) If *JME* simply contains <solidus>, then let *Z* be the result of the <numeric value expression>

```
X / Y
```

If this calculation raises an exception condition, then let *ST* be that exception condition.

C) If *JME* simply contains <percent>, then let *Z* be the result of the <numeric value expression>

```
MOD (X, Y)
```

If this calculation raises an exception condition, then let *ST* be that exception condition.

iv) Case:

1) If *ST* is an exception condition, then the result of *JME* is *ST*.

2) Otherwise, the result of *JME* is *Z*.

j) If <JSON additive expression> *JAE* simply contains <plus sign> or <minus sign>, then:

i) Let *OP1* be the result of the <JSON additive expression> simply contained in *JAE* and let *OP2* be the <JSON multiplicative expression> simply contained in *JAE*.

ii) Case:

1) If either *OP1* or *OP2* is not a singleton numeric value, then let *ST* be the exception condition *data exception — singleton SQL/JSON item required (22038)*.

2) Otherwise, let *X* be an SQL variable whose value is *OP1* and let *Y* be an SQL variable whose value is *OP2*.

Case:

A) If *JAE* simply contains <plus sign>, then let *Z* be the result of the <numeric value expression>

```
X + Y
```

If this calculation raises an exception condition, then let *ST* be that exception condition.

B) If *JAE* simply contains <minus sign>, then let *Z* be the result of the <numeric value expression>

```
X - Y
```

If this calculation raises an exception condition, then let *ST* be that exception condition.

iii) Case:

1) If *ST* is an exception condition, then the result of *JAE* is *ST*.

2) Otherwise, the result of *JAE* is *Z*.

12) The result of evaluating a <JSON path predicate> is a truth value *True*, *False*, or *Unknown*.

a) If <JSON exists path predicate> *JEP* is specified, then:

i) Let *WFF* be the result of evaluating the <JSON path wff> simply contained in *JEP*.

ii) Case:

1) If *WFF* is an exception condition, then the result of *JEP* is *Unknown*.

2) If *WFF* is an empty SQL/JSON sequence, then the result of *JEP* is *False*.

3) Otherwise, the result of *JEP* is *True*.

b) If <JSON delimited predicate> *JDP* specifies <left paren> <JSON path predicate> <right paren>, then the result of evaluating *JDP* is the result of evaluating the <JSON path predicate> simply contained in *JDP*.

c) If <JSON comparison predicate> *JCP* is specified, then:

i) Let *JCO* be the <JSON comp op> simply contained in *JCP*.

ii) Let *A* be the result of evaluating the first <JSON path wff> contained in *JCP* and let *B* be the result of evaluating the second <JSON path wff> contained in *JCP*.

iii) Case:

1) If either *A* or *B* is an exception condition, then the result of *JCP* is *Unknown*.

2) Otherwise:

A) If *MODE* is `lax`, then let *A* be *Unwrap*(*A*) and let *B* be *Unwrap*(*B*).

B) Let *ERR* be initially *False* and let *FOUND* be initially *False*.

C) Let *n* be the number of SQL/JSON items in *A* and let *m* be the number of SQL/JSON items in *B*. Let AI_i, 1 (one) ≤ *i* ≤ *n*, be the *i*-th SQL/JSON item in *A*. Let BI_j, 1 (one) ≤ *j* ≤ *m*, be the *j*-th SQL/JSON item in *B*.

D) For all *i*, 1 (one) ≤ *i* ≤ *n*, and all *j*, 1 (one) ≤ *j* ≤ *m*, AI_i is compared to BI_j using *JCO*, according to the following rules. When comparing character strings, the Unicode codepoint collation is used.

Case:

I) If *JCO* is <double equals> or <JSON path not equals operator>, and AI_i and BI_j are not equality-comparable, then let *ERR* be *True*.

II) If *JCO* is <less than operator>, <greater than operator>, <less than or equals operator>, or <greater than or equals operator>, and AI_i and BI_j are not order-comparable, then let *ERR* be *True*.

NOTE 498 — Otherwise, AI_i and BI_j are either the SQL/JSON null or comparable SQL/JSON scalars.

III) If *JCO* is <double equals>, then

Case:

1) If AI_i and BI_j are both the SQL/JSON null value, then let *FOUND* be *True*.

2) If AI_i and BI_j are equal SQL/JSON scalars, then let *FOUND* be *True*.

IV) If *JCO* is <JSON path not equals operator>, then

Case:

1) If AI_i is the SQL/JSON null value, and BI_j is not the SQL/JSON null value, then let *FOUND* be *True*.

2) If BI_j is the SQL/JSON null value, and AI_i is not the SQL/JSON null value, then let *FOUND* be *True*.

3) If AI_i and BI_j are unequal SQL/JSON scalars, then let *FOUND* be *True*.

V) If *JCO* is <less than operator> and AI_i and BI_j are SQL/JSON scalars and AI_i is less than BI_j, then let *FOUND* be *True*.

VI) If *JCO* is <greater than operator> and AI_i and BI_j are SQL/JSON scalars and AI_i is greater than BI_j, then let *FOUND* be *True*.

VII) If *JCO* is <less than or equals operator>, then

Case:

1) If AI_i and BI_j are both the SQL/JSON null value, then let *FOUND* be *True*.

2) If AI_i and BI_j are SQL/JSON scalars and AI_i is less than or equal to BI_j, then let *FOUND* be *True*.

VIII) If *JCO* is <greater than or equals operator>, then

Case:

1) If AI_i and BI_j are both the SQL/JSON null value, then let *FOUND* be *True*.

2) If AI_i and BI_j are SQL/JSON scalars and AI_i is greater than or equal to BI_j, then let *FOUND* be *True*.

E) Case:

I) If *MODE* is `strict` and *ERR* is *True*, then the result of *JCP* is *Unknown*.

II) If *ERR* is *True* and *FOUND* is *True*, then it is implementation-dependent (UA073) whether the result of *JCP* is *True* or *Unknown*.

NOTE 499 — This means that in lax mode the SQL-implementation can stop on the first error or the first success, in an implementation-dependent order of evaluation.

III) If *FOUND* is *True*, then the result of *JCP* is *True*.

IV) If *ERR* is *True*, then the result of *JCP* is *Unknown*.

V) Otherwise, the result of *JCP* is *False*.

d) If <JSON like_regex predicate> *JLP* is specified, then:

i) Let *SEQ* be the result of evaluating the <JSON path wff> simply contained in *JLP*.

ii) Case:

1) If *SEQ* is an exception condition, then the result of *JLP* is *Unknown*.

2) Otherwise:

A) If *MODE* is `lax`, then let *SEQ* be *Unwrap*(*SEQ*).

B) Let *PATTERN* be a <character string literal> whose value is the value of the <JSON like_regex pattern> simply contained in *JLP*.

C) If <JSON like_regex flags> *JLF* is specified, then let *FLAGS* be a <character string literal> whose value is the value of *JLF*; otherwise, let *FLAGS* be the zero-length character string.

D) Let *ERR* be initially *False* and let *FOUND* be initially *False*.

E) Let *n* be the number of SQL/JSON items in *SEQ*. For all *j*, 1 (one) ≤ *j* ≤ *n*, let I_j be the *j*-th SQL/JSON item in *SEQ*.

Case:

I) If I_j is not a character string, then let *ERR* be *True*.

II) Otherwise, let X_j be an SQL variable whose value is I_j and let TV_j be the result of evaluating

X_j `LIKE_REGEX` *PATTERN* `FLAG` *FLAGS*

If TV_j is *True*, then let *FOUND* be *True*.

NOTE 500 — The LIKE_REGEX predicate cannot raise an exception because the Syntax Rules require that *PATTERN* is an XQuery regular expression and *FLAGS* is an XQuery option flag.

F) Case:

I) If *MODE* is `strict` and *ERR* is *True*, then the result of *JLP* is *Unknown*.

II) If *ERR* is *True* and *FOUND* is *True*, then it is implementation-dependent (UA073) whether the result of *JLP* is *True* or *Unknown*.

III) If *FOUND* is *True*, then the result of *JLP* is *True*.

IV) If *ERR* is *True*, then the result of *JLP* is *Unknown*.

V) Otherwise, the result of *JLP* is *False*.

e) If <JSON starts with predicate> *JSWP* is specified, then:

i) Let *A* be the result of evaluating the <JSON starts with whole> simply contained in *JSWP*, and let *B* be the result of evaluating the <JSON starts with initial> simply contained in *JSWP*.

ii) Case:

1) If either *A* or *B* is an exception condition, then the result of *JSWP* is *Unknown*.

2) If *B* is not a character string, then the result of *JSWP* is *Unknown*.

3) Otherwise:

A) If *MODE* is `lax`, then let *A* with *Unwrap*(*A*).

B) Let *ERR* be initially *False* and let *FOUND* be initially *False*.

C) Let *n* be the number of SQL/JSON items in *A*. For all *i*, 1 (one) ≤ *i* ≤ *n*, let AI_i be the *i*-th SQL/JSON item in *A*.

Case:

I) If AI_i is not a character string, then let *ERR* be *True*.

II) If *B* is an initial substring of AI_i, then let *FOUND* be *True*.

D) Case:

I) If *MODE* is `strict` and *ERR* is *True*, then the result of *JSWP* is *Unknown*.

II) If *ERR* is *True* and *FOUND* is *True*, then it is implementation-dependent (UA073) whether the result of *JSWP* is *True* or *Unknown*.

III) If *FOUND* is *True*, then the result of *JSWP* is *True*.

IV) If *ERR* is *True*, then the result of *JSWP* is *Unknown*.

V) Otherwise, the result of *JSWP* is *False*.

f) If <JSON unknown predicate> *JUP* is specified, then:

i) Let *JPP* be the result of evaluating the <JSON path predicate> simply contained in *JUP*.

ii) Case:

1) If *JPP* is *Unknown*, then the result of *JUP* is *True*.

2) Otherwise, then the result of *JUP* is *False*

g) If <JSON boolean negation> *JBN* simply contains <exclamation mark>, then:

i) Let *JDP* be the result of evaluating the <JSON delimited predicate> simply contained in *JBN*.

ii) The result of evaluating *JBN* is

Case:

1) If *JDP* is *True*, then *False*.

2) If *JDP* is *False*, then *True*.

3) If *JDP* is *Unknown*, then *Unknown*.

h) If <JSON boolean conjunction> *JBC* simply contains <double ampersand>, then:

i) Let *OP1* be the result of evaluating the <JSON boolean conjunction> simply contained in *JBC*, and let *OP2* be the result of evaluating the <JSON boolean negation> simply contained in *JBC*.

ii) The result of *JBC* is

Case:

1) If either *OP1* or *OP2* is *False*, then *False*.

2) If either *OP1* or *OP2* is *Unknown*, then *Unknown*.

3) Otherwise, *True*.

i) If <JSON boolean disjunction> *JBD* simply contains <double vertical bar>, then:

i) Let *OP1* be the result of evaluating the <JSON boolean disjunction> simply contained in *JBD*, and let *OP2* be the result of evaluating the <JSON boolean conjunction> simply contained in *JBD*.

ii) The result of *JBD* is

Case:

1) If either *OP1* or *OP2* is *True*, then *True*.

2) If either *OP1* or *OP2* is *Unknown*, then *Unknown*.

3) Otherwise, *False*.

13) The result of evaluating *P* according to the preceding General Rules is an ordered list of zero or more SQL/JSON items that is an SQL/JSON sequence. Let *SJS* be that SQL/JSON sequence.

14) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *ST* as *STATUS* and *SJS* as *SQL/JSON SEQUENCE*.

Conformance Rules

1) Without Feature T831, "SQL/JSON path language: strict mode", in conforming SQL language, a <JSON path expression> shall not contain a <JSON path mode> that is `strict`.

2) Without Feature T832, "SQL/JSON path language: item method", in conforming SQL language, a <JSON path expression> shall not contain a <JSON item method>.

3) Without Feature T833, "SQL/JSON path language: multiple subscripts", in conforming SQL language, a <JSON path expression> shall not contain a <JSON subscript list> that contains more than one <JSON subscript>.

4) Without Feature T834, "SQL/JSON path language: wildcard member accessor", in conforming SQL language, a <JSON path expression> shall not contain a <JSON wildcard member accessor>.

5) Without Feature T835, "SQL/JSON path language: filter expressions", in conforming SQL language, a <JSON path expression> shall not contain a <JSON filter expression>.

6) Without Feature T836, "SQL/JSON path language: starts with predicate", in conforming SQL language, a <JSON path expression> shall not contain a <JSON starts with predicate>.

7) Without Feature T837, "SQL/JSON path language: regex_like predicate", in conforming SQL language, a <JSON path expression> shall not contain a <JSON like_regex predicate>.

8) Without Feature T865, "SQL/JSON item method: bigint()", in conforming SQL language, a <JSON item method> shall not specify `bigint`.

9) Without Feature T866, "SQL/JSON item method: boolean()", in conforming SQL language, a <JSON item method> shall not specify `boolean`.

10) Without Feature T867, "SQL/JSON item method: date()", in conforming SQL language, a <JSON item method> shall not specify `date`.

11) Without Feature T868, "SQL/JSON item method: decimal()", in conforming SQL language, a <JSON item method> shall not specify `decimal`.

12) Without Feature T869, "SQL/JSON item method: decimal() with precision and scale", in conforming SQL language, a <JSON item method> shall not specify `decimal` and contain a <precision>.

13) Without Feature T870, "SQL/JSON item method: integer()", in conforming SQL language, a <JSON item method> shall not specify `integer`.

14) Without Feature T871, "SQL/JSON item method: number()", in conforming SQL language, a <JSON item method> shall not specify `number`.

15) Without Feature T872, "SQL/JSON item method: string()", in conforming SQL language, a <JSON item method> shall not specify `string`.

16) Without Feature T873, "SQL/JSON item method: time()", in conforming SQL language, a <JSON item method> shall not specify `time`.

17) Without Feature T874, "SQL/JSON item method: time_tz()", in conforming SQL language, a <JSON item method> shall not specify `time_tz`.

18) Without Feature T875, "SQL/JSON item method: time precision", in conforming SQL language, a <JSON item method> shall not contain a <time precision>.

19) Without Feature T876, "SQL/JSON item method: timestamp()", in conforming SQL language, a <JSON item method> shall not specify `timestamp`.

20) Without Feature T877, "SQL/JSON item method: timestamp_tz()", in conforming SQL language, a <JSON item method> shall not specify `timestamp_tz`.

21) Without Feature T878, "SQL/JSON item method: timestamp precision", in conforming SQL language, a <JSON item method> shall not contain a <timestamp precision>.

9.47 Processing <JSON API common syntax>

Function

Process the inputs to JSON_VALUE, JSON_QUERY, JSON_TABLE, and JSON_EXISTS.

Subclause Signature

```
"Processing <JSON API common syntax>" [General Rules] (
  Parameter: "JSON API COMMON SYNTAX"
) Returns: "STATUS" and "SQL/JSON SEQUENCE"
```

JSON API COMMON SYNTAX — a <JSON API common syntax>.

STATUS — an SQLSTATE value that indicates the success or reason for failure of the invocation of this subclause using this signature.

SQL/JSON SEQUENCE — an SQL/JSON sequence comprising zero or more SQL/JSON items that is the result of invoking this subclause using this signature.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *JACS* be the *JSON API COMMON SYNTAX* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS* and *SQL/JSON SEQUENCE*.

2) Let *P* be the <JSON path specification> simply contained in *JACS*, and let *C* be the <JSON context item> simply contained in *JACS*. If *JACS* simply contains a <JSON passing clause>, then let *PC* be that <JSON passing clause>; otherwise, let *PC* be the zero-length character string.

3) General Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with *P* as *PATH SPECIFICATION*, *C* as *CONTEXT ITEM*, *False* as *ALREADY PARSED*, and *PC* as *PASSING CLAUSE*; let *ST* be the *STATUS* and let *SEQ* be the *SQL/JSON SEQUENCE* returned from the application of those General Rules.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *ST* as *STATUS* and *SEQ* as *SQL/JSON SEQUENCE*.

Conformance Rules

None.

9.48 Casting an SQL/JSON sequence to an SQL type

Function

Cast an SQL/JSON sequence to a <predefined type>.

Subclause Signature

```
"Casting an SQL/JSON sequence to an SQL type" [General Rules] (
  Parameter: "STATUS IN",
  Parameter: "SQL/JSON SEQUENCE",
  Parameter: "EMPTY BEHAVIOR",
  Parameter: "ERROR BEHAVIOR",
  Parameter: "DATA TYPE"
) Returns: "STATUS OUT" and "VALUE"
```

STATUS IN — an SQLSTATE value submitted to an invocation of this subclause using this signature.

SQL/JSON SEQUENCE — an SQL/JSON sequence.

EMPTY BEHAVIOR — an indication of the manner in which to behave if the SQL/JSON SEQUENCE is empty (ERROR, EMPTY ARRAY, or EMPTY OBJECT).

ERROR BEHAVIOR — an indication of the manner in which to behave if an exception condition is raised (ERROR, NULL, EMPTY ARRAY, or EMPTY OBJECT).

DATA TYPE — the data type that is the desired type of the VALUE.

STATUS OUT — an SQLSTATE value that indicates the success or reason for failure of the invocation of this subclause using this signature.

VALUE — the resulting value, if successful.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *INST* be the *STATUS IN*, let *SEQ* be the *SQL/JSON SEQUENCE*, let *ONEMPTY* be the *EMPTY BEHAVIOR*, let *ONERROR* be the *ERROR BEHAVIOR*, and let *DT* be the *DATA TYPE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS OUT* and *VALUE*. .

2) Except where explicitly specified, the General Rules of this Subclause are not terminated if an exception condition is raised.

3) Let *TEMPST* be *INST*.

4) If *TEMPST* is *successful completion (00000)*, then

Case:

a) If the length of *SEQ* is greater than 1 (one), then let *TEMPST* be *data exception — more than one SQL/JSON item (22034)*.

b) If the length of *SEQ* is 1 (one), then let *I* be the SQL/JSON item in *SEQ*.

Case:

i) If *I* is an SQL/JSON array or SQL/JSON object, then let *TEMPST* be *data exception — SQL/JSON scalar required (2203F)*.

ii) If *I* is the SQL/JSON null, then let *V* be the SQL null value.

iii) Otherwise, let *IDT* be the data type of *I*.

Case:

1) If *IDT* cannot be cast to target type *DT* according to the Syntax Rules of Subclause 6.13, "<cast specification>", then let *TEMPST* be *data exception — SQL/JSON item cannot be cast to target type (2203G)*.

2) Otherwise, let *X* be an SQL variable whose value is *I*. Let *V* be the value of

```
CAST (X AS DT)
```

If an exception condition is raised by this <cast specification>, then let *TEMPST* be that exception condition.

c) If the length of *SEQ* is 0 (zero), then

Case:

i) If *ONEMPTY* is ERROR, then let *TEMPST* be *data exception — no SQL/JSON item (22035)*.

ii) If *ONEMPTY* is NULL, then let *V* be the SQL null value.

iii) If *ONEMPTY* immediately contains DEFAULT, then let *VE* be the <value expression> immediately contained in *ONEMPTY*. Let *V* be the value of

```
CAST (VE AS DT>)
```

If an exception condition is raised by this <cast specification>, then let *TEMPST* be that exception condition.

5) Case:

a) If *TEMPST* is *successful completion (00000)*, then let *OUTST* be *successful completion (00000)*.

b) If *ONERROR* is ERROR, then let *OUTST* be *TEMPST*.

c) If *ONERROR* is NULL, then let *V* be the SQL null value and let *OUTST* be *successful completion (00000)*.

d) If *ONERROR* immediately contains DEFAULT, then let *VE* be the <value expression> immediately contained in *ONERROR*. Let *V* be the value of

```
CAST (VE AS DT)
```

Case:

i) If an exception condition is raised by this <cast specification>, then let *OUTST* be that exception condition.

ii) Otherwise, let *OUTST* be *successful completion (00000)*.

6) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *OUTST* as *STATUS OUT* and *V* as *VALUE*.

Conformance Rules

None.

9.49 Serializing an SQL/JSON sequence to an SQL string type

Function

Serialize an SQL/JSON sequence to an SQL string type in format JSON or as specified by some <implementation-defined JSON representation option>.

Subclause Signature

```
"Serializing an SQL/JSON sequence to an SQL string type" [General Rules] (
  Parameter: "STATUS IN",
  Parameter: "SQL/JSON SEQUENCE",
  Parameter: "WRAPPER BEHAVIOR",
  Parameter: "QUOTES BEHAVIOR",
  Parameter: "EMPTY BEHAVIOR",
  Parameter: "ERROR BEHAVIOR",
  Parameter: "DATA TYPE",
  Parameter: "FORMAT OPTION"
) Returns: "STATUS OUT" and "VALUE"
```

STATUS IN — an SQLSTATE value submitted to an invocation of this subclause using this signature.

SQL/JSON SEQUENCE — an SQL/JSON sequence.

WRAPPER BEHAVIOR — an indication of whether the SQL/JSON SEQUENCE is to be "wrapped" in an SQL/JSON array (WITH UNCONDITIONAL ARRAY or WITH CONDITIONAL ARRAY) or not (WITHOUT ARRAY).

QUOTES BEHAVIOR — If OMIT, indicates that a sequence containing a single character string item is cast rather than fully serialized.

EMPTY BEHAVIOR — an indication of the manner in which to behave if the SQL/JSON SEQUENCE is empty (ERROR, EMPTY ARRAY, or EMPTY OBJECT).

ERROR BEHAVIOR — an indication of the manner in which to behave if an exception condition is raised (ERROR, NULL, EMPTY ARRAY, or EMPTY OBJECT).

DATA TYPE — the data type that is the desired type of VALUE.

FORMAT OPTION — a value indicating how an SQL/JSON item must be serialized into a character or binary string (JSON implies serialization using JSON rules; other content implies using rules other than JSON rules).

STATUS OUT — an SQLSTATE value that indicates the success or reason for failure of the invocation of this subclause using this signature.

VALUE — the resulting serialized value, if successful.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *INST* be the *STATUS IN*, let *SEQ* be the *SQL/JSON SEQUENCE*, let *WRAPPER* be the *WRAPPER BEHAVIOR*, let *QB* be the *QUOTES BEHAVIOR*, let *ONEMPTY* be the *EMPTY BEHAVIOR*, let *ONERROR* be the *ERROR BEHAVIOR*, let *DT* be the *DATA TYPE*, and let *FO* be the *FORMAT OPTION* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *STATUS OUT* and *VALUE*.

2) Except where explicitly specified, the General Rules of this Subclause are not terminated if an exception condition is raised.

3) Let *TEMPST* be *INST*.

4) If *TEMPST* is *successful completion (00000)*, then:

   a) Case:

      i) If the length of *SEQ* is 0 (zero), then let *WRAPIT* be *False*.

         NOTE 501 — This insures that the ON EMPTY behavior supersedes the WRAPPER behavior.

      ii) If *WRAPPER* is WITHOUT ARRAY, then let *WRAPIT* be *False*.

      iii) If *WRAPPER* is WITH UNCONDITIONAL ARRAY, then let *WRAPIT* be *True*.

      iv) If *WRAPPER* is WITH CONDITIONAL ARRAY, then

         Case:

         1) If *SEQ* has a single SQL/JSON item, and that item is an SQL/JSON array or SQL/JSON object, then let *WRAPIT* be *False*.

         2) Otherwise, let *WRAPIT* be *True*.

   b) Case:

      i) If *WRAPIT* is *False*, then let *SEQ2* be *SEQ*.

      ii) If *WRAPIT* is *True*, then let *SEQ2* be an SQL/JSON sequence with a single SQL/JSON item, that item being an SQL/JSON array whose elements are the items of *SEQ*, preserving the order of the items in the resulting array.

   c) Case:

      i) If the length of *SEQ2* is 0 (zero), then

         Case:

         1) If *ONEMPTY* is ERROR, then let *TEMPST* be the exception condition *data exception — no SQL/JSON item (22035)*.

         2) If *ONEMPTY* is EMPTY ARRAY, then let *V* be an SQL/JSON array with no SQL/JSON elements.

         3) If *ONEMPTY* is EMPTY OBJECT, then let *V* be an SQL/JSON object with no SQL/JSON members.

         NOTE 502 — The case when *ONEMPTY* is NULL is handled later in these General Rules.

      ii) If the length of *SEQ2* is 1 (one), then let *V* be the only SQL/JSON item in *SEQ2*.

      iii) Otherwise, let *TEMPST* be the exception condition *data exception — more than one SQL/JSON item (22034)*.

5) If *TEMPST* is *successful completion (00000)*, then

Case:

a) If the length of *SEQ2* is 0 (zero) and *ONEMPTY* is NULL, then let *JT* be the null value and let *TEMPST* be *successful completion (00000)*.

b) If the length of *SEQ2* is 1 (one), the single SQL/JSON item *I* in *V* is a character string, and *QB* is OMIT, then

Case:

i) If *DT* is a character string type, then let *X* be an SQL variable whose value is *I*. Let *JT* be the result of

```
CAST ( X AS DT )
```

If this conversion results in an exception condition, then let *TEMPST* be that exception condition; otherwise, let *TEMPST* be *successful completion (00000)*.

ii) If *DT* is a binary string type, then let *BS* be a binary string whose octets are the code points of *I* after conversion to UTF8, UTF16, or UTF32 as indicated by the <JSON representation> contained in *FO*. Let *X* be an SQL variable whose value is *BS*. Let *JT* be the result of

```
CAST ( X AS DT )
```

If this conversion results in an exception condition, then let *TEMPST* be that exception condition; otherwise, let *TEMPST* be *successful completion (00000)*.

c) Otherwise, the General Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *V* as *SQL/JSON ITEM*, *FO* as *FORMAT OPTION*, and *DECT* as *TARGET TYPE*; let *TEMPST* be the *STATUS* and let *JT* be the *JSON TEXT* returned from the application of those General Rules.

6) Case:

a) If *TEMPST* is an exception condition, then

Case:

i) If *ONERROR* is ERROR, then let *OUTST* be *TEMPST*.

ii) If *ONERROR* is NULL, then let *JT* be the null value and let *OUTST* be *successful completion (00000)*.

iii) If *ONERROR* is EMPTY ARRAY or EMPTY OBJECT, then:

1) Case:

A) If *ONERROR* is EMPTY ARRAY, then let *X* be an SQL/JSON array that has no SQL/JSON elements.

B) If *ONERROR* is EMPTY OBJECT, then let *X* be an SQL/JSON object that has no SQL/JSON members.

2) The General Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *X* as *SQL/JSON ITEM*, *FO* as *FORMAT OPTION*, and *DECT* as *TARGET TYPE*; let *OUTST* be the *STATUS* and let *JT* be the *JSON TEXT* returned from the application of those General Rules.

b) Otherwise, let *OUTST* be *TEMPST*.

7) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *OUTST* as *STATUS OUT* and *JT* as *VALUE*.

Conformance Rules

None.

9.50 Converting a datetime to a formatted character string

Function

Convert a datetime value to a formatted character string value.

Subclause Signature

```
"Converting a datetime to a formatted character string" [Syntax Rules] (
  Parameter: "DATETIME TYPE",
  Parameter: "TEMPLATE",
  Parameter: "TARGET TYPE"
)
```

DATETIME TYPE — a datetime data type.

TEMPLATE — a <datetime template> (a pattern that expresses the format of a datetime when represented as a character string).

TARGET TYPE — a character string data type.

```
"Converting a datetime to a formatted character string" [General Rules] (
  Parameter: "DATETIME VALUE",
  Parameter: "TEMPLATE",
  Parameter: "TARGET TYPE"
) Returns: "FORMATTED CHARACTER STRING"
```

DATETIME VALUE — a datetime value.

TEMPLATE — a <datetime template> (a pattern that expresses the format of a datetime when represented as a character string).

TARGET TYPE — a character string data type.

FORMATTED CHARACTER STRING — a character string that represents DATETIME VALUE.

Syntax Rules

1) Let *DT* be the *DATETIME TYPE*, let *CT* be the *TEMPLATE*, and let *TT* be the *TARGET TYPE* in an application of the Syntax Rules of this Subclause.

2) The Syntax Rules of Subclause 9.52, "Datetime templates", are applied with *DT* as *DATETIME TYPE* and *CT* as *TEMPLATE*.

3) The length or maximum length of *TT* shall be equal to or greater than the minimum template length of *CT*.

Access Rules

None.

General Rules

1) Let *DV* be the *DATETIME VALUE*, let *CT* be the *TEMPLATE*, and let *TT* be the *TARGET TYPE* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *FORMATTED CHARACTER STRING*.

2) Let *CHARSET* be the character set of *TT*.

3) Let *N* be the number of <datetime template part>s contained in *CT*. Let DTP_i, $1 \leq i \leq N$, be the <datetime template part>s in *CT* enumerated in order of occurrence.

4) Let *SEC* be the SECOND field of *DV*, if any. Let *FP* be the fractional precision of *CT*. Let *RSEC* be *SEC* converted to an exact numeric value whose scale is *FP*, with implementation-defined (IA074) truncation or rounding.

5) For *i* ranging from 1 (one) to *N*, inclusive:

   Case:

   a) If DTP_i is a <datetime template delimiter>, then let S_i be a character string of character set *CHARSET* whose value is DTP_i.

   b) If DTP_i is a <datetime template year> or <datetime template rounded year>, then:

      i) Let V_i be an <unsigned integer> of exactly four <digit>s whose value is the YEAR field of *DV*.

      ii) Let L_i be the number of characters in DTP_i.

      iii) Let S_i be a character string of character set *CHARSET* and length L_i whose value is the last L_i <digit>s of V_i.

   c) If DTP_i is <datetime template month>, then let V_i be an <unsigned integer> of exactly two <digit>s whose value is the MONTH field of *DV*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

   d) If DTP_i is <datetime template day of month>, then let V_i be an <unsigned integer> of exactly two <digit>s whose value is the DAY field of *DV*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

   e) If DTP_i is <datetime template day of year>, then let V_i be an <unsigned integer> of exactly three <digit>s whose value is the calendar day of year of the calendar day identified by the YEAR, MONTH, and DAY fields of *DV*. Let S_i be a character string of character set *CHARSET* and length 3 whose value is V_i.

   f) If DTP_i is <datetime template 12-hour>, then:

      i) Let *H* be the value of the HOUR field of *DV*.

      ii) Case:

         1) If *H* is 0 (zero), then let *HH* be 12.

         2) If *H* is greater than 12, then let *HH* be *H* − 12.

         3) Otherwise, let *HH* be *H*.

iii) Let V_i be an <unsigned integer> of exactly two <digit>s whose value is *HH*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

g) If DTP_i is <datetime template 24-hour>, then let V_i be an <unsigned integer> of exactly two <digit>s whose value is the value of the HOUR field of *DV*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

h) If DTP_i is <datetime template minute>, then let V_i be an <unsigned integer> of exactly two <digit>s whose value is the value of the MINUTE field of *DV*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

i) If DTP_i is <datetime template second of minute>, then let V_i be an <unsigned integer> of exactly two <digit>s whose value is the value of the integer portion of *RSEC*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

j) If DTP_i is <datetime template second of day>, then:

i) Let *SECS* be the number of seconds in the day, using *RSEC* as the number of seconds within the minute.

NOTE 503 — If the SQL-implementation does not support leap seconds, then this value can be computed by the usual rules that there are 60 seconds in a minute and 60 minutes in an hour. If the SQL-implementation supports leap seconds, then this value can also depend on the YEAR, MONTH, and DAY fields in *DV*.

ii) Let V_i be an <unsigned integer> of exactly 5 <digit>s whose value is *SECS*.

iii) Let S_i be a character string of character set *CHARSET* and length 5 whose value is V_i.

k) If DTP_i is a <datetime template fraction>, then:

i) Let V_i be an <exact numeric literal> beginning with a <period> and followed by exactly L_i <digit>s whose value is the value of the fractional portion of *RSEC*.

ii) Let S_i be a character string of character set *CHARSET* and length L_i whose value is the last L_i <digit>s of V_i

l) If DTP_i is <datetime template am/pm>, then let *H* be the HOUR field of *DV*.

Case:

i) If *H* is less than 12, then let S_i be the character string of character set *CHARSET* and length 4 whose characters are 'A.M.'.

ii) Otherwise, let S_i be the character string of character set *CHARSET* and length 4 whose characters are 'P.M.'.

m) If DTP_i is <datetime template time zone hour>, then let *TZH* be the value of the TIMEZONE_HOUR field of *DV*. Let V_i be a <signed integer> containing a <sign> and two <digit>s whose value is *TZH*. If either the TIMEZONE_HOUR or the TIMEZONE_MINUTE fields of *DV* is negative, then the <sign> is <minus sign>; otherwise, the <sign> is <plus sign>. Let S_i be a character string of character set *CHARSET* and length 3 whose value is V_i.

NOTE 504 — For example, if the TIMEZONE_HOUR is 0 (zero) and the TIMEZONE_MINUTE is −30, then S_i is '−00', because the complete time zone is −00:30.

n) If DTP_i is <datetime template time zone minute>, then let *TZM* be the absolute value of the TIMEZONE_MINUTE field of *DV*. Let V_i be an <unsigned integer> containing two <digit>s

whose value is *TZM*. Let S_i be a character string of character set *CHARSET* and length 2 whose value is V_i.

6) Let *FCS* be the concatenation

S_1 || S_2 || ... || S_N

7) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *FCS* as *FORMATTED CHARACTER STRING*.

Conformance Rules

None.

9.51 Converting a formatted character string to a datetime

Function

Convert a formatted character string value to a datetime value.

Subclause Signature

```
"Converting a formatted character string to a datetime" [Syntax Rules] (
  Parameter: "DATETIME TYPE",
  Parameter: "TEMPLATE",
  Parameter: "CHARACTER STRING TYPE"
)
```

DATETIME TYPE — a datetime data type.

TEMPLATE — a <datetime template> (a pattern that expresses the format of a datetime when represented as a character string).

CHARACTER STRING TYPE — a character string data type.

```
"Converting a formatted character string to a datetime" [General Rules] (
  Parameter: "DATETIME TYPE",
  Parameter: "TEMPLATE",
  Parameter: "FORMATTED CHARACTER STRING"
) Returns: "DATETIME VALUE"
```

DATETIME TYPE — a datetime data type.

TEMPLATE — a <datetime template> (a pattern that expresses the format of a datetime when represented as a character string).

FORMATTED CHARACTER STRING — a character string that represents a datetime value.

DATETIME VALUE — a datetime value that corresponds to the datetime represented in FORMATTED CHARACTER STRING.

Syntax Rules

1) Let *DT* be the *DATETIME TYPE*, let *CT* be the *TEMPLATE*, and let *CST* be the *CHARACTER STRING TYPE* in an application of the Syntax Rules of this Subclause.

2) The Syntax Rules of Subclause 9.52, "Datetime templates", are applied with *DT* as *DATETIME TYPE* and *CT* as *TEMPLATE*.

3) The length or maximum length of *CST* shall be equal to or greater than the minimum template length of *CT*.

4) Let *N* be the number of <datetime template part>s contained in *CT*. For all *i*, 1 (one) ≤ *i* ≤ *N*:

   a) Let DTP_i be the *i*-th such <datetime template part>.

   b) Case:

      i) If DTP_i is <datetime template time zone hour>, then let L_i be 2.

      ii) Otherwise, let L_i be the maximum field length of DTP_i.

c) Case:

i) If DTP_i is delimited, then let Q_i be

`{1,`L_i`}`

ii) Otherwise, let Q_i be

`{`L_i`}`

NOTE 505 — Q_i is the quantifier on the regular expression that will match a string of digits. If the field is delimited, then the length of the string can be between 1 (one) and L_i digits; otherwise, the string must be exactly L_i digits.

d) Case:

i) If DTP_i is a <datetime template delimiter>, then let RE_i be the <character string literal>

Case:

1) If DTP_i is a <minus sign>, then

`'\-'`

2) If DTP_i is a <period>, then

`'\.'`

3) If DTP_i is a <solidus>, then

`'/'`

4) If DTP_i is a <comma>, then

`','`

5) If DTP_i is an <apostrophe>, then

`''''`

6) If DTP_i is a <semicolon>, then

`';'`

7) If DTP_i is a <colon>, then

`':'`

8) If DTP_i is a <space>, then

`' '`

NOTE 506 — This is a character string comprising a single <space>.

ii) If DTP_i is a <datetime template am/pm>, then let RE_i be the <character string literal>

`'((A|P)\.M\.)'`

iii) If DTP_i is a <datetime template time zone hour>, then let RE_i be the <character string literal>

'((\+|\-|)[\p{Nd}]Qi)'

NOTE 507 — The first character of a time zone hour field must be a sign or a space.

iv) Otherwise, let RE_i be the <character string literal>

'([\p{Nd}]Qi)'

NOTE 508 — That is, a numeric field matches a sequence of digits. If the field is delimited, then the number of digits can vary from 1 (one) through the length of the field; otherwise, the number of digits must be the length of the field.

5) The *regular expression defined by CT* is the concatenation

'^' || RE_1 || RE_2 || ... || RE_N || '$'

Let *RX* be the regular expression defined by *CT*.

6) For each <datetime template field> F_i, 1 (one) ≤ i ≤ N, contained in *P*, the *regular expression position* of F_i is defined as follows:

a) Let DTP_i be the <datetime template part> that is F_i.

b) Let *NLP* be the number of <left paren>s in *RX* that precede the occurrence of RE_i.

c) The regular expression position of F_i is *NLP* + 1.

7) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

1) Let *DT* be the *DATETIME TYPE*, let *CT* be the *TEMPLATE*, and let *FCS* be the *FORMATTED CHARACTER STRING* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *DATETIME VALUE*.

2) If *FCS* is the null value, then let *DV* be the null value of type *DT*. *DV* is returned as the result of the execution of these General Rules, and no further General Rules of this Subclause are evaluated.

3) Let *RE* be the regular expression defined by *CT*.

4) If the <regex like predicate>

FCS LIKE_REGEX *RE*

is *False*, then the exception condition *data exception — invalid datetime format (22007)* is raised and no further General Rules of this Subclause are evaluated.

5) Let *NOW* be the value of CURRENT_TIMESTAMP. Let *CY* be the YEAR field of *NOW*. Let *CYLIT* be an <unsigned integer> of four <digit>s whose value is *CY*. Let *CM* be the MONTH field of *NOW*. Let *CMLIT* be an <unsigned integer> of two <digit>s whose value is *CM*.

6) Case:

a) If *CT* contains a <datetime template year> *YY*, then:

i) Let *YYPOS* be an <exact numeric literal> whose value is the regular expression position of *YY*.

ii) Let *YYSTR* be the result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP YYPOS )
```

iii) Let *YYLEN* be the length of *YYSTR*.

iv) Let *YYPREFIX* be the first (4 – *YYLEN*) digits of *CYLIT*.

NOTE 509 — If the length of *YYSTR* is 4, then *YYPREFIX* is a zero-length string.

v) Let *YYYY* be the result of

```
YYPREFIX || YYSTR
```

vi) Let *YEAR* be the value of *YYYY* interpreted as an <unsigned integer>.

b) If *CT* contains a <datetime template rounded year> *RR*, then:

i) Let *RRPOS* be an <exact numeric literal> whose value is the regular expression position of *RR*.

ii) Let *RRSTR* be the result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP RRPOS )
```

iii) Let *RRLEN* be the length of *RRSTR*.

iv) Let *RY* be an implementation-defined (IV096) exact numeric value of scale 0 (zero) that is between *CY*−100 and *CY*+100, inclusive. Let *RYLIT* be an <unsigned integer> of four <digit>s whose value is *RY*.

v) Let *RRPREFIX* be the first (4 – *RRLEN*) digits of *RYLIT*.

NOTE 510 — If the length of *RRSTR* is 4, then *RRPREFIX* is a zero-length string.

vi) Let *RRRR* be the result of

```
RRPREFIX || RRSTR
```

vii) Let *YEAR* be the value of *RRRR* interpreted as an <unsigned integer>.

c) Otherwise, let *YEAR* be *CY*.

7) Case:

a) If *CT* contains a <datetime template day of year> *DOY*, then:

i) Let *DOYPOS* be an <exact numeric literal> whose value is the regular expression position of *DOY*.

ii) Let *DOYSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP DOYPOS )
```

iii) Let *DAYOFYEAR* be the value of *DOYSTR* interpreted as an <unsigned integer>.

iv) Let *MONTH* and *DAY* be the numbers of the calendar month and calendar day of month, respectively, corresponding to *DAYOFYEAR* interpreted as the calendar day of year in the current year *CY*.

b) Otherwise:

i) Case:

1) If *CT* contains a <datetime template month> *MM*, then:

A) Let *MMPOS* be an <exact numeric literal> whose value is the regular expression position of *MM*.

B) Let *MMSTR* be the result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP MMPOS )
```

C) Let *MONTH* be the value of *MMSTR* interpreted as an <unsigned literal>.

2) Otherwise, let *MONTH* be *CM*.

ii) Case:

1) If *CT* contains a <datetime template day of month> *DD*, then:

A) Let *DDPOS* be an <exact numeric literal> whose value is the regular expression position of *SJDD*.

B) Let *DDSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP DDPOS )
```

C) Let *DAY* be the value of *DDSTR* interpreted as an <unsigned integer>.

2) Otherwise, let *DAY* be 1 (one).

8) Case:

a) If *CT* contains <datetime template second of day> *SOD*, then:

i) Let *SODPOS* be an <exact numeric literal> whose value is the regular expression position of *SOD*.

ii) Let *SODSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP SODPOS )
```

iii) Let *SECOFDAY* be the value of *SODSTR* interpreted as an <unsigned integer>.

iv) Let *HOUR*, *MINUTE*, and *SECOND* be the number of hours, minutes, and seconds implied by interpreting *SECOFDAY* as the number of seconds since the beginning of the day, taking account of leap seconds if the SQL-implementation supports leap seconds and the day specified by *YEAR*, *MONTH*, and *DAY* has leap seconds.

b) Otherwise:

i) Case:

1) If *CT* contains a <datetime template 24-hour> *HH24*, then:

A) Let *HH24POS* be an <exact numeric literal> whose value is the regular expression position of *HH24*.

B) Let *HH24STR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP HH24POS )
```

C) Let *HOUR* be the value of *HH24STR* interpreted as an <unsigned integer>.

2) If *CT* contains <datetime template 12-hour> *HH12* and <datetime template am/pm> *AMPM*, then:

NOTE 511 — By the Syntax Rules of Subclause 9.52, "Datetime templates", either both are present or neither are present.

A) Let *HH12POS* be an <exact numeric literal> whose value is the regular expression position of *HH12*.

B) Let *HH12STR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP HH12POS )
```

C) Let *HOUR12* be the value of *HH12STR* interpreted as an <unsigned integer>.

D) Case:

I) If *HOUR12* is 12, then let *HOURTEMP* be 0 (zero).

II) Otherwise, let *HOURTEMP* be *HOUR12*.

E) Let *AMPMPOS* be an <exact numeric literal> whose value is the regular expression position of *AMPM*.

F) Let *AMPMSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP AMPMPOS )
```

G) Case:

I) If the first character of *AMPMSTR* is 'P', then let *HOUR* be *HOURTEMP* + 12.

II) Otherwise, let *HOUR* be *HOURTEMP*.

3) Otherwise, let *HOUR* be 0 (zero).

ii) Case:

1) If *CT* contains a <datetime template minute> *MI*, then:

A) Let *MIPOS* be an <exact numeric literal> whose value is the regular expression position of *MI*.

B) Let *MISTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP MIPOS )
```

C) Let *MIN* be the value of *MISTR* interpreted as an <unsigned integer>.

2) Otherwise, let *MIN* be 0 (zero).

iii) Case:

1) If *CT* contains a <datetime template second of minute> *SS*, then:

A) Let *SSPOS* be an <exact numeric literal> whose value is the regular expression position of *SS*.

B) Let *SSSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP SSPOS )
```

C) Let *SEC* be the value of *SSSTR* interpreted as an <unsigned integer>.

2) Otherwise, let *SS* be 0 (zero).

9) Case:

a) If *CT* contains <datetime template fraction> *FF*, then:

i) Let *FFPOS* be an <exact numeric literal> whose value is the regular expression position of *FF*.

ii) Let *FFSTR* be result of

```
'0.' || SUBSTRING_REGEX ( RX IN FCS GROUP FFPOS )
```

iii) Let *FRACTION* be the value of *FFSTR* interpreted as an <exact numeric literal>.

b) Otherwise, let *FRACTION* be the exact numeric value 0.0.

10) Case:

a) If *CT* contains a <datetime template time zone hour> *TZH*, then:

i) Let *TZHPOS* be the regular expression position of *TZH*.

ii) Let *TZHSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP TZHPOS )
```

iii) If the first character of *TZHSTR* is <space>, then the first character of *TZHSTR* is replaced with <plus sign>.

iv) Let *TZSIGN* be the first character of *TZHSTR*.

v) Let *TZHOUR* be the value of *TZHSTR* interpreted as a <signed numeric literal>.

b) Otherwise, let *TZHOUR* be 0 (zero).

11) Case:

a) If *CT* contains a <datetime template time zone minute> *TZM*, then:

i) Let *TZMPOS* be the regular expression position of *TZM*.

ii) Let *TZMSTR* be result of

```
SUBSTRING_REGEX ( RX IN FCS GROUP TZMPOS )
```

iii) Let *TZMINABS* be the value of *TZMSTR* interpreted as an <unsigned integer>.

iv) Case:

1) If *TZSIGN* is <minus sign>, then let *TZMIN* be –*TZMINABS*.

2) Otherwise, let *TZMIN* be *TZMINABS*.

b) Otherwise, let *TZMIN* be 0 (zero).

12) Case:

a) If *DT* is DATE, then let *DV* be a value of type *DT* whose YEAR field is *YEAR*, MONTH field is *MONTH*, and DAY field is *DAY*. If *DV* is not a valid date in the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format (22007)*.

b) If *DT* is TIME WITHOUT TIME ZONE, then let *DV* be a value of type *DT* whose HOUR field is *HOUR*, MINUTE field is *MIN*, and SECOND field is *SEC+FRACTION*. If *DV* is not a valid time in

the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format (22007)*.

c) If *DT* is TIME WITH TIME ZONE, then let *DV* be a value of type *DT* whose HOUR field is *HOUR*, MINUTE field is *MIN*, SECOND field is *SEC+FRACTION*, TIMEZONE_HOUR field is *TZHOUR*, and TIMEZONE_MINUTE field is *TZMIN*. If *DV* is not a valid time in the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format (22007)*.

d) If *DT* is TIMESTAMP WITHOUT TIME ZONE, then let *DV* be a value of type *DT* whose YEAR field is *YEAR*, MONTH field is *MONTH*, DAY field is *DAY*, HOUR field is *HOUR*, MINUTE field is *MIN*, and SECOND field is *SEC+FRACTION*. If *DV* is not a valid time in the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format (22007)*.

e) If *DT* is TIMESTAMP WITH TIME ZONE, then let *DV* be a value of type *DT* whose YEAR field is *YEAR*, MONTH field is *MONTH*, DAY field is *DAY*, HOUR field is *HOUR*, MINUTE field is *MIN*, SECOND field is *SEC+FRACTION*, TIMEZONE_HOUR field is *TZHOUR*, and TIMEZONE_MINUTE field is *TZMIN*. If *DV* is not a valid time in the Gregorian calendar, then an exception condition is raised: *data exception — invalid datetime format (22007)*.

13) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *DV* as *DATETIME VALUE*.

Conformance Rules

None.

9.52 Datetime templates

Function

Specify the templates to use to convert between datetime types and character string types.

Subclause Signature

```
"Datetime templates" [Syntax Rules] (
  Parameter: "DATETIME TYPE",
  Parameter: "TEMPLATE"
)
```

DATETIME TYPE — a datetime data type.

TEMPLATE — a <datetime template>.

Format

```
<datetime template> ::=
  { <datetime template part> }...

<datetime template part> ::=
    <datetime template field>
  | <datetime template delimiter>

<datetime template field> ::=
    <datetime template year>
  | <datetime template rounded year>
  | <datetime template month>
  | <datetime template day of month>
  | <datetime template day of year>
  | <datetime template 12-hour>
  | <datetime template 24-hour>
  | <datetime template minute>
  | <datetime template second of minute>
  | <datetime template second of day>
  | <datetime template fraction>
  | <datetime template am/pm>
  | <datetime template time zone hour>
  | <datetime template time zone minute>

<datetime template delimiter> ::=
    <minus sign>
  | <period>
  | <solidus>
  | <comma>
  | <apostrophe>
  | <semicolon>
  | <colon>
  | <space>

<datetime template year> ::=
    YYYY | YYY | YY | Y

<datetime template rounded year> ::=
    RRRR | RR
```

```
<datetime template month> ::=
  MM

<datetime template day of month> ::=
  DD

<datetime template day of year> ::=
  DDD

<datetime template 12-hour> ::=
    HH | HH12

<datetime template 24-hour> ::=
  HH24

<datetime template minute> ::=
  MI

<datetime template second of minute> ::=
  SS

<datetime template second of day> ::=
  SSSSS

<datetime template fraction> ::=
    FF1 | FF2 | FF3 | FF4 | FF5 | FF6 | FF7 | FF8 | FF9

<datetime template am/pm> ::=
    A.M. | P.M.

<datetime template time zone hour> ::=
  TZH

<datetime template time zone minute> ::=
  TZM
```

Syntax Rules

1) Let *DT* be the *DATETIME TYPE* and let *CT* be the *TEMPLATE* in an application of the Syntax Rules of this Subclause.

2) *CT* shall conform to the lexical grammar of a <datetime template>.

3) *CT* shall not contain two consecutive <datetime template delimiter>s.

4) *CT* shall contain at most one of each of the following: <datetime template year>, <datetime template rounded year>, <datetime template month>, <datetime template day of month>, <datetime template day of year>, <datetime template 12-hour>, <datetime template 24-hour>, <datetime template minute>, <datetime template second of minute>, <datetime template second of day>, <datetime template fraction>, <datetime template am/pm>, <datetime template time zone hour>, and <datetime template time zone minute>.

5) *CT* shall not contain both <datetime template year> and <datetime template rounded year>,

6) If *CT* contains <datetime template day of year>, then *CT* shall not contain <datetime template month> or <datetime template day of month>.

7) If *CT* contains <datetime template 24-hour>, then *CT* shall not contain <datetime template 12-hour> or <datetime template am/pm>.

8) If *CT* contains <datetime template 12-hour>, then *CT* shall contain <datetime template am/pm> and shall not contain <datetime template 24-hour>.

9) If *CT* contains <datetime template am/pm>, then *CT* shall contain <datetime template 12-hour> and shall not contain <datetime template 24-hour>.

10) If *CT* contains <datetime template second of day>, then *CT* shall not contain any of the following: <datetime template 12-hour>, <datetime template 24-hour>, <datetime template minute>, <datetime template second of minute>, or <datetime template am/pm>.

11) If *CT* contains <datetime template time zone minute>, then *CT* shall contain <datetime template time zone hour>.

12) Case:

    a) If *DT* is DATE, then *CT* shall not contain <datetime template 24-hour>, <datetime template 12-hour>, <datetime template minute>, <datetime template second of minute>, <datetime template second of day>, <datetime template fraction>, <datetime template am/pm>, <datetime template time zone hour>, or <datetime template time zone minute>.

    b) If *DT* is TIME WITHOUT TIME ZONE, then *CT* shall not contain <datetime template year>, <datetime template rounded year>, <datetime template month>, <datetime template day of month>, <datetime template day of year>, <datetime template time zone hour>, or <datetime template time zone minute>.

    c) If *DT* is TIME WITH TIME ZONE, then *CT* shall not contain <datetime template year>, <datetime template rounded year>, <datetime template month>, <datetime template day of month>, or <datetime template day of year>

    d) If *DT* is TIMESTAMP WITHOUT TIME ZONE, then *CT* shall not contain <datetime template time zone hour> or <datetime template time zone minute>.

    NOTE 512 — All datetime template fields are permitted with TIMESTAMP WITH TIME ZONE.

13) A <datetime template part> *DTP* contained in *CT* is *left-delimited* if and only if at least one of the following is true:

    a) *DTP* is the first <datetime template part> contained in *CT*.

    b) *DTP* is a <datetime template am/pm> or a <datetime template time zone hour>.

    c) *DTP* is immediately preceded in *CT* by a <datetime template delimiter> or <datetime template am/pm>.

14) A <datetime template part> *DTP* contained in *CT* is *right-delimited* if and only if at least one of the following is true:

    a) *DTP* is the last <datetime template field> contained in *CT*.

    b) *DTP* is a <datetime template am/pm>.

    c) *DTP* is immediately followed in *CT* by a <datetime template delimiter>, <datetime template am/pm>, or <datetime template time zone hour>.

15) A <datetime template part> *DTP* contained in *CT* is *delimited* if it is both left-delimited and right-delimited.

16) The *maximum field length* of a <datetime template part> *DTP* is defined as follows.

    Case:

    a) The maximum field length of a <datetime template 12-hour>, <datetime template 24-hour>, or <datetime template time zone minute> is 2.

    b) The maximum field length of a <datetime template fraction> FF1, FF2, FF3, FF4, FF5, FF6, FF7, FF8, or FF9 is 1 (one), 2, 3, 4, 5, 6, 7, 8, 9, respectively.

c) Otherwise, the maximum field length of *DTP* is the number of characters in *DTP*.

NOTE 513 — The maximum field length of <datetime template time zone hour> is 3 (one character for a sign, plus two digits).

17) The *maximum template length* of *CT* is the arithmetic sum of the maximum field lengths of the <datetime template part>s contained in *CT*.

18) The *minimum field length* of a <datetime template part> *DTP* of *CT* is

Case:

a) If *DTP* is not delimited, then the maximum field length of *DTP*.

b) If *DTP* is a <datetime template time zone hour>, then 2.

NOTE 514 — A <datetime template time zone hour> is matched by a sign and one or two digits.

c) Otherwise, 1 (one).

19) The *minimum template length* of *CT* is the arithmetic sum of the minimum field lengths of all <datetime template part>s contained in *CT*.

20) The fractional precision of *CT* is

Case:

a) If *CT* contains <datetime template fraction> FF1, FF2, FF3, FF4, FF5, FF6, FF7, FF8, or FF9, then the fractional precision is 1 (one), 2, 3, 4, 5, 6, 7, 8, or 9, respectively.

b) Otherwise the fractional precision is 0 (zero).

21) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F555, "Enhanced seconds precision", a <datetime template fraction> shall not be FF7, FF8 or FF9.

2) Without Feature F411, "Time zone specification", a <datetime template field> shall not be <datetime template time zone hour> or <datetime template time zone minute>.

10 Additional common elements

This Clause is modified by Clause 9, "Additional common elements", in ISO/IEC 9075-4.
This Clause is modified by Clause 9, "Additional common elements", in ISO/IEC 9075-9.
This Clause is modified by Clause 9, "Additional common elements", in ISO/IEC 9075-13.
This Clause is modified by Clause 11, "Additional common elements", in ISO/IEC 9075-14.
This Clause is modified by Clause 10, "Additional common elements", in ISO/IEC 9075-15.
This Clause is modified by Clause 10, "Additional common elements", in ISO/IEC 9075-16.

10.1 <interval qualifier>

Function

Specify the precision of an interval data type.

Format

```
<interval qualifier> ::=
    <start field> TO <end field>
  | <single datetime field>

<start field> ::=
  <non-second primary datetime field>
      [ <left paren> <interval leading field precision> <right paren> ]

<end field> ::=
    <non-second primary datetime field>
  | SECOND [ <left paren> <interval fractional seconds precision> <right paren> ]

<single datetime field> ::=
    <non-second primary datetime field>
        [ <left paren> <interval leading field precision> <right paren> ]
  | SECOND [ <left paren> <interval leading field precision>
        [ <comma> <interval fractional seconds precision> ] <right paren> ]

<primary datetime field> ::=
    <non-second primary datetime field>
  | SECOND

<non-second primary datetime field> ::=
    YEAR
  | MONTH
  | DAY
  | HOUR
  | MINUTE

<interval fractional seconds precision> ::=
  <unsigned integer>

<interval leading field precision> ::=
  <unsigned integer>
```

Syntax Rules

1) There is an ordering of significance of <primary datetime field>s. In order from most significant to least significant, the ordering is: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. A <start field> or <single datetime field> with an <interval leading field precision> *i* is more significant than a <start field> or <single datetime field> with an <interval leading field precision> *j* if *i*>*j*. An <end field> or <single datetime field> with an <interval fractional seconds precision> *i* is less significant than an <end field> or <single datetime field> with an <interval fractional seconds precision> *j* if *i*>*j*.

2) If TO is specified, then <start field> shall be more significant than <end field> and <start field> shall not specify MONTH. If <start field> specifies YEAR, then <end field> shall specify MONTH.

3) The maximum value of <interval leading field precision> is implementation-defined (IL059), but shall not be less than 2.

4) The maximum value of <interval fractional seconds precision> is implementation-defined (IL060), but shall not be less than 6.

5) An <interval leading field precision>, if specified, shall be greater than 0 (zero) and shall not be greater than the implementation-defined (IL059) maximum. If <interval leading field precision> is not specified, then an <interval leading field precision> of 2 is implicit.

6) An <interval fractional seconds precision>, if specified, shall be greater than or equal to 0 (zero) and shall not be greater than the implementation-defined (IL060) maximum. If SECOND is specified and <interval fractional seconds precision> is not specified, then an <interval fractional seconds precision> of 6 is implicit.

7) The precision of a field other than the <start field> or <single datetime field> is

   Case:

   a) If the field is not SECOND, then 2.

   b) Otherwise, 2 digits before the decimal point and the explicit or implicit <interval fractional seconds precision> after the decimal point.

Access Rules

None.

General Rules

1) An item qualified by an <interval qualifier> contains the datetime fields identified by the <interval qualifier>.

   Case:

   a) If the <interval qualifier> specifies a <single datetime field>, then the <interval qualifier> identifies a single <primary datetime field>. Any reference to the *most significant* or *least significant* <primary datetime field> of the item refers to that <primary datetime field>.

   b) Otherwise, the <interval qualifier> identifies those datetime fields from <start field> to <end field>, inclusive.

2) An <interval leading field precision> specifies

   Case:

a) If the <primary datetime field> is SECOND, then the number of decimal digits of precision before the specified or implied decimal point of the seconds <primary datetime field>.

b) Otherwise, the number of decimal digits of precision of the first <primary datetime field>.

3) An <interval fractional seconds precision> specifies the number of decimal digits of precision following the specified or implied decimal point in the <primary datetime field> SECOND.

4) The length in positions of an item of type interval is computed as follows.

Case:

a) If the item is a year-month interval, then

Case:

i) If the <interval qualifier> is a <single datetime field>, then the length in positions of the item is 1 (one) (the length of the <sign> of an interval value) plus the implicit or explicit <interval leading field precision> of the <single datetime field>.

ii) Otherwise, the length in positions of the item is 1 (one) (the length of the <sign> of an interval value) plus the implicit or explicit <interval leading field precision> of the <start field> plus 2 (the length of the <non-second primary datetime field> that is the <end field>) plus 1 (one) (the length of the <minus sign> between the <years value> and the <months value> in a <year-month literal>).

b) Otherwise,

Case:

i) If the <interval qualifier> is a <single datetime field> that does not specify SECOND, then the length in positions of the item is 1 (one) (the length of the <sign> of an interval value) plus the implicit or explicit <interval leading field precision> of the <single datetime field>.

ii) If the <interval qualifier> is a <single datetime field> that specifies SECOND, then the length in positions of the item is 1 (one) (the length of the <sign> of an interval value) plus the implicit or explicit <interval leading field precision> of the <single datetime field> plus the implicit or explicit <interval fractional seconds precision>. If <interval fractional seconds precision> is greater than zero, then the length in positions of the item is increased by 1 (one) (the length in positions of the <period> between the <seconds integer value> and the <seconds fraction>).

iii) Otherwise, let *participating datetime fields* mean the datetime fields that are less significant than the <start field> and more significant than the <end field> of the <interval qualifier>. The length in positions of each participating datetime field is 2.

Case:

1) If <end field> is SECOND, then the length in positions of the item is 1 (one) (the length of the <sign> of an interval value) plus the implicit or explicit <interval leading field precision>, plus 3 times the number of participating datetime fields (each participating datetime field has length 2 positions, plus the <minus sign>s or <colon>s that precede them have length 1 (one) position), plus the implicit or explicit <interval fractional seconds precision>, plus 3 (the length in positions of the <end field> other than any <interval fractional seconds precision> plus the length in positions of its preceding <colon>). If <interval fractional seconds precision> is greater than zero, then the length in positions of the item is increased by 1 (one) (the length in positions of the <period> within the field identified by the <end field>).

2) Otherwise, the length in positions of the item is 1 (one) (the length of the <sign> of an interval value) plus the implicit or explicit <interval leading field precision>, plus 3 times the number of participating datetime fields (each participating datetime field has length 2 positions, plus the <minus sign>s or <colon>s that precede them have length 1 (one) position), plus 2 (the length in positions of the <end field>), plus 1 (one) (the length in positions of the <colon> preceding the <end field>).

Conformance Rules

1) Without Feature F052, "Intervals and datetime arithmetic", conforming SQL language shall not contain an <interval qualifier>.

10.2 <language clause>

This Subclause is modified by Subclause 9.1, "<language clause>", in ISO/IEC 9075-13.

Function

Specify a programming language.

Format

```
<language clause> ::=
  LANGUAGE <language name>

<language name> ::=
    ADA
  | C
  | COBOL
  | FORTRAN
  | M | MUMPS
  | PASCAL
  | PLI
  | SQL
```

Syntax Rules

1) If MUMPS is specified, then M is implicit.

Access Rules

None.

General Rules

1) The host language specified by the <language clause> is defined in the International Standard identified by the <language name> keyword. Table 20, "Standard programming languages", specifies the relationship.

Table 20 — Standard programming languages

Language keyword	Relevant standard
ADA	ISO/IEC 8652:2012
C	ISO/IEC 9899:2018
COBOL	ISO 1989:2014
FORTRAN	ISO/IEC 1539-1:2018
M	ISO/IEC 11756:1999
PASCAL	ISO 7185:1990 and ISO/IEC 10206:1991

Language keyword	Relevant standard
PLI	ISO 6160:1979
SQL	ISO/IEC 9075 series

Conformance Rules

None.

10.3 <path specification>

Function

Specify an order for searching for an SQL-invoked routine.

Format

```
<path specification> ::=
  PATH <schema name list>

<schema name list> ::=
  <schema name> [ { <comma> <schema name> }... ]
```

Syntax Rules

1) No two <schema name>s contained in <schema name list> shall be equivalent.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <path specification>.

10.4 <routine invocation>

This Subclause is modified by Subclause 9.4, "<routine invocation>", in ISO/IEC 9075-13.

Function

Invoke an SQL-invoked routine.

Format

```
<routine invocation> ::=
  <routine name> <SQL argument list>

<routine name> ::=
  [ <schema name> <period> ] <qualified identifier>

<SQL argument list> ::=
  <left paren>
      [ <SQL argument> [ { <comma> <SQL argument> }... ] [ <co-partition clause> ] ]
      <right paren>

<SQL argument> ::=
    <value expression>
  | <generalized expression>
  | <target specification>
  | <contextually typed value specification>
  | <named argument specification>
  | <table argument>
  | <descriptor argument>

<generalized expression> ::=
  <value expression> AS <path-resolved user-defined type name>

<named argument specification> ::=
  <SQL parameter name> <named argument assignment token>
      <named argument SQL argument>

<named argument SQL argument> ::=
    <value expression>
  | <target specification>
  | <contextually typed value specification>
  | <table argument>
  | <descriptor argument>

<table argument> ::=
  <table argument proper>
      [ [ AS ] <table argument correlation name>
      [ <table argument parenthesized derived column list> ] ]
      [ <table argument partitioning> ]
      [ <table argument pruning> ]
      [ <table argument ordering> ]

<table argument correlation name> ::=
  <correlation name>

<table argument parenthesized derived column list> ::=
  <parenthesized derived column list>

<table argument proper> ::=
    TABLE <left paren> <table or query name> <right paren>
  | TABLE <table subquery>
```

```
  | <table function invocation>

<table function invocation> ::=
  <routine invocation>

<table argument partitioning> ::=
  PARTITION BY <table argument partitioning list>

<table argument partitioning list> ::=
    <column reference>
  | <left paren> [ <column reference> [ { <comma> <column reference> }... ] ] <right paren>

<table argument pruning> ::=
    PRUNE WHEN EMPTY
  | KEEP WHEN EMPTY

<table argument ordering> ::=
  ORDER BY <table argument ordering list>

<table argument ordering list> ::=
    <table argument ordering column>
  | <left paren>
        <table argument ordering column>
        [ { <comma> <table argument ordering column> }... ]
        <right paren>

<table argument ordering column> ::=
  <column reference> [ <ordering specification> ] [ <null ordering> ]

<co-partition clause> ::=
  COPARTITION <co-partition list>

<co-partition list> ::=
  <co-partition specification> [ { <comma> <co-partition specification> }... ]

<co-partition specification> ::=
  <left paren> <range variable> <comma> <range variable>
      [ { <comma> <range variable> }... ] <right paren>

<range variable> ::=
    <table name>
  | <query name>
  | <correlation name>

<descriptor argument> ::=
    <descriptor value constructor>
  | CAST <left paren> NULL AS DESCRIPTOR <right paren>
```

Syntax Rules

1) Let *RI* be the <routine invocation>, let *TP* be an empty <schema name> list, and let *UDTSM* be the null value.

2) The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *RI* as *ROUTINE INVOCATION*, *TP* as *SQLPATH*, and *UDTSM* as *UDT*; let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

1) Let *SR* be the subject routine determined by the Syntax Rules of this Subclause, and let *SAL* be the static SQL argument list determined by the Syntax Rules of this Subclause.

2) The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *V* be the *VALUE* returned from the application of those General Rules.

3) The value of the <routine invocation> is *V*.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <generalized expression>.

2) Without Feature S203, "Array parameters", conforming SQL language shall not contain an <SQL argument> whose declared type is an array type.

3) Without Feature S202, "SQL-invoked routines on multisets", conforming SQL language shall not contain an <SQL argument> whose declared type is a multiset type.

4) Without Feature B033, "Untyped SQL-invoked function arguments", conforming SQL language shall not contain a <routine invocation> that is not simply contained in a <call statement> and that simply contains an <SQL argument> that is a <dynamic parameter specification>.

5) Without Feature T521, "Named arguments in CALL statement", conforming SQL language shall not contain a <named argument specification> in a <routine invocation> whose subject routing is an SQL-invoked procedure.

6) Without Feature T522, "Default values for IN parameters of SQL-invoked procedures", conforming SQL language shall not contain a <routine invocation> that is simply contained in a <call statement> and that simply contains an <SQL argument> that is a <contextually typed value specification> and that corresponds to an input SQL parameter.

7) Without Feature T523, "Default values for INOUT parameters of SQL-invoked procedures", conforming SQL language shall not contain a <routine invocation> that is simply contained in a <call statement> and that simply contains an <SQL argument> that is a <default specification> and that corresponds to both an input SQL parameter and an output SQL parameter.

8) Without Feature T524, "Named arguments in routine invocations other than a CALL statement", conforming SQL language shall not contain a <named argument specification> in a <routine invocation> whose subject routing is an SQL-invoked function.

9) Without Feature T525, "Default values for parameters of SQL-invoked functions", conforming SQL language shall not contain a <routine invocation> whose subject routine is an SQL-invoked function and that simply contains an <SQL argument> that is a <contextually typed value specification>.

10) Without Feature B200, "Polymorphic table functions", a <routine invocation> shall not contain <table argument> or <descriptor argument>.

11) Without Feature B202, "PTF copartitioning", a <routine invocation> shall not contain <co-partition clause>.

12) Without Feature B203, "More than one copartition specification", a <routine invocation> shall contain at most one <co-partition specification>.

13) Without Feature B204, "PRUNE WHEN EMPTY", a <routine invocation> shall not contain <table argument pruning>.

14) Without Feature B206, "PTF descriptor parameters", a <routine invocation> shall not contain <descriptor argument>.

15) 13 Without Feature B207, "Cross products of partitionings", if a <routine invocation> contains more than one <table argument> that has one or more partitioning columns, then the <routine invocation> shall contain a <co-partition specification> containing precisely one <co-partition specification> that references every <table argument> that has one or more partitioning columns.

10.5 <character set specification>

Function

Identify a character set.

Format

```
<character set specification> ::=
    <standard character set name>
  | <implementation-defined character set name>
  | <user-defined character set name>

<standard character set name> ::=
  <character set name>

<implementation-defined character set name> ::=
  <character set name>

<user-defined character set name> ::=
  <character set name>
```

Syntax Rules

1) The <standard character set name>s and <implementation-defined character set name>s that are supported are implementation-defined (IA040).

2) A character set identified by a <standard character set name>, or by an <implementation-defined character set name> has associated with it a privilege descriptor that was effectively defined by the <grant statement>

   ```
   GRANT USAGE ON
   CHARACTER SET CS
   TO PUBLIC
   ```

   where *CS* is the <character set name> contained in the <character set specification>. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

3) The <standard character set name>s shall include SQL_CHARACTER and those character sets specified in Subclause 4.3.7, "Character sets", as defined by this and other standards.

4) The <implementation-defined character set name>s shall include SQL_TEXT and SQL_IDENTIFIER.

5) Let *C* be the <character set name> contained in the <character set specification>. The schema identified by the explicit or implicit <schema name> of the <character set name> shall include the descriptor of *C*.

6) If a <character set specification> is not contained in a <schema definition>, then the <character set name> immediately contained in the <character set definition> shall contain an explicit <schema name> that is not equivalent to INFORMATION_SCHEMA.

Access Rules

1) Case:

a) If <character set specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on *C*.

b) Otherwise, the current privileges shall include USAGE on *C*.

General Rules

1) A <character set specification> identifies a character set. Let the identified character set be *CS*.

2) A <standard character set name> specifies the name of a character set that is defined by a national or international standard. The character repertoire of *CS* is defined by the standard defining the character set identified by that <standard character set name>. The default collation of the character set is defined by the order of the characters in the standard and has the PAD SPACE characteristic.

3) An <implementation-defined character set name> specifies the name of a character set that is implementation-defined (IE010). The character repertoire of *CS* is implementation-defined (IV187). The default collation of the character set and whether the collation has the NO PAD characteristic or the PAD SPACE characteristic are implementation-defined (IV187).

4) A <user-defined character set name> identifies a character set whose descriptor is included in some schema whose <schema name> is not equivalent to INFORMATION_SCHEMA.

NOTE 515 — The default collation of the character set is defined as in Subclause 11.41, "<character set definition>".

5) There is a character set descriptor for every character set that can be specified by a <character set specification>.

Conformance Rules

1) Without Feature F461, "Named character sets", conforming SQL language shall not contain a <character set specification>.

10.6 <specific routine designator>

Function

Specify an SQL-invoked routine.

Format

```
<specific routine designator> ::=
    SPECIFIC <routine type> <specific name>
  | <routine type> <member name> [ FOR <schema-resolved user-defined type name> ]

<routine type> ::=
    ROUTINE
  | FUNCTION
  | PROCEDURE
  | [ INSTANCE | STATIC | CONSTRUCTOR ] METHOD

<member name> ::=
  <member name alternatives> [ <data type list> ]

<member name alternatives> ::=
    <schema qualified routine name>
  | <method name>

<data type list> ::=
  <left paren> [ <data type> [ { <comma> <data type> }... ] ] <right paren>
```

Syntax Rules

1) If a <specific name> *SN* is specified, then the <specific routine designator> shall identify an SQL-invoked routine whose <specific name> is *SN*.

2) If <routine type> specifies METHOD and none of INSTANCE, STATIC, or CONSTRUCTOR is specified, then INSTANCE is implicit.

3) If a <member name> *MN* is specified, then:

   a) If <schema-resolved user-defined type name> is specified, then <routine type> shall specify METHOD. If METHOD is specified, then <schema-resolved user-defined type name> shall be specified.

   b) Case:

      i) If <routine type> specifies METHOD, then <method name> shall be specified. Let *SCN* be the implicit or explicit <schema name> of <schema-resolved user-defined type name>, let *METH* be the <method name>, and let *RN* be *SCN*.*METH*.

      ii) Otherwise, <schema qualified routine name> shall be specified. Let *RN* be the <schema qualified routine name> of *MN* and let *SCN* be the <schema name> of *MN*.

   c) Case:

      i) If *MN* contains a <data type list>, then:

         1) If <routine type> specifies FUNCTION, then there shall be exactly one SQL-invoked regular function in the schema identified by *SCN* whose <schema qualified routine name> is *RN* such that, for all *i*, 1 (one) ≤ *i* ≤ the number of arguments, when the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared

type of its *i*-th SQL parameter as *TYPE1* and the *i*-th <data type> in the <data type list> of *MN* as *TYPE2*, those Syntax Rules are satisfied. The <specific routine designator> identifies that SQL-invoked function.

2) If <routine type> specifies PROCEDURE, then there shall be exactly one SQL-invoked procedure in the schema identified by *SCN* whose <schema qualified routine name> is *RN* such that, for all *i*, 1 (one) ≤ *i* ≤ the number of arguments, when the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of its *i*-th SQL parameter as *TYPE1* and the *i*-th <data type> in the <data type list> of *MN* as *TYPE2*, those Syntax Rules are satisfied. The <specific routine designator> identifies that SQL-invoked procedure.

3) If <routine type> specifies METHOD, then

Case:

A) If STATIC is specified, then there shall be exactly one static SQL-invoked method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH* such that, for all *i*, 1 (one) ≤ *i* ≤ the number of arguments, when the Syntax Rules of Subclause 9.30, "Data type identity", are applied with declared type of its *i*-th SQL parameter as *TYPE1* and the *i*-th <data type> in the <data type list> of *MN* as *TYPE2*, those Syntax Rules are satisfied. The <specific routine designator> identifies that static SQL-invoked method.

B) If CONSTRUCTOR is specified, then there shall be exactly one SQL-invoked constructor method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH* such that, for all *i*, 1 (one) ≤ *i* ≤ the number of arguments, when the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of its *i*-th SQL parameter in the unaugmented SQL parameter declaration list as *TYPE1* and the *i*-th <data type> in the <data type list> of *MN* as *TYPE2*, those Syntax Rules are satisfied. The <specific routine designator> identifies that SQL-invoked constructor method.

C) Otherwise, there shall be exactly one instance SQL-invoked method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH* such that, for all *i*, 1 (one) ≤ *i* ≤ the number of arguments, when the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of its *i*-th SQL parameter in the unaugmented SQL parameter declaration list as *TYPE1* and the *i*-th <data type> in the <data type list> of *MN* as *TYPE2*, those Syntax Rules are satisfied. The <specific routine designator> identifies that instance SQL-invoked method.

4) If <routine type> specifies ROUTINE, then there shall be exactly one SQL-invoked routine in the schema identified by *SCN* whose <schema qualified routine name> is *RN* such that, for all *i*, 1 (one) ≤ *i* ≤ the number of arguments, when the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of its *i*-th SQL parameter as *TYPE1* and the *i*-th <data type> in the <data type list> of *MN* as *TYPE2*, those Syntax Rules are satisfied. The <specific routine designator> identifies that SQL-invoked routine.

ii) Otherwise:

1) If <routine type> specifies FUNCTION, then there shall be exactly one SQL-invoked regular function in the schema identified by *SCN* whose <schema qualified routine name> is *RN*. The <specific routine designator> identifies that SQL-invoked function.

2) If <routine type> specifies PROCEDURE, then there shall be exactly one SQL-invoked procedure in the schema identified by *SCN* whose <schema qualified routine name> is *RN*. The <specific routine designator> identifies that SQL-invoked procedure.

3) If <routine type> specifies METHOD, then

Case:

A) If STATIC is specified, then there shall be exactly one static SQL-invoked method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH*. The <specific routine designator> identifies that static SQL-invoked method.

B) If CONSTRUCTOR is specified, then there shall be exactly one SQL-invoked constructor method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH*. The <specific routine designator> identifies that SQL-invoked constructor method.

C) Otherwise, there shall be exactly one instance SQL-invoked method of the user-defined type identified by <schema-resolved user-defined type name> whose <method name> is *METH*. The <specific routine designator> identifies that instance SQL-invoked method.

4) If <routine type> specifies ROUTINE, then there shall be exactly one SQL-invoked routine in the schema identified by *SCN* whose <schema qualified routine name> is *RN*. The <specific routine designator> identifies that SQL-invoked routine.

4) If FUNCTION is specified, then the SQL-invoked routine that is identified shall be an SQL-invoked regular function. If PROCEDURE is specified, then the SQL-invoked routine that is identified shall be an SQL-invoked procedure. If STATIC METHOD is specified, then the SQL-invoked routine that is identified shall be a static SQL-invoked method. If CONSTRUCTOR METHOD is specified, then the SQL-invoked routine that is identified shall be an SQL-invoked constructor method. If INSTANCE METHOD is specified or implicit, then the SQL-invoked routine that is identified shall be an instance SQL-invoked method. If ROUTINE is specified, then the SQL-invoked routine that is identified is either an SQL-invoked function or an SQL-invoked procedure.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <specific routine designator> that contains a <routine type> that immediately contains METHOD.

10.7 <collate clause>

Function

Specify a default collation.

Format

```
<collate clause> ::=
  COLLATE <collation name>
```

Syntax Rules

1) Let *C* be the <collation name> contained in the <collate clause>. The schema identified by the explicit or implicit <schema name> of the <collation name> shall include the descriptor of *C*.

Access Rules

1) Case:

   a) If <collate clause> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include USAGE on *C*.

   b) Otherwise, the current privileges shall include USAGE on *C*.

General Rules

None.

Conformance Rules

1) Without Feature F690, "Collation support", conforming SQL language shall not contain a <collate clause>.

10.8 <constraint name definition> and <constraint characteristics>

Function

Specify the name of a constraint and its characteristics.

Format

```
<constraint name definition> ::=
  CONSTRAINT <constraint name>

<constraint characteristics> ::=
    <constraint check time> [ [ NOT ] DEFERRABLE ] [ <constraint enforcement> ]
  | [ NOT ] DEFERRABLE [ <constraint check time> ] [ <constraint enforcement> ]
  | <constraint enforcement>

<constraint check time> ::=
    INITIALLY DEFERRED
  | INITIALLY IMMEDIATE

<constraint enforcement> ::=
  [ NOT ] ENFORCED
```

Syntax Rules

1) If <constraint check time> is not specified, then INITIALLY IMMEDIATE is implicit.

2) Case:

   a) If INITIALLY DEFERRED is specified, then:

      i) NOT DEFERRABLE shall not be specified.

      ii) If DEFERRABLE is not specified, then DEFERRABLE is implicit.

   b) If INITIALLY IMMEDIATE is specified or implicit and neither DEFERRABLE nor NOT DEFERRABLE is specified, then NOT DEFERRABLE is implicit.

3) If neither ENFORCED nor NOT ENFORCED is specified, then ENFORCED is implicit.

Access Rules

None.

General Rules

1) Let *C* be the constraint identified by <constraint name>.

2) If NOT DEFERRABLE is specified, then *C* is not deferrable; otherwise it is deferrable.

3) If <constraint check time> is INITIALLY DEFERRED, then the initial constraint mode for *C* is *deferred*; otherwise, the initial constraint mode for *C* is *immediate*.

4) If NOT ENFORCED is specified, then *C* is not enforced; otherwise, *C* is enforced.

Conformance Rules

1) Without Feature F721, "Deferrable constraints", conforming SQL language shall not contain a <constraint characteristics>, other than a <constraint enforcement>.

NOTE 516 — This means that INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

2) Without Feature F491, "Constraint management", conforming SQL language shall not contain a <constraint name definition>.

3) Without Feature F492, "Optional table constraint enforcement", conforming SQL language shall not contain a <constraint characteristics> that specifies <constraint enforcement>.

10.9 <aggregate function>

This Subclause is modified by Subclause 11.1, "<aggregate function>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 10.1, "<aggregate function>", in ISO/IEC 9075-16.

Function

16 Specify a value computed from a collection of rows.

Format

```
14 16 <aggregate function> ::=
    COUNT <left paren> <asterisk> <right paren> [ <filter clause> ]
  | <general set function> [ <filter clause> ]
  | <binary set function> [ <filter clause> ]
  | <ordered set function> [ <filter clause> ]
  | <array aggregate function> [ <filter clause> ]
  | <row pattern count function> [ <filter clause> ]
  | <JSON aggregate function> [ <filter clause> ]
  | <listagg set function> [ <filter clause> ]

<general set function> ::=
  <set function type> <left paren> [ <set quantifier> ]
      <value expression> <right paren>

<set function type> ::=
  <computational operation>

<computational operation> ::=
    AVG
  | MAX
  | MIN
  | SUM
  | EVERY
  | ANY
  | ANY_VALUE
  | SOME
  | COUNT
  | STDDEV_POP
  | STDDEV_SAMP
  | VAR_SAMP
  | VAR_POP
  | COLLECT
  | FUSION
  | INTERSECTION

<set quantifier> ::=
    DISTINCT
  | ALL

<filter clause> ::=
  FILTER <left paren> WHERE <search condition> <right paren>

<binary set function> ::=
  <binary set function type> <left paren> <dependent variable expression> <comma>
      <independent variable expression> <right paren>

<binary set function type> ::=
    COVAR_POP
  | COVAR_SAMP
```

```
  | CORR
  | REGR_SLOPE
  | REGR_INTERCEPT
  | REGR_COUNT
  | REGR_R2
  | REGR_AVGX
  | REGR_AVGY
  | REGR_SXX
  | REGR_SYY
  | REGR_SXY

<dependent variable expression> ::=
  <numeric value expression>

<independent variable expression> ::=
  <numeric value expression>

<ordered set function> ::=
    <hypothetical set function>
  | <inverse distribution function>

<hypothetical set function> ::=
  <rank function type> <left paren>
      <hypothetical set function value expression list> <right paren>
      <within group specification>

<within group specification> ::=
  WITHIN GROUP <left paren> ORDER BY <sort specification list> <right paren>

<hypothetical set function value expression list> ::=
  <value expression> [ { <comma> <value expression> }... ]

<inverse distribution function> ::=
  <inverse distribution function type> <left paren>
      <inverse distribution function argument> <right paren>
      <within group specification>

<inverse distribution function argument> ::=
  <numeric value expression>

<inverse distribution function type> ::=
    PERCENTILE_CONT
  | PERCENTILE_DISC

<listagg set function> ::=
  LISTAGG <left paren> [ <set quantifier> ] <character value expression>
      <comma> <listagg separator> [ <listagg overflow clause> ] <right paren>
      <within group specification>

<listagg separator> ::=
  <character string literal>

<listagg overflow clause> ::=
  ON OVERFLOW <overflow behavior>

<overflow behavior> ::=
    ERROR
  | TRUNCATE [ <listagg truncation filler> ] <listagg count indication>

<listagg truncation filler> ::=
  <character string literal>

<listagg count indication> ::=
    WITH COUNT
```

```
  | WITHOUT COUNT

<array aggregate function> ::=
  ARRAY_AGG
      <left paren> <value expression> [ ORDER BY <sort specification list> ] <right paren>

<row pattern count function> ::=
  COUNT <left paren> <row pattern variable name> <period> <asterisk> <right paren>
```

Syntax Rules

1) Let *AF* be the <aggregate function>.

2) If STDDEV_POP, STDDEV_SAMP, VAR_POP, or VAR_SAMP is specified, then <set quantifier> shall not be specified.

3) If <general set function> is specified and <set quantifier> is not specified, then ALL is implicit.

4) If COUNT is specified, then the declared type of the result is an implementation-defined (IV166) exact numeric type with scale 0 (zero).

5) If COUNT(*) specified and is contained in <row pattern measures> or <row pattern definition search condition>, then let *URPV* be the universal row pattern variable. *AF* is equivalent to

   ```
   COUNT (URPV.*)
   ```

6) If <row pattern count function> is specified, then *AF* shall be contained in <row pattern measures> or <row pattern definition search condition>.

7) If <general set function> is specified, then:

   a) The <value expression> *VE* shall not contain a <window function>.

   b) Let *DT* be the declared type of the <value expression>.

   c) If *AF* specifies a <general set function> whose <set quantifier> is DISTINCT, then *VE* is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

   d) If *AF* specifies a <set function type> that is MAX or MIN, then *VE* is an operand of an ordering operation.

      i) The Syntax Rules and Conformance Rules of Subclause 9.14, "Ordering operations", apply.

      ii) *DT* shall not be row-ordered.

   e) If EVERY, ANY, or SOME is specified, then *DT* shall be Boolean and the declared type of the result is Boolean.

   f) If MAX, MIN, or ANY_VALUE is specified, then the declared type of the result is *DT*.

   g) If SUM or AVG is specified, then:

      i) *DT* shall be a numeric type or an interval type.

      ii) If SUM is specified and *DT* is exact numeric with scale *S*, then the declared type of the result is an implementation-defined (IV167) exact numeric type with scale *S*.

      iii) If AVG is specified and *DT* is exact numeric, then the declared type of the result is an implementation-defined (IV168) exact numeric type with precision not less than the precision of *DT* and scale not less than the scale of *DT*.

iv) If *DT* is approximate numeric, then the declared type of the result is an implementation-defined (IV169) approximate numeric type with precision not less than the precision of *DT*.

v) If *DT* is decimal floating-point, then the declared type of the result is the decimal floating-point type with implementation-defined (IV170) precision not less than the precision of *DT*.

vi) If *DT* is interval, then the declared type of the result is interval with the same fields as *DT* and an implementation-defined (IV171) <interval leading field precision> not less than that of *DT*.

h) If VAR_POP or VAR_SAMP is specified, then *DT* shall be a numeric type.

Case:

i) If *DT* is the decimal floating-point type, then the declared type of the result is the decimal floating-point type with implementation-defined (IV172) precision not less than the precision of *DT*.

ii) Otherwise, the declared type of the result is an implementation-defined (IV173) approximate numeric type. If *DT* is an approximate numeric type, then the precision of the result is not less than the precision of *DT*.

i) `STDDEV_POP(`*X*`)` is equivalent to `SQRT(VAR_POP(`*X*`))`.

j) `STDDEV_SAMP(`*X*`)` is equivalent to `SQRT(VAR_SAMP(`*X*`))`.

k) If COLLECT is specified, then the declared type of the result is *DT* MULTISET.

l) `COLLECT (`*X*`)` is equivalent to `FUSION (MULTISET [`*X*`])`.

m) If FUSION is specified, then *DT* shall be a multiset type, and DISTINCT shall not be specified. The declared type of the result is *DT*.

n) If INTERSECTION is specified, then *DT* shall be a multiset type, and DISTINCT shall not be specified. *VE* is a multiset operand of a multiset element grouping operation, and the Syntax Rules and Conformance Rules of Subclause 9.13, "Multiset element grouping operations", apply. The declared type of the result is *DT*.

8) A <filter clause> shall not contain a <query expression>, a <window function>, or an outer reference.

9) If <binary set function> is specified, then:

a) The <dependent variable expression> *DVE* and the <independent variable expression> *IVE* shall not contain a <window function>.

b) Let *DTDVE* be the declared type of *DVE* and let *DTIVE* be the declared type of *IVE*.

c) Case:

i) If REGR_COUNT is specified, then the declared type of the result is an implementation-defined (IV174) exact numeric type with scale 0 (zero).

ii) Otherwise,

Case:

1) If either *DTDVE* or *DTIVE* is the decimal floating-point type, then the declared type of the result is the decimal floating-point type with implementation-defined (IV175) precision. If *DTDVE* is the decimal floating-point type, then the precision of the result is not less than the precision of *DTDVE*. If *DTIVE* is the decimal

floating-point type, then the precision of the result is not less than the precision of *DTIVE*.

2) Otherwise, the declared type of the result is an implementation-defined (IV176) approximate numeric type. If *DTDVE* is an approximate numeric type, then the precision of the result is not less than the precision of *DTDVE*. If *DTIVE* is an approximate numeric type, then the precision of the result is not less than the precision of *DTIVE*.

10) If <hypothetical set function> is specified, then:

a) The <hypothetical set function> shall not contain a <window function>, a <set function specification>, or a <query expression>.

b) The number of <value expression>s simply contained in <hypothetical set function value expression list> shall be the same as the number of <sort key>s simply contained in the <sort specification list>.

c) For each <value expression> *HSFVE* simply contained in the <hypothetical set function value expression list>, let *SK* be the corresponding <sort key> simply contained in the <sort specification list>.

Case:

i) If the declared type of *HSFVE* is a character string type, then the declared type of *SK* shall be a character string type with the same character repertoire as that of *HSFVE*. The Syntax Rules of Subclause 9.15, "Collation determination", are applied with set of declared types of *HSFVE* and *SK* as *TYPESET*; let the collation be the *COLL* returned from the application of those Syntax Rules.

ii) Otherwise, the declared types of *HSFVE* and *SK* shall be compatible.

d) Case:

i) If RANK or DENSE_RANK is specified, then the declared type of the result is exact numeric with implementation-defined (IV174) precision and with scale 0 (zero).

ii) Otherwise, it is implementation-defined (IA043) whether the declared type of the result is approximate numeric with implementation-defined (IA043) precision or the decimal floating-point with implementation-defined (IA043) precision.

11) If <listagg set function> is specified, then:

a) The <listagg set function> shall not contain a <window function>, a <set function specification>, or a <query expression>.

b) If DISTINCT is specified, then the <character value expression> is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

c) If <set quantifier> is not specified, then ALL is implicit.

d) It is implementation-defined (IA044) whether the declared type *DTLA* of the result is a variable-length character string with implementation-defined (IA044) maximum length or a character large object type with implementation-defined (IA044) maximum length. Let *MLRT* be the maximum length of *DTLA*.

e) Let *LAS* be the <listagg separator>. The length in characters of *LAS* shall be less than or equal to *MLRT*.

f) If <listagg overflow clause> is not specified, then ON OVERFLOW ERROR is implicit.

g) If <overflow behavior> specifies TRUNCATE and <listagg truncation filler> is not specified, then let *LTF* be a character string value comprising three consecutive <period> characters; otherwise, let *LTF* be the <listagg truncation filler>. The length in characters of *LTF* shall be less than or equal to *MLRT*.

NOTE 517 — The use of three consecutive <period> characters is a default value for *LTF* to indicate that the value is not meaningful.

12) If <inverse distribution function> is specified, then:

a) The <within group specification> shall contain a single <sort specification>.

b) The <inverse distribution function> shall not contain a <window function>, a <set function specification>, or a <query expression>.

c) Let *DT* be the declared type of the <value expression> simply contained in the <sort specification>.

d) *DT* shall be numeric or interval.

e) The declared type of the result is

Case:

i) If *DT* is exact numeric or approximate numeric, then approximate numeric with implementation-defined (IV177) precision.

ii) If *DT* is decimal floating-point, then decimal floating-point with implementation-defined (IV177) precision.

iii) If *DT* is interval, then *DT*.

13) If <array aggregate function> *AAG* is specified, then:

a) Let *VET* be the declared type of the <value expression> immediately contained in *AAG*.

b) The declared type of *AAG* is array with element type *VET* and maximum cardinality equal to the implementation-defined (IL008) maximum cardinality *IDMC* for such array types.

14) 16 A <value expression> *VE* simply contained in *AF* is an *aggregated argument* of *AF* if *AF* is not an <ordered set function> or *VE* is simply contained in a <within group specification>; otherwise, *VE* is a *non-aggregated argument* of *AF*.

15) 14 If *AF* is contained in a <row pattern measure expression> or a <row pattern definition search condition>, and if an aggregated argument of *AF* contains a column reference or a <classifier function>, then there shall be exactly one row pattern variable *RPV* such that *RPV* qualifies every column reference contained in an aggregated argument of *AF* and the explicit or implicit argument of every <classifier function> contained in an aggregated argument of *AF* is *RPV*.

Access Rules

None.

General Rules

1) 16 The *argument source* of an <aggregate function> is

Case:

a) If *AF* is immediately contained in a <set function specification> *SFS* and the aggregation query of *SFS* is a <row pattern measures> and *SFS* is contained in a <window definition>, then

Case:

i) If *SFS* contains a row pattern column reference or <classifier function>, then the set of rows that are mapped by the current designated row pattern match to *RPV*, where *RPV* is the row pattern variable that qualifies the row pattern column references contained in *SFS* or that is the argument of the <classifier function>.

ii) Otherwise, the set of all rows mapped by the current designated row pattern match.

b) If *AF* is immediately contained in a <set function specification> *SFS* and the aggregation query of *SFS* is a <row pattern measures> that is contained in a <row pattern recognition clause>, and either *SFS* specifies FINAL, or the <row pattern recognition clause> that contains *SFS* specifies ONE ROW PER MATCH, then

Case:

i) If *SFS* contains a row pattern column reference or <classifier function>, then the set of rows that are mapped by the current retained row pattern match to *RPV* where *RPV* is the row pattern variable that qualifies the row pattern column references contained in *SFS* or that is the argument of the <classifier function>.

ii) Otherwise, the set of all rows mapped by the current retained row pattern match.

c) 16 If *AF* is immediately contained in a <set function specification> *SFS* and the aggregation query of *SFS* is a <row pattern definition search condition> *RPDSC*, then

Case:

i) If *SFS* contains a row pattern column reference or <classifier function>, then the set of rows that are mapped by the current potential row pattern match to *RPV* where *RPV* is the row pattern variable that qualifies the row pattern column references contained in *SFS* or that is the argument of the <classifier function>.

ii) Otherwise, the set of all rows mapped by the current potential row pattern match and that are prior to or the same as the current row, according to the ordering of rows within row pattern partitions.

d) If *AF* is immediately contained in a <set function specification> *SFS*, then a group of a grouped table of the aggregation query of *SFS*.

e) Otherwise, the collection of rows in the current row's window frame defined by the window structure descriptor identified by the <window function> that simply contains *AF*, as defined in Subclause 7.15, "<window clause>".

2) Let *T* be the argument source of *AF*.

3) If, during the computation of the result of *AF*, an intermediate result is not representable in the declared type of the site that contains that intermediate result, then

Case:

a) If the most specific type of the result of *AF* is an interval type, then an exception condition is raised: *data exception — interval value out of range (2200P)*.

b) 14 If the most specific type of the result of *AF* is a multiset type, then an exception condition is raised: *data exception — multiset value overflow (2200Q)*.

c) If the most specific type of the result of *AF* is an array type, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

d) If the most specific type of the result of *AF* is a character string type, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

e) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.

4) Case:

a) If <filter clause> is specified, then the <search condition> is effectively evaluated for each row of *T*. Let *T1* be the collection of rows of *T* for which the result of the <search condition> is *True*.

b) Otherwise, let *T1* be *T*.

5) If COUNT(*) is specified, then the result is the cardinality of *T1*.

6) [16]If <row pattern count function> is specified, then the result is the cardinality of *T1*.

7) If <general set function> is specified, then:

a) Let *TX* be the single-column table that is the result of applying the <value expression> to each row of *T1* and eliminating null values. If one or more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function (01003)*.

b) Case:

i) If DISTINCT is specified, then let *TXA* be the result of eliminating redundant duplicate values from *TX*, using the comparison rules specified in Subclause 8.2, "<comparison predicate>", to identify the redundant duplicate values.

ii) Otherwise, let *TXA* be *TX*.

c) Let *N* be the cardinality of *TXA*.

d) Case:

i) If COUNT is specified, then the result is *N*.

ii) If *TXA* is empty, then the result is the null value.

iii) If AVG is specified, then the result is the average of the values in *TXA*.

iv) If MAX or MIN is specified, then the result is respectively the maximum or minimum value in *TXA*. These results are determined using the comparison rules specified in Subclause 8.2, "<comparison predicate>". If *DT* is a user-defined type and the comparison of two values in *TXA* results in *Unknown*, then the maximum or minimum of *TXA* is implementation-dependent (UV099).

v) If ANY_VALUE is specified, then the result is an implementation-dependent (UV062) value in *TXA*.

vi) If SUM is specified, then the result is the sum of the values in *TXA*. If the result is not within the range of the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

vii) If EVERY is specified, then

Case:

1) If the value of some element of *TXA* is *False*, then the result is *False*.

2) Otherwise, the result is *True*.

viii) If ANY or SOME is specified, then

Case:

1) If the value of some element of *TXA* is *True*, then the result is *True*.

2) Otherwise, the result is *False*.

ix) If VAR_POP or VAR_SAMP is specified, then let *SX* be the sum of values in the column of *TXA*, and let *SXS* be the sum of the squares of the values in the column of *TXA*.

1) If VAR_POP is specified, then the result is `(SXS-SX*SX/N)/N`.

2) If VAR_SAMP is specified, then

Case:

A) If *N* is 1 (one), then the result is the null value.

B) Otherwise, the result is `(SXS-SX*SX/N)/(N-1)`.

x) If FUSION is specified, then the result is the multiset *M* such that for each value *V* in the element type of *DT*, including the null value, the number of elements of *M* that are identical to *V* is the sum of the number of identical copies of *V* in the multisets that are the values of the column in each row of *TXA*.

xi) If INTERSECTION is specified, then the result is a multiset *M* such that for each value *V* in the element type of *DT*, including the null value, the number of duplicates of *V* in *M* is the minimum of the number of duplicates of *V* in the multisets that are the values of the column in each row of *TXA*.

NOTE 518 — This rule says "the result is a multiset" rather than "the result is the multiset" because the precise duplicate values are not specified. Thus this calculation is non-deterministic for certain element types, namely those based on character string, datetime with time zone and user-defined types.

NOTE 519 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

8) If <binary set function type> is specified, then:

a) Let *TXA* be the two-column table that is the result of applying the <dependent variable expression> and the <independent variable expression> to each row of *T1* and eliminating each row in which either <dependent variable expression> or <independent variable expression> is the null value. If one or more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function (01003)*.

b) Let *N* be the cardinality of *TXA*, let *SX* be the sum of the column of values of <independent variable expression>, let *SY* be the sum of the column of values of <dependent variable expression>, let *SXS* be the sum of the squares of values in the <independent variable expression> column, let *SYS* be the sum of the squares of values in the <dependent variable expression> column, and let *SXY* be the sum of the row-wise products of the value in the <independent variable expression> column times the value in the <dependent variable expression> column.

c) Case:

i) If REGR_COUNT is specified, then the result is *N*.

ii) If *N* is 0 (zero), then the result is the null value.

iii) If REGR_SXX is specified, then the result is `(SXS-SX*SX/N)`.

iv) If REGR_SYY is specified, then the result is `(SYS-SY*SY/N)`.

v) If REGR_SXY is specified, then the result is `(SXY-SX*SY/N)`.

vi) If REGR_AVGX is specified, then the result is `SX/N`.

vii) If REGR_AVGY is specified, then the result is `SY/N`.

viii) If COVAR_POP is specified, then the result is `(SXY-SX*SY/N)/N`.

ix) If COVAR_SAMP is specified, then

Case:

1) If *N* is 1 (one), then the result is the null value.

2) Otherwise, the result is `(SXY-SX*SY/N)/(N-1)`

x) If CORR is specified, then

Case:

1) If *N***SXS* equals *SX***SX*, then the result is the null value.

NOTE 520 — In this case, all remaining values of <independent variable expression> are equal, and consequently the <independent variable expression> does not correlate with the <dependent variable expression>.

2) If *N***SYS* equals *SY***SY*, then the result is the null value.

NOTE 521 — In this case, all remaining values of <dependent variable expression> are equal, and consequently the <dependent variable expression> does not correlate with the <independent variable expression>.

3) Otherwise, the result is `SQRT(POWER(N*SXY-SX*SY,2) / ((N*SXS-SX*SX)*(N*SYS-SY*SY)))`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined (IL054) exponent range for the declared type of the result, then the result is the null value.

xi) If REGR_R2 is specified, then

Case:

1) If *N***SXS* equals *SX***SX*, then the result is the null value.

NOTE 522 — In this case, all remaining values of <independent variable expression> are equal, and consequently the least-squares fit line would be vertical, or, if *N* = 1 (one), there is no uniquely determined least-squares-fit line.

2) If *N***SYS* equals *SY***SY*, then the result is 1 (one).

NOTE 523 — In this case, all remaining values of <dependent variable expression> are equal, and consequently the least-squares fit line is horizontal.

3) Otherwise, the result is `POWER(N*SXY-SX*SY,2) / ((N*SXS-SX*SX) * (N*SYS-SY*SY))`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined (IL054) exponent range for the declared type of the result, then the result is the null value.

xii) If REGR_SLOPE(*Y*, *X*) is specified, then

Case:

1) If *N***SXS* equals *SX***SX*, then the result is the null value.

NOTE 524 — In this case, all remaining values of <independent variable expression> are equal, and consequently the least-squares fit line would be vertical, or, if *N* = 1 (one), then there is no uniquely determined least-squares-fit line.

2) Otherwise, the result is `(N*SXY-SX*SY) / (N*SXS-SX*SX)`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined (IL054) exponent range for the declared type of the result, then the result is the null value.

xiii) If REGR_INTERCEPT is specified, then

Case:

1) If *N*SXS* equals *SX*SX*, then the result is the null value.

NOTE 525 — In this case, all remaining values of <independent variable expression> are equal, and consequently the least-squares fit line would be vertical, or, if *N* = 1 (one), then there is no uniquely determined least-squares-fit line.

2) Otherwise, the result is `(SY*SXS-SX*SXY) / (N*SXS-SX*SX)`. If the exponent of the approximate mathematical result of the operation is not within the implementation-defined (IL054) exponent range for the declared type of the result, then the result is the null value.

9) If <hypothetical set function> is specified, then:

a) Let *WIFT* be the <rank function type>.

b) Let *TNAME* be an implementation-defined name for *T1*.

c) Let *K* be the number of <value expression>s simply contained in <hypothetical set function value expression list>.

d) Let VE_1, ..., VE_K be the <value expression>s simply contained in the <hypothetical set function value expression list>.

e) Let *WIFTVAL*, *MARKER* and CN_1, ..., CN_K be distinct implementation-defined column names.

f) Let SP_1, ..., SP_K be the <sort specification>s simply contained in the <sort specification list>. For each *i*, let WSP_i be the <sort specification> obtained from SP_i by replacing the <sort key> with CN_i.

g) The result is the result of the <scalar subquery>

```
( SELECT WIFTVAL
  FROM ( SELECT MARKER, WIFT() OVER
                ( ORDER BY WSP1, ..., WSPK )
         FROM ( SELECT 0, SK1, ..., SKK
                FROM TNAME
                  UNION ALL
                    VALUES (1, VE1, ..., VEK) )
                AS TXNAME (MARKER, CN1, ..., CNK )
       ) AS TEMPTABLE (MARKER, WIFTVAL)
  WHERE MARKER = 1 )
```

10) If <inverse distribution function> is specified, then:

a) Let *NVE* be the value of the <inverse distribution function argument>.

b) If *NVE* is the null value, then the result is the null value.

c) If *NVE* is less than 0 (zero) or greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.

d) Let *TXA* be the single-column table that is the result of applying the <value expression> simply contained in the <sort specification> to each row of *T1* and eliminating null values. If one or

more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function (01003)*. *TXA* is ordered by the <sort specification> as specified in the General Rules of Subclause 10.10, "<sort specification list>".

e) Let *TXANAME* be an implementation-defined name for *TXA*.

f) Let *TXCOLNAME* be an implementation-defined column name for the column of *TXA*.

g) Let *WSP* be obtained from the <sort specification> by replacing the <sort key> with *TXCOLNAME*.

h) Case:

i) If *TXA* is empty, then the result is the null value.

ii) If PERCENTILE_CONT is specified, then:

1) Let *N* be the cardinality of *TXA*.

2) Let *ROW0* be the greatest exact numeric value with scale 0 (zero) that is less than or equal to *NVE**(*N*−1). Let *ROWLIT0* be a <literal> representing *ROW0*.

3) Let *ROW1* be the least exact numeric value with scale 0 (zero) that is greater than or equal to *NVE**(*N*−1). Let *ROWLIT1* be a <literal> representing *ROW1*.

4) Let *FACTOR* be an <approximate numeric literal> representing *NVE**(*N*−1)−*ROW0*.

5) The result is the result of the <scalar subquery>

```
( WITH TEMPTABLE(X, Y) AS
      ( SELECT ROW_NUMBER()
                 OVER (ORDER BY WSP) - 1,
               TXCOLNAME
        FROM TXANAME )
SELECT CAST ( T0.Y + FACTOR * (T1.Y - T0.Y) AS DT )
FROM TEMPTABLE T0, TEMPTABLE T1
WHERE T0.X = ROWLIT0
  AND T1.X = ROWLIT1 )
```

NOTE 526 — Although ROW_NUMBER is non-deterministic, the values of T0.Y and T1.Y are determined by this expression. Note that the only column of *TXA* is completely ordered by *WSP*. If *NVE**(*N*−1) is a whole number, then the rows selected from T0 and T1 are the same and the result is just T0.Y. Otherwise, the subquery performs a linear interpolation between the two consecutive values whose row numbers in the ordered set, seen as proportions of the whole, bound the argument of the PERCENTILE_CONT operator.

iii) If PERCENTILE_DISC is specified, then:

1) If the <ordering specification> simply contained in *WSP* is DESC, then let *MAXORMIN* be MAX; otherwise let *MAXORMIN* be MIN.

2) Let *NVELIT* be a <literal> representing the value of *NVE*.

3) The result is the result of the <scalar subquery>

```
( SELECT MAXORMIN (TXCOLNAME)
  FROM ( SELECT TXCOLNAME,
                CUME_DIST() OVER (ORDER BY WSP)
  FROM TXANAME ) AS TEMPTABLE (TXCOLNAME, CUMEDIST)
  WHERE CUMEDIST >= NVELIT )
```

11) If <listagg set function> is specified, then:

a) Let *T2* be *T1* ordered by the <sort specification list> as specified in the General Rules of Subclause 10.10, "<sort specification list>".

b) Let *TX* be the single-column table that is the result of applying the <character value expression> to each row of *T2* and eliminating null values. If one or more null values are eliminated, then a completion condition is raised: *warning — null value eliminated in set function (01003)*.

c) Case:

i) If DISTINCT is specified, then let *TXA* be the result of eliminating redundant duplicate values from *TX*, using the comparison rules specified in Subclause 8.2, "<comparison predicate>", to identify the redundant duplicate values.

ii) Otherwise, let *TXA* be *TX*.

d) Let *N* be the cardinality of *TXA*.

e) If *TXA* is empty, then the result is the null value and no further General Rules of this Subclause are applied.

f) Case:

i) If *N* = 1 (one), then let *LIR* be the value of the only column of the only row in *TXA*.

ii) Otherwise:

1) Let R_i, 1 (one) ≤ *i* ≤ *N*, be the rows of *TXA* according to the ordering of *TXA*.

2) Let RV_i, 1 (one) ≤ *i* ≤ *N*, be the value of the only column in R_i.

3) Let *LIR* be:

RV_1 || *LAS* || RV_2 || ... || *LAS* || RV_N

g) Case:

i) If the length in characters of *LIR* is less than or equal to *MLRT*, then the result of the <listagg set function> is *LIR*.

ii) If the length in characters of *LIR* is greater than *MLRT* and the <overflow behavior> is ERROR, then an exception condition is raised: *data exception — string data, right truncation (22001)*.

iii) Otherwise:

1) Let *LTFP1* be *LAS* || *LTF*. Let *LTFP1L* be the length in characters of *LTFP1*.

2) Case:

A) If WITH COUNT is specified, then let $LIRP_i$ be RV_1 || *LAS* || RV_2 || ... || *LAS* || RV_i, for the greatest *i*, 1 (one) ≤ *i* < *N*, such that the length in characters of $LIRP_i$ is less than or equal to:

MLRT − *LTFP1L* − CHAR_LENGTH (CAST (*N* − *i* AS VARCHAR (*MLRT*))) − 2.

Case:

I) If there is no such $LIRP_i$, then let *LIR2* be:

LTFP1 || '(' || CAST (*N* AS VARCHAR (*MLRT*)) || ')'

If the length in character of *LIR2* is greater than *MLRT*, then let *TFS* be a string having the value of three consecutive <period> characters and let *LR* be:

TFS || '(' || CAST (*N* AS VARCHAR (*MLRT*)) || ')'

Otherwise, let *LR* be *LIR2*.

II) Otherwise, let *LR* be:

LIRP$_i$ || *LTFP1* || '(' || CAST (*N* - *i* AS VARCHAR (*MLRT*)) || ')'

B) If WITHOUT COUNT is specified, then let *LIRP*$_i$ be:

RV$_1$ || *LAS* || *RV*$_2$ || ... || *LAS* || *RV*$_i$

for the greatest *i*, 1 (one) ≤ *i* < *N*, such that the length in characters of *LIRP*$_i$ is less than or equal to *MLRT* – *LTFP1L*.

Case:

I) If there is no such *LIRP*$_i$, then

Case:

1) If *LTFP1L* is greater than *MLRT*, then let *LR* be a character string with the value of three consecutive <period> characters.

NOTE 527 — The use of three consecutive <period> characters is a default value for *LR* to indicate that the value is not meaningful.

2) Otherwise, let *LR* be *LTFP1*.

II) Otherwise, let *LR* be *LIRP*$_i$ || *LTFP1*.

3) The result of the <listagg set function> is *LR*.

12) [14]If <array aggregate function> is specified, then:

a) If <sort specification list> is specified, then let *K* be the number of <sort key>s; otherwise, let *K* be 0 (zero).

b) Let *TXA* be the table of *K*+1 columns obtained by applying the <value expression> immediately contained in the <array aggregate function> to each row of *T1* to obtain the first column of *TXA*, and, for all *i*, 1 (one) ≤ *i* ≤ *K*, applying the <value expression> simply contained in the *i*-th <sort key> to each row of *T1* to obtain the (*i*+1)-th column of *TXA*.

c) Let *TXA* be ordered according to the values of the <sort key>s found in the second through (*K*+1)-th columns of *TXA*. If *K* is 0 (zero), then the ordering of *TXA* is implementation-dependent (US041).

d) Let *N* be the number of rows in *TXA*.

e) If *N* is greater than *IDMC*, then an exception condition is raised: *data exception — array data, right truncation (2202F)*.

f) Let *R*$_i$, 1 (one) ≤ *i* ≤ *N*, be the rows of *TXA* according to the ordering of *TXA*.

g) Case:

i) If *TXA* is empty, then the result of <array aggregate function> is the null value.

ii) Otherwise, the result of <array aggregate function> is an array of *N* elements such that for all *i*, 1 (one) ≤ *i* ≤ *N*, the value of the *i*-th element is the value of the first column of *R*$_i$.

NOTE 528 — Null values are not eliminated when computing <array aggregate function>. This, plus the optional <sort specification list>, sets <array aggregate function> apart from <general set function>s.

Conformance Rules

1) Without Feature R030, "Row pattern recognition: full aggregate support", an <aggregate function> contained in a <row pattern measures> or a <row pattern definition search condition> shall specify MIN, MAX, SUM, COUNT, or AVG and shall not specify DISTINCT or <filter clause>.

2) Without Feature T626, "ANY_VALUE", conforming SQL language shall not contain a <computational operation> that immediately contains ANY_VALUE.

3) Without Feature T031, "BOOLEAN data type", conforming SQL language shall not contain a <computational operation> that immediately contains EVERY, ANY, or SOME.

4) Without at least one of Feature F561, "Full value expressions", or Feature F801, "Full set function", conforming SQL language shall not contain a <general set function> that immediately contains DISTINCT and contains a <value expression> that is not a column reference.

5) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <general set function> that contains a <computational operation> that immediately contains COUNT and does not contain a <set quantifier> that immediately contains DISTINCT.

6) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <general set function> that does not contain a <set quantifier> that immediately contains DISTINCT and that contains a <value expression> that contains a column reference that does not reference a column of *T*.

7) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <binary set function> that does not contain either a <dependent variable expression> or an <independent variable expression> that contains a column reference that references a column of *T*.

8) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <value expression> simply contained in a <general set function> that contains a column reference that is an outer reference where the <value expression> is not a column reference.

9) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a <numeric value expression> simply contained in a <dependent variable expression> or an <independent variable expression> that contains a column reference that is an outer reference and in which the <numeric value expression> is not a column reference.

10) Without Feature F441, "Extended set function support", conforming SQL language shall not contain a column reference contained in an <aggregate function> *SFS1* that contains a reference to a column derived from a <value expression> that generally contains an <aggregate function> *SFS2*.

11) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <computational operation> that immediately contains STDDEV_POP, STDDEV_SAMP, VAR_POP, or VAR_SAMP.

12) Without Feature T621, "Enhanced numeric functions", conforming SQL language shall not contain a <binary set function type>.

13) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <hypothetical set function> or an <inverse distribution function>.

14) Without Feature T612, "Advanced OLAP operations", conforming SQL language shall not contain a <filter clause>.

15) Without Feature S271, "Basic multiset support", conforming SQL language shall not contain a <computational operation> that immediately contains COLLECT.

16) Without Feature S275, "Advanced multiset support", conforming SQL language shall not contain a <computational operation> that immediately contains FUSION or INTERSECTION.

NOTE 529 — If INTERSECTION is specified, then the Conformance Rules of Subclause 9.13, "Multiset element grouping operations", also apply.

17) Without Feature F442, "Mixed column references in set functions", conforming SQL language shall not contain a <hypothetical set function value expression list> that simply contains a <value expression> that contains more than one column reference, one of which is an outer reference.

18) Without Feature F442, "Mixed column references in set functions", conforming SQL language shall not contain an <inverse distribution function> that contains an <inverse distribution function argument> that simply contains a <value expression> that contains more than one column reference, one of which is an outer reference.

19) Without Feature F442, "Mixed column references in set functions", conforming SQL language shall not contain an <aggregate function> that contains a <general set function> whose simply contained <value expression> contains more than one column reference, one of which is an outer reference.

20) Without Feature F442, "Mixed column references in set functions", conforming SQL language shall not contain an <aggregate function> that contains a <binary set function> whose simply contained <dependent variable expression> or <independent variable expression> contains more than one column reference, one of which is an outer reference.

21) Without Feature F442, "Mixed column references in set functions", conforming SQL language shall not contain a <within group specification> that simply contains a <value expression> that contains more than one column reference, one of which is an outer reference.

22) Without Feature S098, "ARRAY_AGG", conforming SQL language shall not contain an <array aggregate function>.

23) Without Feature T811, "Basic SQL/JSON constructor functions", conforming SQL language shall not contain <JSON aggregate function>.

24) [14] [16] Without Feature T625, "LISTAGG", conforming SQL language shall not contain a <listagg set function>.

10.10 <sort specification list>

This Subclause is modified by Subclause 10.2, "<sort specification list>", in ISO/IEC 9075-16.

Function

Specify a sort order.

Format

```
16 <sort specification list> ::=
  <sort specification> [ { <comma> <sort specification> }... ]

<sort specification> ::=
  <sort key> [ <ordering specification> ] [ <null ordering> ]

<sort key> ::=
  <value expression>

<ordering specification> ::=
    ASC
  | DESC

<null ordering> ::=
    NULLS FIRST
  | NULLS LAST
```

Syntax Rules

1) Let *DT* be the declared type of the <value expression> simply contained in the <sort key> contained in a <sort specification>.

2) Each <value expression> simply contained in the <sort key> contained in a <sort specification> is an operand of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 9.14, "Ordering operations", apply.

3) 16 If <null ordering> is not specified, then an implementation-defined (ID133) <null ordering> is implicit. The implementation-defined (ID133) default for <null ordering> shall not depend on the context outside of <sort specification list>.

Access Rules

None.

General Rules

1) A <sort specification list> defines an ordering of rows, as follows:

   a) Let *N* be the number of <sort specification>s.

   b) Let K_i, 1 (one) ≤ i ≤ N, be the <sort key> contained in the i-th <sort specification>.

   c) Each <sort specification> specifies the *sort direction* for the corresponding sort key K_i. If DESC is not specified in the i-th <sort specification>, then the sort direction for K_i is ascending and

the applicable <comp op> is the <less than operator>. Otherwise, the sort direction for K_i is descending and the applicable <comp op> is the <greater than operator>.

d) Let P be any row of the collection of rows to be ordered, and let Q be any other row of the same collection of rows.

e) Let PV_i and QV_i be the values of K_i in P and Q, respectively. The relative position of rows P and Q in the result is determined by comparing PV_i and QV_i as follows:

i) The comparison is performed according to the General Rules of Subclause 8.2, "<comparison predicate>", where the <comp op> is the applicable <comp op> for K_i.

ii) The comparison is performed with the following special treatment of null values.

Case:

1) If PV_i and QV_i are both the null value, then they are considered equal to each other.

2) If PV_i is the null value and QV_i is not the null value, then

Case:

A) If NULLS FIRST is specified or implied, then PV_i <comp op> QV_i is considered to be *True*.

B) If NULLS LAST is specified or implied, then PV_i <comp op> QV_i is considered to be *False*.

3) If PV_i is not the null value and QV_i is the null value, then

Case:

A) If NULLS FIRST is specified or implied, then PV_i <comp op> QV_i is considered to be *False*.

B) If NULLS LAST is specified or implied, then PV_i <comp op> QV_i is considered to be *True*.

f) PV_i is said to *precede* QV_i if the value of the <comparison predicate> "`PV_i <comp op> QV_i`" is *True* for the applicable <comp op>.

g) If PV_i and QV_i are not the null value and the result of "PV_i <comp op> QV_i" is *Unknown*, then the relative ordering of PV_i and QV_i is implementation-dependent (US042).

h) The relative position of row P is before row Q if PV_n precedes QV_n for some n, 1 (one) ≤ n ≤ N, and PV_i is not distinct from QV_i for all i < n.

i) Two rows that are not distinct with respect to the <sort specification>s are said to be *peers* of each other. The relative ordering of peers is implementation-dependent (US043).

Conformance Rules

1) [16] Without Feature T611, "Elementary OLAP operations", conforming SQL language shall not contain a <null ordering>.

NOTE 530 — The Conformance Rules of Subclause 9.14, "Ordering operations", also apply.

10.11 <JSON aggregate function>

Function

Construct a JSON object or a JSON array from an aggregation of SQL data.

Format

```
<JSON aggregate function> ::=
    <JSON object aggregate constructor>
  | <JSON array aggregate constructor>

<JSON object aggregate constructor> ::=
  JSON_OBJECTAGG <left paren>
      <JSON name and value>
      [ <JSON constructor null clause> ]
      [ <JSON key uniqueness constraint> ]
      [ <JSON output clause> ]
      <right paren>

<JSON array aggregate constructor> ::=
  JSON_ARRAYAGG <left paren>
      <JSON input expression>
      [ <JSON array aggregate order by clause> ]
      [ <JSON constructor null clause> ]
      [ <JSON output clause> ]
      <right paren>

<JSON array aggregate order by clause> ::=
  ORDER BY <sort specification list>
```

Syntax Rules

1) If <JSON output clause> is not specified, then

   Case:

   a) If the declared type of any <value expression> simply contained in <JSON input expression> simply contained in <JSON aggregate function> is a JSON type, then RETURNING JSON is implicit.

   b) Otherwise, RETURNING *ST* FORMAT JSON is implicit, where *ST* is an implementation-defined (ID134) string type.

2) Let *JACFDT* be the <data type> immediately contained in the explicit or implicit <JSON output clause> *JOC* and let *JACFFO* be the explicit or implicit <JSON representation> of *JOC*.

3) If *JACFDT* is not a JSON type, then the Syntax Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *JACFFO* as *FORMAT OPTION* and *JACFDT* as *TARGET TYPE*.

4) If <JSON object aggregate constructor> is specified, then:

   a) If <JSON constructor null clause> is not specified, then NULL ON NULL is implicit.

   b) If <JSON key uniqueness constraint> is not specified, then WITHOUT UNIQUE KEYS is implicit.

   c) Let *WUK* be the explicit or implicit <JSON key uniqueness constraint>.

   d) The declared type of <JSON object aggregate constructor> is *JACFDT*.

5) If <JSON array aggregate constructor> is specified, then:

   a) If <JSON constructor null clause> is not specified, then ABSENT ON NULL is implicit.

   b) The declared type of <JSON array aggregate constructor> is *JACFDT*.

Access Rules

None.

General Rules

1) Let *AF* be the <JSON aggregate function>.

2) Let *T* be the argument source of *AF*, as defined in the General Rules of Subclause 10.9, "<aggregate function>".

3) Case:

   a) If <filter clause> is specified, then the <search condition> is effectively evaluated for each row of *T*. Let *T1* be the collection of rows of *T* for which the result of the <search condition> is *True*.

   b) Otherwise, let *T1* be *T*.

4) If <JSON object aggregate constructor> is specified, then:

   a) Let *TN* be the cardinality of *T1*.

   b) Case:

      i) If *TN* is 0 (zero), then the result of the <JSON object aggregate constructor> is the null value and no further General Rules of this Subclause are applied.

      ii) Otherwise:

         1) Let *JVE* be the <JSON input expression> simply contained in <JSON name and value>.

         2) For each i, 1 (one) $\leq i \leq TN$, let R_i be the i-th row of *T1*.

            A) Let VJN_i be the value of <JSON name> simply contained in <JSON name and value> evaluated for R_i.

            B) If VJN_i is the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.

            C) Let $VJVE_i$ be the value of *JVE* evaluated for R_i.

            D) Case:

               I) If $VJVE_i$ is a null value, then let JBV_i be the SQL/JSON null.

               II) If the declared type of $VJVE_i$ is JSON, then let VBV_i be $VJVE_i$.

               III) If *JVE* specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.42, "Parsing JSON text", are applied with $VJVE_i$ as *JSON TEXT*, *JFO* as *FORMAT OPTION*, and *WUK* as

UNIQUENESS CONSTRAINT; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

Case:

1) If *ST* is an exception condition, then the exception condition *ST* is raised.

2) Otherwise, let JBV_i be *SJI*.

IV) Otherwise,

Case:

1) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JBV_i be $VJVE_i$.

2) Otherwise, let JBV_i be the result of

```
CAST ( VJVE_i AS SDT )
```

where *SDT* is an implementation-defined (IV179) character string type with character set Unicode.

E) Let M_i be the SQL/JSON member whose key is VJN_i and whose bound value is JBV_i.

3) If WITH UNIQUE KEYS is specified and, for any i, 1 (one) $\leq i \leq TN$, and any j, $i < j \leq TN$, VJN_i and VJN_j would be equivalent when interpreted as keys of an SQL/JSON object, then an exception condition is raised: *data exception — duplicate JSON object key value (22030)*.

4) Case:

A) If the implicit or explicit <JSON constructor null clause> specifies NULL ON NULL, then

Case:

I) If *TN* is greater than the implementation-defined (IL051) maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members (2203E)*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) $\leq i \leq TN$.

B) Otherwise,

Case:

I) If the number of M_i, 1 (one) $\leq i \leq TN$, whose bound values are not the SQL/JSON null is greater than the implementation-defined (IL051) maximum number of members in a JSON object, then an exception condition is raised: *data exception — too many JSON object members (2203E)*.

II) Otherwise, let *CJO* be a JSON object whose members are M_i, 1 (one) $\leq i \leq TN$, whose bound values are not the SQL/JSON null.

NOTE 531 — There is no implied order of the members of the constructed JSON object.

c) Let *JACF* be *CJOA*.

5) If <JSON array aggregate constructor> is specified, then:

a) If <sort specification list> is specified, then let *K* be the number of <sort key>s; otherwise, let *K* be 0 (zero).

b) Let *TXA* be the table of *K*+1 columns obtained by applying the <value expression> immediately contained in the <JSON input expression> *JVE* simply contained in the <JSON array aggregate constructor> to each row of *T1* to obtain the first column of *TXA*, and, for all *i*, 1 (one) ≤ *i* ≤ *K*, applying the <value expression> simply contained in the *i*-th <sort key> to each row of *T1* to obtain the (*i*+1)-th column of *TXA*.

c) Let *TXA* be ordered according to the values of the <sort key>s found in the second through the (*K*+1)-th columns of *TXA*. If *K* is 0 (zero), then the ordering of *TXA* is implementation-dependent (US044).

d) Let *NTXA* be the number of rows in *TXA*.

e) Let R_i, 1 (one) ≤ *i* ≤ *NTXA*, be the rows of *TXA* according to the ordering of *TXA*.

f) Case:

i) If *NTXA* is 0 (zero), then the result of the <JSON array aggregate constructor> is the null value and no further General Rules of this Subclause are applied.

ii) Otherwise, for each *i*, 1 (one) ≤ *i* ≤ *NTXA*:

1) Let $VJVE_i$ be the value of the first column of R_i.

2) Case:

A) If $VJVE_i$ is a null value, then let JE_i be the SQL/JSON null.

B) If the declared type of $VJVE_i$ is JSON, then let JE_i be $VJVE_i$.

C) If *JVE* specifies, explicitly or implicitly, <JSON input clause> *JFO*, then the General Rules of Subclause 9.42, "Parsing JSON text", are applied with $VJVE_i$ as *JSON TEXT*, *JFO* as *FORMAT OPTION*, and an implementation-defined (IV178) <JSON key uniqueness constraint> as *UNIQUENESS CONSTRAINT*; let *ST* be the *STATUS* and let *SJI* be the *SQL/JSON ITEM* returned from the application of those General Rules.

Case:

I) If *ST* is an exception condition, then the exception condition *ST* is raised.

II) Otherwise, let JE_i be *SJI*.

D) Otherwise,

Case:

I) If the declared type of $VJVE_i$ is a character string type, a numeric type, or a Boolean type, then let JE_i be $VJVE_i$.

II) Otherwise, let JE_i be the result of

```
CAST (VJVE_i AS SDT)
```

where *SDT* is an implementation-defined (IV179) character string type with character set Unicode.

g) Case:

i) If the implicit or explicit <JSON constructor null clause> specifies NULL ON NULL, then

Case:

1) If *NJVE* is greater than the implementation-defined (IL052) maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements (2203D)*.

2) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ *i* ≤ *NTXA*.

ii) Otherwise,

Case:

1) If the number of JE_i, 1 (one) ≤ *i* ≤ *NJVE* that are not the SQL/JSON null is greater than the implementation-defined (IL052) maximum number of elements in a JSON array, then an exception condition is raised: *data exception — too many JSON array elements (2203D)*.

2) Otherwise, let *CJA* be a JSON array whose elements are, in order, JE_i, 1 (one) ≤ *i* ≤ *NJVE* that are not the SQL/JSON null.

NOTE 532 — The elements of constructed JSON arrays are ordered and array element indices start with 0 (zero).

h) Let *JACF* be *CJAA*.

6) Case:

a) If *JACFDT* is a JSON type, then the result of <JSON aggregate function> is *JACF*.

b) Otherwise, The General Rules of Subclause 9.43, "Serializing an SQL/JSON item", are applied with *JACF* as *SQL/JSON ITEM*, *JACFFO* as *FORMAT OPTION*, and *JACFDT* as *TARGET TYPE*; let *ST* be the *STATUS* and let *CJV* be the *JSON TEXT* returned from the application of those General Rules.

Case:

i) If *ST* is an exception condition, then the exception condition *ST* is raised.

ii) Otherwise, *CJV* is the result of <JSON aggregate function>.

Conformance Rules

1) Without Feature T811, "Basic SQL/JSON constructor functions", conforming SQL language shall not contain <JSON array aggregate constructor>.

2) Without Feature T812, "SQL/JSON: JSON_OBJECTAGG", conforming SQL language shall not contain <JSON object aggregate constructor>.

3) Without Feature T813, "SQL/JSON: JSON_ARRAYAGG with ORDER BY", conforming SQL language shall not contain <JSON array aggregate constructor> that specifies a <JSON array aggregate order by clause>.

4) Without Feature T814, “Colon in JSON_OBJECT or JSON_OBJECTAGG”, conforming SQL language shall not contain <JSON name and value 2>.

5) Without Feature T830, “Enforcing unique keys in SQL/JSON constructor functions”, conforming SQL language shall not contain a <JSON object aggregate constructor> that specifies a <JSON key uniqueness constraint>.

10.12 <JSON input expression>

Function

Specify a value to be used as input by an SQL/JSON function.

Format

```
<JSON input expression> ::=
  <value expression> [ <JSON input clause> ]

<JSON input clause> ::=
  FORMAT <JSON representation>
```

Syntax Rules

1) FORMAT JSON specifies the data format specified in RFC 8259.

2) FORMAT <implementation-defined JSON representation option> specifies an implementation-defined (IV180) data format.

NOTE 533 — For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON representation option> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of Subclause 9.42, "Parsing JSON text", and Subclause 9.43, "Serializing an SQL/JSON item", respectively.

3) The declared type *DT* of the <value expression> *VE* simply contained in <JSON input expression> *JVE* either shall be a JSON type, a character string type, a binary string type, a numeric type, a datetime type, or Boolean, or shall be a <data type> the values of which can be cast to a character string type according to the Syntax Rules of Subclause 6.13, "<cast specification>".

4) If *VE* is a JSON-returning function *JRF*, and <JSON input clause> is not specified, then

Case:

a) If the explicit or implicit <JSON output clause> simply contained in *JRF* contains FORMAT JSON, then the implicit <JSON input clause> of *JVE* is FORMAT JSON.

b) Otherwise, the implicit <JSON input clause> of *JVE* is FORMAT <implementation-defined JSON representation option>.

5) If *DT* is JSON and <JSON input clause> is not specified, then FORMAT JSON is implicit.

6) If an explicit or implicit <JSON input clause> is specified, then *DT* shall be a string type or a JSON type.

7) If *DT* is a binary string type, then an explicit or implicit <JSON input clause> shall be specified.

8) If the explicit or implicit <JSON input clause> specifies ENCODING, then *DT* shall be a binary string type.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T801, "JSON data type", in conforming SQL language, the declared type of the <value expression> immediately contained in <JSON input expression> shall not be a JSON type.

2) Without Feature T803, "String-based JSON", in conforming SQL language, the declared type of the <value expression> immediately contained in <JSON input expression> that explicitly or implicitly specifies FORMAT JSON shall not be a string type.

3) Without Feature T851, "SQL/JSON: optional keywords for default syntax", conforming SQL language shall not contain a <JSON input expression> that both immediately contains a <value expression> whose declared type is JSON and simply contains FORMAT JSON.

10.13 <JSON output clause>

Function

Specify the data type, format, and encoding of the JSON text created by a JSON-returning function.

Format

```
<JSON output clause> ::=
  RETURNING <data type> [ FORMAT <JSON representation> ]

<JSON representation> ::=
    JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
  | <implementation-defined JSON representation option>

<implementation-defined JSON representation option> ::=
  !! See the Syntax Rules.
```

Syntax Rules

1) If FORMAT is not specified, then FORMAT JSON is implicit.

2) If FORMAT JSON is specified or implicit, then the <data type> *DT* shall identify either a string type or a JSON type.

   Case:

   a) If *DT* identifies a character string type, then *DT* shall have a Universal Character Set, ENCODING shall not be specified, and an implicit choice of UTF8, UTF16, or UTF32 is determined by the character encoding form of *DT* (i.e, the keywords UTF8, UTF16, and UTF32 denote the UTF8, UTF16, and UTF32 character encoding forms, respectively).

   b) If *DT* is a binary string type and ENCODING is not specified, then it is implementation-defined (ID138) whether UTF8, UTF16, and UTF32 is implicit.

3) FORMAT JSON specifies the data format specified in RFC 8259.

4) FORMAT <implementation-defined JSON representation option> specifies an implementation-defined (IV180) data format.

   NOTE 534 — For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON representation option> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of Subclause 9.42, "Parsing JSON text", and Subclause 9.43, "Serializing an SQL/JSON item", respectively.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T801, "JSON data type", conforming SQL language shall not contain a <JSON output clause> that immediately contains a <data type> that identifies a JSON type.

2) Without Feature T803, "String-based JSON", conforming SQL language shall not contain a <JSON output clause> that explicitly or implicitly specifies FORMAT JSON and immediately contains a <data type> that identifies a string type.

3) Without Feature T851, "SQL/JSON: optional keywords for default syntax", conforming SQL language shall not contain a <JSON output clause> that simply contains FORMAT JSON and immediately contains a <data type> that specifies JSON.

10.14 <JSON API common syntax>

Function

Define the inputs to JSON_VALUE, JSON_QUERY, JSON_TABLE, and JSON_EXISTS.

Format

```
<JSON API common syntax> ::=
  <JSON context item> <comma> <JSON path specification> [ AS <JSON table path name> ]
      [ <JSON passing clause> ]

<JSON context item> ::=
  <JSON input expression>

<JSON path specification> ::=
  <character string literal>

<JSON passing clause> ::=
  PASSING <JSON argument> [ { <comma> <JSON argument> }... ]

<JSON argument> ::=
  <JSON input expression> AS <identifier>
```

Syntax Rules

1) The declared type of the <value expression> simply contained in the <JSON input expression> immediately contained in the <JSON context item> shall be a string type or a JSON type. If the <JSON context item> does not implicitly or explicitly specify a <JSON input clause>, then FORMAT JSON is implicit.

2) Case:

   a) If <JSON API common syntax> is not contained in <JSON table>, then <JSON table path name> shall not be specified.

   b) Otherwise, if <JSON table path name> is not specified, then an implementation-dependent (UV084) <JSON table path name> is implicit.

3) Let *P* be the <JSON path specification>.

4) If the <JSON API common syntax> simply contains a <JSON passing clause>, then let *PC* be that <JSON passing clause>; otherwise, let *PC* be the zero-length character string.

5) The Syntax Rules of Subclause 9.46, "SQL/JSON path language: syntax and semantics", are applied with *P* as *PATH SPECIFICATION* and *PC* as *PASSING CLAUSE*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T823, "SQL/JSON: PASSING clause", <JSON API common syntax> shall not contain <JSON passing clause>.

11 Schema definition and manipulation

This Clause is modified by Clause 10, "Schema definition and manipulation", in ISO/IEC 9075-4.
This Clause is modified by Clause 10, "Schema definition and manipulation", in ISO/IEC 9075-9.
This Clause is modified by Clause 10, "Schema definition and manipulation", in ISO/IEC 9075-13.
This Clause is modified by Clause 12, "Schema definition and manipulation", in ISO/IEC 9075-14.
This Clause is modified by Clause 11, "Schema definition and manipulation", in ISO/IEC 9075-15.
This Clause is modified by Clause 11, "Schema definition and manipulation", in ISO/IEC 9075-16.

11.1 <schema definition>

This Subclause is modified by Subclause 10.1, "<schema definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.1, "<schema definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 11.1, "<schema definition>", in ISO/IEC 9075-16.

Function

Define a schema.

Format

```
<schema definition> ::=
  CREATE SCHEMA <schema name clause>
      [ <schema character set or path> ]
      [ <schema element>... ]

<schema character set or path> ::=
    <schema character set specification>
  | <schema path specification>
  | <schema character set specification> <schema path specification>
  | <schema path specification> <schema character set specification>

<schema name clause> ::=
    <schema name>
  | AUTHORIZATION <schema authorization identifier>
  | <schema name> AUTHORIZATION <schema authorization identifier>

<schema authorization identifier> ::=
  <authorization identifier>

<schema character set specification> ::=
  DEFAULT CHARACTER SET <character set specification>

<schema path specification> ::=
  <path specification>

04 09 16 <schema element> ::=
    <table definition>
  | <view definition>
  | <domain definition>
  | <character set definition>
  | <collation definition>
  | <transliteration definition>
  | <assertion definition>
```

```
| <trigger definition>
| <user-defined type definition>
| <user-defined cast definition>
| <user-defined ordering definition>
| <transform definition>
| <schema routine>
| <sequence generator definition>
| <grant statement>
| <role definition>
```

Syntax Rules

1) If <schema name> is not specified, then a <schema name> equal to <schema authorization identifier> is implicit.

2) If AUTHORIZATION <schema authorization identifier> is not specified, then

   Case:

   a) If the <schema definition> is contained in an SQL-client module that has a <module authorization identifier> specified, then an <authorization identifier> equal to that <module authorization identifier> is implicit for the <schema definition>.

   b) Otherwise, an <authorization identifier> equal to the SQL-session user identifier is implicit.

3) The <unqualified schema name> of the explicit or implicit <schema name> shall not be equivalent to the <unqualified schema name> of the <schema name> of any other schema in the catalog identified by the <catalog name> of <schema name>.

4) If a <schema definition> is contained in an <externally-invoked procedure> in an <SQL-client module definition>, then the effective <schema authorization identifier> and <schema name> during processing of the <schema definition> are, respectively, the <schema authorization identifier> and <schema name> specified or implicit in the <schema definition>.

   NOTE 535 — Other SQL-statements executed in <externally-invoked procedure>s in the SQL-client module have the <module authorization identifier> and <schema name> specified or implicit for the SQL-client module.

5) If <schema character set specification> is not specified, then a <schema character set specification> that specifies an implementation-defined (ID139) character set that contains at least every character that is in <SQL language character> is implicit.

6) If <schema path specification> is not specified, then a <schema path specification> containing an implementation-defined (ID140) <schema name list> that contains the <schema name> contained in <schema name clause> is implicit.

7) The explicit or implicit <catalog name> of each <schema name> contained in the <schema name list> of the <schema path specification> shall be equivalent to the <catalog name> of the <schema name> contained in the <schema name clause>.

8) The <schema name list> of the explicit or implicit <schema path specification> is used as the SQL-path of the schema. The SQL-path is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the <schema definition>.

   NOTE 536 — <routine name> is defined in Subclause 5.4, "Names and identifiers".

Access Rules

1) The privileges necessary to execute the <schema definition> are implementation-defined (IW140).

General Rules

1) A <schema definition> creates an SQL-schema *S* in a catalog. *S* includes:

   a) A schema name that is equivalent to the explicit or implicit <schema name>.

   b) A schema authorization identifier that is equivalent to the explicit or implicit <authorization identifier>.

   c) A schema character set name that is equivalent to the explicit or implicit <schema character set specification>.

   d) A schema SQL-path that is equivalent to the explicit or implicit <schema path specification>.

   e) The descriptor created by every <schema element> of the <schema definition>.

2) The owner of *S* is schema authorization identifier.

3) The explicit or implicit <character set specification> is used as the default character set used for all <column definition>s and <domain definition>s that do not specify an explicit character set.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <schema path specification>.

2) Without Feature F461, "Named character sets", conforming SQL language shall not contain a <schema character set specification>.

3) Without Feature F171, "Multiple schemas per user", conforming SQL language shall not contain a <schema name clause> that contains a <schema name>.

4) Without Feature T332, "Extended roles", in conforming SQL language a <schema authorization identifier> shall not be a <role name>.

11.2 <drop schema statement>

This Subclause is modified by Subclause 10.2, "<drop schema statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.2, "<drop schema statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.1, "<drop schema statement>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.2, "<drop schema statement>", in ISO/IEC 9075-16.

Function

Destroy a schema.

Format

```
<drop schema statement> ::=
  DROP SCHEMA <schema name> <drop behavior>

<drop behavior> ::=
    CASCADE
  | RESTRICT
```

Syntax Rules

1) Let *S* be the schema identified by <schema name> *SN*.

2) *SN* shall identify a schema in the catalog identified by the explicit or implicit <catalog name>.

3) *SN* shall not identify the schema named "INFORMATION_SCHEMA" in any catalog.

4) 04 If RESTRICT is specified, then *S* shall not contain any of the following objects:

   a) Persistent base tables.

   b) Global temporary tables.

   c) Created local temporary tables.

   d) Views.

   e) Domains.

   f) Assertions.

   g) Character sets.

   h) Collations.

   i) Transliterations.

   j) Triggers.

   k) User-defined types.

   l) SQL-invoked routines.

   m) Sequence generators.

   n) 09 13 16 Roles.

   and the <schema name> of *S* shall not be contained in the SQL routine body of any routine descriptor.

NOTE 537 — If CASCADE is specified, then such objects will be dropped implicitly by the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by *SN*.

General Rules

1) 09 13 For every base table *T* or temporary table *T* included in *S*:
   a) Let *TN* be the name of *T*.
   b) The following <drop table statement> is effectively executed:

      ```
      DROP TABLE TN CASCADE
      ```

2) For every view *V* included in *S*:
   a) Let *VN* be the name of *V*.
   b) The following <drop view statement> is effectively executed:

      ```
      DROP VIEW VN CASCADE
      ```

3) For every domain *D* included in *S*:
   a) Let *DN* be the name of *D*.
   b) The following <drop domain statement> is effectively executed:

      ```
      DROP DOMAIN DN CASCADE
      ```

4) For every assertion *A* included in *S*:
   a) Let *AN* be the constraint name of *A*.
   b) The following <drop assertion statement> is effectively executed:

      ```
      DROP ASSERTION AN CASCADE
      ```

5) For every collation *CD* included in *S*:
   a) Let *CDN* be the name of *CD*.
   b) The following <drop collation statement> is effectively executed:

      ```
      DROP COLLATION CDN CASCADE
      ```

6) For every transliteration *TD* included in *S*:
   a) Let *TDN* be the name of *TD*.
   b) The following <drop transliteration statement> is effectively executed:

      ```
      DROP TRANSLATION TDN
      ```

7) For every character set *RD* included in *S*:
   a) Let *RDN* be the name of *RD*.

b) The following <drop character set statement> is effectively executed:

```
DROP CHARACTER SET RDN
```

8) 04 For every user-defined type *DT* included in *S*:

a) Let *DTN* be the name of *DT*.

b) The following <drop data type statement> is effectively executed:

```
DROP TYPE DTN CASCADE
```

9) For every trigger *TT* included in *S*:

a) Let *TTN* be the name of *TT*.

b) The following <drop trigger statement> is effectively executed:

```
DROP TRIGGER TTN
```

10) For every SQL-invoked routine *R* included in *S*:

a) Let *SN* be the <specific name> of *R*.

b) The following <drop routine statement> is effectively executed:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

11) Let *RS* be the set of all SQL-invoked routines whose routine descriptor contains the <schema name> of *S* in the <SQL routine body>. For every routine *R* in *RS*:

a) 04 Let *SN* be the <specific name> of *R*.

b) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

12) 13 16 For every sequence generator *SEQ* included in *S*:

a) Let *SEQN* be the sequence generator name of *SEQ*.

b) The following <drop sequence generator statement> is effectively executed:

```
DROP SEQUENCE SEQN CASCADE
```

13) *S* is destroyed.

Conformance Rules

1) Without Feature F032, “CASCADE drop behavior”, conforming SQL language shall not contain a <drop behavior> that contains CASCADE.

2) Without Feature F381, “Extended schema manipulation”, conforming SQL language shall not contain a <drop schema statement>.

11.3 <table definition>

This Subclause is modified by Subclause 10.3, "<table definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.3, "<table definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.2, "<table definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.3, "<table definition>", in ISO/IEC 9075-16.

Function

Define a persistent base table, a created local temporary table, or a global temporary table.

Format

```
<table definition> ::=
  CREATE [ <table scope> ] TABLE <table name> <table contents source>
      [ WITH <system versioning clause> ]
      [ ON COMMIT <table commit action> ROWS ]

<table contents source> ::=
    <table element list>
  | <typed table clause>
  | <as subquery clause>

<table scope> ::=
  <global or local> TEMPORARY

<global or local> ::=
    GLOBAL
  | LOCAL

<system versioning clause> ::=
  SYSTEM VERSIONING

<table commit action> ::=
    PRESERVE
  | DELETE

<table element list> ::=
  <left paren> <table element> [ { <comma> <table element> }... ] <right paren>

<table element> ::=
    <column definition>
  | <table period definition>
  | <table constraint definition>
  | <like clause>

<typed table clause> ::=
  OF <path-resolved user-defined type name> [ <subtable clause> ]
      [ <typed table element list> ]

<typed table element list> ::=
  <left paren> <typed table element>
      [ { <comma> <typed table element> }... ] <right paren>

<typed table element> ::=
    <column options>
  | <table constraint definition>
  | <self-referencing column specification>

<self-referencing column specification> ::=
  REF IS <self-referencing column name> [ <reference generation> ]
```

```
<reference generation> ::=
    SYSTEM GENERATED
  | USER GENERATED
  | DERIVED

<self-referencing column name> ::=
  <column name>

<column options> ::=
  <column name> WITH OPTIONS <column option list>

09 <column option list> ::=
  [ <scope clause> ] [ <default clause> ] [ <column constraint definition>... ]

<subtable clause> ::=
  UNDER <supertable clause>

<supertable clause> ::=
  <supertable name>

<supertable name> ::=
  <table name>

<like clause> ::=
  LIKE <table name> [ <like options> ]

<like options> ::=
  <like option>...

<like option> ::=
    <identity option>
  | <column default option>
  | <generation option>

<identity option> ::=
    INCLUDING IDENTITY
  | EXCLUDING IDENTITY

<column default option> ::=
    INCLUDING DEFAULTS
  | EXCLUDING DEFAULTS

<generation option> ::=
    INCLUDING GENERATED
  | EXCLUDING GENERATED

<as subquery clause> ::=
  [ <left paren> <column name list> <right paren> ] AS <table subquery>
      <with or without data>

<with or without data> ::=
    WITH NO DATA
  | WITH DATA

<table period definition> ::=
  <system or application time period specification>
      <left paren> <period begin column name> <comma> <period end column name> <right paren>

<system or application time period specification> ::=
    <system time period specification>
  | <application time period specification>

<system time period specification> ::=
  PERIOD FOR SYSTEM_TIME
```

```
<application time period specification> ::=
  PERIOD FOR <application time period name>

<application time period name> ::=
  <identifier>

<period begin column name> ::=
  <column name>

<period end column name> ::=
  <column name>
```

Syntax Rules

1) At most one <table element> shall be a <table period definition> that contains a <system time period specification>. At most one <table element> shall be a <table period definition> that contains an <application time period specification>.

2) If a <table period definition> *TPD* is specified, then:

   a) <table scope> shall not be specified.

   b) <table contents source> shall specify <table element list>.

   c) If *TPD* contains a <system time period specification>, then let *PN* be SYSTEM_TIME; otherwise, let *PN* be the <application time period name> contained in *TPD*.

   d) No <column name> in any <column definition> shall be equivalent to *PN*.

   e) <table element list> shall contain two <column definition>s *CD1* and *CD2* such that all of the following are true:

      i) The <column name> *CN1* included in *CD1* is equivalent to the <period begin column name> contained in *TPD*. The column identified by *CN1* is the *PN* period start column.

      ii) The <column name> *CN2* included in *CD2* is equivalent to the <period end column name> contained in *TPD*. The column identified by *CN2* is the *PN* period end column.

      iii) The <data type or domain name> contained in *CD1* is either DATE or a timestamp type and it is equivalent to the <data type or domain name> contained in *CD2*.

      iv) *CD1* and *CD2* both contain either an explicit or an implicit <column constraint definition> that specifies NOT NULL NOT DEFERRABLE ENFORCED.

      v) Case:

         1) If *TPD* contains a <system time period specification>, then *CD1* shall contain a <system time period start column specification> and *CD2* shall contain a <system time period end column specification>.

         2) Otherwise:

            A) Neither *CD1* nor *CD2* shall contain an <identity column specification>, a <generation clause>, a <system time period start column specification>, or a <system time period end column specification>.

            B) Let *S* be the schema identified by the explicit or implicit <schema name> of *TN*. Let *IDCN* be an implementation-defined <constraint name> that is not equivalent to the <constraint name> of any table constraint descriptor included in *S*. The following <table constraint definition> is implicit:

               ```
               CONSTRAINT IDCN CHECK ( CN1 < CN2 )
               ```

3) If WITH <system versioning clause> is specified, then a <table period definition> *STT* that specifies SYSTEM_TIME shall be specified.

4) 04 The <table contents source> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

5) Let *T* be the table defined by the <table definition> *TD*. Let *TN* be the <table name> simply contained in *TD*.

6) If a <table definition> is contained in a <schema definition> *SD* and *TN* contains a <local or schema qualifier>, then that <local or schema qualifier> shall be equivalent to the implicit or explicit <schema name> of *SD*.

7) 16 *TN* shall not identify an existing table descriptor.

8) If the <table definition> is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>. Otherwise, let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *TN*.

9) If <table element list> *TEL* is specified, then:

   a) *TEL* shall contain at least one <column definition> or <like clause>.

   b) For each <like clause> *LC* that is directly contained in *TEL*:

      i) Let *LT* be the table identified by the <table name> contained in *LC*. *LT* shall not be a system-versioned table.

      ii) If *LT* is a viewed table, then <like options> shall not be specified.

      iii) Let *D* be the degree of *LT*. For *i*, 1 (one) ≤ *i* ≤ *D*:

         1) Let LCD_i be the column descriptor of the *i*-th column of *LT*.

         2) Let LCN_i be the column name included in LCD_i.

         3) Let LDT_i be the data type included in LCD_i.

         4) If the nullability characteristic included in LCD_i is known not nullable, then let LNC_i be NOT NULL; otherwise, let LNC_i be the zero-length character string.

         5) Let CD_i be the <column definition>

            LCN_i LDT_i LNC_i

      iv) If <like options> is specified, then:

         1) <identity option> shall not be specified more than once, <column default option> shall not be specified more than once, and <generation option> shall not be specified more than once.

         2) If <identity option> is not specified, then EXCLUDING IDENTITY is implicit.

         3) If <column default option> is not specified, then EXCLUDING DEFAULTS is implicit.

         4) If <generation option> is not specified, then EXCLUDING GENERATED is implicit.

         5) If INCLUDING IDENTITY is specified and *LT* includes an identity column, then let *ICD* be the column descriptor of that column included in the table descriptor of *LT*. Let *SGD* be the sequence generator descriptor included in *ICD*.

A) Let *SV* be the start value included in *SGD*.

B) Let *IV* be the increment included in *SGD*.

C) Let *MAX* be the maximum value included in *SGD*.

D) Let *MIN* be the minimum value included in *SGD*.

E) Let *CYC* be the cycle option included in *SGD*.

F) Let *k* be the ordinal position in which the column described by *ICD* appears in the table identified by *LT*.

G) Case:

I) If *ICD* indicates that values are always generated, then let *G* be GENERATED ALWAYS.

II) If *ICD* indicates that values are generated by default, then let *G* be GENERATED BY DEFAULT.

H) The value of CD_k is replaced by:

```
LCN_k LDT_k
  G AS IDENTITY ( START WITH SV, INCREMENT BY IV,
                  MAXVALUE MAX, MINVALUE MIN, CYC ) LNC_k
```

6) If INCLUDING GENERATED is specified, then let GCD_j, 1 (one) $\leq j \leq D$, be the column descriptors included in the descriptor of *LT*, with *j* being the ordinal position of the column described by GCD_j. For each GCD_j that indicates that the column it describes is a generated column:

A) Let GE_j be the <generation expression> included in GCD_j, where the <table name> contained in every contained <column reference> is replaced by *TN*.

B) The value of CD_j is replaced by

```
LCN_j LDT_j GENERATED ALWAYS AS GE_j LNC_j
```

7) If INCLUDING DEFAULTS is specified, then let DCD_m, 1 (one) $\leq m \leq D$, be the column descriptors included in the descriptor of *LT*, with *m* being the ordinal position of the column described by DCD_m.

For each DCD_m, if DCD_m includes a <default option> DO_m, then the value of CD_m is replaced by

```
LCN_m LDT_m DEFAULT  DO_m LCN_m
```

v) *LC* is effectively replaced by:

```
CD_1, ..., CD_D
```

NOTE 538 — <column constraint>s, except for NOT NULL, are not included in CD_i; <column constraint definition>s are effectively transformed to <table constraint definition>s and are thereby also excluded.

10) If <as subquery clause> is specified, then:

a) Let *ASQ* be the <table subquery> immediately contained in <as subquery clause>. Let *QT* be the table specified by *ASQ*.

b) If any two columns in *QT* have equivalent <column name>s, or if any column of *QT* has an implementation-dependent name, then <column name list> shall be specified.

c) Let *D* be the degree of *QT*.

d) <column name list> shall not contain two or more equivalent <column name>s.

e) The number of <column name>s in <column name list> shall be *D*.

f) For *i*, 1 (one) ≤ *i* ≤ *D*:

i) Case:

1) If <column name list> is specified, then let QCN_i be the *i*-th <column name> in that <column name list>.

2) Otherwise, let QCN_i be the <column name> of the *i*-th column of *QT*.

ii) Let QDT_i be the declared type of the *i*-th column of *QT*.

iii) If the nullability characteristic of the *i*-th column of *QT* is known not nullable, then let QNC_i be NOT NULL; otherwise, let QNC_i be the zero-length character string.

iv) Let CD_i be the <column definition>

QCN_i QDT_i QNC_i

g) <as subquery clause> is effectively replaced by a <table element list> *TEL* of the form:

CD_1, ..., CD_D

11) If <typed table clause> *TTC* is specified, then:

a) The <user-defined type name> simply contained in <path-resolved user-defined type name> shall identify a structured type *ST*.

b) If <subtable clause> is specified, then <self-referencing column specification> shall not be specified. Otherwise, <self-referencing column specification> shall be specified exactly once.

c) If <self-referencing column specification> *SRCS* is specified, then let *RST* be the reference type REF(*ST*).

i) <subtable clause> shall not be specified.

ii) <table scope> shall not be specified.

iii) If SYSTEM GENERATED is specified, then *RST* shall have a system-defined representation.

iv) If USER GENERATED is specified, then *RST* shall have a user-defined representation.

v) If DERIVED is specified, then *RST* shall have a derived representation.

vi) If *RST* has a derived representation, then let *m* be the number of derivational attributes of the derived representation of *RST* and let A_i, 1 (one) ≤ *i* ≤ *m*, be those derivational attributes.

1) *TD* shall contain a <table constraint definition> that specifies a <unique constraint definition> *UCD* whose <unique column list> contains the attribute names of A_1, A_2, ..., A_m in that order.

2) If *UCD* does not specify PRIMARY KEY, then for every attribute A_i, 1 (one) ≤ i ≤ m, *TD* shall contain a <column options> CO_i with a <column name> that is equivalent to the <attribute name> of A_i and with a <column constraint definition> that specifies NOT NULL.

vii) Let CD_0 be the <column definition>:

```
CN0 RST SCOPE(TN) UNIQUE NOT NULL
```

where CN_0 denotes the <self-referencing column name> simply contained in *SRCS*.

d) If <subtable clause> is specified, then:

i) The <table name> contained in the <subtable clause> identifies the *direct supertable* of *T*, which shall be a base table. *T* is called a *direct subtable* of the direct supertable of *T*.

ii) *ST* shall be a direct subtype of the structured type of the direct supertable of *T*.

iii) The SQL-schema identified by the explicit or implicit <schema name> of the <table name> of *T* shall include the descriptor of the direct supertable of *T*.

iv) The subtable family of *T* shall not include a member, other than *T* itself, whose associated structured type is *ST*.

v) *TD* shall not contain a <table constraint definition> that specifies PRIMARY KEY.

vi) Let the term *inherited column* of *T* refer to a column of *T* that corresponds to an inherited attribute of *ST*. For every such inherited attribute *IA*, there is a column *CA* of the direct supertable of *T* such that the <column name> of *CA* is equivalent to the <attribute name> of *IA*. *CA* is called the *direct supercolumn* of *IA* in the direct supertable of *T*.

vii) Let CD_0 be the <column definition>:

```
CN0 RST SCOPE(TN) UNIQUE NOT NULL
```

where CN_0 denotes the <self-referencing column name> simply contained in *SRCS*.

e) [13] Let the term *originally-defined column* of *T* refer to a column of *T* that corresponds to an originally-defined attribute of *ST*.

f) Let *n* be the number of attributes of *ST*. Let AD_i, 1 (one) ≤ i ≤ n, be the attribute descriptors included in the data type descriptor of *ST* and let CD_i be the <column definition> `CNi DTi DCi`, where:

i) CN_i is the attribute name included in AD_i.

ii) DT_i is some <data type> that, under the General Rules of Subclause 6.1, "<data type>", would result in the creation of the data type descriptor included in AD_i.

iii) If AD_i describes an inherited attribute *IA*, then

Case:

1) If the column descriptor of the direct supercolumn of *IA* includes a default value, then DC_i is some <default clause> whose <default option> denotes this default value.

2) Otherwise, DC_i is the zero-length character string.

iv) If AD_i describes an originally-defined attribute *OA*, then

Case:

1) If AD_i includes a default value, then DC_i is some <default clause> whose <default option> denotes this default value.

2) Otherwise, DC_i is the zero-length character string.

g) If <typed table element list> *TTEL* is specified and <column options> *CO* is specified, then:

i) The <column name> *CN* simply contained in *CO* shall be equivalent to the <column name> CN_j specified in some <column definition> CD_j and shall refer to an originally-defined column of *T*.

ii) *CN* shall not be equivalent to the <column name> simply contained in any other <column options> contained in *TTEL*.

iii) 09 A <column option list> shall immediately contain either a <scope clause> or a <default clause>, or at least one <column constraint definition>.

iv) If *CO* specifies a <scope clause> *SC*, then DT_j shall be a <reference type> *RT*. If *RT* contains a <scope clause>, then that <scope clause> is replaced by *SC*; otherwise, *RT* is replaced by *RT SC*.

NOTE 539 — Changes to the scope of a column of a typed table do not affect the scope defined for the underlying attribute. Such an attribute scope serves as a kind of default for the column's scope, at the time the typed table is defined, and is not restored if a column's scope is dropped.

v) If *CO* specifies a <default clause> *DC*, then DC_j is replaced by *DC* in CD_j.

vi) 09 If *CO* specifies a non-empty list *CCDL* of <column constraint definition>s, then CD_j is replaced by CD_j *CCDL*.

vii) *CO* is deleted from *TTEL*.

h) *T* is a referenceable table.

i) If *TTEL* is empty, then let *TEL* be a <table element list> of the form

CD_0, ..., CD_n

Otherwise, then let *TEL* be a <table element list> of the form

CD_0, ..., CD_n *TTEL*

12) If ON COMMIT is specified, then TEMPORARY shall be specified.

13) If TEMPORARY is specified and ON COMMIT is not specified, then ON COMMIT DELETE ROWS is implicit.

14) Every referenceable table referenced by a <scope clause> contained in a <column definition> or <column options> contained in *TD* shall be

Case:

a) If *TD* specifies no <table scope>, then a persistent base table.

b) If *TD* specifies GLOBAL TEMPORARY, then a global temporary table.

c) If *TD* specifies LOCAL TEMPORARY, then a created local temporary table.

15) At most one <table element> shall be a <column definition> that contains an <identity column specification>.

16) The <table name> is a range variable whose scope is the <table definition>, excluding the <as subquery clause>.

Access Rules

1) If a <table definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) If a <like clause> is contained in a <table definition>, then the applicable privileges for *A* shall include SELECT privilege on the table identified in the <like clause>.

3) *A* shall have in its applicable privileges the UNDER privilege on the <supertable name> specified in <subtable clause>.

4) If "OF <path-resolved user-defined type name>" is specified, then the applicable privileges for *A* shall include USAGE on *ST*.

General Rules

1) A <table definition> defines either a regular persistent base table, a system-versioned table, a global temporary table, or a created local temporary table.

   Case:

   a) If GLOBAL is specified, then a global temporary table is defined.

   b) If LOCAL is specified, then a created local temporary table is defined.

   c) If WITH <system versioning clause> is specified and a <table period definition> that specifies SYSTEM_TIME is specified, then a system-versioned table is defined.

   d) Otherwise, a regular persistent base table is defined.

2) 09 The degree of *T* is initially set to 0 (zero); the General Rules of Subclause 11.4, "<column definition>", specify the degree of *T* during the definition of the columns of *T*.

3) If <path-resolved user-defined type name> is specified, then:

   a) Let *R* be the structured type identified by the <user-defined type name> simply contained in <path-resolved user-defined type name>.

   b) *R* is the structured type associated with *T*.

4) A table descriptor *TDS* is created that describes *T*. *TDS* includes:

   a) The table name *TN*.

   b) An indication of whether the table is a regular persistent base table, a system-versioned table, a global temporary table, a created local temporary table, or a declared local temporary table.

   c) The column descriptors of every column of *T*, according to the Syntax Rules and General Rules of Subclause 11.4, "<column definition>", applied to the <column definition>s contained in *TEL*, in the order in which they were specified.

   d) If a <table period definition> that specifies SYSTEM_TIME is specified, then a period descriptor that contains SYSTEM_TIME as the name of the period and the names of the SYSTEM_TIME period start and SYSTEM_TIME period end columns.

e) If a <table period definition> that contains an <application time period specification> is specified that contains <application time period name> *ATPN*, then a period descriptor that contains *ATPN* as the name of the period, the names of the *ATPN* period start and *ATPN* period end columns, and *IDCN* as the name of the implicit *ATPN* period constraint.

f) If <typed table clause> is specified, then:

i) An indication that the table is a referenceable table.

ii) An indication that the column at ordinal position 1 (one) is the self-referencing column of *T*. The column descriptor included in *TDS* that describes that column is marked as identifying a self-referencing column.

iii) If *RST* has a system-defined representation, then an indication that the self-referencing column is a system-generated self-referencing column.

iv) If *RST* has a derived representation, then an indication that the self-referencing column is a derived self-referencing column.

v) If *RST* has a user-defined representation, then an indication that the self-referencing column is a user-generated self-referencing column.

g) The table constraint descriptors specified by each <table constraint definition> contained in *TEL*.

h) If a <path-resolved user-defined type name> is specified, then the user-defined type name of *R*.

i) If <subtable clause> is specified, then the table name of the direct supertable of *T* contained in the <subtable clause>.

j) A non-empty set of functional dependencies, according to the rules given in Subclause 4.26, "Functional dependencies".

k) A non-empty set of candidate keys.

l) A preferred candidate key, which may be (but is not required to be) additionally designated the primary key, according to the Rules in Subclause 4.26, "Functional dependencies".

m) If TEMPORARY is specified, then

Case:

i) If ON COMMIT PRESERVE ROWS is specified, then an indication that ON COMMIT PRESERVE ROWS is specified.

ii) Otherwise, an indication that ON COMMIT DELETE ROWS is specified or implied.

n) Case:

i) If <typed table clause> is not specified, then an indication that *T* is insertable-into.

ii) Otherwise,

Case:

1) If the data type descriptor of *R* indicates that *R* is instantiable, then an indication that *T* is insertable-into.

2) Otherwise, an indication that *T* is not insertable-into.

5) In the descriptor of each direct supertable of *T*, *TN* is added to the end of the list of direct subtables.

6) If <subtable clause> is specified, then a set of privilege descriptors is created that defines the privileges SELECT, UPDATE, and REFERENCES for every inherited column of this table to the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <table name> of the direct supertable from which that column was inherited. These privileges are grantable. The grantor for each of these privilege descriptors is set to the special grantor value "_SYSTEM".

7) A set of privilege descriptors is created that define the privileges INSERT, SELECT, UPDATE, DELETE, TRIGGER, and REFERENCES on this table and SELECT, INSERT, UPDATE, and REFERENCES for every <column definition> in the table definition. If OF <path-resolved user-defined type name> is specified, then a table/method privilege descriptor is created on this table for every method of the structured type identified by the <path-resolved user-defined type name> and the table SELECT privilege has the WITH HIERARCHY OPTION. These privileges are grantable.

   The grantor for each of these privilege descriptors is set to the special grantor value "_SYSTEM". The grantee is <authorization identifier> *A*.

8) If <subtable clause> is specified, then let *ST* be the set of supertables of *T*. Let *PDS* be the set of privilege descriptors that defined SELECT WITH HIERARCHY OPTION privilege on a table in *ST*. For every privilege descriptor in *PDS*, with grantee *G*, grantor *A*,

   Case:

   a) If the privilege is grantable, then let *WGO* be "WITH GRANT OPTION".

   b) Otherwise, let *WGO* be the zero-length character string.

   The following <grant statement> is effectively executed without further Access Rule checking:

   ```
   GRANT SELECT ON TN TO G WGO FROM A
   ```

9) The row type *RT* of the table *T* defined by the <table definition> is the set of pairs (<field name>, <data type>) where <field name> is the name of a column *C* of *T* and <data type> is the declared type of *C*. This set of pairs contains one pair for each column of *T*, in the order of their ordinal position in *T*.

10) If <as subquery clause> is specified and WITH DATA is specified, then let *QE* be the <query expression> simply contained in *ASQ*. The following <insert statement> is effectively executed without further Access Rule checking:

    ```
    INSERT INTO TN QE
    ```

Conformance Rules

1) Without Feature T171, "LIKE clause in table definition", conforming SQL language shall not contain a <like clause>.

2) Without Feature F531, "Temporary tables", conforming SQL language shall not contain a <table scope> and shall not reference any global or local temporary table.

3) Without Feature S051, "Create table of type", conforming SQL language shall not contain "OF <path-resolved user-defined type name>".

4) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <column option list> that contains a <scope clause>.

5) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain <reference generation> that does not contain SYSTEM GENERATED.

6) Without Feature S081, "Subtables", conforming SQL language shall not contain a <subtable clause>.

7) Without Feature T172, "AS subquery clause in table definition", conforming SQL language shall not contain an <as subquery clause>.

8) Without Feature T173, "Extended LIKE clause in table definition", a <like clause> shall not contain <like options>.

9) Without Feature T180, "System-versioned tables", conforming SQL language shall not contain "WITH <system versioning clause>" or a <table period definition> that specifies SYSTEM_TIME.

10) Without Feature T181, "Application-time period tables", conforming SQL language shall not contain a <table period definition> that contains an <application time period specification>.

11.4 <column definition>

This Subclause is modified by Subclause 10.4, "<column definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 12.1, "<column definition>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 11.1, "<column definition>", in ISO/IEC 9075-15.

Function

Define a column of a base table.

Format

```
<column definition> ::=
  <column name> [ <data type or domain name> ]
      [ <default clause> | <identity column specification> | <generation clause>
      | <system time period start column specification>
      | <system time period end column specification> ]
      [ <column constraint definition>... ]
      [ <collate clause> ]

<data type or domain name> ::=
    <data type>
  | <domain name>

<system time period start column specification> ::=
  <timestamp generation rule> AS ROW START

<system time period end column specification> ::=
  <timestamp generation rule> AS ROW END

<timestamp generation rule> ::=
  GENERATED ALWAYS

<column constraint definition> ::=
  [ <constraint name definition> ] <column constraint> [ <constraint characteristics> ]

<column constraint> ::=
    NOT NULL
  | <unique specification>
  | <references specification>
  | <check constraint definition>

<identity column specification> ::=
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
      [ <left paren> <common sequence generator options> <right paren> ]

<generation clause> ::=
  <generation rule> AS <generation expression>

<generation rule> ::=
  GENERATED ALWAYS

14 <generation expression> ::=
  <left paren> <value expression> <right paren>
```

Syntax Rules

1) 04 The <column definition> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

2) Case:

   a) If the <column definition> is contained in a <table definition>, then let *T* be the table defined by that <table definition>.

   b) If the <column definition> is contained in a <temporary table declaration>, then let *T* be the table declared by that <temporary table declaration>.

   c) If the <column definition> is contained in an <alter table statement>, then let *T* be the table identified in the containing <alter table statement>.

3) The <column name> in the <column definition> shall not be equivalent to the <column name> of any other column of *T*.

4) Let *A* be the <authorization identifier> that owns *T*.

5) Let *C* be the <column name> of the <column definition>.

6) If <column definition> immediately contains <system time period start column specification> or <system time period end column specification>, then:

   a) <column definition> shall be contained in a <table definition> that specifies a <table period definition> that specifies SYSTEM_TIME or in an <alter table statement> that specifies ADD PERIOD SYSTEM_TIME.

   b) <data type or domain name> shall specify either DATE or a timestamp type.

   c) The <column constraint definition> NOT NULL NOT DEFERRABLE is implicit.

   d) If <column definition> immediately contains <system time period start column specification>, then *C* is a system-time period start column; otherwise, *C* is a system-time period end column.

7) <data type or domain name> shall unambiguously reference either a <data type> or a <domain name>.

8) If <domain name> is specified, then let *D* be the domain identified by the <domain name>.

9) If the descriptor of *D* includes any domain constraint descriptors, then *T* shall be a persistent base table.

10) If <generation clause> *GC* is specified, then:

   a) Let *GE* be the <generation expression> contained in *GC*.

   b) *C* is a generated column.

   c) Every <column reference> contained in *GE* shall reference a base column of *T*.

   d) *GE* shall not contain a potential source of non-determinism.

   e) *GE* shall not generally contain a <routine invocation> whose subject routine possibly reads SQL-data.

   f) *GE* shall not contain a <query expression>.

11) If <generation clause> is omitted, then either <data type> or <domain name> shall be specified.

12) Case:

   a) If <column definition> immediately contains <domain name>, then it shall not also immediately contain <collate clause>.

b) Otherwise, <collate clause> shall not be both specified in <data type> and immediately contained in <column definition>. If <collate clause> is immediately contained in <column definition>, then it is equivalent to specifying an equivalent <collate clause> in <data type>.

13) The declared type of the column is

Case:

a) If <data type> is specified, then that data type. If <generation clause> is also specified, then the declared type of <generation expression> shall be assignable to the declared type of the column.

b) If <domain name> is specified, then the declared type of *D*. If <generation clause> is also specified, then the declared type of <generation expression> shall be assignable to the declared type of the column.

c) If <generation clause> is specified, then the declared type of *GE*.

14) If *T* is a system-versioned table, then the declared type of the column shall not be a reference type.

15) If a <data type> is specified, then:

a) Let *DT* be the <data type>.

b) If *DT* specifies CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT and does not specify a <character set specification>, then the <character set specification> specified or implicit in the <schema character set specification> of the <schema definition> that created the schema identified by the <schema name> immediately contained in the <table name> of the containing <table definition> or <alter table statement> is implicit.

16) If <identity column specification> *ICS* is specified, then:

a) Case:

i) If the declared type of the column being defined is a distinct type *DIST*, then the source type of *DIST* shall be exact numeric with scale 0 (zero). Let *ICT* be the source type of *DIST*.

ii) Otherwise, the declared type of the column being defined shall be exact numeric with scale 0 (zero). Let *ICT* be the declared type of the column being defined.

b) Let *SGO* be the <common sequence generator options>.

c) The Syntax Rules of Subclause 9.36, "Creation of a sequence generator", are applied with *SGO* as *OPTIONS* and *ICT* as *DATA TYPE*.

d) The <column constraint definition> NOT NULL NOT DEFERRABLE is implicit.

17) If a <column constraint definition> is specified, then let *CND* be the <constraint name definition> if one is specified and let *CND* be the zero-length character character string otherwise; let *CA* be the <constraint characteristics> if specified and let *CA* be the zero-length character string otherwise. The <column constraint definition> is equivalent to a <table constraint definition> as follows.

Case:

a) If a <column constraint definition> is specified that contains the <column constraint> NOT NULL, then it is equivalent to the following <table constraint definition>:

```
CND CHECK ( C IS NOT NULL ) CA
```

b) If a <column constraint definition> is specified that contains a <unique specification> *US*, then it is equivalent to the following <table constraint definition>:

CND US (C) CA

NOTE 540 — The <unique specification> is defined in Subclause 11.7, "<unique constraint definition>".

c) If a <column constraint definition> is specified that contains a <references specification> *RS*, then it is equivalent to the following <table constraint definition>:

CND FOREIGN KEY (*C*) *RS CA*

NOTE 541 — The <references specification> is defined in Subclause 11.8, "<referential constraint definition>".

d) If a <column constraint definition> is specified that contains a <check constraint definition> *CCD*, then it is equivalent to the following <table constraint definition>:

CND CCD CA

Each column reference directly contained in the <search condition> shall reference column *C*.

18) [14] The schema identified by the explicit or implicit <schema name> of the <domain name> shall include the descriptor of *D*.

Access Rules

1) If <domain name> is specified, then the applicable privileges for *A* shall include USAGE on *D*.

General Rules

1) A <column definition> defines a column in a table.

2) If the <column definition> specifies <data type>, then a data type descriptor is created that describes the declared type of the column being defined.

3) The degree of the table *T* being defined in the containing <table definition> or <temporary table declaration>, or being altered by the containing <alter table statement> is increased by 1 (one).

4) If <identity column specification> is specified, then the General Rules of Subclause 9.36, "Creation of a sequence generator", are applied with *SGO* as *OPTIONS* and *ICT* as *DATA TYPE*; let the sequence generator descriptor *DSG* be the *SEQGENDESC* returned from the application of those General Rules.

5) A column descriptor is created that describes the column being defined. The column descriptor includes:

a) *C*, the name of the column.

b) Case:

i) If the <column definition> specifies a <data type> or a <generation clause>, then the data type descriptor of the declared type of the column.

ii) Otherwise, the <domain name> that identifies the domain of the column.

c) The ordinal position of the column.

NOTE 542 — The ordinal position of the column is equal to the degree of *T* at the time this <column definition> is being processed.

d) The nullability characteristic of the column, determined according to the rules in Subclause 4.15, "Columns, fields, and attributes".

NOTE 543 — Both <column constraint definition>s and <table constraint definition>s must be analyzed to determine the nullability characteristics of all columns.

e) If <default clause> is specified, then the <default option>.

f) If <identity column specification> is specified, then:

i) An indication that the column is an identity column.

ii) If ALWAYS is specified, then an indication that values are always generated.

iii) If BY DEFAULT is specified, then an indication that values are generated by default.

iv) *DSG*.

g) If <system time period start column specification> is specified, then an indication that the column is a system-time period start column.

h) If <system time period end column specification> is specified, then an indication that the column is a system-time period end column.

i) If <system time period start column specification> or <system time period end column specification> is specified, then an indication that transaction timestamp values are always generated.

j) Case:

i) If the <column definition> specifies a <generation clause>, then an indication that the column is "ALWAYS" generated and *GE*.

ii) Otherwise, an indication that the column is "NEVER" generated.

k) An indication that the column is updatable.

l) If a <column constraint definition> is specified that contains the <column constraint> NOT NULL, then:

i) An indication that the column is defined as NOT NULL.

ii) The constraint name of the associated table constraint definition.

NOTE 544 — The associated table constraint definition is the <table constraint definition> specified in SR 17)a) of this Subclause.

6) If <domain name> is specified, then, for every domain constraint descriptor *DCD* included in the domain descriptor of *D*, let *DSC* be the template <search condition> included in *DCD*. Let *CSC* be a copy of *DSC* in which every instance of the <general value specification> VALUE is replaced by *C*. A domain constraint usage descriptor is created and added to the set of domain constraint usage descriptors included in *DCD*. The domain constraint usage descriptor created includes:

a) The name of the applicable column.

b) The constraint name included in *DCD*, as the name of the applicable domain constraint.

c) The indication included in *DCD* as to whether the constraint is deferrable.

d) The initial constraint mode included in *DCD*.

e) The applicable <search condition>

```
( SELECT EVERY ( CSC )
  FROM T )
```

NOTE 545 — This is a <scalar subquery> of declared type BOOLEAN.

Conformance Rules

1) Without Feature F692, "Extended collation support", conforming SQL language shall not contain a <column definition> that immediately contains a <collate clause>.

2) Without Feature T174, "Identity columns", conforming SQL language shall not contain an <identity column specification>.

3) Without Feature T175, "Generated columns", conforming SQL language shall not contain a <generation clause>.

4) 14 15 Without Feature T180, "System-versioned tables", conforming SQL language shall not contain <system time period start column specification> or <system time period end column specification>.

11.5 <default clause>

This Subclause is modified by Subclause 10.5, "<default clause>", in ISO/IEC 9075-4.

Function

Specify the default for a column, domain, or attribute.

Format

```
<default clause> ::=
  DEFAULT <default option>

<default option> ::=
    <literal>
  | <datetime value function>
  | USER
  | CURRENT_USER
  | CURRENT_ROLE
  | SESSION_USER
  | SYSTEM_USER
  | CURRENT_CATALOG
  | CURRENT_SCHEMA
  | CURRENT_PATH
  | <implicitly typed value specification>
```

Syntax Rules

1) 04 The subject data type of a <default clause> is the data type specified in the descriptor identified by the containing <column definition>, <domain definition>, <attribute definition>, <alter column definition>, or <alter domain statement>.

2) If USER is specified, then CURRENT_USER is implicit.

3) Case:

   a) If the subject data type of the <default clause> is a collection type or a distinct type whose source type is a collection type, then <default option> shall specify <implicitly typed value specification>.

   b) If the subject data type of the <default clause> is a user-defined type, a reference type, or a row type, then <default option> shall specify <null specification>.

4) Case:

   a) If a <literal> is specified, then

      Case:

      i) If the subject data type is character string, then the <literal> shall be a <character string literal>. If the length of the subject data type is fixed, then the length in characters of the <character string literal> shall not be greater than the length of the subject data type. If the length of the subject data type is variable, then the length in characters of the <character string literal> shall not be greater than the maximum length of the subject data type. The <literal> shall have the same character repertoire as the subject data type.

ii) If the subject data type is binary string, then the <literal> shall be a <binary string literal>. If the length of the subject data type is fixed, then the length in octets of the <binary string literal> shall not be greater than the length of the subject data type. If the length of the subject data type is variable, then the length in octets of the <binary string literal> shall not be greater than the maximum length of the subject data type.

iii) If the subject data type is exact numeric, then the <literal> shall be a <signed numeric literal> that simply contains an <exact numeric literal>. There shall be a representation of the value of the <literal> in the subject data type that does not lose any significant digits.

iv) If the subject data type is approximate numeric, then the <literal> shall be a <signed numeric literal>.

v) If the subject data type is datetime, then the <literal> shall be a <datetime literal> with the same primary datetime fields and the same time zone datetime fields as the subject data type. If SECOND is one of these fields, then the fractional seconds precision of the <datetime literal> shall be less than or equal to the fractional seconds precision of the subject data type.

vi) If the subject data type is interval, then the <literal> shall be an <interval literal> and shall contain the same <interval qualifier> as the subject data type.

vii) If the subject data type is Boolean, then the <literal> shall be a <boolean literal>.

b) If CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, or CURRENT_SCHEMA is specified, then the subject data type shall be character string with character set SQL_IDENTIFIER. If the length of the subject data type is fixed, then its length shall not be less than 128 characters. If the length of the subject data type is variable, then its maximum length shall not be less than 128 characters.

c) If CURRENT_PATH is specified, then the subject data type shall be character string with character set SQL_IDENTIFIER. If the length of the subject data type is fixed, then its length shall not be less than 1031 characters. If the length of the subject data type is variable, then its maximum length shall not be less than 1031 characters.

d) If <datetime value function> is specified, then the subject data type shall be datetime with the same declared datetime data type of the <datetime value function>.

e) If <empty specification> is specified, then the subject data type shall be a collection type or a distinct type whose source type is a collection type. If the <empty specification> specifies ARRAY, then the subject data type shall be an array type or a distinct type whose source type is an array type. If the <empty specification> specifies MULTISET, then the subject data type shall be a multiset type or a distinct type whose source type is a multiset type.

Access Rules

None.

General Rules

1) The default value inserted in the column descriptor, if the <default clause> is to apply to a column, or in the domain descriptor, if the <default clause> is to apply to a domain, or in the attribute descriptor, if the <default clause> is to apply to an attribute, is the <default option>.

2) The value specified by a <default option> is

Case:

a) If the <default option> contains a <literal>, then

Case:

i) If the subject data type is numeric, then the numeric value of the <literal>.

ii) If the subject data type is variable-length character string, then the value of the <literal>.

iii) If the subject data type is fixed-length character string, then the value of the <literal>, extended as necessary on the right with <space>s to the length in characters of the subject data type.

iv) If the subject data type is fixed-length binary string, then the value of the <literal>, extended as necessary to the right with X'00's to the length in octets of the subject data type.

v) If the subject data type is variable-length binary string, then the value of the <literal>.

vi) If the subject data type is datetime or interval, then the value of the <literal>.

vii) If the subject data type is Boolean, then the value of the <literal>.

b) If the <default option> specifies CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH, then

Case:

i) If the subject data type is variable-length character string, then the value obtained by an evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH at the time that the default value is required.

ii) If the subject data type is fixed-length character string, then the value obtained by an evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH at the time that the default value is required, extended as necessary on the right with <space>s to the length in characters of the subject data type.

c) If the <default option> contains a <datetime value function>, then the value of an evaluation of the <datetime value function>.

d) If the <default option> specifies <empty specification>, then an empty collection.

3) When a site *S* is set to its default value,

Case:

a) If the descriptor of *S* indicates that it represents a column of which some underlying column is an identity column, a generated column, a system-time period start column, or a system-time period end column, then *S* is marked as *unassigned*.

NOTE 546 — The notion of a site being unassigned is only for definitional purposes in this document. It is not a state that can persist so as to be visible in SQL-data. The treatment of unassigned sites is given in Subclause 15.11, "Effect of inserting tables into base tables", and Subclause 15.14, "Effect of replacing rows in base tables".

b) If the descriptor of *S* includes a <default option>, then *S* is set to the value specified by that <default option>.

c) If the descriptor of *S* includes a <domain name> that identifies a domain descriptor that includes a <default option>, then *S* is set to the value specified by that <default option>.

d) If the default value is for a column *C* of a candidate row for insertion into or update of a derived table *DT* and *C* has a single counterpart column *CC* in a leaf generally underlying table of *DT*, then *S* is set to the default value of *CC*, which is obtained by applying the General Rules of this Subclause.

e) Otherwise, *S* is set to the null value.

NOTE 547 — If <default option> specifies CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, or CURRENT_PATH, then the "value in the column descriptor" will effectively be the text of the <default option>, whose evaluation occurs at the time that the default value is required.

4) If the <default clause> is contained in an <SQL schema statement> and character representation of the <default option> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — default value too long for information schema (0100B)*.

NOTE 548 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <default option> that contains CURRENT_PATH.

2) Without Feature F321, "User authorization", conforming SQL language shall not contain a <default option> that contains CURRENT_USER, SESSION_USER, or SYSTEM_USER.

NOTE 549 — Although CURRENT_USER and USER are semantically the same, without Feature F321, "User authorization", CURRENT_USER must be specified as USER.

3) Without Feature T332, "Extended roles", conforming SQL language shall not contain a <default option> that contains CURRENT_ROLE.

4) Without Feature F762, "CURRENT_CATALOG", conforming SQL language shall not contain a <default option> that contains CURRENT_CATALOG.

5) Without Feature F763, "CURRENT_SCHEMA", conforming SQL language shall not contain a <default option> that contains CURRENT_SCHEMA.

11.6 <table constraint definition>

Function

Specify an integrity constraint.

Format

```
<table constraint definition> ::=
  [ <constraint name definition> ] <table constraint>
      [ <constraint characteristics> ]

<table constraint> ::=
    <unique constraint definition>
  | <referential constraint definition>
  | <check constraint definition>
```

Syntax Rules

1) If <constraint characteristics> is not specified, then INITIALLY IMMEDIATE NOT DEFERRABLE ENFORCED is implicit.

2) If <constraint name definition> is specified and its <constraint name> contains a <schema name>, then that <schema name> shall be equivalent to the explicit or implicit <schema name> of the <table name> of the table identified by the containing <table definition> or <alter table statement>.

3) If <constraint name definition> is not specified, then a <constraint name definition> that contains an implementation-dependent (UV100) <constraint name> is implicit. The assigned <constraint name> shall obey the Syntax Rules of an explicit <constraint name>.

4) Let *S* be the schema identified by the explicit or implicit <schema name> of the <constraint name>. *S* shall not include a constraint descriptor whose constraint name is <constraint name>.

5) If <unique constraint definition> is specified, then <constraint characteristics> shall not specify a <constraint enforcement>.

Access Rules

None.

General Rules

1) A <table constraint definition> defines a table constraint.

2) A table constraint descriptor is created that describes the table constraint being defined. The table constraint descriptor includes:

   a) The <constraint name> contained in the explicit or implicit <constraint name definition>.

   b) An indication of whether the constraint is deferrable or not deferrable.

   c) An indication of whether the initial constraint mode of the constraint is deferred or immediate.

   d) An indication of whether the constraint is enforced or not enforced.

e) The applicable <search condition> as specified in the General Rules applicable to the particular kind of <table constraint> contained in the <table constraint definition>, and all additional contents specified by those General Rules.

Conformance Rules

None.

11.7 <unique constraint definition>

This Subclause is modified by Subclause 10.4, "<unique constraint definition>", in ISO/IEC 9075-9.

Function

Specify a uniqueness constraint for a table.

Format

```
<unique constraint definition> ::=
    <unique specification> [ <unique null treatment>
        ] <left paren> <unique column list> [ <comma> <without overlap specification> ]
    <right paren>
  | UNIQUE <left paren> VALUE <right paren>

<unique specification> ::=
    UNIQUE
  | PRIMARY KEY

<unique column list> ::=
  <column name list>

<without overlap specification> ::=
  <application time period name> WITHOUT OVERLAPS
```

Syntax Rules

1) 09 Each column identified by a <column name>in the <unique column list> is an operand of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

2) Let *T* be the table identified by the containing <table definition> or <alter table statement>. Let *TN* be the <table name> of *T*.

3) If <without overlap specification> is specified, then:

   a) Let *ATPN* be the <application time period name> contained in the <without overlap specification>.

   b) The table descriptor of *T* shall include an *ATPN* period descriptor.

4) Case:

   a) If *T* is a system-versioned table, then let *TNN* be

      TN FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP

   b) Otherwise, let *TNN* be *TN*.

5) If <unique null treatment> is specified, then <unique specification> shall not be PRIMARY KEY. If <unique null treatment> is not specified, then an implementation-defined (ID106) <unique null treatment> is implicit. Let *UNT* be the implicit or explicit <unique null treatment>.

6) If <unique column list> *UCL* is specified, then:

   a) Each <column name> in the <unique column list> shall identify a column of *T*, and the same column shall not be identified more than once.

b) If the descriptor of *T* includes a system-time period descriptor, then *UCL* shall not include a system-time period start column of *T* or a system-time period end column of *T*.

c) The set of columns in the <unique column list> shall be distinct from the unique columns of every other unique constraint descriptor that is included in the base table descriptor of *T*.

d) Case:

i) If <without overlap specification> is specified, then let *STARTCOL* be the *ATPN* period start column of *T*, let *ENDCOL* be the *ATPN* period end column of *T*, let *DT* be the declared type of *STARTCOL*, and let *SC1* be the <search condition>:

```
UNIQUE UNT ( SELECT UCL, TIMEPOINT FROM TNN AS Y,
                    UNNEST(EXPAND(STARTCOL, ENDCOL)) AS X(TIMEPOINT))
```

where *EXPAND* is the <routine name> of an SQL-invoked function that returns a value of SET(*DT*) type consisting of all values of *DT* that are greater than or equal to the value of *STARTCOL* and less than the value of *ENDCOL*, and *TIMEPOINT* is an <identifier> not equivalent to the <column name> of any column of *T*.

NOTE 550 — The use of the SQL-invoked function invocation above is only for definitional purposes in this document.

ii) Otherwise, let *SC1* be the <search condition>:

```
UNIQUE UNT ( SELECT UCL
             FROM TNN )
```

e) Case:

i) If the <unique specification> specifies PRIMARY KEY, then let *SC* be the <search condition>:

```
SC1
  AND
( SELECT EVERY ( UCL IS NOT NULL )
  FROM TNN )
```

NOTE 551 — The second operand of AND in this expression is a <scalar subquery> of declared type BOOLEAN.

ii) Otherwise, let *SC* be *SC1*.

7) If UNIQUE (VALUE) is specified, then let *SC* be the <search condition>:

```
UNIQUE UNT ( SELECT TNN.* FROM TNN )
```

8) If the <unique specification> specifies PRIMARY KEY, then for each column descriptor identified by a <column name> in the explicit or implicit <unique column list> for which the nullability characteristic is not known not nullable, the nullability characteristic of the column descriptor is set to known not nullable.

9) A <table definition> shall specify at most one implicit or explicit <unique constraint definition> that specifies PRIMARY KEY.

10) If a <unique constraint definition> that specifies PRIMARY KEY is contained in an <add table constraint definition>, then the table identified by the <table name> immediately contained in the containing <alter table statement> shall not have a unique constraint that was defined by a <unique constraint definition> that specified PRIMARY KEY.

Access Rules

None.

General Rules

1) A <unique constraint definition> defines a unique constraint.

2) The applicable <search condition> included in the table constraint descriptor created as a result of execution of the containing <table constraint definition> is *SC*. This descriptor additionally includes:

   a) The <unique specification>, indicating whether the constraint is defined with PRIMARY KEY or UNIQUE.

   b) The names of the unique columns specified in the <unique column list>.

   c) If <without overlap specification> is specified, then *ATPN*.

   d) The explicit or implicit <unique null treatment>.

3) The unique constraint is not satisfied if and only if

   ```
   EXISTS ( SELECT * FROM TNN WHERE NOT ( SC ) )
   ```

   is *True*.

Conformance Rules

1) Without Feature S291, "Unique constraint on entire row", conforming SQL language shall not contain UNIQUE(VALUE).

2) Without Feature T591, "UNIQUE constraints of possibly null columns", in conforming SQL language, if UNIQUE is specified, then the <column definition> for each column whose <column name> is contained in the <unique column list> shall contain NOT NULL.

   NOTE 552 — The Conformance Rules of Subclause 9.12, "Grouping operations", also apply.

3) Without Feature T181, "Application-time period tables", conforming SQL language shall not contain <without overlap specification>.

4) Without Feature F292, "UNIQUE null treatment", conforming SQL language shall not contain an explicit <unique null treatment>.

11.8 <referential constraint definition>

Function

Specify a referential constraint.

Format

```
<referential constraint definition> ::=
  FOREIGN KEY <left paren> <referencing column list>
      [ <comma> <referencing period specification> ] <right paren>
      <references specification>

<references specification> ::=
  REFERENCES <referenced table and columns>
      [ MATCH <match type> ] [ <referential triggered action> ]

<match type> ::=
    FULL
  | PARTIAL
  | SIMPLE

<referencing column list> ::=
  <column name list>

<referencing period specification> ::=
  PERIOD <application time period name>

<referenced table and columns> ::=
  <table name> [ <left paren> <referenced column list>
      [ <comma> <referenced period specification> ] <right paren> ]

<referenced column list> ::=
  <column name list>

<referenced period specification> ::=
  PERIOD <application time period name>

<referential triggered action> ::=
    <update rule> [ <delete rule> ]
  | <delete rule> [ <update rule> ]

<update rule> ::=
  ON UPDATE <referential action>

<delete rule> ::=
  ON DELETE <referential action>

<referential action> ::=
    CASCADE
  | SET NULL
  | SET DEFAULT
  | RESTRICT
  | NO ACTION
```


Syntax Rules

1) If <match type> is not specified, then SIMPLE is implicit.

2) If <referencing period specification> is specified, then:

a) <referenced table and columns> shall immediately contain a <referenced column list> and a <referenced period specification>.

b) <referential action> shall not specify CASCADE, SET NULL, or SET DEFAULT.

3) Let *referencing table* be the table identified by the containing <table definition> or <alter table statement>. Let *referenced table* be the table identified by the <table name> in the <referenced table and columns>. Let *referencing columns* be the column or columns identified by the <referencing column list> and let *referencing column* be one such column.

4) Case:

a) If the <referenced table and columns> specifies a <referenced column list>, then there shall be a one-to-one correspondence between the set of <column name>s contained in that <referenced column list> and the set of <column name>s contained in the <unique column list> of a unique constraint of the referenced table such that corresponding <column name>s are equivalent. Let *referenced columns* be the column or columns identified by that <referenced column list> and let *referenced column* be one such column. Each referenced column shall identify a column of the referenced table and the same column shall not be identified more than once.

b) Otherwise, the table descriptor of the referenced table shall include a unique constraint *UC* that specifies PRIMARY KEY. The table constraint descriptor of *UC* shall not include an application time period name. Let *referenced columns* be the column or columns identified by the unique columns in that unique constraint and let *referenced column* be one such column. The <referenced table and columns> shall be considered to implicitly specify a <referenced column list> that is identical to that <unique column list>.

5) The table constraint descriptor describing the <unique constraint definition> whose <unique column list> identifies the referenced columns shall indicate that the unique constraint is not deferrable.

6) The referenced table shall be a base table.

Case:

a) If the referencing table is a persistent base table, then the referenced table shall be a persistent base table.

b) If the referencing table is a global temporary table, then the referenced table shall be a global temporary table.

c) If the referencing table is a created local temporary table, then the referenced table shall be either a global temporary table or a created local temporary table.

d) If the referencing table is a declared local temporary table, then the referenced table shall be either a global temporary table, a created local temporary table or a declared local temporary table.

7) If the referenced table is a temporary table with ON COMMIT DELETE ROWS specified, then the referencing table shall specify ON COMMIT DELETE ROWS.

8) Each referencing column shall identify a column of the referencing table, and the same column shall not be identified more than once.

9) Each referencing column and its corresponding referenced column are operands of a grouping operation. The Syntax Rules and Conformance Rules of Subclause 9.12, "Grouping operations", apply.

10) If the table descriptor of the referencing table includes a system-time period descriptor, then none of the referencing columns shall identify a system-time period start column of the referencing table or a system-time period end column of the referencing table.

11) The <referencing column list> shall contain the same number of <column name>s as the <referenced column list>. The *i*-th column identified in the <referencing column list> corresponds to the *i*-th column identified in the <referenced column list>. The declared type of each referencing column shall be comparable to the declared type of the corresponding referenced column. There shall not be corresponding constituents of the declared type of a referencing column and the declared type of the corresponding referenced column such that one constituent is datetime with time zone and the other is datetime without time zone.

12) If a <referential constraint definition> does not specify any <update rule>, then an <update rule> with a <referential action> of NO ACTION is implicit.

13) If a <referential constraint definition> does not specify any <delete rule>, then a <delete rule> with a <referential action> of NO ACTION is implicit.

14) If any referencing column is a generated column, then:

    a) <referential action> shall not specify SET NULL or SET DEFAULT.

    b) <update rule> shall not specify ON UPDATE CASCADE.

15) Let *T* be the referenced table. The schema identified by the explicit or implicit <schema name> of the <table name> shall include the descriptor of *T*.

16) Let *TN* be the <table name> of *T*.

    Case:

    a) If *T* is a system-versioned table, then let *TNN* be

       TN `FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP`

    b) Otherwise, let *TNN* be *TN*.

17) Let *U* be the referencing table. Let *UN* be the <table name> of *U*.

    Case:

    a) If *U* is a system-versioned table, then let *UNN* be

       UN `FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP`

    b) Otherwise, let *UNN* be *UN*.

18) Let *UCL* be the <referencing column list>. Let *TCL* be the implicit or explicit <referenced column list>.

    a) Case:

       i) If <referencing period specification> *RPC* is specified, then:

          1) Let *RPP* be the <referenced period specification> immediately contained in <referenced table and columns>.

          2) Let *CATPN* be the <application time period name> contained in *RPC*.

          3) Let *PATPN* be the <application time period name> contained in *RPP*.

          4) The table descriptor of *T* shall include a *PATPN* period descriptor.

          5) The table descriptor of *U* shall include a *CATPN* period descriptor.

6) The table constraint descriptor describing the <unique constraint definition> whose <unique column list> identifies the referenced columns shall include *PATPN*.

7) Let *TSTARTCOL* be the *PATPN* period start column of *T*. Let *TENDCOL* be the *PATPN* period end column of *T*. Let *TDT* be the declared type of *TSTARTCOL*. Let *USTARTCOL* be the *CATPN* period start column of *U*. Let *UENDCOL* be the *CATPN* period end column of *U*. Let *UDT* be the declared type of *USTARTCOL*.

8) Syntax Rules of Subclause 9.30, "Data type identity", are applied with *TDT* as *TYPE1* and *UDT* as *TYPE2*.

9) Let *TTS* be the <table subquery>

```
( SELECT TCL, TIMEPOINT1
  FROM TNN AS Y,
       UNNEST(EXPAND(TSTARTCOL, TENDCOL)) AS X(TIMEPOINT1) )
```

where *EXPAND* is the <routine name> of an SQL-invoked function that returns a value of SET(*TDT*) type consisting of all values of *TDT* that are greater than or equal to the value of *TSTARTCOL* and less than the value of *TENDCOL*, and *TIMEPOINT1* is an <identifier> not equivalent to the <column name> of any column of *T*.

NOTE 553 — The use of the SQL-invoked function invocation above is only for definitional purposes in this document.

10) Let *UNNN* be the <derived table>

```
( SELECT Z.*, TIMEPOINT2
  FROM UNN AS Z,
       UNNEST(EXPAND(USTARTCOL, UENDCOL)) AS X(TIMEPOINT2)
) AS Y
```

where *EXPAND* is the <routine name> of an SQL-invoked function that returns a value of SET(*UDT*) type consisting of all values of *UDT* that are greater than or equal to the value of *USTARTCOL* and less than the value of *UENDCOL*, and *TIMEPOINT2* is an <identifier> not equivalent to the <column name> of any column of *U*.

NOTE 554 — The use of the SQL-invoked function invocation above is only for definitional purposes in this document.

11) Let *UCLL* be the <column name list>

```
UCL, TIMEPOINT2
```

ii) Otherwise:

1) Let *TTS* be the <table subquery>

```
( SELECT TCL FROM TNN )
```

2) Let *UNNN* be the <derived table>

```
( SELECT Z.* FROM UNN AS Z) AS Y
```

3) Let *UCLL* be the <column name list>

```
UCL
```

b) Let *MP* be

Case:

i) If SIMPLE is specified or implicit, then

```
UCLL MATCH SIMPLE TTS
```

ii) If PARTIAL is specified or implicit, then

```
UCLL MATCH PARTIAL TTS
```

iii) If FULL is specified or implicit, then

```
UCLL MATCH FULL TTS
```

c) Let *SC* be

```
( SELECT EVERY ( MP )
  FROM UNNN )
```

NOTE 555 — This is a <scalar subquery> of declared type BOOLEAN.

Access Rules

1) The applicable privileges for the owner of *T* shall include REFERENCES for each referenced column.

General Rules

1) A <referential constraint definition> defines a referential constraint.

2) The applicable <search condition> included in the table constraint descriptor created as a result of executing the containing <table constraint definition> is *SC*. This descriptor additionally includes:

a) A list of the names of the referencing columns specified in the <referencing column list>, in an implementation-defined (IS001) order.

b) The name of the referenced table specified in the <referenced table and columns>.

c) A list of the names of the referenced columns specified in the <referenced table and columns>, ordered such that each element is in the same position as that of its corresponding referencing column name in the list of names of referencing columns.

d) If <referencing period specification> is specified, then *CATPN* as name of the referencing period.

e) If <referenced period specification> is specified, then *PATPN* as name of the referenced period.

f) The explicit or implicit <match type>.

g) The <referential triggered action>, comprising the explicit or implicit <delete rule> and the explicit or implicit <update rule>.

Conformance Rules

1) Without Feature T191, “Referential action RESTRICT ”, conforming SQL language shall not contain a <referential action> that contains RESTRICT.

2) Without Feature F741, “Referential MATCH types”, conforming SQL language shall not contain a <references specification> that contains MATCH.

3) Without Feature F191, “Referential delete actions”, conforming SQL language shall not contain a <delete rule>.

4) Without Feature F701, “Referential update actions”, conforming SQL language shall not contain an <update rule>.

5) Without Feature T201, “Comparable data types for referential constraints”, conforming SQL language shall not contain a <referencing column list> in which the data type of each referencing column is not the same as the data type of the corresponding referenced column.

NOTE 556 — The Conformance Rules of Subclause 9.12, “Grouping operations”, also apply.

6) Without Feature T181, “Application-time period tables”, conforming SQL language shall not contain a <referencing period specification>.

7) Without Feature T181, “Application-time period tables”, conforming SQL language shall not contain a <referenced period specification>.

11.9 <check constraint definition>

This Subclause is modified by Subclause 10.6, "<check constraint definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.5, "<check constraint definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.2, "<check constraint definition>", in ISO/IEC 9075-14.

Function

Specify a condition for the SQL-data.

Format

```
14 <check constraint definition> ::=
     CHECK <left paren> <search condition> <right paren>
```

Syntax Rules

1) 04 The <search condition> shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, an <embedded variable specification>, or a <column reference> that references a system-time period start column or a system-time period end column.

2) The <search condition> shall not contain a <set function specification> that is not contained in a <query expression>.

3) If <check constraint definition> is contained in a <table definition> or <alter table statement>, then let *T* be the table identified by the containing <table definition> or <alter table statement>.

   Case:

   a) If *T* is a persistent base table, or if the <check constraint definition> is contained in a <domain definition> or <alter domain statement>, then no <table reference> generally contained in the <search condition> shall reference a temporary table.

   b) If *T* is a global temporary table, then no <table reference> generally contained in the <search condition> shall reference a table other than a global temporary table.

   c) If *T* is a created local temporary table, then no <table reference> generally contained in the <search condition> shall reference a table other than either a global temporary table or a created local temporary table.

   d) If *T* is a declared local temporary table, then no <table reference> generally contained in the <search condition> shall reference a persistent base table.

4) If the <check constraint definition> is contained in a <table definition> that defines a temporary table and specifies ON COMMIT PRESERVE ROWS or a <temporary table declaration> that specifies ON COMMIT PRESERVE ROWS, then the <search condition> shall not contain a reference to a temporary table defined by a <table definition> or a <temporary table declaration> that specifies ON COMMIT DELETE ROWS.

5) The <boolean value expression> that is simply contained in the <search condition> shall be retrospectively deterministic.

   NOTE 557 — "retrospectively deterministic" is defined in Subclause 6.46, "<boolean value expression>".

6) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

7) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an external routine that possibly reads SQL-data.

8) Let *TN* be the <table name> of *T*.

Case:

a) If *T* is a system-versioned table, then let *TNN* be

```
TN FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP
```

b) Otherwise, let *TNN* be *TN*.

9) 09 14 Let *SCC* be the <search condition> immediately contained in the <check constraint definition>. Let *SC* be

```
( SELECT EVERY ( SCC )
  FROM TNN )
```

NOTE 558 — This is a <scalar subquery> of declared type BOOLEAN.

Access Rules

None.

General Rules

1) A <check constraint definition> defines a check constraint.

2) If the character representation of the <search condition> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — search condition too long for information schema (01009)*.

NOTE 559 — The Information Schema is defined in ISO/IEC 9075-11.

3) The applicable <search condition> included in the table constraint descriptor created as a result of executing the containing <table constraint definition> is *SC*.

Conformance Rules

1) Without Feature F671, "Subqueries in CHECK constraints", conforming SQL language shall not contain a <search condition> contained in a <check constraint definition> that contains a <query expression>.

2) Without Feature F672, "Retrospective CHECK constraints", conforming SQL language shall not contain a <check constraint definition> that contains a potential source of non-determinism.

3) 14 Without Feature F673, "Reads SQL-data routine invocations in CHECK constraints", conforming SQL language shall not contain a <check constraint definition> that contains a <search condition> that generally contains a <routine invocation> whose subject routine is an SQL routine that possibly reads SQL-data.

11.10 <alter table statement>

Function

Change the definition of a table.

Format

```
<alter table statement> ::=
  ALTER TABLE <table name> <alter table action>

<alter table action> ::=
    <add column definition>
  | <alter column definition>
  | <drop column definition>
  | <add table constraint definition>
  | <alter table constraint definition>
  | <drop table constraint definition>
  | <add table period definition>
  | <drop table period definition>
  | <add system versioning clause>
  | <drop system versioning clause>
```

Syntax Rules

1) Let *T* be the table identified by the <table name>.

2) The schema identified by the explicit or implicit <schema name> of the <table name> shall include the descriptor of *T*.

3) The scope of the <table name> is the entire <alter table statement>.

4) *T* shall be a base table.

5) *T* shall not be a declared local temporary table.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by <table name>.

General Rules

1) The base table descriptor of *T* is modified as specified by <alter table action>.

2) If <add column definition> or <drop column definition> is specified, then the row type *RT* of *T* is the set of pairs (<field name>, <data type>) where <field name> is the name of a column *C* of *T* and <data type> is the declared type of *C*. This set of pairs contains one pair for each column of *T* in the order of their ordinal position in *T*.

Conformance Rules

None.

11.11 <add column definition>

Function

Add a column to a table.

Format

```
<add column definition> ::=
  ADD [ COLUMN ] <column definition>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> immediately contained in the containing <alter table statement>.

2) *T* shall not be a referenceable table or a system-versioned table.

3) If <column definition> contains <identity column specification>, then the table descriptor of *T* shall not include a column descriptor of an identity column.

4) If the table descriptor of *T* contains a period descriptor whose period name is *ATPN*, then the <column name> immediately contained in the <column definition> shall not be equivalent to *ATPN*.

Access Rules

None.

General Rules

1) The column defined by the <column definition> is added to *T*.

2) Let *C* be the column added to *T*.

   Case:

   a) If *C* is a generated column, then let *TN* be the <table name> immediately contained in the containing <alter table statement>, let *CN* be the <column name> immediately contained in <column definition>, and let *GE* be the generation expression included in the column descriptor of *C*. The following <update statement: searched> is executed without further Syntax Rule or Access Rule checking:

      ```
      UPDATE TN SET CN = GE
      ```

   b) Otherwise, *C* is a base column.

      Case:

      i) If *C* is an identity column, then, for each row in *T*, let *CS* be the site corresponding to *C*. The General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are applied with the sequence generator descriptor included in the column descriptor of *C* as *SEQUENCE*; let *NV* be the *RESULT* returned from the application of those General Rules.

         Case:

1) If the declared type of *C* is a distinct type *DIST*, then let *CNV* be *DIST*(*NV*).

2) Otherwise, let *CNV* be *NV*.

The General Rules of Subclause 9.2, "Store assignment", are applied with *CS* as *TARGET* and *CNV* as *VALUE*

ii) Otherwise, every value in *C* is the default value for *C*.

NOTE 560 — The default value of a column is defined in Subclause 11.5, "<default clause>".

NOTE 561 — The addition of a column to a table has no effect on any existing <query expression> included in a view descriptor, <triggered action> included in a trigger descriptor, or <search condition> included in a constraint descriptor because any implicit column references in these descriptor elements are syntactically substituted by explicit column references under the Syntax Rules of Subclause 7.16, "<query specification>". Furthermore, by implication (from the lack of any General Rules to the contrary), the meaning of a column reference is never retroactively changed by the addition of a column subsequent to the invocation of the <SQL schema statement> containing that column reference.

3) For every table privilege descriptor that specifies *T* and a privilege of SELECT, UPDATE, INSERT or REFERENCES, a new column privilege descriptor is created that specifies *T*, the same action, grantor, and grantee, and the same grantability, and specifies the <column name> of the <column definition>.

4) In all other respects, the specification of a <column definition> in an <alter table statement> has the same effect as specification of the <column definition> in the <table definition> for *T* would have had.

NOTE 562 — In particular, the degree of *T* is increased by 1 (one) and the ordinal position of that column is equal to the new degree of *T* as specified in the General Rules of Subclause 11.4, "<column definition>".

Conformance Rules

None.

11.12 <alter column definition>

Function

Change a column and its definition.

Format

```
<alter column definition> ::=
  ALTER [ COLUMN ] <column name> <alter column action>

<alter column action> ::=
    <set column default clause>
  | <drop column default clause>
  | <set column not null clause>
  | <drop column not null clause>
  | <add column scope clause>
  | <drop column scope clause>
  | <alter column data type clause>
  | <alter identity column specification>
  | <drop identity property clause>
  | <drop column generation expression clause>
```

Syntax Rules

1) Let *T* be the table identified in the containing <alter table statement>.

2) *T* shall not be a system-versioned table.

3) Let *C* be the column identified by the <column name>.

4) *C* shall be a column of *T*.

5) If *C* is the self-referencing column of *T* or *C* is a generated column of *T*, then <alter column action> shall not contain <add column scope clause> or <drop column scope clause>.

6) If *C* is an identity column, then <alter column action> shall contain either <alter identity column specification> or <drop identity property clause>.

7) If <alter identity column specification> or <drop identity property clause> is specified, then *C* shall be an identity column.

8) If <drop column generation expression clause> is specified, then *C* shall be a generated column of *T*.

9) If *C* is a generated column of *T*, a system-time period start column of *T*, or a system-time period end column of *T*, then <alter column action> shall not contain <set column default clause>.

Access Rules

None.

General Rules

1) The column descriptor of *C* is modified as specified by <alter column action>.

Conformance Rules

1) Without Feature F387, "ALTER TABLE statement: ALTER COLUMN clause", conforming SQL language shall not contain an <alter column definition>.

11.13 <set column default clause>

Function

Set the default clause for a column.

Format

```
<set column default clause> ::=
  SET <default clause>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

2) The default value specified by the <default clause> is placed in the column descriptor of *C*.

Conformance Rules

1) Without Feature F387, "ALTER TABLE statement: ALTER COLUMN clause", conforming SQL language shall not contain a <set column default clause>.

11.14 <drop column default clause>

Function

Drop the default clause from a column.

Format

```
<drop column default clause> ::=
  DROP DEFAULT
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

2) The descriptor of *C* shall include a default value.

Access Rules

None.

General Rules

1) The default value is removed from the column descriptor of *C*.

Conformance Rules

1) Without Feature F387, "ALTER TABLE statement: ALTER COLUMN clause", conforming SQL language shall not contain a <drop column default clause>.

11.15 <set column not null clause>

Function

Add a not null constraint to a column.

Format

```
<set column not null clause> ::=
  SET NOT NULL
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *C* be the column identified by the <column name> *CN* in the containing <alter column definition>. If the column descriptor of *C* does not contain an indication that *C* is defined as NOT NULL, then:

   a) Let *TN* be the <table name> in the containing <alter table statement>.

   b) Let *S* be the schema identified by the explicit or implicit <schema name> of *TN*. Let *IDCN* be an implementation-dependent (UV100) <constraint name> that is not equivalent to the <constraint name> of any table constraint descriptor included in *S*.

   c) The column descriptor of *C* is modified as follows:

      i) An indication that *C* is defined as NOT NULL is added.

      ii) *IDCN* is added as the constraint name of the associated table constraint definition.

   d) The following <alter table statement> is executed without further Access Rule checking:

      ```
      ALTER TABLE TN ADD CONSTRAINT IDCN CHECK ( CN IS NOT NULL )
      ```

Conformance Rules

1) Without Feature F383, "Set column not null clause", conforming SQL language shall not contain a <set column not null clause>.

11.16 <drop column not null clause>

Function

Drop a not null constraint on a column.

Format

```
<drop column not null clause> ::=
  DROP NOT NULL
```

Syntax Rules

1) Let *C* be the column identified by the <column name> *CN* in the containing <alter column definition>. Let *T* be the table identified by the <table name> *TN* in the containing <alter table statement>.

2) *C* shall be neither the system-time period start column of *T* nor the system-time period end column of *T*.

3) If the descriptor of *T* contains an application-time period descriptor, then let *ATPN* be the period name included in that descriptor. *C* shall be neither the *ATPN* period start column of *T* nor the *ATPN* period end column of *T*.

4) *T* shall not include a table constraint descriptor that indicates that the constraint was defined with PRIMARY KEY and that includes *CN* as a name of a unique column.

Access Rules

None.

General Rules

1) If the column descriptor of *C* contains an indication that *C* is defined as NOT NULL, then:

   a) Let *ACN* be the constraint name of the associated table constraint definition included in the column descriptor of *C*.

   b) The column descriptor of *C* is modified as follows:

      i) The indication that *C* is defined as NOT NULL is removed.

      ii) The constraint name of the associated table constraint definition is removed.

   c) The following <alter table statement> is executed without further Access Rule checking:

      ALTER TABLE *TN* DROP CONSTRAINT *ACN* CASCADE

Conformance Rules

1) Without Feature F383, "Set column not null clause", conforming SQL language shall not contain a <drop column not null clause>.

11.17 <add column scope clause>

Function

Add a non-empty scope for an existing column of data type REF in a base table.

Format

```
<add column scope clause> ::=
  ADD <scope clause>
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>. The declared type of *C* shall be some reference type. Let *RTD* be the reference type descriptor included in the descriptor of *C*.

2) Let *T* be the table identified by the <table name> in the containing <alter table statement>. If *T* is a referenceable table, then *C* shall be an originally-defined column of *T*.

3) *RTD* shall not include a scope.

4) Let *UDTN* be the name of the referenced type included in *RTD*.

5) The <table name> *STN* contained in the <scope clause> shall identify a referenceable table whose structured type is *UDTN*.

Access Rules

None.

General Rules

1) *STN* is included as the scope in the reference type descriptor included in the column descriptor of *C*.

2) For every proper subtable *PST* of *T*:

   a) Let *PSC* be the column whose corresponding column in *T* is *C*.

   b) *STN* is included as the scope in the reference type descriptor included in the column descriptor of *PSC*.

Conformance Rules

1) Without Feature F387, "ALTER TABLE statement: ALTER COLUMN clause", conforming SQL language shall not contain an <add column scope clause>.

2) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain an <add column scope clause>.

11.18 <drop column scope clause>

This Subclause is modified by Subclause 10.7, "<drop column scope clause>", in ISO/IEC 9075-4.

Function

Drop the scope from an existing column of data type REF in a base table.

Format

```
<drop column scope clause> ::=
  DROP SCOPE <drop behavior>
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>. The declared type of *C* shall be some reference type whose reference type descriptor includes a scope.

2) Let *T* be the table identified by the <table name> in the containing <alter table statement>. If *T* is a referenceable table, then *C* shall be an originally-defined column of *T*.

3) Let *SC* be the set of columns consisting of *C* and, for every proper subtable of *T*, the column whose supercolumn is *C*.

4) An *impacted dereference operation* is a <dereference operation> whose <reference value expression> is a column reference that identifies a column in *SC*, a <method reference> whose <value expression primary> is a column reference that identifies a column in *SC*, or a <reference resolution> whose <reference value expression> is a column reference that identifies a column in *SC*.

5) If RESTRICT is specified, then no impacted dereference operation shall be contained in any of the following:

   a) The SQL routine body of any routine descriptor.

   b) The original <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) 04 The triggered action of any trigger descriptor.

   NOTE 563 — If CASCADE is specified, then such referencing objects will be dropped implicitly by the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

None.

General Rules

1) 04 For every SQL-invoked routine *R* whose routine descriptor includes an SQL routine body that contains an impacted dereference operation, let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed for every *R* without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

2) For every view *V* whose view descriptor includes an original <query expression> that contains an impacted dereference operation, let *VN* be the <table name> of *V*. The following <drop view statement> is effectively executed for every *V* without further Access Rule checking:

```
DROP VIEW VN CASCADE
```

3) For every assertion *A* whose assertion descriptor includes a <search condition> that contains an impacted dereference operation, let *AN* be the <constraint name> of *A*. The following <drop assertion statement> is effectively executed for every *A* without further Access Rule checking:

```
DROP ASSERTION AN CASCADE
```

4) 04 For every table check constraint *CC* whose table check constraint descriptor includes a <search condition> that contains an impacted dereference operation, let *CN* be the <constraint name> of *CC* and let *TN* be the <table name> of the table whose descriptor includes descriptor of *CC*. The following <alter table statement> is effectively executed for every *CC* without further Access Rule checking:

```
ALTER TABLE TN DROP CONSTRAINT CN CASCADE
```

5) The scope included in the reference type descriptor included in the column descriptor of every column in *SC* is made empty.

Conformance Rules

1) Without Feature F387, "ALTER TABLE statement: ALTER COLUMN clause", conforming SQL language shall not contain a <drop column scope clause>.

2) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <drop column scope clause>.

11.19 <alter column data type clause>

This Subclause is modified by Subclause 10.6, "<alter column data type clause>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.3, "<alter column data type clause>", in ISO/IEC 9075-14.

Function

Change the declared type of a column.

Format

```
<alter column data type clause> ::=
  SET DATA TYPE <data type>
```

Syntax Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

2) Let *T* be the table identified by the <table name> in the containing <alter table statement>.

3) *C* shall not be the self-referencing column of *T*.

4) If the table descriptor of *T* includes a system-time period descriptor, then *C* shall not be the system-time period start column or the system-time period end column of *T*.

5) If the table descriptor of *T* includes a period descriptor that contains *ATPN* as its period name, then *C* shall not be the *ATPN* period start column of *T* or the *ATPN* period end column of *T*.

6) *C* shall not be referenced in the <search condition> of any constraint descriptor other than a table constraint descriptor that contains references to no other column than *C* and that is included in the table descriptor of *T*.

7) *C* shall not be referenced in either an explicit trigger column list or a triggered action column set of any trigger descriptor.

8) *C* shall not be referenced in the SQL routine body of any routine descriptor.

9) *C* shall not be referenced in the original <query expression> of any view descriptor.

10) *C* shall not be referenced in the generation expression of any column descriptor.

11) Let *DTC* be the declared type of *C*.

12) Case:

    a) If <data type> contains <character string type> *CST*, then:

        i) Case:

            1) If <data type> contains <character set specification>, then let *CSS* be that <character set specification>.

            2) Otherwise, let *CSS* be a <character set specification> that identifies the character set of *DTC*.

        ii) Case:

            1) If <data type> contains <collation name>, then let *COLL* be that <collation name>.

2) Otherwise, let *COLL* be a <collation name> that identifies the collation of *DTC*.

iii) Let *D* be the data type specified by

```
CST CHARACTER SET CSS COLLATION COLL
```

b) If <data type> contains <national character string type> *NCST*, then:

i) Case:

1) If <data type> contains <collation name>, then let *COLL* be that <collation name>.

2) Otherwise, let *COLL* be a <collation name> that identifies the collation of *DTC*.

ii) Let *D* be the data type specified by

```
NCST COLLATION COLL
```

c) Otherwise, let *D* be the data type specified by <data type>.

13) *D* shall be a predefined type.

14) *DTC* shall be a predefined type.

15) 09 Case:

a) If *DTC* is a character string type, then:

i) *D* shall be a character string type.

ii) Let *LD* be the length or maximum length of *D*.

iii) Let *LDTC* be the length or maximum length of *DTC*.

iv) *LD* shall be greater than or equal to *LDTC*.

v) The character set and collation of *D* shall be the character set and collation of *DTC*.

b) If *DTC* is a binary string type, then:

i) *D* shall be a binary string type.

ii) Let *LD* be the length or maximum length of *D*.

iii) Let *LDTC* be the length or maximum length of *DTC*.

iv) *LD* shall be greater than or equal to *LDTC*.

c) If *DTC* is an exact numeric type, then:

i) *D* shall be an exact numeric type.

ii) Let *PD* and *SD* be the precision and scale, respectively, of *D*.

iii) Let *PDTC* and *SDTC* be the precision and scale, respectively, of *DTC*.

iv) *PD* shall be greater than or equal to *PDTC*.

v) *SD* shall be greater than or equal to *SDTC*.

vi) (*PD* − *SD*) shall be greater than or equal to (*PDTC* − *SDTC*).

d) If *DTC* is an approximate numeric type, then:

i) *D* shall be an approximate numeric type.

ii) Let *PD* be the precision of *D*.

iii) Let *PDTC* be the precision of *DTC*.

iv) *PD* shall be greater than or equal to *PDTC*.

e) If *DTC* is DATE, then *D* shall be DATE.

f) If *DTC* is a time type with time zone or a time type without time zone, then *D* shall be a time type with time zone or a time type without time zone, respectively.

i) Let *PD* be the value of the <time fractional seconds precision> of *D*.

ii) Let *PDTC* be the value of the <time fractional seconds precision> of *DTC*.

iii) *PD* shall be greater than or equal to *PDTC*.

g) If *DTC* is a timestamp type with time zone or a timestamp type without time zone, then *D* shall be a timestamp type with time zone or a timestamp type without time zone, respectively.

i) Let *PD* be the value of the <time fractional seconds precision> of *D*.

ii) Let *PDTC* be the value of the <time fractional seconds precision> of *DTC*.

iii) *PD* shall be greater than or equal to *PDTC*.

h) If *DTC* is a year-month interval type, then *D* shall be a year-month interval type.

i) If *DTC* specifies <start field> and <end field>, then *D* shall specify the same <start field> and <end field>.

ii) If *DTC* specifies <single datetime field>, then *D* shall specify the same <single datetime field>.

iii) The explicit or implicit <interval leading field precision> of *D* shall be greater than or equal to the explicit or implicit <interval leading field precision> of *DTC*.

i) If *DTC* is a day-time interval type, then *D* shall be a day-time interval type.

i) If *DTC* specifies <start field> and <end field>, then *D* shall specify the same <start field> and <end field>.

ii) If *DTC* specifies <single datetime field>, then *D* shall specify the same <single datetime field>.

iii) The explicit or implicit <interval leading field precision> of *D* shall be greater than or equal to the explicit or implicit <interval leading field precision> of *DTC*.

iv) If *DTC* specifies an explicit or implicit <interval fractional seconds precision> *PDTC*, then *D* shall specify an explicit or implicit <interval fractional seconds precision> that is greater than or equal to *PDTC*.

j) [14] If *DTC* is BOOLEAN, then *D* shall be BOOLEAN.

k) Otherwise, *D* shall be *DTC*.

Access Rules

None.

General Rules

1) The column descriptor of *C* is modified by replacing the existing data type descriptor with a data type descriptor describing *D*.

Conformance Rules

1) [14] Without Feature F382, "Alter column data type", conforming SQL language shall not contain an <alter column data type clause>.

11.20 <alter identity column specification>

Function

Change the options specified for an identity column.

Format

```
<alter identity column specification> ::=
    <set identity column generation clause> [ <alter identity column option>... ]
  | <alter identity column option>...

<set identity column generation clause> ::=
  SET GENERATED { ALWAYS | BY DEFAULT }

<alter identity column option> ::=
    <alter sequence generator restart option>
  | SET <basic sequence generator option>
```

Syntax Rules

1) If at least one instance of <alter identity column option> is specified, then:

   a) Let *SEQ* be the sequence generator descriptor included in the column descriptor identified by the <column name> in the containing <alter column definition>.

   b) Let *OPT* be the character string obtained by concatenating all <alter identity column option>s while each instance of the keyword SET is replaced with a single space character such that *OPT* conforms to the Format of <alter sequence generator options>.

   c) The Syntax Rules of Subclause 9.37, "Altering a sequence generator", are applied with *OPT* as *OPTIONS* and *SEQ* as *SEQUENCE*.

Access Rules

None.

General Rules

1) If at least one <alter identity column option> is specified, then the General Rules of Subclause 9.37, "Altering a sequence generator", are applied with *OPT* as *OPTIONS* and *SEQ* as *SEQUENCE*.

2) If <set identity column generation clause> is specified, then:

   a) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

   b) Case:

      i) If ALWAYS is specified, then the column descriptor of *C* is modified to indicate that values are always generated.

      ii) If BY DEFAULT is specified, then the column descriptor of C is modified to indicate that values are generated by default.

Conformance Rules

1) Without Feature T174, “Identity columns”, in conforming SQL language, an <alter column definition> shall not contain an <alter identity column specification>.

2) Without Feature T178, “Identity columns: simple restart option”, in conforming SQL language, an <alter sequence generator restart option> contained in an <alter identity column specification> shall contain a <sequence generator restart value>.

3) Without Feature F386, “Set identity column generation clause”, conforming SQL language shall not contain a <set identity column generation clause>.

11.21 <drop identity property clause>

Function

Convert an identity column to a column that is not an identity column.

Format

```
<drop identity property clause> ::=
  DROP IDENTITY
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.
2) The column descriptor of *C* is modified as follows:
   a) The indication that *C* is an identity column is removed.
   b) The indication that values are always generated, if any, is removed.
   c) The indication that values are generated by default, if any, is removed.
   d) The sequence generator *SG* included in the column descriptor of *C* is removed.
3) *SG* is destroyed.

Conformance Rules

1) Without Feature F384, "Drop identity property clause", conforming SQL language shall not contain a <drop identity property clause>.

11.22 <drop column generation expression clause>

Function

Convert a generated column to a column that is not a generated column.

Format

```
<drop column generation expression clause> ::=
  DROP EXPRESSION
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *C* be the column identified by the <column name> in the containing <alter column definition>.

2) The column descriptor of *C* is modified as follows:

   a) The generation expression of *C* is removed.

   b) The indication that values are "ALWAYS" generated is removed.

   c) An indication is added that the column is "NEVER" generated.

Conformance Rules

1) Without Feature F385, "Drop column generation expression clause", conforming SQL language shall not contain a <drop column generation expression clause>.

11.23 <drop column definition>

This Subclause is modified by Subclause 10.8, "<drop column definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.7, "<drop column definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 11.4, "<drop column definition>", in ISO/IEC 9075-16.

Function

Destroy a column of a base table.

Format

```
<drop column definition> ::=
  DROP [ COLUMN ] <column name> <drop behavior>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> in the containing <alter table statement> and let *TN* be the name of *T*.

2) Let *C* be the column identified by the <column name> *CN*.

3) *T* shall not be a referenceable table or a system-versioned table.

4) *C* shall be a column of *T* and *C* shall not be the only column of *T*.

5) If the table descriptor of *T* includes a system-time period descriptor, then *C* shall not be the system-time period start column of *T* or the system-time period end column of *T*.

6) If the table descriptor of *T* includes a period descriptor that contains *ATPN* as its period name, then *C* shall not be the *ATPN* period start column of *T* or the *ATPN* period end column of *T*.

7) If RESTRICT is specified, then *C* shall not be referenced in any of the following:

   a) The original <query expression> of any view descriptor.

   b) The <search condition> of any constraint descriptor other than a table constraint descriptor that contains references to no other column and that is included in the table descriptor of *T*.

   c) The SQL routine body of any routine descriptor.

   d) The <parameter default> of any SQL parameter of any routine descriptor.

   e) Either an explicit trigger column list or a triggered action column set of any trigger descriptor.

   f) The generation expression of any column descriptor.

NOTE 564 — A <drop column definition> that does not specify CASCADE will fail if there are any references to that column resulting from the use of CORRESPONDING, NATURAL, SELECT * (except where contained in an <exists predicate>), a multi-column <unique constraint definition>, or REFERENCES without a <referenced column list> in its <referenced table and columns>. Also note that a <drop column definition> whose <drop behavior> is RESTRICT that references a column that is referenced by a multi-column <unique constraint definition> will fail, while dropping a column that is referenced by a single-column <unique constraint definition> will succeed.

NOTE 565 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

NOTE 566 — *CN* can be contained in an implicit trigger column list of a trigger descriptor.

Access Rules

None.

General Rules

1) For every trigger *TR* having an explicit trigger column list or a triggered action column set that contains *CN*:

   a) Let *TRN* be the name of *TR*.

   b) The following <drop trigger statement> is effectively executed without further Access Rule checking:

      ```
      DROP TRIGGER TRM
      ```

2) 09 Let *A* be the <authorization identifier> that owns *T*. The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

   ```
   REVOKE INSERT(CN), UPDATE(CN), SELECT(CN), REFERENCES(CN)
   ON TABLE TN
   FROM A CASCADE
   ```

3) For every generated column *GC* of *T* in whose descriptor the generation expression contains a <column reference> that references *C*:

   a) Let *GCN* be the name of *GC*.

   b) The following <alter table statement> is effectively executed without further Access Rule checking:

      ```
      ALTER TABLE TN DROP COLUMN GCN CASCADE
      ```

4) If the column is not based on a domain, then its data type descriptor is destroyed; otherwise, all domain constraint usages that reference *C* are destroyed.

5) The data associated with *C* is destroyed.

6) The descriptor of *C* is removed from the descriptor of *T*.

7) The descriptor of *C* is destroyed.

8) The degree of *T* is reduced by 1 (one). The ordinal position of all columns having an ordinal position greater than the ordinal position of *C* is reduced by 1 (one).

Conformance Rules

1) Without Feature F033, "ALTER TABLE statement: DROP COLUMN clause", conforming SQL language shall not contain a <drop column definition>.

11.24 <add table constraint definition>

Function

Add a constraint to a table.

Format

```
<add table constraint definition> ::=
  ADD <table constraint definition>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> contained in the containing <alter table statement>.

2) If PRIMARY KEY is specified, then *T* shall not have any proper supertable.

Access Rules

None.

General Rules

1) The table constraint descriptor for the <table constraint definition> is included in the table descriptor for *T*.

2) Let *TC* be the table constraint added to *T*. If *TC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to known not nullable.

NOTE 567 — The nullability characteristic of a column is defined in Subclause 4.15, "Columns, fields, and attributes".

Conformance Rules

1) Without Feature F388, "ALTER TABLE statement: ADD/DROP CONSTRAINT clause", conforming SQL language shall not contain an <add table constraint definition>.

11.25 <alter table constraint definition>

Function

Change the definition of a table constraint.

Format

```
<alter table constraint definition> ::=
  ALTER CONSTRAINT <constraint name> <constraint enforcement>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> in the containing <alter table statement>.

2) The <constraint name> shall identify a table constraint *TC* of *T*.

3) *TC* shall not identify a unique constraint.

Access Rules

None.

General Rules

1) The table constraint descriptor of *TC* is modified as follows.

   Case:

   a) If NOT ENFORCED is specified, then the indication of whether the constraint is enforced or not enforced is replaced with an indication that the constraint is not enforced.

   b) Otherwise, the indication of whether the constraint is enforced or not enforced is replaced with an indication that the constraint is enforced.

Conformance Rules

1) Without Feature F492, "Optional table constraint enforcement", conforming SQL language shall not contain an <alter table constraint definition> that contains a <constraint enforcement>.

11.26 <drop table constraint definition>

This Subclause is modified by Subclause 10.9, "<drop table constraint definition>", in ISO/IEC 9075-4.

Function

Destroy a constraint on a table.

Format

```
<drop table constraint definition> ::=
  DROP CONSTRAINT <constraint name> <drop behavior>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> in the containing <alter table statement>.

2) The <constraint name> shall identify a table constraint *TC* of *T*.

3) If *TC* is a unique constraint, *T* is the referenced table of a referential constraint *RC* and the referenced columns of *RC* are the unique columns of *TC*, then *RC* is said to be *dependent on TC*.

4) Let *QS* be a <query specification> that contains an implicit or explicit <group by clause> and that contains a column reference to a column *C* in its <select list> that is not contained in an aggregated argument of a <set function specification>, let *G* be the set of grouping columns of *QS*. If *TC* is needed to conclude that $G \mapsto C$ is a known functional dependency in *QS*, then *QS* is said to be *dependent on TC*.

5) Let *V1* be a view that contains a <query specification> that is dependent on *TC*. *V1* is said to be *dependent on TC*.

6) Let *R1* be an SQL routine whose <SQL routine body> contains a <query specification> that is dependent on *TC*. *R1* is said to be *dependent on TC*.

7) Let *C1* be a constraint or assertion whose <search condition> contains a <query specification> that is dependent on *TC*. *C1* is said to be *dependent on TC*.

8) Let *TR1* be a trigger whose triggered action contains a <query specification> that is dependent on *TC*. *TR1* is said to be *dependent on TC*.

9) If *T* is a referenceable table with a derived self-referencing column, then *TC* shall not be a unique constraint whose unique columns correspond to the derivational attributes of the derived representation of the reference type whose referenced type is the structured type of *T*.

10) If the descriptor of *T* includes an application-time period descriptor *ATPD*, then:

    a) *TC* shall not be the period constraint included in *ATPD*.

    b) Let *ATPN* be the <application time period name> included in *ATPD*. Destruction of *TC* shall not cause the nullability characteristic of the *ATPN* start column of *T* or the *ATPN* end column of *T* to change from known not nullable to possibly nullable.

11) Destruction of *TC* shall not cause the nullability characteristic of any of the following columns of *T* to change from known not nullable to possibly nullable:

    a) A column that is a constituent of the primary key of *T*, if any.

    b) The system-time period start column, if any.

c) The system-time period end column, if any.

d) The identity column, if any.

12) *TC* shall not be a unique constraint whose unique column is the self-referencing column of *T*.

13) If RESTRICT is specified, then:

a) No table constraint shall be dependent on *TC*.

b) The <constraint name> of *TC* shall not be contained in the SQL routine body of any routine descriptor.

c) No view shall be dependent on *TC*.

d) No SQL routine shall be dependent on *TC*.

e) No constraint or assertion shall be dependent on *TC*.

f) No trigger shall be dependent on *TC*.

NOTE 568 — If CASCADE is specified, then any such dependent object will be dropped implicitly by the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

None.

General Rules

1) For every table constraint *TCD* that is dependent on *TC*:

a) Let *TCDN* be the constraint name of *TCD* and let *TN2* be the table name included in the table descriptor that includes the descriptor of *TCD*.

b) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE TN2 DROP CONSTRAINT TCDN CASCADE
```

2) Let *RS* be the set of all SQL-invoked routines whose routine descriptor contains the constraint name of *TC* in the <SQL routine body> and all SQL routines that are dependent on *TC*.

For every routine *R* in *RS*:

a) Let *SN* be the specific name of *R*.

b) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

3) For every view *V* that is dependent on *TC*:

a) Let *VN* be the table name of *V*.

b) The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW VN CASCADE
```

4) For every assertion *A* that is dependent on *TC*:

   a) Let *AN* be the constraint name of *A*.

   b) The following <drop assertion statement> is effectively executed without further Access Rule checking:

      ```
      DROP ASSERTION AN CASCADE
      ```

5) For every trigger *TR* that is dependent on *TC*:

   a) Let *TRN* be the trigger name of *TR*.

   b) The following <drop trigger statement> is effectively executed without further Access Rule checking:

      ```
      DROP TRIGGER TRN
      ```

6) The descriptor of *TC* is removed from the descriptor of *T*.

7) If *TC* causes some column *COL* to be known not nullable and no other constraint causes *COL* to be known not nullable, then the nullability characteristic of the column descriptor of *COL* is changed to possibly nullable.

   NOTE 569 — The nullability characteristic of a column is defined in Subclause 4.15, "Columns, fields, and attributes".

8) The descriptor of *TC* is destroyed.

Conformance Rules

1) Without Feature F388, "ALTER TABLE statement: ADD/DROP CONSTRAINT clause", conforming SQL language shall not contain a <drop table constraint definition>.

11.27 <add table period definition>

Function

Add a system-time period or an application-time period to a persistent base table.

Format

```
<add table period definition> ::=
  ADD <table period definition> [ <add system time period column list> ]

<add system time period column list> ::=
  ADD [ COLUMN ] <column definition 1> ADD [ COLUMN ] <column definition 2>

<column definition 1> ::=
  <column definition>

<column definition 2> ::=
  <column definition>
```

Syntax Rules

1) Let *TN* be the <table name> immediately contained in the containing <alter table statement>. Let *T* be the table identified by *TN*.

2) *T* shall be a persistent base table.

3) *T* shall not be a referenceable table.

4) If <table period definition> contains <system time period specification>, then:

   a) The table descriptor of *T* shall not include a system-time period descriptor.

   b) No column of *T* shall have a name that is equivalent to SYSTEM_TIME.

   c) <add system time period column list> shall be specified.

   d) <add system time period column list> shall contain exactly one <column definition> *CD1* that contains <system time period start column specification> and exactly one <column definition> *CD2* that contains <system time period end column specification>.

   e) Let *SCN1* be the <period begin column name> contained in the <table period definition>. *SCN1* shall be equivalent to the <column name> contained in *CD1*. The column identified by *SCN1* is the system-time period start column.

   f) Let *SCN2* be the <period end column name> contained in the <table period definition>. *SCN2* shall be equivalent to the <column name> contained in *CD2*. The column identified by *SCN2* is the system-time period end column.

   g) The <data type or domain name> contained in *CD1* shall be equivalent to the <data type or domain name> contained in *CD2*.

5) If <table period definition> contains <application time period specification> *ATPS*, then:

   a) Let *ATPN* be the <application time period name> contained in *ATPS*.

   b) The table descriptor of *T* shall not include a period descriptor other than a system-time period descriptor.

c) No column of *T* shall have a column name that is equivalent to *ATPN*.

d) <add system time period column list> shall not be specified.

e) Let *BCN1* be the <period begin column name> contained in the <table period definition>. The table descriptor of *T* shall include a column descriptor that includes *BCN1* as the column name. The column *BC1* identified by *BCN1* is the *ATPN* period start column.

f) Let *BCN2* be the <period end column name> contained in the <table period definition>. The table descriptor of *T* shall include a column descriptor that includes *BCN2* as the column name. The column *BC2* identified by *BCN2* is the *ATPN* period end column.

g) The declared type of *BC1* shall be either DATE or a timestamp type and shall be equivalent to the declared type of *BC2*.

h) *BC1* and *BC2* shall be known not nullable.

i) Neither *BC1* nor *BC2* shall be an identity column, a generated column, a system-time period start column, or a system-time period end column.

Access Rules

None.

General Rules

1) If <table period definition> contains <system time period specification>, then:

a) Let *CD1* be <column definition 1>. The following <alter table statement> is executed without further Access Rule checking:

```
ALTER TABLE TN ADD COLUMN CD1
```

b) Let *CD2* be <column definition 2>. The following <alter table statement> is executed without further Access Rule checking:

```
ALTER TABLE TN ADD COLUMN CD2
```

c) Let *C1* be the column specified by *CD1*. Let *C2* be the column specified by *CD2*. Let *DT* be the data type of *C2*. Let *TTS* be an implementation-defined (IV181) value of type *DT*. Every row of *T* is effectively updated at the end of each SQL-statement such that the value of *C1* is set to *TTS* and the value of *C2* is set to the greatest value supported by *DT*.

NOTE 570 — This update does not create a new state change in the most recent statement execution context.

d) Let *SCN1* be the <period begin column name> contained in the <table period definition>. Let *SCN2* be the <period end column name> contained in the <table period definition>. The table descriptor of *T* is modified to include a period descriptor that contains SYSTEM_TIME as the name of the period, *SCN1* as the name of the system-time period start column and *SCN2* as the name of the system-time period end column.

2) If <table period definition> contains <application time period specification>, then:

a) Let *S* be the schema identified by the explicit or implicit <schema name> of *TN*. Let *BCN1* be the <period begin column name> contained in the <table period definition>. Let *BCN2* be the <period end column name> contained in the <table period definition>. Let *IDCN* be an implementation-dependent (UV100) <constraint name> that is not equivalent to the <constraint name> of any table constraint descriptor included in *S*.

b) The following <alter table statement> is executed without further Access Rule checking:

```
ALTER TABLE TN ADD CONSTRAINT IDCN CHECK ( BCN1 < BCN2 )
```

c) The table descriptor of *T* is modified to include a period descriptor that contains *ATPN* as the name of the period, *BCN1* as the name of the *ATPN* period start column, *BCN2* as the name of the *ATPN* period end column, and *IDCN* as the *ATPN* period constraint name.

Conformance Rules

1) Without Feature T180, "System-versioned tables", conforming SQL language shall not contain ADD <system time period specification>.

2) Without Feature T181, "Application-time period tables", conforming SQL language shall not contain ADD <application time period specification>.

11.28 <drop table period definition>

Function

Remove a system-time period or application-time period from a persistent base table.

Format

```
<drop table period definition> ::=
  DROP <system or application time period specification> <drop behavior>
```

Syntax Rules

1) Let *TN* be the <table name> immediately contained in the containing <alter table statement>. Let *T* be the table identified by *TN*.

2) If <system or application time period specification> specifies PERIOD SYSTEM_TIME, then:

   a) *T* shall not be a system-versioned table.

   b) The descriptor of *T* shall include a system-time period descriptor.

   c) If RESTRICT is specified, then neither the system-time period start column of *T* nor the system-time period end column of *T* shall be referenced in any of the following:

      i) The original <query expression> of any view descriptor.

      ii) The SQL routine body of any routine descriptor.

      iii) The <parameter default> of any SQL parameter of any routine descriptor.

      iv) The <triggered action> of any trigger descriptor.

      v) The generation expression of any column descriptor.

      NOTE 571 — A <drop table period definition> that does not specify CASCADE will fail if there are any references to the system-time period start column of *T* or to the system-period end column of *T* resulting from the use of CORRESPONDING, NATURAL, or SELECT * (except where contained in an <exists predicate>).

      NOTE 572 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <drop column definition> specified in the General Rules of this Subclause.

   d) If RESTRICT is specified, then the SYSTEM_TIME period of *T* shall not be referenced in any of the following:

      i) The original <query expression> of any view descriptor.

      ii) The <search condition> of any constraint descriptor.

      iii) The <search condition> of any assertion descriptor.

      iv) The <parameter default> of any SQL parameter of any routine descriptor.

      v) The SQL routine body of any routine descriptor.

      vi) The <triggered action> of any trigger descriptor.

      vii) The generation expression of any column descriptor.

      NOTE 573 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the General Rules of this Subclause.

3) If <system or application time period specification> specifies an <application time period name> *ATPN*, then:

   a) The descriptor of *T* shall include an *ATPN* period descriptor.

   b) If RESTRICT is specified, then a <delete statement: searched> that contains a <target table> that is *T* and that contains an <application time period name> that is equivalent to *ATPN* shall not be referenced in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The <triggered action> of any trigger descriptor.

      NOTE 574 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the General Rules of this Subclause.

   c) If RESTRICT is specified, then an <update statement: searched> that contains a <target table> that is *T* and that contains an <application time period name> that is equivalent to *ATPN* shall not be referenced in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The <triggered action> of any trigger descriptor.

      NOTE 575 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the General Rules of this Subclause.

   d) If RESTRICT is specified, then the period identified by *ATPN* shall not be referenced in any of the following:

      i) The original <query expression> of any view descriptor.

      ii) The <search condition> of any constraint descriptor.

      iii) The <search condition> of any assertion descriptor.

      iv) The <parameter default> of any SQL parameter of any routine descriptor.

      v) The SQL routine body of any routine descriptor.

      vi) The <triggered action> of any trigger descriptor.

      vii) The generation expression of any column descriptor.

      NOTE 576 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the General Rules of this Subclause.

Access Rules

None.

General Rules

1) If <system or application time period specification> specifies PERIOD SYSTEM_TIME, then:

   a) Let *SCN* be the <column name> of the system-time period start column of *T*. The following <alter table statement> is executed without further Access Rule checking:

      ```
      ALTER TABLE TN DROP COLUMN SCN CASCADE
      ```

   b) Let *ECN* be the <column name> of the system-time period end column of *T*. The following <alter table statement> is executed without further Access Rule checking:

```
ALTER TABLE TN DROP COLUMN ECN CASCADE
```

c) The system-time period descriptor is removed from the table descriptor of *T*.

d) For every period reference *PR* that references the SYSTEM_TIME period of *T*.

Case:

i) If *PR* is contained in the original <query specification> of a view descriptor, then for every such view descriptor:

1) Let *SON* be the name of the view included in that view descriptor.

2) The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

ii) If *PR* is contained in the <search condition> of an assertion descriptor, then for every such assertion descriptors:

1) Let *SON* be the name of the constraint included in that assertion descriptor.

2) The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

iii) If *PR* is contained in the <search condition> of any table constraint descriptor, then for every such table constraint descriptor:

1) Let *SOD* be the table constraint descriptor, let *SON* be the name of the constraint included in *SOD*, and let *CTN* be the <table name> included in the table descriptor that includes *SOD*.

2) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

iv) If *PR* is contained in the SQL routine body of any routine descriptor or is contained in the <parameter default> of any SQL parameter of any routine descriptor, then for every such routine descriptor:

1) Let *SON* be the specific name included in that routine descriptor.

2) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

v) If *PR* is contained in the <triggered action> of any trigger descriptor, then for every such trigger descriptor:

1) Let *SON* be the trigger name included in that trigger descriptor.

2) The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON
```

2) If <system or application time period specification> specifies an <application time period name> *ATPN*, then:

a) Let *IDCN* be the name of the *ATPN* period constraint contained in the *ATPN* period descriptor of *T*.

b) The *ATPN* period descriptor is removed from the table descriptor of *T*.

c) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE TN DROP CONSTRAINT IDCN CASCADE
```

d) For every SQL routine *R* such that the SQL routine body in the descriptor of *R* contains either a <delete statement: searched> *SS* or an <update statement: searched> *SS* such that *SS* contains a <target table> that is *T* and *SS* contains an <application time period name> that is equivalent to *APTN*:

i) Let *SON* be the specific name included in the routine descriptor of *R*.

ii) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

e) For every trigger *TT* such that the <triggered action> of *TT* contains either a <delete statement: searched> *SS* or an <update statement: searched> *SS* such that *SS* contains a <target table> that is *T* and *SS* contains an <application time period name> that is equivalent to *APTN*:

i) Let *TTN* be the name of *TT*.

ii) The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER TTN
```

f) For every period reference *PR* that references the *ATPN* period of *T*,

Case:

i) If *PR* is contained in the original <query specification> of a view descriptor, then for every such view descriptor:

1) Let *SON* be the name of the view included in that view descriptor.

2) The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

ii) If *PR* is contained in the <search condition> of an assertion descriptor, then for every such assertion descriptor:

1) Let *SON* be the name of the constraint included in that assertion descriptor.

2) The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

iii) If *PR* is contained in the <search condition> of any table constraint descriptor, then for every such table constraint descriptor:

1) Let *SOD* be the table constraint descriptor, let *SON* be the name of the constraint included in *SOD*, and let *CTN* be the <table name> included in the table descriptor that includes *SOD*.

2) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

iv) If *PR* is contained in the SQL routine body of any routine descriptor or is contained in the <parameter default> of any SQL parameter of any routine descriptor, then for every such routine descriptor:

1) Let *SON* be the specific name included in that routine descriptor.

2) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

v) If *PR* is contained in the <triggered action> of any trigger descriptor, then for every such trigger descriptor:

1) Let *SON* be the trigger name included in that trigger descriptor.

2) The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON
```

Conformance Rules

1) Without Feature T180, "System-versioned tables", conforming SQL language shall not contain DROP PERIOD SYSTEM_TIME.

2) Without Feature T181, "Application-time period tables", conforming SQL language shall not contain a <drop table period definition> that contains an <application time period name>.

11.29 <add system versioning clause>

Function

Alter a regular persistent base table to a system-versioned table.

Format

```
<add system versioning clause> ::=
  ADD <system versioning clause>
```

Syntax Rules

1) Let *TN* be the <table name> immediately contained in the containing <alter table statement>. Let *T* be the table identified by *TN*.

2) *T* shall be a regular persistent base table.

3) *T* shall not be a referenceable table.

4) The descriptor of *T* shall include a system-time period descriptor.

5) *T* shall not be a system-versioned table.

6) The descriptor of *T* shall not include the descriptor of a column whose declared type is a reference type.

Access Rules

None.

General Rules

1) The table descriptor of *T* is modified such that the indication that *T* is a regular persistent base table is modified to indicate that *T* is a system-versioned table.

Conformance Rules

1) Without Feature T180, "System-versioned tables", conforming SQL language shall not contain <add system versioning clause>.

11.30 <drop system versioning clause>

Function

Change a system-versioned table into a regular persistent base table.

Format

```
<drop system versioning clause> ::=
  DROP SYSTEM VERSIONING <drop behavior>
```

Syntax Rules

1) Let *TN* be the <table name> immediately contained in the containing <alter table statement>. Let *T* be the table identified by *TN*.

2) *T* shall be a system-versioned table.

3) If RESTRICT is specified, then a <table primary> that specifies <query system time period specification> and references *T* shall not be referenced in any of the following:

   a) The original <query expression> of any view descriptor.

   b) The <search condition> of any constraint descriptor.

   c) The <search condition> of any assertion descriptor.

   d) The <parameter default> of any SQL parameter of any routine descriptor.

   e) The SQL routine body of any routine descriptor.

   f) The <triggered action> of any trigger descriptor.

   g) The generation expression of any column descriptor.

   NOTE 577 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the General Rules of this Subclause.

Access Rules

None.

General Rules

1) The table descriptor of *T* is modified such that the indication that *T* is a system-versioned table is modified to indicate it is a regular persistent base table.

2) Every row of *T* that corresponds to a historical system row is effectively deleted at the end of the SQL-statement.

   NOTE 578 — This deletion does not create a new state change in the most recent statement execution context.

3) For each <table primary> *TP* that specifies <query system time period specification> and references *T*,

   Case:

a) If *TP* is contained in the original <query specification> of a view descriptor, then for every such view descriptor:

   i) Let *SON* be the name of the view included in that view descriptor.

   ii) The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

b) If *TP* is contained in the <search condition> of an assertion descriptor, then for every such assertion descriptor:

   i) Let *SON* be the name of the constraint included in that assertion descriptor.

   ii) The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

c) If *TP* is contained in the <search condition> of any table constraint descriptor:

   i) Let *SOD* that table constraint descriptor, let *SON* be the name of the constraint included in *SOD*, and let *CTN* be the <table name> included in the table descriptor that includes *SOD*.

   ii) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

d) If *TP* is contained in the SQL routine body of any routine descriptor or is contained in the <parameter default> of any SQL parameter of any routine descriptor, then for every such routine descriptor:

   i) Let *SON* be the specific name included in that routine descriptor.

   ii) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

e) If *TP* is contained in the <triggered action> of any trigger descriptor, then for every such trigger descriptor:

   i) Let *SON* be the trigger name included in that trigger descriptor.

   ii) The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON
```

Conformance Rules

1) Without Feature T180, “System-versioned tables”, conforming SQL language shall not contain <drop system versioning clause>.

11.31 <drop table statement>

This Subclause is modified by Subclause 10.10, "<drop table statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.5, "<drop table statement>", in ISO/IEC 9075-16.

Function

Destroy a table.

Format

```
<drop table statement> ::=
  DROP TABLE <table name> <drop behavior>
```

Syntax Rules

1) Let *T* be the table identified by the <table name> and let *TN* be that <table name>.

2) The schema identified by the explicit or implicit <schema name> of the <table name> shall include the descriptor of *T*.

3) *T* shall be a base table.

4) *T* shall not be a declared local temporary table.

5) An *impacted dereference operation* is any of the following:

   a) A <dereference operation> *DO*, where *T* is the scope of the reference type of the <reference value expression> immediately contained in *DO*.

   b) A <method reference> *MR*, where *T* is the scope of the reference type of the <value expression primary> immediately contained in *MR*.

   c) A <reference resolution> *RR*, where *T* is the scope of the reference type of the <reference value expression> immediately contained in *RR*.

6) If RESTRICT is specified, then *T* shall not have any proper subtables.

7) 04 If RESTRICT is specified, then *T* shall not be referenced and no impacted dereference operation shall be contained in any of the following:

   a) The original <query expression> of any view descriptor.

   b) The <search condition> of any constraint descriptor that is not a table check constraint descriptor included in the base table descriptor of *T*.

   c) The <search condition> of any assertion descriptor.

   d) The table descriptor of the referenced table of any referential constraint descriptor of any table other than *T*.

   e) The SQL routine body of any routine descriptor.

   f) The <parameter default> of any SQL parameter of any routine descriptor.

   g) 16 The <triggered action> of any trigger descriptor.

NOTE 579 — If CASCADE is specified, then such objects will be dropped by the execution of the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

8) If RESTRICT is specified and *T* is a referenceable table, then *TN* shall not be the scope included in a reference type descriptor generally included in any of the following:

   a) The attribute descriptor of an attribute of a user-defined type.

   b) The column descriptor of a column of a table other than *T*.

   c) The descriptor of an SQL parameter or the result type included in the routine descriptor of any <SQL-invoked routine>.

   d) The descriptor of an SQL parameter or the result type included in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

   e) The descriptor of any user-defined cast.

   NOTE 580 — A descriptor that "generally includes" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

9) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by *TN*.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) Let *STN* be the <table name> of any direct subtable of *T*. The following <drop table statement> is effectively executed without further Access Rule checking:

```
DROP TABLE STN CASCADE
```

2) For every proper supertable of *T*, every superrow of every row of *T* is effectively deleted at the end of the SQL-statement, prior to the checking of any integrity constraints.

   NOTE 581 — This deletion does not create a new state change in the most recent statement execution context.

3) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE ALL PRIVILEGES ON TN FROM A CASCADE
```

4) If *T* is a referenceable table, then:

   a) For every reference type descriptor *RTD* that includes a scope of *TN* and is generally included in any of the following:

      i) The attribute descriptor of an attribute of a user-defined type.

      ii) The column descriptor of a column of a table other than *T*.

      iii) The descriptor of an SQL parameter or the result type included in the routine descriptor of any SQL-invoked routine.

      iv) The descriptor of an SQL parameter or the result type in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

      v) The descriptor of any user-defined cast.

      the scope of *RTD* is made empty.

b) Let *SOD* be the descriptor of a schema object dependent on the table descriptor of *T*.

NOTE 582 — A descriptor that "depends on" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

Case:

i) If *SOD* is a view descriptor, then let *SON* be the name of the view included in *SOD*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

ii) If *SOD* is an assertion descriptor, then let *SON* be the name of the constraint included in *SOD*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

iii) If *SOD* is a table constraint descriptor, then let *SON* be the name of the constraint included in *SOD*. Let *CTN* be the <table name> included in the table descriptor that includes *SOD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

iv) If *SOD* is a routine descriptor, then let *SON* be the specific name included in *SOD*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

v) If *SOD* is a trigger descriptor, then let *SON* be the trigger name included in *SOD*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON
```

5) For each direct supertable *DST* of *T*, the table name of *T* is removed from the list of table names of direct subtables of *DST* that is included in the table descriptor of *DST*.

6) The descriptor of *T* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop table statement> that contains <drop behavior> that contains CASCADE.

11.32 <view definition>

This Subclause is modified by Subclause 10.11, "<view definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.3, "<view definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.4, "<view definition>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 11.2, "<view definition>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 11.6, "<view definition>", in ISO/IEC 9075-16.

Function

Define a viewed table.

Format

```
<view definition> ::=
  CREATE [ RECURSIVE ] VIEW <table name> <view specification>
      AS <query expression> [ WITH [ <levels clause> ] CHECK OPTION ]

<view specification> ::=
    <regular view specification>
  | <referenceable view specification>

<regular view specification> ::=
  [ <left paren> <view column list> <right paren> ]

<referenceable view specification> ::=
  OF <path-resolved user-defined type name> [ <subview clause> ]
      [ <view element list> ]

<subview clause> ::=
  UNDER <table name>

<view element list> ::=
  <left paren> <view element> [ { <comma> <view element> }... ] <right paren>

<view element> ::=
    <self-referencing column specification>
  | <view column option>

<view column option> ::=
  <column name> WITH OPTIONS <scope clause>

<levels clause> ::=
    CASCADED
  | LOCAL

<view column list> ::=
  <column name list>
```


Syntax Rules

1) Let *QE* be the <query expression>.

2) 04 *QE* shall not contain a <host parameter specification>, an <SQL parameter reference>, a <dynamic parameter specification>, or an <embedded variable specification>.

3) If a <view definition> is contained in a <schema definition> and the <table name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

4) [16] The schema identified by the explicit or implicit <schema name> of the <table name> shall not include a table descriptor whose table name is <table name>.

5) Let *TN* be the <table name>. There shall not exist a viewed table *V2* identified by <table name> *TN2* whose view descriptor includes an original <query expression> *QE2* such that *TN* is generally contained in *QE2* and *TN2* is generally contained in *QE*.

6) No <table reference> generally contained in *QE* shall identify any declared local temporary table.

7) No <table reference> generally contained in *QE* shall contain a <data change delta table>.

8) If a <table reference> generally contained in *QE* identifies the viewed table *VT* defined by <view definition> *VD*, then *VD* and *VT* are said to be *recursive*.

9) If *VD* is recursive, then:

   a) <view column list> shall be specified.

   b) RECURSIVE shall be specified.

   c) CHECK OPTION shall not be specified.

   d) <referenceable view specification> shall not be specified.

   e) *VD* is equivalent to

```
CREATE VIEW <table name> AS
    WITH RECURSIVE <table name> (<view column list>)
      AS (<query expression>)
    SELECT <view column list> FROM <table name>
```

10) The viewed table is *generally updatable* if *QE* is generally updatable.

11) The viewed table is *simply updatable* if *QE* is simply updatable.

12) The viewed table is *effectively updatable* if *QE* is effectively updatable.

13) The viewed table is *insertable-into* if *QE* is insertable-into.

14) The number of <column name>s in the <view column list> shall be the same as the degree of the table specified by *QE*.

15) Every column in the table specified by *QE* whose declared type is a character string type shall have a declared type collation.

16) If WITH CHECK OPTION is specified and <levels clause> is not specified, then a <levels clause> of CASCADED is implicit.

17) If CASCADED CHECK OPTION is specified or implied, then the viewed table shall be effectively updatable.

18) If LOCAL CHECK OPTION is specified, then the viewed table shall be simply updatable.

19) If CHECK OPTION is specified, then no generally underlying table of *QE* shall be a view whose view descriptor includes an indication that the view is trigger updatable, trigger insertable-into, or trigger deletable.

20) If LOCAL CHECK OPTION is specified, then *QE* shall not contain a potential source of non-determinism except for a leaf underlying table specification of *QE*, or a <column reference> whose implicit or explicit qualifying range variable is a leaf underlying table of *QE*.

    If CASCADED CHECK OPTION is specified, then *QE* shall not contain a potential source of non-determinism.

21) Let *V* be the view defined by the <view definition>. The *underlying columns* of every *i*-th column of *V* are the underlying columns of the *i*-th column of the <query expression> and the underlying columns of *V* are the underlying columns of the <query expression>. The *generally underlying columns* of every *i*-th column of *V* are the generally underlying columns of the *i*-th column of the <query expression> and the generally underlying columns of *V* are the generally underlying columns of the <query expression>.

22) <subview clause>, if present, identifies the *direct superview SV* of *V*. *V* is said to be a *direct subview* of *SV*. View *V1* is a *superview* of view *V2* if and only if one of the following is true:

a) *V1* and *V2* are the same view.

b) *V1* is a direct superview of *V2*.

c) There exists a view *V3* such that *V1* is a direct superview of *V3* and *V3* is a superview of *V2*.

If *V1* is a superview of *V2*, then *V2* is said to be a *subview* of *V1*.

If *V1* is a superview of *V2* and *V1* and *V2* are not the same view, then *V2* is a *proper subview* of *V1* and *V1* is a *proper superview* of *V2*.

If *V2* is a direct subview of *V1*, then *V2* is a direct subtable of *V1*.

NOTE 583 — It follows that the subviews of the superviews of *V* together constitute the subtable family of *V*, every implication of which applies.

23) If <referenceable view specification> is specified, then:

a) *V* is a *referenceable view*.

b) RECURSIVE shall not be specified.

c) [13] The <user-defined type name> simply contained in <path-resolved user-defined type name> shall identify a structured type *ST*.

d) The subtable family of *V* shall not include a member, other than *V* itself, whose associated structured type is *ST*.

e) If <subview clause> is not specified, then <self-referencing column specification> shall be specified.

f) Let *n* be the number of attributes of *ST*. Let A_i, 1 (one) ≤ *i* ≤ *n* be the attributes of *ST*.

g) Let *RT* be the row type of *QE*.

h) If <self-referencing column specification> is specified, then:

i) Exactly one <self-referencing column specification> shall be specified.

ii) <subview clause> shall not be specified.

iii) SYSTEM GENERATED shall not be specified.

iv) Let *RST* be the reference type REF(*ST*).

Case:

1) If USER GENERATED is specified, then:

A) *RST* shall have a user-defined representation.

B) Let *m* be 1 (one).

2) If DERIVED is specified, then:

A) *RST* shall have a derived representation.

B) Let *m* be 0 (zero).

i) If <subview clause> is specified, then:

i) The <table name> contained in the <subview clause> shall identify a referenceable table *SV* that is a view.

ii) *ST* shall be a direct subtype of the structured type of the direct supertable *SUPERT* of *V*.

iii) The SQL-schema identified by the explicit or implicit <schema name> of the <table name> of *V* shall include the descriptor of *SV*.

iv) Let *MSV* be the maximal superview of the subtable family of *V*. Let *MSVTY* be the structured type of *MSV*. Let *RMSV* be the reference type REF(*MSVTY*).

Case:

1) If *RMSV* has a user-defined representation, then let *m* be 1 (one).

2) Otherwise, *RMSV* has a derived representation. Let *m* be 0 (zero).

j) The degree of *RT* shall be $n+m$.

k) Let F_i, 1 (one) $\leq i \leq n$, be the fields of *RT*.

l) For *i* varying from 1 (one) to *n*:

i) The declared type $DDTF_{i+m}$ of F_{i+m} shall be compatible with the declared type $DDTA_i$ of A_i.

ii) The Syntax Rules of Subclause 9.30, "Data type identity", are applied with $DDTF_{i+m}$ as *TYPE1* and $DDTA_i$ as *TYPE2*.

m) *QE* shall consist of a single <query specification> *QS*.

n) The <from clause> of *QS* shall simply contain a single <table reference> *TR*.

o) *TR* shall immediately contain a <table or query name> that is a <table name>. Let *TQN* be the table identified by the <table or query name>. *TQN* is the *basis table* of *V*.

p) If *TQN* is a referenceable table, then *TR* shall simply contain ONLY.

q) *QS* shall not simply contain a <group by clause> or a <having clause>.

r) *QS* shall not generally contain an <aggregate function>.

s) If <self-referencing column specification> is specified, then

Case:

i) If *RST* has a user-defined representation, then:

1) *TQN* shall have a candidate key consisting of a single column *RC*.

2) Let *SS* be the first <select sublist> in the <select list> of *QS*.

3) *SS* shall consist of a single <cast specification> *CS* whose leaf column is *RC*.

NOTE 584 — "Leaf column of a <cast specification>" is defined in Subclause 6.13, "<cast specification>".

4) The declared type of F_1 shall be REF(*ST*).

ii) Otherwise, *RST* has a derived representation.

1) Let u be the number of attributes of the derived representation of *RST*. Let C_i, 1 (one) $\leq i \leq u$, be the columns of *V* that correspond to the attributes of the derived representation of *RST*.

2) For each i, 1 (one) $\leq i \leq u$, the <value expression> simply contained in the <derived column> that defines C_i shall be a column reference.

3) *TQN* shall have a candidate key consisting of some subset of the underlying columns of C_i, 1 (one) $\leq i \leq u$.

t) If <subview clause> is specified, then:

i) *TQN* shall be a proper subtable or a proper subview of the basis table of *SV*.

ii) If *SUPERT* is effectively updatable, then *QS* shall be effectively updatable.

iii) Let k be the number of columns of *SUPERT*. For all j, 1 (one) $< j \leq k$, if the j-th column of the original <query expression> *SOQE* of *SUPERT* is updatable, then let VE_j be the <value expression> simply contained in the j-th <derived column> simply contained in the <select list> simply contained in *SOQE*. Since VE_j is updatable, it is a column reference; let CN_j be the column name of VE_j. The <value expression> simply contained in the (j+m)-th <derived column> simply contained in the <select list> of *QS* shall be a column reference whose column name is CN_j and whose qualifying table is *TQN*.

NOTE 585 — This ensures that the updatable columns of *SUPERT* can be determined when the view is created and will not be subject to change as a result of adding the subview. It also ensures that UPDATE column privileges on *SUPERT* can be established solely by examining the original <query expression> of *SUPERT*, and need not change as the result of adding a subview. In more detail, the rule says that if a column of a referenceable view *RV* is a column reference to a column of the basis table of *RV*, then in every subview *SUBRV*, that column must be a column reference to the corresponding column in the basis table of *SUBRV*.

iv) If the view descriptor of *SUPERT* or any supertable of *SUPERT* includes an indication that LOCAL CHECK OPTION was specified, then *QS* shall not contain a potential source of non-determinism except for a leaf underlying table specification of *QE*.

NOTE 586 — This ensures that a view having LOCAL CHECK OPTION will remain deterministic (except possibly its leaf underlying table specification) when any subview is added.

v) If the view descriptor of *SUPERT* or any supertable of *SUPERT* includes an indication that CASCADED CHECK OPTION was specified, then *QS* shall not contain a potential source of non-determinism.

NOTE 587 — This ensures that a view having CASCADED CHECK OPTION will remain deterministic when any subview is added.

vi) If *SUPERT* or any supertable of *SUPERT* is referenced in any check constraint descriptor, assertion descriptor, or the original <query expression> of any view having CASCADED CHECK OPTION, then *QS* shall not contain a potential source of non-determinism.

NOTE 588 — This ensures that if a view must be deterministic for the sake of a check constraint, assertion, or some other view that has CASCADED CHECK OPTION, then the view will remain deterministic when any subview is added.

vii) If *SUPERT* or any supertable of *SUPERT* is referenced, except as a leaf underlying table, in the original <query expression> of any view having LOCAL CHECK OPTION, then *QS* shall not contain a potential source of non-determinism.

NOTE 589 — This ensures that if a view must be deterministic for the sake of some other view that has LOCAL CHECK OPTION, then the view will remain deterministic when any subview is added.

viii) If the view descriptor of *SUPERT* or any supertable of *SUPERT* includes an indication that LOCAL CHECK OPTION was specified, then *QS* shall be simply updatable.

NOTE 590 — When LOCAL CHECK OPTION is enforced during an update operation on a supertable of *QS*, LOCAL CHECK OPTION must also be enforced on any rows of *QS* that are subrows of the supertable. By an earlier Syntax Rule, this requires that *QS* be simply updatable.

ix) For every

1) Original <query expression> *OBJ* of the view descriptor of any view.

2) <search condition> *OBJ* of any constraint descriptor or assertion descriptor.

3) <triggered action> *OBJ* of any trigger descriptor.

4) SQL routine body *OBJ* of any routine descriptor.

5) 04 <parameter default> *OBJ* of any SQL parameter of any routine descriptor.

OBJ shall still satisfy the Syntax Rules and Conformance Rules applicable to *OBJ* under the assumption that *V* is created as a subtable of *SUPERT*.

NOTE 591 — This prevents the following general scenario: 1) create a referenceable view *SUPERT*; 2) create an SQL-schema object *OBJ* that depends on some property *P* of *SUPERT*; 3) create a subtable of *SUPERT* that lacks the property *P* on which *OBJ* depends, thereby invalidating *OBJ*. Examples of properties *P* are: containing a potential source of non-determinism; not invoking an SQL-invoked routine that possibly reads SQL-data; or the usage of domains.

u) If <view element list> *TEL1* is specified, then:

i) Let *r* be the number of <view column option>s. For every <view column option> VCO_j, 1 (one) $\le j \le r$, <column name> shall be equivalent to the <attribute name> of some originally-defined attribute of *ST*.

ii) Distinct <view column option>s contained in *TEL1* shall specify distinct <column name>s.

iii) Let CN_j, 1 (one) $\le j \le r$, be the <column name> contained in VCO_j and let SCL_j be the <scope clause> contained in VCO_j.

1) CN_j shall be equivalent to some <attribute name> of *ST*, whose declared type is some reference type $CORT_j$.

2) The <table name> contained in SCL_j shall identify a referenceable table *SRT*.

3) *SRT* shall be based on the referenced type of $CORT_j$.

NOTE 592 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

24) Case:

a) If <regular view specification> is specified, then:

i) If any two columns in the table specified by the <query expression> have equivalent <column name>s, or if any column of that table has an implementation-dependent name, then a <view column list> shall be specified.

ii) Equivalent <column name>s shall not be specified more than once in the <view column list>.

b) Otherwise,

Case:

i) If <subview clause> is specified, then the name of the self-referencing column of *SV* shall not be equivalent to the name of any attribute of *ST*.

ii) Otherwise, <self-referencing column specification> shall not be equivalent to the name of any attribute of *ST*.

25) A column of *V* is called an *updatable column* of *V* if its underlying column is updatable.

26) If the <view definition> is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>; otherwise, let *A* be the <authorization identifier> that owns the schema identified by the explicit or implicit <schema name> of the <table name>.

Access Rules

1) If a <view definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <table name>.

2) If <referenceable view specification> is specified, then the applicable privileges for *A* shall include USAGE on *ST*.

3) If <subview clause> is specified, then

Case:

a) If <view definition> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the schema shall include UNDER for *SV*.

b) Otherwise, the current privileges shall include UNDER for *SV*.

General Rules

1) A view descriptor *VD* is created that describes *V*. *VD* includes:

a) The <table name> *TN*.

b) Case:

i) If *RMSV* has a derived representation, then let *SL* be the <select list> simply contained in *QS*, and let *TE* be the <table expression> simply contained in *QS*. Let *IDV* be an implementation-dependent (UV101) <value expression> that computes the REF value that references a row of *V*. Let *OQE* be the <query expression>

```
SELECT IDV, SL
TE
```

ii) Otherwise, let *OQE* be *QE*.

c) *OQE*, as both the hierarchical <query expression> of the descriptor and the original <query expression> of the descriptor.

d) *QE*, as the user-specified <query expression> of the descriptor.

e) Case:

i) If <regular view specification> is specified, then the column descriptors taken from the table specified by the <query expression>.

Case:

1) If a <view column list> is specified, then the <column name> of the *i*-th column of the view is the *i*-th <column name> in that <view column list>.

2) Otherwise, the <column name>s of the view are the <column name>s of the table specified by the <query expression>.

ii) Otherwise:

1) A column descriptor in which:

A) The name of the column is

Case:

I) If <self-referencing column name> is specified, then <self-referencing column name>.

II) Otherwise, the name of the first column of the direct supertable of *V*.

B) The data type descriptor is that generated by the <data type> "REF(*ST*) SCOPE(*TN*)".

C) The nullability characteristic is *known not nullable*.

D) The ordinal position is 1 (one).

E) The column is indicated to be self-referencing.

2) The column descriptor *ODCD* of each inherited column and each originally-defined column *ODC* of *V* in which:

A) The <column name> included in *ODCD* is replaced by the <attribute name> of its corresponding attribute of *ST*.

B) The declared type included in *ODCD* is the declared type of its corresponding attribute of *ST*.

C) If the declared type of the column is a reference type and some VCO_i contains the <attribute name> of *ST* that corresponds to the column, then the (possibly empty) scope contained in the reference type descriptor immediately included in the column descriptor is replaced by SCO_i.

3) If DERIVED is specified, then an indication that the self-referencing column is a derived self-referencing column.

4) If USER GENERATED is specified, then an indication that the self-referencing column is a user-generated self-referencing column.

f) In each column descriptor, an indication that the column is updatable if *V* is effectively updatable and the corresponding column of *QE* is updatable.

g) An indication as to whether CHECK OPTION was omitted, specified with LOCAL, or specified with CASCADED.

h) An indication that *V* is not trigger insertable-into.

i) An indication that *V* is not trigger updatable.

j) An indication that *V* is not trigger deletable.

k) If *V* is insertable-into, then an indication that *V* is insertable-into.

l) If *V* is simply updatable, then an indication that *V* is simply updatable.

m) If *V* is effectively updatable, then an indication that *V* is effectively updatable.

2) Let *VN* be the <table name>. Let *HQE* be the hierarchical <query expression> included in the view descriptor *VD* of the view identified by *VN*. Let *OQE* be the original <query expression> included in *VD*.

a) Case:

i) If a <view column list> is specified, then let *VCL* be the <view column list> preceded by a <left paren> and followed by a <right paren>.

ii) If *V* is a referenceable view, then let *SRCN* be the name of the self-referencing column of *V*, let *n* be the number of attributes of *ST*, and let AN_1, ..., AN_n be the names of those attributes of *ST*. Let *VCL* be

```
( SRCN, AN1, ..., ANn )
```

iii) Otherwise, let *VCL* be the zero-length character string.

b) Case:

i) If *VN* is immediately contained in some SQL-schema statement, then *VN* identifies the view descriptor *VD*.

ii) If *VN* is immediately contained in a <table reference> that specifies ONLY, then *VN* references the same table as the <table reference>:

```
( OQE ) AS VN VCL
```

iii) Otherwise, *VN* references the same table as the <table reference>:

```
( HQE ) AS VN VCL
```

3) Let *UT* be the set of all distinct leaf underlying tables of the <query expression> of *V*. Let *CU* be the set of all columns from all tables in *UT*. For every column *CV* in *V*:

a) Let *CUCV* be the set of all columns in *CU* that are underlying columns of *CV*.

b) A set of column privilege descriptors with the grantor for each set to the special grantor value "_SYSTEM" is created as follows:

i) A column privilege descriptor is created that defines the privilege SELECT on *CV* to *A*. That privilege is grantable if and only if all the following are true:

1) The applicable privileges for *A* include grantable SELECT privileges on all of the columns in *CUCV*.

2) The applicable privileges for *A* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

3) The applicable privileges for *A* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

4) The applicable privileges for *A* include grantable SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

ii) If, for each column *CUCVC* in *CUCV*, the applicable privileges for *A* include REFERENCES privilege on *CUCVC*, and if, for every table *UTT* in *UT*, the applicable privileges for *A* include REFERENCES on some column in *UTT*, then a column privilege descriptor is created that defines the REFERENCES privilege on *CV* to *A*. That privilege is grantable if and only if all the following are true:

1) The applicable privileges for *A* include grantable REFERENCES privileges on all of the columns in *CUCV*.

2) The applicable privileges for *A* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

3) The applicable privileges for *A* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

4) The applicable privileges for *A* include grantable SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

4) A privilege descriptor is created that defines the privilege SELECT on *V* to *A*. That privilege is grantable if and only if the applicable privileges for *A* include grantable SELECT privilege on every column of *V*. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

5) A privilege descriptor is created that defines the privilege TRIGGER on *V* to *A*. That privilege is not grantable. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

6) If the applicable privileges for *A* include REFERENCES privilege on every column of *V*, then a privilege descriptor is created that defines the privilege REFERENCES on *V* to *A*. That privilege is grantable if and only if the applicable privileges for *A* include grantable REFERENCES privilege on every column of *V*. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

7) If *V* is effectively updatable, then the General Rules of Subclause 9.39, "Determination of view privileges", are applied with *V* as *VIEW*.

8) If *V* is a referenceable view, then a set of privilege descriptors with the grantor for each set to the special grantor value "_SYSTEM" are created as follows:

a) A privilege descriptor is created that defines the SELECT WITH HIERARCHY OPTION privilege on *V* to *A*. That privilege is grantable.

b) For every method *M* of the structured type identified by <path-resolved user-defined type name>, a privilege descriptor is created that defines the SELECT privilege on the table/privilege pair consisting of table *V* and method *M* to *A*. That privilege is grantable.

c) Case:

i) If <subview clause> is not specified, then a privilege descriptor is created that defines the UNDER privilege on *V* to *A*. That privilege is grantable.

ii) Otherwise, a privilege descriptor is created that defines the UNDER privilege on *V* to *A*. That privilege is grantable if and only if the applicable privileges for *A* include grantable UNDER privilege on the direct supertable of *V*.

9) If <subview clause> is specified, then let *ST* be the set of supertables of *V*. Let *PDS* be the set of privilege descriptors that define SELECT WITH HIERARCHY OPTION privilege on a table in *ST*.

10) For every privilege descriptor in *PDS*, with grantee *G* and grantor *A*,

Case:

a) If the privilege is grantable, then let *WGO* be "WITH GRANT OPTION".

b) Otherwise, let *WGO* be the zero-length character string.

The following <grant statement> is effectively executed without further Access Rule checking:

```
GRANT SELECT ON VN TO G WGO FROM A
```

11) If <subview clause> is specified, then for every proper superview *SUPERV* of *V*, the General Rules of Subclause 9.38, "Generation of the hierarchical <query expression> of a view", are applied with *SUPERV* as *VIEW*.

NOTE 593 — As a consequence of rewriting the hierarchical <query expression> of *SUPERV*, the interpretation of a reference to *SUPERV* in a <table reference> that does not specify ONLY has changed.

12) If the character representation of the user-specified <query expression> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — query expression too long for information schema (0100A)*.

NOTE 594 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature T131, "Recursive query", conforming SQL language shall not contain a <view definition> that immediately contains RECURSIVE.

2) Without Feature F751, "View CHECK enhancements", conforming SQL language shall not contain a <levels clause>.

3) Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain a <referenceable view specification>.

4) Without Feature F751, "View CHECK enhancements", conforming SQL language shall not contain <view definition> that contains a <query expression> that contains a <query expression> and contains CHECK OPTION.

5) Without Feature F751, "View CHECK enhancements", in conforming SQL language, if the <view definition>, or any supertable of the viewed table, specifies CASCADED CHECK OPTION, then the viewed table shall be simply updatable.

6) Without Feature F852, "Top-level ORDER BY in views", in conforming SQL language, a <query expression> immediately contained in a <view definition> shall not immediately contain an <order by clause>.

7) Without Feature F864, "Top-level OFFSET in views", in conforming SQL language, a <query expression> immediately contained in a <view definition> shall not immediately contain a <result offset clause>.

8) Without Feature F859, "Top-level FETCH FIRST in views", in conforming SQL language, a <query expression> immediately contained in a <view definition> shall not immediately contain a <fetch first clause>.

9) 14 15 Without Feature S081, "Subtables", conforming SQL language shall not contain a <subview clause>.

11.33 <drop view statement>

This Subclause is modified by Subclause 10.12, "<drop view statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.7, "<drop view statement>", in ISO/IEC 9075-16.

Function

Destroy a view.

Format

```
<drop view statement> ::=
  DROP VIEW <table name> <drop behavior>
```

Syntax Rules

1) Let *V* be the table identified by the <table name> and let *VN* be that <table name>. The schema identified by the explicit or implicit <schema name> of *VN* shall include the descriptor of *V*.

2) *V* shall be a viewed table.

3) An *impacted dereference operation* is any of the following:

   a) A <dereference operation> *DO*, where *V* is the scope of the reference type of the <reference value expression> immediately contained in *DO*.

   b) A <method reference> *MR*, where *V* is the scope of the reference type of the <value expression primary> immediately contained in *MR*.

   c) A <reference resolution> *RR*, where *V* is the scope of the reference type of the <reference value expression> immediately contained in *RR*.

4) If RESTRICT is specified, then *V* shall not have any proper subtables.

5) 04 If RESTRICT is specified, then *V* shall not be referenced and no impacted dereference operation shall be contained in any of the following:

   a) The original <query expression> of the view descriptor of any view other than *V*.

   b) The <search condition> of any constraint descriptor or assertion descriptor.

   c) The <triggered action> of any trigger descriptor.

   d) 16 The SQL routine body of any routine descriptor.

   NOTE 595 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

6) If RESTRICT is specified and *V* is a referenceable view, then *VN* shall not be the scope included in a reference type descriptor generally included in any of the following:

   a) The attribute descriptor of an attribute of a user-defined type.

   b) The column descriptor of a column of a table other than *V*.

   c) The descriptor of an SQL parameter or the result type included in the routine descriptor of any <SQL-invoked routine>.

d) The descriptor of an SQL parameter or the result type included in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

e) The descriptor of any user-defined cast.

NOTE 596 — A descriptor that "generally includes" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

7) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the table identified by *VN*.

Access Rules

1) The enabled authorization identifier shall include *A*.

General Rules

1) Let *SVN* be the <table name> of any direct subview of *V*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SVN CASCADE
```

2) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE ALL PRIVILEGES ON VN FROM A CASCADE
```

3) If *V* is a referenceable view, then:

a) For every reference type descriptor *RTD* that includes a scope of *VN* and is generally included in any of the following:

i) The attribute descriptor of an attribute of a user-defined type.

ii) The column descriptor of a column of a table other than *V*.

iii) The descriptor of an SQL parameter or the result type included in the routine descriptor of any SQL-invoked routine.

iv) The descriptor of an SQL parameter or the result type included in a method specification descriptor included in the user-defined type descriptor of any user-defined type.

v) The descriptor of any user-defined cast.

the scope of *RTD* is made empty.

b) Let *SOD* be the descriptor of a schema object dependent on the view descriptor of *V*.

NOTE 597 — A descriptor that "depends on" another descriptor is defined in Subclause 6.3.4, "Descriptors", in ISO/IEC 9075-1.

Case:

i) If *SOD* is a view descriptor, then let *SON* be the name of the view included in *SOD*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW SON CASCADE
```

ii) If *SOD* is an assertion descriptor, then let *SON* be the name of the constraint included in *SOD*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION SON CASCADE
```

iii) If *SOD* is a table constraint descriptor, then let *SON* be the name of the constraint included in *SOD*. Let *CTN* be the <table name> included in the table descriptor that includes *SOD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE CTN DROP CONSTRAINT SON CASCADE
```

iv) If *SOD* is a routine descriptor, then let *SON* be the specific name included in *SOD*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SON CASCADE
```

v) If *SOD* is a trigger descriptor, then let *SON* be the trigger name included in *SOD*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER SON
```

4) If *V* has a direct supertable *DST*, then:

a) The table name of *V* is removed from the list of table names of direct subtables of *DST* that is included in the table descriptor of *DST*.

b) For every superview *SV* of *DST* (including *DST* itself), the General Rules of Subclause 9.38, "Generation of the hierarchical <query expression> of a view", are applied with *SV* as *VIEW*.

5) The descriptor of *V* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop view statement> that contains a <drop behavior> that contains CASCADE.

11.34 <domain definition>

This Subclause is modified by Subclause 10.8, "<domain definition>", in ISO/IEC 9075-9.

Function

Define a domain.

Format

```
<domain definition> ::=
  CREATE DOMAIN <domain name> [ AS ] <predefined type>
      [ <default clause> ]
      [ <domain constraint>... ]
      [ <collate clause> ]

<domain constraint> ::=
  [ <constraint name definition> ] <check constraint definition> [
      <constraint characteristics> ]
```

Syntax Rules

1) 09 If a <domain definition> is contained in a <schema definition>, and if the <domain name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

2) If <constraint name definition> is specified and its <constraint name> contains a <schema name>, then that <schema name> shall be equivalent to the explicit or implicit <schema name> of the <domain name> of the domain identified by the containing <domain definition> or <alter domain statement>.

3) The schema identified by the explicit or implicit <schema name> of the <domain name> shall not include a domain descriptor whose domain name is equivalent to <domain name> nor a user-defined type descriptor whose user-defined type name is equivalent to <domain name>.

4) If <predefined type> specifies a <character string type> and does not specify <character set specification>, then the character set name of the default character set of the schema identified by the implicit or explicit <schema name> of <domain name> is implicit.

5) <collate clause> shall not be both specified in <predefined type> and immediately contained in <domain definition>. If <collate clause> is immediately contained in <domain definition>, then it is equivalent to specifying an equivalent <collate clause> in <predefined type>.

6) Let *D1* be some domain. *D1* is *in usage by* a domain constraint *DC* if and only if the <search condition> of *DC* generally contains the <domain name> either of *D1* or of some domain *D2* such that *D1* is in usage by some domain constraint of *D2*. No domain shall be in usage by any of its own constraints.

7) If <collate clause> is specified, then <predefined type> shall be a character string type.

8) For every <domain constraint> specified:

   a) If <constraint characteristics> is specified, then neither ENFORCED nor NOT ENFORCED shall be specified.

   b) If <constraint characteristics> is not specified, then INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

c) If <constraint name definition> is not specified, then a <constraint name definition> that contains an implementation-dependent (UV100) <constraint name> is implicit. The assigned <constraint name> shall obey the Syntax Rules of an explicit <constraint name>.

d) Let *S* be the schema identified by the explicit or implicit <schema name> of the <constraint name> contained in <domain constraint>. *S* shall not include a constraint descriptor whose constraint name is <constraint name>.

Access Rules

1) If a <domain definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <domain name>.

General Rules

1) A <domain definition> defines a domain.

2) A data type descriptor is created that describes the declared type of the domain being created.

3) A domain descriptor is created that describes the domain being created. The domain descriptor contains the name of the domain, the data type descriptor of the declared type, the value of the <default clause> if the <domain definition> immediately contains <default clause>, and a domain constraint descriptor for every immediately contained <domain constraint>.

4) A privilege descriptor is created that defines the USAGE privilege on this domain to the <authorization identifier> *A* of the schema or SQL-client module in which the <domain definition> appears. This privilege is grantable if and only if the applicable privileges for *A* include a grantable REFERENCES privilege for each <column reference> contained in the <search condition> of every domain constraint descriptor included in the domain descriptor and a grantable USAGE privilege for each <domain name>, <collation name>, <character set name>, and <transliteration name> contained in the <search condition> of every domain constraint descriptor included in the domain descriptor. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

5) A domain constraint descriptor is created that describes the domain constraint being defined. The domain constraint descriptor includes:

   a) The <constraint name> contained in the explicit or implicit <constraint name definition>.

   b) An indication of whether the constraint is deferrable or not deferrable.

   c) An indication of whether the initial constraint mode of the constraint is deferred or immediate.

   d) The <search condition> contained in the <domain definition> as the template <search condition>.

   e) The applicable <search condition>:

```
( 1=1 )
```

Conformance Rules

1) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <domain definition>.

2) Without Feature F692, “Extended collation support”, conforming SQL language shall not contain a <domain definition> that immediately contains a <collate clause>.

11.35 <alter domain statement>

Function

Change a domain and its definition.

Format

```
<alter domain statement> ::=
  ALTER DOMAIN <domain name> <alter domain action>

<alter domain action> ::=
    <set domain default clause>
  | <drop domain default clause>
  | <add domain constraint definition>
  | <drop domain constraint definition>
```

Syntax Rules

1) Let *D* be the domain identified by <domain name>. The schema identified by the explicit or implicit <schema name> of the <domain name> shall include the descriptor of *D*.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of <domain name>.

General Rules

1) The domain descriptor of *D* is modified as specified by <alter domain action>.

NOTE 598 — The changed domain descriptor of *D* is applicable to every column that is dependent on *D*.

Conformance Rules

1) Without Feature F711, "ALTER domain", conforming SQL language shall not contain an <alter domain statement>.

11.36 <set domain default clause>

Function

Set the default value in a domain.

Format

```
<set domain default clause> ::=
  SET <default clause>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *D* be the domain identified by the <domain name> in the containing <alter domain statement>.

2) The default value specified by the <default clause> is placed in the domain descriptor of *D*.

Conformance Rules

1) Without Feature F711, "ALTER domain", conforming SQL language shall not contain a <set domain default clause>.

11.37 <drop domain default clause>

Function

Remove the default clause of a domain.

Format

```
<drop domain default clause> ::=
  DROP DEFAULT
```

Syntax Rules

1) Let *D* be the domain identified by the <domain name> in the containing <alter domain statement>.

2) The descriptor of *D* shall contain a default value.

Access Rules

None.

General Rules

1) Let *C* be the set of columns whose column descriptors contain the <domain name> that identifies *D*.

2) For every column belonging to *C*, if the column descriptor does not already contain a default value, then the default value from the domain descriptor of *D* is placed in that column descriptor.

3) The default value is removed from the domain descriptor of *D*.

Conformance Rules

1) Without Feature F711, "ALTER domain", conforming SQL language shall not contain a <drop domain default clause>.

11.38 <add domain constraint definition>

Function

Add a constraint to a domain.

Format

```
<add domain constraint definition> ::=
  ADD <domain constraint>
```

Syntax Rules

1) Let *D* be the domain identified by the <domain name> in the <alter domain statement>.

2) Let *D1* be some domain. *D1* is *in usage by* a domain constraint *DC* if and only if the <search condition> of *DC* generally contains the <domain name> either of *D1* or of some domain *D2* such that *D1* is in usage by some domain constraint of *D2*. No domain shall be in usage by any of its own constraints.

3) *D* shall not be included in the column descriptor of any column of any global temporary table, created local temporary table, or declared local temporary table.

Access Rules

None.

General Rules

1) The constraint descriptor of the <domain constraint> is added to the domain descriptor of *D*.

2) If *DC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to known not nullable.

NOTE 599 — The nullability characteristic of a column is defined in Subclause 4.15, "Columns, fields, and attributes".

Conformance Rules

1) Without Feature F711, "ALTER domain", conforming SQL language shall not contain an <add domain constraint definition>.

11.39 <drop domain constraint definition>

Function

Destroy a constraint on a domain.

Format

```
<drop domain constraint definition> ::=
  DROP CONSTRAINT <constraint name>
```

Syntax Rules

1) Let *D* be the domain identified by the <domain name> *DN* in the containing <alter domain statement>.

2) Let *DC* be the descriptor of the constraint identified by <constraint name>.

3) *DC* shall be included in the domain descriptor of *D*.

Access Rules

None.

General Rules

1) The constraint descriptor *DC* is removed from the domain descriptor of *D*.

2) If *DC* causes some column *CN* to be known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of the column descriptor of *CN* is changed to possibly nullable.

   NOTE 600 — The nullability characteristic of a column is defined in Subclause 4.15, "Columns, fields, and attributes".

3) The descriptor of *DC* is destroyed.

Conformance Rules

1) Without Feature F711, "ALTER domain", conforming SQL language shall not contain a <drop domain constraint definition>.

2) Without Feature F491, "Constraint management", conforming SQL language shall not contain a <drop domain constraint definition>.

11.40 <drop domain statement>

This Subclause is modified by Subclause 10.13, "<drop domain statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.8, "<drop domain statement>", in ISO/IEC 9075-16.

Function

Destroy a domain.

Format

```
<drop domain statement> ::=
  DROP DOMAIN <domain name> <drop behavior>
```

Syntax Rules

1) Let *D* be the domain identified by <domain name> and let *DN* be that <domain name>. The schema identified by the explicit or implicit <schema name> of *DN* shall include the descriptor of *D*.

2) 04 If RESTRICT is specified, then *D* shall not be referenced in any of the following:

   a) A column descriptor.

   b) The original <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) 16 The SQL routine body of any routine descriptor.

3) Let *UA* be the <authorization identifier> that owns the schema identified by the <schema name> of the domain identified by *DN*.

Access Rules

1) The enabled authorization identifiers shall include *UA*.

General Rules

1) For every column descriptor *CD* that includes *DN*:

   a) Let *T* be the table described by the table descriptor that includes *CD*, and let *TN* be the table name of *T*.

   b) *CD* is modified as follows:

      i) *DN* is removed from *CD*. A copy of the data type descriptor of *D* is included in *CD*.

      ii) If *CD* does not include a <default clause> and the domain descriptor of *D* includes a <default clause>, then a copy of the <default clause> of *D* is included in *CD*.

      iii) For every domain constraint descriptor *DCD* included in the domain descriptor of *D*, for every domain constraint usage descriptor *DCU* included in *DCD*:

         1) Let *SC* be the applicable <search condition> included in *DCU*.

2) Let *TCD* be a <table constraint definition> consisting of a <constraint name definition> whose <constraint name> is implementation-dependent (UV100), whose <constraint characteristics> are the <constraint characteristics> of the domain constraint descriptor, and whose <table constraint> is:

```
CHECK ( SC )
```

3) If the applicable privileges for *UA* include all of the privileges necessary for *UA* to successfully execute the <alter table statement>

```
ALTER TABLE TN ADD TCD
```

then the following <alter table statement> is effectively executed with a current authorization identifier of *UA*:

```
ALTER TABLE TN ADD TCD
```

2) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON DOMAIN DN
FROM UA CASCADE
```

3) The descriptor of *D* is destroyed.

Conformance Rules

1) Without Feature F251, “Domain support”, conforming SQL language shall not contain a <drop domain statement>.

11.41 <character set definition>

Function

Define a character set.

Format

```
<character set definition> ::=
  CREATE CHARACTER SET <character set name> [ AS ]
      <character set source> [ <collate clause> ]

<character set source> ::=
  GET <character set specification>
```

Syntax Rules

1) If a <character set definition> is contained in a <schema definition> and if the <character set name> immediately contained in the <character set definition> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <schema definition>.

2) The schema identified by the explicit or implicit <schema name> of the <character set name> shall not include a character set descriptor whose character set name is <character set name>.

3) The character set *CS* identified by the <character set specification> contained in <character set source> shall have been associated with a privilege descriptor that was effectively defined by the <grant statement>

   ```
   GRANT USAGE ON CHARACTER SET CSN TO PUBLIC
   ```

   where *CSN* is a <character set name> that identifies *CS*.

4) If <collate clause> is specified, then the <collation name> contained in <collate clause> shall identify a collation descriptor *CD* included in the schema identified by the explicit or implicit <schema name> contained in the <collation name>. The collation shall be applicable to the character repertoire of the character set identified by <character set source>. The list of applicable character set names included in *CD* shall include one that identifies *CS*.

5) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of <character set name>.

Access Rules

1) If a <character set definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include USAGE on the character set identified by the <character set specification>.

General Rules

1) A <character set definition> defines a character set.

2) A character set descriptor is created for the defined character set.

3) The descriptor created for the character set being defined is identical to the descriptor for the character set identified by <character set specification>, except that the included character set name is <character set name> and, if <collate clause> is specified, then the included name of the default collation is the <collation name> contained in <collate clause>.

4) A privilege descriptor is created that defines the USAGE privilege on this character set to the <authorization identifier> of the <schema definition> or <SQL-client module definition> in which the <character set definition> appears. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM". This privilege is grantable.

Conformance Rules

1) Without Feature F451, “Character set definition”, conforming SQL language shall not contain a <character set definition>.

11.42 <drop character set statement>

This Subclause is modified by Subclause 10.14, "<drop character set statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.9, "<drop character set statement>", in ISO/IEC 9075-16.

Function

Destroy a character set.

Format

```
<drop character set statement> ::=
  DROP CHARACTER SET <character set name>
```

Syntax Rules

1) Let *C* be the character set identified by the <character set name> and let *CN* be the name of *C*.

2) The schema identified by the explicit or implicit <schema name> of *CN* shall include the descriptor of *C*.

3) The explicit or implicit <schema name> contained in *CN* shall not be equivalent to INFORMATION_SCHEMA.

4) 04 *C* shall not be referenced in any of the following:

   a) The data type descriptor included in any column descriptor.

   b) The data type descriptor included in any domain descriptor.

   c) The data type descriptor generally included in any user-defined type descriptor.

   d) The data type descriptor included in any field descriptor.

   e) The original <query expression> of any view descriptor.

   f) The <search condition> of any constraint descriptor.

   g) The collation descriptor of any collation.

   h) The transliteration descriptor of any transliteration.

   i) The SQL routine body, the <SQL parameter declaration>s, or the <returns data type> of any routine descriptor.

   j) 16 The <SQL parameter declaration>s or <returns data type> of any method specification descriptor.

5) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <character set name>.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the character set identified by *C*. The enabled authorization identifiers shall include *A*.

General Rules

1) If *C* is the current default character set of any active or dormant SQL-session, then an exception condition is raised: *invalid character set name (2C000)*.

2) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON CHARACTER SET CN
FROM A CASCADE
```

3) The descriptor of *C* is destroyed.

Conformance Rules

1) Without Feature F451, "Character set definition", conforming SQL language shall not contain a <drop character set statement>.

11.43 <collation definition>

Function

Define a collation.

Format

```
<collation definition> ::=
  CREATE COLLATION <collation name> FOR <character set specification>
      FROM <existing collation name> [ <pad characteristic> ]

<existing collation name> ::=
  <collation name>

<pad characteristic> ::=
    NO PAD
  | PAD SPACE
```

Syntax Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <collation name>.

2) If a <collation definition> is contained in a <schema definition> and if the <collation name> immediately contained in the <collation definition> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <schema definition>.

3) The schema identified by the explicit or implicit <schema name> of the <collation name> *CN* immediately contained in <collation definition> shall not include a collation descriptor whose collation name is *CN*.

4) The schema identified by the explicit or implicit <schema name> of the <collation name> *ECN* immediately contained in <existing collation name> shall include a collation descriptor whose collation name is *ECN*.

5) The collation identified by *ECN* shall be a collation whose descriptor includes a character repertoire name that is equivalent to that included in the descriptor of the character set identified by <character set specification>.

6) If <pad characteristic> is not specified, then the <pad characteristic> of the collation identified by *ECN* is implicit.

7) If NO PAD is specified, then the collation is said to have the NO PAD characteristic. If PAD SPACE is specified, then the collation is said to have the PAD SPACE characteristic.

Access Rules

1) If a <collation definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include USAGE on *ECN*.

General Rules

1) A <collation definition> defines a collation.

2) A privilege descriptor is created that defines the USAGE privilege on this collation for *A*. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

3) This privilege descriptor is grantable if and only if the USAGE privilege for *A* on the collation identified by *ECN* is grantable.

4) A collation descriptor is created for the defined collation.

5) The collation descriptor *CD* created is identical to the collation descriptor for *ECN*, except that the collation name included in *CD* is *CN* and, if <pad characteristic> is specified, then the pad characteristic included in *CD* is <pad characteristic>.

Conformance Rules

1) Without Feature F690, "Collation support", conforming SQL language shall not contain a <collation definition>.

11.44 <drop collation statement>

This Subclause is modified by Subclause 10.15, "<drop collation statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.10, "<drop collation statement>", in ISO/IEC 9075-16.

Function

Destroy a collation.

Format

```
<drop collation statement> ::=
  DROP COLLATION <collation name> <drop behavior>
```

Syntax Rules

1) Let *C* be the collation identified by the <collation name> and let *CN* be the name of *C*.

2) The schema identified by the explicit or implicit <schema name> of *CN* shall include the descriptor of *C*.

3) The explicit or implicit <schema name> contained in *CN* shall not be equivalent to INFORMATION_SCHEMA.

4) 04 If RESTRICT is specified, then *C* shall not be referenced in any of the following:

   a) Any character set descriptor.

   b) The triggered action of any trigger descriptor.

   c) The original <query expression> of any view descriptor.

   d) The <search condition> of any constraint descriptor.

   e) The SQL routine body, the <SQL parameter declaration>s, or the <returns data type> of any routine descriptor.

   f) 16 The <SQL parameter declaration>s or the <returns data type> of any method specification descriptor.

5) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the collation identified by *C*.

6) Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <collation name>.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) For every character set descriptor *CSD* that includes *CN*, *CSD* is modified such that it does not include *CN*; if *CSD* does not include any collation name, then *CSD* is modified to indicate that it utilizes the default collation for its character repertoire.

2) For every data type descriptor *DD* that includes *CN*, *DD* is modified such that it includes the collation name of the character set collation of the character set of *DD*.

NOTE 601 — This causes the column, domain, attribute, or field described by *DD* to revert to the default collation for its character set.

3) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON COLLATION CN
FROM A CASCADE
```

4) The descriptor of *C* is destroyed.

Conformance Rules

1) Without Feature F690, “Collation support”, conforming SQL language shall not contain a <drop collation statement>.

11.45 <transliteration definition>

Function

Define a character transliteration.

Format

```
<transliteration definition> ::=
  CREATE TRANSLATION <transliteration name> FOR <source character set specification>
      TO <target character set specification> FROM <transliteration source>

<source character set specification> ::=
  <character set specification>

<target character set specification> ::=
  <character set specification>

<transliteration source> ::=
    <existing transliteration name>
  | <transliteration routine>

<existing transliteration name> ::=
  <transliteration name>

<transliteration routine> ::=
  <specific routine designator>
```

Syntax Rules

1) If a <transliteration definition> is contained in a <schema definition> and if the <transliteration name> immediately contained in the <transliteration definition> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <schema definition>.

2) The schema identified by the explicit or implicit <schema name> of the <transliteration name> *TN* immediately contained in <transliteration definition> shall not include a transliteration descriptor whose transliteration name is *TN*.

3) The schema identified by the explicit or implicit <schema name> of the <character set name> *SCSN* contained in the <character set specification> contained in <source character set specification> shall include a character set descriptor whose character set name is *SCSN*.

4) The schema identified by the explicit or implicit <schema name> of the <character set name> *TCSN* contained in the <character set specification> contained in <target character set specification> shall include a character set descriptor whose character set name is *TCSN*.

5) If <existing transliteration name> *ETN* is specified, then:

   a) The schema identified by the explicit or implicit <schema name> of *ETN* shall include a transliteration descriptor whose transliteration name is *ETN*.

   b) The character set identified by *SCSN* shall have the same character repertoire and character encoding form as the source character set of the transliteration identified by *ETN*.

   c) The character set identified by *TCSN* shall have the same character repertoire and character encoding form as the target character set of the transliteration identified by *ETN*.

6) If <transliteration routine> is specified, then:

   a) The schema identified by the explicit or implicit <schema name> of the <specific routine designator> *SRD* contained in <transliteration routine> shall include a routine descriptor that identifies a routine having a <specific routine designator> *SRD*.

   b) The routine identified by *SRD* shall be an SQL-invoked function that has one parameter whose declared type is character string and whose character set is the character set specified by *SCSN*; the <returns type> of the routine shall be character string whose character set is the character set specified by *TCSN*.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of <transliteration name>. If a <transliteration definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) If <existing transliteration name> is specified, then the applicable privileges for *A* shall include USAGE on the transliteration identified by *ETN*.

3) If <transliteration routine> is specified, then the applicable privileges for *A* shall include EXECUTE on the routine identified by *SRD*.

General Rules

1) A <transliteration definition> defines a transliteration.

2) If <transliteration source> contains <existing transliteration name>, then let *SRDN* be the specific name included in the transliteration descriptor whose transliteration name is *TN*; otherwise, let *SRDN* be the specific name of the SQL-invoked routine identified by <transliteration routine>.

3) A transliteration descriptor is created that includes:

   a) The name of the transliteration *TN*.

   b) The name of the character set *SCSN* from which it translates.

   c) The name of the character set *TCSN* to which it translates.

   d) *SRDN*, the specific name of the SQL-invoked routine that performs the transliteration.

4) A privilege descriptor *PD* is created that defines the USAGE privilege on this transliteration to the <authorization identifier> of the <schema definition> or <SQL-client module definition> in which the <transliteration definition> appears. The grantor of the privilege descriptor is set to the special grantor value "_SYSTEM".

5) *PD* is grantable if and only if the USAGE privilege for the <authorization identifier> of the <schema definition> or <SQL-client module definition> in which the <transliteration definition> appears is also grantable on every character set identified by a <character set name> contained in the <transliteration definition>.

Conformance Rules

1) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <transliteration definition>.

11.46 <drop transliteration statement>

This Subclause is modified by Subclause 10.16, "<drop transliteration statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.11, "<drop transliteration statement>", in ISO/IEC 9075-16.

Function

Destroy a character transliteration.

Format

```
<drop transliteration statement> ::=
  DROP TRANSLATION <transliteration name>
```

Syntax Rules

1) Let *T* be the transliteration identified by the <transliteration name> and let *TN* be the name of *T*.

2) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the transliteration identified by *TN*.

3) 04 The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *T*.

4) *T* shall not be referenced in any of the following:

   a) The triggered action of any trigger descriptor.

   b) The original <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) The collation descriptor of any collation.

   e) The transliteration descriptor of any translation.

   f) 16 The SQL routine body of any routine descriptor.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE USAGE ON TRANSLATION TN
FROM A CASCADE
```

2) The descriptor of *T* is destroyed.

Conformance Rules

1) Without Feature F695, “Translation support”, conforming SQL language shall not contain a <drop transliteration statement>.

11.47 <assertion definition>

This Subclause is modified by Subclause 10.17, "<assertion definition>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.9, "<assertion definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.5, "<assertion definition>", in ISO/IEC 9075-14.

Function

Specify an integrity constraint.

Format

```
14 <assertion definition> ::=
  CREATE ASSERTION <constraint name>
       CHECK <left paren> <search condition> <right paren>
       [ <constraint characteristics> ]
```

Syntax Rules

1) If an <assertion definition> is contained in a <schema definition> and if the <constraint name> contains a <schema name>, then that <schema name> shall be equivalent to the explicit or implicit <schema name> of the containing <schema definition>.

2) The schema identified by the explicit or implicit <schema name> of the <constraint name> shall not include a constraint descriptor whose constraint name is <constraint name>.

3) If <constraint characteristics> is specified, then neither ENFORCED nor NOT ENFORCED shall be specified.

4) If <constraint characteristics> is not specified, then INITIALLY IMMEDIATE NOT DEFERRABLE is implicit.

5) 04 The <search condition> shall not contain a <host parameter name>, an <SQL parameter name>, an <embedded variable specification>, a <dynamic parameter specification>, or a <column reference> that references a system-time period start column or a system-time period end column of any system-versioned table.

   NOTE 602 — <SQL parameter name> is excluded because of the scoping rules for <SQL parameter name>.

6) No <query expression> in the <search condition> shall reference a temporary table.

7) The <boolean value expression> that is simply contained in the <search condition> shall be retrospectively deterministic.

   NOTE 603 — "retrospectively deterministic" is defined in Subclause 6.46, "<boolean value expression>".

8) The <search condition> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

9) 09 14 The <search condition> shall not generally contain a <routine invocation> whose subject routine is an external routine that possibly reads SQL-data.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the <assertion definition>. If an <assertion definition> is contained in an <SQL-client module definition>, then the enabled authorization identifier shall include *A*.

General Rules

1) An <assertion definition> defines an assertion. An assertion is a constraint.

2) Let *SC* be the <search condition> simply contained in the <assertion definition>.

3) An assertion descriptor is created that describes the assertion being defined. This descriptor includes:

   a) The <constraint name>.

   b) Whether the constraint is deferrable or not deferrable, as specified in <constraint characteristics>.

   c) The initial constraint mode, as specified in <constraint characteristics>.

   d) The applicable <search condition> *SC*.

4) If the character representation of *SC* cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — search condition too long for information schema (01009)*.

   NOTE 604 — The Information Schema is defined in ISO/IEC 9075-11.

5) If *SC* causes some column *CN* be to known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of *CN* is changed to known not nullable.

   NOTE 605 — The nullability characteristic of a column is defined in Subclause 4.15, "Columns, fields, and attributes".

6) If any of the <query expression>s contained in *SC* references a system-versioned table *T*, only those rows of *T* that correspond to the current system rows are considered while evaluating the assertion.

Conformance Rules

1) Without Feature F521, "Assertions", conforming SQL language shall not contain an <assertion definition>.

2) [14] Without Feature F672, "Retrospective CHECK constraints", conforming SQL language shall not contain an <assertion definition> that contains a potential source of non-determinism.

11.48 <drop assertion statement>

This Subclause is modified by Subclause 10.18, "<drop assertion statement>", in ISO/IEC 9075-4.

Function

Destroy an assertion.

Format

```
<drop assertion statement> ::=
  DROP ASSERTION <constraint name> [ <drop behavior> ]
```

Syntax Rules

1) Let *A* be the assertion identified by <constraint name> and let *AN* be the name of *A*.

2) The schema identified by the explicit or implicit <schema name> of *AN* shall include the descriptor of *A*.

3) If <drop behavior> is not specified, then RESTRICT is implicit.

4) If RESTRICT is specified or implied, then *AN* shall not be referenced in the SQL routine body of any routine descriptor.

5) If *QS* is a <query specification> that contains a column reference to a column *C* in its <select list> that is not contained in a <set function specification>, and if *G* is the set of columns defined by the <grouping column reference list> of *QS*, and if the assertion A is needed to conclude that $G \mapsto C$ is a known functional dependency in *QS*, then *QS* is said to be *dependent on A*.

6) If *V* is a view that contains a <query specification> that is dependent on *A*, then *V* is said to be *dependent on A*.

7) If *R* is an SQL routine whose <SQL routine body> contains a <query specification> that is dependent on *A*, then *R* is said to be *dependent on A*.

8) If *C* is a constraint or assertion whose <search condition> contains a <query specification> that is dependent on *A*, then *C* is said to be *dependent on A*.

9) If *T* is a trigger whose triggered action contains a <query specification> that is dependent on *A*, then *T* is said to be *dependent on A*.

10) If RESTRICT is specified or implicit, or <drop behavior> is not specified, then:

    a) No table constraint shall be dependent on *A*.

    b) No view shall be dependent on *A*.

    c) No SQL routine shall be dependent on *A*.

    d) No constraint or assertion shall be dependent on *A*.

    e) No trigger shall be dependent on *A*.

NOTE 606 — If CASCADE is specified, then any such dependent object will be dropped by the execution of the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the <schema name> of the assertion identified by *AN*.

General Rules

1) Let *RS* be the set of all SQL-invoked routines whose routine descriptor contains the <constraint name> of *A* in the <SQL routine body>.

2) For every routine *R* in *RS*:

   a) Let *SN* be the <specific name> of *R*.

   b) 04 The following <drop routine statement> is effectively executed without further Access Rule checking:

      ```
      DROP SPECIFIC ROUTINE SN CASCADE
      ```

3) For every view *V* that is dependent on *A*:

   a) Let *VN* be the table name of *V*.

   b) The following <drop view statement> is effectively executed:

      ```
      DROP VIEW VN CASCADE
      ```

4) For every SQL routine *SR* that is dependent on *A* or that contains a reference to *A*:

   a) Let *SRN* be the specific name of *SR*.

   b) The following <drop routine statement> is effectively executed:

      ```
      DROP SPECIFIC ROUTINE SRN CASCADE
      ```

5) For every table constraint *C* that is dependent on *A*:

   a) Let *CN* be the constraint name of *C* and let *TN* be the name of the table constrained by *C*.

   b) The following <alter table statement> is effectively executed:

      ```
      ALTER TABLE TN DROP CONSTRAINT CN CASCADE
      ```

6) For every assertion *A2* that is dependent on *A*:

   a) Let *AN2* be the constraint name of *A2*.

   b) The following <drop assertion statement> is effectively executed:

      ```
      DROP ASSERTION AN2 CASCADE
      ```

7) For every trigger *T* that is dependent on *A*:

   a) Let *TN* be the trigger name of *T*.

   b) The following <drop trigger statement> is effectively executed:

      ```
      DROP TRIGGER TN
      ```

8) Let *SC* be the <search condition> included in the descriptor of *A*. If *SC* causes some column *CN* be to known not nullable and no other constraint causes *CN* to be known not nullable, then the nullability characteristic of *CN* is changed to possibly nullable.

NOTE 607 — The nullability characteristic of a column is defined in Subclause 4.15, "Columns, fields, and attributes".

9) The descriptor of *A* is destroyed.

Conformance Rules

1) Without Feature F521, "Assertions", conforming SQL language shall not contain a <drop assertion statement>.

11.49 <trigger definition>

This Subclause is modified by Subclause 10.19, "<trigger definition>", in ISO/IEC 9075-4.

Function

Define triggered SQL-statements.

Format

```
<trigger definition> ::=
  CREATE TRIGGER <trigger name> <trigger action time> <trigger event>
      ON <table name> [ REFERENCING <transition table or variable list> ]
      <triggered action>

<trigger action time> ::=
    BEFORE
  | AFTER
  | INSTEAD OF

<trigger event> ::=
    INSERT
  | DELETE
  | UPDATE [ OF <trigger column list> ]

<trigger column list> ::=
  <column name list>

<triggered action> ::=
  [ FOR EACH { ROW | STATEMENT } ]
      [ <triggered when clause> ]
      <triggered SQL statement>

<triggered when clause> ::=
  WHEN <left paren> <search condition> <right paren>

<triggered SQL statement> ::=
    <SQL procedure statement>
  | <compound triggered SQL statement>

<compound triggered SQL statement> ::=
  BEGIN ATOMIC { <SQL procedure statement> <semicolon> }... END

<transition table or variable list> ::=
  <transition table or variable>...

<transition table or variable> ::=
    OLD [ ROW ] [ AS ] <old transition variable name>
  | NEW [ ROW ] [ AS ] <new transition variable name>
  | OLD TABLE [ AS ] <old transition table name>
  | NEW TABLE [ AS ] <new transition table name>

<old transition table name> ::=
  <transition table name>

<new transition table name> ::=
  <transition table name>

<transition table name> ::=
  <identifier>
```

```
<old transition variable name> ::=
  <correlation name>

<new transition variable name> ::=
  <correlation name>
```

Syntax Rules

1) Case:

   a) If a <trigger definition> is contained in a <schema definition> and if the <trigger name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

   b) If a <trigger definition> is contained in an <SQL-client module definition> and if the <trigger name> contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the <SQL-client module definition>.

2) Let *TN* be the <table name> of a <trigger definition>. The table *T* identified by *TN* is the *subject table* of the <trigger definition>.

3) The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *T*.

4) The schema identified by the explicit or implicit <schema name> of a <trigger name> *TRN* shall not include a trigger descriptor whose trigger name is *TRN*.

5) Case:

   a) If INSTEAD OF is specified, then *T* shall be a viewed table.

      Case:

      i) If <trigger event> specifies INSERT, then *T* shall not be trigger insertable-into.

      ii) If <trigger event> specifies UPDATE, then *T* shall not be trigger updatable.

      iii) If <trigger event> specifies DELETE, then *T* shall not be trigger deletable.

   b) Otherwise, *T* shall be a base table that is not a declared local temporary table.

6) If a <trigger column list> is specified, then:

   a) No <column name> shall appear more than once in the <trigger column list>.

   b) The <column name>s of the <trigger column list> shall identify columns of *T*.

7) If REFERENCING is specified, then:

   a) Let *OR*, *OT*, *NR*, and *NT* be the <old transition variable name>, <old transition table name>, <new transition variable name>, and <new transition table name>, respectively.

   b) OLD or OLD ROW, NEW or NEW ROW, OLD TABLE, and NEW TABLE shall be specified at most once each within the <transition table or variable list>.

   c) Case:

      i) If <trigger event> specifies INSERT, then neither OLD ROW nor OLD TABLE shall be specified.

      ii) If <trigger event> specifies DELETE, then neither NEW ROW nor NEW TABLE shall be specified.

d) No two of *OR*, *OT*, *NR*, and *NT* shall be equivalent.

e) Both *OR* and *NR* are range variables. The associated column lists of *OR* and *NR* comprise every column of the old transition table and the new transition table, respectively. The associated period lists of *OR* and *NR* comprise every period of the old transition table and the new transition table, respectively.

NOTE 608 — "range variable" is defined in Subclause 4.17.11, "Range variables".

f) The scope of *OR*, *OT*, *NR*, and *NT* is the <triggered action>, excluding any <SQL schema statement>s that are contained in the <triggered action>.

8) If neither FOR EACH ROW nor FOR EACH STATEMENT is specified, then FOR EACH STATEMENT is implicit.

9) If *OR* or *NR* is specified, then FOR EACH ROW shall be specified.

10) The <triggered action> shall not contain an <SQL parameter reference>, a <host parameter name>, a <dynamic parameter specification>, or an <embedded variable name>.

11) The <triggered when clause> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

12) If the descriptor of *T* includes a system-time period descriptor, then no <column reference> broadly contained in the <triggered action> shall reference the system-time period start column of *T* or the system-time period end column of *T*.

13) It is implementation-defined (IA046) whether the <triggered SQL statement> shall not broadly contain an <SQL transaction statement>, an <SQL connection statement>, an <SQL schema statement>, an <SQL dynamic statement>, or an <SQL session statement>.

NOTE 609 — These kinds of statements are prohibited at runtime by Subclause 15.21, "Execution of triggers", GR 6)e)iii). It is implementation-defined (IA046) whether the check is made as a Syntax Rule check as well.

14) If BEFORE is specified, then:

a) It is implementation-defined (IA047) whether the <triggered action> shall not contain an SQL-statement that possibly modifies SQL-data.

b) Neither OLD TABLE nor NEW TABLE shall be specified.

c) The <triggered action> shall not contain a <field reference> that references a field in the new transition variable corresponding to a generated column of *T*, system-time period start column of *T*, system-time period end column of *T*, *ATPN* period start column of *T*, or *ATPN* period end column of *T*, where *ATPN* is the <application time period name> included in a period descriptor, if any, contained in the descriptor of *T*.

15) 04 If INSTEAD OF is specified, then:

a) <triggered when clause> shall not be specified.

b) <trigger column list> shall not be specified.

c) <table name> shall not identify a view that is either recursive or referenceable, or whose view descriptor includes an indication that CHECK OPTION has been specified.

d) <table name> shall not identify a view that is identified by any generally underlying table specification of the original <query expression> of a view whose view descriptor includes an indication that CHECK OPTION has been specified.

e) <table name> shall not identify a view that is identified by a <target table> or <insertion target> *TT* of a <data change statement> *DCS* contained in a <data change delta table> that specifies

FINAL, or is a target generally underlying table of *TT*, where *DCS* is contained in any of the following:

i) The SQL routine body of any SQL routine.

ii) The triggered action of any trigger descriptor.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <trigger name> of the <trigger definition>. If a <trigger definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include TRIGGER on *T*.

3) 04 If the <triggered action> *TA* of a <trigger definition> contains an <old transition table name> *OTTN*, an <old transition variable name> *OTVN*, a <new transition table name> *NTTN*, or a <new transition variable name> *NTVN*, then the applicable privileges for *A* shall include SELECT on *T*.

General Rules

1) A <trigger definition> defines a trigger.

2) *OT* identifies the old transition table. *NT* identifies the new transition table. *OR* identifies the old transition variable. *NR* identifies the new transition variable.

NOTE 610 — "old transition table", "new transition table", "old transition variable", and "new transition variable" are defined in Subclause 4.46.1, "General description of triggers".

3) The transition table identified by *OT* is the table associated with *OR*. The transition table identified by *NT* is the table associated with *NR*.

4) If the character representation of the <triggered SQL statement> cannot be represented in the Information Schema without truncation, then a completion condition is raised: *warning — statement too long for information schema (0100F)*.

NOTE 611 — The Information Schema is defined in ISO/IEC 9075-11.

5) A trigger descriptor is created for <trigger definition>s as follows:

a) The trigger name included in the trigger descriptor is <trigger name>.

b) The subject table included in the trigger descriptor is <table name>.

c) The trigger action time included in the trigger descriptor is <trigger action time>.

d) If FOR EACH STATEMENT is specified or implicit, then an indication that the trigger is a statement-level trigger is included in the trigger descriptor; otherwise, an indication that the trigger is a row-level trigger is included in the trigger descriptor.

e) The trigger event included in the trigger descriptor is <trigger event>.

f) Any <old transition variable name>, <new transition variable name>, <old transition table name>, or <new transition table name> specified in the <trigger definition> is included in the trigger descriptor as the old transition variable name, new transition variable name, old transition table name, or new transition table name, respectively.

g) The triggered action included in the trigger descriptor is the specified <triggered action>.

h) If a <trigger column list> *TCL* is specified, then *TCL* is the trigger column list included in the trigger descriptor; otherwise, that trigger column list is empty.

i) The *triggered action column set* included in the trigger descriptor is the set of all distinct, fully qualified names of columns contained in the <triggered action>.

j) The timestamp of creation included in the trigger descriptor is the timestamp of creation of the trigger.

6) If INSTEAD OF is specified, then:

a) Let *VD* be the view descriptor of the view *V* identified by *TN*. Let *U* be the <authorization identifier> that owns the schema identified by the <schema name> of *TN*.

b) Case:

i) If the <trigger event> is INSERT, then

1) *VD* is modified to include an indication that *V* is trigger insertable-into.

2) A privilege descriptor is created that defines the privilege INSERT on *V* to *U*. This privilege is grantable. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

3) For each column *C* of *V*, a privilege descriptor is created that defines the privilege INSERT on *C* to *U*. This privilege is grantable. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

ii) If the <trigger event> is UPDATE, then

1) *VD* is modified to include an indication that *V* is trigger updatable.

2) A privilege descriptor is created that defines the privilege UPDATE on *V* to *U*. This privilege is grantable. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

3) For each column *C* of *V*, a privilege descriptor is created that defines the privilege UPDATE on *C* to *U*. This privilege is grantable. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

iii) If the <trigger event> is DELETE, then

1) *VD* is modified to include an indication that *V* is trigger deletable.

2) A privilege descriptor is created that defines the privilege DELETE on *V* to *U*. This privilege is grantable. The grantor of that privilege descriptor is set to the special grantor value "_SYSTEM".

Conformance Rules

1) Without Feature T200, "Trigger DDL", conforming SQL language shall not contain a <trigger definition>.

2) Without Feature T212, "Enhanced trigger capability", in conforming SQL language, a <triggered action> shall contain FOR EACH ROW.

3) Without Feature T213, "INSTEAD OF triggers", in conforming SQL language, a <trigger action time> shall not immediately contain INSTEAD OF.

4) Without Feature T214, "BEFORE triggers", conforming SQL language shall not contain a <trigger action time> that immediately contains BEFORE.

5) Without Feature T215, "AFTER triggers", conforming SQL language shall not contain a <trigger action time> that immediately contains AFTER.

6) Without Feature T216, "Ability to require true search condition before trigger is invoked", conforming SQL language shall not contain a <triggered when clause>.

11.50 <drop trigger statement>

Function

Destroy a trigger.

Format

```
<drop trigger statement> ::=
  DROP TRIGGER <trigger name>
```

Syntax Rules

1) Let *TR* be the trigger identified by the <trigger name> and let *TRN* be the name of *TR*.

2) The schema identified by the explicit or implicit <schema name> of *TRN* shall include the descriptor of *TR*.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of *TR*. The enabled authorization identifiers shall include *A*.

General Rules

1) If *TR* is an INSTEAD OF trigger, then:

   a) Let *V* be the subject table of *TR*. Let *VD* be the view descriptor of *V*. Let *VN* be the <table name> included in *VD*. Let *QE* be the original <query expression> included in *VD*. Let *U* be the <authorization identifier> that owns the schema identified by the <schema name> of *VN*.

   b) Case:

      i) If *TR* is an insert INSTEAD OF trigger, then:

         1) *VD* is modified to include an indication that *V* is not trigger insertable-into.

         2) Case:

            A) If *V* is not insertable-into, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

               ```
               REVOKE INSERT ON VN FROM U CASCADE
               ```

            B) Otherwise:

               I) The General Rules of Subclause 9.40, "Determination of view component privileges", are applied with *V* as *VIEW*; let a set of view component privilege descriptors be the *DESCRIPTOR SET* returned from the application of those General Rules.

               II) Case:

1) If there is no view component table privilege descriptor whose identified object is *QE*, whose action is INSERT, whose grantor is "_SYSTEM", and whose grantee is *U*, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE INSERT ON VN FROM U CASCADE
```

2) Otherwise, let *VCTPD* be the view component table privilege descriptor whose identified object is *QE*, whose action is INSERT, whose grantor is "_SYSTEM", and whose grantee is *U*. If *VCTPD* indicates that the privilege is not grantable, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE GRANT OPTION FOR INSERT ON VN FROM U CASCADE
```

ii) If *TR* is an update INSTEAD OF trigger, then:

1) *VD* is modified to include an indication that *V* is not trigger updatable.

2) Case:

A) If *V* is not effectively updatable, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE UPDATE ON VN FROM U CASCADE
```

B) Otherwise:

I) The General Rules of Subclause 9.40, "Determination of view component privileges", are applied with *V* as *VIEW*; let a set of view component privilege descriptors be the *DESCRIPTOR SET* returned from the application of those General Rules.

II) Case:

1) If there is no view component table privilege descriptor whose identified object is *QE*, whose action is UPDATE, whose grantor is "_SYSTEM", and whose grantee is *U*, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE UPDATE ON VN FROM U CASCADE
```

2) Otherwise, let *VCTPD* be the view component table privilege descriptor whose identified object is *QE*, whose action is UPDATE, whose grantor is "_SYSTEM", and whose grantee is *U*. If *VCTPD* indicates that the privilege is not grantable, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE GRANT OPTION FOR UPDATE ON VN FROM U CASCADE
```

iii) If *TR* is a delete INSTEAD OF trigger, then:

1) *VD* is modified to include an indication that *V* is not trigger deletable.

2) Case:

A) If *V* is not effectively updatable, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE DELETE ON VN FROM U CASCADE
```

B) Otherwise:

I) The General Rules of Subclause 9.40, "Determination of view component privileges", are applied with *V* as *VIEW*; let a set of view component privilege descriptors be the *DESCRIPTOR SET* returned from the application of those General Rules.

II) Case:

1) If there is no view component table privilege descriptor whose identified object is *QE*, whose action is DELETE, whose grantor is "_SYSTEM", and whose grantee is *U*, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE DELETE ON VN FROM U CASCADE
```

2) Otherwise, let *VCTPD* be the view component table privilege descriptor whose identified object is *QE*, whose action is DELETE, whose grantor is "_SYSTEM", and whose grantee is *U*. If *VCTPD* indicates that the privilege is not grantable, then the following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE GRANT OPTION FOR DELETE ON VN FROM U CASCADE
```

2) All view component privilege descriptors are destroyed, if any.

3) The descriptor of *TR* is destroyed.

Conformance Rules

1) Without Feature T200, "Trigger DDL", conforming SQL language shall not contain a <drop trigger statement>.

11.51 <user-defined type definition>

This Subclause is modified by Subclause 10.10, "<user-defined type definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.4, "<user-defined type definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.6, "<user-defined type definition>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 11.3, "<user-defined type definition>", in ISO/IEC 9075-15.

Function

Define a user-defined type.

Format

```
<user-defined type definition> ::=
  CREATE TYPE <user-defined type body>

13 <user-defined type body> ::=
  <schema-resolved user-defined type name>
      [ <subtype clause> ]
      [ AS <representation> ]
      [ <user-defined type option list> ]
      [ <method specification list> ]

<user-defined type option list> ::=
  <user-defined type option> [ <user-defined type option>... ]

<user-defined type option> ::=
    <instantiable clause>
  | <finality>
  | <reference type specification>
  | <cast to ref>
  | <cast to type>
  | <cast to distinct>
  | <cast to source>

<subtype clause> ::=
  UNDER <supertype name>

<supertype name> ::=
  <path-resolved user-defined type name>

<representation> ::=
    <predefined type>
  | <collection type>
  | <member list>

<member list> ::=
  <left paren> <member> [ { <comma> <member> }... ] <right paren>

<member> ::=
  <attribute definition>

<instantiable clause> ::=
    INSTANTIABLE
  | NOT INSTANTIABLE

<finality> ::=
    FINAL
  | NOT FINAL

<reference type specification> ::=
```

```
    <user-defined representation>
  | <derived representation>
  | <system-generated representation>

<user-defined representation> ::=
  REF USING <predefined type>

<derived representation> ::=
  REF FROM <list of attributes>

<system-generated representation> ::=
  REF IS SYSTEM GENERATED

<cast to ref> ::=
  CAST <left paren> SOURCE AS REF <right paren> WITH <cast to ref identifier>

<cast to ref identifier> ::=
  <identifier>

<cast to type> ::=
  CAST <left paren> REF AS SOURCE <right paren> WITH <cast to type identifier>

<cast to type identifier> ::=
  <identifier>

<list of attributes> ::=
  <left paren> <attribute name> [ { <comma> <attribute name> }... ] <right paren>

<cast to distinct> ::=
  CAST <left paren> SOURCE AS DISTINCT <right paren>
      WITH <cast to distinct identifier>

<cast to distinct identifier> ::=
  <identifier>

<cast to source> ::=
  CAST <left paren> DISTINCT AS SOURCE <right paren>
      WITH <cast to source identifier>

<cast to source identifier> ::=
  <identifier>

<method specification list> ::=
  <method specification> [ { <comma> <method specification> }... ]

13 <method specification> ::=
    <original method specification>
  | <overriding method specification>

<original method specification> ::=
  <partial method specification> [ SELF AS RESULT ] [ SELF AS LOCATOR ]
      [ <method characteristics> ]

<overriding method specification> ::=
  OVERRIDING <partial method specification>

<partial method specification> ::=
  [ INSTANCE | STATIC | CONSTRUCTOR ]
      METHOD <method name> <SQL parameter declaration list>
      <returns clause>
      [ SPECIFIC <specific method name> ]

<specific method name> ::=
  [ <schema name> <period> ] <qualified identifier>
```

```
<method characteristics> ::=
  <method characteristic>...

13 <method characteristic> ::=
     <language clause>
   | <parameter style clause>
   | <deterministic characteristic>
   | <SQL-data access indication>
   | <null-call clause>
```

Syntax Rules

1) Let *UDTD* be the <user-defined type definition>, let *UDTB* be the <user-defined type body> immediately contained in *UDTD*, let *UDTN* be the <schema-resolved user-defined type name> immediately contained in *UDTB*, let *SN* be the specified or implicit <schema name> of *UDTN*, let *SS* be the SQL-schema identified by *SN*, and let *UDT* be the data type defined by *UDTD*.

2) If *UDTD* is contained in a <schema definition> and *UDTN* contains a <schema name>, then that <schema name> shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

3) 13 *SS* shall not include a user-defined type descriptor or a domain descriptor whose name is equivalent to *UDTN*.

4) None of <instantiable clause>, <finality>, <reference type specification>, <cast to ref>, <cast to type>, <cast to distinct>, or <cast to source> shall be specified more than once.

5) Case:

   a) If <representation> specifies <predefined type> or <collection type>, then *UDTD* defines a *distinct type*.

   b) Otherwise, *UDTD* defines a *structured type*.

6) If <finality> specifies FINAL, then <instantiable clause> shall not specify NOT INSTANTIABLE.

7) If *UDTD* defines a distinct type, then:

   a) 15 Let *PSDT* be the data type identified by <predefined type> or <collection type>.

      Case:

      i) If *PSDT* is an exact numeric type, then let *SDT* be an implementation-defined (IV223) exact numeric type whose precision is equal to the precision of *PSDT* and whose scale is equal to the scale of *PSDT*.

      ii) If *PSDT* is an approximate numeric type, then let *SDT* be an implementation-defined (IV223) approximate numeric type whose precision is equal to the precision of *PSDT*.

      iii) Otherwise, let *SDT* be *PSDT*.

   b) <instantiable clause> shall not be specified.

   c) If <finality> is not specified, then FINAL is implicit; otherwise, FINAL shall be specified.

   d) <subtype clause> shall not be specified.

   e) <reference type specification> shall not be specified.

   f) Neither <cast to ref> nor <cast to type> shall be specified.

g) If <cast to distinct> is specified, then let *FNUDT* be <cast to distinct identifier>; otherwise, let *FNUDT* be the <qualified identifier> of *UDTN*.

h) If <cast to source> is specified, then let *FNSDT* be <cast to source identifier>; otherwise, the Syntax Rules of Subclause 9.9, "Type name determination", are applied with *SDT* as *TYPE*; let *FNSDT* be the *IDENTIFIER* returned from the application of those Syntax Rules.

8) If *UDTD* specifies a structured type, then:

a) Neither <cast to distinct> nor <cast to source> shall be specified.

b) If <subtype clause> is specified, then <reference type specification> shall not be specified.

c) If neither <subtype clause> nor <reference type specification> are specified, then <system-generated representation> is implicit.

d) If <instantiable clause> is not specified, then INSTANTIABLE is implicit.

e) <finality> shall be specified.

f) The *originally-defined attributes* of *UDT* are those defined by <attribute definition>s contained in <member list>. No two originally-defined attributes of *UDT* shall have equivalent <attribute name>s.

g) For each <attribute definition> *ATD* contained in <member list>, let *AN* be the <attribute name> contained in *ATD* and let *DT* be the <data type> contained in *ATD*. The following <original method specification>s are implicit:

```
METHOD AN ( )
    RETURNS DT
    LANGUAGE SQL
    DETERMINISTIC
    CONTAINS SQL
  RETURNS NULL ON NULL INPUT
```

This is the *original method specification* of the observer function of attribute *AN*.

```
METHOD AN ( ATTR DT )
    RETURNS UDTN
    SELF AS RESULT
    LANGUAGE SQL
    DETERMINISTIC
    CONTAINS SQL
  CALLED ON NULL INPUT
```

This is the *original method specification* of the mutator function of attribute *AN*.

h) If <user-defined representation> is specified, then:

i) Let *BT* be <predefined type>. *BT* is the *representation type of the referencing type* of *UDT*.

ii) *BT* shall be exact numeric or a character string type that is not a large object string type.

iii) If <cast to ref> is specified, then let *FNREF* be <cast to ref identifier>; otherwise, let *FNREF* be the <qualified identifier> of *UDTN*.

iv) Case:

1) If <cast to type> is specified, then let *FNTYP* be <cast to type identifier>.

2) Otherwise, the Syntax Rules of Subclause 9.9, "Type name determination", are applied with *BT* as *TYPE*; let *FNTYP* be the *IDENTIFIER* returned from the application of those Syntax Rules.

i) If <derived representation> is specified, then no two <attribute name>s in <list of attributes> shall be equivalent. The attributes identified by the <attribute name>s are the *derivational attributes of the derived representation*.

j) If <subtype clause> is specified, then:

i) <supertype name> shall not be equivalent to *UDTN*.

ii) The <supertype name> immediately contained in the <subtype clause> shall identify the descriptor of some structured type *SST*. *UDT* is a direct subtype of *SST*, and *SST* is a direct supertype of *UDT*.

iii) The descriptor of *SST* shall not include an indication that *SST* is final.

iv) The inherited attributes of *UDT* are the attributes described by the attribute descriptors included in the descriptor of *SST*.

v) If <member list> is specified, then no <attribute name> contained in <member list> shall have an attribute name that is equivalent to the attribute name of an inherited attribute.

vi) If the user-defined type descriptor of *SST* indicates that the referencing type of *SST* has a user-defined representation, then let *BT* be the data type described by the data type descriptor of the representation type of the referencing type of *SST* included in the user-defined type descriptor of *SST*.

1) If <cast to ref> is specified, then let *FNREF* be <cast to ref identifier>; otherwise, let *FNREF* be the <qualified identifier> of *UDTN*.

2) Case:

A) If <cast to type> is specified, then let *FNTYP* be <cast to type identifier>.

B) Otherwise, the Syntax Rules of Subclause 9.9, "Type name determination", are applied with *BT* as *TYPE*; let *FNTYP* be the *IDENTIFIER* returned from the application of those Syntax Rules.

k) If <cast to ref> or <cast to type> is specified, then exactly one of the following shall be true:

i) <user-defined representation> is specified.

ii) <subtype clause> is specified and the user-defined type descriptor of the direct supertype of *UDT* indicates that the referencing type of the direct supertype of *UDT* has a user-defined representation.

9) If <method specification list> is specified, then:

a) Let *M* be the number of <method specification>s MS_i, 1 (one) $\leq i \leq M$, contained in <method specification list>. Let MN_i be the <method name> of MS_i.

b) For *i* ranging from 1 (one) to *M*:

i) If MS_i does not specify INSTANCE, CONSTRUCTOR, or STATIC, then INSTANCE is implicit.

ii) If MS_i specifies STATIC, then:

1) None of SELF AS RESULT, SELF AS LOCATOR, and OVERRIDING shall be specified.

2) MS_i specifies a *static method*.

iii) If MS_i specifies CONSTRUCTOR, then:

1) SELF AS RESULT shall be specified.

2) OVERRIDING shall not be specified.

3) MN_i shall be equivalent to the <qualified identifier> of *UDTN*.

4) The <returns data type> shall specify *UDTN*.

5) *UDTD* shall define a structured type.

6) 13 MS_i specifies an *SQL-invoked constructor method*.

iv) Let RN_i be $SN.MN_i$.

v) If <specific method name> is not specified, then an implementation-dependent (UV102) <specific method name> whose <schema name> is equivalent to *SN* is implicit.

vi) If <specific method name> contains a <schema name>, then that <schema name> shall be equivalent to *SN*. If <specific method name> does not contain a <schema name>, then the <schema name> of *SN* is implicit.

vii) The schema identified by the explicit or implicit <schema name> of the <specific method name> shall not include a routine descriptor whose specific name is equivalent to <specific method name> or a user-defined type descriptor that includes a method specification descriptor whose specific method name is equivalent to <specific method name>.

viii) Let PDL_i be the <SQL parameter declaration list> contained in MS_i.

1) No two <SQL parameter name>s contained in PDL_i shall be equivalent.

2) No <SQL parameter name> contained in PDL_i shall be equivalent to SELF.

ix) Let N_i be the number of <SQL parameter declaration>s contained in MS_i. For every <SQL parameter declaration> $PD_{i,j}$, 1 (one) $\leq j \leq N_i$:

1) $PD_{i,j}$ shall not contain <parameter mode>. A <parameter mode> of IN is implicit.

2) $PD_{i,j}$ shall not specify RESULT.

3) <parameter type> $PT_{i,j}$ immediately contained in $PD_{i,j}$ shall not specify ROW.

4) If $PT_{i,j}$ simply contains <locator indication>, then:

A) MS_i shall not specify or imply LANGUAGE SQL.

B) 13 $PT_{i,j}$ shall specify either binary large object type, character large object type, array type, multiset type, or user-defined type.

x) If <returns data type> *RT* simply contains <locator indication>, then:

1) LANGUAGE SQL shall not be specified or implied.

2) *RT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

3) 13 <result cast> shall not be specified.

xi) If SELF AS RESULT is specified, then the <returns data type> shall specify *UDTN*.

xii) For *k* ranging from (*i*+1) to *M*, at least one of the following shall be false:

1) MN_i and the <method name> of MS_k are equivalent.

2) Both MS_i and MS_k either specify CONSTRUCTOR or neither specifies CONSTRUCTOR.

3) MS_k has N_i <SQL parameter declaration>s.

4) The data type of $PT_{i,j}$, 1 (one) ≤ j ≤ N_i, is compatible with $PT_{k,j}$.

xiii) 13 The *unaugmented SQL parameter declaration list* of MS_i is the <SQL parameter declaration list> contained in MS_i.

xiv) If MS_i specifies <original method specification>, then:

1) 13 The <method characteristics> of MS_i shall contain at most one <language clause>, at most one <parameter style clause>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, and at most one <null-call clause>.

2) 13 If <language clause> is not specified, then LANGUAGE SQL is implicit.

3) If <deterministic characteristic> is not specified, then NOT DETERMINISTIC is implicit.

4) <SQL-data access indication> shall be specified.

5) If <null-call clause> is not specified, then CALLED ON NULL INPUT is implicit.

6) Case:

A) If LANGUAGE SQL is specified or implied, then:

I) The <returns clause> shall not specify a <result cast>.

II) <SQL-data access indication> shall not specify NO SQL.

III) <parameter style clause> shall not be specified.

IV) Every <SQL parameter declaration> contained in <SQL parameter declaration list> shall contain an <SQL parameter name>.

B) Otherwise:

I) 13 If <parameter style> is not specified, then PARAMETER STYLE SQL is implicit.

II) If a <result cast> is specified, then let *V* be some value of the <data type> specified in the <result cast> and let *RT* be the <returns data type>. The following shall be valid according to the Syntax Rules of Subclause 6.13, "<cast specification>":

```
CAST ( V AS RT )
```

III) If <result cast from type> *RCT* simply contains <locator indication>, then *RCT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

7) Let a *conflicting method specification CMS* be a method specification that is included in the descriptor of a proper supertype of *UDT*, such that all of the following are true:

   A) The method names of *CMS* and MN_i are equivalent.

   B) *CMS* and MS_i have the same number of SQL parameters N_i.

   C) Let $PCMS_j$, 1 (one) ≤ j ≤ N_i, be the j-th SQL parameter in the unaugmented SQL parameter declaration list of *CMS*. Let $PMS_{i,j}$, 1 (one) ≤ j ≤ N_i, be the j-th SQL parameter in the unaugmented SQL parameter declaration list of MS_i.

   D) For j varying from 1 (one) to N_i, the declared type of $PCMS_j$ and the declared type of $PMS_{i,j}$ are compatible.

   E) MS_i does not specify CONSTRUCTOR.

   F) *CMS* and MS_i either both are not static methods or one of *CMS* and MS_i is a static method and the other is not a static method.

8) There shall be no conflicting method specification.

9) The *augmented SQL parameter declaration list* NPL_i of MS_i is defined as follows.

   Case:

   A) If MS_i specifies STATIC, then let NPL_i be:

      ($PD_{i,1}$, ..., PD_{i,N_i})

   B) If MS_i specifies SELF AS RESULT and SELF AS LOCATOR, then let NPL_i be:

      (SELF *UDTN* RESULT AS LOCATOR, $PD_{i,1}$, ..., PD_{i,N_i})

   C) If MS_i specifies SELF AS LOCATOR, then let NPL_i be:

      (SELF *UDTN* AS LOCATOR, $PD_{i,1}$, ..., PD_{i,N_i})

   D) If MS_i specifies SELF AS RESULT, then let NPL_i be:

      (SELF *UDTN* RESULT, $PD_{i,1}$, ..., PD_{i,N_i})

   E) Otherwise, let NPL_i be:

      (SELF *UDTN*, $PD_{i,1}$, ..., PD_{i,N_i})

10) Let AN_i be the number of <SQL parameter declaration>s in NPL_i.

11) If MS_i does not specify STATIC or CONSTRUCTOR, then there shall be no SQL-invoked function F that satisfies all the following conditions:

   A) The routine name of F and RN_i have equivalent <qualified identifier>s.

   B) If F is not a static method, then F has AN_i SQL parameters; otherwise, F has (AN_i−1) SQL parameters.

   C) The data type being defined is a proper subtype of

Case:

I) If *F* is not a static method, then the declared type of the first SQL parameter of *F*.

II) Otherwise, the user-defined type whose user-defined type descriptor includes the routine descriptor of *F*.

D) The declared type of the *j*-th SQL parameter in NPL_i, $2 \le j \le AN_i$ is compatible with

Case:

I) If *F* is not a static method, then the declared type of *j*-th SQL parameter of *F*.

II) Otherwise, the declared type of the (*j*−1)-th SQL parameter of *F*.

12) If MS_i specifies STATIC, then there shall be no SQL-invoked function *F* that is not a static method that satisfies all the following conditions:

A) The routine name of *F* and RN_i have equivalent <qualified identifier>s.

B) *F* has (AN_i+1) SQL parameters.

C) The data type being defined is a subtype of the declared type of the first SQL parameter of *F*.

D) The declared type of the *j*-th SQL parameter in *F*, $2 \le j \le (AN_i+1)$, is compatible with the declared type of the (*j*−1)-th SQL parameter of NPL_i.

xv) 13 If MS_i specifies <overriding method specification>, then:

1) MS_i shall not specify STATIC or CONSTRUCTOR.

2) A <returns clause> contained in MS_i shall not specify a <result cast> or <locator indication>.

3) Let the candidate original method specification *COMS* be an original method specification whose descriptor is included in the descriptor of a proper supertype of the user-defined type being defined, such that all of the following are true:

A) The <method name> of *COMS* and MN_i are equivalent.

B) *COMS* and MS_i have the same number of SQL parameters N_i.

C) Let $PCOMS_j$, 1 (one) $\le j \le N_i$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *COMS*. Let $POVMS_j$, 1 (one) $\le j \le N_i$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of MS_i.

D) For *j* varying from 1 (one) to N_i, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of $PCOMS_j$ as *TYPE1* and the declared type of $POVMS_j$ as *TYPE2*.

E) The descriptor of *COMS* shall not include an indication that STATIC or CONSTRUCTOR was specified.

4) There shall exist exactly one *COMS*.

5) *COMS* shall not be the corresponding method specification of a mutator or observer function.

NOTE 612 — "Corresponding method specification" is defined in Subclause 11.60, "<SQL-invoked routine>".

6) For *j* ranging from 1 (one) to N_i, all of the following shall be true:

A) If $POVMS_j$ contains an <SQL parameter name> *PNM1*, then $PCOMS_j$ contains an <SQL parameter name> that is equivalent to *PNM1*.

B) If $PCOMS_j$ contains an <SQL parameter name> *PNM2*, then $POVMS_j$ contains an <SQL parameter name> that is equivalent to *PNM2*.

C) If $POVMS_j$ contains a <locator indication>, then $PCOMS_j$ contains a <locator indication>.

D) If $PCOMS_j$ contains a <locator indication>, then $POVMS_j$ contains a <locator indication>.

7) Let *ROVMS* be the <returns data type> of MS_i. Let *RCOMS* be the <returns data type> of *COMS*.

Case:

A) If *RCOMS* is a user-defined type, then:

I) Let a *candidate overriding method specification COVRMS* be a method specification that is included in the descriptor of a proper supertype of *UDT*, such that all of the following are true:

1) The <method name> of *COVRMS* and MN_i are equivalent.

2) *COVRMS* and MS_i have the same number of SQL parameters N_i.

3) Let $PCOVRMS_j$, 1 (one) $\leq j \leq N_i$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *COVRMS*. Let $POVMS_j$, 1 (one) $\leq j \leq N_i$, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of MS_i.

4) For *j* varying from 1 (one) to N_i, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of $PCOVRMS_j$ as *TYPE1* and the declared type of $POVMS_j$ as *TYPE2*.

II) Let *NOVMS* be the number of candidate overriding method specifications. For *i* varying from 1 (one) to *NOVMS*, *ROVMS* shall be a subtype of the <returns data type> of the *i*-th candidate overriding method specification.

B) Otherwise, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with *RCOMS* as *TYPE1* and *ROVMS* as *TYPE2*.

8) The augmented SQL parameter declaration list *ASPDL* of MS_i is formed from the augmented SQL parameter declaration list of *COMS* by replacing the <data type> of the first parameter (named SELF) with *UDTN*.

9) There shall be no SQL-invoked function *F* that satisfies all the following conditions:

A) The routine name of *F* and the RN_i have equivalent <qualified identifier>s.

B) *F* and *ASPDL* have the same number *N* of SQL parameters.

C) The data type being defined is a proper subtype of the declared type of the first SQL parameter of *F*.

D) The declared type of $POVMS_i$, 1 (one) $\leq i \leq N$, is compatible with the declared type of SQL parameter P_{i+1} of *F*.

E) *F* is not an SQL-invoked method.

Access Rules

1) Let *A* be the <authorization identifier> that owns *SS*. If a <user-defined type definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include UNDER on the <user-defined type name> specified in <subtype clause>.

General Rules

1) A user-defined type descriptor *UDTDS* that describes *UDT* is created. *UDTDS* includes:

   a) The user-defined type name *UDTN*.

   b) If *UDT* is a distinct type or INSTANTIABLE is specified or implicit, then an indication that *UDT* is instantiable; otherwise, an indication that *UDT* is not instantiable.

   c) An indication of whether the user-defined type is final or not final.

   d) An indication of whether *UDT* is a distinct type or a structured type.

   e) If *UDT* is a distinct type, then the data type descriptor of *SDT*.

   f) If *UDT* is a structured type, then:

      i) For each inherited attribute *IA* of *UDT*, the attribute descriptor of *IA* and an indication that *IA* is an inherited attribute.

      ii) For each originally-defined attribute *ODA* of *UDT*, the attribute descriptor of *ODA* and an indication that *ODA* is an originally-defined attribute.

      iii) The name of the direct supertype of *UDT*.

      iv) A transform descriptor with an empty list of groups.

      v) Case:

         1) If <user-defined representation> is specified, then an indication that the referencing type of *UDT* has a user-defined representation, along with the data type descriptor of the representation type of the referencing type of *UDT*.

         2) If <derived representation> is specified, then an indication that the referencing type of *UDT* has a derived representation, along with the list of derivational attributes of the derived representation specified by <list of attributes>.

         3) Otherwise, an indication that the referencing type of *UDT* has a system-defined representation.

vi) If <subtype clause> is specified, then let *SUDT* be the direct supertype of *UDT* and let *DSUDT* be the user-defined type descriptor of *SUDT*. Let *RUDT* be the referencing type of *UDT* and let *RSUDT* be the referencing type of *SUDT*.

Case:

1) If *DSUDT* indicates that *RSUDT* has a user-defined representation, then an indication that *RUDT* has a user-defined representation and the data type descriptor of the representation type of *RSUDT* included in *DSUDT*.

2) If *DSUDT* indicates that *RSUDT* has a derived representation, then an indication that *RUDT* has a derived representation and the list of derivational attributes of the derived representation included in *DSUDT*.

3) If *DSUDT* indicates that *RSUDT* has a system-defined representation, then an indication that *RUDT* has a system-defined representation.

vii) The ordering form NONE.

viii) The ordering category STATE.

g) If <method specification list> is specified, then for every <original method specification> *ORMS* contained in <method specification list>, a method specification descriptor that includes:

i) The <method name> of *ORMS*.

ii) The <specific method name> of *ORMS*.

iii) An indication that the method specification is original.

iv) An indication of whether STATIC or CONSTRUCTOR is specified.

v) The <SQL parameter declaration list> contained in *ORMS* (augmented, if STATIC is not specified in *ORMS*, to include the implicit first parameter with parameter name SELF).

vi) For every <SQL parameter declaration> in the <SQL parameter declaration list>, a <locator indication>, if any.

vii) The <returns data type>.

viii) The <result cast from type>, if any.

ix) The <locator indication> contained in the <returns clause>, if any.

x) The <language name> contained in the explicit or implicit <language clause>.

xi) If the <language name> is not SQL, then the explicit or implicit <parameter style>.

xii) An indication of whether the method is deterministic.

xiii) An indication of whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

xiv) An indication of whether the method is to be invoked if any argument is the null value.

xv) The CURRENT_TIMESTAMP as the value of the creation timestamp.

xvi) The CURRENT_TIMESTAMP as the value of the last-altered timestamp.

h) If <method specification list> is specified, then for every <overriding method specification> *OVMS* contained in <method specification list>, let *DCMS* be the descriptor of the corresponding original method specification. The method specification descriptor of *OVMS* includes:

i) The <method name> of *OVMS*.

ii) The <specific method name> of *OVMS*.

iii) An indication that the method specification is overriding.

iv) The <SQL parameter declaration list> contained in *OVMS* (augmented to include the implicit first parameter with parameter name SELF).

v) For every <SQL parameter declaration> in the <SQL parameter declaration list>, a <locator indication>, if any.

vi) The <returns data type> of *OVMS*.

vii) The <result cast from type> included in *DCMS* (if any).

viii) The <locator indication> contained in the <returns clause> included in the *DCMS*, if any.

ix) The <language name> included in *DCMS*.

x) The <parameter style> included in *DCMS* (if any).

xi) The determinism indication included in *DCMS*.

xii) The SQL-data access indication included in *DCMS*.

xiii) The indication included in *DCMS*, whether the method is to be invoked if any argument is the null value.

xiv) The CURRENT_TIMESTAMP as the value of the creation timestamp.

xv) The CURRENT_TIMESTAMP as the value of the last-altered timestamp.

2) If *UDTD* specifies a distinct type, then:

a) The degree of *UDT* is 0 (zero).

b) The following SQL-statements are executed without further Access Rule checking:

```
CREATE FUNCTION SN.FNUDT ( SDTP SDT )
    RETURNS UDTN
    LANGUAGE SQL
    DETERMINISTIC
  RETURN RV1
CREATE FUNCTION SN.FNSDT ( UDTP UDTN )
    RETURNS SDT
    LANGUAGE SQL
    DETERMINISTIC
  RETURN RV2
CREATE CAST ( UDTN AS SDT )
    WITH FUNCTION FNSDT ( UDTN )
    AS ASSIGNMENT
CREATE CAST (SDT AS UDTN)
    WITH FUNCTION SN.FNUDT ( SDT )
    AS ASSIGNMENT
CREATE TRANSFORM FOR UDTN
    FNUDT ( FROM SQL WITH FUNCTION FNSDT ( UDTN ),
              TO SQL WITH FUNCTION SN.FNUDT(SDT) )
```

where: *SN* is the explicit or implicit <schema name> of *UDTN*; *RV1* is an implementation-dependent (UV103) <value expression> such that for every invocation of *SN.FNUDT* with argument value *AV1*, *RV1* evaluates to the representation of *AV1* in the data type identified by *UDTN*; *RV2* is an implementation-dependent (UV103) <value expression> such that for every invocation of *SN.FNSDT* with argument value *AV2*, *RV2* evaluates to the representation of *AV2* in the data type *SDT*, and *SDTP* and *UDTP* are <SQL parameter name>s arbitrarily chosen.

c) Case:

i) 09 If *SDT* is neither a large object type nor a collection type, then the following SQL-statement is executed without further Access Rule checking:

```
CREATE ORDERING FOR UDTN
ORDER FULL BY
MAP WITH FUNCTION FNSDT(UDTN)
FOR UDTN
```

ii) If *SDT* is a large object type, and the SQL-implementation supports Feature T042, "Extended LOB data type support", then the following SQL-statement is executed without further Access Rule checking:

```
CREATE ORDERING FOR UDTN
ORDER EQUALS ONLY BY
MAP WITH FUNCTION FNSDT(UDTN)
FOR UDTN
```

NOTE 613 — If *SDT* is a large object type, and the SQL-implementation does not support Feature T042, "Extended LOB data type support", then no ordering for *UDTN* is created.

3) If *UDTD* specifies a structured type, then:

a) Case:

i) If <subtype clause> is specified, then let *SST* be the direct supertype of *UDT* and let *D* be the degree of *SST*.

ii) Otherwise, let *D* be 0 (zero).

b) The degree of *UDT* is initially set to *D*; the General Rules of Subclause 11.52, "<attribute definition>", specify the degree of *UDT* during the definition of the attributes of *UDT*.

c) If INSTANTIABLE is specified, then let *V* be a value of the most specific type *UDT* such that, for every attribute *ATT* of *UDT*, invocation of the corresponding observer function on *V* yields the default value for *ATT*. The following <SQL-invoked routine> is effectively executed:

```
CREATE FUNCTION UDTN ( ) RETURNS UDTN
    RETURN V
```

This SQL-invoked function is the *constructor function* for *UDT*.

d) If <user-defined representation> is specified or if <subtype clause> is specified and the user-defined type descriptor of the direct supertype of *UDT* indicates that the referencing type of the direct supertype of *UDT* has a user-defined representation, then the following SQL-statements are executed without further Access Rule checking:

```
CREATE FUNCTION SN.FNREF ( BTP BT )
    RETURNS REF(UDTN)
    LANGUAGE SQL
    DETERMINISTIC
    STATIC DISPATCH
  RETURN RV1
CREATE FUNCTION SN.FNTYP ( UDTNP REF(UDTN) )
    RETURNS BT
    LANGUAGE SQL
    DETERMINISTIC
    STATIC DISPATCH
  RETURN RV2
CREATE CAST ( BT AS REF(UDTN) )
    WITH FUNCTION SN.FNREF(BT)
CREATE CAST ( REF(UDTN) AS BT )
    WITH FUNCTION SN.FNTYP(REF(UDTN) )
```

where: *SN* is the explicit or implicit <schema name> of *UDTN*; *RV1* is an implementation-dependent (UV103) <value expression> such that for every invocation of *SN.FNREF* with argument value *AV1*, *RV1* evaluates to the representation of *AV1* in the data type identified by REF(*UDTN*); *RV2* is an implementation-dependent (UV103) <value expression> such that for every invocation of *SN.FNTYP* with argument value *AV2*, *RV2* evaluates to the representation of *AV2* in the data type *BT*; and *UDTNP* is an <SQL parameter name> arbitrarily chosen.

4) A privilege descriptor is created that defines the USAGE privilege on *UDT* to *A*. This privilege is grantable. The grantor for this privilege descriptor is set to the special grantor value "_SYSTEM".

5) If *UDTD* specifies a structured type, then a privilege descriptor is created that defines the UNDER privilege on *UDT* to *A*. The grantor for the privilege descriptor is set to the special grantor value "_SYSTEM". This privilege is grantable if and only if *A* holds the UNDER privilege on the direct supertype of *UDT* WITH GRANT OPTION.

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <member list>.

2) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <instantiable clause> that contains NOT INSTANTIABLE.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <original method specification> that immediately contains SELF AS RESULT.

4) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <method characteristics> that contains a <parameter style> that contains GENERAL.

5) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <original method specification> that contains an <SQL-data access indication> that immediately contains NO SQL.

6) Without Feature S401, "Distinct types based on array types", in conforming SQL language, <representation> shall not contain <array type>.

7) Without Feature S402, "Distinct types based on multiset types", in conforming SQL language, <representation> shall not contain <multiset type>.

8) Without Feature T571, "Array-returning external SQL-invoked functions", conforming SQL language shall not contain a <method specification> that contains a <returns clause> for which any of the following are true:

   a) A <result cast from type> is specified that simply contains an <array type> and does not contain a <locator indication>.

   b) A <result cast from type> is not specified and <returns data type> simply contains an <array type> and does not contain a <locator indication>.

9) Without Feature T572, "Multiset-returning external SQL-invoked functions", conforming SQL language shall not contain a <method specification> that contains a <returns clause> for which any of the following are true:

   a) A <result cast from type> is specified that simply contains a <multiset type> and does not contain a <locator indication>.

   b) A <result cast from type> is not specified and <returns data type> simply contains a <multiset type> and does not contain a <locator indication>.

10) Without Feature S043, “Enhanced reference types”, conforming SQL language shall not contain a <reference type specification>.

11) Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <partial method specification> that contains INSTANCE or STATIC.

12) Without Feature S023, “Basic structured types”, conforming SQL language shall not contain a <method specification list>.

13) Without Feature S025, “Final structured types”, in conforming SQL language, a <user-defined type definition> that defines a structured type shall contain a <finality> that is NOT FINAL.

14) [13][14]Without Feature S028, “Permutable UDT options list”, conforming SQL language shall not contain a <user-defined type option list> in which <instantiable clause>, if specified, <finality>, <reference type specification>, if specified, <cast to ref>, if specified, <cast to type>, if specified, <cast to distinct>, if specified, and <cast to source>, if specified, do not appear in that sequence.

11.52 <attribute definition>

This Subclause is modified by Subclause 10.5, "<attribute definition>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.7, "<attribute definition>", in ISO/IEC 9075-14.

Function

Define an attribute of a structured type.

Format

```
13 <attribute definition> ::=
  <attribute name> <data type>
      [ <attribute default> ]
      [ <collate clause> ]

<attribute default> ::=
  <default clause>
```

Syntax Rules

1) 13 An <attribute definition> defines a certain component of some structured type. Let *UDT* be that structured type, let *UDTN* be its name, and let *SS* be the SQL-schema whose descriptor includes the descriptor of *UDT*.

2) Let *AN* be the <attribute name> contained in the <attribute definition>.

3) The declared type *DT* of the attribute is <data type>.

4) <collate clause> shall not be both specified in <data type> and immediately contained in <attribute definition>. If <collate clause> is immediately contained in <attribute definition>, then it is equivalent to specifying an equivalent <collate clause> in <data type>.

5) *DT* shall not be based on *UDT*.

NOTE 614 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

6) If *DT* is a <character string type> and does not contain a <character set specification>, then the default character set for *SS* is implicit.

Access Rules

None.

General Rules

1) A data type descriptor is created that describes *DT*.

2) Let *A* be the attribute defined by <attribute definition>.

3) The degree of *UDT* is increased by 1 (one).

4) An attribute descriptor is created that describes *A*. The attribute descriptor includes:

   a) *AN*, the name of the attribute.

   b) The data type descriptor of *DT*.

c) The ordinal position of the attribute.

NOTE 615 — The ordinal position of the attribute is equal to the degree of *UDT* at the time this <attribute definition> is being processed.

d) The implicit or explicit <attribute default>.

e) [13] The name *UDTN* of the user-defined type *UDT*.

5) An SQL-invoked method *OF* is created whose signature and result data type are as given in the descriptor of the original method specification of the observer function of *A*. Let *V* be a value in *UDT*.

Case:

a) If *V* is the null value, then the invocation *V.AN*() of *OF* returns the result of:

```
CAST (NULL AS DT)
```

b) [13] Otherwise, *V.AN*() returns the value of *A* in *V*.

NOTE 616 — The original method specification of the observer function of *A* is defined in the Syntax Rules of Subclause 11.51, "<user-defined type definition>".

NOTE 617 — The descriptor of *OF* is created under the General Rules of Subclause 11.60, "<SQL-invoked routine>".

6) An SQL-invoked method *MF* is created whose signature and result data type are as given in the descriptor of the original method specification of the mutator function of *A*. Let *V* be a value in *UDT* and let *AV* be a value in *DT*.

Case:

a) If *V* is the null value, then the invocation *V.AN*(*AV*) of *MF* raises an exception condition: *data exception — null value substituted for mutator subject parameter (2202D)*;

b) [13] Otherwise, the invocation *V.AN*(*AV*) returns *V2* such that *V2.AN*() = *AV* and for every other observer function *ANX* of *UDT*, *V2.ANX*() = *V.ANX*().

NOTE 618 — The original method specification of the mutator function of *A* is defined in the Syntax Rules of Subclause 11.51, "<user-defined type definition>".

NOTE 619 — The descriptor of *MF* is created under the General Rules of Subclause 11.60, "<SQL-invoked routine>".

Conformance Rules

1) Without Feature S023, "Basic structured types", conforming SQL language shall not contain an <attribute definition>.

2) Without Feature F692, "Extended collation support", conforming SQL language shall not contain an <attribute definition> that immediately contains a <collate clause>.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <attribute default>.

4) [14] Without Feature S026, "Self-referencing structured types", conforming SQL language shall not contain a <data type> simply contained in an <attribute definition> that is a <reference type> whose <referenced type> is equivalent to the <schema-resolved user-defined type name> simply contained in the <user-defined type definition> that contains <attribute definition>.

11.53 <alter type statement>

This Subclause is modified by Subclause 10.6, "<alter type statement>", in ISO/IEC 9075-13.

Function

Change the definition of a user-defined type.

Format

```
<alter type statement> ::=
  ALTER TYPE <schema-resolved user-defined type name> <alter type action>

<alter type action> ::=
    <add attribute definition>
  | <drop attribute definition>
  | <add original method specification>
  | <add overriding method specification>
  | <drop method specification>
```

Syntax Rules

1) [13]Let *DN* be the <schema-resolved user-defined type name> and let *D* be the data type identified by *DN*.

2) The schema identified by the explicit or implicit <schema name> of the <schema-resolved user-defined type name> shall include the descriptor of *D*. Let *S* be that schema.

3) The scope of the <schema-resolved user-defined type name> is the entire <alter type statement>.

4) If <alter type action> contains <add attribute definition>, <drop attribute definition>, or <add overriding method specification>, then *D* shall be a structured type.

5) Let *A* be the <authorization identifier> that owns the schema *S*.

Access Rules

1) If an <alter type statement> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

2) The applicable privileges for *A* shall include UNDER on each proper supertype of *D*.

General Rules

1) The user-defined type descriptor of *D* is modified as specified by <alter type action>.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain an <alter type statement>.

11.54 <add attribute definition>

Function

Add an attribute to a user-defined type.

Format

```
<add attribute definition> ::=
  ADD ATTRIBUTE <attribute definition>
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> immediately contained in the containing <alter type statement>. Let *SPD* be the set of all supertypes of *D*. Let *SBD* be the set of all subtypes of *D*.

2) Let *RD* be the reference type whose referenced type is *D*. Let *SPRD* be the set of all supertypes of *RD*. Let *SBRD* be the set of all subtypes of *RD*. Let *SPAD* be the set of all collection types whose element type is in *SPD* or *SPRD*. Let *SBAD* be the set of all collection types whose element type is in *SBD* or *SBRD*.

3) The declared type of a column of a base table shall not be in *SPRD*, *SBRD*, *SPAD*, or *SBAD*.

4) The declared type of a column of a base table shall not be based on *D*.

   NOTE 620 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

5) No referenceable table shall have a structured type in *SBD*.

6) Let *M* be the mutator function resulting from the <attribute definition>, had that <attribute definition> been simply contained in the <user-defined type definition> for *D*. There shall be no SQL-invoked routine *F* for which all of the following are true:

   a) The routine name included in the descriptor of *F* and the <schema qualified routine name> of *M* have equivalent <qualified identifier>s.

   b) *F* has two SQL parameters.

   c) The declared type of the first SQL parameter of *F* is a subtype or supertype of *D*.

   d) The declared type of the second SQL parameter of *F* is a compatible with the second SQL parameter of *M*.

7) Let *O* be the observer function resulting from the <attribute definition>, had that <attribute definition> been simply contained in the <user-defined type definition> for *D*. There shall be no SQL-invoked routine *F* for which all of the following are true:

   a) The <schema qualified routine name> of *O* and the routine name included in the descriptor of *F* have equivalent <qualified identifier>s.

   b) *F* has 1 (one) SQL parameter.

   c) The declared type of the first SQL parameter of *F* is a subtype or supertype of *D*.

Access Rules

None.

General Rules

1) The attribute defined by the <attribute definition> is added to *D*.

2) In all other respects, the specification of an <attribute definition> in an <alter type statement> has the same effect as specification of the <attribute definition> simply contained in the <user-defined type definition> for *D* would have had.

NOTE 621 — In particular, the degree of *D* is increased by 1 (one) and the ordinal position of that attribute is equal to the new degree of *D* as specified in the General Rules of Subclause 11.52, "<attribute definition>".

3) Let *A* be the attribute defined by <attribute definition>. Let *CPA* be a copy of the descriptor of *A*, modified to include an indication that the attribute is an inherited attribute.

4) For each proper subtype *PSBD* of *D*:

a) Let *DPSBD* be the descriptor of *PSBD*, let *N* be the number of attribute descriptors included in *DPSBD*, and let DA_i, 1 (one) $\leq i \leq N$, be the attribute descriptors included in *DPSBD*.

b) For every *i* between 1 (one) and *N*, if DA_i is the descriptor of an originally-defined attribute, then increase the ordinal position included in DA_i by 1 (one).

c) Include *CPA* in *DPSBD*.

Conformance Rules

None.

11.55 <drop attribute definition>

This Subclause is modified by Subclause 11.12, "<drop attribute definition>", in ISO/IEC 9075-16.

Function

Destroy an attribute of a user-defined type.

Format

```
<drop attribute definition> ::=
  DROP ATTRIBUTE <attribute name> RESTRICT
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> immediately contained in the containing <alter type statement>.

2) Let *A* be the attribute identified by the <attribute name> *AN*.

3) *A* shall be an attribute of *D* that is not an inherited attribute, and *A* shall not be the only attribute of *D*.

4) Let *SPD* be the set of all supertypes of *D*. Let *SBD* be the set of all subtypes of *D*. Let *RD* be the reference type whose referenced type is *D*. Let *SPRD* be the set of all supertypes of *RD*. Let *SBRD* be the set of all subtypes of *RD*. Let *SPAD* be the set of all collection types whose element type is in *SPD* or *SPRD*. Let *SBAD* be the set of all collection types whose element type is in *SBD* or *SBRD*.

5) The declared types of all columns of all base tables shall not be in *SPRD*, *SBRD*, *SPAD*, or *SBAD*.

6) The declared types of all columns of all base tables shall not be based on *D*.

   NOTE 622 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

7) No referenceable table shall have a structured type in *SBD*.

8) Let *R1* be the mutator function and let *R2* be the observer function of *A*.

   a) *R1* and *R2* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The original <query expression> of any view descriptor.

      iii) The <search condition> of any constraint descriptor.

      iv) [16] The triggered action of any trigger descriptor.

   b) The specific names of *R1* and *R2* shall not be included in any user-defined cast descriptor.

   c) *R1* and *R2* shall not be the ordering function in the descriptor of any user-defined type.

Access Rules

None.

General Rules

1) The descriptor of *A* is removed from the descriptor of every *SBD*.

2) The descriptor of *A* is destroyed.

3) The descriptors of the mutator and observer functions of *A* are destroyed.

4) The degree of every *SBD* is reduced by 1 (one). The ordinal position of all attributes having an ordinal position greater than the ordinal position of *A* in *SBD* is reduced by 1 (one).

Conformance Rules

None.

11.56 <add original method specification>

Function

Add an original method specification to a user-defined type.

Format

```
<add original method specification> ::=
  ADD <original method specification>
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> *DN* immediately contained in the containing <alter type statement>. Let *SN* be the specified or implied <schema name> of *DN*. Let *SPD* be the set of all supertypes of *D*, if any. Let *SBD* be the set of all subtypes of *D*, if any.

2) Let *ORMS* and *PORMS* be the <original method specification> and its immediately contained <partial method specification>, respectively.

3) Let *MN*, *MPDL* and *MCH* be the <method name>, the <SQL parameter declaration list> and the <method characteristics>, respectively, that are simply contained in *ORMS*. *MPDL* is called the *unaugmented SQL parameter declaration list* of *ORMS*.

4) If *PORMS* does not specify INSTANCE, CONSTRUCTOR, or STATIC, then INSTANCE is implicit.

5) If *PORMS* specifies CONSTRUCTOR, then:

   a) SELF AS RESULT shall be specified.

   b) *MN* shall be equivalent to the <qualified identifier> of *DN*.

   c) The <returns data type> shall specify *DN*.

   d) *D* shall be a structured type.

   e) *PORMS* specifies an *SQL-invoked constructor method*.

6) If *PORMS* specifies STATIC, then:

   a) Neither SELF AS RESULT nor SELF AS LOCATOR shall be specified.

   b) *PORMS* specifies a static method.

7) Let *RN* be *SN.MN*.

8) Case:

   a) If *PORMS* does not specify <specific method name>, then an implementation-dependent (UV102) <specific method name> is implicit whose <schema name> is equivalent to *SN*.

   b) Otherwise,

      Case:

      i) If <specific method name> contains a <schema name>, then that <schema name> shall be equivalent to *SN*.

      ii) Otherwise, the <schema name> *SN* is implicit.

The schema identified by the explicit or implicit <schema name> of the <specific method name> shall not include a routine descriptor whose specific name is equivalent to <specific method name> or a user-defined type descriptor that includes a method specification descriptor whose specific method name is equivalent to <specific method name>.

9) *MCH* shall contain at most one <language clause>, at most one <parameter style clause>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, and at most one <null-call clause>.

   a) If <language clause> is not specified in *MCH*, then LANGUAGE SQL is implicit.

   b) Case:

      i) If LANGUAGE SQL is specified or implied, then:

         1) <parameter style clause> shall not be specified.

         2) <SQL-data access indication> shall not specify NO SQL.

         3) Every <SQL parameter declaration> contained in <SQL parameter declaration list> shall contain an <SQL parameter name>.

         4) The <returns clause> shall not specify a <result cast>.

      ii) Otherwise:

         1) If <parameter style clause> is not specified, then PARAMETER STYLE SQL is implicit.

         2) If a <result cast> is specified, then let *V* be some value of the <data type> specified in the <result cast> and let *RT* be the <returns data type>. The following shall be valid according to the Syntax Rules of Subclause 6.13, "<cast specification>":

            ```
            CAST ( V AS RT )
            ```

         3) If <result cast from type> *RCT* simply contains <locator indication>, then *RCT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

   c) If <deterministic characteristic> is not specified in *MCH*, then NOT DETERMINISTIC is implicit.

   d) If <SQL-data access indication> is not specified, then CONTAINS SQL is implicit.

   e) If <null-call clause> is not specified in *MCH*, then CALLED ON NULL INPUT is implicit.

10) No two <SQL parameter name>s contained in *MPDL* shall be equivalent.

11) No <SQL parameter name> contained in *MPDL* shall be equivalent to SELF.

12) Let *N* be the number of <SQL parameter declaration>s contained in *MPDL*. For every <SQL parameter declaration> PD_j, 1 (one) $\leq j \leq N$:

   a) PD_j shall not contain <parameter mode>. A <parameter mode> of IN is implicit.

   b) PD_j shall not specify RESULT.

   c) <parameter type> PT_j immediately contained in PD_j shall not specify ROW.

   d) If PT_j simply contains <locator indication>, then:

      i) *MCH* shall not specify LANGUAGE SQL, nor shall LANGUAGE SQL be implied.

ii) PT_j shall specify either binary large object type, character large object type, array type, multiset type, or user-defined type.

13) If <returns data type> *RT* simply contains <locator indication>, then:

a) *MCH* shall not be specify LANGUAGE SQL, nor shall LANGUAGE SQL be implied.

b) *RT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

c) <result cast> shall not be specified.

14) If SELF AS RESULT is specified, then the <returns data type> shall specify *DN*.

15) Case:

a) If *ORMS* specifies CONSTRUCTOR, then let a *conflicting method specification CMS* be a method specification whose descriptor is included in the descriptor of *D*, such that all of the following are all true:

i) *MPDL* and the unaugmented SQL parameter declaration list of *CMS* have the same number *N* of SQL parameters.

ii) Let $PCMS_j$, 1 (one) ≤ *j* ≤ *N*, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *CMS*. Let PMS_j, 1 (one) ≤ *j* ≤ *N*, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list *MPDL*.

iii) For *j* varying from 1 (one) to *N*, the declared type of $PCMS_j$ and the declared type of PMS_j are compatible.

iv) *CMS* is an SQL-invoked constructor method.

b) Otherwise, let a *conflicting method specification CMS* be a method specification whose descriptor is included in the descriptor of some type in either *SPD* or *SBD*, such that the following are all true:

i) *MN* and the method name included in the descriptor of *CMS* are equivalent.

ii) *MPDL* and the unaugmented SQL parameter declaration list of *CMS* have the same number *N* of SQL parameters.

iii) Let $PCMS_j$, 1 (one) ≤ *j* ≤ *N*, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list of *CMS*. Let PMS_j, 1 (one) ≤ *j* ≤ *N*, be the *j*-th SQL parameter in the unaugmented SQL parameter declaration list *MPDL*.

iv) For *j* varying from 1 (one) to *N*, the declared type of $PCMS_j$ and the declared type of PMS_j are compatible.

v) *CMS* and *ORMS* either both are not instance methods or one of *CMS* and *ORMS* is a static method and the other is an instance method.

16) There shall be no conflicting method specification.

17) Let MP_i, 1 (one) ≤ *i* ≤ *N*, be the *i*-th <SQL parameter declaration> contained in *MPDL*. The augmented SQL parameter declaration list *NPL* of *ORMS* is defined as follows.

Case:

a) If *PORMS* specifies STATIC, then let *NPL* be:

(MP_1, ..., MP_N)

b) If *ORMS* specifies SELF AS RESULT and SELF AS LOCATOR, then let *NPL* be:

```
( SELF DN RESULT AS LOCATOR,
MP1, ...,
MPN )
```

c) If *ORMS* specifies SELF AS LOCATOR, then let *NPL* be:

```
( SELF DN AS LOCATOR,
MP1, ...,
MPN )
```

d) If *ORMS* specifies SELF AS RESULT, then let *NPL* be:

```
( SELF DN RESULT, MP1, ...,
MPN )
```

e) Otherwise, let *NPL* be:

```
( SELF DN, MP1, ...,
MPN )
```

Let *AN* be the number of <SQL parameter declaration>s in *NPL*.

18) If *PORMS* does not specify STATIC or CONSTRUCTOR, then there shall be no SQL-invoked function *F* for which all of the following are true:

a) *F* is not an SQL-invoked method.

b) The <routine name> of *F* and *RN* have equivalent <qualified identifier>s.

c) *F* has *AN* SQL parameters.

d) *D* is a subtype or supertype of the declared type of the first SQL parameter of *F*.

e) The declared type of the *i*-th SQL parameter in *NPL*, $2 \le i \le AN$ is compatible with the declared type of *i*-th SQL parameter of *F*.

19) If *PORMS* specifies STATIC, then there shall be no SQL-invoked function *F* that is not a static method for which all of the following are true:

a) The <routine name> of *F* and *RN* have equivalent <qualified identifier>s.

b) *F* has (*AN*+1) SQL parameters.

c) *D* is a subtype or supertype of the declared type of the first SQL parameter of *F*.

d) The declared type of the *i*-th SQL parameter of *F*, $2 \le i \le (AN+1)$, is compatible with the declared type of the $(i-1)$-th SQL parameter of *NPL*.

Access Rules

None.

General Rules

1) Let *STDS* be the descriptor of *D*. A method specification descriptor *DOMS* is created for *ORMS*. *DOMS* includes:

a) The <method name> *MN*.

b) The <specific method name> contained in *PORMS*.

c) An indication that the method specification is original.

d) An indication of whether STATIC or CONSTRUCTOR is specified.

e) The augmented SQL parameter declaration list *NPL*.

f) For every SQL parameter declaration in *NPL*, a locator indication (if specified).

g) The <returns data type> contained in *PORMS*.

h) The <result cast from type> contained in *PORMS* (if any).

i) The locator indication, if a <locator indication> is contained in the <returns clause> of *PORMS* (if any).

j) The <language name> explicitly or implicitly contained in *MCH*.

k) If the <language name> is not SQL, then the explicit or implicit <parameter style> contained in *MCH*.

l) The determinism indication contained in *MCH*.

m) An indication of whether the method possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

n) An indication of whether the method is to be invoked if any argument is the null value.

o) The CURRENT_TIMESTAMP as the value of the creation timestamp.

p) The CURRENT_TIMESTAMP as the value of the last-altered timestamp.

2) *DOMS* is added to *STDS*.

3) Let *N* be the number of table descriptors that include the user-defined type name of a subtype of *D*.

For *i* varying from 1 (one) to *N*:

a) Let TN_i be the object included in the *i*-th such table descriptor.

b) For every table privilege descriptor that specifies object TN_i and action SELECT, a new table/method privilege descriptor is created that specifies as object the table/method pair consisting of the table TN_i and method *DOMS*, action SELECT, the same grantor and grantee, and the same grantability.

Conformance Rules

None.

11.57 <add overriding method specification>

Function

Add an overriding method specification to a user-defined type.

Format

```
<add overriding method specification> ::=
  ADD <overriding method specification>
```

Syntax Rules

1) Let *OVMS* be the <overriding method specification> immediately contained in <add overriding method specification>. Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> *DN* immediately contained in the <alter type statement> containing *OVMS*. Let *SN* be the specified or implied <schema name> of *DN*. Let *SPD* be the set of all supertypes of *D*, if any. Let *SBD* be the set of all subtypes of *D*, if any.

2) Let *POVMS* be the <partial method specification> immediately contained in *OVMS*. *POVMS* shall not specify STATIC or CONSTRUCTOR.

3) Let *MN*, *RTC* and *MPDL* be <routine name>, the <returns clause> and the <SQL parameter declaration list> immediately contained in *POVMS*.

4) *MN* shall not be equivalent to the <qualified identifier> of the user-defined type name of any type in either *SPD* or *SBD* other than *D*.

5) Let *RN* be *SN.MN*.

6) Case:

   a) If *POVMS* does not specify <specific method name>, then an implementation-dependent (UV102) <specific method name> is implicit whose <schema name> is equivalent to *SN*.

   b) Otherwise,

      Case:

      i) If <specific method name> contains a <schema name>, then that <schema name> shall be equivalent to *SN*.

      ii) Otherwise, the <schema name> *SN* is implicit.

7) The schema identified by the explicit or implicit <schema name> of the <specific method name> shall not include a routine descriptor whose specific name is equivalent to <specific method name> or a user-defined type descriptor that includes a method specification descriptor whose specific method name is equivalent to <specific method name>.

8) *RTC* shall not specify a <result cast> or <locator indication>.

9) Let the *candidate original method specification COMS* be an original method specification that is included in the descriptor of a proper supertype of the user-defined type of *D*, such that all of the following are true:

   a) *MN* and the <method name> of *COMS* are equivalent.

b) Let *N* be the number of elements of the augmented SQL parameter declaration list *UPCOMS* generally included in the descriptor of *COMS*. *MPDL* contains (*N*−1) SQL parameter declarations.

c) For *i* varying from 2 to *N*, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of SQL parameter $PCOMS_i$ of *UPCOMS* as *TYPE1* and the declared type of SQL parameter $POVMS_{i-1}$ of *MPDL* as *TYPE2*.

d) The descriptor of *COMS* shall not include an indication that STATIC or CONSTRUCTOR was specified.

10) There shall exist exactly one such *COMS*.

11) *COMS* shall not be the corresponding method specification of a mutator or observer function.

NOTE 623 — "Corresponding method specification" is defined in Subclause 11.60, "<SQL-invoked routine>".

12) For *i* varying from 2 to *N*:

a) If $POVMS_{i-1}$ contains an <SQL parameter name> *PNM1*, then the *i*-th element of the augmented SQL parameter declaration list included in the descriptor of *COMS* shall have an SQL parameter name that is equivalent to *PNM1*.

b) If the *i*-th element of the augmented SQL parameter declaration list included in the descriptor of *COMS* has an SQL parameter name *PNM2*, then $POVMS_{i-1}$ shall contain an <SQL parameter name> that is equivalent to *PNM2*.

c) $POVMS_{i-1}$ shall not contain <parameter mode>. A <parameter mode> IN is implicit.

d) $POVMS_{i-1}$ shall not specify RESULT.

e) If the <parameter type> PT_{i-1} immediately contained in $POVMS_{i-1}$ contains a <locator indication>, then the *i*-th element of the augmented SQL parameter declaration list included in the descriptor of *COMS* shall include a <locator indication>.

f) If the *i*-th element of the augmented SQL parameter declaration list included in the descriptor of *COMS* includes a <locator indication>, then the <parameter type> PT_{i-1} immediately contained in $POVMS_{i-1}$ shall contain a <locator indication>.

13) Let *ROVMS* be the <returns data type> of *RTC*. Let *RCOMS* be the <returns data type> of *COMS*.

Case:

a) If *RCOMS* is a user-defined type, then:

i) Let a *candidate overriding method specification COVRMS* be a method specification that is included in the descriptor of a proper supertype or a proper subtype of *UDT*, such that all of the following are true:

1) The <method name> of *COVRMS* and *MN* are equivalent.

2) *COVRMS* and *OVMS* have the same number of SQL parameters *N*.

3) Let $PCOVRMS_i$, 1 (one) ≤ *i* ≤ *N*, be the *i*-th SQL parameter in the unaugmented SQL parameter declaration list of *COVRMS*. Let $POVMS_i$, 1 (one) ≤ *i* ≤ *N*, be the *i*-th SQL parameter in the unaugmented SQL parameter declaration list of *OVMS*.

4) For *i* varying from 1 (one) to *N*, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with the declared type of $PCOVRMS_i$ as *TYPE1* and the declared type of $POVMS_i$ as *TYPE2*.

ii) Let *NOVMS* be the number of candidate overriding method specifications. For *i* varying from 1 (one) to *NOVMS*, let $COVRMS_i$ be the *i*-th candidate overriding method specification.

Case:

1) If $COVRMS_i$ is included in the descriptor of a proper supertype of *D*, then *ROVMS* shall be a subtype of the <returns data type> of $COVRMS_i$.

2) Otherwise, *ROVMS* shall be a supertype of the <returns data type> of $COVRMS_i$.

b) Otherwise, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with *RCOMS* as *TYPE1* and *ROVMS* as *TYPE2*.

14) Let a *conflicting overriding method specification COVMS* be an overriding method specification that is included in the descriptor of *D*, such that all of the following are true:

a) *MN* and the method name of *COVMS* are equivalent.

b) The augmented SQL parameter declaration list of *COVMS* contains *N* elements.

c) For *i* varying from 2 to *N*, the data types of the SQL parameter $POVMS_{i-1}$ and the SQL parameter $PCOVMS_{i-1}$ of *COVMS* are compatible.

15) There shall be no conflicting overriding method specification.

16) The augmented SQL parameter declaration list *ASPDL* of *OVMS* is formed from the augmented SQL parameter declaration list of *COMS* by replacing the <data type> of the first parameter (named SELF) with the <schema-resolved user-defined type name> *DN*.

17) There shall be no SQL-invoked function *F* for which all of the following are true:

a) *F* is not an SQL-invoked method.

b) The <routine name> of *F* and the <routine name> *MS* have equivalent <qualified identifier>s.

c) Let *NPF* be the number of SQL parameters in *ASPDL*. *F* has *NPF* SQL parameters.

d) *D* is a subtype or supertype of the declared type of the first SQL parameter of *F*.

e) The declared type of the *i*-th SQL parameter in *ASPDL*, $2 \le i \le NPF$ is compatible with the declared type of *i*-th SQL parameter of *F*.

18) If the descriptor of *D* includes any method specification descriptor, then:

a) Let *M* be the number of method specification descriptors MSD_i, 1 (one) $\le i \le M$, included in the descriptor of *D*.

b) For *i* ranging from 1 (one) to *M*:

i) Let N_i be the number of <SQL parameter declaration>s contained in the augmented SQL parameter declaration list included in MSD_i. Let $PT_{i,j}$, 1 (one) $\le j \le N_i$, be the *j*-th <parameter type> contained in MSD_i.

ii) At least one of the following conditions shall be false:

1) The <routine name> included in MSD_i is equivalent to *MN*.

2) *ASPDL* has N_i <SQL parameter declaration>s.

3) The data type of $PT_{i,j}$, 1 (one) ≤ j ≤ N_i, is compatible with the data type of the j-th <SQL parameter declaration> of *MPDL*.

4) MSD_i does not include an indication that CONSTRUCTOR was specified.

Access Rules

None.

General Rules

1) Let *STDS* be the descriptor of *D*, and let *DCMS* be the descriptor of the corresponding original method specification *COMS*. A method specification descriptor *DOMS* is created for *OVMS*. *DOMS* includes:

   a) The <method name> *MN*.

   b) The <specific method name> contained in *POVMS*.

   c) An indication that the method specification is overriding.

   d) The augmented SQL parameter declaration list *APDL*.

   e) For every SQL parameter in *APDL*, the locator indication of the corresponding SQL parameter included in *DCMS* (if any).

   f) The <returns data type> contained in *POVMS*.

   g) The <result cast from type> included in *DCMS* (if any).

   h) The locator indication contained in the <returns clause> included in the *DCMS*.

   i) The <language name> included in *DCMS*.

   j) If the <language name> is not SQL, then the <parameter style> included in *DCMS*.

   k) The determinism indication included in *DCMS*.

   l) The SQL-data access indication included in *DCMS* (if any).

   m) The indication included in *DCMS* whether the method is to be invoked if any argument is the null value.

   n) The CURRENT_TIMESTAMP as the value of the creation timestamp.

   o) The CURRENT_TIMESTAMP as the value of the last-altered timestamp.

2) *DOMS* is added to *STDS*.

3) Let *N* be the number of table descriptors that include the user-defined type name of a subtype of *D*.

   For *i* varying from 1 (one) to *N*:

   a) Let TN_i be the object included in the *i*-th such table descriptor.

   b) Let *M* be the number of table/method privilege descriptors that specify as object the table/method pair consisting of TN_i and the method identified by the <specific method name> contained in *COMS*. For *j* varying from 1 (one) to *M*:

      i) Let $TMPD_j$ be the *j*-th such table/method privilege descriptor.

ii) A new table/method privilege descriptor is created that specifies as object the table/method pair consisting of TN_i and the method identified by the <specific method name> contained in *OVMS*, action SELECT, the same grantor and grantee, and the same grantability.

iii) $TMPD_j$ is deleted.

Conformance Rules

None.

11.58 <drop method specification>

Function

Remove a method specification from a user-defined type.

Format

```
<drop method specification> ::=
  DROP <specific method specification designator> RESTRICT

<specific method specification designator> ::=
  [ INSTANCE | STATIC | CONSTRUCTOR ]
      METHOD <method name> <data type list>
```

Syntax Rules

1) Let *D* be the user-defined type identified by the <schema-resolved user-defined type name> *DN* immediately contained in the <alter type statement> containing the <drop method specification> *DORMS*. Let *DSN* be the explicit or implicit <schema name> of *DN*. Let *SMSD* be the <specific method specification designator> immediately contained in *DORMS*.

2) If *SMSD* immediately contains a <specific method name> *SMN*, then:

   a) If *SMN* contains a <schema name>, then that <schema name> shall be equivalent to *DSN*. Otherwise, the <schema name> *DSN* is implicit.

   b) The descriptor of *D* shall include a method specification descriptor *DOOMS* whose specific method name is equivalent to *SMN*.

   c) Let *PDL* be the augmented SQL parameter declaration list included in *DOOMS*.

   d) Let *MN* be the <method name> included in *DOOMS*.

3) If *SMSD* immediately contains a <method name> *ME*, then:

   a) If none of INSTANCE, STATIC, or CONSTRUCTOR is immediately contained in *SMSD*, then INSTANCE is implicit.

   b) The descriptor of *D* shall include a method specification descriptor *DOOMS* whose method name *MN* is equivalent to *ME*.

   c) If *SMSD* immediately contains a <data type list> *DTL*, then

      Case:

      i) If STATIC is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes:

         1) An indication that the method specification is STATIC.

         2) An indication that the method specification is original.

         3) An augmented SQL parameter declaration list *PDL* such that the declared type of its *i*-th parameter, for all *i*, is identical to the *i*-th declared type in *DTL*.

      ii) If CONSTRUCTOR is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes:

1) An indication that the method specification is CONSTRUCTOR.

2) An indication that the method specification is original.

3) An augmented SQL parameter declaration list *PDL* such that the declared type of its *i*-th parameter, for all *i* > 1 (one), is identical to the (*i*−1)-th declared type in *DTL* and the declared type of the first parameter of *PDL* is identical to *DN*.

iii) Otherwise, the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* for which:

1) If *DOOMS* includes an indication that the method specification is original, then *DOOMS* shall not include an indication that the method specification is either STATIC or CONSTRUCTOR.

2) *DOOMS* includes an augmented SQL parameter declaration list *PDL* such that the declared type of its *i*-th parameter, for all *i* > 1 (one), is identical to the (*i*−1)-th declared type in *DTL* and the declared type of the first parameter of *PDL* is identical to *DN*.

d) If *SMSD* does not immediately contain a <data type list>, then

Case:

i) If STATIC is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes indications that the method specification is both original and STATIC.

ii) If CONSTRUCTOR is specified, then the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* that includes indications that the method specification is both original and CONSTRUCTOR.

iii) Otherwise, the descriptor of *D* shall include exactly one method specification descriptor *DOOMS* for which if *DOOMS* includes an indication that the method specification is original, then *DOOMS* shall not include an indication that the method specification is either STATIC or CONSTRUCTOR.

4) Case:

a) If *DOOMS* includes an indication that the method specification is original, then

Case:

i) If *DOOMS* includes an indication that the method specification specified STATIC, then there shall be no SQL-invoked function *F* for which all of the following are true:

1) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

2) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

3) The declared type of the first SQL parameter of *F* is *D*.

4) The declared type of the *i*-th element of *PDL*, 1 (one) ≤ *i* ≤ *N*, is compatible with the declared type of SQL parameter P_i of *F*.

5) *F* is an SQL-invoked method.

6) *F* includes an indication that STATIC is specified.

ii) If *DOOMS* includes an indication that the method specification specified CONSTRUCTOR, then there shall be no SQL-invoked function *F* for which all of the following are true:

1) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

2) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

3) The declared type of the first SQL parameter of *F* is *D*.

4) The declared type of the *i*-th element of *PDL*, $2 \leq i \leq N$, is compatible with the declared type of SQL parameter P_i of *F*.

5) *F* is an SQL-invoked method.

6) *F* includes an indication that CONSTRUCTOR is specified.

iii) Otherwise:

1) There shall be no proper subtype *PSBD* of *D* whose descriptor includes the descriptor *DOVMS* of an overriding method specification such that all of the following are true:

A) *MN* and the <method name> included in *DOVMS* have equivalent <qualified identifier>s.

B) If *N* is the number of elements in *PDL*, then the augmented SQL parameter declaration list *APDL* included in *DOVMS* has *N* SQL parameters.

C) *PSBD* is the declared type of the first SQL parameter of *APDL*.

D) The declared type of the *i*-th element of *PDL*, $2 \leq i \leq N$, is compatible with the declared type of SQL parameter P_i of *APDL*.

2) There shall be no SQL-invoked function *F* for which all of the following are true:

A) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

B) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

C) The declared type of the first SQL parameter of *F* is *D*.

D) The declared type of the *i*-th element of *PDL*, $2 \leq i \leq N$, is compatible with the declared type of SQL parameter P_i of *F*.

E) *F* is an SQL-invoked method.

F) *F* does not include an indication that either STATIC or CONSTRUCTOR is specified.

b) Otherwise, there shall be no SQL-invoked function *F* for which all of the following are true:

i) The <routine name> of *F* and *MN* have equivalent <qualified identifier>s.

ii) If *N* is the number of elements in *PDL*, then *F* has *N* SQL parameters.

iii) The declared type of the first SQL parameter of *F* is *D*.

iv) The declared type of the *i*-th element of *PDL*, $2 \leq i \leq N$, is compatible with the declared type of SQL parameter P_i of *F*.

v) *F* is an SQL-invoked method.

vi) *F* does not include an indication that either STATIC or CONSTRUCTOR is specified.

Access Rules

None.

General Rules

1) Let *STDS* be the descriptor of *D*.

2) *DOOMS* is removed from *STDS*.

3) *DOOMS* is destroyed.

Conformance Rules

None.

11.59 <drop data type statement>

This Subclause is modified by Subclause 10.23, "<drop data type statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.7, "<drop data type statement>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.13, "<drop data type statement>", in ISO/IEC 9075-16.

Function

Destroy a user-defined type.

Format

```
<drop data type statement> ::=
  DROP TYPE <schema-resolved user-defined type name> <drop behavior>
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> and let *D* be the data type identified by *DN*.

2) Let *RD* be the reference type whose referenced type is *D*. Let *SRD* be the set of all supertypes of *RD*. Let *AD* be the set of all collection types whose element type is *D*. Let *SAD* be set of all collection types whose element type is a supertype of *D* or *RD*.

3) The schema identified by the explicit or implicit <schema name> of *DN* shall include the descriptor of *D*.

4) If RESTRICT is specified, then:

   a) No column, field, or attribute shall have a declared type in either *SRD* or *SAD*, unless its descriptor is included in the descriptor of *D*.

   b) The declared type of no column, attribute, or field shall be based on *D*.

   c) *D* shall have no proper subtypes.

   d) *D* shall not be the structured type of a referenceable table.

   e) The transform descriptor included in the user-defined type descriptor of *D* shall include an empty list of transform groups.

   f) None of the following shall reference *D*, *RD*, or any collection type in *AD*:

      i) The original <query expression> of any view descriptor.

      ii) The <search condition> of any constraint descriptor.

      iii) A triggered action of any trigger descriptor.

      iv) A user-defined cast descriptor.

      v) 04 16 A user-defined type descriptor other than that of *D* itself.

   g) There shall be no SQL-invoked routine that is not dependent on *D* and whose routine descriptor includes the descriptor of *D*, *RD*, or any collection type in *AD*, or whose SQL routine body references *D*, *RD*, or any collection type in *AD*.

   h) For every SQL-invoked routine *R* dependent on *D* and whose routine descriptor includes the descriptor of *D* or *RD*:

i) *R* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

1) The SQL routine body of any routine descriptor.

2) The original <query expression> of any view descriptor.

3) The <search condition> of any constraint descriptor.

4) 04 16 The triggered action of any trigger descriptor.

ii) The specific name of *R* shall not be included in any user-defined cast descriptor.

iii) *R* shall not be the ordering function included in the descriptor of any user-defined type.

NOTE 624 — If CASCADE is specified, then such referenced objects and such dependent objects will be dropped by the execution of the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

NOTE 625 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.35, "SQL-invoked routines".

NOTE 626 — The notion of one data type being based on another data type is defined in Subclause 4.2, "Data types".

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *D*.

General Rules

1) For every SQL-invoked routine *R* that references *D*, *RD*, or any collection type in *AD* or whose routine descriptor includes the descriptor of *D*, *RD*, or any collection type in *AD* and that is not dependent on *D*:

a) Let *SN* be the <specific name> of *R*.

b) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

NOTE 627 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.35, "SQL-invoked routines".

2) The following <drop transform statement> is effectively executed without further Access Rule checking:

```
DROP TRANSFORM ALL FOR DN CASCADE
```

NOTE 628 — This Rule normally has no effect, since any external routine that depends on the transform being dropped also depends on the data type for which the transform is defined and hence will have already been dropped because of GR 1).

3) Let *UDCD* be the user-defined cast descriptor that references *DN* as the source data type. Let *TDN* be the name of the target data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( DN AS TDN ) CASCADE
```

4) Let *UDCD* be the user-defined cast descriptor that references *DN* as the target data type. Let *SDN* be the name of the source data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( SDN AS DN ) CASCADE
```

5) Let *UDCD* be the user-defined cast descriptor that references the reference type whose referenced type is *DN* as the source data type. Let *TDN* be the name of the target data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( REF (DN) AS TDN ) CASCADE
```

6) Let *UDCD* be the user-defined cast descriptor that references the reference type whose referenced type is *DN* as the target data type. Let *SDN* be the name of the source data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( SDN AS REF (DN) ) CASCADE
```

7) For every privilege descriptor that references *D*, the following <revoke statement> is effectively executed:

```
REVOKE PRIV ON TYPE DN FROM GRANTEE CASCADE
```

where *PRIV* and *GRANTEE* are respectively the action and grantee in the privilege descriptor.

8) The descriptor of every SQL-invoked routine that is said to be dependent on *D* is destroyed.

NOTE 629 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.35, "SQL-invoked routines".

9) The descriptor of *D* is destroyed.

Conformance Rules

1) [13] Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop data type statement> that contains a <drop behavior> that contains CASCADE.

11.60 <SQL-invoked routine>

This Subclause is modified by Subclause 10.24, "<SQL-invoked routine>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.11, "<SQL-invoked routine>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.8, "<SQL-invoked routine>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.8, "<SQL-invoked routine>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 11.4, "<SQL-invoked routine>", in ISO/IEC 9075-15.

Function

Define an SQL-invoked routine.

Format

```
04 <SQL-invoked routine> ::=
  <schema routine>

<schema routine> ::=
    <schema procedure>
  | <schema function>

<schema procedure> ::=
  CREATE <SQL-invoked procedure>

<schema function> ::=
  CREATE <SQL-invoked function>

<SQL-invoked procedure> ::=
  PROCEDURE <schema qualified routine name> <SQL parameter declaration list>
      <routine characteristics>
      <routine body>

<SQL-invoked function> ::=
  { <function specification> | <method specification designator> } <routine body>

<SQL parameter declaration list> ::=
  <left paren> [ <SQL parameter declaration>
      [ { <comma> <SQL parameter declaration> }... ] ] <right paren>

14 <SQL parameter declaration> ::=
  [ <parameter mode> ]
      [ <SQL parameter name> ]
      <parameter type> [ RESULT ]
      [ DEFAULT <parameter default> ]

<parameter default> ::=
    <value expression>
  | <contextually typed value specification>
  | <descriptor value constructor>

<parameter mode> ::=
    IN
  | OUT
  | INOUT

14 <parameter type> ::=
    <data type> [ <locator indication> ]
  | <generic table parameter type>
  | <descriptor parameter type>
```

```
<generic table parameter type> ::=
  TABLE [ <pass through option> ] [ <generic table semantics> ]

<pass through option> ::=
    PASS THROUGH
  | NO PASS THROUGH

<generic table semantics> ::=
    WITH ROW SEMANTICS
  | WITH SET SEMANTICS [ <generic table pruning> ]

<generic table pruning> ::=
    PRUNE ON EMPTY
  | KEEP ON EMPTY

<descriptor parameter type> ::=
  DESCRIPTOR

<locator indication> ::=
  AS LOCATOR

<function specification> ::=
  FUNCTION <schema qualified routine name> <SQL parameter declaration list>
      <returns clause>
      <routine characteristics>
      [ <dispatch clause> ]

<method specification designator> ::=
    SPECIFIC METHOD <specific method name>
  | [ INSTANCE | STATIC | CONSTRUCTOR ]
       METHOD <method name> <SQL parameter declaration list>
       [ <returns clause> ]
       FOR <schema-resolved user-defined type name>

<routine characteristics> ::=
  [ <routine characteristic>... ]

<routine characteristic> ::=
    <language clause>
  | <parameter style clause>
  | SPECIFIC <specific name>
  | <deterministic characteristic>
  | <SQL-data access indication>
  | <null-call clause>
  | <returned result sets characteristic>
  | <savepoint level indication>

<savepoint level indication> ::=
    NEW SAVEPOINT LEVEL
  | OLD SAVEPOINT LEVEL

<returned result sets characteristic> ::=
  DYNAMIC RESULT SETS <maximum returned result sets>

<parameter style clause> ::=
  PARAMETER STYLE <parameter style>

<dispatch clause> ::=
  STATIC DISPATCH

14 <returns clause> ::=
  RETURNS <returns type>

<returns type> ::=
```

```
    <returns data type> [ <result cast> ]
  | <returns table type>

<returns table type> ::=
    TABLE [ <table function column list> ]
  | ONLY PASS THROUGH

<table function column list> ::=
  <left paren> <table function column list element>
      [ { <comma> <table function column list element> }... ] <right paren>

<table function column list element> ::=
  <column name> <data type>

<result cast> ::=
  CAST FROM <result cast from type>

<result cast from type> ::=
  <data type> [ <locator indication> ]

14 <returns data type> ::=
  <data type> [ <locator indication> ]

<routine body> ::=
    <SQL routine spec>
  | <external body reference>
  | <polymorphic table function body>

<SQL routine spec> ::=
  [ <rights clause> ] <SQL routine body>

<rights clause> ::=
    SQL SECURITY INVOKER
  | SQL SECURITY DEFINER

<SQL routine body> ::=
  <SQL procedure statement>

<external body reference> ::=
  EXTERNAL [ NAME <external routine name> ]
      [ <parameter style clause> ]
      [ <transform group specification> ]
      [ <external security clause> ]

<polymorphic table function body> ::=
  [ <PTF private parameters> ]
      [ DESCRIBE WITH <PTF describe component procedure> ]
      [ START WITH <PTF start component procedure> ]
      FULFILL WITH <PTF fulfill component procedure>
      [ FINISH WITH <PTF finish component procedure> ]

<PTF private parameters> ::=
  PRIVATE [ DATA ] <private parameter declaration list>

<private parameter declaration list> ::=
  <left paren> [ <SQL parameter declaration>
      [ { <comma> <SQL parameter declaration> }... ] ]
      <right paren>

<PTF describe component procedure> ::=
  <specific routine designator>

<PTF start component procedure> ::=
  <specific routine designator>
```

```
<PTF fulfill component procedure> ::=
  <specific routine designator>

<PTF finish component procedure> ::=
  <specific routine designator>

<external security clause> ::=
    EXTERNAL SECURITY DEFINER
  | EXTERNAL SECURITY INVOKER
  | EXTERNAL SECURITY IMPLEMENTATION DEFINED

13 <parameter style> ::=
    SQL
  | GENERAL

<deterministic characteristic> ::=
    DETERMINISTIC
  | NOT DETERMINISTIC

<SQL-data access indication> ::=
    NO SQL
  | CONTAINS SQL
  | READS SQL DATA
  | MODIFIES SQL DATA

<null-call clause> ::=
    RETURNS NULL ON NULL INPUT
  | CALLED ON NULL INPUT

<maximum returned result sets> ::=
  <unsigned integer>

<transform group specification> ::=
  TRANSFORM GROUP { <single group specification> | <multiple group specification> }

<single group specification> ::=
  <group name>

<multiple group specification> ::=
  <group specification> [ { <comma> <group specification> }... ]

<group specification> ::=
  <group name> FOR TYPE <path-resolved user-defined type name>
```

Syntax Rules

1) 09 An <SQL-invoked routine> specifies an *SQL-invoked routine*. Let *R* be the SQL-invoked routine specified by <SQL-invoked routine>.

2) If <SQL-invoked routine> immediately contains <schema routine>, then the SQL-invoked routine identified by <schema qualified routine name> is a *schema-level routine*.

3) 13 An <SQL-invoked routine> specified as an <SQL-invoked procedure> is called an *SQL-invoked procedure*; an <SQL-invoked routine> specified as an <SQL-invoked function> is called an *SQL-invoked function*. An <SQL-invoked function> that specifies a <method specification designator> is further called an *SQL-invoked method*. An SQL-invoked method that specifies STATIC is called a *static SQL-invoked method*. An SQL-invoked method that specifies CONSTRUCTOR is called an *SQL-invoked constructor method*.

4) If <returns type> *RST* specifies <returns table type>, then

   Case:

a) If <table function column list> is not specified, or if some <parameter type> is either a <generic table parameter type> or <descriptor parameter type>, then *R* is a *polymorphic table function*.

b) Otherwise, *R* is a *monomorphic table function*.

5) An SQL-invoked function that is not an SQL-invoked method or a polymorphic table function is an *SQL-invoked regular function*.

6) If *R* is a monomorphic table function, then <table function column list> shall be specified. Let *TCL* be the <table function column list> contained in <returns table type>.

a) For every <column name> *CN* contained in *TCL*, *CN* shall not be equivalent to any other <column name> contained in *TCL*.

b) *RST* is equivalent to the <returns type>

```
ROW TCL MULTISET
```

7) If *R* is not a polymorphic table function, then:

a) No <parameter type> shall be <generic table parameter type> or <descriptor parameter type>.

b) No <parameter default> shall be <descriptor value constructor>.

c) <polymorphic table function body> shall not be specified.

8) If <SQL-invoked routine> specifies an SQL-invoked method, then

Case:

a) If a <specific method name> *SMN* is specified, then:

i) Case:

1) If *SMN* does not contain <schema name>, then

Case:

A) If the <SQL-invoked routine> is contained in a <schema definition>, then the <schema name> that is specified or implicit in the <schema definition> is implicit.

B) Otherwise, the <schema name> that is specified or implicit for the <SQL-client module definition> is implicit.

2) Otherwise, if <SQL-invoked routine> is contained in a <schema definition>, then the <schema name> contained in *SMN* shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>.

ii) Let *S* be the schema identified by the implicit or explicit <schema name> of *SMN*.

iii) Let *UDT* be a user-defined type included in *S*. There shall exist a method specification descriptor *DMS* included in the descriptor of *UDT* whose <specific method name> is *SMN*.

iv) Let *MN* be the number of SQL parameters in the unaugmented SQL parameter declaration list in *DMS*. *MN* is the number of SQL parameters in the unaugmented SQL parameter declaration list of *R*.

v) If *DMS* includes <result cast> *RC*, then *RC* is the <result cast> of *R*.

vi) Let *SPN* be the <specific method name> in *DMS*. *SPN* is the <specific name> of *R*.

vii) Let *NPL* be the augmented SQL parameter declaration list of *DMS*. *NPL* is the augmented SQL parameter declaration list of *R*.

viii) Let *RN* be *SN*.<method name>, where *SN* is the <schema name> of the schema that includes the descriptor of *UDT*.

b) Otherwise:

i) Let *UDTN* be the <schema-resolved user-defined type name> immediately contained in <method specification designator>.

ii) [13] Let *UDT* be the user-defined type identified by *UDTN*.

iii) There shall exist a method specification descriptor *DMS* in the descriptor of *UDT* such that the <method name> of *DMS* is equivalent to the <method name> *DMS* indicates STATIC if and only if the <method specification designator> specifies STATIC, *DMS* indicates CONSTRUCTOR if and only if the <method specification designator> specifies CONSTRUCTOR, and the declared type of every SQL parameter in the unaugmented SQL parameter declaration list in *DMS* is compatible with the declared type of the corresponding SQL parameter in the <SQL parameter declaration list> contained in the <method specification designator>. *DMS* identifies the corresponding method specification of the <method specification designator>.

iv) Let *MN* be the number of SQL parameters in the unaugmented SQL parameter declaration list in *DMS*.

v) Let $PCOMS_i$, 1 (one) ≤ *i* ≤ *MN*, be the *i*-th SQL parameter in the unaugmented SQL parameter declaration list of *DMS*. Let $POVMS_i$, 1 (one) ≤ *i* ≤ *MN*, be the *i*-th SQL parameter contained in <method specification designator>.

vi) For *i* varying from 1 (one) to *MN*, the <SQL parameter name>s contained in $PCOMS_i$ and $POVMS_i$ shall be equivalent.

vii) Let $PDMS_i$, 1 (one) ≤ *i* ≤ *MN*, be the declared type of the *i*-th SQL parameter in the unaugmented SQL parameter declaration list in *DMS*. Let PSM_i be the declared type of the *i*-th SQL parameter contained in <method specification designator>.

viii) With *i* ranging from 1 (one) to *MN*, the Syntax Rules of Subclause 9.30, "Data type identity", are applied with $PDMS_i$ as *TYPE1* and PSM_i as *TYPE2*.

ix) Case:

1) If <returns clause> is specified, then let *RT* be the <returns data type> of *R*. Let *RDMS* be the <returns data type> in *DMS*. The Syntax Rules of Subclause 9.30, "Data type identity", are applied with *RT* as *TYPE1* and *RDMS* as *TYPE2*.

2) Otherwise, let *RDMS* be the <returns data type> of *R*.

x) If *DMS* includes <result cast> *RC*, then

Case:

1) If <returns clause> is specified, then <returns clause> shall contain <result cast>. Let *RDCT* be the <data type> specified in *RC*. Let *RCT* be the <data type> specified in the <result cast> contained in <returns clause>. The Syntax Rules of Subclause 9.30, "Data type identity", are applied with *RDCT* as *TYPE1* and *RCT* as *TYPE2*.

2) Otherwise, *RC* is the <result cast> of *R*.

xi) Let *SPN* be the <specific method name> in *DMS*. *SPN* is the <specific name> of *R*.

xii) Let *NPL* be the augmented SQL parameter declaration list of *DMS*.

xiii) Let *RN* be *SN*.<method name>, where *SN* is the <schema name> of the schema that includes the descriptor of *UDT*.

9) If <SQL-invoked routine> specifies an SQL-invoked procedure or an SQL-invoked regular function, then:

a) 13 <routine characteristics> shall contain at most one <language clause>, at most one <parameter style clause>, at most one <specific name>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, at most one <null-call clause>, and at most one <returned result sets characteristic>.

b) <parameter style clause> shall not be specified both in <routine characteristics> and in <external body reference>.

c) The <routine characteristics> of a <function specification> shall not contain a <returned result sets characteristic>.

d) If the SQL-invoked routine is an SQL-invoked procedure and <returned result sets characteristic> is not specified, then DYNAMIC RESULT SETS 0 (zero) is implicit.

e) If <deterministic characteristic> is not specified, then NOT DETERMINISTIC is implicit.

f) Case:

i) If PROCEDURE is specified, then:

1) <null-call clause> shall not be specified.

2) <routine characteristics> shall not contain more than one <savepoint level indication>.

ii) Otherwise, if <null-call clause> is not specified, then CALLED ON NULL INPUT is implicit.

g) <SQL-data access indication> shall be specified.

h) 04 If <language clause> is not specified, then LANGUAGE SQL is implicit.

i) 13 An <SQL-invoked routine> that specifies or implies LANGUAGE SQL is called an *SQL routine*; an <SQL-invoked routine> that does not specify LANGUAGE SQL is called an *external routine*.

j) If <savepoint level indication> is specified, then PROCEDURE shall be specified.

k) If PROCEDURE is specified and <savepoint level indication> is not specified, then OLD SAVEPOINT LEVEL is implicit.

l) If NEW SAVEPOINT LEVEL is specified, then MODIFIES SQL DATA shall be specified.

m) If *R* is an SQL routine, then:

i) The <returns clause> shall not specify a <result cast>.

ii) <SQL-data access indication> shall not specify NO SQL.

iii) <parameter style clause> shall not be specified.

n) An *array-returning external function* is an SQL-invoked function that is an external routine for which all of the following are true:

i) A <result cast from type> is specified that does not contain a <locator indication> but simply contains one of the following:

1) An <array type>.

2) A <path-resolved user-defined type name> whose source type is an array type.

ii) A <result cast from type> is not specified and <returns data type> does not contain a <locator indication> but simply contains one of the following:

1) An <array type>.

2) A <path-resolved user-defined type name> whose source type is an array type.

o) A *multiset-returning external function* is an SQL-invoked function that is an external routine for which at least one of the following is true:

i) A <result cast from type> is specified that does not contain a <locator indication> but simply contains one of the following:

1) A <multiset type>.

2) A <path-resolved user-defined type name> whose source type is a multiset type.

ii) A <result cast from type> is not specified and <returns data type> does not contain a <locator indication> but simply contains one of the following:

1) A <multiset type>.

2) A <path-resolved user-defined type name> whose source type is a multiset type.

p) 15 Let *RN* be the <schema qualified routine name> of *R*.

q) 04 If <SQL-invoked routine> is contained in a <schema definition> and *RN* contains a <schema name> *SN*, then *SN* shall be equivalent to the specified or implicit <schema name> of the containing <schema definition>. Let *S* be the SQL-schema identified by *SN*.

r) Case:

i) If *R* is an SQL-invoked regular function and the <SQL parameter declaration list> contains an <SQL parameter declaration> that specifies a <data type> that is one of:

1) A user-defined type.

2) A collection type whose element type is a user-defined type.

3) A collection type whose element type is a reference type.

4) A reference type.

then <dispatch clause> shall be specified.

ii) Otherwise, <dispatch clause> shall not be specified.

s) If <specific name> is not specified, then an implementation-dependent (UV102) <specific name> whose <schema name> is the equivalent to the <schema name> of *S* is implicit.

t) If <specific name> contains a <schema name>, then that <schema name> shall be equivalent to the <schema name> of *S*. If <specific name> does not contain a <schema name>, then the <schema name> of *S* is implicit.

u) The schema identified by the explicit or implicit <schema name> of the <specific name> shall not include a routine descriptor whose specific name is equivalent to <specific name> or a

user-defined type descriptor that includes a method specification descriptor whose specific name is equivalent to <specific name>.

v) If <returns data type> *RT* simply contains <locator indication>, then:

i) *R* shall be an external routine.

ii) *RT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

iii) <result cast> shall not be specified.

w) If <result cast from type> *RCT* simply contains <locator indication>, then:

i) *R* shall be an external routine.

ii) *RCT* shall be either binary large object type, character large object type, array type, multiset type, or user-defined type.

x) If *R* is an external routine, then:

i) If <parameter style> is not specified, then PARAMETER STYLE SQL is implicit.

ii) [13]If *R* is a collection-returning external function, then PARAMETER STYLE SQL shall be either specified or implied.

iii) Case:

1) If <transform group specification> is not specified, then a <multiple group specification> with a <group specification> *GS* for each <SQL parameter declaration> contained in <SQL parameter declaration list> whose <parameter type> *UDT1* identifies a user-defined type with no <locator indication> is implicit. The <group name> of *GS* is implementation-defined (ID155) and its <path-resolved user-defined type name> is *UDT1*.

2) If <single group specification> with a <group name> *GN* is specified, then <transform group specification> is equivalent to a <transform group specification> that contains a <multiple group specification> that contains a <group specification> *GS* for each <SQL parameter declaration> contained in <SQL parameter declaration list> whose <parameter type> *UDT1* identifies a user-defined type with no <locator indication>. The <group name> of *GS* is *GN* and its <path-resolved user-defined type name> is *UDT1*.

3) Otherwise, <multiple group specification> is extended with a <group specification> *GS* for each <SQL parameter declaration> contained in <SQL parameter declaration list> whose <parameter type> *UDT1* identifies a user-defined type with no <locator indication> and no equivalent of *UDT1* is contained in any <group specification> contained in <multiple group specification>. The <group name> of *GS* is implementation-defined (ID155) and its <path-resolved user-defined type name> is *UDT1*.

iv) If a <result cast> is specified, then let *V* be some value of the <data type> specified in the <result cast> and let *RT* be the <returns data type>. The following shall be valid according to the Syntax Rules of Subclause 6.13, "<cast specification>":

```
CAST ( V AS RT )
```

y) [14]Let *NPL* be the <SQL parameter declaration list> contained in the <SQL-invoked routine>.

10) If *R* is a polymorphic table function, then:

a) <method specification designator>, <language clause>, <parameter style clause>, <null-call clause>, <returned result sets characteristic>, <savepoint level indication>, and <dispatch clause> shall not be specified.

b) If <SQL-data access indication> is not specified, then

Case:

i) If *R* has a parameter whose declared type is <generic table parameter type>, then READS SQL DATA is implicit.

ii) Otherwise, CONTAINS SQL is implicit.

c) Let *PTFSDAI* be the explicit or implicit <SQL-data access indication>. *PTFSDAI* shall not be MODIFIES SQL DATA.

d) <polymorphic table function body> shall be specified.

e) If <PTF private parameters> *PTFPRIV* is specified, then for each <SQL parameter declaration> *PTFPRIVPD* contained in *PTFPRIV*:

i) *PTFPRIVPD* shall not specify <parameter mode>, <locator indication>, RESULT, or a <parameter type> that is ROW, <generic table parameter type>, or <descriptor parameter type>.

ii) If *PTFPRIVPD* contains a <parameter default> *PDEF*, then let *PV* be a variable whose declared type is specified by the <data type> contained in *PTFPRIVPD*. Syntax Rules of Subclause 9.2, "Store assignment", are applied with *PV* as *TARGET* and *PDEF* as *VALUE*.

iii) The SQL parameter declared by *PTFPRIVPD* is a *private parameter* of *R*.

f) Let *PTFPUB* be the <SQL parameter declaration list> simply contained in *R*.

g) For every <SQL parameter declaration> contained in *PTFPUB*:

i) <locator indication> shall not be specified.

ii) If <parameter type> is <generic table parameter type>, then:

1) <parameter default> shall not be specified.

2) If <pass through option> is not specified, then NO PASS THROUGH is implicit.

3) If <generic table semantics> is not specified, then WITH SET SEMANTICS is implicit.

4) If WITH SET SEMANTICS is specified or implicit, and <generic table pruning> is not specified, then KEEP ON EMPTY is implicit.

iii) If <parameter type> is <descriptor parameter type> and <parameter default> is specified, then <parameter default> shall be a <descriptor value constructor>; otherwise, <descriptor value constructor> shall not be specified.

h) In the following rules, a parameter list *ACTUALPL* is said to *conform* to a parameter list *TEMPLATEPL* if they have the same number of parameters, and, for each parameter APL_i of *ACTUALPL*, the <parameter type> of APL_i is the same as the <parameter type>, disregarding the presence or absence of <locator indication>, of the parameter TPL_i at the same ordinal position in *TEMPLATEPL*.

i) If <PTF describe component procedure> is specified, then the Syntax Rules of Subclause 9.26, "Signatures of PTF component procedures", are applied with "describe" as *COMPONENT*, *PTFPUB* as *PTF PARAMETER LIST*, and *PTFPRIV* as *PRIVATE PARA-*

METER LIST; let *DESCPL* be the *COMPONENT PARAMETER LIST* returned from the application of those Syntax Rules. <PTF describe component procedure> shall identify an SQL-invoked procedure *DESCPROC* whose parameter list conforms to *DESCPL*.

ii) If <PTF start component procedure> is specified, then the Syntax Rules of Subclause 9.26, "Signatures of PTF component procedures", are applied with "start" as *COMPONENT*, *PTFPUB* as *PTF PARAMETER LIST*, and *PTFPRIV* as *PRIVATE PARAMETER LIST*; let *STARTPL* be the *COMPONENT PARAMETER LIST* returned from the application of those Syntax Rules. <PTF start component procedure> shall identify an SQL-invoked procedure *STARTPROC* whose parameter list conforms to *STARTPL*.

iii) The Syntax Rules of Subclause 9.26, "Signatures of PTF component procedures", are applied with "fulfill" as *COMPONENT*, *PTFPUB* as *PTF PARAMETER LIST*, and *PTFPRIV* as *PRIVATE PARAMETER LIST*; let *FULPL* be the *COMPONENT PARAMETER LIST* returned from the application of those Syntax Rules. <PTF fulfill component procedure> shall identify an SQL-invoked procedure *FULPROC* whose parameter list conforms to *FULPL*.

iv) If <PTF finish component procedure> is specified, then the Syntax Rules of Subclause 9.26, "Signatures of PTF component procedures", are applied with "finish" as *COMPONENT*, *PTFPUB* as *PTF PARAMETER LIST*, and *PTFPRIV* as *PRIVATE PARAMETER LIST*; let *FINPL* be the *COMPONENT PARAMETER LIST* returned from the application of those Syntax Rules. <PTF finish component procedure> shall identify an SQL-invoked procedure *FINPROC* whose parameter list conforms to *FINPL*.

i) If specified, *DESCPROC* is the *PTF describe component procedure* of *R*. If specified, *STARTPROC* is the *PTF start component procedure* of *R*. *FULPROC* is the *PTF fulfill component procedure* of *R*. If specified, *FINPROC* is the *PTF finish component procedure* of *R*. Collectively, these are the *PTF component procedures* of *R*.

j) For each PTF component procedure *COMPPROC* of *R*:

i) Let *COMPSDAI* be the <SQL-data access indication> of *COMPPROC*. *COMPSDAI* shall not specify MODIFIES SQL DATA.

ii) If *PTFSDAI* is CONTAINS SQL, then *COMPSDAI* shall not be READS SQL DATA.

iii) If *PTFSDAI* is NO SQL, then *COMPSDAI* shall be NO SQL.

iv) *COMPPROC* shall not specify <returned result sets characteristic> or <savepoint level indication>.

k) If *DESCPROC* is specified, then *DESCPROC* shall be deterministic.

l) If *R* is deterministic, then every PTF component procedure of *R* shall be deterministic.

m) If the <returns table type> does not contain <table function column list> or ONLY PASS THROUGH, then *DESCPROC* shall be specified.

n) Let *NPL* be the <SQL parameter declaration list> contained in the <function specification>.

11) *NPL* specifies the list of SQL parameters of *R*. Each SQL parameter of *R* is specified by an <SQL parameter declaration>. If <SQL parameter name> is specified, then that SQL parameter of *R* is identified by an SQL parameter name.

12) *NPL* shall specify at most one <SQL parameter declaration> that specifies RESULT.

13) If *R* is an SQL-invoked function, then no <SQL parameter declaration> in *NPL* shall contain a <parameter mode>.

14) If *R* is an SQL routine, then every <SQL parameter declaration> in *NPL* shall contain an <SQL parameter name>.

15) No two <SQL parameter name>s contained in *NPL* shall be equivalent.

16) Let *N* and *PN* be the number of <SQL parameter declaration>s contained in *NPL*. For every <SQL parameter declaration> PD_i, 1 (one) ≤ i ≤ N:

    a) <parameter type> PT_i immediately contained in PD_i shall not specify ROW.

    b) If PT_i simply contains <locator indication>, then:

        i) *R* shall be an external routine.

        ii) PT_i shall specify either binary large object type, character large object type, array type, multiset type, or user-defined type.

    c) If PD_i immediately contains RESULT, then:

        i) *R* shall be an SQL-invoked function.

        ii) PT_i shall specify a structured type *ST*. Let *STN* be the <user-defined type name> that identifies *ST*.

        iii) The <returns data type> shall specify *STN*.

        iv) *R* is a type-preserving function and PD_i specifies the result SQL parameter of *R*.

    d) If PD_i does not contain a <parameter mode>, then a <parameter mode> that specifies IN is implicit.

    e) Let P_i be the *i*-th SQL parameter.

        Case:

        i) If the <parameter mode> specifies IN, then P_i is an input SQL parameter. If a <parameter default> $PDEF_i$ is specified, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with P_i as *TARGET* and $PDEF_i$ as *VALUE*.

        ii) If the <parameter mode> specifies OUT, then P_i is an output SQL parameter. A <parameter default> shall not be specified.

        iii) If the <parameter mode> specifies INOUT, then P_i is both an input SQL parameter and an output SQL parameter. If a <parameter default> $PDEF_i$ is specified, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with P_i as *TARGET* and $PDEF_i$ as *VALUE*.

17) The scope of *RN* is the <routine body> of *R*.

18) The scope of an <SQL parameter name> contained in *NPL* is the <routine body> *RB* of the <SQL-invoked procedure> or <SQL-invoked function> that contains *NPL*.

19) An <SQL-invoked routine> shall not contain a <host parameter name>, a <dynamic parameter specification>, or an <embedded variable name>.

20) Case:

    a) If *R* is an SQL-invoked procedure, then *S* shall not include another SQL-invoked procedure whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

    b) Otherwise:

i) Case:

1) If *R* is a static SQL-invoked method, then let *SCR* be the set containing every static SQL-invoked method of type *UDT*, including *R*, whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

2) If *R* is an SQL-invoked constructor method, then let *SCR* be the set containing every SQL-invoked constructor method of type *UDT*, including *R*, whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

3) Otherwise, let *SCR* be the set containing every SQL-invoked function in *S* that is neither a static SQL-invoked method nor an SQL-invoked constructor method, including *R*, whose <schema qualified routine name> is equivalent to *RN* and whose number of SQL parameters is *PN*.

ii) Let *AL* be an <SQL argument list> constructed from a list of arbitrarily-selected values in which the declared type of every value A_i in *AL* is compatible with the declared type of the corresponding SQL parameter P_i of *R*.

iii) For every A_i, eliminate from *SCR* every SQL-invoked routine *SIR* for which the type designator of the declared type of the SQL parameter P_i of *SIR* is not in the type precedence list of the declared type of A_i.

iv) The Syntax Rules of Subclause 9.6, "Subject routine determination", are applied with *AL* as *SQL ARGUMENT LIST* and *SCR* as *SET OF SQL-INVOKED ROUTINES*; let *SR* be the *SET OF SUBJECT ROUTINES* returned from the application of those Syntax Rules. There shall be exactly one subject routine in *SR*.

21) If *R* is an SQL-invoked method but not a static SQL-invoked method, then the first SQL parameter of *NPL* is called the *subject parameter* of *R*.

22) If *R* is an SQL-invoked regular function *F* whose first SQL parameter has a declared type that is a user-defined type, then:

a) Let *UDT2* be the declared type of the first SQL parameter of *F*.

b) Let *DMS* be a method specification descriptor of an instance method in the descriptor of *UDT2* such that:

i) The <schema qualified routine name> of *F* and the <routine name> of *DMS* have equivalent <qualified identifier>s.

ii) *F* and the augmented SQL parameter declaration list of *DMS* have the same number of SQL parameters.

c) Let $PDMS_i$, 1 (one) $\le i \le PN-1$, be the declared type of the *i*-th SQL parameter in the unaugmented SQL parameter declaration list in *DMS* and let PMS_i, 1 (one) $\le i \le PN$, be the declared type of the *i*-th SQL parameter contained in <function specification>.

d) Exactly one of the following shall be false:

i) The declared type of $PDMS_i$, 1 (one) $\le i \le PN-1$ is compatible with the declared type of SQL parameter PMS_{i+1}.

ii) *UDT2* is a subtype or a supertype of the declared type of PMS_1.

23) If *R* is an SQL routine, then:

a) <SQL routine spec> shall be specified.

b) If <rights clause> is not specified, then SQL SECURITY DEFINER is implicit.

c) If READS SQL DATA is specified, then it is implementation-defined (IA075) whether the <SQL routine body> shall not contain an SQL-statement that possibly modifies SQL-data.

d) If CONTAINS SQL is specified, then it is implementation-defined (IA076) whether the <SQL routine body> shall not contain an SQL-statement that possibly reads SQL-data or possibly modifies SQL-data.

e) If DETERMINISTIC is specified, then it is implementation-defined (IA077) whether the <SQL routine body> shall not contain a potential source of non-determinism.

f) It is implementation-defined (IA078) whether the <SQL routine body> shall not contain an <SQL connection statement>, an <SQL schema statement>, an <SQL dynamic statement>, or an <SQL transaction statement> other than a <savepoint statement>, <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>.

NOTE 630 — Conforming SQL language must not contain an <SQL connection statement> or an <SQL transaction statement> other than a <savepoint statement>, a <release savepoint statement>, or a <rollback statement> that specifies a <savepoint clause>, but an SQL-implementation is not required to treat this as a syntax error.

g) An <SQL routine body> shall not immediately contain an <SQL procedure statement> that simply contains a <schema definition>.

h) 04 An <SQL routine body> shall not immediately contain an <SQL procedure statement> that simply contains a <table reference> that identifies an SQL-client module declared local temporary table.

24) If *R* is an external routine, then:

a) <SQL routine spec> shall not be specified.

b) If <external security clause> is not specified, then EXTERNAL SECURITY IMPLEMENTATION DEFINED is implicit.

c) If an <external routine name> is not specified, then an <external routine name> that is equivalent to the <qualified identifier> of *R* is implicit.

d) If PARAMETER STYLE SQL is specified, then:

i) Case:

1) If *R* is a collection-returning external function with the element type being a row type, then let *FRN* be the degree of the element type.

2) Otherwise, let *FRN* be 1 (one).

ii) If *R* is a collection-returning external function, then let *AREF* be *FRN*+6. Otherwise, let *AREF* be *FRN*+4.

iii) If *R* is an SQL-invoked function, then let the *effective SQL parameter list* be a list of *PN*+*FRN*+*N*+*AREF* SQL parameters, as follows:

1) For *i* ranging from 1 (one) to *PN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

A) 14 If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> contains <locator indication>, then the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by INTEGER.

B) If the <parameter type> T_i immediately contained in the *i*-th <SQL parameter declaration> is a <path-resolved user-defined type name> without a <locator indication>, then:

I) Case:

1) If *R* is an SQL-invoked method that is an overriding method, then the Syntax Rules of Subclause 9.32, "Determination of a from-sql function for an overriding method", are applied with *R* as *ROUTINE* and *i* as *POSITION*; let FSF_i be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules.

2) Otherwise, the Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let FSF_i be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules.

II) FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

III) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

C) Otherwise, the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration>.

2) Case:

A) If *FRN* is 1 (one), then effective SQL parameter list entry *PN*+*FRN* has <parameter mode> OUT; its <parameter type> *PT* is defined as follows:

I) If <result cast> is specified, then let *RT* be <result cast from type>; otherwise, let *RT* be <returns data type>.

II) Case:

1) [14] If *RT* simply contains <locator indication>, then *PT* is INTEGER.

2) If *RT* specifies a <path-resolved user-defined type name> without a <locator indication>, then:

a) Case:

i) If *R* is an SQL-invoked method that is an overriding method, then the Syntax Rules of Subclause 9.34, "Determination of a to-sql function for an overriding method", are applied with *R* as *ROUTINE*; let *TSF* be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable to-sql function *TSF*.

ii) Otherwise, the Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with the data type identified by *RT* as *TYPE* and the <group name> contained in the <group specific-

ation> that contains *RT* as *GROUP*; let *TSF* be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules.

b) *TSF* is called the *to-sql function*.

c) Case:

i) If *TSF* is an SQL-invoked method, then *PT* is the <parameter type> of the second SQL parameter of *TSF*.

ii) Otherwise, *PT* is the <parameter type> of the first SQL parameter of *TSF*.

3) If *R* is a collection-returning external function, then *PT* is the element type of *RT*.

4) Otherwise, *PT* is *RT*.

B) Otherwise, for *i* ranging from *PN*+1 to *PN*+*FRN*, the *i*-th effective SQL parameter list entry is defined as follows:

I) Its <parameter mode> is OUT.

II) Let $RFT_{i\text{-}PN}$ be the data type of the *i*-*PN*-th field of the element type of the <returns data type>. The <parameter type> PT_i of the *i*-th effective SQL parameter list entry is determined as follows.

Case:

1) If $RFT_{i\text{-}PN}$ specifies a <path-resolved user-defined type name>, then:

a) Case:

i) If *R* is an SQL-invoked method that is an overriding method, then the Syntax Rules of Subclause 9.34, "Determination of a to-sql function for an overriding method", are applied with *R* as *ROUTINE*; let *TSF* be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable to-sql function *TSF*.

ii) Otherwise, the Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with the data type identified by $RFT_{i\text{-}PN}$ as *TYPE* and the <group name> contained in the <group specification> that contains $RFT_{i\text{-}PN}$ as *GROUP*; let *TSF* be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable to-sql function *TSF*.

b) *TSF* is called the *to-sql function* associated with $RFT_{i\text{-}PN}$.

c) Case:

i) If *TSF* is an SQL-invoked method, then PT_i is the <parameter type> of the second SQL parameter of *TSF*.

ii) Otherwise, PT_i is the <parameter type> of the first SQL parameter of *TSF*.

2) [14]Otherwise, PT_i is $RFT_{i\text{-}PN}$.

3) Effective SQL parameter list entries (*PN*+*FRN*)+1 to (*PN*+*FRN*)+*N*+*FRN* are *N*+*FRN* occurrences of SQL parameters of an implementation-defined (IV199) <data type> that is an exact numeric type with scale 0 (zero). For *i* ranging from (*PN*+*FRN*)+1 to (*PN*+*FRN*)+*N*+*FRN*, the <parameter mode> for the *i*-th such effective SQL parameter is the same as that of the *i*−*FRN*−*PN*-th effective SQL parameter.

4) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+1 is an SQL parameter of a <data type> that is character string of length 5 and the character set specified for SQLSTATE values, with <parameter mode> INOUT.

NOTE 631 — The character set specified for SQLSTATE values is defined in Subclause 24.1, "SQLSTATE".

5) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+2 is an SQL parameter of a <data type> that is character string of implementation-defined (IV049) length and character set SQL_TEXT with <parameter mode> IN.

6) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+3 is an SQL parameter of a <data type> that is character string of implementation-defined (IV049) length and character set SQL_TEXT with <parameter mode> IN.

7) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+4 is an SQL parameter of a <data type> that is character string of implementation-defined (IV049) length and character set SQL_TEXT with <parameter mode> INOUT.

8) If *R* is a collection-returning external function, then:

A) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+5 is an SQL parameter whose <data type> is character string of implementation-defined (IV198) length and character set SQL_TEXT with <parameter mode> INOUT.

B) Effective SQL parameter list entry (*PN*+*FRN*)+(*N*+*FRN*)+6 is an SQL parameter whose <data type> is an exact numeric type with scale 0 (zero) and with <parameter mode> IN.

iv) If *R* is an SQL-invoked procedure, then let the *effective SQL parameter list* be a list of *PN*+*N*+4 SQL parameters, as follows:

1) For *i* ranging from 1 (one) to *PN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

A) [14]If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> simply contains <locator indication>, then the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by INTEGER.

B) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> is a <path-resolved user-defined type name> without a <locator indication>, then:

I) Case:

1) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is IN, then:

   a) The Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let FSF_i be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules. FSF_i is called the *from-sql function* associated with the *i*-th SQL parameter.

   b) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

2) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is OUT, then:

   a) The Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let TSF_i be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. TSF_i is called the *to-sql function* associated with the *i*-th SQL parameter.

   b) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by

      Case:

      i) If TSF_i is an SQL-invoked method, then the <parameter type> of the second SQL parameter of TSF_i.

      ii) Otherwise, the <parameter type> of the first SQL parameter of TSF_i.

3) Otherwise:

   a) The Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let FSF_i be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules. FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

   b) The Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let TSF_i be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. TSF_i is called the *to-sql function* associated with the *i*-th SQL parameter.

c) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

C) Otherwise, the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration>.

2) Effective SQL parameter list entries *PN*+1 to *PN*+*N* are *N* occurrences of an SQL parameter of an implementation-defined (IV199) <data type> that is an exact numeric type with scale 0. The <parameter mode> for the *i*-th such effective SQL parameter is the same as that of the *i*–*PN*-th effective SQL parameter.

3) Effective SQL parameter list entry (*PN*+*N*)+1 is an SQL parameter of a <data type> that is character string of length 5 and character set SQL_TEXT with <parameter mode> INOUT.

4) Effective SQL parameter list entry (*PN*+*N*)+2 is an SQL parameter of a <data type> that is character string of implementation-defined (IV049) length and character set SQL_TEXT with <parameter mode> IN.

5) Effective SQL parameter list entry (*PN*+*N*)+3 is an SQL parameter of a <data type> that is character string of implementation-defined (IV049) length and character set SQL_TEXT with <parameter mode> IN.

6) Effective SQL parameter list entry (*PN*+*N*)+4 is an SQL parameter of a <data type> that is character string of implementation-defined (IV049) length and character set SQL_TEXT with <parameter mode> INOUT.

e) 13 If PARAMETER STYLE GENERAL is specified, then let the *effective SQL parameter list* be a list of *PN* parameters such that, for *i* ranging from 1 (one) to *PN*, the *i*-th effective SQL parameter list entry is defined as follows.

Case:

i) 14 If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> simply contains <locator indication>, then the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by INTEGER.

ii) If the <parameter type> T_i simply contained in the *i*-th <SQL parameter declaration> is a <path-resolved user-defined type name> without a <locator indication>, then:

1) Case:

A) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is IN, then:

I) The Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let FSF_i be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules. FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

II) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

B) If the <parameter mode> immediately contained in the *i*-th <SQL parameter declaration> is OUT, then:

I) The Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let TSF_i be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. TSF_i is called the *to-sql function* associated with the *i*-th SQL parameter.

II) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by

Case:

1) If TSF_i is an SQL-invoked method, then the <parameter type> of the second SQL parameter of TSF_i.

2) Otherwise, the <parameter type> of the first SQL parameter of TSF_i.

C) Otherwise:

I) The Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let FSF_i be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules. FSF_i is called the *from-sql function associated with the i-th SQL parameter*.

II) The Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with the data type identified by T_i as *TYPE* and the <group name> contained in the <group specification> that contains T_i as *GROUP*; let TSF_i be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. TSF_i is called the *to-sql function* associated with the *i*-th SQL parameter.

III) The *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration> with the <parameter type> replaced by the <returns data type> of FSF_i.

iii) Otherwise, the *i*-th effective SQL parameter list entry is the *i*-th <SQL parameter declaration>.

NOTE 632 — If the SQL-invoked routine is an SQL-invoked function, then the value returned from the external routine is passed to the SQL-implementation in an implementation-dependent manner. An SQL parameter is not used for this purpose.

f) Depending on whether the <language clause> specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the *operative data type correspondences table* be Table 21, "Data type correspondences for Ada", Table 22, "Data type correspondences for C", Table 23, "Data type correspondences for COBOL", Table 24, "Data type correspondences for Fortran", Table 25, "Data type correspondences for M", Table 26, "Data type correspondences for Pascal", or Table 27, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

g) 13Every <data type> in an effective SQL parameter list entry shall specify a data type listed in the SQL data type column for which the corresponding row in the host data type column is not "None".

h) 13If *R* is an SQL-invoked function and PARAMETER STYLE GENERAL is specified, then the <data type> immediately contained in a <returns data type> shall specify a data type listed in the SQL data type column for which the corresponding row in the host data type column is not "None".

25) 13Case:

a) If <method specification designator> is specified, then:

i) *R* is *deterministic* if *DMS* indicates that the method is deterministic; otherwise, *R* is *possibly non-deterministic*.

ii) *R* possibly modifies SQL-data if the SQL-data access indication of *DMS* indicates that the method possibly modifies SQL-data. *R* possibly reads SQL-data if the SQL-data access indication of *DMS* indicates that the method possibly reads SQL-data. *R* possibly contains SQL if the SQL-data access indication of *DMS* indicates that the method possibly contains SQL. Otherwise, *R* does not possibly contain SQL.

b) Otherwise:

i) If DETERMINISTIC is specified, then *R* is *deterministic*; otherwise, it is *possibly non-deterministic*.

ii) An <SQL-invoked routine> *possibly modifies SQL-data* if and only if <SQL-data access indication> specifies MODIFIES SQL DATA.

iii) An <SQL-invoked routine> *possibly reads SQL-data* if and only if <SQL-data access indication> specifies READS SQL DATA.

iv) An <SQL-invoked routine> *possibly contains SQL* if and only if <SQL-data access indication> specifies CONTAINS SQL.

v) An <SQL-invoked routine> *does not possibly contain SQL* if and only if <SQL-data access indication> specifies NO SQL.

26) If *R* is a schema-level routine, then let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in <schema qualified routine name>.

27) 14If the <SQL-invoked routine> is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>; otherwise, let *A* be the <authorization identifier> that owns the schema identified by the explicit or implicit <schema name> of the <schema qualified routine name>.

Access Rules

1) 13If an <SQL-invoked routine> is contained in an <SQL-client module definition> *M* with no intervening <schema definition>, then the enabled authorization identifiers shall include the <authorization identifier> that owns *S*.

2) If *R* is an external routine and if any of its SQL parameters have an associated from-sql function or a to-sql function, or if *R* has a to-sql function associated with the result, then

Case:

a) If <SQL-invoked routine> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the

<authorization identifier> that owns the containing schema shall include EXECUTE on all from-sql functions (if any) and on all to-sql functions (if any) associated with the SQL parameters and on the to-sql function associated with the result (if any).

b) Otherwise, the current privileges shall include EXECUTE on all from-sql functions (if any) and on all to-sql functions (if any) associated with the SQL parameters and on the to-sql function associated with the result (if any).

3) If *R* is a polymorphic table function, then

Case:

a) If <SQL-invoked routine> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then the applicable privileges of the <authorization identifier> that owns the containing schema shall include EXECUTE on all PTF component procedures of *R*.

b) Otherwise, the current privileges shall include EXECUTE on all PTF component procedures of *R*.

General Rules

1) If *R* is a schema-level routine, then a privilege descriptor is created that defines the EXECUTE privilege on *R* to the <authorization identifier> that owns the schema that includes *R*. The grantor for the privilege descriptor is set to the special grantor value "_SYSTEM". This privilege is grantable if and only if exactly one of the following is true:

a) *R* is an SQL routine and all of the privileges necessary for the <authorization identifier> to successfully execute the <SQL procedure statement> contained in the <routine body> are grantable. The necessary privileges include the EXECUTE privilege on every subject routine of every <routine invocation> contained in the <SQL procedure statement>.

b) *R* is an SQL routine and SQL SECURITY INVOKER is specified.

c) *R* is an external routine.

2) Case:

a) If <SQL-invoked routine> is contained in a <schema definition>, then let *DP* be the SQL-path of that <schema definition>.

b) If <SQL-invoked routine> is contained in a <preparable statement> or in a <direct SQL statement>, then let *DP* be the SQL-path of the current SQL-session.

c) Otherwise, let *DP* be the SQL-path of the <SQL-client module definition> that contains <SQL-invoked routine>.

3) If <method specification designator> is not specified, then a routine descriptor is created that describes the SQL-invoked routine being defined:

a) The routine name included in the routine descriptor is <schema qualified routine name>.

b) The specific name included in the routine descriptor is <specific name>.

c) The routine descriptor includes, for each SQL parameter in *NPL*:

i) The name.

ii) The declared type. If <parameter type> is <generic table parameter type>, then the declared type is *generic table type*. If <parameter type> is <descriptor parameter type>, then the declared type is *descriptor type*.

iii) The ordinal position.

iv) An indication of whether the SQL parameter is input, output, or both.

v) An indication of whether the SQL parameter is a result SQL parameter.

vi) An indication of whether the SQL parameter has a default value and, if so, the <parameter default>.

vii) If the declared type is generic table type, then:

1) An indication of whether PASS THROUGH or NO PASS THROUGH is specified.

2) An indication of whether WITH SET SEMANTICS or WITH ROW SEMANTICS is specified.

3) If WITH SET SEMANTICS is specified, then an indication of whether KEEP WHEN EMPTY or PRUNE WHEN EMPTY is specified.

d) If the SQL-invoked routine is a polymorphic table function, then the routine descriptor includes, for each private parameter:

i) The name.

ii) The declared type.

iii) The ordinal position.

iv) An indication of whether the private parameter has a default value and, if so, the <parameter default>.

e) The maximum number of returned result sets included in the routine descriptor is

Case:

i) If the SQL-invoked routine is an SQL-invoked procedure, then the explicit or implicit value of <maximum returned result sets>.

ii) Otherwise, 0 (zero).

f) The routine descriptor includes an indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.

g) If the SQL-invoked routine is an SQL-invoked function, then:

i) The routine descriptor includes an indication that the SQL-invoked function is not an SQL-invoked method.

ii) Case:

1) If the <returns clause> contains a <returns table type>, then the routine descriptor includes the <returns table type>.

2) Otherwise, the routine descriptor includes the data type in the <returns data type>. If the <returns data type> simply contains <locator indication>, then the routine descriptor includes an indication that the return value is a locator.

iii) The SQL-invoked routine descriptor includes an indication of whether the SQL-invoked routine is a null-call function.

h) If the SQL-invoked routine is a type-preserving function, then the routine descriptor includes an indication that the SQL-invoked routine is a type-preserving function.

i) The name of the language in which the body of the SQL-invoked routine was written is the <language name> contained in the <language clause>.

j) Case:

  i) If the SQL-invoked routine is an SQL routine, then the SQL routine body of the routine descriptor is the <SQL routine body>.

  ii) If the SQL-invoked routine is a polymorphic table function, then the routine descriptor includes the descriptors of the PTF describe component procedure (if specified), the PTF start component procedure (if specified), the PTF fulfill component procedure and the PTF finish component procedure (if specified).

k) If the SQL-invoked routine is an SQL-invoked function or NEW SAVEPOINT LEVEL is specified, then the routine descriptor includes an indication that a new savepoint level is to be established whenever the routine is invoked.

NOTE 633 — The use of savepoint levels is dependent on Feature T272, "Enhanced savepoint management".

l) Case:

  i) If SQL SECURITY INVOKER is specified, then the SQL security characteristic in the routine descriptor is INVOKER.

  ii) Otherwise, the SQL security characteristic in the routine descriptor is DEFINER.

m) If the SQL-invoked routine is an external routine, then:

  i) The external name of the routine descriptor is <external routine name>.

  ii) [13] The routine descriptor includes an indication of whether the *parameter passing style* is:

    1) PARAMETER STYLE SQL.

    2) PARAMETER STYLE GENERAL.

n) The SQL-invoked routine descriptor includes an indication of whether the SQL-invoked routine is DETERMINISTIC or NOT DETERMINISTIC.

o) The routine descriptor includes an indication of whether the SQL-invoked routine does not possibly contain SQL, possibly contains SQL, possibly reads SQL-data, or possibly modifies SQL-data.

p) If the SQL-invoked routine specifies a <result cast>, then the routine descriptor includes an indication that the SQL-invoked routine specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then the routine descriptor includes an indication that the <data type> specified in the <result cast> has a locator indication.

q) For every SQL parameter that has an associated from-sql function *FSF*, the routine descriptor includes the specific name of *FSF*.

r) [14] For every SQL parameter that has an associated to-sql function *TSF*, the routine descriptor includes the specific name of *TSF*.

s) [14] If *R* is an external function and if *R* has a to-sql function associated with its result *TRF*, then the routine descriptor includes the specific name of *TRF*.

t) For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, the routine descriptor includes an indication that the SQL parameter is a locator parameter.

u) The routine authorization identifier included in the routine descriptor is the <authorization identifier> that owns *S*.

v) The routine SQL-path included in the routine descriptor is *DP*.

NOTE 634 — The routine SQL-path is used to set the routine SQL-path of the current SQL-session when *R* is invoked. The routine SQL-path of the current SQL-session is used by the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", to define the subject routines of <routine invocation>s contained in *R*. The same routine SQL-path is used whenever *R* is invoked.

w) 04 The routine descriptor includes an indication of whether the routine is a schema-level routine.

x) The routine descriptor includes an indication of whether the SQL-invoked routine is dependent on a user-defined type.

NOTE 635 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.35, "SQL-invoked routines".

4) If <method specification designator> is specified, then let *DMS* be the descriptor of the corresponding method specification. A routine descriptor is created that describes the SQL-invoked routine being defined.

a) The routine name included in the routine descriptor is *RN*.

b) The specific name included in the routine descriptor is <specific name>.

c) The routine descriptor includes, for each SQL parameter in *NPL*, the name, declared type, ordinal position, an indication of whether the SQL parameter is input, output, or both, and an indication of whether the SQL parameter is a result SQL parameter.

d) The routine descriptor includes an indication that the SQL-invoked routine is an SQL-invoked function that is an SQL-invoked method, the name of *UDT*, and an indication of whether STATIC or CONSTRUCTOR was specified.

e) If the SQL-invoked routine is a type-preserving function, then the routine descriptor includes an indication that the SQL-invoked routine is a type-preserving function.

f) If the SQL-invoked routine is a mutator function, then the routine descriptor includes an indication that the SQL-invoked routine is a mutator function.

g) The routine descriptor includes the data type in the <returns data type>.

h) The name of the language in which the body of the SQL-invoked routine was written is the <language name> contained in the <language clause> in *DMS*.

i) If the SQL-invoked routine is an SQL routine, then the SQL routine body of the routine descriptor is the <SQL routine body>.

j) Case:

i) If SQL SECURITY INVOKER is specified, then the SQL security characteristic in the routine descriptor is INVOKER.

ii) Otherwise, the SQL security characteristic in the routine descriptor is DEFINER.

k) If the SQL-invoked routine is an external routine, then:

i) The external name of the routine descriptor is <external routine name>.

ii) The routine descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL, which is the same as the indication of <parameter style> in *DMS*.

l) The routine descriptor includes an indication of whether the SQL-invoked routine is deterministic.

m) The routine descriptor includes an indication of whether the SQL-invoked routine possibly modifies SQL-data, possibly read SQL-data, possibly contains SQL, or does not possibly contain SQL.

n) The routine descriptor includes an indication of whether the SQL-invoked routine is a null-call function, which is the same as the indication in *DMS*.

o) If *DMS* specifies a <result cast>, then the routine descriptor includes an indication that the SQL-invoked routine specifies a <result cast> and the <data type> specified in the <result cast> of *DMS*.

p) The routine authorization identifier included in the routine descriptor is the <authorization identifier> that owns *S*.

q) The routine SQL-path included in the routine descriptor is *DP*.

NOTE 636 — The routine SQL-path is used to set the routine SQL-path of the current SQL-session when *R* is invoked. The routine SQL-path of the current SQL-session is used by the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", to define the subject routine of <routine invocation>s contained in *R*. The same routine SQL-path is used whenever *R* is invoked.

r) The routine descriptor includes an indication of whether the routine is a schema-level routine.

s) The routine descriptor includes an indication of whether the SQL-invoked routine is dependent on a user-defined type.

NOTE 637 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.35, "SQL-invoked routines".

5) The creation timestamp and the last-altered timestamp included in the routine descriptor are the values of CURRENT_TIMESTAMP.

6) If *R* is an external routine, then the routine descriptor of *R* includes further elements determined as follows:

a) Case:

i) If <SQL-data access indication> in the descriptor of *R* is MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL, then:

1) Let *P* be the program identified by the <external routine name>.

2) The external routine SQL-path is the <schema name list> immediately contained in the <path specification> that is immediately contained in the <module path specification> of the <SQL-client module definition> of *P*.

ii) Otherwise, the external routine SQL-path is implementation-defined (ID156).

NOTE 638 — The external routine SQL-path is used to set the routine SQL-path of the current SQL-session when *R* is invoked. The routine SQL-path of the current SQL-session is used by the Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", to define the subject routines of <routine invocation>s contained in the <SQL-client module definition> of *P*. The same external routine SQL-path is used whenever *R* is invoked.

b) The external security characteristic in the routine descriptor is

Case:

i) If <external security clause> specifies EXTERNAL SECURITY DEFINER, then DEFINER.

ii) If <external security clause> specifies EXTERNAL SECURITY INVOKER, then INVOKER.

iii) Otherwise, EXTERNAL SECURITY IMPLEMENTATION DEFINED.

c) The effective SQL parameter list is the *effective SQL parameter list*.

Conformance Rules

1) Without Feature T471, "Result sets return value", conforming SQL language shall not contain a <returned result sets characteristic>.

2) Without Feature T341, "Overloading of SQL-invoked functions and SQL-invoked procedures", conforming SQL language shall not contain a <schema routine> in which the schema identified by the explicit or implicit <schema name> of the <schema qualified routine name> includes a routine descriptor whose routine name is <schema qualified routine name>.

3) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <method specification designator>.

4) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <transform group specification>.

5) Without Feature S024, "Enhanced structured types", an <SQL parameter declaration> shall not contain RESULT.

6) Without Feature T571, "Array-returning external SQL-invoked functions", conforming SQL language shall not contain an <SQL-invoked routine> that defines an array-returning external function.

7) Without Feature T572, "Multiset-returning external SQL-invoked functions", conforming SQL language shall not contain an <SQL-invoked routine> that defines a multiset-returning external function.

8) Without Feature S203, "Array parameters", conforming SQL language shall not contain a <parameter type> that is based on an array type.

9) Without Feature S204, "Array as result type of functions", conforming SQL language shall not contain a <returns data type> that is based on an array type.

10) Without Feature S202, "SQL-invoked routines on multisets", conforming SQL language shall not contain a <parameter type> that is based on a multiset type.

11) Without Feature S202, "SQL-invoked routines on multisets", conforming SQL language shall not contain a <returns data type> that is based on a multiset type.

12) Without Feature T323, "Explicit security for external routines", conforming SQL language shall not contain an <external security clause>.

13) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a structured type.

14) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a structured type.

15) Without Feature S232, "Array locators", conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies an array type.

16) Without Feature S232, "Array locators", conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies an array type.

17) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a multiset type.

18) Without Feature S233, “Multiset locators”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a multiset type.

19) Without Feature T049, “BLOB locator: non-holdable”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a binary large object type.

20) Without Feature T049, “BLOB locator: non-holdable”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a binary large object type.

21) Without Feature T039, “CLOB locator: non-holdable”, conforming SQL language shall not contain a <parameter type> that contains a <locator indication> and that simply contains a <data type> that identifies a character large object type.

22) Without Feature T039, “CLOB locator: non-holdable”, conforming SQL language shall not contain a <returns data type> that contains a <locator indication> and that simply contains a <data type> that identifies a character large object type.

23) Without Feature S027, “Create method by specific method name”, conforming SQL language shall not contain a <method specification designator> that contains SPECIFIC METHOD.

24) Without Feature T324, “Explicit security for SQL routines”, conforming SQL language shall not contain a <rights clause>.

25) Without Feature T326, “Table functions”, conforming SQL language shall not contain a <returns table type>.

26) Without Feature T651, “SQL-schema statements in SQL routines”, conforming SQL language shall not contain an <SQL routine body> that contains an SQL-schema statement.

27) Without Feature T652, “SQL-dynamic statements in SQL routines”, conforming SQL language shall not contain an <SQL routine body> that contains an SQL-dynamic statement.

28) Without Feature T653, “SQL-schema statements in external routines”, conforming SQL language shall not contain an <external routine name> that identifies a program in which an SQL-schema statement appears.

29) Without Feature T654, “SQL-dynamic statements in external routines”, conforming SQL language shall not contain an <external routine name> that identifies a program in which an SQL-dynamic statement appears.

30) Without Feature T655, “Cyclically dependent routines”, conforming SQL language shall not contain an <SQL routine body> that contains a <routine invocation> whose subject routine is generally dependent on the routine descriptor of the SQL-invoked routine specified by <SQL-invoked routine>.

31) Without Feature T272, “Enhanced savepoint management”, conforming SQL language shall not contain a <routine characteristics> that contains a <savepoint level indication>.

32) Without Feature T522, “Default values for IN parameters of SQL-invoked procedures”, conforming SQL language shall not contain an <SQL-invoked procedure> that contains an <SQL parameter declaration> that explicitly or implicitly specifies a <parameter mode> of IN and that specifies a <parameter default>.

33) Without Feature T523, “Default values for INOUT parameters of SQL-invoked procedures”, conforming SQL language shall not contain an <SQL-invoked procedure> that contains an <SQL parameter declaration> that explicitly or implicitly specifies a <parameter mode> of INOUT and that specifies a <parameter default>.

34) Without Feature T525, "Default values for parameters of SQL-invoked functions", conforming SQL language shall not contain an <SQL-invoked function> that contains an <SQL parameter declaration> that specifies a <parameter default>.

35) Without Feature B121, "Routine language Ada", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains ADA.

36) Without Feature B122, "Routine language C", conforming SQL language shall not contain a <routine characteristic>that contains a <language clause> that contains C.

37) Without Feature B123, "Routine language COBOL", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains COBOL.

38) Without Feature B124, "Routine language Fortran", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains FORTRAN.

39) Without Feature B125, "Routine language MUMPS", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains M.

40) Without Feature B126, "Routine language Pascal", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains PASCAL.

41) Without Feature B127, "Routine language PL/I", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains PLI.

42) Without Feature B128, "Routine language SQL", conforming SQL language shall not contain a <routine characteristic> that contains a <language clause> that contains SQL.

43) Without Feature B221, "Routine language Ada: VARCHAR and NUMERIC support", if the <language clause> of the <SQL-invoked routine> specifies ADA, then <parameter type> and <returns data type> shall not specify CHARACTER VARYING or NUMERIC.

44) Without Feature B200, "Polymorphic table functions", conforming SQL language shall not contain an <SQL-invoked routine> that defines a polymorphic table function.

45) Without Feature B201, "More than one PTF generic table parameter", an <SQL-invoked routine> shall contain at most one <generic table parameter type>.

46) Without Feature B204, "PRUNE WHEN EMPTY", an <SQL-invoked routine> shall not contain <generic table pruning>.

47) Without Feature B205, "Pass-through columns", an <SQL-invoked routine> shall not contain <pass through option>.

48) Without Feature B205, "Pass-through columns", <returns table type> shall not be ONLY PASS THROUGH.

49) [13] [14] Without Feature B206, "PTF descriptor parameters", an <SQL-invoked routine> shall not contain <descriptor parameter type>.

11.61 <alter routine statement>

This Subclause is modified by Subclause 10.9, "<alter routine statement>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.14, "<alter routine statement>", in ISO/IEC 9075-16.

Function

Alter a characteristic of an SQL-invoked routine.

Format

```
<alter routine statement> ::=
  ALTER <specific routine designator>
      <alter routine characteristics> <alter routine behavior>

<alter routine characteristics> ::=
  <alter routine characteristic>...

<alter routine characteristic> ::=
    <language clause>
  | <parameter style clause>
  | <SQL-data access indication>
  | <null-call clause>
  | <returned result sets characteristic>
  | NAME <external routine name>

<alter routine behavior> ::=
  RESTRICT
```

Syntax Rules

1) 13 Let *SR* be the SQL-invoked routine identified by the <specific routine designator> and let *SN* be the <specific name> of *SR*. The schema identified by the explicit or implicit <schema name> of *SN* shall include the descriptor of *SR*.

2) *SR* shall be a schema-level routine.

3) *SR* shall not be an SQL-invoked routine that is dependent on a user-defined type.

   NOTE 639 — "SQL-invoked routine dependent on a user-defined type" is defined in Subclause 4.35, "SQL-invoked routines".

4) If RESTRICT is specified, then:

   a) *SR* shall not be the ordering function included in the descriptor of any user-defined type *UDT*.

   b) *SR* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, <method reference>, or a PTF component procedure of any polymorphic table function that is the subject routine of a <PTF derived table> that is contained in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The original <query expression> of any view descriptor.

      iii) The <search condition> of any constraint descriptor.

      iv) 16 The triggered action of any trigger descriptor.

   c) *SN* shall not be included in any of the following:

i) A group descriptor of any transform descriptor.

ii) A user-defined cast descriptor.

5) *SR* shall be an external routine.

6) *SR* shall not be an SQL-invoked method that is an overriding method and the set of overriding methods of *SR* shall be empty.

7) <alter routine characteristics> shall contain at most one <language clause>, at most one <parameter style clause>, at most one <SQL-data access indication>, at most one <null-call clause>, at most one <returned result sets characteristic>, and at most one <external routine name>.

8) If <returned result sets characteristic> is specified, then *SR* shall be an SQL-invoked procedure and *SR* shall not be a PTF component procedure of a polymorphic table function.

9) If <null-call clause> is specified, then *SR* shall be an SQL-invoked function.

10) If *SR* is a PTF component procedure of a polymorphic table function, then:

a) <SQL-data access indication> shall not specify MODIFIES SQL DATA.

b) <returned result sets characteristic> and <savepoint level indication> shall not be specified.

11) If <language clause> is specified, then:

a) <language clause> shall not specify SQL.

b) Depending on whether the <language clause> specifies ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, let the operative data type correspondences table be Table 21, "Data type correspondences for Ada", Table 22, "Data type correspondences for C", Table 23, "Data type correspondences for COBOL", Table 24, "Data type correspondences for Fortran", Table 25, "Data type correspondences for M", Table 26, "Data type correspondences for Pascal", or Table 27, "Data type correspondences for PL/I", respectively. Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

c) Any <data type> in the effective SQL parameter list entry of *SR* shall specify a data type listed in the SQL data type column for which the corresponding row in the host data type column is not "None".

NOTE 640 — "Effective SQL parameter list" is defined in Subclause 11.60, "<SQL-invoked routine>".

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *SN*.

General Rules

1) If *SR* is not a method, then the routine descriptor of *SR* is modified:

a) If <returned result sets characteristic> is specified, then the maximum number of returned result sets is value of <maximum returned result sets>.

b) If <language clause> is specified, then the name of the language in which the body of the SQL-invoked routine is written is <language name> contained in the <language clause>.

c) If <external routine name> is specified, then the external name of the routine descriptor is <external routine name>.

d) If <parameter style clause> is specified, then the routine descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

e) If the <SQL-data access indication> is specified, then the routine descriptor includes an indication of whether the SQL-invoked routine's <SQL-data access indication> is READS SQL DATA, MODIFIES SQL DATA, CONTAINS SQL, or NO SQL.

f) If <null-call clause> is specified, then the routine descriptor includes an indication of whether the SQL-invoked routine is a null-call function.

2) If *SR* is a method, then let *DMS* be the descriptor of the corresponding method specification. *DMS* is modified:

a) If <language clause> is specified, then the method specification descriptor includes the <language name> contained in the <language clause>.

b) If <parameter style clause> is specified, then the method specification descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

c) If the <SQL-data access indication> is specified, then the method specification descriptor includes an indication of whether the SQL-invoked routine's <SQL-data access indication> is READS SQL DATA, MODIFIES SQL DATA, CONTAINS SQL, or NO SQL.

d) If <null-call clause> is specified, then the method specification descriptor includes an indication of whether the method is to be invoked if any argument is the null value.

3) If *SR* is a method, then the routine descriptor of *SR* is modified:

a) If <external routine name> is specified, then the external name of the routine descriptor is <external routine name>.

b) If <parameter style clause> is specified, then the routine descriptor includes an indication of whether the parameter passing style is PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

4) The last-altered timestamp included in the routine descriptor of *SR* is the value of CURRENT_TIMESTAMP.

Conformance Rules

1) Without Feature F381, "Extended schema manipulation", conforming SQL language shall not contain an <alter routine statement>.

11.62 <drop routine statement>

This Subclause is modified by Subclause 10.25, "<drop routine statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.12, "<drop routine statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.10, "<drop routine statement>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.15, "<drop routine statement>", in ISO/IEC 9075-16.

Function

Destroy an SQL-invoked routine.

Format

```
<drop routine statement> ::=
  DROP <specific routine designator> <drop behavior>
```

Syntax Rules

1) Let *SR* be the SQL-invoked routine identified by the <specific routine designator> and let *SN* be the <specific name> of *SR*. The schema identified by the explicit or implicit <schema name> of *SN* shall include the descriptor of *SR*.

2) *SR* shall be a schema-level routine.

3) *SR* shall not be dependent on any user-defined type.

   NOTE 641 — The notion of an SQL-invoked routine being dependent on a user-defined type is defined in Subclause 4.35, "SQL-invoked routines".

4) If RESTRICT is specified, then:

   a) *SR* shall not be the ordering function included in the descriptor of any user-defined type *UDT*.

   b) *SR* shall not be a PTF component procedure of a polymorphic table function.

   c) *SR* shall not be the subject routine of any <routine invocation>, <method invocation>, <static method invocation>, or <method reference> that is contained in any of the following:

      i) The SQL routine body of any routine descriptor.

      ii) The <parameter default> of any SQL parameter of any routine descriptor.

      iii) The original <query expression> of any view descriptor.

      iv) The <search condition> of any constraint descriptor.

      v) 04 16 The triggered action of any trigger descriptor.

   d) *SN* shall not be included in any of the following:

      i) A group descriptor of any transform descriptor.

      ii) 09 A user-defined cast descriptor.

   NOTE 642 — If CASCADE is specified, then such referencing objects will be dropped by the execution of the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

5) 13 Let the containing schema be the schema identified by the <schema name> explicitly or implicitly contained in *SN*.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of *SN*. The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
REVOKE EXECUTE ON SPECIFIC ROUTINE SN FROM A CASCADE
```

2) 09 Let *DN* be the <user-defined type name> of a user-defined type whose descriptor includes *SN* in the group descriptor of any transform descriptor. Let *GN* be the <group name> of that group descriptor. The following <drop transform statement> is effectively executed without further Access Rule checking:

```
DROP TRANSFORM GN FOR DN CASCADE
```

3) Let *UDCD* be a user-defined cast descriptor that includes *SN* as its cast function. Let *SDN* be the name of the source data type included in *UDCD*. Let *TDN* be the name of the target data type included in *UDCD*. The following <drop user-defined cast statement> is effectively executed without further Access Rule checking:

```
DROP CAST ( SDN AS TDN ) CASCADE
```

4) If *SR* is the ordering function included in the descriptor of a user-defined type *UDT*, then let *UDTN* be a <path-resolved user-defined type name> that identifies *UDT*. The following <drop user-defined ordering statement> is effectively executed without further Access Rule checking:

```
DROP ORDERING FOR UDTN CASCADE
```

5) If *SR* is a PTF component procedure of a polymorphic table function *PTF*, then let *PTFN* be a <specific routine designator> that identifies *PTF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP ROUTINE PTFN
```

6) The descriptor of *SR* is destroyed.

Conformance Rules

1) Without Feature F032, "CASCADE drop behavior", conforming SQL language shall not contain a <drop routine statement> that contains a <drop behavior> that contains CASCADE.

2) 13 Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <drop routine statement> that contains a <specific routine designator> that identifies a method.

11.63 <user-defined cast definition>

This Subclause is modified by Subclause 10.13, "<user-defined cast definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 12.9, "<user-defined cast definition>", in ISO/IEC 9075-14.

Function

Define a user-defined cast.

Format

```
<user-defined cast definition> ::=
  CREATE CAST <left paren> <source data type> AS <target data type> <right paren>
      WITH <cast function>
      [ AS ASSIGNMENT ]

<cast function> ::=
  <specific routine designator>

<source data type> ::=
  <data type>

<target data type> ::=
  <data type>
```

Syntax Rules

1) 09 Let *SDT* be the <source data type>. The data type identified by *SDT* is called the *source data type*.

2) Let *TDT* be the <target data type>. The data type identified by *TDT* is called the *target data type*.

3) There shall be no user-defined cast for *SDT* and *TDT*.

4) At least one of *SDT* or *TDT* shall contain a <schema-resolved user-defined type name> or a <reference type>.

5) If *SDT* contains a <schema-resolved user-defined type name>, then let *SSDT* be the schema that includes the descriptor of the user-defined type identified by *SDT*.

6) If *SDT* contains a <reference type>, then let *SSDT* be the schema that includes the descriptor of the referenced type of the reference type identified by *SDT*.

7) If *TDT* contains a <schema-resolved user-defined type name>, then let *STDT* be the schema that includes the descriptor of the user-defined type identified by *TDT*.

8) If *TDT* contains a <reference type>, then let *STDT* be the schema that includes the descriptor of the referenced type of the reference type identified by *TDT*.

9) If both *SDT* and *TDT* contain a <schema-resolved user-defined type name> or a <reference type>, then the <authorization identifier> that owns *SSDT* and the <authorization identifier> that owns *STDT* shall be equivalent.

10) 14 Let *F* be the SQL-invoked routine identified by <cast function>. *F* is called the *cast function* for source data type *SDT* and target data type *TDT*.

    a) *F* shall have exactly one SQL parameter, and its declared type shall be *SDT*.

    b) The result data type of *F* shall be *TDT*.

c) The <authorization identifier> that owns *SSDT* or *STDT* (both, if both *SDT* and *TDT* are <schema-resolved user-defined type name>s) shall own the schema that includes the SQL-invoked routine descriptor of *F*.

d) *F* shall be deterministic.

e) *F* shall not possibly modify SQL-data.

f) *F* shall not possibly read SQL-data.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema that includes the routine descriptor of *F*.

2) If *SDT* contains a <schema-resolved user-defined type name> or a <reference type>, then the enabled authorization identifiers shall include the <authorization identifier> that owns *SSDT*.

3) If *TDT* contains a <schema-resolved user-defined type name> or a <reference type>, then the enabled authorization identifiers shall include the <authorization identifier> that owns *STDT*.

General Rules

1) A user-defined cast descriptor *CFD* is created that describes the user-defined cast. *CFD* includes the name of the source data type, the name of the target data type, the specific name of the cast function, and, if and only if AS ASSIGNMENT is specified, an indication that the cast function is implicitly invocable.

Conformance Rules

1) Without Feature S211, "User-defined cast functions", conforming SQL language shall not contain a <user-defined cast definition>.

11.64 <drop user-defined cast statement>

This Subclause is modified by Subclause 10.26, "<drop user-defined cast statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.16, "<drop user-defined cast statement>", in ISO/IEC 9075-16.

Function

Destroy a user-defined cast.

Format

```
<drop user-defined cast statement> ::=
  DROP CAST <left paren> <source data type> AS <target data type> <right paren>
      <drop behavior>
```

Syntax Rules

1) Let *SDT* be the <source data type> and let *TDT* be the <target data type>.

2) Let *CF* be the user-defined cast whose user-defined cast descriptor includes *SDT* as the source data type and *TDT* as the target data type.

3) Let *SN* be the specific name of the cast function *F* included in the user-defined cast descriptor of *CF*.

4) The schema identified by the <schema name> of *SN* shall include the descriptor of *F*.

5) A *relevant cast specification* is a <cast specification> *CS* such that:

   a) The <value expression> of *CS* has declared type *P*.

   b) The <cast target> of *CS* is either *TDT* or a domain with declared type *TDT*.

   c) The type designator of *SDT* is in the type precedence list of *P*.

   d) No other data type *Q* whose type designator precedes *SDT* in the type precedence list of *P* such that there is a user-defined cast CF_q whose user-defined cast descriptor includes *Q* as the source data type and *TDT* as the target data type.

6) A *relevant procedure statement* is an <SQL procedure statement> that is dependent on *F*.

7) If RESTRICT is specified, then no relevant cast specification and no relevant procedure statement shall be contained in any of the following:

   a) The SQL routine body of any routine descriptor.

   b) The original <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) 04 16 The triggered action of any trigger descriptor.

   NOTE 643 — If CASCADE is specified, then such referencing objects will be dropped implicitly by the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifier shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *SN*.

General Rules

1) For every SQL-invoked routine that contains a relevant cast specification or a relevant procedure statement in its SQL routine body:

   a) Let *R* be the SQL-invoked routine and let *SN* be the specific name of *R*.

   b) The following <drop routine statement> is effectively executed without further Access Rule checking:

      ```
      DROP SPECIFIC ROUTINE SN CASCADE
      ```

2) For every view that contains a relevant cast specification or a relevant procedure statement in its original <query expression>:

   a) Let *V* be the view and let *VN* be the table name of *V*.

   b) The following <drop view statement> is effectively executed without further Access Rule checking:

      ```
      DROP VIEW VN CASCADE
      ```

3) For every table that contains a relevant cast specification or a relevant procedure statement in the <search condition> of any constraint descriptor included in its table descriptor:

   a) Let *T* be the table and let *TN* be the table name of *T*.

   b) The following <drop table statement> is effectively executed without further Access Rule checking:

      ```
      DROP TABLE TN CASCADE
      ```

4) For every assertion that contains a relevant cast specification or a relevant procedure statement in its <search condition>:

   a) Let *A* be the assertion and let *AN* be the constraint name of *A*.

   b) The following <drop assertion statement> is effectively executed without further Access Rule checking:

      ```
      DROP ASSERTION AN CASCADE
      ```

5) For every domain that contains a relevant cast specification or a relevant procedure statement in the <search condition> of any constraint descriptor included in the domain descriptor of the domain:

   a) Let *D* be the domain and let *DN* be the domain name of *D*.

   b) The following <drop domain statement> is effectively executed without further Access Rule checking:

      ```
      DROP DOMAIN DN CASCADE
      ```

6) For every trigger whose trigger descriptor includes a triggered action that contains a relevant cast specification or a relevant procedure statement:

a) Let *T* be the trigger and let *TN* be the trigger name of *T*.

b) The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER TN
```

7) The descriptor of *CF* is destroyed.

Conformance Rules

1) Without Feature S211, "User-defined cast functions", conforming SQL language shall not contain a <drop user-defined cast statement>.

11.65 <user-defined ordering definition>

This Subclause is modified by Subclause 10.14, "<user-defined ordering definition>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 10.11, "<user-defined ordering definition>", in ISO/IEC 9075-13.

Function

Define a user-defined ordering for a user-defined type.

Format

```
<user-defined ordering definition> ::=
  CREATE ORDERING FOR <schema-resolved user-defined type name> <ordering form>

<ordering form> ::=
    <equals ordering form>
  | <full ordering form>

<equals ordering form> ::=
  EQUALS ONLY BY <ordering category>

<full ordering form> ::=
  ORDER FULL BY <ordering category>

13 <ordering category> ::=
    <relative category>
  | <map category>
  | <state category>

<relative category> ::=
  RELATIVE WITH <relative function specification>

<map category> ::=
  MAP WITH <map function specification>

<state category> ::=
  STATE [ <specific name> ]

<relative function specification> ::=
  <specific routine designator>

<map function specification> ::=
  <specific routine designator>
```

Syntax Rules

1) Let *UDTN* be the <schema-resolved user-defined type name>. Let *UDT* be the user-defined type identified by *UDTN*.

2) The descriptor of *UDT* shall include an ordering form that specifies NONE.

3) If *UDT* is not a maximal supertype, then

   Case:

   a) If <equals ordering form> is specified, then the comparison form of every direct supertype of *UDT* shall be EQUALS.

   b) Otherwise, the comparison form of every direct supertype of *UDT* shall be FULL.

4) 13 *UDT* shall be a maximal supertype if at least one of the following is specified:

   a) <relative category>

   b) <state category>

5) If <map category> is specified and *UDT* is not a maximal supertype, then the comparison category of every direct supertype of *UDT* shall be MAP.

   NOTE 644 — The comparison categories of two user-defined types in the same subtype family must be the same.

6) Case:

   a) If <state category> is specified, then:

      i) *UDT* shall not be a distinct type.

      ii) EQUALS ONLY shall be specified.

      iii) 09 The declared type of each attribute of *UDT* shall not be UDT-NC-ordered.

      iv) Case:

         1) If <specific name> is specified, then let *SN* be <specific name>. If *SN* contains a <schema name>, then that <schema name> shall be equivalent to the <schema name> of *UDTN*.

         2) Otherwise, let *SN* be an implementation-dependent (UV102) <specific name> whose <schema name> is equivalent to the <schema name> *S* of *UDTN*. This implementation-dependent (UV102) <specific name> shall not be equivalent to the <specific name> of any other routine descriptor in the schema identified by *S*.

   b) 13 Otherwise:

      i) Let *F* be the SQL-invoked routine identified by the <specific routine designator> *SRD*.

      ii) *F* shall be deterministic.

      iii) *F* shall not possibly modify SQL-data.

7) If <relative function specification> is specified, then:

   a) *F* shall have exactly two SQL parameters whose declared type is *UDT*.

   b) *F* shall be an SQL-invoked regular function.

   c) The result data type of *F* shall be INTEGER.

   NOTE 645 — The Syntax Rules and General Rules of Subclause 8.2, "<comparison predicate>", expect that the result of *F* is 0 (zero) if the two arguments are deemed to be equal, -1 if the first argument is deemed to be less than the second argument, and 1 (one) if the first argument is deemed to be greater than the second argument.

8) If <map function specification> is specified, then:

   a) *F* shall have exactly one SQL parameter whose declared type is *UDT*.

   b) The result data type of *F* shall be a predefined data type.

   c) The result data type of *F* is an operand of an equality operation. The Syntax Rules and Conformance Rules of Subclause 9.11, "Equality operations", apply.

   d) If FULL is specified, then the result data type of *F* is an operand of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 9.14, "Ordering operations", apply.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema that includes the descriptor of *UDT* and the schema that includes the routine descriptor of *F*.

General Rules

1) If <state category> is specified, then:

   a) Let C_1, ..., C_n be the components of the representation of the user-defined type.

   b) Let *SNUDT* be the <schema name> of the schema that includes the descriptor of *UDT*.

   c) The following <SQL-invoked routine> is effectively executed:

```
CREATE FUNCTION SNUDT.EQUALS ( UDT1 UDTN, UDT2 UDTN )
    RETURNS BOOLEAN
    SPECIFIC SN
    DETERMINISTIC
    CONTAINS SQL
    STATIC DISPATCH
    RETURN
      ( TRUE AND
        UDT1.SPECIFICTYPE = UDT2.SPECIFICTYPE AND
        UDT1.C1 = UDT2.C1 AND
        ...
        UDT1.Cn = UDT2.Cn )
```

2) Case:

   a) If EQUALS is specified, then the ordering form in the user-defined type descriptor of *UDT* is set to EQUALS.

   b) Otherwise, the ordering form in the user-defined type descriptor of *UDT* is set to FULL.

3) Case:

   a) If RELATIVE is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to RELATIVE.

   b) If MAP is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to MAP.

   c) 13 Otherwise, the ordering category in the user-defined type descriptor of *UDT* is set to STATE.

4) The <specific routine designator> identifying the ordering function, depending on the ordering category, in the descriptor of *UDT* is set to *SRD*.

Conformance Rules

1) 13 Without Feature S251, "User-defined orderings", conforming SQL shall not contain a <user-defined ordering definition>.

NOTE 646 — If MAP is specified, then the Conformance Rules of Subclause 9.11, "Equality operations", apply. If ORDER FULL BY MAP is specified, then the Conformance Rules of Subclause 9.14, "Ordering operations", also apply.

11.66 <drop user-defined ordering statement>

This Subclause is modified by Subclause 10.20, "<drop user-defined ordering statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 10.12, "<drop user-defined ordering statement>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 11.17, "<drop user-defined ordering statement>", in ISO/IEC 9075-16.

Function

Destroy a user-defined ordering method.

Format

```
<drop user-defined ordering statement> ::=
  DROP ORDERING FOR <schema-resolved user-defined type name> <drop behavior>
```

Syntax Rules

1) Let *UDTN* be the <schema-resolved user-defined type name>. Let *UDT* be the user-defined type identified by *UDTN*.

2) The descriptor of *UDT* shall include an ordering form that specifies EQUALS or FULL.

3) Let *OF* be the ordering function of *UDT*.

4) If RESTRICT is specified, then none of the following shall contain an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*:

   a) The SQL routine body of any routine descriptor.

   b) The original <query expression> of any view descriptor.

   c) The <search condition> of any constraint descriptor.

   d) 04 16 The triggered action of any trigger descriptor.

   NOTE 647 — If CASCADE is specified, then such referencing objects will be dropped implicitly by the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *UDTN*.

General Rules

1) Let *RS* be the set of all SQL-invoked routines whose <SQL routine body> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*. For every routine in *RS*:

   a) 04 Let *R* be the routine and let *SN* be the specific name of *R*.

   b) The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

2) For every view whose original <query expression> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*:

   a) Let *V* be the view and let *VN* be the table name of *V*.

   b) The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW VN CASCADE
```

3) For every table constraint *C* whose <search condition> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type whose comparison type is *UDT*:

   a) Let *T* be the table constrained by *C*, let *TN* be the table name of *T*, and let *TCN* be the constraint name of *C*.

   b) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE TN DROP CONSTRAINT TCN CASCADE
```

4) For every assertion whose <search condition> contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type *T1* whose comparison type is *UDT*:

   a) Let *A* be the assertion and let *AN* be the constraint name of *A*.

   b) The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION AN CASCADE
```

5) For every domain whose descriptor includes a constraint descriptor that includes an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type whose comparison type is *UDT*:

   a) Let *D* be the domain and let *DN* be the <domain name> of *D*.

   b) The following <drop domain statement> is effectively executed without further Access Rule checking:

```
DROP DOMAIN DN CASCADE
```

6) For every trigger whose triggered action contains an operand of an equality operation, grouping operation, or ordering operation whose declared type is some user-defined type whose comparison type is *UDT*:

   a) Let *T* be the trigger and let *TN* be the <trigger name> of *T*.

   b) The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER TN
```

7) In the descriptor of *UDT*, the ordering form is set to NONE and the ordering category is set to STATE. No ordering function is included in the descriptor of *UDT*.

Conformance Rules

1) 13 Without Feature S251, “User-defined orderings”, conforming SQL language shall not contain a <drop user-defined ordering statement>.

11.67 <transform definition>

Function

Define one or more transform functions for a user-defined type.

Format

```
<transform definition> ::=
  CREATE { TRANSFORM | TRANSFORMS } FOR
      <schema-resolved user-defined type name> <transform group>...

<transform group> ::=
  <group name> <left paren> <transform element list> <right paren>

<group name> ::=
  <identifier>

<transform element list> ::=
  <transform element> [ <comma> <transform element> ]

<transform element> ::=
    <to sql>
  | <from sql>

<to sql> ::=
  TO SQL WITH <to sql function>

<from sql> ::=
  FROM SQL WITH <from sql function>

<to sql function> ::=
  <specific routine designator>

<from sql function> ::=
  <specific routine designator>
```

Syntax Rules

1) Let *TD* be the <transform definition>. Let *DTN* be the <schema-resolved user-defined type name> immediately contained in *TD*. Let *DT* be the data type identified by *DTN*. Let *SDT* be the schema that includes the descriptor of *DT*. Let *TRD* be the transform descriptor included in the data type descriptor of *DT*.

2) No two <transform group>s immediately contained in *TD* shall have the same <group name>.

3) The SQL-invoked function identified by <to sql function> is called the *to-sql function*. The SQL-invoked function identified by <from sql function> is called the *from-sql function*.

4) Let *n* be the number of <transform group>s immediately contained in *TD*. For *i* ranging from 1 to *n*:

   a) Let TG_i be the *i*-th <transform group> immediately contained in *TD*. Let GN_i be the <group name> contained in TG_i.

   b) Each of <to sql> and <from sql> immediately contained in TG_i shall be contained at most once in a <transform element list>.

c) The SQL-invoked routines identified by <to sql function> and <from sql function> shall be SQL-invoked functions that are deterministic and do not possibly modify SQL-data.

d) *TRD* shall not include a transform group descriptor *GD* that includes a group name that is equivalent to GN_i.

e) Let *SDTT* be the set that includes every data type DTT_j that is either a proper supertype or a proper subtype of *DT* such that the transform descriptor included in the data type descriptor of DTT_j includes a group descriptor $GDT_{j,k}$ that includes a group name that is equivalent to GN_i. *SDTT* shall be empty.

f) If <to sql> is specified, then let TSF_i be the SQL-invoked function identified by <to sql function>.

i) Case:

1) If TSF_i is an SQL-invoked method, then TSF_i shall have exactly two SQL parameters such that the declared type of the first SQL parameter is *DT* and the declared type of the second SQL parameter is a predefined data type. The result data type of TSF_i shall be *DT*.

2) Otherwise, TSF_i shall have exactly one SQL parameter whose declared type is a predefined data type. The result data type of TSF_i shall be *DT*.

ii) If *DT* is a structured type and TSF_i is an SQL-invoked method, then TSF_i shall be a type-preserving function.

g) If <from sql> is specified, then let FSF_i be the SQL-invoked function identified by <from sql function>. FSF_i shall have exactly one SQL parameter whose declared type is *DT*. The result data type of FSF_i shall be a predefined data type.

h) If <to sql> and <from sql> are both specified, then

Case:

i) If TSF_i is an SQL-invoked method, then the result data type of FSF_i and the data type of the second SQL parameter of TSF_i shall be compatible.

ii) Otherwise, the result data type of FSF_i and the data type of the first SQL parameter of TSF_i shall be compatible.

Access Rules

1) For i ranging from 1 to n, the enabled authorization identifiers shall include the <authorization identifier> that owns *SDT* and the schema that includes the routine descriptors of TSF_i, if any, and FSF_i, if any.

General Rules

1) A <group name> specifies the group name that identifies a transform group.

2) For every TG_i, 1 (one) $\leq i \leq n$:

a) A new group descriptor GD_i is created that includes the <group name> immediately contained in TG_i. GD_i is included in the list of transform group descriptors included in *TRD*.

b) If <to sql> is specified, then the specific name of the to-sql function in GD_i is set to TSF_i.

c) If <from sql> is specified, then the specific name of the from-sql function in GD_i is set to FSF_i.

Conformance Rules

1) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <transform definition>.

11.68 <alter transform statement>

Function

Change the definition of one or more transform groups.

Format

```
<alter transform statement> ::=
  ALTER { TRANSFORM | TRANSFORMS }
       FOR <schema-resolved user-defined type name> <alter group>...

<alter group> ::=
  <group name> <left paren> <alter transform action list> <right paren>

<alter transform action list> ::=
  <alter transform action> [ { <comma> <alter transform action> }... ]

<alter transform action> ::=
     <add transform element list>
   | <drop transform element list>
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> and let *D* be the data type identified by *DN*. The schema identified by the explicit or implicit <schema name> of *DN* shall include the data type descriptor of *D*. Let *S* be that schema. Let *TD* be the transform descriptor included in the data type descriptor of *D*.

2) The scope of *DN* is the entire <alter transform statement> *AT*.

3) Let *n* be the number of <group name>s contained in *AT*. For *i* ranging from 1 to *n*:

   a) Let GN_i be the *i*-th <group name> contained in *AT*.

   b) For each GN_i, there shall be a transform group descriptor included in *TD* whose group name is equivalent to GN_i. Let GD_i be this transform group descriptor.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns *S*.

General Rules

1) For *i* ranging from 1 to *n*, GD_i is modified as specified by <alter transform action list>.

Conformance Rules

1) Without Feature S242, "Alter transform statement", conforming SQL language shall not contain an <alter transform statement>.

11.69 <add transform element list>

Function

Add a transform element (<to sql> and/or <from sql>) to an existing transform group.

Format

```
<add transform element list> ::=
  ADD <left paren> <transform element list> <right paren>
```

Syntax Rules

1) Let *AD* be the <add transform element list>.

2) Let *DN* be the <schema-resolved user-defined type name> immediately contained in the containing <alter transform statement>. Let *D* be the user-defined type identified by *DN*. Let *TD* be the transform descriptor included in the data type descriptor of *D*.

3) Let *GD* be the transform group descriptor included in *TD* whose group name is equivalent to <group name> immediately contained in the containing <alter group>.

4) Each of <to sql> and <from sql> (immediately contained in *AD*) shall be contained at most once in the <transform element list>.

5) If *GD* includes a specific name of the to-sql function, then *AD* shall not contain <to sql>.

6) If *GD* includes a specific name of the from-sql function, then *AD* shall not contain <from sql>.

7) The SQL-invoked routine identified by either <to sql function> or <from sql function> shall be an SQL-invoked function that is deterministic and does not possibly modify SQL-data.

8) If <to sql> is specified, then let *TSF* be the SQL-invoked function identified by <to sql function>.

   a) Case:

      i) If *TSF* is an SQL-invoked method, then *TSF* shall have exactly two SQL parameters such that the declared type of the first SQL parameter is *D* and the declared type of the second SQL parameter is a predefined data type. The result data type of *TSF* shall be *D*.

      ii) Otherwise, *TSF* shall have exactly one SQL parameter whose declared type is a predefined data type. The result data type of *TSF* shall be *D*.

   b) If *D* is a structured type, then *TSF* shall be a type-preserving function.

   c) If *GD* includes the specific name of a from-sql function, then let *FS* be the SQL-invoked function that is identified by this specific name.

      Case:

      i) If *TSF* is an SQL-invoked method, then the result data type of *FS* and the data type of the second SQL parameter of *TSF* shall be compatible.

      ii) Otherwise, the result data type of *FS* and the data type of the first SQL parameter of *TSF* shall be compatible.

9) If <from sql> is specified, then let *FSF* be the SQL-invoked function identified by <from sql function>.

a) *FSF* shall have exactly one SQL parameter whose declared type is *D*. The result data type of *FSF* shall be a predefined data type.

b) If *GD* includes the specific name of a to-sql function, then let *TS* be the SQL-invoked routine that is identified by this specific name.

Case:

i) If *TS* is an SQL-invoked method, then the result data type of *FSF* and the data type of the second SQL parameter of *TS* shall be compatible.

ii) Otherwise, the result data type of *FSF* and the data type of the first SQL parameter of *TS* shall be compatible.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema that includes the routine descriptors of *TSF*, if any, and *FSF*, if any.

General Rules

1) If <to sql> is specified, then the specific name of the to-sql function in *GD* is set to *TSF*.

2) If <from sql> is specified, then the specific name of the from-sql function in *GD* is set to *FSF*.

Conformance Rules

None.

11.70 <drop transform element list>

Function

Remove a transform element (<to sql> and/or <from sql>) from a transform group.

Format

```
<drop transform element list> ::=
  DROP <left paren> <transform kind>
      [ <comma> <transform kind> ] <drop behavior> <right paren>

<transform kind> ::=
    TO SQL
  | FROM SQL
```

Syntax Rules

1) Let *DN* be the <schema-resolved user-defined type name> immediately contained in the containing <alter transform statement>. Let *D* be the user-defined type identified by *DN*. Let *TD* be the transform descriptor included in the data type descriptor of *D*.

2) Let *GD* be the transform group descriptor included in *TD* whose group name is equivalent to <group name> immediately contained in the containing <alter group>.

3) Each of TO SQL and FROM SQL shall only be specified at most once in the <drop transform element list>.

4) If TO SQL is specified then *GD* shall include the specific name of a to-sql function. Let *TSF* be this function.

5) If FROM SQL is specified then *GD* shall include the specific name of a from-sql function. Let *FSF* be this function.

6) If RESTRICT is specified, then:

   a) If TO SQL is specified, then there shall be no external routine that has an SQL parameter whose associated to-sql function is *TSF* nor shall there be an external function that has *TSF* as the to-sql function associated with the result.

   b) If FROM SQL is specified, then there shall be no external routine that has an SQL parameter whose associated from-sql function is *FSF*.

Access Rules

None.

General Rules

1) If FROM SQL is specified, then:

   a) Let *FSN* be the <specific name> of any external routine that has an SQL parameter whose associated from-sql function is *FSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE FSN CASCADE
```

b) The specific name of the from-sql function is removed from *GD*.

2) If TO SQL is specified, then:

a) Let *TSN* be the <specific name> of any external routine that has an SQL parameter whose associated to-sql function is *TSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE TSN CASCADE
```

b) Let *RSN* be the <specific name> of any external function that has *TSF* as the to-sql function associated with the result. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE RSN CASCADE
```

c) The specific name of the to-sql function is removed from *GD*.

Conformance Rules

None.

11.71 <drop transform statement>

Function

Remove one or more transform functions associated with a transform.

Format

```
<drop transform statement> ::=
  DROP { TRANSFORM | TRANSFORMS } <transforms to be dropped>
      FOR <schema-resolved user-defined type name> <drop behavior>

<transforms to be dropped> ::=
    ALL
  | <transform group element>

<transform group element> ::=
  <group name>
```

Syntax Rules

1) Let *DT* be the data type identified by <schema-resolved user-defined type name>. Let *SDT* be the schema that includes the descriptor of *DT*. Let *TRD* be the transform descriptor included in the data type descriptor of *DT*. Let *n* be the number of transform group descriptors in *TRD*.

2) If <transform group element> is specified, then *TRD* shall include a transform group descriptor *GD* that includes a group name that is equivalent to the <group name> immediately contained in <transform group element>.

3) If RESTRICT is specified, then

   Case:

   a) If ALL is specified, then for *i* ranging from 1 (one) to *n*:

      i) Let GD_i be the *i*-th transform group descriptor included in *TRD*.

      ii) If GD_i includes the specific name of a from-sql function FSF_i then there shall be no external routine that has an SQL parameter whose associated from-sql function is FSF_i.

      iii) If GD_i includes the specific name of a to-sql function TSF_i then there shall be no external routine that has an SQL parameter whose associated to-sql function is TSF_i nor shall there be an external function that has TSF_i as the to-sql function associated with the result.

   b) Otherwise:

      i) If *GD* includes the specific name of a from-sql function *FSF* then there shall be no external routine that has an SQL parameter whose associated from-sql function is *FSF*.

      ii) If *GD* includes the specific name of a to-sql function *TSF* then there shall be no external routine that has an SQL parameter whose associated to-sql function is *TSF* nor shall there be an external function that has *TSF* as the to-sql function associated with the result.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns *SDT*.

General Rules

1) Case:
   a) If ALL is specified, then, for *i* ranging from 1 (one) to *n*:
      i) Let GD_i be the *i*-th transform group descriptor included in *TRD*.
      ii) If GD_i includes the specific name of a from-sql function FSF_i, then let *FSN* be the <specific name> of any external routine that has an SQL parameter whose associated from-sql function is FSF_i. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE FSN CASCADE
```

      iii) If GD_i includes the specific name of a to-sql function TSF_i, then:
         1) Let *TSN* be the <specific name> of any external routine that has an SQL parameter whose associated to-sql function is TSF_i. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE TSN CASCADE
```

         2) Let *RSN* be the <specific name> of any external function that has TSF_i as the to-sql function associated with the result. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE RSN CASCADE
```

      iv) GD_i is removed from *TRD*.
   b) Otherwise:
      i) If *GD* includes the specific name of a from-sql function *FSF*, then let *FSN* be the <specific name> of any external routine that has an SQL parameter whose associated from-sql function is *FSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE FSN CASCADE
```

      ii) If *GD* includes the specific name of a to-sql function *TSF*, then:
         1) Let *TSN* be the <specific name> of any external routine that has an SQL parameter whose associated to-sql function is *TSF*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE TSN CASCADE
```

         2) Let *RSN* be the <specific name> of any external function that has *TSF* as the to-sql function associated with the result. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE RSN CASCADE
```

iii) *GD* is removed from *TRD*.

Conformance Rules

1) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <drop transform statement>.

11.72 <sequence generator definition>

Function

Define an external sequence generator.

Format

```
<sequence generator definition> ::=
  CREATE SEQUENCE <sequence generator name> [ <sequence generator options> ]

<sequence generator options> ::=
  <sequence generator option>...

<sequence generator option> ::=
    <sequence generator data type option>
  | <common sequence generator options>

<common sequence generator options> ::=
  <common sequence generator option>...

<common sequence generator option> ::=
    <sequence generator start with option>
  | <basic sequence generator option>

<basic sequence generator option> ::=
    <sequence generator increment by option>
  | <sequence generator max-value option>
  | <sequence generator min-value option>
  | <sequence generator cycle option>

<sequence generator data type option> ::=
  AS <data type>

<sequence generator start with option> ::=
  START WITH <sequence generator start value>

<sequence generator start value> ::=
  <signed numeric literal>

<sequence generator increment by option> ::=
  INCREMENT BY <sequence generator increment>

<sequence generator increment> ::=
  <signed numeric literal>

<sequence generator max-value option> ::=
    MAXVALUE <sequence generator max value>
  | NO MAXVALUE

<sequence generator max value> ::=
  <signed numeric literal>

<sequence generator min-value option> ::=
    MINVALUE <sequence generator min value>
  | NO MINVALUE

<sequence generator min value> ::=
  <signed numeric literal>

<sequence generator cycle option> ::=
    CYCLE
```

```
| NO CYCLE
```

Syntax Rules

1) Let *SEQ* be the sequence generator defined by the <sequence generator definition> *SEQD*.

2) If *SEQD* is contained in a <schema definition> *SD* and the <sequence generator name> *SQN* contains a <schema name>, then that <schema name> shall be equivalent to the implicit or explicit <schema name> of *SD*.

3) The schema identified by the explicit or implicit <schema name> of *SQN* shall not include a sequence generator descriptor whose sequence generator name is equivalent to *SQN*.

4) If *SEQD* is contained in a <schema definition>, then let *A* be the explicit or implicit <authorization identifier> of the <schema definition>. Otherwise, let *A* be the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of *SQN*.

5) Each of <sequence generator data type option>, <sequence generator start with option>, <sequence generator increment by option>, <sequence generator max-value option>, <sequence generator min-value option>, and <sequence generator cycle option> shall be specified at most once.

6) If <sequence generator data type option> is specified, then <data type> shall be an exact numeric type *DT* with scale 0 (zero); otherwise, let *DT* be an implementation-defined (ID157) exact numeric type with scale 0 (zero).

7) The Syntax Rules of Subclause 9.36, "Creation of a sequence generator", are applied with <common sequence generator options> as *OPTIONS* and *DT* as *DATA TYPE*.

Access Rules

1) If a <sequence generator definition> is contained in an <SQL-client module definition>, then the enabled authorization identifiers shall include *A*.

General Rules

1) The General Rules of Subclause 9.36, "Creation of a sequence generator", are applied with <common sequence generator options> as *OPTIONS* and *DT* as *DATA TYPE*; let a sequence generator descriptor *SEQDS* be the *SEQGENDESC* returned from the application of those General Rules. The sequence generator name included in *SEQDS* is set to *SQN*.

2) A privilege descriptor is created that defines the USAGE privilege on *SEQ* to *A*. This privilege is grantable. The grantor for this privilege descriptor is set to the special grantor value "_SYSTEM".

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <sequence generator definition>.

11.73 <alter sequence generator statement>

Function

Change the definition of an external sequence generator.

Format

```
<alter sequence generator statement> ::=
  ALTER SEQUENCE <sequence generator name> <alter sequence generator options>

<alter sequence generator options> ::=
  <alter sequence generator option>...

<alter sequence generator option> ::=
    <alter sequence generator restart option>
  | <basic sequence generator option>

<alter sequence generator restart option> ::=
  RESTART [ WITH <sequence generator restart value> ]

<sequence generator restart value> ::=
  <signed numeric literal>
```

Syntax Rules

1) Let *SEQ* be the sequence generator descriptor identified by the <sequence generator name> *SQN*. Let *DT* be the data type of *SEQ*.

2) The schema identified by the explicit or implicit <schema name> of *SQN* shall include *SEQ*.

3) The scope of *SQN* is the <alter sequence generator statement>.

4) The Syntax Rules of Subclause 9.37, "Altering a sequence generator", are applied with <alter sequence generator options> as *OPTIONS* and *SEQ* as *SEQUENCE*.

Access Rules

1) The enabled authorization identifiers shall include the <authorization identifier> that owns the schema identified by the explicit or implicit <schema name>

General Rules

1) The General Rules of Subclause 9.37, "Altering a sequence generator", are applied with <alter sequence generator options> as *OPTIONS* and *SEQ* as *SEQUENCE*.

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain an <alter sequence generator statement>.

2) Without Feature T177, "Sequence generator support: simple restart option", in conforming SQL language an <alter sequence generator restart option> contained in an <alter sequence generator statement> shall contain a <sequence generator restart value>.

11.74 <drop sequence generator statement>

This Subclause is modified by Subclause 11.18, "<drop sequence generator statement>", in ISO/IEC 9075-16.

Function

Destroy an external sequence generator.

Format

```
<drop sequence generator statement> ::=
  DROP SEQUENCE <sequence generator name> <drop behavior>
```

Syntax Rules

1) Let *SEQ* be the sequence generator identified by the <sequence generator name> *SQN*.

2) The schema identified by the explicit or implicit <schema name> of *SQN* shall include the descriptor of *SEQ*.

3) If RESTRICT is specified, then *SEQ* shall not be referenced in any of the following:

   a) The SQL routine body of any routine descriptor.

   b) [16] The triggered action of any trigger descriptor.

   NOTE 648 — If CASCADE is specified, then such objects will be dropped implicitly by the <revoke statement> and/or explicitly by the SQL-schema manipulation statements specified in the General Rules of this Subclause.

4) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of the sequence generator identified by *SQN*.

Access Rules

1) The enabled authorization identifiers shall include *A*.

General Rules

1) The following <revoke statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

   ```
   REVOKE USAGE ON SEQUENCE SQN FROM A CASCADE
   ```

2) The descriptor of *SEQ* is destroyed.

Conformance Rules

1) Without Feature T176, "Sequence generator support", conforming SQL language shall not contain a <drop sequence generator statement>.

12 Access control

This Clause is modified by Clause 11, "Access control", in ISO/IEC 9075-4.
This Clause is modified by Clause 11, "Access control", in ISO/IEC 9075-9.
This Clause is modified by Clause 11, "Access control", in ISO/IEC 9075-13.
This Clause is modified by Clause 12, "Access control", in ISO/IEC 9075-16.

12.1 <grant statement>

This Subclause is modified by Subclause 11.1, "<grant statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 12.1, "<grant statement>", in ISO/IEC 9075-16.

Function

Define privileges and role authorizations.

Format

```
<grant statement> ::=
    <grant privilege statement>
  | <grant role statement>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) For every involved grantee *G* and for every domain *D1* with name *D1N* owned by *G*, if all of the following are true:

   a) The applicable privileges for *G* include the grantable REFERENCES privilege on every column referenced in the <search condition> *SC* included in a domain constraint descriptor included in the domain descriptor of *D1*.

   b) The applicable privileges for *G* include the grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *SC*.

   c) The applicable privileges for *G* include the grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *SC* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR* included in a domain constraint descriptor included in the domain descriptor of *D1*.

d) The applicable privileges for *G* include the grantable SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> contained in *SC*.

e) The applicable privileges for *G* include the grantable USAGE privilege on all domains, character sets, collations, and transliterations whose <domain name>s, <character set name>s, <collation name>s, and <transliteration name>s, respectively, are included in the domain descriptor of *D1*.

then for every privilege descriptor with action USAGE, a grantor of "_SYSTEM", object *D1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed with a current user identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT USAGE ON DOMAIN D1N
TO G WITH
GRANT OPTION
```

2) For every involved grantee *G* and for every collation *C1* with name *C1N* owned by *G*, if the applicable privileges for *G* include a grantable USAGE privilege for the character set name included in the collation descriptor of *C1* and a grantable USAGE privilege for the transliteration name, if any, included in the collation descriptor of *C1*, then for every privilege descriptor with action USAGE, a grantor of "_SYSTEM", object of *C1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed with a current user identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT USAGE ON
  COLLATION C1N
  TO G
  WITH GRANT OPTION
```

3) For every involved grantee *G* and for every transliteration *T1* with name *T1N* owned by *G*, if the applicable privileges for *G* contain a grantable USAGE privilege for every character set identified by a <character set specification> contained in the <transliteration definition> of *T1*, then for every privilege descriptor with action USAGE, a grantor of "_SYSTEM", object of *T1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT USAGE
    ON TRANSLATION T1N
    TO G
    WITH GRANT OPTION
```

4) For every table *T* specified by some involved privilege descriptor and for each view *V* with name *VN* owned by some involved grantee *G* such that *T* or some column *CT* of *T* is referenced in the original <query expression> *QE* of *V*, or *T* is a supertable of the scoped table of a <reference resolution> contained in *QE*, let *UT* be the set of underlying tables of *QE* and let *CU* be the set of columns of the tables in *UT*.

a) For every column *CV* of *V*:

i) Let *CUCV* be the set of columns in *CU* that are underlying columns of *CV*.

ii) Let *CVN* be the column name of *CV*.

iii) If, following successful execution of the <grant statement>, all of the following are true:

1) The applicable privileges for *G* include grantable SELECT privileges on all of the columns $CRT_{i,j}$.

2) The applicable privileges for *G* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

3) The applicable privileges for *G* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

4) The applicable privileges for *G* include grantable SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

then the following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT SELECT (CVN)
    ON VN
    TO G
    WITH GRANT OPTION
```

iv) If, following successful execution of the <grant statement>, the applicable privileges for *G* will include REFERENCES privilege on $CRT_{i,j}$ for all *i* and for all *j*, and will include a REFERENCES privilege on some column of RT_i for all *i*, then:

1) Case:

A) If all of the following are true, then let *WGO* be "WITH GRANT OPTION".

I) The applicable privileges for *G* will include grantable REFERENCES privilege on $CRT_{i,j}$ for all *i* and for all *j*, and will include a grantable REFERENCES privilege on some column of RT_i for all *i*.

II) The applicable privileges for *G* include grantable EXECUTE privileges on all SQL-invoked routines that are subject routines of <routine invocation>s contained in *QE*.

III) The applicable privileges for *G* include grantable SELECT privilege on every table *T1* and every method *M* such that there is a <method reference>. *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the method identified by the <method name> of *MR*.

IV) The applicable privileges for *G* include grantable SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> that is contained in *QE*.

B) Otherwise, let *WGO* be the zero-length character string.

2) The following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT REFERENCES (CVN)
    ON VN
    TO G
    WGO
```

b) If, following successful execution of the <grant statement>, the applicable privileges for *G* include grantable SELECT privilege on every column of *V*, then the following <grant statement>

is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT SELECT
    ON VN
    TO G
    WITH GRANT OPTION
```

c) Following successful execution of the <grant statement>, if the applicable privileges for *G* include REFERENCES privilege on every column of *V*, then

Case:

i) If the applicable privileges for *G* include grantable REFERENCES privilege on every column of *V*, then let *WGO* be "WITH GRANT OPTION".

ii) Otherwise, let *WGO* be the zero-length character string.

The following <grant statement> is effectively executed as though the current user identifier were "_SYSTEM" and without further Access Rule checking:

```
GRANT REFERENCES
    ON VN
    TO G
    WGO
```

5) Following the successful execution of the <grant statement>, for every table *T* specified by some involved privilege descriptor and for every effectively updatable view *V* owned by some involved grantee *G* such that *T* is some target leaf underlying table of the original <query expression> of *V*, the General Rules of Subclause 9.39, "Determination of view privileges", are applied with *V* as *VIEW*.

6) For every involved grantee *G* and for every referenceable view *V*, named *VN*, owned by *G*, if following the successful execution of the <grant statement>, the applicable privileges for *G* include grantable UNDER privilege on the direct supertable of *V*, then the following <grant statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT UNDER
    ON VN
    TO G
    WITH GRANT OPTION
```

7) [16] For every involved grantee *G* and for every schema-level SQL-invoked routine *R1* having <specific name> *R1SN* owned by *G*, if the applicable privileges for *G* contain all of the privileges necessary to successfully execute every <SQL procedure statement> contained in the <routine body> of *R1*, and those privileges are grantable, then for every privilege descriptor with action EXECUTE, a grantor of "_SYSTEM", object of *R1*, and grantee *G* that is not grantable, the following <grant statement> is effectively executed with a current authorization identifier of "_SYSTEM" and without further Access Rule checking:

```
GRANT EXECUTE
    ON SPECIFIC ROUTINE R1SN
    TO G
    WITH GRANT OPTION
```

NOTE 649 — The privileges necessary include the EXECUTE privilege on every subject routine of every <routine invocation> contained in the <SQL procedure statement>.

8) If two privilege descriptors are identical except that one indicates that the privilege is grantable and the other indicates that the privilege is not grantable, then both privilege descriptors are set to indicate that the privilege is grantable.

9) If two privilege descriptors are identical except that one indicates WITH HIERARCHY OPTION and the other does not, then both privilege descriptors are set to indicate that the privilege has the WITH HIERARCHY OPTION.

10) Redundant duplicate privilege descriptors are removed from the collection of all privilege descriptors.

11) 04 Redundant duplicate view privilege dependency descriptors are removed from the collection of all view privilege dependency descriptors.

Conformance Rules

None.

12.2 <grant privilege statement>

This Subclause is modified by Subclause 11.1, "<grant privilege statement>", in ISO/IEC 9075-13.

Function

Define privileges.

Format

```
<grant privilege statement> ::=
  GRANT <privileges> TO <grantee> [ { <comma> <grantee> }... ]
      [ WITH HIERARCHY OPTION ]
      [ WITH GRANT OPTION ]
      [ GRANTED BY <grantor> ]
```

Syntax Rules

1) Let *O* be the object identified by the <object name> contained in <privileges>.

2) The schema identified by the explicit or implicit <schema name> of the <object name> shall include the descriptor of *O*.

3) If WITH HIERARCHY OPTION is specified, then:

   a) <privileges> shall specify an <action> of SELECT without a <privilege column list> and without a <privilege method list>.

   b) *O* shall be a table of a structured type.

4) Case:

   a) If GRANTED BY is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

5) Syntax Rules of Subclause 12.8, "Grantor determination", are applied with *G* as *GRANTOR*; let *A* be the *AUTHORIZATION IDENTIFIER* returned from the application of those Syntax Rules.

Access Rules

1) The applicable privileges for *A* shall include a privilege identifying *O* or, if *O* is a table, a column of *O* or a table/method pair whose table is *O*.

2) If <privileges> contains a <privilege column list> *PCL*, then for every <column name> *CN* contained in *PCL*, the applicable privileges for *A* shall include a column privilege whose object is the column identified by *CN*.

3) If <privileges> contains a <privilege method list> *PML*, then for every <specific routine designator> *SRD* contained in *PML*, the applicable privileges for *A* shall include a privilege whose object is the method identified by *SRD*.

General Rules

1) Let *SPDK* be the set of privilege descriptor kernels specified by <privileges>. Let *N* be the number of privilege descriptor kernels in *SPDK*. Let PDK_i, 1 (one) ≤ *i* ≤ *N*, be an enumeration of the privilege descriptor kernels in *SPDK*. For all *i*, 1 (one) ≤ *i* ≤ *NPDK*, let PA_i be the action of PDK_i and let PO_i be the object of PDK_i.

2) A set of privilege descriptors is identified. The privilege descriptors identified are those defining, for each *i*, 1 (one) ≤ *i* ≤ *NPDK*, that action PA_i and object PO_i is held by *A* with grant option.

NOTE 650 — It is possible that some privilege descriptor kernels will not have a corresponding identified privilege descriptor. For such privilege descriptor kernels, no privilege descriptor is created.

3) For every identified privilege descriptor *IPD*, a privilege descriptor is created for each <grantee>, that specifies grantee <grantee>, action the same as the action of *IPD*, object the same as the object of *IPD*, and grantor *A*. Let *CPD* be the set of privilege descriptors created.

4) For every table privilege descriptor in *CPD* whose action is INSERT, UPDATE, or REFERENCES, column privilege descriptors are also created and added to *CPD* for each column *C* in *O* for which *A* holds the corresponding privilege with grant option. For each such column, a column privilege descriptor is created that specifies the identical <grantee>, the identical action, object *C*, and grantor *A*.

5) For every table privilege descriptor in *CPD* whose action is SELECT, column privilege descriptors are also created and added to *CPD* for each column *C* in *O* for which *A* holds the corresponding privilege with grant option. For each such column, a privilege descriptor is created that specifies the identical <grantee>, action SELECT, object *C*, and grantor *A*.

6) For every privilege descriptor in *CPD* whose action is SELECT, if the table *T* identified by the object of the privilege descriptor is a table of a structured type *TY*, then table/method privilege descriptors are also created and added to *CPD* for each method *M* of *TY* for which *A* holds the EXECUTE privilege with grant option. For each such method, a table/method privilege descriptor is created that specifies the identical <grantee>, action SELECT, object consisting of the pair of table *T* and method *M*, and grantor *A*.

7) If WITH GRANT OPTION was specified, then each created privilege descriptor also indicates that the privilege is grantable.

8) Let *SWH* be the set of privilege descriptors in *CPD* whose action is SELECT and that indicates WITH HIERARCHY OPTION. Let *ST* be the set of subtables of *O*. For every table *T* in *ST* and for every privilege descriptor in *SWH*, with grantee *G*, and grantor *A*,

Case:

a) If the privilege is grantable, then let *WGO* be "WITH GRANT OPTION".

b) Otherwise, let *WGO* be the zero-length character string.

Let *TN* be the table name of *T*. The following <grant statement> is effectively executed without further Access Rule checking:

```
GRANT SELECT
    ON TN
    TO G
    WGO
    GRANTED BY A
```

9) For every *i*, 1 (one) ≤ *i* ≤ *NPDK*, if there is no privilege descriptor in *CPD* corresponding to the privilege descriptor kernel PDK_i, then a completion condition is raised: *warning — privilege not granted (01007)*.

10) If ALL PRIVILEGES was specified, then for each grantee *G*, if there is no privilege descriptor in *CPD* specifying grantee *G*, then a completion condition is raised: *warning — privilege not granted (01007)*.

11) The *set of involved privilege descriptors* is defined to be *CPD*.

12) The *set of involved grantees* is defined as the set of specified <grantee>s.

Conformance Rules

1) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <specific routine designator> contained in a <grant privilege statement> that identifies a method.

2) 13 Without Feature S081, "Subtables", conforming SQL language shall not contain a <grant privilege statement> that contains WITH HIERARCHY OPTION.

12.3 <privileges>

This Subclause is modified by Subclause 11.2, "<privileges>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.1, "<privileges>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 11.2, "<privileges>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.2, "<privileges>", in ISO/IEC 9075-16.

Function

Specify privileges.

Format

```
<privileges> ::=
  <object privileges> ON <object name>

04 | 09 | 13 | 16 <object name> ::=
    [ TABLE ] <table name>
  | DOMAIN <domain name>
  | COLLATION <collation name>
  | CHARACTER SET <character set name>
  | TRANSLATION <transliteration name>
  | TYPE <schema-resolved user-defined type name>
  | SEQUENCE <sequence generator name>
  | <specific routine designator>

<object privileges> ::=
    ALL PRIVILEGES
  | <action> [ { <comma> <action> }... ]

<action> ::=
    SELECT
  | SELECT <left paren> <privilege column list> <right paren>
  | SELECT <left paren> <privilege method list> <right paren>
  | DELETE
  | INSERT [ <left paren> <privilege column list> <right paren> ]
  | UPDATE [ <left paren> <privilege column list> <right paren> ]
  | REFERENCES [ <left paren> <privilege column list> <right paren> ]
  | USAGE
  | TRIGGER
  | UNDER
  | EXECUTE

<privilege method list> ::=
  <specific routine designator> [ { <comma> <specific routine designator> }... ]

<privilege column list> ::=
  <column name list>

<grantee> ::=
    PUBLIC
  | <authorization identifier>

<grantor> ::=
    CURRENT_USER
  | CURRENT_ROLE
```

Syntax Rules

1) Let *P* be the <privileges>, let *GOR* be the <grant privilege statement> or <revoke statement> that simply contains *P*, and let *A* be the grantor of *GOR*.

2) If the <object name> *ON* simply contained in *GOR* specifies <table name> *TN*, then let *T* be the table identified by *TN*. *T* shall not be a declared local temporary table.

3) If *T* is a temporary table, then <object privileges> shall specify ALL PRIVILEGES.

4) If <action> *AC* is specified, then

   Case:

   a) 09 13 If *ON* specifies a <domain name>, <collation name>, <character set name>, <transliteration name>, <schema-resolved user-defined type name>, or <sequence generator name>, then *AC* shall specify USAGE.

   b) 16 If *T* is a base table or a viewed table, then *AC* shall specify SELECT, DELETE, INSERT, UPDATE, REFERENCES, or TRIGGER.

   c) If *ON* specifies a <schema-resolved user-defined type name> that identifies a structured type or specifies a <table name>, then *AC* shall specify UNDER.

   d) 04 If the object identified by *ON* is an SQL-invoked routine, then *AC* shall specify EXECUTE.

5) Each <column name> in a <privilege column list> shall identify a column of *T*.

6) If <privilege method list> is specified, then *ON* shall specify a <table name> that identifies a table of a structured type *TY* and each <specific routine designator> in the <privilege method list> shall identify a method of *TY*.

7) Let *O* be the object identified by *ON*.

8) *P* specifies a set *SPDK* of one or more privilege descriptor kernels, as follows.

   Case:

   a) ALL PRIVILEGES specifies the set union of the following sets of privilege descriptor kernels:

      i) The set of all privilege descriptor kernels whose object is *O* and whose action is one of the actions on *O* for which *A* has grantable privilege descriptors.

      ii) If *O* is a table, then all privilege descriptor kernels whose object *O2* is a column of *O* and whose action is one of the actions on *O2* for which *A* has grantable column privilege descriptors.

      iii) If *O* is a table, then all privilege descriptor kernels whose object *O3* is a table/method pair in which the table is *O* and whose action is one of the actions on *O3* for which *A* has grantable column privilege descriptors.

   b) Otherwise:

      i) UPDATE (<privilege column list>) is equivalent to the specification of UPDATE (<column name>) for each <column name> in <privilege column list>. INSERT (<privilege column list>) is equivalent to the specification of INSERT (<column name>) for each <column name> in <privilege column list>. REFERENCES (<privilege column list>) is equivalent to the specification of REFERENCES (<column name>) for each <column name> in <privilege column list>. SELECT (<privilege column list>) is equivalent to the specification of SELECT (<column name>) for each <column name> in <privilege column list>.

SELECT (<privilege method list>) is equivalent to the specification of SELECT (<specific routine designator>) for each <specific routine designator> in <privilege method list>.

ii) After making the preceding transformations, let *N* be the number of <action>s and let AC_1, ..., AC_N be an enumeration of the <action>s. For each *i*, 1 (one) ≤ *i* ≤ *N*, a privilege descriptor kernel PDK_i is specified, with action PA_i and object PO_i, as follows.

Case:

1) If <action> is SELECT (<column name>), then PA_i is SELECT and PO_i is the column of *O* identified by the <column name>.

2) If <action> is SELECT (<specific routine designator>), then PA_i is SELECT and PO_i is the table/method pair whose table is *O* and whose method is identified by the <specific routine designator>.

3) If <action> is INSERT (<column name>), then PA_i is INSERT and PO_i is the column of *O* identified by the <column name>.

4) If <action> is UPDATE (<column name>), then PA_i is UPDATE and PO_i is the column of *O* identified by the <column name>.

5) Otherwise, PA_i is <action> and PO_i is *O*.

Access Rules

None.

General Rules

1) A <grantee> of PUBLIC denotes at all times a list of <grantee>s containing all of the <authorization identifier>s in the SQL-environment.

2) An <authorization identifier> *B* has the WITH ADMIN OPTION on a role if a role authorization descriptor identifies the role as granted to *B* WITH ADMIN OPTION or a role authorization descriptor identifies it as granted WITH ADMIN OPTION to another applicable role for *B*.

Conformance Rules

1) Without Feature T332, "Extended roles", conforming SQL language shall not contain a <grantor>.

2) Without Feature T217, "TRIGGER privilege", conforming SQL language shall not contain an <action> that contains TRIGGER.

3) Without Feature S081, "Subtables", conforming SQL language shall not contain a <privileges> that contains an <action> that contains UNDER and that contains an <object name> that contains a <table name>.

4) Without Feature S023, "Basic structured types", conforming SQL language shall not contain a <privileges> that contains an <action> that contains UNDER and that contains an <object name> that contains a <schema-resolved user-defined type name> that identifies a structured type.

5) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <privileges> that contains an <action> that contains USAGE and that contains an <object name> that contains a <schema-resolved user-defined type name> that identifies a structured type.

6) Without Feature T281, “SELECT privilege with column granularity”, in conforming SQL language, an <action> that contains SELECT shall not contain a <privilege column list>.

7) Without Feature F731, “INSERT column privileges”, in conforming SQL language, an <action> that contains INSERT shall not contain a <privilege column list>.

8) 04 13 16 Without Feature S024, “Enhanced structured types”, conforming SQL language shall not contain a <privilege method list>.

12.4 <role definition>

Function

Define a role.

Format

```
<role definition> ::=
  CREATE ROLE <role name> [ WITH ADMIN <grantor> ]
```

Syntax Rules

1) The specified <role name> shall not be equivalent to any other <authorization identifier> in the SQL-environment.

2) Case:

   a) If WITH ADMIN is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

3) Syntax Rules of Subclause 12.8, "Grantor determination", are applied with *G* as *GRANTOR*; let *A* be the *AUTHORIZATION IDENTIFIER* returned from the application of those Syntax Rules.

Access Rules

1) The privileges necessary to execute the <role definition> are implementation-defined (IW141).

General Rules

1) A role descriptor whose role name is <role name> is created in the SQL-environment.

2) A grantable role authorization descriptor is created whose role name is <role name>, whose grantor is "_SYSTEM", and whose grantee is *A*.

Conformance Rules

1) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <role definition>.

2) Without Feature T332, "Extended roles", conforming SQL language shall not contain a <role definition> that immediately contains WITH ADMIN.

12.5 <grant role statement>

Function

Define role authorizations.

Format

```
<grant role statement> ::=
  GRANT <role granted> [ { <comma> <role granted> }... ]
      TO <grantee> [ { <comma> <grantee> }... ]
      [ WITH ADMIN OPTION ]
      [ GRANTED BY <grantor> ]

<role granted> ::=
  <role name>
```

Syntax Rules

1) No role identified by a specified <grantee> shall be applicable for any role identified by a specified <role granted>.

   NOTE 651 — That is, no cycles of role authorizations are allowed.

2) Case:

   a) If <grantor> is omitted, then let *G* be OMITTED.

   b) Otherwise, let *G* be <grantor>.

3) Syntax Rules of Subclause 12.8, "Grantor determination", are applied with *G* as *GRANTOR*; let *A* be the *AUTHORIZATION IDENTIFIER* returned from the application of those Syntax Rules.

Access Rules

None.

General Rules

1) For each <role granted> *R*, if no grantable role authorization descriptor exists whose role name is *R* and whose grantee is *A* or an applicable role for *A*, then an exception condition is raised: *invalid role specification (0P000)*.

2) For each <grantee> *GEE*, for each <role granted> *R*, a role authorization descriptor is created with role name *R*, grantee *GEE*, and grantor *A*.

3) If WITH ADMIN OPTION is specified, then each role authorization descriptor is grantable.

4) If two role authorization descriptors are identical except that one is grantable and the other is not, then both role authorization descriptors are set to indicate that the role authorization is grantable.

5) Redundant duplicate role authorization descriptors are destroyed.

6) The *set of involved privilege descriptors* is the union of the sets of privilege descriptors corresponding to the applicable privileges for every <role granted> specified.

7) The *set of involved grantees* is the union of the set of <grantee>s and the set of <role name>s for which at least one of the <role name>s that is possibly specified as a <grantee> is applicable.

Conformance Rules

1) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <grant role statement>.

12.6 <drop role statement>

Function

Destroy a role.

Format

```
<drop role statement> ::=
  DROP ROLE <role name> <drop behavior>
```

Syntax Rules

1) Let *R* be the role identified by the specified <role name>.

2) Let *DB* be the <drop behavior>.

NOTE 652 — RESTRICT behavior is enforced in the General Rules by performing a <revoke role statement>, which will fail with an exception if there are any dependencies on the role.

Access Rules

1) There shall exist at least one grantable role authorization descriptor whose role name is *R* and whose grantee is an enabled authorization identifier.

General Rules

1) For every <authorization identifier> *A* identified by a role authorization descriptor as having been granted to *R*, the following <revoke role statement> is effectively executed without further Access Rule checking:

```
REVOKE R FROM A DB
```

2) The descriptor of *R* is destroyed.

Conformance Rules

1) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <drop role statement>.

12.7 <revoke statement>

This Subclause is modified by Subclause 11.3, "<revoke statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.2, "<revoke statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 11.3, "<revoke statement>", in ISO/IEC 9075-13.
This Subclause is modified by Subclause 12.3, "<revoke statement>", in ISO/IEC 9075-16.

Function

Destroy privileges and role authorizations.

Format

```
<revoke statement> ::=
    <revoke privilege statement>
  | <revoke role statement>

<revoke privilege statement> ::=
  REVOKE [ <revoke option extension> ] <privileges>
      FROM <grantee> [ { <comma> <grantee> }... ]
      [ GRANTED BY <grantor> ]
      <drop behavior>

<revoke option extension> ::=
    GRANT OPTION FOR
  | HIERARCHY OPTION FOR

<revoke role statement> ::=
  REVOKE [ ADMIN OPTION FOR ] <role revoked> [ { <comma> <role revoked> }... ]
      FROM <grantee> [ { <comma> <grantee> }... ]
      [ GRANTED BY <grantor> ]
      <drop behavior>

<role revoked> ::=
  <role name>
```

Syntax Rules

1) If <revoke privilege statement> is specified, then:

   a) Let *O* be the object identified by the <object name> contained in <privileges>. If *O* is a table *T*, then let *S* be the set of subtables of *O*. If *T* is a table of a structured type, then let *TY* be that type.

   b) If HIERARCHY OPTION FOR is specified, then the <privileges> shall specify an <action> of SELECT without a <privilege column list> and without a <privilege method list> and *O* shall be a table of a structured type.

   c) If *O* is a table, then SELECT without HIERARCHY OPTION FOR, a <privilege column list> or a <privilege method list> is equivalent to specifying both the SELECT table privilege and SELECT (<privilege column list>) for all columns of <table name>. If *T* is a table of a structured type *TY*, then SELECT also specifies SELECT (<privilege column list>) for all columns inherited from *T* in each of the subtables of *T*, and SELECT (<privilege method list>) for all methods of *TY* in each of the subtables of *T*.

   d) INSERT without a <privilege column list> is equivalent to specifying both the INSERT table privilege and INSERT (<privilege column list>) for all columns of *T*.

e) UPDATE without a <privilege column list> is equivalent to specifying both the UPDATE table privilege and UPDATE (<privilege column list>) for all columns of *T*, as well as UPDATE (<privilege column list>) for all columns inherited from *T* in each of the subtables of *T*.

f) REFERENCES without a <privilege column list> is equivalent to specifying both the REFERENCES table privilege and REFERENCES (<privilege column list>) for all columns of *T*, as well as REFERENCES (<privilege column list>) for all columns inherited from *T* in each of the subtables of *T*.

g) Let *SP* be the set of <privileges> implied by the preceding rules.

h) Let *NSP* be the number of <privileges> in *SP*. Let P_i, 1 (one) ≤ *i* ≤ *NSP*, be an enumeration of the <privileges> in *SP*. For all *i*, 1 (one) ≤ *i* ≤ *NSP*, let $SPDK_i$ be the set of privilege descriptor kernels specified by P_i. Let *USPDK* be the union of the sets $SPDK_i$ for *i*, 1 (one) ≤ *i* ≤ *NSP*.

2) Case:

a) If GRANTED BY is omitted, then let *G* be OMITTED.

b) Otherwise, let *G* be <grantor>.

3) Syntax Rules of Subclause 12.8, "Grantor determination", are applied with *G* as *GRANTOR*; let *A* be the *AUTHORIZATION IDENTIFIER* returned from the application of those Syntax Rules. *A* is the *grantor* of the <revoke statement>.

Access Rules

1) Case:

a) If the <revoke statement> is a <revoke privilege statement>, then:

i) The applicable privileges for *A* shall include a privilege identifying *O* or, if *O* is a table, a column of *O*, or a table/method pair whose table is *O*.

ii) If <privileges> contains a <privilege column list> *PCL*, then for every <column name> *CN* contained in *PCL*, the applicable privileges shall include a column privilege whose object is the column identified by *CN*.

iii) If <privileges> contains a <privilege method list> *PML*, then for every <specific routine designator> *SRD* contained in *PML*, the applicable privileges shall include a privilege whose object is the method identified by *SRD*.

b) If the <revoke statement> is a <revoke role statement>, then, for every role *R* identified by a <role revoked>, there shall exist a grantable role authorization descriptor whose role name is *R* and whose grantee is *A* or an applicable role of *A*.

General Rules

1) Case:

a) If the <revoke statement> is a <revoke privilege statement>, then, for every <grantee> specified, a set of privilege descriptors is identified.

i) Let *NPDK* be the number of privilege descriptor kernels in *USPDK*. Let PDK_i, 1 (one) ≤ *i* ≤ *NPDK*, be an enumeration of the privilege descriptor kernels in *USPDK*. For all *i*, 1 (one) ≤ *i* ≤ *NPDK*, let PA_i be the action of PDK_i and let PO_i be the object of PDK_i.

ii) A privilege descriptor *P* is said to be *identified* if it belongs to the set of privilege descriptors that define, for some *i*, 1 (one) ≤ *i* ≤ *NPDK*, action PA_i and object PO_i, granted by *A* to <grantee>.

b) If the <revoke statement> is a <revoke role statement>, then, for every <grantee> specified, a set of role authorization descriptors is identified. A role authorization descriptor is said to be *identified* if it defines the grant of at least one of the specified <role revoked>s to <grantee> with grantor *A*.

2) A privilege descriptor *D* is said to be *directly dependent* on another privilege descriptor *P* if

Case:

a) If *D* identifies a view or a column of a view, the grantor of *D* is the special grantor value "_SYSTEM", and the action of *D* is INSERT, UPDATE, or DELETE, then *D* is *directly dependent* on *P* if there exists a view privilege dependency descriptor whose supporting privilege descriptor is *P* and whose dependent privilege descriptor is *D*.

b) Otherwise, *D* is directly dependent on *P* if exactly one of the following is true:

i) All of the following are true:

1) *P* indicates that the privilege that it represents is grantable.

2) The grantee of *P* is the same as the grantor of *D*, or the grantee of *P* is PUBLIC, or, if the grantor of *D* is a <role name>, the grantee of *P* is an applicable role for the grantor of *D*.

3) Case:

A) *P* and *D* are both column privilege descriptors. The action and the identified column of *P* are the same as the action and identified column of *D*, respectively.

B) [16] *P* and *D* are both table privilege descriptors. The action and the identified table of *P* are the same as the action and the identified table of *D*, respectively.

C) *P* and *D* are both execute privilege descriptors. The action and the identified SQL-invoked routine of *P* are the same as the action and the identified SQL-invoked routine of *D*, respectively.

D) [13] *P* and *D* are both usage privilege descriptors. The action and the identified domain, character set, collation, transliteration, user-defined type, or sequence generator of *P* are the same as the action and the identified domain, character set, collation, transliteration, user-defined type, or sequence generator of *D*, respectively.

E) *P* and *D* are both under privilege descriptors. The action and the identified user-defined type or table of *P* are the same as the action and the identified user-defined type or table of *D*, respectively.

F) *P* and *D* are both table/method privilege descriptors. The action and the identified method and table of *P* are the same as the action and the identified method and table of *D*, respectively.

ii) All of the following are true:

1) The privilege descriptor for *D* indicates that its grantor is the special grantor value "_SYSTEM".

2) The action of *P* is the same as the action of *D*.

3) The grantee of *P* is the owner of the table, collation, or transliteration identified by *D* or the grantee of *P* is PUBLIC.

4) Exactly one of the following is true:

A) *P* and *D* are both column privilege descriptors, the privilege descriptor *D* identifies a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V*, and one of the following is true:

I) There exists a table *T* identified by a <table reference> contained in the original <query expression> of *V* and a column *CT* that is a column of *T* and an underlying column of *CV*, such that the action for *P* is REFERENCES and either the identified column of *P* is *CT* or the identified table of *P* is *T*.

II) There exists a table *T* identified by a <table reference> contained in the original <query expression> of *V* and a column *CT* that is a column of *T* and an underlying column of *CV*, such that the action for *P* is SELECT and either the identified column of *P* is *CT* or the identified table of *P* is *T*.

B) The privilege descriptor *D* identifies the <collation name> of a <collation definition> *CO* and the identified character set name of *P* is included in the collation descriptor for *CO*, or the identified transliteration name of *P* is included in the collation descriptor for *CO*.

C) The privilege descriptor *D* identifies the <transliteration name> of a <transliteration definition> *TD* and the identified character set name of *P* is contained in the <source character set specification> or the <target character set specification> immediately contained in *TD*.

iii) All of the following are true:

1) The privilege descriptor for *D* indicates that its grantor is the special grantor value "_SYSTEM".

2) The grantee of *P* is the owner of the domain identified by *D* or the grantee of *P* is PUBLIC.

3) The privilege descriptor *D* identifies the <domain name> of a <domain definition> *DO* and either the column privilege descriptor *P* has an action of REFERENCES and identifies a column referenced in the <search condition> included in the domain descriptor for *DO*, or the privilege descriptor *P* has an action of USAGE and identifies a domain, collation, character set, or transliteration whose <domain name>, <collation name>, <character set name>, or <transliteration name>, respectively, is contained in the <search condition> of the domain descriptor for *DO*.

3) The *privilege dependency graph* is a directed graph such that all of the following are true:

a) Each node represents a privilege descriptor.

b) Each arc from node *P1* to node *P2* represents the fact that *P2* directly depends on *P1*.

An *independent node* is a node that has no incoming arcs.

4) A privilege descriptor *P* is said to be *modified* if all of the following are true:

a) *P* indicates that the privilege that it represents is grantable.

b) *P* directly depends on an identified privilege descriptor or a modified privilege descriptor.

c) Case:

i) If *P* is a column privilege descriptor that includes a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V* with an action of neither SELECT nor a REFERENCES, then let *XO* and *XA* respectively be the identifier of the object identified by a privilege descriptor *X* and the action of *X*. Within the set of privilege descriptors upon which *P* directly depends, there exist some *XO* and *XA* for which the set of identified privilege descriptors unioned with the set of modified privilege descriptors include all privilege descriptors specifying the grant of *XA* on *XO* WITH GRANT OPTION.

ii) If *P* is a column privilege descriptor that identifies a column *CV* identified by a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V* with an action *PA* of REFERENCES or SELECT, then let *SP* be the set of privileges upon which *P* directly depends. For every table *T* identified by a <table reference> contained in the <query expression> of *V*, let *RT* be the <table name> of *T*. There exists a column *CT* whose <column name> is *CRT*, such that all of the following are true:

1) *CT* is a column of *T* and an underlying column of *CV*.

2) Every privilege descriptor *PD* that is the descriptor of some member of *SP* that specifies the action *PA* on *CRT* WITH GRANT OPTION is either an identified privilege descriptor for *CRT* or a modified privilege descriptor for *CRT*.

d) At least one of the following is true:

i) GRANT OPTION FOR is specified and the grantor of *P* is the special grantor value "_SYSTEM".

ii) There exists a path to *P* from an independent node that includes no identified or modified privilege descriptors. *P* is said to be a *marked modified privilege descriptor*.

iii) *P* directly depends on a marked modified privilege descriptor, and the grantor of *P* is the special grantor value "_SYSTEM". *P* is said to be a *marked modified privilege descriptor*.

5) A role authorization descriptor *D* is said to be *directly dependent* on another role authorization descriptor *RD* if all of the following are true:

a) *RD* indicates that the role that it represents is grantable.

b) The role name of *D* is the same as the role name of *RD*.

c) The grantee of *RD* is the same as the grantor of *D*, or the grantee of *RD* is PUBLIC, or, if the grantor of *D* is a <role name>, the grantee of *RD* is an applicable role for the grantor of *D*.

6) The *role dependency graph* is a directed graph such that all of the following are true:

a) Each node represents a role authorization descriptor.

b) Each arc from node *R1* to node *R2* represents the fact that *R2* directly depends on *R1*.

An independent node is one that has no incoming arcs.

7) A role authorization descriptor *RD* is said to be *abandoned* if *RD* is not an independent node, and is not itself an identified role authorization descriptor, and there exists no path to *RD* from any independent node other than paths that include an identified role authorization descriptor.

8) An arc from a node *P* to a node *D* of the privilege dependency graph is said to be *unsupported* if all of the following are true:

   a) The grantor of *D* and the grantee of *P* are both <role name>s.

   b) The destruction of all abandoned role authorization descriptors and, if ADMIN OPTION FOR is not specified, all identified role authorization descriptors would result in the grantee of *P* no longer being an applicable role for the grantor of *D*.

9) A privilege descriptor *P* is *abandoned* if at least one of the following is true:

   a) *P* is not an independent node, and is not itself an identified or a modified privilege descriptor, and there exists no path to *P* from any independent node other than paths that include an identified privilege descriptor or a modified privilege descriptor or an unsupported arc, and, if <revoke statement> specifies WITH HIERARCHY OPTION, then *P* has the WITH HIERARCHY OPTION.

   b) All of the following are true:

      i) *P* is a column privilege descriptor that includes a <column name> *CVN* explicitly or implicitly contained in the <view column list> of a <view definition> *V*, with an action *PA* of REFERENCES or SELECT.

      ii) Letting *SP* be the set of privileges upon which *P* directly depends, at least one of the following is true:

         1) There exists some table name *RT* such that all of the following are true:

            A) *RT* is the name of the table identified by some <table reference> contained in the original <query expression> of *V*.

            B) For every column privilege descriptor *CPD* that is the descriptor of some member of *SP* that specifies the action *PA* on *RT*, *CPD* is either an identified privilege descriptor for *RT* or an abandoned privilege descriptor for *RT*.

         2) There exists some column name *CRT* such that all of the following are true:

            A) *CRT* is the name of some column of the table identified by some <table reference> contained in the original <query expression> of *V*.

            B) For every column privilege descriptor *CPD* that is the descriptor of some member of *SP* that specifies the action *PA* on *CRT*, *CPD* is either an identified privilege descriptor for *CRT* or an abandoned privilege descriptor for *CRT*.

10) The *revoke destruction action* is defined as

    Case:

    a) If the <revoke statement> is a <revoke privilege statement>, then

       Case:

       i) If the <revoke statement> specifies the WITH HIERARCHY OPTION, then the removal of the WITH HIERARCHY OPTION from all identified and abandoned privilege descriptors.

       ii) Otherwise, the destruction of all abandoned privilege descriptors and, if GRANT OPTION FOR is not specified, all identified privilege descriptors.

    b) If the <revoke statement> is a <revoke role statement>, then the destruction of all abandoned role authorization descriptors, all abandoned privilege descriptors and, if ADMIN OPTION FOR is not specified, all identified role authorization descriptors.

11) Let *REPO* be a <search condition>, generation expression, or <SQL routine body>. The *required execute privileges* of *REPO* are defined as:

a) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <routine invocation> contained in *REPO*.

b) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <method invocation> contained in *REPO*.

c) EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <static method invocation> contained in *REPO*.

d) 04 EXECUTE privilege on every SQL-invoked routine that is the subject routine of any <method reference> contained in *REPO*.

12) 09 Let *S1* be, in turn, the name of every schema and *A1* be the <authorization identifier> that owns the schema identified by *S1*.

a) For every view descriptor *V* included in *S1*:

i) Let *QE* be the original <query expression> of *V*. *V* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

1) SELECT privilege on at least one column of every table identified by a <table reference> contained in *QE*.

2) SELECT privilege on every column identified by a <column reference> contained in *QE*.

3) USAGE privilege on every domain, every collation, every character set, and every transliteration whose names are contained in *QE*.

4) USAGE privilege on every user-defined type *UDT* such that some data type contained in *V* is usage-dependent on *UDT*.

5) The required execute privileges of *QE*.

6) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in *QE* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is subject routine of *MR*.

7) SELECT privilege on every column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in *QE*.

8) SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of each <reference resolution> that is contained in *QE*.

9) SELECT privilege on every scoped table that is the scoped table of each <reference resolution> that is contained in *QE*.

10) If *V* is the descriptor of a referenceable table, then USAGE privilege on the structured type associated with the view described by *V*.

11) UNDER privilege on every direct supertable of the view described by *V*.

12) 16 SELECT WITH HIERARCHY OPTION privilege on at least one supertable of every typed table identified by each <table reference> that simply contains an <only spec> and that is contained in *QE*.

b) 09 16 For every table descriptor *T* included in *S1*:

i) *T* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having any of the following:

1) If *T* is the descriptor of a referenceable table, then USAGE privilege on the structured type associated with the table described by *T*.

2) UNDER privilege on every direct supertable of the table described by *T*.

c) For every table constraint descriptor *TC* included in *S1*:

i) *TC* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

1) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in the applicable <search condition> of *TC*.

2) REFERENCES privilege on every column identified by a <column reference> contained in the applicable <search condition> of *TC*.

3) USAGE privilege on every domain, every collation, every character set, and every transliteration whose names are contained in the applicable <search condition> of *TC*.

4) USAGE privilege on every user-defined type *UDT* such that some data type contained in the applicable <search condition> of *TC* is usage-dependent on *UDT*.

5) The required execute privileges of the applicable <search condition> of *TC*.

6) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in the applicable <search condition> of *TC* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

7) SELECT privilege on every column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in the applicable <search condition>s of *TC*.

8) SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of each <reference resolution> that is contained in the applicable <search condition>s of *TC*.

9) SELECT privilege on the scoped table of each <reference resolution> that is contained in the applicable <search condition>s of *TC*.

10) [16] SELECT WITH HIERARCHY OPTION privilege on at least one supertable of each typed table identified by every <table reference> that simply contains an <only spec> and that is contained in the applicable <search condition> of *TC*.

d) For each assertion descriptor *AX* included in *S1*:

i) *AX* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

1) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in the applicable <search condition> of *AX*.

2) REFERENCES privilege on every column identified by a <column reference> contained in the applicable <search condition> of *AX*.

3) USAGE privilege on every domain, every collation, every character set, and every transliteration whose names are contained in the applicable <search condition>s of *AX*.

4) USAGE privilege on every user-defined type *UDT* such that some data type contained in the applicable <search condition>s of *AX* is usage-dependent on *UDT*.

5) The required execute privileges of the applicable <search condition>s of *AX*.

6) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in the applicable <search condition>s of *AX* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

7) SELECT privilege on every column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in the applicable <search condition>s of *AX*.

8) SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of each <reference resolution> that is contained in the applicable <search condition>s of *AX*.

9) SELECT privilege on the scoped table of every <reference resolution> that is contained in the applicable <search condition>s of *AX*.

10) [16]SELECT WITH HIERARCHY OPTION privilege on at least one supertable of every typed table identified by every <table reference> that simply contains an <only spec> and that is contained in the applicable <search condition>s of *AX*.

e) For every trigger descriptor *TR* included in *S1*:

i) *TR* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

1) TRIGGER privilege on the subject table of *TR*.

2) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in the <search condition>s of *TR*.

3) SELECT privilege on every column identified by a <column reference> contained in the <search condition>s of *TR*.

4) USAGE privilege on every domain, collation, character set, and transliteration whose name is contained in the <search condition>s of *TR*.

5) USAGE privilege on every user-defined type *UDT* such that some data type contained in the <search condition>s of *TR* is usage-dependent on *UDT*.

6) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in any <search condition> of *TR* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

7) The required execute privileges of the <search condition>s of *TR*.

8) The required execute privileges of the <triggered SQL statement> of *TR*.

9) SELECT privilege on at least one column of every table identified by a <table reference> contained in a <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

10) SELECT privilege on at least one column of every table identified by a <table reference> contained in a <table expression> or <select list> immediately contained

in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

11) SELECT privilege on at least one column of every table identified by a <table reference> and <column reference> contained in a <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

12) SELECT privilege on at least one column of every table identified by a <table reference> and <column reference> contained in a <value expression> simply contained in an <update source> or an <assigned row>contained in the <triggered SQL statement> of *TR*.

13) INSERT privilege on every column

    Case:

    A) Identified by a <column name> contained in the <insert column list> of an <insert statement> or a <merge statement> contained in the <triggered SQL statement> of *TR*.

    B) Of the table identified by the <table name> immediately contained in an <insert statement> that does not contain an <insert column list> and that is contained in the <triggered SQL statement> of *TR*.

    C) Of the table identified by the <target table> contained in a <merge statement> that contains a <merge insert specification> and that does not contain an <insert column list> and that is contained in the <triggered SQL statement> of *TR*.

14) UPDATE privilege on every column identified by a <column name> is contained in an <object column> contained in either an <update statement: positioned>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

15) DELETE privilege on every table identified by a <table name> contained in either a <delete statement: positioned> or a <delete statement: searched> contained in the <triggered SQL statement> of *TR*.

16) DELETE privilege on the table identified by the <target table> contained in a <merge statement> that contains a <merge delete specification> and that is contained in the <triggered SQL statement> of *TR*.

17) USAGE privilege on every domain, collation, character set, transliteration, and sequence generator whose name is contained in the <triggered SQL statement> of *TR*.

18) USAGE privilege on every user-defined type *UDT* such that some data type contained in the <triggered SQL statement> of *TR* is usage-dependent on *UDT*.

19) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in each <triggered SQL statement> of *TR* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

20) SELECT privilege on any column identified by a <column reference> contained in the <scalar subquery> that is equivalent to some <dereference operation> contained in any of the following:

    A) A <search condition> of *TR*.

B) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

C) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

D) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

E) A <value expression> contained in an <update source> or an <assigned row> contained in the <triggered SQL statement> of *TR*.

21) SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> that is contained in any of the following:

A) A <search condition> of *TR*.

B) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

C) A <table expression> or <select list> immediately contained in every <select statement: single row> contained in the <triggered SQL statement> of *TR*.

D) A <search condition> contained in every <delete statement: searched>, every <update statement: searched>, or every <merge statement> contained in the <triggered SQL statement> of *TR*.

E) Every <value expression> contained in an <update source> or an <assigned row> contained in the <triggered SQL statement> of *TR*.

22) SELECT privilege on every scoped table that is the scoped table of each <reference resolution> contained in any of the following:

A) A <search condition> of *TR*.

B) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

C) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <triggered SQL statement> of *TR*.

D) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <triggered SQL statement> of *TR*.

E) A <value expression> contained in an <update source> or an <assigned row> contained in the <triggered SQL statement> of *TR*.

23) 04 16 SELECT WITH HIERARCHY OPTION privilege on at least one supertable of every typed table identified by every <table reference> that simply contains an <only spec> and that is contained in the <triggered SQL statement> of *TR*.

24) SELECT privilege on each column identified by a <column reference> contained in a <value expression> simply contained in an <SQL control statement> contained in the <triggered SQL statement> of *TR*.

f) For every domain constraint descriptor *DC* included in *S1*.

i) *DC* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

1) REFERENCES privilege on at least one column of every table identified by a <table reference> contained in any <search condition> of *DC*.

2) REFERENCES privilege on every column identified by a <column reference> contained in the <search condition> of *DC*.

3) USAGE privilege on every domain, every user-defined type, every collation, every character set, and every transliteration whose names are contained in any <search condition> of *DC*.

4) USAGE privilege on each user-defined type *UDT* such that some data type contained in the <search condition>s of *DC* is usage-dependent on *UDT*.

5) The required execute privileges of the <search condition>s of *DC*.

6) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in the <search condition>s of *DC* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

7) SELECT privilege on every column identified by a <column reference> contained in a <scalar subquery> that is equivalent to some <dereference operation> contained in the <search condition>s of *DC*.

8) SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of every <reference resolution> that is contained in the <search condition>s of *DC*.

9) SELECT privilege on every scoped table that is the scoped table of every <reference resolution> that is contained in the <search condition>s of *DC*.

10) [16] SELECT WITH HIERARCHY OPTION privilege on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in a <search condition> of *DC*.

g) For every domain descriptor *DO* included in *S1*, *DO* is said to be *lost* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on every character set included in the data type descriptor included in *DO*.

h) For every table descriptor *TD* contained in *S1*, for every column descriptor *CD* included in *TD*, *CD* is said to be *lost* if any of the following are true:

i) The revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any character set included in the data type descriptor included in *CD*.

ii) The revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any user-defined type *UDT* such that a data type descriptor included in *CD* describes a type that is usage-dependent on *UDT*.

iii) The name of the domain *DN* included in *CD*, if any, identifies a lost domain descriptor and the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any character set included in the data type descriptor of the domain descriptor of *DN*.

iv) *CD* has a generation expression *GE* and the revoke destruction action would result in *A1* no longer having in its applicable privileges the required execute privileges of *GE*.

i) For every SQL-client module *MO*, let *G* be the <module authorization identifier> that owns *MO*. *MO* is said to be *lost* if the revoke destruction action would result in *G* no longer having in its applicable privileges USAGE privilege on the character set referenced in the <module character set specification> of *MO*.

j) For every user-defined type descriptor *DT* included in *S1*, *DT* is said to be *abandoned* if any of the following are true:

i) The revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on any user-defined type *UDT* such that a data type descriptor included in *DT* describes a type that is usage-dependent on *UDT*.

ii) 13 The revoke destruction action would result in *A1* no longer having in its applicable privileges the UNDER privilege on any user-defined type that is a direct supertype of *DT*.

k) *S1* is said to be *lost* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the default character set included in *S1*.

l) For every collation descriptor *CN* contained in *S1*, *CN* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the collation whose name is contained in the <existing collation name> of *CN*.

m) For every character set descriptor *CSD* contained in *S1*, *CSD* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the collation whose name is contained in *CSD*.

n) 13 For every descriptor included in *S1* that includes a data type descriptor *DTD*, *DTD* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on the collation whose name is included in *DTD*.

o) For every routine descriptor *RD* with an SQL security characteristic of DEFINER that is included in *S1*:

i) *RD* is said to be *abandoned* if the revoke destruction action would result in *A1* no longer having in its applicable privileges any of the following:

1) 04 The required execute privileges of the <SQL routine body> of *RD* or the <parameter default> of every SQL parameter of *RD*.

2) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <SQL routine body> of *RD*.

3) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <SQL routine body> of *RD*.

4) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

5) SELECT privilege on at least one column of each table identified by a <table reference> contained in a <value expression> simply contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

6) SELECT privilege on each column identified by a <column reference> contained in a <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

7) SELECT privilege on each column identified by a <column reference> contained in a <value expression> simply contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

8) INSERT privilege on each column

   Case:

   A) Identified by a <column name> contained in the <insert column list> of an <insert statement> or a <merge statement> contained in the <SQL routine body> of *RD*.

   B) Of the table identified by the <table name> immediately contained in an <insert statement> that does not contain an <insert column list> and that is contained in the <SQL routine body> of *RD*.

   C) Of the table identified by the <target table> immediately contained in a <merge statement> that contains a <merge insert specification> and that does not contain an <insert column list> and that is contained in the <SQL routine body> of *RD*.

9) UPDATE privilege on each column whose name is contained in an <object column> contained in either an <update statement: positioned>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

10) DELETE privilege on each table whose name is contained in a <table name> contained in either a <delete statement: positioned> or a <delete statement: searched> contained in the <SQL routine body> of *RD*.

11) DELETE privilege on the table identified by the <target table> contained in a <merge statement> that contains a <merge delete specification> and that is contained in the <SQL routine body> of *RD*.

12) USAGE privilege on each domain, collation, character set, transliteration, and sequence generator whose name is contained in the <SQL routine body> of *RD*.

13) USAGE privilege on each user-defined type *UDT* such that a declared type of any SQL parameter, returns data type, or result cast included in *RD* is usage-dependent on *UDT*.

14) USAGE privilege on each user-defined type *UDT* such that some data type contained in the <SQL routine body> of *RD* is usage-dependent on *UDT*.

15) The table/method privilege on every table *T1* and every method *M* such that there is a <method reference> *MR* contained in the <SQL routine body> of *RD* such that *T1* is in the scope of the <value expression primary> of *MR* and *M* is the subject routine of *MR*.

16) SELECT privilege on every column identified by a <column reference> contained in a <scalar subquery> that is equivalent to a <dereference operation> contained in any of the following:

   A) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <SQL routine body> of *RD*.

B) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <SQL routine body> of *RD*.

C) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

D) A <value expression> contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

17) SELECT WITH HIERARCHY OPTION privilege on at least one supertable of the scoped table of each <reference resolution> that is contained in any of the following:

A) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <SQL routine body> of *RD*.

B) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <SQL routine body> of *RD*.

C) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

D) A <value expression> simply contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

18) SELECT privilege on the scoped table of every <reference resolution> that is contained in any of the following:

A) A <query expression> simply contained in a <cursor specification>, an <insert statement>, or a <merge statement> contained in the <SQL routine body> of *RD*.

B) A <table expression> or <select list> immediately contained in a <select statement: single row> contained in the <SQL routine body> of *RD*.

C) A <search condition> contained in a <delete statement: searched>, an <update statement: searched>, or a <merge statement> contained in the <SQL routine body> of *RD*.

D) A <value expression> contained in an <update source> or an <assigned row> contained in the <SQL routine body> of *RD*.

19) 13 16 SELECT WITH HIERARCHY OPTION privilege on at least one supertable of every typed table identified by a <table reference> that simply contains an <only spec> and that is contained in the <SQL routine body> of *RD*.

20) SELECT privilege on each column identified by a <column reference> contained in a <value expression> simply contained in an <SQL control statement> contained in the <SQL routine body> of *RD*.

p) 09 For every table descriptor *TD* included in *S1*, for every column descriptor *CD* included in *TD*, *CD* is said to be *contaminated* if *CD* includes at least one of the following:

i) A user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

ii) A reference type descriptor that includes a user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

iii) A collection type descriptor that includes a user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

iv) A collection type descriptor that includes a reference type descriptor that includes a user-defined type descriptor that describes a supertype of a user-defined type described by an abandoned user-defined type descriptor.

q) 09 13 16 If RESTRICT is specified, and there exists an abandoned privilege descriptor, abandoned view, abandoned table constraint, abandoned assertion, abandoned domain constraint, lost domain, lost column, lost schema, or a descriptor that includes an impacted data type descriptor, impacted collation, impacted character set, abandoned user-defined type, or abandoned routine descriptor, then an exception condition is raised: *dependent privilege descriptors still exist (2B000)*.

r) 04 If CASCADE is specified, then the impact on an SQL-client module that is determined to be a lost module is implementation-defined (IA048).

s) Case:

i) If the <revoke statement> is a <revoke privilege statement>, then

Case:

1) If neither WITH HIERARCHY OPTION nor GRANT OPTION FOR is specified, then:

A) All abandoned privilege descriptors are destroyed.

B) The identified privilege descriptors are destroyed.

C) The modified privilege descriptors are set to indicate that they are not grantable.

2) If WITH HIERARCHY OPTION is specified, then the WITH HIERARCHY OPTION is removed from all identified and abandoned privilege descriptors, if present.

3) If GRANT OPTION FOR is specified, then

Case:

A) If CASCADE is specified, then all abandoned privilege descriptors are destroyed.

B) Otherwise, if there are any privilege descriptors directly dependent on an identified privilege descriptor that are not modified privilege descriptors, then an exception condition is raised: *dependent privilege descriptors still exist (2B000)*.

The identified privilege descriptors and the modified privilege descriptors are set to indicate that they are not grantable.

ii) If the <revoke statement> is a <revoke role statement>, then:

1) If CASCADE is specified, then all abandoned role authorization descriptors are destroyed.

2) All abandoned privilege descriptors are destroyed.

3) Case:

A) If ADMIN OPTION FOR is specified, then the identified role authorization descriptors are set to indicate that they are not grantable.

B) If ADMIN OPTION FOR is not specified, then the identified role authorization descriptors are destroyed.

t) For every abandoned view descriptor *V*, let *S1.VN* be the <table name> of *V*. The following <drop view statement> is effectively executed without further Access Rule checking:

```
DROP VIEW S1.VN CASCADE
```

u) For every abandoned table descriptor *T*, let *S1.TN* be the <table name> of *T*. The following <drop table statement> is effectively executed without further Access Rule checking:

```
DROP TABLE S1.TN CASCADE
```

v) For every abandoned table constraint descriptor *TC*, let *S1.TCN* be the <constraint name> of *TC* and let *S2.T2* be the <table name> of the table that contains *TC* (*S1* and *S2* possibly equivalent). The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE S2.T2
    DROP CONSTRAINT S1.TCN
    CASCADE
```

w) For every abandoned assertion descriptor *AX*, let *S1.AXN* be the <constraint name> of *AX*. The following <drop assertion statement> is effectively executed without further Access Rule checking:

```
DROP ASSERTION S1.AXN CASCADE
```

x) For every abandoned trigger descriptor *TR*, let *S1.TRN* be the <trigger name> of *TR*. The following <drop trigger statement> is effectively executed without further Access Rule checking:

```
DROP TRIGGER S1.TRN
```

y) For every abandoned domain constraint descriptor *DC*, let *S1.DCN* be the <constraint name> of *DC* and let *S2.DN* be the <domain name> of the domain that contains *DC*. The following <alter domain statement> is effectively executed without further Access Rule checking:

```
ALTER DOMAIN S2.DN
    DROP CONSTRAINT S1.DCN
```

z) For every lost column descriptor *CD*, let *S1.TN* be the <table name> of the table whose descriptor includes the descriptor *CD* and let *CN* be the <column name> of *CD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE S1.TN
    DROP COLUMN CN CASCADE
```

aa) For every lost domain descriptor *DO*, let *S1.DN* be the <domain name> of *DO*. The following <drop domain statement> is effectively executed without further Access Rule checking:

```
DROP DOMAIN S1.DN CASCADE
```

ab) For every lost schema *S1*, the default character set of that schema is modified to include the name of the implementation-defined (ID139) <character set specification> that would have

been this schema's default character set had the <schema definition> not specified a <schema character set specification>.

ac) If the object identified by *O* is a collation, let *OCN* be the name of that collation.

ad) For every descriptor that includes an impacted data type descriptor *DTD*, *DTD* is modified such that it does not include *OCN*.

ae) For every impacted collation descriptor *CD* with included collation name *CN*, the following <drop collation statement> is effectively executed without further Access Rule checking:

```
DROP COLLATION CN CASCADE
```

af) For every impacted character set descriptor *CSD* with included character set name *CSN*, *CSD* is modified so that the included collation name is the name of the default collation for the character set on which *CSD* is based.

ag) For every abandoned user-defined type descriptor *DT* with <user-defined type name> *S1.DTN*, the following <drop data type statement> is effectively executed without further Access Rule checking:

```
DROP TYPE S1.DTN CASCADE
```

ah) 04 13 16 For every abandoned SQL-invoked routine descriptor *RD*, let *R* be the SQL-invoked routine whose descriptor is *RD*. Let *SN* be the <specific name> of *R*. The following <drop routine statement> is effectively executed without further Access Rule checking:

```
DROP SPECIFIC ROUTINE SN CASCADE
```

ai) If the <revoke statement> is a <revoke privilege statement>, then:

i) For every combination of <grantee> and <action> on *O* specified in <privileges>, if there is no corresponding privilege descriptor in the set of identified privilege descriptors, then a completion condition is raised: *warning — privilege not revoked (01006)*.

ii) If ALL PRIVILEGES was specified, then for each <grantee>, if no privilege descriptors were identified, then a completion condition is raised: *warning — privilege not revoked (01006)*.

aj) For every contaminated column descriptor *CD*, let *S1.TN* be the <table name> of the table whose descriptor includes the descriptor *CD* and let *CN* be the <column name> of *CD*. The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE S1.TN
    DROP COLUMN CN CASCADE
```

Conformance Rules

1) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <revoke role statement>.

2) Without Feature F035, "REVOKE with CASCADE", conforming SQL language shall not contain a <revoke statement> that contains a <drop behavior> that contains CASCADE.

3) Without Feature F037, "REVOKE statement: GRANT OPTION FOR clause", conforming SQL language shall not contain a <revoke option extension> that contains GRANT OPTION FOR.

4) Without Feature F036, "REVOKE statement performed by non-owner", conforming SQL language shall not contain a <revoke statement> that contains a <privileges> that contains an <object name> where the owner of the SQL-schema that is specified explicitly or implicitly in the <object name> is not the current authorization identifier.

5) Without Feature F038, "REVOKE of a WITH GRANT OPTION privilege", conforming SQL language shall not contain a <revoke statement> such that there exists a privilege descriptor *PD* that satisfies all the following conditions:

   a) *PD* identifies the object identified by <object name> simply contained in <privileges> contained in the <revoke statement>.

   b) *PD* identifies the <grantee> identified by any <grantee> simply contained in <revoke statement> and that <grantee> does not identify the owner of the SQL-schema that is specified explicitly or implicitly in the <object name> simply contained in <privileges> contained in the <revoke statement>.

   c) *PD* identifies the action identified by the <action> simply contained in <privileges> contained in the <revoke statement>.

   d) *PD* indicates that the privilege is grantable.

6) [13] Without Feature S081, "Subtables", conforming SQL language shall not contain a <revoke option extension> that contains HIERARCHY OPTION FOR.

12.8 Grantor determination

Function

Determine the grantor of a privilege or role authorization, or the intended owner of a role.

Subclause Signature

```
"Grantor determination" [Syntax Rules] (
  Parameter: "GRANTOR"
) Returns: "AUTHORIZATION IDENTIFIER"
```

GRANTOR — a value indicating the mechanism to be used to determine the authorization identifier of one or more privileges (OMITTED, CURRENT_USER, CURRENT_ROLE, or an authorization identifier).

AUTHORIZATION IDENTIFIER — an authorization identifier representing the grantor of one or more privileges.

Syntax Rules

1) Let *G* be the *GRANTOR* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is returned as *AUTHORIZATION IDENTIFIER*.

2) The grantor *A* is derived from *G* as follows.

   Case:

   a) If *G* is OMITTED, then

      Case:

      i) If there is a current user identifier, then *A* is the current user identifier.

      ii) Otherwise, *A* is the current role name.

   b) If *G* is CURRENT_USER, then

      Case:

      i) There shall be a current user identifier.

      ii) Otherwise, *A* is the current user identifier.

   c) If *G* is CURRENT_ROLE, then

      Case:

      i) There shall be a current role name.

      ii) Otherwise, *A* is the current role name.

3) Evaluation of the Syntax Rules is terminated and control is returned to the invoking Subclause, which receives *A* as *AUTHORIZATION IDENTIFIER*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature T332, “Extended roles”, conforming SQL language shall contain no <grantor>.

13 SQL-client modules

This Clause is modified by Clause 12, "SQL-client modules", in ISO/IEC 9075-4.
This Clause is modified by Clause 12, "SQL-client modules", in ISO/IEC 9075-9.
This Clause is modified by Clause 13, "SQL-client modules", in ISO/IEC 9075-14.
This Clause is modified by Clause 12, "SQL-client modules", in ISO/IEC 9075-15.
This Clause is modified by Clause 13, "SQL-client modules", in ISO/IEC 9075-16.

13.1 <SQL-client module definition>

This Subclause is modified by Subclause 12.1, "<SQL-client module definition>", in ISO/IEC 9075-9.

Function

Define an SQL-client module.

Format

```
<SQL-client module definition> ::=
  <module name clause> <language clause> <module authorization clause>
      [ <module path specification> ]
      [ <module transform group specification> ]
      [ <module collations> ]
      [ <temporary table declaration>... ]
      <module contents>...

<module authorization clause> ::=
    SCHEMA <schema name>
  | AUTHORIZATION <module authorization identifier>
      [ FOR STATIC { ONLY | AND DYNAMIC } ]
  | SCHEMA <schema name> AUTHORIZATION <module authorization identifier>
      [ FOR STATIC { ONLY | AND DYNAMIC } ]

<module authorization identifier> ::=
  <authorization identifier>

<module path specification> ::=
  <path specification>

<module transform group specification> ::=
  <transform group specification>

<module collations> ::=
  <module collation specification>...

<module collation specification> ::=
  COLLATION <collation name> [ FOR <character set specification list> ]

<character set specification list> ::=
  <character set specification> [ { <comma> <character set specification> }... ]

<module contents> ::=
    <declare cursor>
  | <dynamic declare cursor>
  | <externally-invoked procedure>
```

Syntax Rules

1) The <language clause> shall not specify SQL.

2) If SCHEMA <schema name> is not specified, then a <schema name> equivalent to <module authorization identifier> is implicit.

3) If the explicit or implicit <schema name> does not specify a <catalog name>, then an implementation-defined (ID159) <catalog name> is implicit.

4) The implicit or explicit <catalog name> is the implicit <catalog name> for all unqualified <schema name>s in the <SQL-client module definition>.

5) If <module path specification> is not specified, then a <module path specification> containing an implementation-defined (ID160) <schema name list> that contains the <schema name> contained in <module authorization clause> is implicit.

6) The explicit or implicit <catalog name> of each <schema name> contained in the <schema name list> of the <module path specification> shall be equivalent to the <catalog name> of the explicit or implicit <schema name> contained in <module authorization clause>.

7) The <schema name list> of the explicit or implicit <module path specification> is used as the SQL-path of the <SQL-client module definition>. The SQL-path is used to effectively qualify unqualified <routine name>s that are immediately contained in <routine invocation>s that are contained in the <SQL-client module definition>.

8) Case:

   a) If <module transform group specification> is not specified, then a <module transform group specification> containing a <multiple group specification> with a <group specification> *GS* for each <host parameter declaration> contained in <host parameter declaration list> of each <externally-invoked procedure> contained in <SQL-client module definition> whose <host parameter data type> *UDT* identifies a user-defined type with no <locator indication> is implicit. The <group name> of *GS* is implementation-defined (ID177) and its <path-resolved user-defined type name> is *UDT*.

   b) If <module transform group specification> contains a <single group specification> with a <group name> *GN*, then a <module transform group specification> containing a <multiple group specification> that contains a <group specification> *GS* for each <host parameter declaration> contained in <host parameter declaration list> of each <externally-invoked procedure> contained in <SQL-client module definition> whose <host parameter data type> *UDT* identifies a user-defined type with no <locator indication> is implicit. The <group name> of *GS* is *GN* and its <path-resolved user-defined type name> is *UDT*.

   c) If <module transform group specification> contains a <multiple group specification> *MGS*, then a <module transform group specification> containing <multiple group specification> that contains *MGS* extended with a <group specification> *GS* for each <host parameter declaration> contained in <host parameter declaration list> of each <externally-invoked procedure> contained in <SQL-client module definition> whose <host parameter data type> *UDT* identifies a user-defined type with no <locator indication> and no equivalent of *UDT* is contained in any <group specification> contained in *MGS* is implicit. The <group name> of *GS* is implementation-defined (ID177) and its <path-resolved user-defined type name> is *UDT*.

9) No two <character set specification>s contained in any <module collation specification> shall be equivalent.

10) A <module collation specification> *MCS* specifies the SQL-client module collation for one or more character sets for the SQL-client module. Let *CO* be the collation identified by the <collation name> contained in *MCS*.

Case:

a) If <character set specification list> is specified, then the collation specified by *CO* shall be applicable to every character set identified by a <character set specification> simply contained in the <module collation specification>. For each character set specified, the SQL-client module collation for that character set is set to *CO*.

b) Otherwise, the character sets for which the SQL-client module collation is set to *CO* are implementation-defined (ID177).

11) A <declare cursor> shall precede in the text of the <SQL-client module definition> any <externally-invoked procedure> that references the <cursor name> of the <declare cursor>.

12) A <dynamic declare cursor> shall precede in the text of the <SQL-client module definition> any <externally-invoked procedure> that references the <cursor name> of the <dynamic declare cursor>.

13) If neither FOR STATIC ONLY nor FOR STATIC AND DYNAMIC is specified, then FOR STATIC AND DYNAMIC is implicit.

14) For every <declare cursor> in an <SQL-client module definition>, the <SQL-client module definition> shall contain exactly one <open statement> that specifies the <cursor name> declared in the <declare cursor>.

NOTE 653 — See the Syntax Rules of Subclause 14.1, "<declare cursor>".

15) Let *EIP1* and *EIP2* be two <externally-invoked procedure>s contained in an <SQL-client module definition> that have the same number of <host parameter declaration>s and immediately contain a <fetch statement> referencing the same <cursor name>. Let *n* be the number of <host parameter declaration>s. Let $P1_i$, 1 (one) ≤ i ≤ n, be the i-th <host parameter declaration> of *EIP1*. Let $DT1_i$ be the <data type> contained in $P1_i$. Let $P2_i$ be the i-th <host parameter declaration> of *EIP2*. Let $DT2_i$ be the <data type> contained in $P2_i$. For each i, 1 (one) ≤ i ≤ n,

Case:

a) If $DT1_i$ and $DT2_i$ both identify a binary large object type, then either $P1_i$ and $P2_i$ shall both be binary large object locator parameters or neither shall be binary large object locator parameters.

b) If $DT1_i$ and $DT2_i$ both identify a character large object type, then either $P1_i$ and $P2_i$ shall both be character large object locator parameters or neither shall be character large object locator parameters.

c) If $DT1_i$ and $DT2_i$ both identify a user-defined type, then either $P1_i$ and $P2_i$ shall both be user-defined type locator parameters or neither shall be user-defined type locator parameters.

Access Rules

None.

General Rules

1) If the SQL-agent that performs a call of an <externally-invoked procedure> in an <SQL-client module definition> is not a program that conforms to the specification for the programming language specified by the <language clause> of that <SQL-client module definition>, then the effect is implementation-dependent (UA059).

2) If the SQL-agent performs calls of <externally-invoked procedure>s from more than one Ada task, then the results are implementation-dependent (UV104).

3) If FOR STATIC ONLY is specified, then the SQL-client module includes an indication that prepared statements resulting from execution of externally-invoked procedures included in that module have no owner.

4) After the last time that an SQL-agent performs a call of an <externally-invoked procedure>:

   a) 09 A <rollback statement> or a <commit statement> is effectively executed. If an unrecoverable error has occurred, or if the SQL-agent terminated unexpectedly, or if any enforced constraint is not satisfied, then a <rollback statement> is performed. Otherwise, the choice of which of these SQL-statements to perform is implementation-dependent (UA060). If the SQL-implementation choice is <commit statement>, then all holdable cursors are first closed. The determination of whether an SQL-agent has terminated unexpectedly is implementation-dependent (UA061).

   b) For every SQL descriptor area that is currently allocated within an SQL-session associated with the SQL-agent, let *D* be the <descriptor name> of that SQL descriptor area; a <deallocate descriptor statement> that specifies

      ```
      DEALLOCATE DESCRIPTOR D
      ```

      is effectively executed.

   c) All SQL-sessions associated with the SQL-agent are terminated.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <module path specification>.

2) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <module transform group specification>.

3) Without Feature F693, "SQL-session and client module collations", conforming SQL language shall not contain a <module collation specification>.

4) Without Feature B051, "Enhanced execution rights", conforming SQL language shall not contain a <module authorization clause> that immediately contains FOR STATIC ONLY or FOR STATIC AND DYNAMIC.

5) Without Feature B111, "Module language Ada", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains ADA.

6) Without Feature B112, "Module language C", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains C.

7) Without Feature B113, "Module language COBOL ", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains COBOL.

8) Without Feature B114, "Module language Fortran", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains FORTRAN.

9) Without Feature B115, "Module language MUMPS ", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains M.

10) Without Feature B116, "Module language Pascal", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains PASCAL.

11) Without Feature B117, "Module language PL/I", conforming SQL language shall not contain an <SQL-client module definition> that contains a <language clause> that contains PLI.

13.2 <module name clause>

Function

Name an SQL-client module.

Format

```
<module name clause> ::=
  MODULE [ <SQL-client module name> ] [ <module character set specification> ]

<module character set specification> ::=
  NAMES ARE <character set specification>
```

Syntax Rules

1) If a <module name clause> does not specify an <SQL-client module name>, then the <SQL-client module definition> is unnamed.

2) The <SQL-client module name> shall not be equivalent to the <SQL-client module name> of any other <SQL-client module definition> in the same SQL-environment.

NOTE 654 — An SQL-environment can have multiple <SQL-client module definition>s that are unnamed.

3) If the <language clause> of the containing <SQL-client module definition> specifies ADA, then an <SQL-client module name> shall be specified, and that <SQL-client module name> shall be a valid Ada library unit name.

4) If a <module character set specification> is not specified, then a <module character set specification> that specifies an implementation-defined (ID178) character set that contains at least every character that is in <SQL language character> is implicit.

Access Rules

None.

General Rules

1) If <SQL-client module name> is specified, then, in the SQL-environment, the containing <SQL-client module definition> has the name given by <SQL-client module name>.

Conformance Rules

1) Without Feature F461, "Named character sets", conforming SQL language shall not contain a <module character set specification>.

13.3 <externally-invoked procedure>

This Subclause is modified by Subclause 12.1, "<externally-invoked procedure>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 12.2, "<externally-invoked procedure>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 13.1, "<externally-invoked procedure>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 12.1, "<externally-invoked procedure>", in ISO/IEC 9075-15.
This Subclause is modified by Subclause 13.1, "<externally-invoked procedure>", in ISO/IEC 9075-16.

Function

Define an externally-invoked procedure.

Format

```
<externally-invoked procedure> ::=
  PROCEDURE <procedure name> <host parameter declaration list> <semicolon>
      <SQL procedure statement> <semicolon>

<host parameter declaration list> ::=
  <left paren> <host parameter declaration>
      [ { <comma> <host parameter declaration> }... ] <right paren>

<host parameter declaration> ::=
    <host parameter name> <host parameter data type>
  | <status parameter>

<host parameter data type> ::=
  <data type> [ <locator indication> ]

<status parameter> ::=
  SQLSTATE
```

Syntax Rules

1) 09 The <procedure name> shall not be equivalent to the <procedure name> of any other <externally-invoked procedure> in the containing <SQL-client module definition>.

   NOTE 655 — The <procedure name> must be a standard-conforming procedure, function, or routine name of the language specified by the subject <language clause>. Failure to observe this recommendation will have implementation-dependent effects.

2) The <host parameter name> of each <host parameter declaration> in an <externally-invoked procedure> shall not be equivalent to the <host parameter name> of any other <host parameter declaration> in that <externally-invoked procedure>.

3) Any <host parameter name> contained in the <SQL procedure statement> of an <externally-invoked procedure> shall be specified in a <host parameter declaration> in that <externally-invoked procedure>.

4) If <locator indication> is simply contained in <host parameter declaration>, then:

   a) 15 The declared type *T* identified by the <data type> immediately contained in <host parameter data type> shall be one of:

      i) binary large object type;

      ii) character large object type;

      iii) array type;

iv) multiset type;

v) 15 user-defined type.

b) If *T* is a binary large object type, then the host parameter identified by <host parameter name> is called a *binary large object locator parameter*.

c) If *T* is a character large object type, then the host parameter identified by <host parameter name> is called a *character large object locator parameter*.

d) If *T* is an array type, then the host parameter identified by <host parameter name> is called an *array locator parameter*.

e) If *T* is a multiset type, then the host parameter identified by <host parameter name> is called a *multiset locator parameter*.

f) If *T* is a user-defined type, then the host parameter identified by <host parameter name> is called a *user-defined type locator parameter*.

5) Let *n* be the number of <host parameter declaration>s in the <externally-invoked procedure> *EP*.

6) A call of an <externally-invoked procedure> shall supply *n* arguments.

7) An <externally-invoked procedure> shall contain one <status parameter> referred to as an *SQLSTATE host parameter*. The SQLSTATE host parameter is referred to as a *status parameter*.

8) For each <host parameter declaration>, to determine whether the corresponding host parameter is an input host parameter, an output host parameter, or both an input host parameter and an output host parameter, the Syntax Rules of Subclause 9.8, "Host parameter mode determination", are applied with <host parameter declaration> as *HOST PARAM DECL* and <SQL procedure statement> as *SQL PROC STMT*; let *PMODE* be the *PARAMETER MODE* returned from the application of those Syntax Rules.

9) Let PD_i, 1 (one) $\le i \le n$, be the *i*-th <host parameter declaration>. Let PDT_i be the <data type> contained in PD_i. Let ADT_i be the data type of the argument corresponding to PD_i.

10) If the caller language of the <externally-invoked procedure> is ADA, then:

a) The SQL-implementation shall generate the source code of an Ada library unit package *ALUP*, the name of which shall be

Case:

i) If the <SQL-client module name> *SCMN* of the <SQL-client module definition> is a valid Ada identifier, then equivalent to *SCMN*.

ii) Otherwise, implementation-defined (IV225).

b) For each <externally-invoked procedure> of the <SQL-client module definition>, there shall appear within *ALUP* a subprogram declaration declaring a procedure.

i) If <procedure name> is a valid Ada identifier, then the name of that procedure *PN* shall be equivalent to <procedure name>; otherwise, *PN* shall be implementation-defined (IV225).

ii) The parameters in each Ada procedure declaration *APD* shall appear in the same order as the <host parameter declaration>s of the corresponding <externally-invoked procedure> *EIP*. If the names of the parameters declared in the <host parameter declaration>s of *EIP* are valid Ada identifiers, then the parameters in *APD* shall have parameter names that are equivalent to the names of the corresponding parameters declared in

the <host parameter declaration>s contained in *EIP*; otherwise, the parameters in *APD* shall have parameter names that are implementation-defined (IV225).

iii) The parameter modes and subtype marks used in the parameter specifications are constrained by the remaining paragraphs of this Subclause.

c) For each i, 1 (one) $\leq i \leq n$, if PD_i does not contain <locator indication>, then PDT_i shall not identify a data type listed in the "SQL data type" column of Table 21, "Data type correspondences for Ada", for which the corresponding row in the "Ada data type" column is "None".

d) The types of parameter specifications within the Ada subprogram declarations shall be taken from the library unit package `Interfaces.SQL` and its children `Numerics` and `Varying` and optional children *`Adacsn`* and *`Adacsn`*`.Varying`.

e) 04 09 14 15 16 The declaration of the library unit package `Interfaces.SQL` shall conform to the following template:

```
package Interfaces.SQL is
-- The declarations of CHAR and NCHAR may be subtype declarations
  type CHAR is (See the Syntax Rules)
  type NCHAR is (See the Syntax Rules)
  type SMALLINT is range bs .. ts;
  type INT is range bi .. ti;
  type BIGINT is range bb .. tb;
  type REAL is digits dr;
  type DOUBLE_PRECISION is digits dd;
  type BOOLEAN is new Boolean;
  subtype INDICATOR_TYPE is t;
  type SQLSTATE_TYPE is new CHAR (1 .. 5);
  package SQLSTATE_CODES is
    SUCCESSFUL_COMPLETION_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "00000";
    WARNING_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "01000";
    WARNING_CURSOR_OPERATION_CONFLICT:
      constant SQLSTATE_TYPE := "01001";
    WARNING_DISCONNECT_ERROR:
      constant SQLSTATE_TYPE := "01002";
    WARNING_NULL_VALUE_ELIMINATED_IN_SET_FUNCTION:
      constant SQLSTATE_TYPE := "01003";
    WARNING_STRING_DATA_RIGHT_TRUNCATION:
      constant SQLSTATE_TYPE := "01004";
    WARNING_INSUFFICIENT_ITEM_DESCRIPTOR_AREAS:
      constant SQLSTATE_TYPE := "01005";
    WARNING_PRIVILEGE_NOT_REVOKED:
      constant SQLSTATE_TYPE := "01006";
    WARNING_PRIVILEGE_NOT_GRANTED:
      constant SQLSTATE_TYPE := "01007";
    WARNING_SEARCH_CONDITION_TOO_LONG_FOR_INFORMATION_SCHEMA:
      constant SQLSTATE_TYPE := "01009";
    WARNING_QUERY_EXPRESSION_TOO_LONG_FOR_INFORMATION_SCHEMA:
      constant SQLSTATE_TYPE := "0100A";
    WARNING_DEFAULT_VALUE_TOO_LONG_FOR_INFORMATION_SCHEMA:
      constant SQLSTATE_TYPE := "0100B";
    WARNING_RESULT_SETS_RETURNED:
      constant SQLSTATE_TYPE := "0100C";
    WARNING_ATTEMPT_TO_RETURN_TOO_MANY_RESULT_SETS:
      constant SQLSTATE_TYPE := "0100E";
    WARNING_STATEMENT_TOO_LONG_FOR_INFORMATION_SCHEMA:
      constant SQLSTATE_TYPE := "0100F";
    WARNING_INVALID_NUMBER_OF_CONDITIONS:
      constant SQLSTATE_TYPE := "01012";
    WARNING_ARRAY_DATA_RIGHT_TRUNCATION:
      constant SQLSTATE_TYPE := "0102F";
    NO_DATA_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "02000";
    NO_DATA_NO_ADDITIONAL_RESULT_SETS_RETURNED:
```

```
    constant SQLSTATE_TYPE := "02001";
  DYNAMIC_SQL_ERROR_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "07000";
 DYNAMIC_SQL_ERROR_USING_CLAUSE_DOES_NOT_MATCH_DYNAMIC_PARAMETER_SPECIFICATIONS:

    constant SQLSTATE_TYPE := "07001";
  DYNAMIC_SQL_ERROR_USING_CLAUSE_DOES_NOT_MATCH_TARGET_SPECIFICATIONS:
    constant SQLSTATE_TYPE := "07002";
  DYNAMIC_SQL_ERROR_CURSOR_SPECIFICATION_CANNOT_BE_EXECUTED:
    constant SQLSTATE_TYPE := "07003";
  DYNAMIC_SQL_ERROR_USING_CLAUSE_REQUIRED_FOR_DYNAMIC_PARAMETERS:
    constant SQLSTATE_TYPE := "07004";
  DYNAMIC_SQL_ERROR_PREPARED_STATEMENT_NOT_A_CURSOR_SPECIFICATION:
    constant SQLSTATE_TYPE := "07005";
  DYNAMIC_SQL_ERROR_RESTRICTED_DATA_TYPE_ATTRIBUTE_VIOLATION:
    constant SQLSTATE_TYPE := "07006";
  DYNAMIC_SQL_ERROR_USING_CLAUSE_REQUIRED_FOR_RESULT_FIELDS:
    constant SQLSTATE_TYPE := "07007";
  DYNAMIC_SQL_ERROR_INVALID_DESCRIPTOR_COUNT:
    constant SQLSTATE_TYPE := "07008";
  DYNAMIC_SQL_ERROR_INVALID_DESCRIPTOR_INDEX:
    constant SQLSTATE_TYPE := "07009";
  DYNAMIC_SQL_ERROR_DATA_TYPE_TRANSFORM_FUNCTION_VIOLATION:
    constant SQLSTATE_TYPE := "0700B";
  DYNAMIC_SQL_ERROR_UNDEFINED_DATA_VALUE:
    constant SQLSTATE_TYPE := "0700C";
  DYNAMIC_SQL_ERROR_INVALID_DATA_TARGET:
    constant SQLSTATE_TYPE := "0700D";
  DYNAMIC_SQL_ERROR_INVALID_LEVEL_VALUE:
    constant SQLSTATE_TYPE := "0700E";
  DYNAMIC_SQL_ERROR_INVALID_DATETIME_INTERVAL_CODE:
    constant SQLSTATE_TYPE := "0700F";
  DYNAMIC_SQL_ERROR_INVALID_PASS_THROUGH_SURROGATE_VALUE:
    constant SQLSTATE_TYPE := "0700G";
  DYNAMIC_SQL_ERROR_PIPE_ROW_NOT_DURING_PTF_EXECUTION:
    constant SQLSTATE_TYPE := "0700H";
  CONNECTION_EXCEPTION_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "08000";
  CONNECTION_EXCEPTION_SQL_CLIENT_UNABLE_TO_ESTABLISH_SQL_CONNECTION:
    constant SQLSTATE_TYPE := "08001";
  CONNECTION_EXCEPTION_CONNECTION_NAME_IN_USE:
    constant SQLSTATE_TYPE := "08002";
  CONNECTION_EXCEPTION_CONNECTION_DOES_NOT_EXIST:
    constant SQLSTATE_TYPE := "08003";
  CONNECTION_EXCEPTION_SQL_SERVER_REJECTED_ESTABLISHMENT_OF_SQL_CONNECTION:
    constant SQLSTATE_TYPE := "08004";
  CONNECTION_EXCEPTION_CONNECTION_FAILURE:
    constant SQLSTATE_TYPE := "08006";
  CONNECTION_EXCEPTION_TRANSACTION_RESOLUTION_UNKNOWN:
    constant SQLSTATE_TYPE := "08007";
  TRIGGERED_ACTION_EXCEPTION_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "09000";
  FEATURE_NOT_SUPPORTED_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "0A000";
  FEATURE_NOT_SUPPORTED_MULTIPLE_SERVER_TRANSACTIONS:
    constant SQLSTATE_TYPE := "0A001";
  INVALID_TARGET_TYPE_SPECIFICATION_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "0D000";
  INVALID_SCHEMA_NAME_LIST_SPECIFICATION_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "0E000";
  LOCATOR_EXCEPTION_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "0F000";
  LOCATOR_EXCEPTION_INVALID_SPECIFICATION:
    constant SQLSTATE_TYPE := "0F001";
  INVALID_SQL_INVOKED_PROCEDURE_REFERENCE_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "0M000";
  INVALID_ROLE_SPECIFICATION_NO_SUBCLASS:
    constant SQLSTATE_TYPE := "0P000";
  INVALID_TRANSFORM_GROUP_NAME_SPECIFICATION_NO_SUBCLASS:
```

```
            constant SQLSTATE_TYPE := "0S000";
          TARGET_TABLE_DISAGREES_WITH_CURSOR_SPECIFICATION_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "0T000";
          ATTEMPT_TO_ASSIGN_TO_NON_UPDATABLE_COLUMN_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "0U000";
          ATTEMPT_TO_ASSIGN_TO_ORDERING_COLUMN_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "0V000";
          PROHIBITED_STATEMENT_ENCOUNTERED_DURING_TRIGGER_EXECUTION_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "0W000";
          PROHIBITED_STATEMENT_ENCOUNTERED_DURING_TRIGGER_EXECUTION_MODIFY_TABLE_MODI-
      FIED_BY_DATA_CHANGE_DELTA_TABLE:
            constant SQLSTATE_TYPE := "0W001";
          DIAGNOSTICS_EXCEPTION_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "0Z000";
          DIAGNOSTICS_EXCEPTION_MAXIMUM_NUMBER_OF_STACKED_DIAGNOSTICS_AREAS_EXCEEDED:
            constant SQLSTATE_TYPE := "0Z001";
          PROHIBITED_COLUMN_REFERENCE_ENCOUNTERED_DURING_TRIGGER_EXECUTION_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "11000";
          CARDINALITY_VIOLATION_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "21000";
          DATA_EXCEPTION_NO_SUBCLASS:
            constant SQLSTATE_TYPE := "22000";
          DATA_EXCEPTION_STRING_DATA_RIGHT_TRUNCATION:
            constant SQLSTATE_TYPE := "22001";
          DATA_EXCEPTION_NULL_VALUE_NO_INDICATOR_PARAMETER:
            constant SQLSTATE_TYPE := "22002";
          DATA_EXCEPTION_NUMERIC_VALUE_OUT_OF_RANGE:
            constant SQLSTATE_TYPE := "22003";
          DATA_EXCEPTION_NULL_VALUE_NOT_ALLOWED:
            constant SQLSTATE_TYPE := "22004";
          DATA_EXCEPTION_ERROR_IN_ASSIGNMENT:
            constant SQLSTATE_TYPE := "22005";
          DATA_EXCEPTION_INVALID_INTERVAL_FORMAT:
            constant SQLSTATE_TYPE := "22006";
          DATA_EXCEPTION_INVALID_DATETIME_FORMAT:
            constant SQLSTATE_TYPE := "22007";
          DATA_EXCEPTION_DATETIME_FIELD_OVERFLOW:
            constant SQLSTATE_TYPE := "22008";
          DATA_EXCEPTION_INVALID_TIME_ZONE_DISPLACEMENT_VALUE:
            constant SQLSTATE_TYPE := "22009";
          DATA_EXCEPTION_ESCAPE_CHARACTER_CONFLICT:
            constant SQLSTATE_TYPE := "2200B";
          DATA_EXCEPTION_INVALID_USE_OF_ESCAPE_CHARACTER:
            constant SQLSTATE_TYPE := "2200C";
          DATA_EXCEPTION_INVALID_ESCAPE_OCTET:
            constant SQLSTATE_TYPE := "2200D";
          DATA_EXCEPTION_NULL_VALUE_IN_ARRAY_TARGET:
            constant SQLSTATE_TYPE := "2200E";
          DATA_EXCEPTION_ZERO_LENGTH_CHARACTER_STRING:
            constant SQLSTATE_TYPE := "2200F";
          DATA_EXCEPTION_MOST_SPECIFIC_TYPE_MISMATCH:
            constant SQLSTATE_TYPE := "2200G";
          DATA_EXCEPTION_SEQUENCE_GENERATOR_LIMIT_EXCEEDED:
            constant SQLSTATE_TYPE := "2200H";
          DATA_EXCEPTION_INTERVAL_VALUE_OUT_OF_RANGE:
            constant SQLSTATE_TYPE := "2200P";
          DATA_EXCEPTION_MULTISET_VALUE_OVERFLOW:
            constant SQLSTATE_TYPE := "2200Q";
          DATA_EXCEPTION_INVALID_INDICATOR_PARAMETER_VALUE:
            constant SQLSTATE_TYPE := "22010";
          DATA_EXCEPTION_SUBSTRING_ERROR:
            constant SQLSTATE_TYPE := "22011";
          DATA_EXCEPTION_DIVISION_BY_ZERO:
            constant SQLSTATE_TYPE := "22012";
          DATA_EXCEPTION_INVALID_PRECEDING_OR_FOLLOWING_SIZE_IN_WINDOW_FUNCTION:
            constant SQLSTATE_TYPE := "22013";
          DATA_EXCEPTION_INVALID_ARGUMENT_FOR_NTILE_FUNCTION:
            constant SQLSTATE_TYPE := "22014";
          DATA_EXCEPTION_INTERVAL_FIELD_OVERFLOW:
```

```
    constant SQLSTATE_TYPE := "22015";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_NTH_VALUE_FUNCTION:
    constant SQLSTATE_TYPE := "22016";
DATA_EXCEPTION_INVALID_CHARACTER_VALUE_FOR_CAST:
    constant SQLSTATE_TYPE := "22018";
DATA_EXCEPTION_INVALID_ESCAPE_CHARACTER:
    constant SQLSTATE_TYPE := "22019";
DATA_EXCEPTION_INVALID_REGULAR_EXPRESSION:
    constant SQLSTATE_TYPE := "2201B";
DATA_EXCEPTION_NULL_ROW_NOT_PERMITTED_IN_TABLE:
    constant SQLSTATE_TYPE := "2201C";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_NATURAL_LOGARITHM:
    constant SQLSTATE_TYPE := "2201E";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_POWER_FUNCTION:
    constant SQLSTATE_TYPE := "2201F";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_WIDTH_BUCKET_FUNCTION:
    constant SQLSTATE_TYPE := "2201G";
DATA_EXCEPTION_INVALID_ROW_VERSION:
    constant SQLSTATE_TYPE := "2201H";
DATA_EXCEPTION_INVALID_XQUERY_REGULAR_EXPRESSION:
    constant SQLSTATE_TYPE := "2201S";
DATA_EXCEPTION_INVALID_XQUERY_OPTION_FLAG:
    constant SQLSTATE_TYPE := "2201T";
DATA_EXCEPTION_ATTEMPT_TO_REPLACE_A_ZERO_LENGTH_STRING:
    constant SQLSTATE_TYPE := "2201U";
DATA_EXCEPTION_INVALID_XQUERY_REPLACEMENT_STRING:
    constant SQLSTATE_TYPE := "2201V";
DATA_EXCEPTION_INVALID_ROW_COUNT_IN_FETCH_FIRST_CLAUSE:
    constant SQLSTATE_TYPE := "2201W";
DATA_EXCEPTION_INVALID_ROW_COUNT_IN_RESULT_OFFSET_CLAUSE:
    constant SQLSTATE_TYPE := "2201X";
DATA_EXCEPTION_ZERO_LENGTH_BINARY_STRING:
    constant SQLSTATE_TYPE := "2201Y";
DATA_EXCEPTION_INVALID_PERIOD_VALUE:
    constant SQLSTATE_TYPE := "22020";
DATA_EXCEPTION_CHARACTER_NOT_IN_REPERTOIRE:
    constant SQLSTATE_TYPE := "22021";
DATA_EXCEPTION_INDICATOR_OVERFLOW:
    constant SQLSTATE_TYPE := "22022";
DATA_EXCEPTION_INVALID_PARAMETER_VALUE:
    constant SQLSTATE_TYPE := "22023";
DATA_EXCEPTION_UNTERMINATED_C_STRING:
    constant SQLSTATE_TYPE := "22024";
DATA_EXCEPTION_INVALID_ESCAPE_SEQUENCE:
    constant SQLSTATE_TYPE := "22025";
DATA_EXCEPTION_TRIM_ERROR:
    constant SQLSTATE_TYPE := "22027";
DATA_EXCEPTION_NON_CHARACTER_IN_CHARACTER_STRING:
    constant SQLSTATE_TYPE := "22029";
DATA_EXCEPTION_NULL_VALUE_SUBSTITUTED_FOR_MUTATOR_SUBJECT_PARAMETER:
    constant SQLSTATE_TYPE := "2202D";
DATA_EXCEPTION_ARRAY_ELEMENT_ERROR:
    constant SQLSTATE_TYPE := "2202E";
DATA_EXCEPTION_ARRAY_DATA_RIGHT_TRUNCATION:
    constant SQLSTATE_TYPE := "2202F";
DATA_EXCEPTION_INVALID_REPEAT_ARGUMENT_IN_A_SAMPLE_CLAUSE:
    constant SQLSTATE_TYPE := "2202G";
DATA_EXCEPTION_INVALID_SAMPLE_SIZE:
    constant SQLSTATE_TYPE := "2202H";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_ROW_PATTERN_NAVIGATION_FUNCTION:
    constant SQLSTATE_TYPE := "2202J";
DATA_EXCEPTION_SKIP_TO_NON_EXISTENT_ROW:
    constant SQLSTATE_TYPE := "2202K";
DATA_EXCEPTION_SKIP_TO_FIRST_ROW_OF_MATCH:
    constant SQLSTATE_TYPE := "2202L";
DATA_EXCEPTION_NON_BOOLEAN_SQL_JSON_ITEM:
    constant SQLSTATE_TYPE := "2202V";
DATA_EXCEPTION_NON_DATE_SQL_JSON_ITEM:
    constant SQLSTATE_TYPE := "2202W";
```

```
DATA_EXCEPTION_NON_STRING_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "2202X";
DATA_EXCEPTION_NON_TIME_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "2202Y";
DATA_EXCEPTION_NON_TIMESTAMP_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "2202Z";
DATA_EXCEPTION_DUPLICATE_JSON_OBJECT_KEY_VALUE:
  constant SQLSTATE_TYPE := "22030";
DATA_EXCEPTION_INVALID_ARGUMENT_FOR_SQL_JSON_DATETIME_FUNCTION:
  constant SQLSTATE_TYPE := "22031";
DATA_EXCEPTION_INVALID_JSON_TEXT:
  constant SQLSTATE_TYPE := "22032";
DATA_EXCEPTION_INVALID_SQL_JSON_SUBSCRIPT:
  constant SQLSTATE_TYPE := "22033";
DATA_EXCEPTION_MORE_THAN_ONE_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "22034";
DATA_EXCEPTION_NO_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "22035";
DATA_EXCEPTION_NON_NUMERIC_SQL_JSON_ITEM:
  constant SQLSTATE_TYPE := "22036";
DATA_EXCEPTION_NON_UNIQUE_KEYS_IN_A_JSON_OBJECT:
  constant SQLSTATE_TYPE := "22037";
DATA_EXCEPTION_SINGLETON_SQL_JSON_ITEM_REQUIRED:
  constant SQLSTATE_TYPE := "22038";
DATA_EXCEPTION_SQL_JSON_ARRAY_NOT_FOUND:
  constant SQLSTATE_TYPE := "22039";
DATA_EXCEPTION_SQL_JSON_MEMBER_NOT_FOUND:
  constant SQLSTATE_TYPE := "2203A";
DATA_EXCEPTION_SQL_JSON_NUMBER_NOT_FOUND:
  constant SQLSTATE_TYPE := "2203B";
DATA_EXCEPTION_SQL_JSON_OBJECT_NOT_FOUND:
  constant SQLSTATE_TYPE := "2203C";
DATA_EXCEPTION_TOO_MANY_JSON_ARRAY_ELEMENTS:
  constant SQLSTATE_TYPE := "2203D";
DATA_EXCEPTION_TOO_MANY_JSON_OBJECT_MEMBERS:
  constant SQLSTATE_TYPE := "2203E";
DATA_EXCEPTION_SQL_JSON_SCALAR_REQUIRED:
  constant SQLSTATE_TYPE := "2203F";
DATA_EXCEPTION_SQL_JSON_ITEM_CANNOT_BE_CAST_TO_TARGET_TYPE:
  constant SQLSTATE_TYPE := "2203G";
INTEGRITY_CONSTRAINT_VIOLATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE := "23000";
INTEGRITY_CONSTRAINT_VIOLATION_RESTRICT_VIOLATION:
  constant SQLSTATE_TYPE := "23001";
INVALID_CURSOR_STATE_NO_SUBCLASS:
  constant SQLSTATE_TYPE := "24000";
INVALID_TRANSACTION_STATE_NO_SUBCLASS:
  constant SQLSTATE_TYPE := "25000";
INVALID_TRANSACTION_STATE_ACTIVE_SQL_TRANSACTION:
  constant SQLSTATE_TYPE := "25001";
INVALID_TRANSACTION_STATE_BRANCH_TRANSACTION_ALREADY_ACTIVE:
  constant SQLSTATE_TYPE := "25002";
INVALID_TRANSACTION_STATE_INAPPROPRIATE_ACCESS_MODE_FOR_BRANCH_TRANSACTION:
  constant SQLSTATE_TYPE := "25003";
INVALID_TRANSACTION_STATE_INAPPROPRIATE_ISOLATION_LEVEL_FOR_BRANCH_TRANSACTION:
  constant SQLSTATE_TYPE := "25004";
INVALID_TRANSACTION_STATE_NO_ACTIVE_SQL_TRANSACTION_FOR_BRANCH_TRANSACTION:
  constant SQLSTATE_TYPE := "25005";
INVALID_TRANSACTION_STATE_READ_ONLY_SQL_TRANSACTION:
  constant SQLSTATE_TYPE := "25006";
INVALID_TRANSACTION_STATE_SCHEMA_AND_DATA_STATEMENT_MIXING_NOT_SUPPORTED:
  constant SQLSTATE_TYPE := "25007";
INVALID_TRANSACTION_STATE_HELD_CURSOR_REQUIRES_SAME_ISOLATION_LEVEL:
  constant SQLSTATE_TYPE := "25008";
INVALID_SQL_STATEMENT_NAME_NO_SUBCLASS:
  constant SQLSTATE_TYPE := "26000";
TRIGGERED_DATA_CHANGE_VIOLATION_NO_SUBCLASS:
  constant SQLSTATE_TYPE := "27000";
```

```
TRIGGERED_DATA_CHANGE_VIOLATION_MODIFY_TABLE_MODIFIED_BY_DATA_CHANGE_DELTA_TABLE:

   constant SQLSTATE_TYPE := "27001";
INVALID_AUTHORIZATION_SPECIFICATION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "28000";
DEPENDENT_PRIVILEGE_DESCRIPTORS_STILL_EXIST_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "2B000";
INVALID_CHARACTER_SET_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "2C000";
INVALID_TRANSACTION_TERMINATION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "2D000";
INVALID_CONNECTION_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "2E000";
SQL_ROUTINE_EXCEPTION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "2F000";
SQL_ROUTINE_EXCEPTION_MODIFYING_SQL_DATA_NOT_PERMITTED:
   constant SQLSTATE_TYPE := "2F002";
SQL_ROUTINE_EXCEPTION_PROHIBITED_SQL_STATEMENT_ATTEMPTED:
   constant SQLSTATE_TYPE := "2F003";
SQL_ROUTINE_EXCEPTION_READING_SQL_DATA_NOT_PERMITTED:
   constant SQLSTATE_TYPE := "2F004";
SQL_ROUTINE_EXCEPTION_FUNCTION_EXECUTED_NO_RETURN_STATEMENT:
   constant SQLSTATE_TYPE := "2F005";
INVALID_COLLATION_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "2H000";
INVALID_SQL_STATEMENT_IDENTIFIER_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "30000";
INVALID_SQL_DESCRIPTOR_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "33000";
INVALID_CURSOR_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "34000";
INVALID_CONDITION_NUMBER_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "35000";
CURSOR_SENSITIVITY_EXCEPTION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "36000";
CURSOR_SENSITIVITY_EXCEPTION_REQUEST_REJECTED:
   constant SQLSTATE_TYPE := "36001";
CURSOR_SENSITIVITY_EXCEPTION_REQUEST_FAILED:
   constant SQLSTATE_TYPE := "36002";
EXTERNAL_ROUTINE_EXCEPTION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "38000";
EXTERNAL_ROUTINE_EXCEPTION_CONTAINING_SQL_NOT_PERMITTED:
   constant SQLSTATE_TYPE := "38001";
EXTERNAL_ROUTINE_EXCEPTION_MODIFYING_SQL_DATA_NOT_PERMITTED:
   constant SQLSTATE_TYPE := "38002";
EXTERNAL_ROUTINE_EXCEPTION_PROHIBITED_SQL_STATEMENT_ATTEMPTED:
   constant SQLSTATE_TYPE := "38003";
EXTERNAL_ROUTINE_EXCEPTION_READING_SQL_DATA_NOT_PERMITTED:
   constant SQLSTATE_TYPE := "38004";
EXTERNAL_ROUTINE_INVOCATION_EXCEPTION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "39000";
EXTERNAL_ROUTINE_INVOCATION_EXCEPTION_NULL_VALUE_NOT_ALLOWED:
   constant SQLSTATE_TYPE := "39004";
SAVEPOINT_EXCEPTION_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "3B000";
SAVEPOINT_EXCEPTION_INVALID_SPECIFICATION:
   constant SQLSTATE_TYPE := "3B001";
SAVEPOINT_EXCEPTION_TOO_MANY:
   constant SQLSTATE_TYPE := "3B002";
AMBIGUOUS_CURSOR_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "3C000";
INVALID_CATALOG_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "3D000";
INVALID_SCHEMA_NAME_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "3F000";
TRANSACTION_ROLLBACK_NO_SUBCLASS:
   constant SQLSTATE_TYPE := "40000";
TRANSACTION_ROLLBACK_SERIALIZATION_FAILURE:
   constant SQLSTATE_TYPE := "40001";
```

```
    TRANSACTION_ROLLBACK_INTEGRITY_CONSTRAINT_VIOLATION:
      constant SQLSTATE_TYPE := "40002";
    TRANSACTION_ROLLBACK_STATEMENT_COMPLETION_UNKNOWN:
      constant SQLSTATE_TYPE := "40003";
    TRANSACTION_ROLLBACK_TRIGGERED_ACTION_EXCEPTION:
      constant SQLSTATE_TYPE := "40004";
    SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "42000";
    WITH_CHECK_OPTION_VIOLATION_NO_SUBCLASS:
      constant SQLSTATE_TYPE := "44000";
  end SQLSTATE_CODES;
end Interfaces.SQL;
```

where *bs*, *ts*, *bi*, *ti*, *bb*, *tb*, *dr*, *dd*, *bsc*, and *tsc* are implementation-defined (IV226) integer values. *t* is INT or SMALLINT, corresponding with an implementation-defined (IV227) <exact numeric type> of indicator parameters.

The library unit package `Interfaces.SQL` may contain additional declarations as specified in subsequent Syntax Rules.

The text of the Ada library unit package Interfaces.SQL is also available from the ISO website as a "digital artifact". See `https://standards.iso.org/iso-iec/9075/-2/ed-6/en/` to download digital artifacts for this document. To download the library unit package, select the file named `ISO_IEC_9075-2(E)_Foundation-Interfaces.SQL.ada`.

NOTE 656 — The Ada identifier INVALID_SQL_STATEMENT appears for compatibility with earlier editions of the ISO/IEC 9075 series. However, the intended symbol is INVALID_SQL_STATEMENT_IDENTIFIER_NO_SUBCLASS, which was added to correspond correctly with the exception condition name.

f) The library unit package `Interfaces.SQL.Numerics` shall contain a sequence of decimal fixed point type declarations of the following form:

```
type Scale_s is delta 10.0 ** - s digits max_p;
```

where *s* is an integer ranging from 0 (zero) to an implementation-defined (IL066) maximum value and *max_p* is an implementation-defined (IL066) integer maximum precision.

g) The library unit package `Interfaces.SQL.VARYING` shall contain type or subtype declarations with the defining identifiers CHAR and NCHAR. The definitions of these type or subtype declarations are implementation-defined (IV228), but shall support the SQL data types CHARACTER VARYING and NATIONAL CHARACTER VARYING, respectively.

h) Let *SQLcsn* be a <character set name> and let *Adacsn* be the result of replacing <period>s in *SQLcsn* with <underscore>s. If *Adacsn* is a valid Ada identifier, then the library unit packages `Interfaces.SQL.`*Adacsn* and `Interfaces.SQL.`*Adacsn*`.Varying` shall contain a type or subtype declaration with defining identifier CHAR. If *Adacsn* is not a valid Ada identifier, then the names of these packages shall be implementation-defined (IV225).

i) `Interfaces.SQL` and its children may contain context clauses and representation items as needed. These packages may also contain declarations of Ada character types as needed to support the declarations of the types CHAR and NCHAR. The definitions of `Interfaces.SQL.CHAR` and `Interfaces.SQL.NCHAR` are implementation-defined (IV229), but shall support the SQL data types CHARACTER and NATIONAL CHARACTER, respectively.

NOTE 657 — If the implementation-defined (IV229) character set specification used by default with a fixed-length character string type is Latin1, then the declaration

```
subtype CHAR is String;
```

within `Interfaces.SQL` and the declaration

```
subtype CHAR is
    Ada.Strings.Unbounded.Unbounded_String;
```

within `Interfaces.SQL.Varying` (assuming the appropriate context clause) conform to the requirements of this paragraph of this Subclause. If the character set underlying NATIONAL CHARACTER is supported by an Ada package specification `Host_Char_Pkg` that declares a type `String_Type` that stores strings over the given character set, and furthermore the package specification `Host_Char_Pkg_Varying` (not necessary distinct from `Host_Char_Pkg`) declares a type `String_Type_Varying` that reproduces the functionality of `Ada.Strings.Unbounded.Unbounded_String` over the national character string type (rather than Latin1), then the declaration

```
subtype NCHAR is Host_Char_Pkg.String_Type;
```

within `Interfaces.SQL` and the declaration

```
subtype NCHAR is Host_Char_Pkg_Varying.String_Type_Varying;
```

within `Interfaces.SQL.Varying` conform to the requirements of this paragraph. Similar comments apply to other character sets and the packages `Interfaces.SQL.`*`Adacsn`* and `Interfaces.SQL.`*`Adacsn`*`_Varying`.

j) The library unit package `Interfaces.SQL` shall contain declarations of the following form:

```
package CHARACTER_SET renames Interfaces.SQL.Adacsn;
subtype CHARACTER_TYPE is CHARACTER_SET.cst;
```

where *cst* is a data type capable of storing a single character from the default character set. The package `Interfaces.SQL.`*`Adacsn`* shall contain the necessary declaration for *cst*.

NOTE 658 — If the default character set is Latin1, then a declaration of the form:

```
package CHARACTER_SET is
   subtype cst is Character;
end CHARACTER_SET;
```

can be substituted for the renaming declaration of CHARACTER_SET.

k) The base type of the SQLSTATE parameter shall be `Interfaces.SQL.SQLSTATE_TYPE`.

l) The Ada parameter mode of the SQLSTATE parameter is out.

m) If the *i*-th <host parameter declaration> specifies a <data type> that is:

i) CHARACTER(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.CHAR`.

ii) CHARACTER VARYING(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.VARYING.CHAR`.

iii) NATIONAL CHARACTER(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.NCHAR`.

iv) NATIONAL CHARACTER VARYING(*L*) for some *L*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.VARYING.NCHAR`.

v) CHARACTER(*L*) CHARACTER SET *csn* for some *L* and some character set name *csn*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.`*`Adacsn`*`.CHAR`.

vi) CHARACTER VARYING(*L*) CHARACTER SET *csn* for some *L* and some character set name *csn*, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.`*`Adacsn`*`.VARYING.CHAR`.

If *P* is an actual parameter associated with the *i*-th parameter in a call to the encompassing procedure, then *P* shall be sufficient to hold a character string of length *L* in the appropriate character set.

NOTE 659 — If a character set uses fixed length encodings, then the definition of the subtype CHAR for fixed-length strings can be an array type whose element type is an Ada character type. If that Ada character type is defined so as to use the number of bits per character used by the SQL encoding, then the restriction on P is precisely P'LENGTH = *L*. For variable-length strings using fixed length encodings, if the definition of CHAR in

the appropriate VARYING package is based on the type `Ada.Strings.Unbounded.Unbounded_String`, there is no restriction on *P*. Otherwise, a precise statement of the restriction on *P* is implementation-defined (IA232).

n) If the *i*-th <host parameter declaration> specifies a <data type> that is NUMERIC(*P*,*S*) for some <precision> *P* and <scale> *S*, then the Ada library unit package generated for the encompassing module shall contain a declaration equivalent to:

```
subtype Numeric_p_s is
  Interfaces.SQL.Numerics.Scale_s digits p;
```

The subtype mark in the *i*-th parameter declaration shall specify this subtype.

o) If the *i*-th <host parameter declaration> specifies a <data type> that is SMALLINT, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.SMALLINT`.

p) If the *i*-th <host parameter declaration> specifies a <data type> that is INTEGER, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.INT`.

q) If the *i*-th <host parameter declaration> specifies a <data type> that is BIGINT, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.BIGINT`.

r) If the *i*-th <host parameter declaration> specifies a <data type> that is REAL, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.REAL`.

s) If the *i*-th <host parameter declaration> specifies a <data type> that is DOUBLE_PRECISION, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.DOUBLE_PRECISION`.

t) If the *i*-th <host parameter declaration> specifies <locator indication>, then the subtype mark in the *i*-th parameter declaration shall specify `Interfaces.SQL.INT`.

u) For every parameter,

Case:

i) If *PMODE* is equal to "IN", then the Ada parameter mode is **in**.

ii) If *PMODE* is equal to "OUT", then the Ada parameter mode is **out**.

iii) If *PMODE* is equal to "INOUT", then the Ada parameter mode is **in out**.

iv) Otherwise, the Ada parameter mode is **in**, **out**, or **in out**.

v) The following Ada library unit renaming declaration exists:

```
with Interfaces.SQL;
package SQL_Standard renames Interfaces.SQL.
```

11) If the caller language of the <externally-invoked procedure> is C, then:

a) The declared type of an SQLSTATE host parameter shall be C `char` with length 6.

b) For each *i*, 1 (one) ≤ *i* ≤ *n*,

Case:

i) If PD_i contains <locator indication>, then ADT_i shall be C `unsigned long`.

ii) Otherwise:

1) PDT_i shall not identify a data type listed in the "SQL data type" column of Table 22, "Data type correspondences for C", for which the corresponding row in the "C data type" column is "None".

2) ADT_i shall be the data type listed in the "C data type" column of Table 22, "Data type correspondences for C", for which the corresponding row in the "SQL data type" column is PDT_i.

12) If the caller language of the <externally-invoked procedure> is COBOL, then:

a) The declared type of an SQLSTATE host parameter shall be COBOL `PICTURE X(5)`.

b) For each i, 1 (one) $\leq i \leq n$,

Case:

i) If PD_i contains <locator indication>, then ADT_i shall be COBOL `PIC S9(9) USAGE IS BINARY`.

ii) Otherwise:

1) PDT_i shall not identify a data type listed in the "SQL data type" column of Table 23, "Data type correspondences for COBOL", for which the corresponding row in the "COBOL data type" column is "None".

2) ADT_i shall be the data type listed in the "COBOL data type" column of Table 23, "Data type correspondences for COBOL", for which the corresponding row in the "SQL data type" column is PDT_i.

13) If the caller language of the <externally-invoked procedure> is FORTRAN, then:

a) The declared type of an SQLSTATE host parameter shall be Fortran CHARACTER with length 5.

b) For each i, 1 (one) $\leq i \leq n$,

Case:

i) If PD_i contains <locator indication>, then ADT_i shall be Fortran `INTEGER`.

ii) Otherwise:

1) PDT_i shall not identify a data type listed in the "SQL data type" column of Table 24, "Data type correspondences for Fortran", for which the corresponding row in the "Fortran data type" column is "None".

2) ADT_i shall be the data type listed in the "Fortran data type" column of Table 24, "Data type correspondences for Fortran", for which the corresponding row in the "SQL data type" column is PDT_i.

14) If the caller language of the <externally-invoked procedure> is M, then:

a) The declared type of an SQLSTATE host parameter shall be M character.

b) For each i, 1 (one) $\leq i \leq n$,

Case:

i) If PD_i contains <locator indication>, then ADT_i shall be M character.

ii) Otherwise:

1) PDT_i shall not identify a data type listed in the "SQL data type" column of Table 25, "Data type correspondences for M", for which the corresponding row in the "M data type" column is "None".

2) ADT_i shall be the data type listed in the "M data type" column of Table 25, "Data type correspondences for M", for which the corresponding row in the "SQL data type" column is PDT_i.

15) If the caller language of the <externally-invoked procedure> is PASCAL, then:

a) The declared type of an SQLSTATE host parameter shall be Pascal `PACKED ARRAY[1..5] OF CHAR`.

b) For each i, 1 (one) $\leq i \leq n$,

Case:

i) If PD_i contains <locator indication>, then ADT_i shall be Pascal `INTEGER`.

ii) Otherwise:

1) PDT_i shall not identify a data type listed in the "SQL data type" column of Table 26, "Data type correspondences for Pascal", for which the corresponding row in the "Pascal data type" column is "None".

2) ADT_i shall be the data type listed in the "Pascal data type" column of Table 26, "Data type correspondences for Pascal", for which the corresponding row in the "SQL data type" column is PDT_i.

16) If the caller language of the <externally-invoked procedure> is PLI, then:

a) The declared type of an SQLSTATE host parameter shall be PL/I `CHARACTER(5)`.

b) For each i, 1 (one) $\leq i \leq n$,

Case:

i) If PD_i contains <locator indication>, then ADT_i shall be PL/I `INTEGER`.

ii) Otherwise:

1) PDT_i shall not identify a data type listed in the "SQL data type" column of Table 27, "Data type correspondences for PL/I", for which the corresponding row in the "PL/I data type" column is "None".

2) ADT_i shall be the data type listed in the "PL/I data type" column of Table 27, "Data type correspondences for PL/I", for which the corresponding row in the "SQL data type" column is PDT_i.

Access Rules

None.

General Rules

1) An <externally-invoked procedure> defines an *externally-invoked procedure* that may be called by an SQL-agent.

2) If the <SQL-client module definition> that contains the <externally-invoked procedure> is associated with an SQL-agent that is associated with another <SQL-client module definition> that contains an <externally-invoked procedure> with equivalent <procedure name>s, then the effect is implementation-defined (IA058).

3) The language identified by the <language name> contained in the <language clause> of the <SQL-client module definition> that contains an <externally-invoked procedure> is the *caller language* of the <externally-invoked procedure>.

4) If the SQL-agent that performs a call of an <externally-invoked procedure> is not a program that conforms to the specification for the programming language specified by the caller language of the <externally-invoked procedure>, then the effect is implementation-dependent (UA059).

5) If the caller language of an <externally-invoked procedure> is ADA and the SQL-agent performs calls of <externally-invoked procedure>s from more than one Ada task, then the results are implementation-dependent (UV104).

6) If the <SQL-client module definition> that contains the <externally-invoked procedure> has an explicit <module authorization identifier> *MAI* that is not equivalent to the SQL-session <authorization identifier> *SAI*, then:

   a) Whether or not *SAI* can invoke <externally-invoked procedure>s in an <SQL-client module definition> with explicit <module authorization identifier> *MAI* is implementation-defined (IA231), as are any restrictions pertaining to such invocation.

   b) If *SAI* is restricted from invoking an <externally-invoked procedure> in an <SQL-client module definition> with explicit <module authorization identifier> *MAI*, then an exception condition is raised: *invalid authorization specification (28000)*.

7) If the value of any input host parameter provided by the SQL-agent falls outside the set of allowed values of the declared type of the host parameter, or if the value of any output host parameter resulting from the execution of the <externally-invoked procedure> falls outside the set of values supported by the SQL-agent for that host parameter, then the effect is implementation-defined (IA082). If the implementation-defined (IA082) effect is the raising of an exception condition, then an exception condition is raised: *data exception — invalid parameter value (22023)*.

8) A copy of the top cell of the authorization stack is pushed onto the authorization stack. If the SQL-client module *M* that includes the externally-invoked procedure has an owner, then the top cell of the authorization stack is set to contain only the authorization identifier of the owner of *M*.

9) If this is the first invocation of any externally-invoked procedure of *M* in the current SQL-session, then for each cursor declaration descriptor *CDD* of a declared cursor of *M*, let *CID* be a new cursor instance descriptor whose cursor declaration descriptor is *CDD*, whose SQL-session is the current SQL-session, and whose state is closed.

10) Let *S* be the <SQL procedure statement> of the <externally-invoked procedure>.

11) The General Rules of Subclause 9.17, "Executing an <SQL procedure statement>", are applied with *S* as *EXECUTING STATEMENT*.

12) Upon completion of execution, the top cell in the authorization stack is removed.

Conformance Rules

1) Without Feature S231, "Structured type locators", conforming SQL language shall not contain a <host parameter data type> that simply contains a <data type> that specifies a structured type and that contains a <locator indication>.

2) Without Feature S232, "Array locators", conforming SQL language shall not contain a <host parameter data type> that simply contains an <array type> and that contains a <locator indication>.

3) Without Feature S233, "Multiset locators", conforming SQL language shall not contain a <host parameter data type> that simply contains a <multiset type> and that contains a <locator indication>.

4) 04 Without Feature B221, "Routine language Ada: VARCHAR and NUMERIC support", if the caller language of the <externally-invoked procedure> is Ada, then a <host parameter data type> shall not be CHARACTER VARYING or NUMERIC.

13.4 <SQL procedure statement>

This Subclause is modified by Subclause 12.2, "<SQL procedure statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 12.3, "<SQL procedure statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 13.2, "<SQL procedure statement>", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 13.2, "<SQL procedure statement>", in ISO/IEC 9075-16.

Function

Define all of the SQL-statements that are <SQL procedure statement>s.

Format

```
<SQL procedure statement> ::=
  <SQL executable statement>

<SQL executable statement> ::=
    <SQL schema statement>
  | <SQL data statement>
  | <SQL control statement>
  | <SQL transaction statement>
  | <SQL connection statement>
  | <SQL session statement>
  | <SQL diagnostics statement>
  | <SQL dynamic statement>

<SQL schema statement> ::=
    <SQL schema definition statement>
  | <SQL schema manipulation statement>

04 | 09 | 16 <SQL schema definition statement> ::=
    <schema definition>
  | <table definition>
  | <view definition>
  | <SQL-invoked routine>
  | <grant statement>
  | <role definition>
  | <domain definition>
  | <character set definition>
  | <collation definition>
  | <transliteration definition>
  | <assertion definition>
  | <trigger definition>
  | <user-defined type definition>
  | <user-defined cast definition>
  | <user-defined ordering definition>
  | <transform definition>
  | <sequence generator definition>

04 | 09 | 16 <SQL schema manipulation statement> ::=
    <drop schema statement>
  | <alter table statement>
  | <drop table statement>
  | <drop view statement>
  | <alter routine statement>
  | <drop routine statement>
  | <drop user-defined cast statement>
  | <revoke statement>
  | <drop role statement>
```

```
  | <alter domain statement>
  | <drop domain statement>
  | <drop character set statement>
  | <drop collation statement>
  | <drop transliteration statement>
  | <drop assertion statement>
  | <drop trigger statement>
  | <alter type statement>
  | <drop data type statement>
  | <drop user-defined ordering statement>
  | <alter transform statement>
  | <drop transform statement>
  | <alter sequence generator statement>
  | <drop sequence generator statement>

<SQL data statement> ::=
    <open statement>
  | <fetch statement>
  | <close statement>
  | <select statement: single row>
  | <free locator statement>
  | <hold locator statement>
  | <SQL data change statement>

<SQL data change statement> ::=
    <delete statement: positioned>
  | <delete statement: searched>
  | <insert statement>
  | <update statement: positioned>
  | <update statement: searched>
  | <truncate table statement>
  | <merge statement>

04 <SQL control statement> ::=
    <call statement>
  | <return statement>

<SQL transaction statement> ::=
    <start transaction statement>
  | <set transaction statement>
  | <set constraints mode statement>
  | <savepoint statement>
  | <release savepoint statement>
  | <commit statement>
  | <rollback statement>

<SQL connection statement> ::=
    <connect statement>
  | <set connection statement>
  | <disconnect statement>

09 14 <SQL session statement> ::=
    <set session user identifier statement>
  | <set role statement>
  | <set local time zone statement>
  | <set session characteristics statement>
  | <set catalog statement>
  | <set schema statement>
  | <set names statement>
  | <set path statement>
  | <set transform group statement>
  | <set session collation statement>
```

```
04 <SQL diagnostics statement> ::=
  <get diagnostics statement>

<SQL dynamic statement> ::=
    <SQL descriptor statement>
  | <prepare statement>
  | <deallocate prepared statement>
  | <describe statement>
  | <execute statement>
  | <execute immediate statement>
  | <SQL dynamic data statement>
  | <copy descriptor statement>
  | <pipe row statement>

<SQL dynamic data statement> ::=
    <allocate extended dynamic cursor statement>
  | <allocate received cursor statement>
  | <dynamic open statement>
  | <dynamic fetch statement>
  | <dynamic close statement>
  | <dynamic delete statement: positioned>
  | <dynamic update statement: positioned>

<SQL descriptor statement> ::=
    <allocate descriptor statement>
  | <deallocate descriptor statement>
  | <set descriptor statement>
  | <get descriptor statement>
```

Syntax Rules

1) Let *S* be the <SQL procedure statement>.

2) 04 If *S* is an <SQL schema definition statement> that is an <SQL-invoked routine>, then the SQL-invoked routine specified by <SQL-invoked routine> shall be a schema-level routine.

NOTE 660 — "schema-level routine" is defined in Subclause 11.60, "<SQL-invoked routine>".

Access Rules

None.

General Rules

1) Let *S* be the <SQL procedure statement>.

2) The General Rules of Subclause 9.17, "Executing an <SQL procedure statement>", are applied with *S* as *EXECUTING STATEMENT*.

Conformance Rules

None.

13.5 Data type correspondences

This Subclause is modified by Subclause 12.4, "Data type correspondences", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 13.3, "Data type correspondences", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 12.2, "Data type correspondences", in ISO/IEC 9075-15.

Function

Specify the data type correspondences for SQL data types and host language types.

Tables

In the following tables, let *P* be <precision>, *S* be <scale>, *L* be <length> or the numeric value of <large object length>, *U* be the <char length units>, *CS* be the <character set specification>, *T* be <time fractional seconds precision>, *Q* be <interval qualifier>, and *N* be the implementation-defined (IV188) size of a structured type reference.

Table 21 — Data type correspondences for Ada

SQL Data Type	Ada Data Type
SQLSTATE	`Interfaces.SQL.SQLSTATE_TYPE`
CHARACTER (*L U*) CHARACTER SET *CS*	`Interfaces.SQL.CHAR(1..L*k)` [1]
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	`Interfaces.SQL.VARYING.CHAR (1..L*k)` or `Interfaces.SQL.VARYING.NCHAR (1..L*k)` [1 3]

CHARACTER LARGE OBJECT (*L U*)
CHARACTER SET *CS*
	`TYPE HVN IS`
	`  RECORD`
	`  HVN_RESERVED : Interfaces.SQL.INT;`
	`  HVN_LENGTH : Interfaces.SQL.INT;`
	`  HVN_DATA : Interfaces.SQL.CHAR(1..L*k);`
	`  END RECORD;`
	1 2 4

BINARY	`Interfaces.SQL.CHAR(1..L)`

BINARY VARYING	`TYPE HVN IS`
	`  RECORD`
	`  HVN_RESERVED : Interfaces.SQL.INT;`
	`  HVN_LENGTH : Interfaces.SQL.INT;`
	`  HVN_DATA : Interfaces.SQL.CHAR(1..L);`
	`  END RECORD;`
	2 4

BINARY LARGE OBJECT(*L*)	`TYPE HVN IS`
	`  RECORD`
	`  HVN_RESERVED : Interfaces.SQL.INT;`
	`  HVN_LENGTH : Interfaces.SQL.INT;`
	`  HVN_DATA : Interfaces.SQL.CHAR(1..L);`
	`  END RECORD;`
	2 4

SQL Data Type	Ada Data Type
NUMERIC(*P*,*S*)	`Interfaces.SQL.NUMERIC_P_S` [3]
DECIMAL(*P*,*S*)	*None*
SMALLINT	`Interfaces.SQL.SMALLINT`
INTEGER	`Interfaces.SQL.INT`
BIGINT	`Interfaces.SQL.BIGINT`
DECFLOAT(*P*)	*None*
FLOAT(*P*)	*None*
REAL	`Interfaces.SQL.REAL`
DOUBLE PRECISION	`Interfaces.SQL.DOUBLE_PRECISION`
BOOLEAN	`Interfaces.SQL.BOOLEAN`
DATE	*None*
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
INTERVAL(*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	`Interfaces.SQL.CHAR(1..N)`
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] If *U* is OCTETS, then *k* is 1 (one); Otherwise, *k* is the maximum number of octets per character in the character set *CS*.

[2] *HVN* is the name of the host variable defined to correspond to the SQL data type.

[3] Support for the SQL types CHARACTER VARYING and NUMERIC in Ada is conditional on support for at least one of Feature B211, "Module language Ada: VARCHAR and NUMERIC support" or Feature B221, "Routine language Ada: VARCHAR and NUMERIC support".

[4] In an Ada value *AV* of this type, the *length portion* of *AV* is the field of *AV* called `HVN_LENGTH`, and the *data portion* of *AV* is the field of *AV* called `HVN_DATA`.

Table 22 — Data type correspondences for C

SQL Data Type	C Data Type
SQLSTATE	`char`, with length 6
CHARACTER (*L U*) CHARACTER SET *CS*	*`unit`*, with length (*L*+1)**k* [1]
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	*`unit`*, with length (*L*+1)**k* [1]

CHARACTER LARGE OBJECT (*L U*)
CHARACTER SET *CS*
	`struct {`
	`  long hvn_reserved;`
	`  unsigned long hvn_length;`
	`  unit hvn_data[L * k];`
	`} hvn;`
	1 2 3

BINARY (*L*)	`char`, with length *L*

BINARY VARYING (*L*)	`struct {`
	`  long hvn_reserved`
	`  unsigned long hvn_length`
	`  char hvn_data[L];`
	`} hvn`
	2 3

BINARY LARGE OBJECT (*L*)	`struct {`
	`  long hvn_reserved`
	`  unsigned long hvn_length`
	`  char hvn_data[L];`
	`} hvn`
	2 3

NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	*None*
SMALLINT	pointer to `short`
INTEGER	pointer to `long`
BIGINT	pointer to `long long`
DECFLOAT(*P*)	*None*
FLOAT (*P*)	*None*
REAL	pointer to `float`
DOUBLE PRECISION	pointer to `double`
BOOLEAN	pointer to `long`
DATE	*None*

SQL Data Type	C Data Type
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	`char`, with length *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] If *U* is OCTETS, then *unit* is `char` and *k* is 1 (one). Otherwise (that is, if *U* is CHARACTERS) then *unit* is an appropriate implementation-defined (IV230) C data type (typically `char`, `unsigned char`, `unsigned short`, or `unsigned int`) and *k* is an appropriate sizing factor given the choice of *unit*. The choice of *unit* and *k* is implementation-defined (IV230), based on the character set *CS*. For example, for UTF32, *unit* might be `char` and *k* might be 4; or *unit* might be `unsigned int` and *k* might be 1 (one).

[2] *hvn* is the name of the host variable defined to correspond to the SQL data type.

[3] In a C value *CV* of this type, the *length portion* of *CV* is the field of *CV* called *hvn*`_length`, and the *data portion* of *CV* is the field of *CV* called *hvn*`_data`.

Table 23 — Data type correspondences for COBOL

SQL Data Type	COBOL Data Type
SQLSTATE	`PICTURE X(5)`
CHARACTER (*L U*) CHARACTER SET *CS*	`PICTURE X(`*L*`)` [3]
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	*None*

CHARACTER LARGE OBJECT (*L U*)
CHARACTER SET *CS*
	`01` *hvn*`.`
	`  49` *hvn*`-RESERVED PIC S9(9) USAGE IS BINARY.`
	`  49` *hvn*`-LENGTH PIC S9(9) USAGE IS BINARY.`
	`  49` *hvn*`-DATA PIC X(`*L*`).`
	[2] [3] [4]

BINARY (*L*)	`PICTURE X(`*L*`)`
BINARY VARYING (*L*)	*None*

SQL Data Type	COBOL Data Type

BINARY LARGE OBJECT (*L*)	`01 hvn.`
	`  49 hvn-RESERVED PIC S9(9) USAGE IS BINARY.`
	`  49 hvn-LENGTH PIC S9(9) USAGE IS BINARY.`
	`  49 hvn-DATA PIC X(L).`
	2 4

NUMERIC (*P*,*S*)	`USAGE DISPLAY SIGN LEADING SEPARATE`, with `PICTURE` as specified [1]
DECIMAL(*P*,*S*)	*None*
SMALLINT	`PICTURE S9(SPI) USAGE BINARY`, where *SPI* is implementation-defined (IV231)
INTEGER	`PICTURE S9(PI) USAGE BINARY`, where *PI* is implementation-defined (IV231)
BIGINT	`PICTURE S9(BPI) USAGE BINARY`, where *BPI* is implementation-defined (IV231)
DECFLOAT(*P*)	*None*
FLOAT (*P*)	*None*
REAL	*None*
DOUBLE PRECISION	*None*
BOOLEAN	`PICTURE X`
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	alphanumeric with length *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] Case:

a) If *S*=*P*, then a PICTURE with an 'S' followed by a 'V' followed by *P* '9's.

b) If *P* > *S* > 0, then a PICTURE with an 'S' followed by *P*–*S* '9's followed by a 'V' followed by *S* '9's.

SQL Data Type	COBOL Data Type
c) If *S*=0, then a PICTURE with an 'S' followed by *P* '9's optionally followed by a 'V'. [2] *hvn* is the name of the host variable defined to correspond to the SQL data type [3] If *U* is CHARACTERS, then for the implementation-defined (IV231) COBOL native character set, `PICTURE X(L)` is used; for any other character set, `PICTURE N(L)` is used with a `LOCALE` phrase that specifies the appropriate locale name for that character set. [4] In a COBOL value *CV* of this type, the *length portion* of *CV* is the field of *CV* called *hvn*`-LENGTH`, and the *data portion* of *CV* is the field of *CV* called *hvn*`-DATA`.	

Table 24 — Data type correspondences for Fortran

SQL Data Type	Fortran Data Type
SQLSTATE	`CHARACTER`, with length 5
CHARACTER (*L U*) CHARACTER SET *CS*	`CHARACTER` [2], with length *L*
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	*None*

CHARACTER LARGE OBJECT (*L U*)
CHARACTER SET *CS*
	`CHARACTER` *hvn*`(`*L*`+8)`
	`  INTEGER*4` *hvn*`_RESERVED`
	`  INTEGER*4` *hvn*`_LENGTH`
	`  CHARACTER` *hvn*`_DATA`
	`  EQUIVALENCE(`*hvn*`(1),` *hvn*`_RESERVED)`
	`  EQUIVALENCE(`*hvn*`(5),` *hvn*`_LENGTH)`
	`  EQUIVALENCE(`*hvn*`(9),` *hvn*`_DATA)`
	1 2 3

BINARY (*L*)	`CHARACTER`, with length *L*

BINARY VARYING (*L*)	`CHARACTER` *hvn*`(`*L*`+8)`
	`  INTEGER*4` *hvn*`_RESERVED`
	`  INTEGER*4` *hvn*`_LENGTH`
	`  CHARACTER` *hvn*`_DATA`
	`  EQUIVALENCE(`*hvn*`(1),` *hvn*`_RESERVED)`
	`  EQUIVALENCE(`*hvn*`(5),` *hvn*`_LENGTH)`
	`  EQUIVALENCE(`*hvn*`(9),` *hvn*`_DATA)`
	1 3

BINARY LARGE OBJECT (*L*)	`CHARACTER` *hvn*`(`*L*`+8)`
	`  INTEGER*4` *hvn*`_RESERVED`
	`  INTEGER*4` *hvn*`_LENGTH`
	`  CHARACTER` *hvn*`_DATA`
	`  EQUIVALENCE(`*hvn*`(1),` *hvn*`_RESERVED)`
	`  EQUIVALENCE(`*hvn*`(5),` *hvn*`_LENGTH)`
	`  EQUIVALENCE(`*hvn*`(9),` *hvn*`_DATA)`
	1 3

NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	*None*

SQL Data Type	Fortran Data Type
SMALLINT	*None*
INTEGER	`INTEGER`
BIGINT	*None*
DECFLOAT(*P*)	*None*
FLOAT (*P*)	*None*
REAL	`REAL`
DOUBLE PRECISION	`DOUBLE PRECISION`
BOOLEAN	`LOGICAL`
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	`CHARACTER` with length *N*
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] *hvn* is the name of the host variable defined to correspond to the SQL data type.

[2] If *U* is CHARACTERS, then for character set UTF16, as well as other implementation-defined (IV232) character sets in which a code unit occupies more than one octet, "CHARACTER KIND=*n*" should be used; in this case, the value of *n* that corresponds to a given character set is implementation-defined (IV232). Otherwise, "CHARACTER" (without KIND) should be used.

[3] In a Fortran value *FV* of this type, the *length portion* of *FV* is the field of *FV* called *hvn*`_LENGTH`, and the *data portion* of *FV* is the field of *FV* called *hvn*`_DATA`.

Table 25 — Data type correspondences for M

SQL Data Type	M Data Type
SQLSTATE	character
CHARACTER (*L U*) CHARACTER SET *CS*	*None*

SQL Data Type	M Data Type
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	character
CHARACTER LARGE OBJECT (*L U*) CHARACTER SET *CS*	*None*
BINARY (*L*)	*None*
BINARY VARYING (*L*)	*None*
BINARY LARGE OBJECT (*L*)	*None*
NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	character
SMALLINT	*None*
INTEGER	character
BIGINT	*None*
DECFLOAT(*P*)	*None*
FLOAT (*P*)	*None*
REAL	character
DOUBLE PRECISION	*None*
BOOLEAN	*None*
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	character
ROW	*None*
ARRAY	*None*
MULTISET	*None*

Table 26 — Data type correspondences for Pascal

SQL Data Type	Pascal Data Type
SQLSTATE	`PACKED ARRAY [1..5] OF CHAR`
CHARACTER (*L U*) CHARACTER SET *CS*	`CHAR`, if *L***k* = 1 (one); otherwise, `PACKED ARRAY [1..`*L***k*`] OF CHAR` [1]
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	*None*

CHARACTER LARGE OBJECT (*L U*) CHARACTER SET *CS*	`VAR` *HVN* `= RECORD`
	HVN`_RESERVED : INTEGER;`
	HVN`_LENGTH : INTEGER;`
	HVN`_DATA : PACKED ARRAY [1..`*L***k*`] OF CHAR;`
	`END;`
	1 2 3

BINARY (*L*)	`PACKED ARRAY [1..`*L*`] OF CHAR`
BINARY VARYING (*L*)	*None*

BINARY LARGE OBJECT (*L*)	`VAR` *HVN* `= RECORD`
	HVN`_RESERVED : INTEGER;`
	HVN`_LENGTH : INTEGER;`
	HVN`_DATA : PACKED ARRAY [1..`*L*`] OF CHAR;`
	`END;`
	2 3

NUMERIC (*P*,*S*)	*None*
DECIMAL (*P*,*S*)	*None*
SMALLINT	*None*
INTEGER	`INTEGER`
BIGINT	*None*
DECFLOAT(*P*)	*None*
FLOAT (*P*)	*None*
REAL	`REAL`
DOUBLE PRECISION	*None*
BOOLEAN	`BOOLEAN`
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*

SQL Data Type	Pascal Data Type
INTERVAL (*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	`PACKED ARRAY[1..N] OF CHAR`
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] If *U* is OCTETS, then *k* is 1 (one); otherwise, *k* is the maximum number of octets per character in the character set *CS*.

[2] *HVN* is the name of the host variable defined to correspond to the SQL data type.

[3] In a Pascal value *PV* of this type, the *length portion* of *PV* is the field of *PV* called *HVN*`_LENGTH`, and the *data portion* of *PV* is the field of *PV* called *HVN*`_DATA`.

Table 27 — Data type correspondences for PL/I

SQL Data Type	PL/I Data Type
SQLSTATE	`CHARACTER(5)`
CHARACTER (*L U*) CHARACTER SET *CS*	`CHARACTER(L*k)` [2]
CHARACTER VARYING (*L U*) CHARACTER SET *CS*	`CHARACTER (L*k) VARYING` [2]

CHARACTER LARGE OBJECT (*L U*) CHARACTER SET *CS*	`DCL 01 hvn`
	`  49 hvn_reserved FIXED BINARY (31)`
	`  49 hvn_length FIXED BINARY (31)`
	`  49 hvn_data CHAR (L*k);`
	1 2 3

BINARY (*L*)	`CHARACTER (L)`
BINARY VARYING (*L*)	`CHARACTER (L) VARYING`

BINARY LARGE OBJECT (*L*)	`DCL 01 hvn`
	`  49 hvn_reserved FIXED BINARY (31)`
	`  49 hvn_length FIXED BINARY (31)`
	`  49 hvn_data CHAR (L);`
	1 3

NUMERIC(*P*,*S*)	*None*
DECIMAL (*P*,*S*)	`FIXED DECIMAL (P,S)`
SMALLINT	`FIXED BINARY(SPI)`, where *SPI* is implementation-defined (IV189)

SQL Data Type	PL/I Data Type
INTEGER	`FIXED BINARY(`*PI*`)`, where *PI* is implementation-defined (IV189)
BIGINT	`FIXED BINARY(`*BPI*`)`, where *BPI* is implementation-defined (IV189)
DECFLOAT(*P*)	*None*
FLOAT (*P*)	`FLOAT BINARY (`*P*`)`
REAL	*None*
DOUBLE PRECISION	*None*
BOOLEAN	`BIT(1)`
DATE	*None*
TIME (*T*)	*None*
TIMESTAMP (*T*)	*None*
INTERVAL (*Q*)	*None*
JSON	*None*
user-defined type	*None*
REF	`CHARACTER(`*W*`) VARYING`
ROW	*None*
ARRAY	*None*
MULTISET	*None*

[1] *hvn* is the name of the host variable defined to correspond to the SQL data type

[2] If *U* is OCTETS, then *k* is 1 (one); otherwise, *k* is the maximum number of octets per character in the character set *CS*.

[3] In a PL/I value *PV* of this type, the *length portion* of *PV* is the field of *PV* called *hvn*`_length`, and the *data portion* of *PV* is the field of *PV* called *hvn*`_data`.

Conformance Rules

None.

14 Data manipulation

This Clause is modified by Clause 13, "Data manipulation", in ISO/IEC 9075-4.
This Clause is modified by Clause 8, "Data manipulation", in ISO/IEC 9075-10.
This Clause is modified by Clause 14, "Data manipulation", in ISO/IEC 9075-14.
This Clause is modified by Clause 13, "Data manipulation", in ISO/IEC 9075-15.

14.1 <declare cursor>

This Subclause is modified by Subclause 13.1, "<declare cursor>", in ISO/IEC 9075-4.

Function

Declare a standing cursor.

Format

```
<declare cursor> ::=
  DECLARE <cursor name> <cursor properties>
      FOR <cursor specification>
```

Syntax Rules

1) 04 If a <declare cursor> is contained in an <SQL-client module definition> *M*, then:

   a) The <cursor name> shall not be equivalent to the <cursor name> of any other <declare cursor>, <dynamic declare cursor>, or <allocate received cursor statement> in *M*.

   b) The scope of the <cursor name> is *M* with the exception of any <SQL schema statement> contained in *M*.

   c) Any <host parameter name> contained in the <cursor specification> shall be defined in a <host parameter declaration> in the <externally-invoked procedure> that contains an <open statement> that specifies the <cursor name> and is contained in the scope of that <cursor name>.

NOTE 661 — See the Syntax Rules of Subclause 13.1, "<SQL-client module definition>".

Access Rules

None.

General Rules

1) A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

   a) The kind of cursor is a standing cursor.

   b) 04 The provenance of the cursor is an indication of the SQL-client module whose <SQL-client module definition> contains the <declare cursor>.

c) The name of the cursor is the <cursor name>.

d) The cursor's origin is the <cursor specification> contained in the <declare cursor>.

e) The cursor's declared properties are as determined by the <cursor properties>.

Conformance Rules

1) Without Feature F832, "Updatable scrollable cursors", conforming SQL language shall not contain a <declare cursor> that contains both a <cursor specification> that contains an <updatability clause> that specifies FOR UPDATE and <cursor properties> that contain a <cursor scrollability>.

14.2 <cursor properties>

Function

Specify the declared properties of a cursor.

Format

```
<cursor properties> ::=
  [ <cursor sensitivity> ] [ <cursor scrollability> ] CURSOR
      [ <cursor holdability> ]
      [ <cursor returnability> ]

<cursor sensitivity> ::=
    SENSITIVE
  | INSENSITIVE
  | ASENSITIVE

<cursor scrollability> ::=
    SCROLL
  | NO SCROLL

<cursor holdability> ::=
    WITH HOLD
  | WITHOUT HOLD

<cursor returnability> ::=
    WITH RETURN
  | WITHOUT RETURN
```

Syntax Rules

1) If <cursor sensitivity> is not specified, then ASENSITIVE is implicit.

2) If <cursor scrollability> is not specified, then NO SCROLL is implicit.

3) If <cursor holdability> is not specified, then WITHOUT HOLD is implicit.

4) If <cursor returnability> is not specified, then WITHOUT RETURN is implicit.

Access Rules

None.

General Rules

1) The declared properties of the cursor declaration descriptor associated with <cursor properties> are given by:

   a) The declared sensitivity property is the explicit or implicit <cursor sensitivity>.

   b) The declared scrollability property is the explicit or implicit <cursor scrollability>.

   c) The declared holdability property is the explicit or implicit <cursor holdability>.

   d) The declared returnability property is the explicit or implicit <cursor returnability>.

Conformance Rules

1) Without Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains SENSITIVE.

2) Without Feature F791, “Insensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains INSENSITIVE.

3) Without Feature F791, “Insensitive cursors”, or Feature T231, “Sensitive cursors”, conforming SQL language shall not contain a <cursor sensitivity> that immediately contains ASENSITIVE.

4) Without Feature F438, “Scrollable cursors”, conforming SQL language shall not contain a <cursor scrollability>.

5) Without Feature T471, “Result sets return value”, conforming SQL language shall not contain a <cursor returnability>.

6) Without Feature T551, “Optional key words for default syntax”, conforming SQL language shall not contain a <cursor holdability> that immediately contains WITHOUT HOLD.

14.3 <cursor specification>

Function

Define a result set.

Format

```
<cursor specification> ::=
  <query expression> [ <updatability clause> ]

<updatability clause> ::=
  FOR { READ ONLY | UPDATE [ OF <column name list> ] }
```

Syntax Rules

1) Let *CS* be the <cursor specification>.

2) Let *QE* be the <query expression> immediately contained in *CS*.

3) Case:

   a) If *CS* is contained in a <declare cursor> *DC* that contains a <cursor properties>, then let *CP* be that <cursor properties>.

   b) If *CS* is the <cursor specification> of a PTF, dynamic cursor, then the <cursor properties> of *CS* are given by

      ```
      ASENSITIVE NO SCROLL CURSOR WITHOUT HOLD WITHOUT RETURN
      ```

   c) If *CS* is a <preparable statement> being prepared by a <prepare statement> or re-prepared by an <allocate extended dynamic cursor statement> or a <dynamic open statement>, then:

      i) Let *STMT* be the <prepare statement> that is preparing *CS*, or the <allocate extended dynamic cursor statement> or <dynamic open statement> that is re-preparing *CS*. Let *SCMD* be the <SQL-client module definition> that contains *STMT*.

      ii) Case:

         1) If *CS* is being prepared by a <prepare statement>, then let *PS* be that <prepare statement>.

         2) Otherwise, let *PS* be the <prepare statement> that previously prepared *CS*.

      iii) Let *SSV* be the <SQL statement variable> contained in *PS*.

      iv) Case:

         1) If *CS* is being prepared by a <prepare statement>, then

            Case:

            A) If *SSV* is a <statement name> and there is exactly one <dynamic declare cursor> *DDC* contained in *SCMD* whose <statement name> is equivalent to *SSV*, then let *CP1* be the <cursor properties> contained in *DDC*.

            B) If *SSV* is an <extended statement name> that specifies or implies a <scope option> that is LOCAL, and there is exactly one <allocate extended dynamic cursor statement> *ACS* contained in *SCMD* whose <extended statement

name> specifies or implies LOCAL, then let *CP1* be the <cursor properties> contained in *ACS*.

C) Otherwise, let *CP1* be the zero-length character string.

2) If *CS* is being re-prepared by an <allocate extended dynamic cursor statement> *ACS*, then let *CP1* be the <cursor properties> contained in *ACS*.

3) If *CS* is being re-prepared by a <dynamic open statement>, then let *DDC* be the <dynamic declare cursor> whose <cursor name> is equivalent to the <cursor name> contained in *STMT*. Let *CP1* be the <cursor properties> contained in *DDC*.

v) If *PS* contains an <attributes variable>, then let *CP2* be the value of that <attributes variable>; otherwise, let *CP2* be the zero-length character string.

vi) Case:

1) If *CP2* contains <cursor sensitivity>, then let *SENS* be that <cursor sensitivity>.

2) If *CP1* contains <cursor sensitivity>, then let *SENS* be that <cursor sensitivity>.

3) Otherwise, let *SENS* be the zero-length character string.

vii) Case:

1) If *CP2* contains <cursor scrollability>, then let *SCRO* be that <cursor scrollability>.

2) If *CP1* contains <cursor scrollability>, then let *SCRO* be that <cursor scrollability>.

3) Otherwise, let *SCRO* be the zero-length character string.

viii) Case:

1) If *CP2* contains <cursor holdability>, then let *HOLD* be that <cursor holdability>.

2) If *CP1* contains <cursor holdability>, then let *HOLD* be that <cursor holdability>.

3) Otherwise, let *HOLD* be the zero-length character string.

ix) Case:

1) If *CP2* contains <cursor returnability>, then let *RET* be that <cursor returnability>.

2) If *CP1* contains <cursor returnability>, then let *RET* be that <cursor returnability>.

3) Otherwise, let *RET* be the zero-length character string.

x) Let *CP* be the <cursor properties>:

```
SENS SCRO CURSOR HOLD RET
```

4) If <updatability clause> is not specified and either *CS* is contained in a <declare cursor> or is being re-prepared by an <allocate extended dynamic cursor statement> or a <dynamic open statement>, then

Case:

a) If *CP* contains INSENSITIVE or SCROLL, or *QE* immediately contains an <order by clause>, or *QE* is not a simply updatable <query specification>, then an <updatability clause> of READ ONLY is implicit.

b) Otherwise, an <updatability clause> of FOR UPDATE without a <column name list> is implicit.

NOTE 662 — If *CS* is being prepared by a <prepare statement>, then defaulting the <updatability clause> is postponed until *CS* is re-prepared.

5) If an <updatability clause> of FOR UPDATE with or without a <column name list> is specified, then *CP* shall not contain INSENSITIVE, *QE* shall be effectively updatable, and *QE* shall have only one target leaf underlying table *LUT*.

6) Case:

   a) If an <updatability clause> specifying FOR UPDATE is specified or implicit, then *CS* is *updatable*.

   b) If an <updatability clause> specifying FOR READ ONLY is specified or implicit, then *CS* is *not updatable*.

   c) Otherwise, the determination of updatability of *CS* is postponed until *CS* is re-prepared.

7) If *CS* is updatable, then let *LUTN* be a <table name> that references *LUT*. *LUTN* is an exposed <table or query name> whose scope is <updatability clause>.

8) If an <updatability clause> of FOR UPDATE without a <column name list> is specified or implicit, then a <column name list> that consists of the <column name> of every column of *LUT* is implicit.

9) If an <updatability clause> of FOR UPDATE with a <column name list> is specified, then each <column name> in the <column name list> shall be the <column name> of a column of *LUT*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F833, "Updatable ordered cursors", conforming SQL language shall not contain a <cursor specification> that contains both an <updatability clause> that specifies FOR UPDATE and an <order by clause>.

14.4 <open statement>

This Subclause is modified by Subclause 13.2, "<open statement>", in ISO/IEC 9075-4.

Function

Open a standing cursor.

Format

```
<open statement> ::=
  OPEN <cursor name>
```

Syntax Rules

1) 04 Let *CN* be the <cursor name> in the <open statement>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

Access Rules

1) The Access Rules for the <query expression> simply contained in the <declare cursor> identified by the <cursor name> are applied.

General Rules

1) Let *CR* be the cursor instance descriptor in the current SQL-session whose cursor declaration descriptor is *CDD*.

2) The General Rules of Subclause 15.1, "Effect of opening a cursor", are applied with *CR* as *CURSOR*.

Conformance Rules

None.

14.5 <fetch statement>

This Subclause is modified by Subclause 13.3, "<fetch statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 8.1, "<fetch statement>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 14.1, "<fetch statement>", in ISO/IEC 9075-14.

Function

Position a standing cursor on a specified row of the standing cursor's result set and retrieve values from that row.

Format

```
10 <fetch statement> ::=
  FETCH [ [ <fetch orientation> ] FROM ] <cursor name> INTO <fetch target list>

<fetch orientation> ::=
    NEXT
  | PRIOR
  | FIRST
  | LAST
  | { ABSOLUTE | RELATIVE } <simple value specification>

<fetch target list> ::=
  <target specification> [ { <comma> <target specification> }... ]
```

Syntax Rules

1) <fetch target list> shall not contain a <target specification> that specifies a <column reference>.

2) If the <fetch orientation> is omitted, then NEXT is implicit.

3) 04 10 Let *CN* be the <cursor name> in the <fetch statement>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

4) *CN* shall identify a standing cursor.

5) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

6) Let *T* be the result set defined by the <cursor specification> of *CDD*.

7) If the implicit or explicit <fetch orientation> is not NEXT, then the declared scrollability property of *CDD* shall be SCROLL.

8) If a <fetch orientation> that contains a <simple value specification> is specified, then the declared type of that <simple value specification> shall be exact numeric with a scale of 0 (zero).

9) Case:

   a) If the <fetch target list> contains a single <target specification> *TS* and the degree of *T* is greater than 1 (one), then the declared type of *TS* shall be a row type.

      Case:

      i) 04 If *TS* is an <SQL parameter reference>, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with *TS* as *TARGET* and an arbitrary value of the row type of *T* as *VALUE*.

ii) Otherwise, the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with *TS* as *TARGET* and an arbitrary value of the row type of *T* as *VALUE*.

b) Otherwise:

i) The number of <target specification>s *NTS* in the <fetch target list> shall be the same as the degree of *T*. The *i*-th <target specification>, 1 (one) ≤ *i* ≤ *NTS*, in the <fetch target list> corresponds with the *i*-th column of *T*.

ii) For *i* varying from 1 (one) to *NTS*, let CS_i be an arbitrary value of the declared type of the *i*-th column of *T*.

iii) 04 For each <target specification> $TS1_i$, 1 (one) ≤ *i* ≤ *NTS*, that is either an <SQL parameter reference> or a <target array element specification>,

Case:

1) If $TS1_i$ contains a <simple value specification>, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with a temporary site whose declared type is the declared type of $TS1_i$ as *TARGET* and CS_i as *VALUE*.

2) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", are applied with $TS1_i$ as *TARGET* and CS_i as *VALUE*.

iv) 10 For each <target specification> $TS2_i$, 1 (one) ≤ *i* ≤ *NTS*, that is a <host parameter specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with $TS2_i$ as *TARGET* and CS_i as *VALUE*.

v) For each <target specification> $TS2_i$, 1 (one) ≤ *i* ≤ *NTS*, that is an <embedded variable specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with $TS2_i$ as *TARGET* and CS_i as *VALUE*.

Access Rules

None.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state (24000)*.

3) The General Rules of Subclause 15.3, "Determination of the current row of a cursor", are applied with *CR* as *CURSOR* and <fetch orientation> as *FETCH ORIENTATION*.

4) If a completion condition *no data (02000)* has been raised, then no further General Rules of this Subclause are applied.

5) Case:

a) If the <fetch target list> contains a single <target specification> *TS* and the degree of *T* is greater than 1 (one), then the current row is assigned to *TS* and

Case:

i) 04 If *TS* is an <SQL parameter reference>, then:

1) Let *PTEMP* be the null value.

2) 14 the General Rules of Subclause 9.2, "Store assignment", are applied with *TS* as *TARGET* and the current row as *VALUE*.

ii) Otherwise, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with *TS* as *TARGET* and the current row as *VALUE*.

b) Otherwise, if the <fetch target list> contains more than one <target specification>, then values from the current row are assigned to their corresponding targets identified by the <fetch target list>. The assignments are made in an implementation-dependent (US046) order unless otherwise specified. Let *TV* be a target and let *SV* denote its corresponding value in the current row of *CR*.

Case:

i) 04 If *TV* is either an <SQL parameter reference> or a <target array element specification>, then for each <target specification> in the <fetch target list>, let TV_i be the *i*-th <target specification> in the <fetch target list> and let SV_i denote the *i*-th corresponding value in the current row of *CR*.

Case:

1) If <target array element specification> is specified, then

Case:

A) If the value of TV_i is the null value, then an exception condition is raised: *data exception — null value in array target (2200E)*.

B) Otherwise:

I) Let *N* be the maximum cardinality of TV_i.

II) Let *M* be the cardinality of the value of TV_i.

III) Let *I* be the value of the <simple value specification> immediately contained in TV_i.

IV) Let *EDT* be the element type of TV_i.

V) Case:

1) If *I* is greater than zero and less than or equal to *M*, then the value of TV_i is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

a) For *j* varying from 1 (one) to *I*−1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in TV_i.

b) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with *I*-th element of *A* as *TARGET* and SV_i as *VALUE*.

2) If *I* is greater than *M* and less than or equal to *N*, then the value of TV_i is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

a) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in TV_i.

b) For *j* varying from *M*+1 to *I*, the *j*-th element in *A* is the null value.

c) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with *I*-th element of *A* as *TARGET* and SV_i as *VALUE*.

3) Otherwise, an exception condition is raised: *data exception — array element error (2202E)*.

2) Otherwise:

A) Let *PTEMP* be the null value.

B) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with TV_i as *TARGET* and SV_i as *VALUE*.

ii) 10 If *TV* is a <host parameter name>, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied with *TV* as *TARGET* and *SV* as *VALUE*.

iii) If *TV* is an <embedded variable specification>, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied with *TV* as *TARGET* and *SV* as *VALUE*.

NOTE 663 — SQL parameters cannot have as their data types any row type.

6) If an exception condition occurs during the assignment of a value to a target, then the values of all targets are implementation-dependent (UV106).

NOTE 664 — It is implementation-dependent whether *CR* remains positioned on the current row when an exception condition is raised during the derivation of any <derived column>.

Conformance Rules

1) Without Feature F432, "FETCH with explicit NEXT", in conforming SQL language, a <fetch statement> shall not contain a <fetch orientation> that immediately contains NEXT.

2) Without Feature F433, "FETCH FIRST", conforming SQL language shall not contain a <fetch orientation> that immediately contains FIRST.

3) Without Feature F434, "FETCH LAST", conforming SQL language shall not contain a <fetch orientation> that immediately contains LAST.

4) Without Feature F435, "FETCH PRIOR", conforming SQL language shall not contain a <fetch orientation> that immediately contains PRIOR.

5) Without Feature F436, "FETCH ABSOLUTE", conforming SQL language shall not contain a <fetch orientation> that immediately contains ABSOLUTE.

6) Without Feature F437, "FETCH RELATIVE", conforming SQL language shall not contain a <fetch orientation> that immediately contains RELATIVE.

14.6 <close statement>

This Subclause is modified by Subclause 13.4, "<close statement>", in ISO/IEC 9075-4.

Function

Close a standing cursor.

Format

```
<close statement> ::=
  CLOSE <cursor name>
```

Syntax Rules

1) 04 Let *CN* be the <cursor name> in the <close statement>. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

Access Rules

None.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) The General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

Conformance Rules

None.

14.7 <select statement: single row>

This Subclause is modified by Subclause 13.5, "<select statement: single row>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 8.2, "<select statement: single row>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 14.2, "<select statement: single row>", in ISO/IEC 9075-14.

Function

Retrieve values from a specified row of a table.

Format

```
<select statement: single row> ::=
  SELECT [ <set quantifier> ] <select list>
      INTO <select target list>
      <table expression>

<select target list> ::=
  <target specification> [ { <comma> <target specification> }... ]
```

Syntax Rules

1) <select target list> shall not contain a <target specification> that specifies a <column reference>.

2) Let *T* be the table defined by the <table expression>.

3) Case:

   a) If the <select target list> contains a single <target specification> *TS* and the degree of *T* is greater than 1 (one), then the declared type of *TS* shall be a row type.

      Case:

      i) If *TS* is an <SQL parameter reference>, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with *TS* as *TARGET* and an arbitrary value of the row type of *T* as *VALUE*.

      ii) Otherwise, the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with *TS* as *TARGET* and an arbitrary value of the row type of *T* as *VALUE*.

   b) Otherwise:

      i) The number of elements *NOE* in the <select list> shall be the same as the number of elements in the <select target list>. The *i*-th <target specification>, 1 (one) ≤ *i* ≤ *NOE*, in the <select target list> corresponds with the *i*-th element of the <select list>.

      ii) 04 For *i* varying from 1 (one) to *NOE*, let $TS1_i$ be the *i*-th <target specification> in the <select target list> that is either an <SQL parameter reference> or a <target array element specification>, and let SL_i be the *i*-th element of the <select list> that corresponds to the <target specification> in the <select target list>.

      Case:

      1) If <target array element specification> is specified, then the Syntax Rules of Subclause 9.2, "Store assignment", are applied with a temporary site whose declared type is the declared type of $TS1_i$ as *TARGET* and SL_i as *VALUE*.

2) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", are applied with $TS1_i$ as *TARGET* and the corresponding element of the <select list> as *VALUE*.

iii) For each <target specification> $TS2_i$, 1 (one) ≤ i ≤ *NOE*, that is a <host parameter specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with $TS2_i$ as *TARGET* and corresponding element of the <select list> as *VALUE*.

iv) 10 For each <target specification> $TS2_i$, 1 (one) ≤ i ≤ *NOE*, that is an <embedded variable specification>, the Syntax Rules of Subclause 9.1, "Retrieval assignment", are applied with $TS2_i$ as *TARGET* and the corresponding element of the <select list> as *VALUE*.

4) Let *S* be a <query specification> whose <select list> and <table expression> are those specified in the <select statement: single row> and that specifies the <set quantifier> if it is specified in the <select statement: single row>. *S* shall be a valid <query specification>.

5) Let *STL* be the <select target list>. Let *N* be the number of columns in the result of *S*. Let *CORR* and ID_i, for 1 (one) ≤ i ≤ *N* be implementation-defined identifiers that are all distinct from one another and distinct from every identifier in *STL* and in *S*. <select statement: single row> is equivalent to:

```
SELECT ID1, ID2, ..., IDn INTO STL
FROM ( S ) AS CORR ( ID1, ID2, ..., IDn )
```

NOTE 665 — This transformation separates the <select statement: single row> into one part containing the INTO clause, and another part containing the constructed <query specification> *S*, which is subject to additional syntactic transformations.

Access Rules

None.

General Rules

1) Let *Q* be the result of <query specification> *S*.

2) Case:

a) If the cardinality of *Q* is greater than 1 (one), then an exception condition is raised: *cardinality violation (21000)*. It is implementation-dependent (UA062) whether or not SQL-data values are assigned to the targets identified by the <select target list>.

b) If *Q* is empty, then no SQL-data values are assigned to any targets identified by the <select target list>, and a completion condition is raised: *no data (02000)*.

c) Otherwise, values in the row of *Q* are assigned to their corresponding targets.

3) If a completion condition *no data (02000)* has been raised, then no further General Rules of this Subclause are applied.

4) Case:

a) If the <select target list> contains a single <target specification> *TS* and the degree of table *T* is greater than 1 (one), then the current row is assigned to *TS* and

Case:

i) If *TS* is an <SQL parameter reference>, then:

1) Let *PTEMP* be the null value.

2) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with *TS* as *TARGET* and the current row as *VALUE*.

ii) Otherwise, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with *TS* as *TARGET* and the current row as *VALUE*.

b) Otherwise:

i) Let *NOE* be the number of elements in the <select list>.

ii) 04 For *i* varying from 1 (one) to *NOE*, let TS_i be the *i*-th <target specification> in the <select target list> that is either an <SQL parameter reference> or a <target array element specification>, and let SL_i denote the corresponding (*i*-th) value in the row of *Q*. The assignment of values to targets in the <select target list> is in an implementation-dependent (US046) order.

Case:

1) If TS_i is a <target array element specification>, then

Case:

A) If the value of TS_i is the null value, then an exception condition is raised: *data exception — null value in array target (2200E)*.

B) Otherwise:

I) Let *N* be the maximum cardinality of TS_i.

II) Let *M* be the cardinality of the value of TS_i.

III) Let *I* be the value of the <simple value specification> immediately contained in TS_i.

IV) Let *EDT* be the element type of TS_i.

V) Case:

1) If *I* is greater than zero and less than or equal to *M*, then the value of TS_i is replaced by an array *A* with element type *EDT* and cardinality *M* derived as follows:

a) For *j* varying from 1 (one) to *I*−1 and from *I*+1 to *M*, the *j*-th element in *A* is the value of the *j*-th element in TS_i.

b) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with *I*-th element of *A* as *TARGET* and SL_i as *VALUE*.

2) If *I* is greater than *M* and less than or equal to *N*, then the value of TS_i is replaced by an array *A* with element type *EDT* and cardinality *I* derived as follows:

a) For *j* varying from 1 (one) to *M*, the *j*-th element in *A* is the value of the *j*-th element in TS_i.

b) For *j* varying from *M*+1 to *I*−1, the *j*-th element in *A* is the null value.

c) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with *I*-th element of *A* as *TARGET* and SL_i as *VALUE*.

3) Otherwise, an exception condition is raised: *data exception — array element error (2202E)*.

2) Otherwise:

A) Let *PTEMP* be the null value.

B) 14 The General Rules of Subclause 9.2, "Store assignment", are applied with TS_i as *TARGET* and corresponding value SL_i in the row of *Q* as *VALUE*.

iii) For each <target specification> *TS* that is a <host parameter specification>, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with *TS* as *TARGET* and the corresponding value in the row of *Q* as *VALUE*. The assignment of values to targets in the <select target list> is in an implementation-dependent (US046) order.

iv) 10 For each <target specification> *TS* that is an <embedded variable specification>, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with *TS* as *TARGET* and the corresponding value in the row of *Q* as *VALUE*. The assignment of values to targets in the <select target list> is in an implementation-dependent (US046) order.

5) If an exception condition is raised during the assignment of a value to a target, then the values of all targets are implementation-dependent (UV106).

Conformance Rules

None.

14.8 <delete statement: positioned>

This Subclause is modified by Subclause 13.6, "<delete statement: positioned>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 8.3, "<delete statement: positioned>", in ISO/IEC 9075-10.

Function

Delete a row of a table.

Format

```
10 <delete statement: positioned> ::=
  DELETE FROM <target table> [ [ AS ] <correlation name> ]
      WHERE CURRENT OF <cursor name>

<target table> ::=
    <table name>
  | ONLY <left paren> <table name> <right paren>
```

Syntax Rules

1) 04 Let *DSP* be the <delete statement: positioned> and let *CN* be the <cursor name> immediately contained in *DSP*. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

4) The cursor specification of *CDD* shall be updatable.

5) Let *TU* be the simply underlying table specification of the cursor identified by *CN*. Let *LUT* be the target leaf underlying table of *TU*.

6) Let *TT* be the <target table> and let *TN* be the <table name> contained in *TT*. *TN* shall identify *LUT*.

7) *LUT* shall not be an old transition table or a new transition table.

8) If *TT* immediately contains ONLY and *LUT* is not a typed table, then *TT* is equivalent to *TN*.

9) *TT* shall specify ONLY if and only if the <table reference> contained in *TU* that references *LUT* specifies ONLY.

10) The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *LUT*.

11) 10 Case:

    a) If <correlation name> is specified, then let *COR* be that <correlation name>. *COR* is an exposed <correlation name>. The associated column list and the associated period list of *COR* are empty.

    b) Otherwise, let *COR* be *TN*. *COR* is an exposed <table or query name>. The associated column list and the associated period list of *COR* are empty.

NOTE 666 — *COR* has no scope.

Access Rules

1) Case:

   a) If *DSP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the authorization identifier that owns that schema. The applicable privileges for *A* shall include DELETE for the table identified by *TN*.

   b) Otherwise, the current privileges shall include DELETE for the table identified by *TN*.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) The General Rules of Subclause 15.6, "Effect of a positioned delete", are applied with *CR* as *CURSOR*, *DSP* as *STATEMENT*, and *TT* as *TARGET*.

Conformance Rules

1) Without Feature S111, "ONLY in query expressions", conforming SQL language shall not contain a <target table> that contains ONLY.

14.9 <delete statement: searched>

This Subclause is modified by Subclause 14.3, "<delete statement: searched>", in ISO/IEC 9075-14.

Function

Delete rows of a table.

Format

```
14 <delete statement: searched> ::=
  DELETE FROM <target table>
      [ FOR PORTION OF <application time period name>
       FROM <point in time 1> TO <point in time 2> ]
      [ [ AS ] <correlation name> ]
      [ WHERE <search condition> ]
```

Syntax Rules

1) Let *DSS* be the <delete statement: searched> and let *TT* be the <target table>.

2) Let *TN* be the <table name> contained in *TT*. Let *T* be the table identified by *TN*.

3) *T* shall be an effectively updatable table or a trigger deletable table.

4) *T* shall not be an old transition table or a new transition table.

5) If WHERE <search condition> is not specified, then WHERE TRUE is implicit.

6) Let *DSC* be the implicit or explicit <search condition>. *DSC* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

   Case:

   a) If *T* is a system-versioned table, then let *ENDCOL* be the system-time period end column of *T*. Let *ENDVAL* be the highest value supported by the declared type of *ENDCOL*. Let *TSC* be

      (*DSC*) AND (*ENDCOL* = *ENDVAL*)

   b) Otherwise, let *TSC* be *DSC*.

7) Case:

   a) If FOR PORTION OF <application time period name> *ATPN* is specified, then the table descriptor of *T* shall include a period descriptor whose period name is equivalent to *ATPN*.

      i) Let *BSTARTCOL* be the name of the *ATPN* period start column of *T*; let *BENDCOL* be the name of the *ATPN* period end column of *T*.

      ii) Let *FROMVAL* be <point in time 1>. *FROMVAL* shall not generally contain a reference to a column of *T* or a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

      iii) Let *TOVAL* be <point in time 2>. *TOVAL* shall not generally contain a reference to a column of *T* or a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

      iv) Let *SC* be

```
TSC AND
(FROMVAL < TOVAL) AND
(BENDCOL > FROMVAL) AND
(BSTARTCOL < TOVAL)
```

b) Otherwise, let *SC* be *TSC*.

8) If *DSS* is contained in a <triggered SQL statement>, then *SC* shall not contain a <value specification> that specifies a parameter reference.

9) Case:

a) If <correlation name> is specified, then let *CN* be that <correlation name>. *CN* is an exposed <correlation name>.

b) Otherwise, let *CN* be *TN*. *CN* is an exposed <table or query name>.

10) [14] The scope of *CN* is *SC*. The associated column list of *CN* comprises every column of *T*. The associated period list of *CN* comprises every period of *T*.

Access Rules

1) Case:

a) If *DSS* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema.

i) The applicable privileges for *A* shall include DELETE for *T*.

ii) If *TT* immediately contains ONLY, then the applicable privileges for *A* shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

b) Otherwise,

i) The current privileges shall include DELETE for *T*.

ii) If *TT* immediately contains ONLY, then the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

Case:

a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined (IA233).

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

5) *SC* is effectively evaluated for each row of *T* with the exposed <correlation name>s or <table or query name>s bound to that row.

6) Case:

   a) If *TT* contains ONLY, then the rows for which the result of *SC* is *True* and for which there is no subrow in a proper subtable of *T* are identified for deletion from *T*.

   b) Otherwise, the rows for which the result of *SC* is *True* are identified for deletion from *T*.

   NOTE 667 — Identifying a row for deletion is an implementation-dependent mechanism.

7) Let *S* be the set consisting of every row identified for deletion from *T*. *S* is the *old delta table of delete operation* on *T*. If FOR PORTION OF is specified, then *FROMVAL* and *TOVAL* are associated with every row in *S* as the *associated for portion of from-value* and the *associated for portion of to-value*, respectively.

8) Case:

   a) If *T* is a base table, then:

      i) Case:

         1) If *TT* specifies ONLY, then *T* is *identified for deletion processing without subtables*.

         2) Otherwise, *T* is *identified for deletion processing with subtables*.

         NOTE 668 — Identifying a base table for deletion processing, with or without subtables, is an implementation-dependent mechanism.

      ii) The General Rules of Subclause 15.8, "Effect of deleting rows from base tables", are applied.

   b) If *T* is a viewed table, then the General Rules of Subclause 15.10, "Effect of deleting some rows from a viewed table", are applied with *TT* as *VIEW NAME*.

9) If any row that is marked for deletion by *DSS* has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

10) If no rows are marked for deletion, then a completion condition is raised: *no data (02000)*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <delete statement: searched> in which at least one of the following is true:

   a) A leaf generally underlying table of *T* is a leaf generally underlying table of any <query expression> broadly contained in the <search condition>.

b) The <search condition> broadly contains a <routine invocation>, <method invocation>, <static method invocation>, or <method reference> whose subject routine is an external routine that possibly reads SQL-data.

2) [14] Without Feature T181, "Application-time period tables", in conforming SQL language, a <delete statement: searched> shall not contain FOR PORTION OF.

14.10 <truncate table statement>

Function

Delete all rows of a base table without causing any triggered action.

Format

```
<truncate table statement> ::=
  TRUNCATE TABLE <target table> [ <identity column restart option> ]

<identity column restart option> ::=
    CONTINUE IDENTITY
  | RESTART IDENTITY
```

Syntax Rules

1) Let *TTS* be the <truncate table statement> and let *TT* be the <target table> contained in *TTS*.

2) Let *TN* be the <table name> contained in *TT* and let *T* be the table identified by *TN*. The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *T*.

3) *T* shall be a base table and shall not be a system-versioned table.

4) *T* shall not be identified by the name of the referenced table in any referential constraint descriptor.

5) If <identity column restart option> is not specified, then CONTINUE IDENTITY is implicit.

Access Rules

1) Let *A* be the <authorization identifier> that owns the schema identified by the <schema name> of *T*.

2) The enabled authorization identifiers shall include *A*.

General Rules

1) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined (IA233).

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

5) Case:

   a) If *TT* contains ONLY, then the rows for which there is no subrow in a proper subtable of *T* are deleted from *T*.

   b) Otherwise, all rows are deleted from *T*.

6) If any row that is deleted from *T* by *TTS* has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

7) If no rows are deleted from *T*, then a completion condition is raised: *no data (02000)*.

8) If RESTART IDENTITY is specified and the table descriptor of *T* includes a column descriptor *IDCD* of an identity column, then:

   a) Let *CN* be the column name included in *IDCD* and let *SV* be the start value included in *IDCD*.

   b) The following <alter table statement> is effectively executed without further Access Rule checking:

```
ALTER TABLE TN ALTER COLUMN CN RESTART WITH SV
```

Conformance Rules

1) Without Feature F200, "TRUNCATE TABLE statement", conforming SQL language shall not contain a <truncate table statement>.

2) Without Feature F202, "TRUNCATE TABLE: identity column restart option", conforming SQL language shall not contain an <identity column restart option>.

14.11 <insert statement>

This Subclause is modified by Subclause 14.4, "<insert statement>", in ISO/IEC 9075-14.

Function

Create new rows in a table.

Format

```
14 <insert statement> ::=
  INSERT INTO <insertion target> <insert columns and source>

<insertion target> ::=
  <table name>

<insert columns and source> ::=
    <from subquery>
  | <from constructor>
  | <from default>

<from subquery> ::=
  [ <left paren> <insert column list> <right paren> ]
      [ <override clause> ]
      <query expression>

<from constructor> ::=
  [ <left paren> <insert column list> <right paren> ]
      [ <override clause> ]
      <contextually typed table value constructor>

<override clause> ::=
    OVERRIDING USER VALUE
  | OVERRIDING SYSTEM VALUE

<from default> ::=
  DEFAULT VALUES

<insert column list> ::=
  <column name list>
```

Syntax Rules

1) Let *IS* be the <insert statement>.

2) Let *TN* be the <table name> contained in <insertion target>. Let *T* be the table identified by *TN*.

3) *T* shall be an insertable-into table or a trigger insertable-into table.

4) *T* shall not be an old transition table or a new transition table.

5) For each leaf generally underlying table of *T* whose descriptor includes a user-defined type name *UDTN*, the data type descriptor of the user-defined type *UDT* identified by *UDTN* shall indicate that *UDT* is instantiable.

6) An <insert columns and source> that specifies DEFAULT VALUES is implicitly replaced by an <insert columns and source> that specifies a <contextually typed table value constructor> of the form

```
VALUES (DEFAULT, DEFAULT, ..., DEFAULT)
```


where the number of instances of "DEFAULT" is equal to the number of columns of *T*.

7) If the <insert column list> is omitted, then an <insert column list> that identifies all columns of *T* in the ascending sequence of their ordinal positions within *T* is implicit.

8) A column identified by the <insert column list> is an object column.

9) No <column name> of *T* shall be specified more than once.

10) If *T* is not trigger insertable-into, then each object column shall be an updatable column of *T*.

NOTE 669 — The notion of updatable columns of base tables is defined in Subclause 4.17, "Tables". The notion of updatable columns of viewed tables is defined in Subclause 11.32, "<view definition>".

11) For every object column *OC*:

a) Let *n* be the sequential number of the <column name> in the <insert column list> that identifies *OC*.

b) *OC* is *defaulted* if exactly one of the following is true:

i) <from default> is specified.

ii) <from constructor> *FC* is specified, and, for every <contextually typed row value expression> *CTRVE* simply contained in *FC*, *CTRVE* is a <contextually typed row value constructor> having an *n*-th <contextually typed row value constructor element> that is <default specification>.

12) Every object column of which some underlying column is a generated column shall be defaulted.

13) In the following circumstances, an <override clause> is permitted or required; if none of the following circumstances pertain, then <override clause> shall not be specified.

a) If some underlying column of an object column is a system-generated self-referencing column or a derived self-referencing column, then <override clause> shall be specified.

b) If some underlying column of an object column *OC* is an identity column whose values are always generated, and *OC* is not defaulted, then <override clause> shall be specified.

c) If some underlying column of an object column is an identity column whose values are generated by default, and <override clause> is specified, then <override clause> shall specify OVERRIDING USER VALUE.

d) If some underlying column of an object column *OC* is a system-time period start column or a system-time period end column whose values are always generated, then:

i) If *OC* is not defaulted, then <override clause> shall be specified.

ii) If <override clause> is specified, then <override clause> shall specify OVERRIDING USER VALUE.

14) If <contextually typed table value constructor> *CVC* is specified, then the data type of every <contextually typed value specification> *CVS* specified in every <contextually typed row value expression> *CRVS* contained in *CVC* is the data type *DT* indicated in the column descriptor for the positionally corresponding column in the explicit or implicit <insert column list>. If *CVS* is an <empty specification> that specifies ARRAY, then *DT* shall be an array type. If *CVS* is an <empty specification> that specifies MULTISET, then *DT* shall be a multiset type.

15) Let *QT* be the table specified by the <query expression> or <contextually typed table value constructor>. The degree of *QT* shall be equal to the number of <column name>s in the <insert column list>. The column of table *T* identified by the *i*-th <column name> in the <insert column list> corresponds with the *i*-th column of *QT*.

16) For each column of *T* identified by the <column name> in the <insert column list>, the Syntax Rules of Subclause 9.2, "Store assignment", are applied with the column as *TARGET* and the corresponding column of *QT* as *VALUE*.

17) If *IS* is contained in a <triggered SQL statement>, then <insert columns and source> shall not contain a <value specification> that specifies a parameter reference.

18) [14] A <query expression> simply contained in a <from subquery> shall not be a <table value constructor>.

NOTE 670 — This rule removes a syntactic ambiguity; otherwise, "VALUES (1)" could be parsed either as

```
<insert columns and source> ::=
   <from subquery> ::=
   <query expression> ::=
   <table value constructor> ::=
   VALUES (1)
```

or

```
<insert columns and source> ::=
   <from constructor> ::=
   <contextually typed table value constructor> ::=
   VALUES (1)
```

Access Rules

1) Case:

   a) If *IS* is contained in, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* for *T* shall include INSERT for each object column.

   b) Otherwise, the current privileges for *T* shall include INSERT for each object column.

General Rules

1) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined (IA233).

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

5) *QT* is effectively evaluated before insertion of any rows into *T*.

6) Let *Q* be the result of evaluating *QT*.

7) For each row *R* of *Q*:

   a) A candidate row of *T* is effectively created in which the value of each column is its default value, as specified in the General Rules of Subclause 11.5, "<default clause>". The candidate row consists of every column of *T*.

   b) For each object column in the candidate row, let C_i be the object column identified by the *i*-th <column name> in the <insert column list> and let SV_i be the *i*-th value of *R*.

   c) For every C_i such that C_i is not marked as unassigned and no underlying column of C_i is a self-referencing column, the General Rules of Subclause 9.2, "Store assignment", are applied with C_i as *TARGET* and SV_i as *VALUE*. C_i is no longer marked as unassigned.

   d) If *T* has a column *RC* of which some underlying column is a self-referencing column, then

      Case:

      i) If *RC* is a system-generated self-referencing column, then the value of *RC* is effectively replaced by the REF value of the candidate row.

      ii) If *RC* is a derived self-referencing column, then the value of *RC* is effectively replaced by a value derived from the columns in the candidate row that correspond to the list of derivational attributes of the derived representation of the reference type of *RC* in an implementation-dependent (UA063) manner.

   e) For every C_i for which one of the following conditions is satisifed:

      i) Some underlying column of C_i is a user-generated self-referencing column.

      ii) Some underlying column of C_i is a self-referencing column and OVERRIDING SYSTEM VALUE is specified.

      iii) Some underlying column of C_i is an identity column and the *i*-th column of *R* is not derived from <default specification> and OVERRIDING SYSTEM VALUE is specified.

      iv) Some underlying column of C_i is an identity column whose values are generated by default and neither OVERRIDING USER VALUE is specified nor is the *i*-th column derived from <default specification>.

      The General Rules of Subclause 9.2, "Store assignment", are applied with C_i as *TARGET* and SV_i as *VALUE*. C_i is no longer marked as unassigned.

   NOTE 671 — If OVERRIDING USER VALUE is specified, then some columns of the candidate row(s) can continue to be marked as unassigned as a result of the preceding rules. The value of such columns is ultimately determined by the General Rules of Subclause 15.11, "Effect of inserting tables into base tables", which has the effect of overriding user values specified in <insert columns and source>.

   NOTE 672 — The data values allowable in the candidate row can be constrained by a CHECK OPTION constraint. The effect of a CHECK OPTION constraint is defined in the General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION".

8) Let *S* be the table consisting of the candidate rows.

   Case:

   a) If *T* is a base table, then:

      i) *T* is *identified for insertion of source table S*.

         NOTE 673 — Identifying a base table for insertion of a source table is an implementation-dependent operation.

ii) The General Rules of Subclause 15.11, "Effect of inserting tables into base tables", are applied with *S* as *SOURCE* and *T* as *TARGET*

b) If *T* is a viewed table, then:

i) The General Rules of Subclause 15.13, "Effect of inserting a table into a viewed table", are applied with *S* as *SOURCE* and *T* as *TARGET*.

ii) The General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION", are applied with INSERT as *OPERATION*.

9) If *Q* is empty, then a completion condition is raised: *no data (02000)*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain an <insert statement> in which either of the following is true:

a) The <table name> of a leaf generally underlying table of *T* is broadly contained in the <from subquery> except as the table name of a qualifying table of a column reference.

b) The <from subquery> broadly contains a <routine invocation>, <method invocation>, <static method invocation>, or <method reference> whose subject routine is an external routine that possibly reads SQL-data.

2) Without Feature F222, "INSERT statement: DEFAULT VALUES clause", conforming SQL language shall not contain a <from default>.

3) Without Feature S024, "Enhanced structured types", in conforming SQL language, for each column *C* identified in the explicit or implicit <insert column list>, if the declared type of *C* is a structured type *TY*, then the declared type of the corresponding column of the <query expression> or <contextually typed table value constructor> shall be *TY*.

4) [14] Without Feature S043, "Enhanced reference types", conforming SQL language shall not contain an <override clause>.

14.12 <merge statement>

This Subclause is modified by Subclause 14.5, "<merge statement>", in ISO/IEC 9075-14.

Function

Conditionally update and/or delete rows of a table and/or insert new rows into a table.

Format

```
14 <merge statement> ::=
  MERGE INTO <target table> [ [ AS ] <merge correlation name> ]
      USING <table reference>
      ON <search condition> <merge operation specification>

<merge correlation name> ::=
  <correlation name>

<merge operation specification> ::=
  <merge when clause>...

<merge when clause> ::=
    <merge when matched clause>
  | <merge when not matched clause>

<merge when matched clause> ::=
  WHEN MATCHED [ AND <search condition> ]
      THEN <merge update or delete specification>

<merge update or delete specification> ::=
    <merge update specification>
  | <merge delete specification>

<merge when not matched clause> ::=
  WHEN NOT MATCHED [ AND <search condition> ]
      THEN <merge insert specification>

<merge update specification> ::=
  UPDATE SET <set clause list>

<merge delete specification> ::=
  DELETE

<merge insert specification> ::=
  INSERT [ <left paren> <insert column list> <right paren> ]
      [ <override clause> ]
      VALUES <merge insert value list>

<merge insert value list> ::=
  <left paren>
      <merge insert value element> [ { <comma> <merge insert value element> }... ]
      <right paren>

<merge insert value element> ::=
    <value expression>
  | <contextually typed value specification>
```

Syntax Rules

1) Let *TN* be the <table name> contained in <target table> *TT* and let *T* be the table identified by *TN*.

2) If <merge when not matched clause> is specified, then *T* shall be insertable-into or trigger insertable-into.

3) If <merge update specification> is specified, then *T* shall be effectively updatable or trigger updatable.

4) If <merge delete specification> is specified, then *T* shall be effectively updatable or trigger deletable.

5) *T* shall not be an old transition table or a new transition table.

6) For each leaf generally underlying table of *T* whose descriptor includes a user-defined type name *UDTN*, the data type descriptor of the user-defined type *UDT* identified by *UDTN* shall indicate that *UDT* is instantiable.

7) If *T* is a view, then <target table> is effectively replaced by:

```
ONLY ( TN )
```

8) Case:

   a) If <merge correlation name> is specified, then let *CN* be the <correlation name> contained in <merge correlation name>. *CN* is an exposed <correlation name>.

   b) Otherwise, let *CN* be the <table name> contained in <target table>. *CN* is an exposed <table or query name>.

9) The scope of *CN* is the <search condition> immediately contained in the <merge statement>, the <search condition> immediately contained in a <merge when matched clause>, the <search condition> immediately contained in a <merge when not matched clause>, and the <set clause list>.

10) The associated column list of *CN* comprises every column of *T*. The associated period list of *CN* comprises every period of *T*.

11) Let *TR* be the <table reference> immediately contained in <merge statement>. *TR* shall not directly contain a <joined table>.

12) The <correlation name> or <table or query name> that is exposed by *TR* shall not be equivalent to *CN*.

13) The <search condition> immediately contained in a <merge statement>, the <search condition> immediately contained in a <merge when matched clause>, and the <search condition> immediately contained in a <merge when not matched clause> shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

14) Each column identified by an <object column> in a <set clause list> is an *update object column*. Each column identified by a <column name> in an implicit or explicit <insert column list> is an *insert object column*. Each update object column and each insert object column is an *object column*.

15) If <merge when not matched clause> is specified and if *T* is not trigger insertable-into, then every insert object column shall identify an updatable column of *T*.

16) If <merge when matched clause> is specified and if *T* is not trigger updatable, then every update object column shall identify an updatable column of *T*.

    NOTE 674 — The notion of updatable columns of base tables is defined in Subclause 4.17, "Tables". The notion of updatable columns of viewed tables is defined in Subclause 11.32, "<view definition>".

17) No <column name> of *T* shall be identified more than once in an <insert column list>.

18) For each <merge when not matched clause> *MWNMC*:

   a) If an <insert column list> is omitted, then an <insert column list> that identifies all columns of *T* in the ascending sequence of their ordinal position within *T* is implicit.

b) For every insert object column *IOC* of *MWNMC*:

i) Let *n* be the sequential number of the <column name> in the <insert column list> of *MWNMC* that identifies *IOC*.

ii) If the *n*-th <merge insert value element> of *MWNMC* is <default specification>, then *IOC* is *defaulted*.

c) Let *NI* be the number of <merge insert value element>s contained in <merge insert value list> of *MWNMC*. Let EXP_1, EXP_2, ..., EXP_{NI} be those <merge insert value element>s.

d) The number of <column name>s in the <insert column list> of *MWNMC* shall be equal to *NI*.

e) The declared type of every <contextually typed value specification> *CVS* in a <merge insert value list> is the data type *DT* indicated in the column descriptor for the positionally corresponding insert object column of *MWNMC* in the explicit or implicit <insert column list>. If *CVS* is an <empty specification> that specifies ARRAY, then *DT* shall be an array type. If *CVS* is an <empty specification> that specifies MULTISET, then *DT* shall be a multiset type.

f) Every insert object column of *MWNMC* of which some underlying column is a generated column shall be defaulted.

g) For 1 (one) $\leq i \leq NI$, the Syntax Rules of Subclause 9.2, "Store assignment", are applied with EXP_i as *VALUE* and the column of table *T* identified by the *i*-th <column name> in the <insert column list> of *MWNMC* as *TARGET*.

h) In the following circumstances, an <override clause> is permitted or required; if none of the following circumstances pertain, then <override clause> shall not be specified.

i) If some underlying column of an insert object column of *MWNMC* is a system-generated self-referencing column or a derived self-referencing column, then *MWNMC* shall specify <override clause>.

ii) If some underlying column of an insert object column *IOC* of *MWNMC* is an identity column whose values are always generated and *IOC* is not defaulted, then *MWNMC* shall specify <override clause>.

iii) If some underlying column of an insert object column *IOC* of *MWNMC* is an identity column whose values are generated by default and *MWNMC* specifies <override clause>, then that <override clause> shall specify OVERRIDING USER VALUE.

iv) If some underlying column of an insert object column *IOC* of *MWNMC* is a system-time period start column or a system-time period end column whose values are always generated, then:

1) If *IOC* is not defaulted, then *MWNMC* shall specify <override clause>.

2) If *MWNMC* specifies <override clause> *OC*, then *OC* shall specify OVERRIDING USER VALUE.

19) [14] Let *DSC* be the <search condition> immediately contained in <merge statement>.

Case:

a) If *T* is a system-versioned table, then let *ENDCOL* be the system-time period end column of *T*. Let *ENDVAL* be the highest value supported by the declared type of *ENDCOL*. Let *SC1* be

```
(DSC) AND (ENDCOL = ENDVAL)
```

b) Otherwise, let *SC1* be *DSC*.

Access Rules

1) Case:

   a) If <merge statement> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema.

      i) If <merge update specification> is specified, then the applicable privileges for *A* shall include UPDATE for each update object column.

      ii) If <merge delete specification> is specified, then the applicable privileges for *A* shall include DELETE for *T*.

      iii) If <merge insert specification> is specified, then the applicable privileges for *A* shall include INSERT for each insert object column.

      iv) If *TT* immediately contains ONLY, then the applicable privileges for *A* shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

   b) Otherwise,

      i) If <merge update specification> is specified, then the current privileges shall include UPDATE for each update object column.

      ii) If <merge delete specification> is specified, then the current privileges for *A* shall include DELETE for *T*.

      iii) If <merge insert specification> is specified, then the current privileges shall include INSERT for each insert object column.

      iv) If *TT* immediately contains ONLY, then the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined (IA233).

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

5) Let *QT* be the table specified by the <table reference>. *QT* is effectively evaluated before update, deletion, or insertion of any rows in *T*. Let *Q* be the result of evaluating *QT*.

6) Let *NMWC* be the number of <merge when clause>s immediately contained in the <merge operation specification>.

a) For each <merge when clause> MWC_j, 1 (one) $\leq j \leq NMWC$, in the order specified in the <merge operation specification>,

Case:

i) If <merge when matched clause> $MWMC_j$ is specified, then:

1) For each row *R1* of *T*:

A) *SC1* and the <search condition> SC_j immediately contained in $MWMC_j$, if any, are effectively evaluated for *R1* with *CN* bound to *R1* and to each row of *Q* with the exposed <correlation name>s or <table or query name>s of the <table reference> bound to that row. Both *SC1* and SC_j are effectively evaluated for *R1* before updating or deleting any row of *T* and prior to the invocation of any <triggered action> caused by the update or deletion of any row of *T* and before inserting any rows into *T* and prior to the invocation of any <triggered action> caused by the insert of any row of *T*.

Case:

I) If *TT* contains ONLY, then *R1* is a subject row if *R1* has no subrow in a proper subtable of *T* and the result of both *SC1* and SC_j are *True* for some row *R2* of *Q* and *R1* is not a subject row identified by any other <merge when matched clause> that precedes $MWMC_j$ in the <merge operation specification>. *R2* is the matching row.

II) Otherwise, *R1* is a subject row if the result of both *SC1* and SC_j are *True* for some row *R2* of *Q* and *R1* is not a subject row identified by any other <merge when matched clause> that precedes $MWMC_j$ in the <merge operation specification>. *R2* is the matching row.

B) If *R1* is a subject row, then:

I) Let *M* be the number of matching rows in *Q* for *R1*.

II) If *M* is greater than 1 (one), then an exception condition is raised: *cardinality violation (21000)*.

III) If <merge update specification> is specified, then:

1) Let *SCL* be the <set clause list>.

2) The <update source> of each <set clause> in *SCL* is effectively evaluated for *R1* before any row of *T* is updated and prior to the invocation of any <triggered action> caused by the update of any row of *T*. The value resulting from that evaluation is the *update value*.

3) The General Rules of Subclause 15.5, "Evaluating a <set clause list>", are applied with *R1* as *SUBJECT ROW* and *SCL* as *SET CLAUSE LIST*; let *CNR* be the *CANDIDATE NEW ROW* returned from the application of those General Rules.

2) Let S_j be the set consisting of every subject row.

3) If *T* is a base table, then each subject row is also an object row; otherwise, an object row is any row of a leaf generally underlying table of *T* from which a subject row is derived.

NOTE 675 — The data values allowable in the object rows can be constrained by a CHECK OPTION constraint. The effect of a CHECK OPTION constraint is defined in the General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION".

4) If any row in the set of object rows has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

5) If <merge update specification> is specified, then:

A) Let *CL* be the columns of *T* identified by the <object column>s contained in the <set clause list>.

B) Each subject row *SR* is identified for replacement, by its corresponding candidate new row *CNR*, in *T*. The set of (*SR*, *CNR*) pairs is the replacement set for *T*.

NOTE 676 — Identifying a row for replacement, associating a replacement row with an identified row, and associating a replacement set with a table are implementation-dependent operations.

C) Case:

I) If *T* is a base table, then:

1) Case:

a) If *TT* specifies ONLY, then *T* is *identified for replacement processing without subtables* with respect to object columns *CL*.

b) Otherwise, *T* is *identified for replacement processing with subtables* with respect to object columns *CL*.

NOTE 677 — Identifying a base table for replacement processing, with or without subtables, is an implementation-dependent mechanism. In general, though not here, the list of object columns can be empty.

2) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

II) If *T* is a viewed table, then the General Rules of Subclause 15.16, "Effect of replacing some rows in a viewed table", are applied with *TT* as *VIEW NAME* and the replacement set for *T* as *REPLACEMENT SET FOR VIEW NAME*.

6) Let ND_j be a copy of the new delta table of update operation on *T*, if any. The new delta table of update operation on *T* is destroyed.

NOTE 678 — "new delta table of update operation" is defined in Subclause 15.14, "Effect of replacing rows in base tables", and Subclause 15.16, "Effect of replacing some rows in a viewed table".

7) If <merge delete specification> is specified, then:

A) Each subject row is identified for deletion from *T*.

B) Case:

I) If *T* is a base table, then:

1) Case:

a) If *TT* specifies ONLY, then *T* is *identified for deletion processing without subtables*.

b) Otherwise, *T* is *identified for deletion processing with subtables*.

NOTE 679 — Identifying a base table for deletion processing, with or without subtables, is an implementation-dependent mechanism.

2) The General Rules of Subclause 15.8, "Effect of deleting rows from base tables", are applied.

II) Otherwise, *T* is a viewed table and the General Rules of Subclause 15.10, "Effect of deleting some rows from a viewed table", are applied with *TT* as *VIEW NAME*.

ii) If <merge when not matched clause> $MWNMC_j$ is specified, then:

1) Let *TR1* be the <target table> immediately contained in <merge statement> and let *TR2* be the <table reference> immediately contained in <merge statement>. If <merge correlation name> is specified, then let *MCN* be "AS <merge correlation name>"; otherwise, let *MCN* be the zero-length character string. If $MWNMC_j$ immediately contains a <search condition> SC_j, then let $ONSC_j$ be "`OR NOT` SC_j"; otherwise, let $ONSC_j$ be the zero-length character string. Let *S1* be the result of

```
SELECT *
FROM TR1 MCN, TR2
WHERE SC1 ONSCj
```

2) Let *S2* be the collection of rows of *Q* for which there exists in *S1* some row that is the concatenation of some row *R1* of *T* and some row *R2* of *Q*.

3) Let *S3* be the collection of rows of *Q* that are not in *S2*. Let *SN3* be the effective distinct name for *S3*. Let *ER* be the exposed range variable of *TR2*. If *ER* is a <table name>, then let *EN* be the zero-length character string; otherwise, let *EN* be "AS *ER*".

4) Let *S4* be the result of:

```
SELECT EXP1, EXP2, ..., EXPNI
FROM SN3 EN
```

5) Let *S5* be the collection of rows of *S4* for which no candidate rows have been effectively created by any other <merge when not matched clause> that precedes $MWNMC_j$ in the <merge operation specification>.

6) *S5* is effectively evaluated before deletion of any rows from, insertion of any rows into, or update of any rows in *T*.

7) For each row *R* of *S5*:

A) A candidate row of *T* is effectively created in which the value of each column is its default value, as specified in the General Rules of Subclause 11.5, "<default clause>". The candidate row consists of every column of *T*.

B) If *T* has a column *RC* of which some underlying column is a self-referencing column, then

Case:

I) If *RC* is a system-generated self-referencing column, then the value of *RC* is effectively replaced by the REF value of the candidate row.

II) If *RC* is a derived self-referencing column, then the value of *RC* is effectively replaced by a value derived from the columns in the candidate row that correspond to the list of derivational attributes of the derived representation of the reference type of *RC* in an implementation-dependent (UA063) manner.

C) For each object column in the candidate row, let C_i be the object column identified by the *i*-th <column name> in the <insert column list> and let SV_i be the *i*-th value of *R*.

D) For every C_i for which at least one of the following conditions is true:

I) C_i is not marked as unassigned and no underlying column of C_i is a self-referencing column.

II) Some underlying column of C_i is a user-generated self-referencing column.

III) Some underlying column of C_i is a self-referencing column and OVERRIDING SYSTEM VALUE is specified.

IV) Some underlying column of C_i is an identity column and the *i*-th column of *R* is not derived from <default specification> and OVERRIDING SYSTEM VALUE is specified.

V) Some underlying column of C_i is an identity column whose values are generated by default and neither OVERRIDING USER VALUE is specified nor is the *i*-th column derived from <default specification>.

the General Rules of Subclause 9.2, "Store assignment", are applied with C_i as *TARGET* and SV_i as *VALUE*. C_i is no longer marked as unassigned.

NOTE 680 — If OVERRIDING USER VALUE is specified, then some columns of the candidate row(s) can continue to be marked as unassigned as a result of the preceding rules. The value of such columns is ultimately determined by the General Rules of Subclause 15.11, "Effect of inserting tables into base tables", which has the effect of overriding user values specified in <insert columns and source>.

NOTE 681 — The data values allowable in the candidate row can be constrained by a CHECK OPTION constraint. The effect of a CHECK OPTION constraint is defined in the General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION".

8) Let S_j be the table consisting of the candidate rows.

Case:

A) If *T* is a base table, then:

I) *T* is *identified for insertion of source table* S_j.

NOTE 682 — Identifying a base table for insertion of a source table is an implementation-dependent operation.

II) The General Rules of Subclause 15.11, "Effect of inserting tables into base tables", are applied.

B) If *T* is a viewed table, then the General Rules of Subclause 15.13, "Effect of inserting a table into a viewed table", are applied with *S* as *SOURCE* and *T* as *TARGET*.

9) Let ND_j be a copy of the new delta table of insert operation on *T*, if any. The new delta table of insert operation on *T* is destroyed.

NOTE 683 — "new delta table of insert operation" is defined in Subclause 15.11, "Effect of inserting tables into base tables", and Subclause 15.13, "Effect of inserting a table into a viewed table".

b) Let *ODT* be the union of all S_j, 1 (one) $\le j \le$ *NMWC*, where MWC_j immediately contains a <merge when matched clause>. *ODT* is the *old delta table of merge operation* on *T*.

c) Let *NDT* be the union of all ND_j, 1 (one) $\le j \le$ *NMWC*, where MWC_j immediately contains either a <merge when matched clause> that specifies UPDATE or a <merge when not matched clause>. *NDT* is the *new delta table of merge operation* on *T*.

7) The General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION", are applied with MERGE as *OPERATION*

8) If *Q* is empty, then a completion condition is raised: *no data (02000)*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <merge statement> in which either of the following is true:

a) A leaf generally underlying table of *T* is broadly contained in a <query expression> immediately contained in the <table reference> except as the <table or query name> or <correlation name> of a column reference.

b) A <query expression> immediately contained in the <table reference> broadly contains a <routine invocation>, <method invocation>, <static method invocation>, or <method reference> whose subject routine is an external routine that possibly reads SQL-data.

2) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <merge statement> in which either of the following is true:

a) A leaf generally underlying table of *T* is a leaf generally underlying table of any <query expression> broadly contained in any <search condition>.

b) Any <search condition> broadly contains a <routine invocation>, <method invocation>, <static method invocation>, or <method reference> whose subject routine is an external routine that possibly reads SQL-data.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <merge statement> that does not satisfy the condition: for each column *C* identified in the explicit or implicit <insert column list>, if the declared type of *C* is a structured type *TY*, then the declared type of the corresponding column of the <query expression> or <contextually typed table value constructor> is *TY*.

4) Without Feature F312, "MERGE statement", conforming SQL language shall not contain a <merge statement>.

5) Without Feature F313, “Enhanced MERGE statement”, in conforming SQL language, a <merge statement> shall not contain each of <merge when matched clause> and <merge when not matched clause> more than once.

6) Without Feature F313, “Enhanced MERGE statement”, in conforming SQL language, a <merge when matched clause> or a <merge when not matched clause> shall not immediately contain a <search condition>.

7) [14] Without Feature F314, “MERGE statement with DELETE branch”, in conforming SQL language, a <merge when matched clause> shall not immediately contain a <merge delete specification>.

14.13 <update statement: positioned>

This Subclause is modified by Subclause 13.7, "<update statement: positioned>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 8.4, "<update statement: positioned>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 14.6, "<update statement: positioned>", in ISO/IEC 9075-14.

Function

Update a row of a table.

Format

```
10 14 <update statement: positioned> ::=
  UPDATE <target table> [ [ AS ] <correlation name> ]
      SET <set clause list>
      WHERE CURRENT OF <cursor name>
```

Syntax Rules

1) 04 Let *USP* be the <update statement: positioned> and let *CN* be the <cursor name> immediately contained in *USP*. *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*.

2) *CN* shall identify a standing cursor.

3) Let *CDD* be the cursor declaration descriptor of the standing cursor identified by *CN*.

4) The cursor specification of *CDD* shall be updatable.

5) Let *TU* be the simply underlying table specification of the cursor identified by *CN*. Let *LUT* be the target leaf underlying table of *TU*.

6) Let *TT* be the <target table> and let *TN* be the <table name> contained in *TT*. *TN* shall identify *LUT*.

7) *LUT* shall not be an old transition table or a new transition table.

8) It *TT* immediately contains ONLY and *LUT* is not a typed table, then *TT* is equivalent to *TN*.

9) *TT* shall specify ONLY if and only if the <table reference> contained in *TU* that references *LUT* specifies ONLY.

10) The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *LUT*.

11) Case:

    a) If <correlation name> is specified, then let *COR* be that <correlation name>. *COR* is an exposed <correlation name>.

    b) Otherwise, let *COR* be the <table name> contained in *TT*. *COR* is an exposed <table or query name>.

12) The scope of *COR* is <set clause list>. The associated column list of *COR* comprises every column of *LUT*. The associated period list of *COR* comprises every period of *LUT*.

13) If the declared <cursor specification> *CS* of *CDD* is ordered, then for each <object column> *OC* contained in <set clause list>, no generally underlying column of a <sort key> in the <order by clause>

simply contained in the <query expression> of *CS* shall be *OC* or a generally underlying column of *OC*.

14) 10 14 Each <column name> specified as an <object column> shall identify a column in the explicit or implicit <column name list> contained in the explicit or implicit <updatability clause> of the <cursor specification> of *CDD*.

Access Rules

1) Case:

   a) If *USP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* shall include UPDATE for each <object column>.

   b) Otherwise, the current privileges shall include UPDATE for each <object column>.

General Rules

1) Let *CR* be the cursor instance descriptor of the current SQL-session whose cursor declaration descriptor is *CDD*.

2) Let *SCL* be the <set clause list>.

3) The General Rules of Subclause 15.7, "Effect of a positioned update", are applied with *CR* as *CURSOR*, *SCL* as *SET CLAUSE LIST*, *USP* as *STATEMENT*, and *TT* as *TARGET*.

Conformance Rules

1) 14 Without Feature F833, "Updatable ordered cursors", conforming SQL language shall not contain an <update statement: positioned> in which the declared <cursor specification> of *CDD* is ordered.

14.14 <update statement: searched>

This Subclause is modified by Subclause 14.7, "<update statement: searched>", in ISO/IEC 9075-14.

Function

Update rows of a table.

Format

```
14 <update statement: searched> ::=
  UPDATE <target table>
      [ FOR PORTION OF <application time period name>
       FROM <point in time 1> TO <point in time 2> ]
      [ [ AS ] <correlation name> ]
      SET <set clause list>
      [ WHERE <search condition> ]
```

Syntax Rules

1) Let *USS* be the <update statement: searched>, let *TT* be the <target table> contained in *USS*, and let *SCL* be the <set clause list> contained in *USS*.

2) Let *TN* be the <table name> contained in *TT* and let *T* be the table identified by *TN*.

3) *T* shall be an effectively updatable table or a trigger updatable table.

4) *T* shall not be an old transition table or a new transition table.

5) If WHERE is not specified, then WHERE TRUE is implicit.

6) Let *DSC* be the implicit or explicit <search condition>. *DSC* shall not generally contain a <routine invocation> whose subject routine is an SQL-invoked routine that possibly modifies SQL-data.

   Case:

   a) If *T* is a system-versioned table, then let *ENDCOL* be the system-time period end column of *T*. Let *ENDVAL* be the highest value supported by the declared type of *ENDCOL*. Let *TSC* be

      ```
      (DSC) AND (ENDCOL = ENDVAL)
      ```

   b) Otherwise, let *TSC* be *DSC*.

7) Case:

   a) If FOR PORTION OF <application time period name> *ATPN* is specified, then the table descriptor of *T* shall include an *ATPN* period descriptor.

      i) Let *BSTARTCOL* be the name of the *ATPN* period start column of *T*; let *BENDCOL* be the name of the *ATPN* period end column of *T*. Let *BCD* be the declared type of the *ATPN* period start column of *T*.

      ii) Neither *BSTARTCOL* nor *BENDCOL* shall be an explicit <object column> contained in the <set clause list>.

      iii) Let *FROMVAL* be <point in time 1>. *FROMVAL* shall not generally contain a reference to a column of *T* or a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

iv) Let *TOVAL* be <point in time 2>. *TOVAL* shall not generally contain a reference to a column of *T* or a <routine invocation> whose subject routine is an SQL-invoked routine that is possibly non-deterministic or that possibly modifies SQL-data.

v) Let *SC* be

```
TSC AND
(FROMVAL < TOVAL) AND
(BENDCOL > FROMVAL) AND
(BSTARTCOL < TOVAL)
```

vi) The following two <set clause>s are implicitly added to *SCL*:

```
BSTARTCOL = CASE
              WHEN BSTARTCOL > FROMVAL
                THEN BSTARTCOL
              ELSE CAST ( FROMVAL AS BCD )
            END,
BENDCOL   = CASE
              WHEN BENDCOL < TOVAL
                THEN BENDCOL
              ELSE CAST ( TOVAL AS BCD )
            END
```

b) Otherwise, let *SC* be *TSC*.

8) If *USS* is contained in a <triggered SQL statement>, then *SC* shall not contain a <value specification> that specifies a parameter reference.

9) Case:

a) If <correlation name> is specified, then let *CN* be that <correlation name>. *CN* is an exposed <correlation name>.

b) Otherwise, let *CN* be the <table name> contained in *TT*. *CN* is an exposed <table or query name>.

10) [14] The scope of *CN* is *SCL* and *SC*. The associated column list of *CN* comprises every column of *T*. The associated period list of *CN* comprises every period of *T*.

Access Rules

1) Case:

a) If *USS* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema.

i) The applicable privileges for *A* for *T* shall include UPDATE for each <object column>.

ii) If *TT* immediately contains ONLY, then the applicable privileges for *A* shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

b) Otherwise,

i) The current privileges for *T* shall include UPDATE for each <object column>.

ii) If *TT* immediately contains ONLY, then the current privileges shall include SELECT WITH HIERARCHY OPTION on at least one supertable of *T*.

General Rules

1) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only and *T* is not a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

2) If there is any sensitive cursor *CR* that is currently open in the SQL-transaction in which this SQL-statement is being executed, then

   Case:

   a) If *CR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of this statement shall be made visible to *CR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

   b) Otherwise, whether the change resulting from the successful execution of this SQL-statement is made visible to *CR* is implementation-defined (IA233).

3) If there is any open, insensitive cursor *CR*, then either the change resulting from the successful execution of this statement shall be invisible to *CR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

4) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

5) Case:

   a) If *TT* contains ONLY, then *SC* is effectively evaluated for each row of *T* with the exposed <correlation name>s or <table or query name>s of *TT* bound to that row, and the subject rows are those rows for which the result of *SC* is *True* and for which there is no subrow in a proper subtable of *T*. *SC* is effectively evaluated for each row of *T* before updating any row of *T*.

   b) Otherwise, *SC* is effectively evaluated for each row of *T* with the exposed <correlation name>s or <table or query name>s of *TT* bound to that row, and the subject rows are those rows for which the result of *SC* is *True*. *SC* is effectively evaluated for each row of *T* before updating any row of *T*.

6) Let *S* be the set consisting of every subject row. *S* is the *old delta table of update operation* on *T*. If FOR PORTION OF is specified, then *FROMVAL* and *TOVAL* are associated with every row in *S* as the *associated for portion of from-value* and the *associated for portion of to-value*, respectively.

7) If *T* is a base table, then each subject row is also an *object row*; otherwise, an *object row* is any row of a leaf generally underlying table of *T* from which a subject row is derived.

8) If any row in the set of object rows has been marked for deletion by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

9) *SC* is evaluated for each row of *T* prior to the invocation of any <triggered action> caused by the update of any row of *T*.

10) The <update source> of each <set clause> contained in *SCL* is effectively evaluated for each row of *T* before any row of *T* is updated.

11) Let *N* be the cardinality of *S*. For each subject row SR_i, 1 (one) ≤ *i* ≤ *N*, in *S*:

a) The General Rules of Subclause 15.5, "Evaluating a <set clause list>", are applied with SR_i as *SUBJECT ROW* and *SCL* as *SET CLAUSE LIST*; let CNR_i be the *CANDIDATE NEW ROW* returned from the application of those General Rules.

b) SR_i is identified for replacement in *T* by CNR_i.

NOTE 684 — The data values allowable in the object rows can be constrained by a CHECK OPTION constraint. The effect of a CHECK OPTION constraint is defined in the General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION".

12) Let *RS* be the set of (SR_i, CNR_i) pairs, 1 (one) ≤ i ≤ N. *RS* is the *replacement set* for *T*.

NOTE 685 — Identifying a row for replacement, associating a replacement row with an identified row, and associating a replacement set with a table are implementation-dependent operations.

13) Let *CL* be the columns of *T* identified by the <object column>s contained in *SCL*.

14) Case:

a) If *T* is a base table, then:

i) Case:

1) If *TT* specifies ONLY, then *T* is *identified for replacement processing without subtables* with respect to object columns *CL*.

2) Otherwise, *T* is *identified for replacement processing with subtables* with respect to object columns *CL*.

NOTE 686 — Identifying a base table for replacement processing, with or without subtables, is an implementation-dependent mechanism. In general, though not here, the list of object columns can be empty.

ii) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

b) If *T* is a viewed table, then:

i) The General Rules of Subclause 15.16, "Effect of replacing some rows in a viewed table", are applied with *TT* as *VIEW NAME* and the replacement set for *T* as *REPLACEMENT SET FOR VIEW NAME*.

ii) The General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION", are applied with UPDATE as *OPERATION*.

15) If the set of object rows is empty, then a completion condition is raised: *no data (02000)*.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain an <update statement: searched> in which either of the following is true:

a) A leaf generally underlying table of *T* is a leaf generally underlying table of any <query expression> broadly contained in the <search condition>.

b) The <search condition> broadly contains a <routine invocation>, <method invocation>, <static method invocation>, or <method reference> whose subject routine is an external routine that possibly reads SQL-data.

2) [14] Without Feature T181, "Application-time period tables", in conforming SQL language, an <update statement: searched> shall not contain FOR PORTION OF.

14.15 <set clause list>

This Subclause is modified by Subclause 13.1, "<set clause list>", in ISO/IEC 9075-15.

Function

Specify a list of updates.

Format

```
<set clause list> ::=
  <set clause> [ { <comma> <set clause> }... ]

<set clause> ::=
    <multiple column assignment>
  | <set target> <equals operator> <update source>

<set target> ::=
    <update target>
  | <mutated set clause>

<multiple column assignment> ::=
  <set target list> <equals operator> <assigned row>

<set target list> ::=
  <left paren> <set target> [ { <comma> <set target> }... ] <right paren>

<assigned row> ::=
  <contextually typed row value expression>

15 <update target> ::=
    <object column>
  | <object column>
       <left bracket or trigraph> <simple value specification> <right bracket or trigraph>

<object column> ::=
  <column name>

<mutated set clause> ::=
  <mutated target> <period> <method name>

<mutated target> ::=
    <object column>
  | <mutated set clause>

<update source> ::=
    <value expression>
  | <contextually typed value specification>
```


Syntax Rules

1) Let *T* be the table identified by the <target table> contained in the containing <update statement: positioned>, <update statement: searched>, or <merge statement>.

2) If *T* is not trigger updatable, then each <column name> specified as an <object column> shall identify an updatable column of *T*.

NOTE 687 — The notion of updatable columns of base tables is defined in Subclause 4.17, "Tables". The notion of updatable columns of viewed tables is defined in Subclause 11.32, "<view definition>".

3) No <object column> shall reference a column of which some underlying column is a self-referencing column, system-time period start column, or a system-time period end column.

4) Each <set clause> *SC* that immediately contains a <multiple column assignment> is effectively replaced by a <set clause list> *MSCL* as follows:

   a) Let *STN* be the number of <set target>s contained in <set target list>.

   b) *STN* shall be equal to the degree of the <assigned row> *AR* contained in *SC*.

   c) Let ST_i, 1 (one) ≤ i ≤ *STN*, be the i-th <set target> contained in the <set target list> of *SC* and let DT_i be the declared type of the i-th field of *AR*. The i-th <set clause> in *MSCL* is:

```
ST_i =
CAST ( AR AS ROW ( F1 DT_1, F2 DT_2, ..., FSTN DT_STN ) ).Fi
```

      NOTE 688 — "F*n*" here stands for the <field name> consisting of the letter "F" followed, with no intervening <separator> by the decimal <digit> or <digit>s comprising a <literal> corresponding to the value *n*.

5) If <set clause> *SC* specifies an <object column> that references a column of which some underlying column is either a generated column or an identity column whose descriptor indicates that values are always generated, then the <update source> specified in *SC* shall consist of a <default specification>.

6) A <value expression> simply contained in an <update source> in a <set clause> shall not directly contain a <set function specification>.

7) If the <set clause list> *OSCL* contains one or more <set clause>s that contain a <mutated set clause>, then:

   a) Let *N* be the number of <set clause>s in *OSCL* that contain a <mutated set clause>.

   b) For 1 (one) ≤ i ≤ *N*:

      i) Let SC_i be the i-th <set clause> that contains a <mutated set clause>.

      ii) Let $RCVE_i$ be the <update source> immediately contained in SC_i.

      iii) Let MSC_i be the <mutated set clause> immediately contained in the <set target> immediately contained in SC_i.

      iv) Let OC_i be the <object column> contained in MSC_i. The declared type of the column identified by OC_i shall be a structured type.

      v) Let M_i be the number of <method name>s contained in MSC_i.

      vi) For 1 (one) ≤ j ≤ M_i:

         Case:

         1) If j = 1 (one), then:

            A) Let $MT_{i,1}$ be the <mutated target> immediately contained in MSC_i.

            B) Let $MN_{i,1}$ be the <method name> immediately contained in MSC_i.

            C) Let $V_{i,1}$ be:

```
MT_i,1.MN_i,1 ( RCVE_i )
```

         2) Otherwise:

A) Let $MT_{i,j}$ be the <mutated target> immediately contained in the <mutated set clause> immediately contained in $MT_{i,j-1}$.

B) Let $MN_{i,j}$ be the <method name> immediately contained in the <mutated set clause> immediately contained in $MT_{i,j-1}$.

C) Let $V_{i,j}$ be

$MT_{i,j}.MN_{i,j}$ ($V_{i,j-1}$)

c) *OSCL* is equivalent to a <set clause list> *NSCL* derived as follows:

i) Let *NSCL* be a <set clause list> derived from *OSCL* by replacing every <set clause> SC_a, 1 (one) ≤ a ≤ N, that contains a <mutated set clause> with:

$MT_{a,Ma}$ = $V_{a,Ma}$

ii) For 1 (one) ≤ b ≤ N, if there exists a c such that c < b and OC_c is equivalent to OC_b, then:

1) Every occurrence of OC_b in $V_{b,Mb}$ is replaced by $V_{c,Mc}$.

2) SC_c is deleted from *NSCL*.

8) Equivalent <object column>s shall not appear more than once in a <set clause list>.

NOTE 689 — Multiple occurrences of equivalent <object column>s within <mutated set clause>s are eliminated by the preceding Syntax Rule of this Subclause.

9) If the <update source> of <set clause> *SC* specifies a <contextually typed value specification> *CVS*, then the data type of *CVS* is the data type *DT* of the <update target> or <mutated set clause> specified in *SC*.

10) If *CVS* is an <empty specification>, then *DT* shall be a collection type or a distinct type whose source type is a collection type. If *CVS* specifies ARRAY, then *DT* shall be an array type or a distinct type whose source type is an array type. If *CVS* specifies MULTISET, then *DT* shall be a multiset type or a distinct type whose source type is a multiset type.

11) For every <object column> in a <set clause>,

Case:

a) [15]If the <update target> immediately contains <simple value specification>, then the declared type of the column of *T* identified by the <object column> shall be an array type or a distinct type whose source type is an array type. The Syntax Rules of Subclause 9.2, "Store assignment", are applied with a temporary site whose declared type is element type of the column of *T* identified by the <object column> as *TARGET* and the <update source> of the <set clause> as *VALUE*.

b) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", are applied with the column of *T* identified by the <object column> as *TARGET* and the <update source> of the <set clause> as *VALUE*.

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F781, "Self-referencing operations", conforming SQL language shall not contain a <set clause> in which either of the following is true:

   a) A leaf generally underlying table of *T* is a leaf generally underlying table of any <query expression> broadly contained in any <value expression> simply contained in an <update source> or <assigned row> immediately contained in the <set clause>.

   b) An <update source> or <assigned row> immediately contained in the <set clause> broadly contains a <routine invocation>, <method invocation>, <static method invocation>, or <method reference> whose subject routine is an external routine that possibly reads SQL-data.

2) Without Feature S090, "Minimal array support", conforming SQL language shall not contain an <update target> that immediately contains a <simple value specification>.

3) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <set clause> in which the declared type of the <update target> in the <set clause> is a structured type *TY* and the declared type of the <update source> or corresponding field of the <assigned row> contained in the <set clause> is not *TY*.

4) Without Feature S024, "Enhanced structured types", conforming SQL language shall not contain a <set clause> that contains a <mutated set clause> and in which the declared type of the last <method name> identifies a structured type *TY*, and the declared type of the <update source> contained in the <set clause> is not *TY*.

5) [15] Without Feature T641, "Multiple column assignment", conforming SQL language shall not contain a <multiple column assignment>.

14.16 <temporary table declaration>

This Subclause is modified by Subclause 13.8, "<temporary table declaration>", in ISO/IEC 9075-4.

Function

Declare a declared local temporary table.

Format

```
<temporary table declaration> ::=
  DECLARE LOCAL TEMPORARY TABLE <table name> <table element list>
      [ ON COMMIT <table commit action> ROWS ]
```

Syntax Rules

1) Let *TN* be the <table name> of a <temporary table declaration> *TTD* and let *T* be the <qualified identifier> of *TN*.

2) 04 *TTD* shall be contained in an <SQL-client module definition> or in a <direct SQL statement>.

3) Case:

   a) If *TN* contains a <local or schema qualifier> *LSQ*, then *LSQ* shall be "MODULE".

   b) If *TN* does not contain a <local or schema qualifier>, then "MODULE" is implicit.

4) 04 If a <temporary table declaration> is contained in an <SQL-client module definition> *M*, then the <qualified identifier> of *TN* shall not be equivalent to the <qualified identifier> of the <table name> of any other <temporary table declaration> that is contained in *M*.

5) The descriptor of the table defined by a <temporary table declaration> includes *TN* and the column descriptor specified by each <column definition>. The *i*-th column descriptor is given by the *i*-th <column definition>.

6) <table element list> shall contain at least one <column definition> or at least one <like clause>.

7) <table element list> shall not contain a <table element> that is a <table period definition>.

8) If ON COMMIT is not specified, then ON COMMIT DELETE ROWS is implicit.

Access Rules

None.

General Rules

1) Let *U* be the implementation-dependent (UV107) <schema name> of the schema that contains the declared local temporary table such that the schema identified by *U* does not contain a table whose <table name> is equivalent to *TN*.

2) Let *UI* be the current user identifier and let *R* be the current role name.

   Case:

   a) If *UI* is not the null value, then let *A* be *UI*.

b) Otherwise, let *A* be *R*.

3) 04 The definition of *T* within an SQL-client module is effectively equivalent to the definition of a persistent base table *U.T*. Within the SQL-client module, any reference to MODULE.*T* is equivalent to a reference to *U.T*.

4) A set of privilege descriptors is created that define the privileges INSERT, SELECT, UPDATE, DELETE, and REFERENCES on this table and INSERT, SELECT, UPDATE, and REFERENCES for every <column definition> in the table definition to *A*. These privileges are not grantable. The grantor for each of these privilege descriptors is set to the special grantor value "_SYSTEM". The grantee is “PUBLIC”.

5) The definition of a temporary table persists for the duration of the SQL-session. The termination of the SQL-session is effectively followed by the execution of the following <drop table statement> with the current authorization identifier *A* and current <schema name> *U* without further Access Rule checking:

```
DROP TABLE T CASCADE
```

6) The definition of a declared local temporary table does not appear in any view of the Information Schema.

NOTE 690 — The Information Schema is defined in ISO/IEC 9075-11.

Conformance Rules

1) Without Feature F531, “Temporary tables”, conforming SQL language shall not contain a <temporary table declaration>.

14.17 <free locator statement>

Function

Remove the association between a locator variable and the value that is represented by that locator.

Format

```
<free locator statement> ::=
  FREE LOCATOR <locator reference> [ { <comma> <locator reference> }... ]

<locator reference> ::=
    <host parameter name>
  | <embedded variable name>
  | <dynamic parameter specification>
```

Syntax Rules

1) Each host parameter identified by <host parameter name> immediately contained in <locator reference> shall be a binary large object locator parameter, a character large object locator parameter, an array locator parameter, a multiset locator parameter, or a user-defined type locator parameter.

2) Each host variable identified by the <embedded variable name> immediately contained in <locator reference> shall be a binary large object locator variable, a character large object locator variable, an array locator variable, a multiset locator variable, or a user-defined type locator variable.

Access Rules

None.

General Rules

1) For every <locator reference> *LR* immediately contained in <free locator statement>, let *L* be the value of *LR*.

   Case:

   a) If *L* is not a valid locator value, then an exception condition is raised: *locator exception — invalid specification (0F001)*.

   b) Otherwise, *L* is marked invalid.

Conformance Rules

1) Without Feature T561, "Holdable locators", conforming SQL language shall not contain a <free locator statement>.

14.18 <hold locator statement>

Function

Mark a locator variable as being holdable.

Format

```
<hold locator statement> ::=
  HOLD LOCATOR <locator reference> [ { <comma> <locator reference> }... ]
```

Syntax Rules

1) Each host parameter identified by <host parameter name> immediately contained in <locator reference> shall be a binary large object locator parameter, a character large object locator parameter, an array locator parameter, a multiset locator parameter, or a user-defined type locator parameter.

Access Rules

None.

General Rules

1) For every <locator reference> *LR* immediately contained in <hold locator statement>, let *L* be the value of *LR*.

   Case:

   a) If *L* is not a valid locator value, then an exception condition is raised: *locator exception — invalid specification (0F001)*.

   b) Otherwise, *L* is marked *holdable*.

Conformance Rules

1) Without Feature T561, "Holdable locators", conforming SQL language shall not contain a <hold locator statement>.

15 Additional data manipulation rules

This Clause is modified by Clause 8, "Additional data manipulation rules", in ISO/IEC 9075-3.
This Clause is modified by Clause 14, "Additional data manipulation rules", in ISO/IEC 9075-4.
This Clause is modified by Clause 13, "Additional data manipulation rules", in ISO/IEC 9075-9.
This Clause is modified by Clause 14, "Additional data manipulation rules", in ISO/IEC 9075-15.

15.1 Effect of opening a cursor

This Subclause is modified by Subclause 8.1, "Effect of opening a cursor", in ISO/IEC 9075-3.
This Subclause is modified by Subclause 14.1, "Effect of opening a cursor", in ISO/IEC 9075-4.

Function

Specify the effect of opening a cursor that is not a received cursor.

Subclause Signature

```
"Effect of opening a cursor" [General Rules] (
  Parameter: "CURSOR"
)
```

CURSOR — a cursor instance descriptor.

Syntax Rules

None.

Access Rules

None.

General Rules

1) 03 Let *CR* be the *CURSOR* in an application of the General Rules of this Subclause.
2) If *CR* is not in the closed state, then an exception condition is raised: *invalid cursor state (24000)*.
3) Let *CDD* be the cursor declaration descriptor of *CR*.
4) Case:
   a) 03 If the kind of cursor described by *CDD* is a standing cursor, then let *S* be the declared <cursor specification> of *CDD*.
   b) If the kind of cursor described by *CDD* is a declared dynamic cursor, then let *S* be the prepared statement indicated by the <statement name> that is the origin of *CDD*.
   c) If the kind of cursor described by *CDD* is an extended dynamic cursor, then let *S* be the prepared statement that is the origin of *CDD*.

d) If the kind of cursor described by *CDD* is a PTF dynamic cursor, then let *S* be the <cursor specification> that is the origin of *CDD*.

NOTE 691 — If the kind of cursor is a received cursor, then this Subclause does not apply; instead, Subclause 15.2, "Effect of receiving a result set", applies.

5) *CR* is opened, and a result set descriptor *RSD* is created and included in *CR*, in the following steps:

a) The <cursor specification> of *RSD* is a copy *CS* of *S* that is effectively created as follows:

i) 04 Each <embedded variable specification>, <host parameter specification>, <SQL parameter reference>, and <dynamic parameter specification> is replaced by a <literal> denoting the value resulting from evaluating the <embedded variable specification>, <host parameter specification>, <SQL parameter reference>, and <dynamic parameter specification>, respectively, with all such evaluations effectively done at the same instant in time.

ii) Each <value specification> generally contained in *S* that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is replaced by a <literal> denoting the value resulting from evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>, respectively, with all such evaluations effectively done at the same instant in time.

b) Case:

i) If *CR* is a standing cursor, then the operational properties of *RSD* are the same as the corresponding declared properties of *CDD*.

ii) Otherwise, *CR* is a dynamic cursor.

1) The operational properties of *RSD* are initially copied from the corresponding declared properties of *CDD*.

2) If the <prepare statement> that prepared *S* contained an <attributes variable>, then let *CA* be the value of that <attributes variable>.

A) If *CA* contains a <cursor sensitivity>, then the operational sensitivity property of *RSD* is set to that <cursor sensitivity>.

B) If *CA* contains a <cursor scrollability>, then the operational scrollability property of *RSD* is set to that <cursor scrollability>.

C) If *CA* contains a <cursor holdability>, then the operational holdability property of *RSD* is set to that <cursor holdability>.

D) If *CA* contains a <cursor returnability>, then the operational returnability property of *RSD* is set to that <cursor returnability>.

c) If *CR* is being opened during either

i) the execution of an SQL-invoked function, or

ii) the execution of an externally-invoked procedure

without the intervening execution of an SQL-invoked procedure, then the operational returnability property of *RSD* is set to WITHOUT RETURN.

d) Let *QE* be the <query expression> simply contained in *CS*.

e) Let *T* be the table specified by *QE*.

f) Case:

i) If the operational sensitivity property of *RSD* is INSENSITIVE, then let *TT* be a copy of *T*.

ii) Otherwise, let *TT* be *T*.

g) The sequence of rows of *RSD* consists of the rows of *TT*, ordered as determined by the General Rules of Subclause 7.17, "<query expression>".

h) *CR* is placed in the open state and its position is before the first row of *TT*.

i) The cursor instance descriptor of *CR* is added to the SQL-session context.

6) If *CR* is insensitive, and the SQL-implementation is unable to guarantee that significant changes will be invisible through *CR* during the SQL-transaction in which *CR* is opened and every subsequent SQL-transaction during which it may be held open, then an exception condition is raised: *cursor sensitivity exception — request rejected (36001)*.

7) If *CR* is sensitive, and the SQL-implementation is unable to guarantee that significant changes will be visible through *CR* during the SQL-transaction in which *CR* is opened, then an exception condition is raised: *cursor sensitivity exception — request rejected (36001)*.

NOTE 692 — The visibility of significant changes through a sensitive holdable cursor during a subsequent SQL-transaction is implementation-defined (IA234).

8) Whether an SQL-implementation is able to disallow significant changes that would not be visible through a currently open cursor is implementation-defined (IA236).

9) If the operational returnability property of *RSD* is WITH RETURN, then let *SIP* be the active SQL-invoked procedure, let *INV* be the invoker of *SIP*, and let *RSS* be the result set sequence for *SIP* and *INV* in the active SQL-session context. *RSD* is added to the end of *RSS*.

10) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.2 Effect of receiving a result set

Function

Advance a received cursor to the next result set in a result set sequence.

Subclause Signature

```
"Effect of receiving a result set" [General Rules] (
  Parameter: "CURSOR",
  Parameter: "RESULT SET SEQUENCE"
)
```

CURSOR — a cursor instance descriptor.

RESULT SET SEQUENCE — a sequence of cursor result sets (rows of a derived table).

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* and let *RSS* be the *RESULT SET SEQUENCE* in an application of the General Rules of this Subclause.

2) Let *RS* be the first result set descriptor in *RSS*.

3) *CR* is placed in the open state, with *RS* as the result set descriptor included in *CR*.

4) *RS* is removed from *RSS*.

5) If the operational scrollability property of *RS* is NO SCROLL, then the position of *RS* is set to before the first row.

6) The operational returnability property of *RS* is implementation-defined (IV031).

7) The <cursor specification> of *RS* is made not updatable by replacing the explicit or implicit <updatability clause> with FOR READ ONLY.

8) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.3 Determination of the current row of a cursor

Function

Specify how the current row of a cursor is determined.

Subclause Signature

```
"Determination of the current row of a cursor" [General Rules] (
  Parameter: "CURSOR",
  Parameter: "FETCH ORIENTATION"
)
```

CURSOR — a cursor instance descriptor.

FETCH ORIENTATION — an indication of how the current row of CURSOR is to be determined (FIRST, LAST, NEXT, PRIOR, ABSOLUTE, or RELATIVE).

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* and let *FO* be the *FETCH ORIENTATION* in an application of the General Rules of this Subclause.

2) If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state (24000)*.

3) Let *RSD* be the result set descriptor of *CR*.

4) If *FO* is not NEXT and the operational scrollability property of *RSD* is not SCROLL, then an exception condition is raised: *syntax error or access rule violation (42000)*.

5) Case:

   a) If *FO* contains a <simple value specification>, then let *J* be the value of that <simple value specification>.

   b) If *FO* specifies NEXT or FIRST, then let *J* be +1.

   c) If *FO* specifies PRIOR or LAST, then let *J* be −1 (negative one).

6) Let *T* be the sequence of rows of *RSD*.

7) Let T_t be a sequence of rows of the same degree as *T*.

   Case:

   a) If *FO* specifies ABSOLUTE, FIRST, or LAST, then let T_t contain all rows of *T*, preserving their order in *T*.

b) If *FO* specifies NEXT or specifies RELATIVE with a positive value of *J*, then:

i) If *T* is empty or if the position of *CR* is on or after the last row of *T*, then let T_t be an empty sequence of rows.

ii) If the position of *CR* is on a row *R* that is other than the last row of *T*, then let T_t contain all rows of *T* ordered after row *R*, preserving their order in *T*.

iii) If the position of *CR* is before a row *R*, then let T_t contain row *R* and all rows of *T* ordered after row *R*, preserving their order in *T*.

c) If *FO* specifies PRIOR or specifies RELATIVE with a negative value of *J*, then:

i) If *T* is empty or if the position of *CR* is on or before the first row of *T*, then let T_t be an empty sequence of rows.

ii) If the position of *CR* is on a row *R* that is other than the first row of *T*, then let T_t contain all rows of *T* ordered before row *R*, preserving their order in *T*.

iii) If the position of *CR* is before a row *R* that is not the first row of *T*, then let T_t contain all rows of *T* ordered before row *R*, preserving their order in *T*.

iv) If the position of *CR* is after the last row of *T*, then let T_t contain all rows of *T*, preserving their order in *T*.

d) If RELATIVE is specified with a zero value of *J*, then

Case:

i) If the position of *CR* is on a row of *T*, then let T_t be a sequence of rows comprising that one row.

ii) Otherwise, let T_t be an empty sequence of rows.

8) Let *N* be the number of rows in T_t. If *J* is positive, then let *K* be *J*. If *J* is negative, then let *K* be *N*+*J*+1. If *J* is zero and ABSOLUTE is specified, then let *K* be zero; if *J* is zero and RELATIVE is specified, then let *K* be 1 (one).

9) Case:

a) If *K* is greater than 0 (zero) and not greater than *N*, then *CR* is positioned on the *K*-th row of T_t and the corresponding row of *T*. That row becomes the current row of *CR*.

b) Otherwise, a completion condition is raised: *no data (02000)*.

Case:

i) If *FO* specifies RELATIVE with *J* equal to 0 (zero), then the position of *CR* is unchanged.

ii) If *FO* implicitly or explicitly specifies NEXT, specifies ABSOLUTE or RELATIVE with *K* greater than *N*, or specifies LAST, then *CR* is positioned after the last row.

iii) Otherwise, *FO* specifies PRIOR, FIRST, or ABSOLUTE, or RELATIVE with *K* not greater than *N* and *CR* is positioned before the first row.

10) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.4 Effect of closing a cursor

Function

Specify the effect of closing a cursor.

Subclause Signature

```
"Effect of closing a cursor" [General Rules] (
  Parameter: "CURSOR",
  Parameter: "DISPOSITION"
)
```

CURSOR — a cursor instance descriptor.

DISPOSITION — if DESTROY, the result set of the cursor is destroyed.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR* and let *DISP* be the *DISPOSITION* in an application of the General Rules of this Subclause.

2) If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state (24000)*.

3) Let *RS* be the result set descriptor of *CR*.

4) *CR* is placed in the closed state.

5) The cursor instance descriptor of *CR* is removed from the SQL-session context.

6) Case:

   a) If the operational returnability property of *RS* is WITHOUT RETURN, then *RS* is destroyed.

   b) If *DISP* is DESTROY, then *RS* is removed from the result set sequence that includes *RS*, if any, and *RS* is destroyed.

   NOTE 693 — Otherwise, *RS* is not destroyed.

7) If *CR* is a received cursor, and the result set sequence *RSS* of the SQL-invoked routine specified in the cursor declaration descriptor of *CR* is not empty, then the General Rules of Subclause 15.2, "Effect of receiving a result set", are applied with *CR* as *CURSOR* and *RSS* as *RESULT SET SEQUENCE*.

8) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.5 Evaluating a <set clause list>

This Subclause is modified by Subclause 14.1, "Evaluating a <set clause list>", in ISO/IEC 9075-15.

Function

Specify the effects of evaluating a <set clause list>.

Subclause Signature

```
"Evaluating a <set clause list>" [General Rules] (
  Parameter: "SUBJECT ROW",
  Parameter: "SET CLAUSE LIST"
) Returns: "CANDIDATE NEW ROW"
```

SUBJECT ROW — a row of a derived table (the row to be updated).

SET CLAUSE LIST — a list of <set clause>s as specified in an <update statement: positioned> or an <update statement: searched>.

CANDIDATE NEW ROW — SUBJECT ROW modified by the <set clause>s in SET CLAUSE LIST.

Syntax Rules

None.

Access Rules

None.

General Rules

NOTE 694 — At the time that the General Rules of this Subclause are evaluated, *SCL*, the <set clause list>, has been transformed by the Syntax Rules of Subclause 14.15, "<set clause list>", to replace all <multiple column assignment>s and all <mutated set clause>s with ordinary <set clause>s that do not need special processing.

1) Let *SR* be the *SUBJECT ROW* and let *SCL* be the *SET CLAUSE LIST* in an application of the General Rules of this Subclause. The result of the application of this Subclause is returned as *CANDIDATE NEW ROW*.

2) Let *CNR* be a copy of *SR*.

   NOTE 695 — *CNR* has the same number of columns as *SR*, each having the same column name and the same declared type as the corresponding column of *SR*.

3) Let *N* be the number of <set clause>s in *SCL*. For each <set clause> SC_i, 1 (one) $\leq i \leq N$, in *SCL*:

   a) Let UT_i be the <update target> contained in SC_i, let OC_i be the column in *CNR* corresponding to the <object column> contained in UT_i, let US_i be the <update source> contained in SC_i, and let UV_i be the *update value* specified by US_i.

   b) The value of OC_i is replaced as follows.

      Case:

i) If UT_i immediately contains a <simple value specification>, then

NOTE 696 — This condition indicates that the target is an array element, e.g., COL2[7].

Case:

1) If the value of OC_i is the null value, then an exception condition is raised: *data exception — null value in array target (2200E)*.

2) Otherwise:

A) Let N be the maximum cardinality of OC_i.

B) Let M be the cardinality of the value of OC_i.

C) Let k be the value of the <simple value specification> immediately contained in UT_i.

D) Let *EDT* be the element type of OC_i.

E) Case:

I) If k is greater than zero and less than or equal to M, then the value of OC_i is replaced by an array A with element type *EDT* and cardinality M derived as follows:

1) For j varying from 1 (one) to $k-1$ and from $k+1$ to M, the j-th element in A is the value of the j-th element in OC_i.

2) The General Rules of Subclause 9.2, "Store assignment", are applied with A_k as *TARGET* and UV_i as *VALUE*.

II) If k is greater than M and less than or equal to N, then the value of OC_i is replaced by an array A with element type *EDT* and cardinality k derived as follows:

1) For j varying from 1 (one) to M, the j-th element in A is the value of the j-th element in OC_i.

2) For j varying from $M+1$ to $k-1$, the j-th element in A is the null value.

3) The General Rules of Subclause 9.2, "Store assignment", are applied with A_k as *TARGET* and UV_i as *VALUE*.

III) Otherwise, an exception condition is raised: *data exception — array element error (2202E)*.

ii) [15] The General Rules of Subclause 9.2, "Store assignment", are applied with OC_i as *TARGET* and UV_i as *VALUE*.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause, which receives *CNR* as *CANDIDATE NEW ROW*.

Conformance Rules

None.

15.6 Effect of a positioned delete

Function

Specify the effect of a positioned delete.

Subclause Signature

```
"Effect of a positioned delete" [General Rules] (
  Parameter: "CURSOR",
  Parameter: "STATEMENT",
  Parameter: "TARGET"
)
```

CURSOR — a cursor instance descriptor.

STATEMENT — a <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned>.

TARGET — unless TARGET contains ONLY, rows in subtables of the virtual table identified by CURSOR are also to be deleted.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR*, let *DSP* be the *STATEMENT*, and let *TT* be the *TARGET* in an application of the General Rules of this Subclause.

2) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only, and not every leaf generally underlying table of *CR* is a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

3) If there is any sensitive cursor *SCR*, other than *CR*, that is currently open in the SQL-transaction in which *DSP* is being executed, then

   Case:

   a) If *SCR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of *DSP* is made visible to *SCR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

   b) Otherwise, whether the change resulting from the successful execution of *DSP* is made visible to *SCR* is implementation-defined (IA233).

4) If there is any insensitive cursor *ICR*, other than *CR*, that is currently open, then either the change resulting from the successful execution of *DSP* is invisible to *ICR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

5) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

6) If *CR* is not positioned on a row, then an exception condition is raised: *invalid cursor state (24000)*.

7) If *CR* is a holdable cursor and a <fetch statement> has not been issued against *CR* within the current SQL-transaction, then an exception condition is raised: *invalid cursor state (24000)*.

8) Let *T* be the simply underlying table specification of *CR* and let *LUT* be the target leaf underlying table of *T*.

9) Let *R* be the current row of *CR*. Exactly one row *R1* in *LUT* such that each field in *R* is identical to the corresponding field in *R1* is identified for deletion from *LUT*.

NOTE 697 — In case more than one row *R1* satisfies the stated condition, it is implementation-dependent which one is identified for deletion.

NOTE 698 — Identifying a row for deletion is an implementation-dependent mechanism.

10) Whether the current row is removed from the sequence of rows of the result set descriptor of *CR* is implementation-defined (IA170).

11) Case:

a) If *LUT* is a base table, then:

i) Case:

1) If *TT* specifies ONLY, then *LUT* is *identified for deletion processing without subtables*.

2) Otherwise, *LUT* is *identified for deletion processing with subtables*.

NOTE 699 — Identifying a base table for deletion processing, with or without subtables, is an implementation-dependent mechanism.

ii) The General Rules of Subclause 15.8, "Effect of deleting rows from base tables", are applied.

b) If *LUT* is a viewed table, then the General Rules of Subclause 15.10, "Effect of deleting some rows from a viewed table", are applied with *TT* as *VIEW NAME*.

12) If, while *CR* is open, the row from which the current row of *CR* is derived has been marked for deletion by any <delete statement: searched>, by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies any cursor other than *CR*, or by any <merge statement>, or has been updated by any <update statement: searched>, by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies any cursor other than *CR*, or by any <merge statement>, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

13) If the execution of *DSP* deleted the last row of *CR*, then the position of *CR* is after the last row; otherwise, the position of *CR* is before the next row.

14) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.7 Effect of a positioned update

Function

Specify the effect of a positioned update.

Subclause Signature

```
"Effect of a positioned update" [General Rules] (
  Parameter: "CURSOR",
  Parameter: "SET CLAUSE LIST",
  Parameter: "STATEMENT",
  Parameter: "TARGET"
)
```

CURSOR — a cursor instance descriptor.

SET CLAUSE LIST — a list of <set clause>s as specified in STATEMENT.

STATEMENT — an <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned>.

TARGET — unless TARGET contains ONLY, rows in subtables of the virtual table identified by CURSOR are also to be updated.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *CR* be the *CURSOR*, let *SCL* be the *SET CLAUSE LIST*, let *USP* be the *STATEMENT*, and let *TT* be the *TARGET* in an application of the General Rules of this Subclause.

2) If the transaction access mode of the current SQL-transaction or the transaction access mode of the branch of the current SQL-transaction at the current SQL-connection is read-only and not every leaf generally underlying table of *CR* is a temporary table, then an exception condition is raised: *invalid transaction state — read-only SQL-transaction (25006)*.

3) If there is any sensitive cursor *SCR*, other than *CR*, that is currently open in the SQL-transaction in which *USP* is being executed, then

   Case:

   a) If *SCR* has not been held into a subsequent SQL-transaction, then either the change resulting from the successful execution of *USP* is made visible to *SCR* or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

   b) Otherwise, whether the change resulting from the successful execution of *USP* is made visible to *SCR* is implementation-defined (IA233).

4) If there is any insensitive cursor *ICR*, other than *CR*, that is currently open, then either the change resulting from the successful execution of *USP* is invisible to *ICR*, or an exception condition is raised: *cursor sensitivity exception — request failed (36002)*.

5) The extent to which an SQL-implementation may disallow independent changes that are not significant is implementation-defined (IA234).

6) If *CR* is not positioned on a row, then an exception condition is raised: *invalid cursor state (24000)*.

7) If *CR* is a holdable cursor and a <fetch statement> has not been issued against *CR* within the current SQL-transaction, then an exception condition is raised: *invalid cursor state (24000)*.

8) An object row is any row of a base table from which the current row of *CR* is derived.

9) If, while *CR* is open, an object row has been marked for deletion by any <delete statement: searched>, by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies any cursor other than *CR*, or by <merge statement>, or has been updated by any <update statement: searched>, any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies any cursor other than *CR*, or by any <merge statement>, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

10) The value associated with DEFAULT is the default value for the <object column> in the containing <set clause> contained in *SCL*, as indicated in the General Rules of Subclause 11.5, "<default clause>".

11) Each <update source> contained in *SCL* is effectively evaluated for the current row before any of the current row's object rows is updated.

12) *CR* remains positioned on its current row, even if an exception condition is raised during evaluation of any <update source>.

13) The General Rules of Subclause 15.5, "Evaluating a <set clause list>", are applied with the current row of *CR* as *SUBJECT ROW* and *SCL* as *SET CLAUSE LIST*; let *CNR* be the *CANDIDATE NEW ROW* returned from the application of those General Rules

NOTE 700 — The data values allowable in an object row can be constrained by a WITH CHECK OPTION constraint. The effect of a WITH CHECK OPTION constraint is defined in the General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION".

14) Let *SUTS* be the simply underlying table specification of *CR*, let *T* be the table identified by *SUTS*, and let *LUT* be the target leaf underlying table of *SUTS*.

15) Let *CL* be the columns of *T* identified by the <object column>s contained in *SCL*.

16) Let *R1* be *CNR* and let *R* be the current row of *CR*. Exactly one row *TR* in *T*, such that the value of each field in *R* that is derived from one or more fields in *TR* is identical to the corresponding value that is derived from the same one or more fields in *TR* is identified for replacement in *T*. The current row *R* of *CR* is replaced by *R1*. Let *TR1* be a row consisting of the fields of *R1* and the fields of *TR* that have no corresponding fields in *R1*, ordered according to the order of their corresponding columns in *T*. *TR1* is the replacement row for *TR* and { (*TR*, *TR1*) } is the replacement set for *T*.

NOTE 701 — In case more than one row *R1* satisfies the stated condition, it is implementation-dependent which one is identified for replacement.

NOTE 702 — Identifying a row for replacement, associating a replacement row with an identified row, and associating a replacement set with a table are implementation-dependent mechanisms.

17) Case:

a) If *LUT* is a base table, then:

i) Case:

1) If *TT* specifies ONLY, then *LUT* is *identified for replacement processing without subtables* with respect to object columns *CL*.

2) Otherwise, *LUT* is *identified for replacement processing with subtables* with respect to object columns *CL*.

NOTE 703 — Identifying a base table for replacement processing, with or without subtables, is an implementation-dependent mechanism. In general, though not here, the list of object columns can be empty.

ii) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

b) If *LUT* is a viewed table, then:

i) The General Rules of Subclause 15.16, "Effect of replacing some rows in a viewed table", are applied with *TT* as *VIEW NAME* and the replacement set for *T* as *REPLACEMENT SET FOR VIEW NAME*.

ii) The General Rules of Subclause 15.17, "Checking of views that specify CHECK OPTION", are applied with UPDATE as *OPERATION*.

18) If a column *C* of a base table is modified, and the evaluation of the <value expression> of some <sort key> simply contained in the <query expression> of the <cursor specification> of *CR* references *C*, then the position of *CR* is implementation-dependent (US047).

19) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.8 Effect of deleting rows from base tables

This Subclause is modified by Subclause 13.1, "Effect of deleting rows from base tables", in ISO/IEC 9075-9.

Function

Specify the effect of deleting rows that have been marked for deletion from one or more base tables.

NOTE 704 — The rows to be deleted have been *marked for deletion* as a result of processing <delete statement: searched> and <merge statement>, as well as General Rules in other Subclauses in Clause 15, "Additional data manipulation rules".

Subclause Signature

"Effect of deleting rows from base tables" [General Rules] ()

Syntax Rules

None.

Access Rules

None.

General Rules

1) The General Rules of this Subclause are applied without any symbolic arguments.

2) Let *TT* be the set consisting of every base table that is identified for deletion processing, with or without subtables. Let *S* be the set consisting of every row identified for deletion in some table in *TT*.

3) Let *TT2* be the set consisting of the following tables:

   a) Every supertable of every table in *TT*.

   b) Every subtable of every table in *TT* that is identified for deletion processing with subtables.

      NOTE 705 — The purpose of deletion processing with subtables for a table *T* is to ensure that statement-level triggers are fired for the proper subtables of *T*, even if the rows identified for deletion have no proper subrows.

4) For every row *R* in *S*:

   a) Every superrow *SR* of *R* is identified for deletion from the base table *BT* containing *SR*.

   b) If the table containing *R* is identified for deletion processing with subtables, then every subrow *SR* of *R* is identified for deletion from the base table *BT* containing *SR*.

5) Let *SSC* be the set of state changes in the most recent statement execution context.

6) For every table *ST* in *TT2*,

   Case:

   a) If a state change *SC* exists in *SSC* with subject table *ST* and trigger event DELETE, then one copy each of every row of *ST* that is identified for deletion in *ST* is added to the set of transitions of *SC*.

b) Otherwise, a state change *SC* is added to *SSC* as follows:

i) The set of transitions of *SC* consists of one copy each of every row of *ST* that is identified for deletion in *ST*.

ii) The trigger event of *SC* is DELETE.

iii) The subject table of *SC* is *ST*.

iv) The column list of *SC* is empty.

v) The set of statement-level triggers for which *SC* is considered as executed is empty.

vi) The set of row-level triggers consists of each row-level trigger that is activated by *SC*, paired with the empty set (of rows considered as executed).

7) The General Rules of Subclause 15.19, "Execution of BEFORE triggers", are applied with *SSC* as *SET OF STATE CHANGES*.

8) 09 Every row that is identified for deletion in some table in *TT2* is marked for deletion. These rows are no longer identified for deletion, nor are their containing tables identified for deletion processing (with or without subtables).

NOTE 706 — "Marking for deletion" is an implementation-dependent mechanism.

9) For every row *BR* that has an associated for portion of from-value and the associated for portion of to-value in *S*:

a) Let *ATPN* be the period name included in the application-time period descriptor included in the descriptor of *T*. Let *FROMVAL* be the associated for portion of from-value. Let *TOVAL* be the associated for portion of to-value. Let *BSTARTVAL* be the value of the *ATPN* period start column. Let *BCD* be the declared type of the *ATPN* period start column. Let *BENDVAL* be the value of the *ATPN* period end column. Let *TN* be the table name included in the descriptor of *T*. Let *d* be the degree of *BR*.

b) For *i*, 1 (one) ≤ *i* ≤ *d*:

Case:

i) If the column descriptor that corresponds to the *i*-th field of *BR* describes an identity column, a generated column, a system-time period start column, or a system-time period end column, then let V_i be DEFAULT.

ii) Otherwise, let V_i be the value of the *i*-th field of *BR*.

c) If *BSTARTVAL* < *FROMVAL* and *FROMVAL* < *BENDVAL*, then:

i) For *j*, 1 (one) ≤ *j* ≤ *d*,

Case:

1) If the *j*-th field of *BR* corresponds to the *ATPN* period end column of *T*, then let VL_j be CAST (*FROMVAL* AS *BCD*).

2) Otherwise, let VL_j be V_j.

ii) The following <insert statement> is effectively executed without further Access Rule and constraint checking:

```
INSERT INTO TN VALUES (VL1, ..., VLd)
```

NOTE 707 — Constraint checking is performed at the end of the triggering DELETE statement.

d) If *BSTARTVAL* < *TOVAL* and *TOVAL* < *BENDVAL*, then:

i) For *k*, 1 (one) ≤ *k* ≤ *d*,

Case:

1) If the *k*-th field of *BR* corresponds to the *ATPN* period start column of *T*, then let VR_k be CAST (*TOVAL* AS *BCD*).

2) Otherwise, let VR_k be V_k.

ii) The following <insert statement> is effectively executed without further Access Rule and constraint checking:

```
INSERT INTO TN VALUES (VR1, ..., VRd)
```

NOTE 708 — Constraint checking is performed at the end of the triggering DELETE statement.

Conformance Rules

None.

15.9 Effect of deleting some rows from a derived table

Function

Specify the effect of deleting some rows from a derived table.

Subclause Signature

```
"Effect of deleting some rows from a derived table" [General Rules] (
  Parameter: "TABLE"
)
```

TABLE — a derived table.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *QE* be the *TABLE* in an application of the General Rules of this Subclause.

2) Let *T* be the result of evaluating *QE*.

3) Case:

   a) If *QE* simply contains a <query primary> that immediately contains a <query expression body>, then let *QEB* be that <query expression body>. The General Rules of this Subclause are applied with the table identified by *QEB* as *TABLE*.

   b) If *QE* simply contains a <query expression body> *QEB* that specifies UNION ALL, then let *LO* and *RO* be the <query expression body> and the <query term>, respectively, that are immediately contained in *QEB*. Let *T1* and *T2* be the tables identified by *LO* and *RO*, respectively.

      i) For every row *R* in *T* that has been identified for deletion, let *RD* be the row in either *T1* or *T2* from which *R* has been derived and let *TD* be that table. Identify *RD* for deletion.

      ii) The General Rules of this Subclause are applied with *LO* as *TABLE*.

      iii) The General Rules of this Subclause are applied with *RO* as *TABLE*.

   c) Otherwise, let *QS* be the <query specification> simply contained in *QE*. Let *TE* be the <table expression> immediately contained in *QS*, and *TREF* be the <table reference>s simply contained in the <from clause> of *TE*.

      i) Case:

         1) If *TREF* contains only one <table reference>, then let TR_1 be that <table reference>, and let *m* be 1 (one).

2) Otherwise, let m be the number of <table reference>s that identify tables with respect to which *QS* is one-to-one. Let TR_i, 1 (one) ≤ i ≤ m, be those <table reference>s.

NOTE 709 — The notion of one-to-one <query specification>s is defined in Subclause 7.16, "<query specification>".

ii) Let TT_i, 1 (one) ≤ i ≤ m, be the table identified by TR_i.

iii) For every row *R* of *T* that has been identified for deletion, and for i ranging from 1 (one) to m, let *RD* be the row in TT_i from which *R* has been derived. Identify that *RD* for deletion.

iv) For i ranging from 1 (one) to m,

Case:

1) If TT_i is a base table, then

Case:

A) If TR_i specifies ONLY, then TT_i is identified for deletion processing without subtables.

B) Otherwise, TT_i is identified for deletion processing with subtables.

2) If TT_i is a viewed table, then the General Rules of Subclause 15.10, "Effect of deleting some rows from a viewed table", are applied with the <table name> of TT_i as *VIEW NAME*.

3) Otherwise, the General Rules of this Subclause are applied with TR_i as *TABLE*.

v) The General Rules of Subclause 15.8, "Effect of deleting rows from base tables", are applied.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.10 Effect of deleting some rows from a viewed table

Function

Specify the effect of deleting some rows from a viewed table.

Subclause Signature

```
"Effect of deleting some rows from a viewed table" [General Rules] (
  Parameter: "VIEW NAME"
)
```

VIEW NAME — the name of a view.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *VN* be the *VIEW NAME* in an application of the General Rules of this Subclause.

2) Let *V* be the viewed table identified by *VN*.

3) Case:

   a) If *V* is trigger deletable, then:

      i) Let *TR* be the delete INSTEAD OF trigger whose subject table is *V*.

      ii) A state change *SC* is created as follows:

         1) The set of transitions of *SC* consists of one copy each of every row of *V* that is identified for deletion.

         2) The trigger event of *SC* is DELETE.

         3) The subject table of *SC* is *V*.

         4) The column list of *SC* is empty.

         5) The set of statement-level triggers for which *SC* is considered as executed is empty.

         6) The set of row-level triggers consists of *TR* paired with the empty set (of rows considered as executed).

      iii) The General Rules of Subclause 15.21, "Execution of triggers", are applied with *TR* as *TRIGGER* and *SC* as *STATE CHANGE*.

   b) Otherwise:

i) If *VN* specifies ONLY, then let *QE* be the original <query expression> included in the descriptor of the view *V* identified by *VN*; otherwise, let *QE* be the hierarchical <query expression> contained in that descriptor. Let *T* be the result of evaluating *QE*.

ii) For each row *R* of *V* that has been identified for deletion, let *RD* be the row in *T* from which *R* has been derived; identify that row for deletion.

iii) The General Rules of Subclause 15.9, "Effect of deleting some rows from a derived table", are applied with *QE* as *TABLE*.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.11 Effect of inserting tables into base tables

This Subclause is modified by Subclause 13.2, "Effect of inserting tables into base tables", in ISO/IEC 9075-9.

Function

Specify the effect of inserting each of one or more given tables into its associated base table.

Subclause Signature

```
"Effect of inserting tables into base tables" [General Rules] (
  Parameter: "SOURCE",
  Parameter: "TARGET"
)
```

SOURCE — a derived table that is the source for insertions into TARGET.

TARGET — a base table into which rows are to be inserted.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the *SOURCE* and let *T* be the *TARGET* in an application of the General Rules of this Subclause.

2) Let *SSC* be the set of state changes in the most recent statement execution context.

3) For each base table *T* that is identified for insertion, let *S* be the source table for *T*.

   a) If some column *IC* of *T* is the identity column of *T*, then for each row in *S* whose site *ICS* corresponding to *IC* is marked as *unassigned*:

      i) *ICS* is no longer marked as unassigned.

      ii) The General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are applied with the sequence generator descriptor included in the column descriptor of *IC* as *SEQUENCE*; let *NV* be the *RESULT* returned from the application of those General Rules.

      Case:

      1) If the declared type of *IC* is a distinct type *DIST*, then let *ICNV* be *DIST*(*NV*).

      2) Otherwise, let *ICNV* be *NV*.

      iii) The General Rules of Subclause 9.2, "Store assignment", are applied with *ICS* as *TARGET* and *ICNV* as *VALUE*.

b) If some column *IC* of *T* is the system-time period start column of *T*, then for each row in *S* whose site *ICS* corresponding to *IC* is marked as unassigned:

i) *ICS* is no longer marked as unassigned.

ii) Let *NV* be the transaction timestamp of the current SQL-transaction. Let *DT* be the declared type of the system-time period start column of *T*. Let *NVV* be the result of

```
CAST (NV AS DT)
```

iii) The General Rules of Subclause 9.2, "Store assignment", are applied with *ICS* as *TARGET* and *NVV* as *VALUE*.

c) If some column *IC* of *T* is the system-time period end column of *T*, then for each row in *S* whose site *ICS* corresponding to *IC* is marked as unassigned:

i) *ICS* is no longer marked as unassigned.

ii) Let *NV* be the highest value supported by the declared type of the system-time period end column of *T*.

iii) The General Rules of Subclause 9.2, "Store assignment", are applied with *ICS* as *TARGET* and *NV* as *VALUE*.

d) Every proper supertable *ST* of *T* is identified for insertion. A source table for insertion into each *ST* is constructed as follows:

i) Let *S* be the source table for the insertion into *T*. Let *TVC* be some <table value constructor> whose value is *S*.

ii) Let *n* be the number of column descriptors included in the table descriptor of *ST* and let CD_i, 1 (one) $\le i \le n$, be those column descriptors. Let *SL* be a <select list> containing *n* <select sublist>s such that, for *i* ranging from 1 (one) to *n*, the *i*-th <select sublist> consists of the column name included in CD_i.

iii) The source table for insertion into *ST* consists of the rows in the result of the <query expression>:

```
SELECT SL FROM TVC
```

4) For every base table *BT* that is identified for insertion,

Case:

a) If a state change *SC* exists in *SSC* with subject table *BT* and trigger event INSERT, then the rows in the source table for *BT* are added to the set of transitions of *SC*.

b) Otherwise, a state change *SC* is added to *SSC* as follows:

i) The set of transitions of *SC* consists of the rows in the source table for *BT*.

ii) The trigger event of *SC* is INSERT.

iii) The subject table of *SC* is *BT*.

iv) The column list of *SC* is empty.

v) The set of statement-level triggers for which *SC* is considered as executed is empty.

vi) The set of row-level triggers consists of each row-level trigger that is activated by *SC*, paired with the empty set (of rows considered as executed).

5) The General Rules of Subclause 15.19, "Execution of BEFORE triggers", are applied with *SSC* as *SET OF STATE CHANGES*.

6) For every state change *SC* in *SSC*, let *SOT* be the set of transitions in *SC* and let *BT* be the subject table of *SC*.

   a) In each row *R* in *SOT*, for each site *GCS* in *R* corresponding to a generated column *GC*, let *GCR* be the result of evaluating, for R, the generation expression included in the column descriptor of *GC*. The General Rules of Subclause 9.2, "Store assignment", are applied with *GCS* as *TARGET* and *GCR* as *VALUE*.

   b) Case:

      i) If *BT* has no new delta table of insert operation on *BT*, then *SOT* is the *new delta table of insert operation* on *BT*.

      ii) Otherwise, the rows of *SOT* are added to the new delta table of insert operation on *BT*.

      NOTE 710 — The latter scenario can arise during a <merge statement> that has more than one <merge insert specification>.

   c) Every row in *SOT* is inserted into *BT*.

7) For every pair of base tables *BT1* and *BT2* identified for insertion, if *BT1* is a direct subtable of *BT2*, then the new delta table of insert operation on *BT1* is a direct subtable of the new delta table of insert operation on *BT2*.

8) 09 For every base table *BT* identified for insertion, *BT* is no longer identified for insertion.

9) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.12 Effect of inserting a table into a derived table

Function

Specify the effect of inserting a table into a derived table.

Subclause Signature

```
"Effect of inserting a table into a derived table" [General Rules] (
  Parameter: "SOURCE",
  Parameter: "TARGET"
)
```

SOURCE — a derived table that is the source for insertions into TARGET.

TARGET — a derived table into which rows are to be inserted.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *Q* be the *SOURCE* and let *T* be the *TARGET* in an application of the General Rules of this Subclause.

2) Let *QE* be the <query expression> included in the descriptor of *T*.

   Case:

   a) If *QE* simply contains a <query primary> that immediately contains a <query expression body>, then let *QEB* be that <query expression body>. The General Rules of this Subclause are applied with *Q* as *SOURCE* and the result of *QEB* as *TARGET*.

   b) Otherwise, let *QS* be the <query specification> simply contained in *QE*. Let *TE* be the <table expression> immediately contained in *QS*, and *TREF* be the <table reference>s simply contained in the <from clause> of *TE*. Let *SL* be the <select list> immediately contained in *QS*, and *n* be the number of <value expression>s VE_j, 1 (one) ≤ *j* ≤ *n*, simply contained in *SL*.

      i) Case:

         1) If *TREF* contains only one <table reference>, then let TR_1 be that <table reference>, and let *m* be 1 (one).

         2) Otherwise, let *m* be the number of <table reference>s that identify tables with respect to which *QS* is one-to-one. Let TR_i, 1 (one) ≤ *i* ≤ *m*, be those <table reference>s.

      ii) Let TT_i, 1 (one) ≤ *i* ≤ *m*, be the table identified by TR_i, and let S_i be an initially empty table of candidate rows for TT_i.

iii) For every row *R* of *Q*, and for *i* ranging from 1 (one) to *m*:

1) A candidate row of TT_i is effectively created in which the value of each column is its default value, as specified the General Rules of Subclause 11.5, "<default clause>". The candidate row includes every column of TT_i.

2) For *j* ranging from 1 (one) to *n*, let *C* be a column of some candidate row identified by VE_j, and let *SV* be the *j*-th value of *R*. The General Rules of Subclause 9.2, "Store assignment", are applied with *C* as *TARGET* and *SV* as *VALUE*

3) The candidate row is added to the corresponding S_i.

iv) For *i* ranging from 1 (one) to *m*,

Case:

1) If TT_i is a base table, then TT_i is identified for insertion of source table S_i.

2) If TT_i is a viewed table, the General Rules of Subclause 15.13, "Effect of inserting a table into a viewed table", are applied with S_i as *SOURCE* and TT_i as *TARGET*.

3) Otherwise, the General Rules of this Subclause are applied with S_i as *SOURCE* and TT_i as *TARGET*.

v) The General Rules of Subclause 15.11, "Effect of inserting tables into base tables", are applied.

3) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.13 Effect of inserting a table into a viewed table

Function

Specify the effect of inserting a table into a viewed table.

Subclause Signature

```
"Effect of inserting a table into a viewed table" [General Rules] (
  Parameter: "SOURCE",
  Parameter: "TARGET"
)
```

SOURCE — a derived table that is the source for insertions into TARGET.

TARGET — a view into which rows are to be inserted.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the *SOURCE* and let *T* be the *TARGET* in an application of the General Rules of this Subclause.

2) Let *TD* be the view descriptor of *T*. Let *QE* be the original <query expression> included in *TD*.

3) Case:

   a) If *T* is trigger insertable-into, then:

      i) Let *TR* be the insert INSTEAD OF trigger whose subject table is *T*.

      ii) A state change *SC* is created as follows:

         1) The set of transitions of *SC* consists of one copy each of every row of *S*.

         2) The trigger event of *SC* is INSERT.

         3) The subject table of *SC* is *T*.

         4) The column list of *SC* is empty.

         5) The set of statement-level triggers for which *SC* is considered as executed is empty.

         6) The set of row-level triggers consists of *TR* paired with the empty set (of rows considered as executed).

      iii) The General Rules of Subclause 15.21, "Execution of triggers", are applied with *TR* as *TRIGGER* and *SC* as *STATE CHANGE*.

   b) Otherwise:

i) If *TD* indicates CHECK OPTION, then *TD* is added to the set of views to be checked in the current statement execution context.

ii) The General Rules of Subclause 15.12, "Effect of inserting a table into a derived table", are applied with *S* as *SOURCE* and *T* as *TARGET*.

4) Let *n* be the number of target leaf generally underlying tables of *QE*. Let T_i, 1 (one) ≤ *i* ≤ *n*, be the target leaf generally underlying tables of *QE*. Let NT_i, 1 (one) ≤ *i* ≤ *n*, be the new delta table of insert operation on T_i. Let *S* be the result of evaluating *QE* with every reference to T_i, 1 (one) ≤ *i* ≤ *n*, being replaced with a reference to NT_i. *S* is the *new delta table of insert operation* on *T*.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.14 Effect of replacing rows in base tables

This Subclause is modified by Subclause 13.3, "Effect of replacing rows in base tables", in ISO/IEC 9075-9.

Function

Specify the effect of replacing rows that have been marked for replacement in one or more base tables.

NOTE 711 — The rows to be replaced have been *marked for replacement* as a result of processing <merge statement> and <update statement: searched>, as well as General Rules in other Subclauses in Clause 15, "Additional data manipulation rules".

Subclause Signature

"Effect of replacing rows in base tables" [General Rules] ()

Syntax Rules

None.

Access Rules

None.

General Rules

1) The General Rules of this Subclause are applied without any symbolic arguments.

2) Let *TT* be the set consisting of every base table that is identified for replacement processing, with or without subtables. Let *S* be the set consisting of every row identified for replacement in every table in *TT*.

3) Let *TT2* be the set comprising the following tables:

   a) Every supertable of every table in *TT*.

   b) Every subtable of every table in *TT* that is identified for replacement processing with subtables.

      NOTE 712 — The purpose of replacement processing with subtables for a table *T* is to ensure that statement-level triggers are fired for the proper subtables of *T*, even if the rows identified for replacement have no proper subrows.

4) For every base table *T* in *TT*:

   a) Let *OC*(*T*) be the set consisting of the name of every object column with respect to which *T* is identified for replacement processing and the name of every generated column of *T* that depends on at least one of these object columns.

   b) For every table *ST* in *TT2* that is a subtable or supertable of *T*, let *OC*(*ST*) be the intersection (possibly empty) of *OC*(*T*) and the set of names of the columns of *ST*.

5) For every row *R* that is identified for replacement in some table *T* in *TT*, every row *SR* that is a proper subrow or a proper superrow of *R* is identified for replacement in the base table *ST* that contains *SR*. The replacement set *RST* for *ST* is derived from the replacement set *RR* for *T* as follows.

   Case:

a) If *ST* is a subtable of *T*, then each replacement row in *RST* is the corresponding replacement row in *RR* extended with those fields of the corresponding identified row in *ST* that have no corresponding column in *T*.

b) If *ST* is a supertable of *T*, then each replacement row in *RST* is the corresponding replacement row in *RR* minus those fields that have no corresponding column in *ST*.

6) Let *SSC* be the set of state changes in the most recent statement execution context.

7) For every table *ST* in *TT2*:

a) If some column *IC* of *T* is the identity column of *ST*, then, for each row *RR* identified for replacement in *ST* whose site *ICS* corresponding to *IC* in the replacement row for *RR* is marked as *unassigned*:

i) *ICS* is no longer marked as unassigned.

ii) The General Rules of Subclause 9.35, "Generation of the next value of a sequence generator", are applied with the sequence generator descriptor included in the column descriptor of *IC* as *SEQUENCE*; let *NV* be the *RESULT* returned from the application of those General Rules.

Case:

1) If the declared type of *IC* is a distinct type *DIST*, then let *ICNV* be *DIST*(*NV*).

2) Otherwise, let *ICNV* be *NV*.

iii) The General Rules of Subclause 9.2, "Store assignment", are applied with *ICS* as *TARGET* and *ICNV* as *VALUE*.

b) Let *TL* be the set consisting of the names of the columns of *ST*. For every subset *STL* of *TL* such that either *STL* is empty or the intersection of *STL* and *OC*(*ST*) is not empty,

Case:

i) If a state change *SC* exists in *SSC* with subject table *ST*, trigger event UPDATE, and column list *STL*, then the row pairs formed by pairing each row identified for replacement in *ST* with its corresponding replacement row are added to the set of transitions of *SC*.

ii) Otherwise, a state change *SC* is added to *SSC* as follows:

1) The set of transitions of *SC* consists of row pairs formed by pairing each row identified for replacement in *ST* with its corresponding replacement row.

2) The trigger event of *SC* is UPDATE.

3) The subject table of *SC* is *ST*.

4) The column list of *SC* is *STL*.

5) The set of statement-level triggers for which *SC* is considered as executed is empty.

6) The set of row-level triggers consists of each row-level trigger that is activated by *SC*, paired with the empty set (of rows considered as executed).

8) The General Rules of Subclause 15.19, "Execution of BEFORE triggers", are applied with *SSC* as *SET OF STATE CHANGES*.

9) For each set of transitions *RST* in each state change *SC* in *SSC*, in each row *R* in *RST*, for each site *GCS* in *R* corresponding to a generated column *GC* in the subject table of *SC*, let *GCR* be the result of

evaluating, for *R*, the generation expression included in the column descriptor of *GC*. The General Rules of Subclause 9.2, "Store assignment", are applied with *GCS* as *TARGET* and *GCR* as *VALUE*.

10) 09 For every table *T* in *TT2*,

Case:

a) If the descriptor of *T* includes a system-time period descriptor, then:

i) Let *START* be the system-time period start column of *T* and let *END* be the system-time period end column of *T*. Let *DT* be the declared type of *START*.

ii) Let *TTS* be the transaction timestamp of the current SQL-transaction. Let *CTTS* be the result of

```
CAST (TTS AS DT)
```

iii) For every row *R* that is identified for replacement in *T*, let *STARTVAL* be the value of system-time period start column.

1) Case:

A) If *CTTS* < *STARTVAL*, then an exception condition is raised: *data exception — invalid row version (2201H)*.

B) If *CTTS* = *STARTVAL*, then *R* is replaced by its corresponding replacement row.

C) Otherwise:

I) If *T* is a system-versioned table, then a copy of *R* with the value of *END* effectively replaced by *CTTS* is inserted into *T*.

II) *R* is replaced by its corresponding replacement row with the value of *START* effectively replaced by *CTTS*.

2) *R* is no longer identified for replacement. *T* is no longer identified for replacement processing, with or without subtables.

iv) Let *SUP* be the set consisting of every replacement row of every *R*. *SUP* is the *new delta table of update operation* on *T*.

b) Otherwise, for every row *R* that is identified for replacement in *T*, *R* is replaced by its corresponding replacement row. *R* is no longer identified for replacement. *T* is no longer identified for replacement processing, with or without subtables. Let *SUP* be the set consisting of every replacement row of every *R*.

Case:

i) If *T* has no new delta table of update operation, then *SUP* is the *new delta table of update operation* on *T*.

ii) Otherwise, the rows of *SUP* are added to the new delta table of update operation on *T*.

NOTE 713 — The latter scenario can arise when updating a view *V* with proper subtables, since this Subclause will be invoked for each operand of a UNION ALL CORRESPONDING in the hierarchical <query expression> of *V*. It can also arise during a <merge statement> that has more than one <merge update specification>.

11) For every pair of tables *T1* and *T2* in *TT2*, if *T1* is a direct subtable of *T2*, then the new delta table of update operation on *T1* is a direct subtable of the new delta table of update operation on *T2*.

12) For every row *BR* that has an associated for portion of from-value and the associated for portion of to-value in *S*:

a) Let *ATPN* be the period name included in the application-time period descriptor included in the descriptor of *T*. Let *FROMVAL* be the associated for portion of from-value. Let *TOVAL* be the associated for portion of to-value. Let *BSTARTVAL* be the value of the *ATPN* period start column. Let *BCD* be the declared type of the *ATPN* period start column. Let *BENDVAL* be the value of the *ATPN* period end column. Let *TN* be the table name included in the descriptor of *T*. Let *d* be the degree of *BR*.

b) For *i*, 1 (one) ≤ *i* ≤ *d*:

Case:

i) If the column descriptor that corresponds to the *i*-th field of *BR* describes an identity column, a generated column, a system-time period start column, or a system-time period end column, then let V_i be DEFAULT.

ii) Otherwise, let V_i be the value of the *i*-th field of *BR*.

c) If *BSTARTVAL* < *FROMVAL* and *FROMVAL* < *BENDVAL*, then:

i) For *j*, 1 (one) ≤ *j* ≤ *d*,

Case:

1) If the *j*-th field of *BR* corresponds to the *ATPN* period end column of *T*, then let VL_j be CAST (*FROMVAL* AS *BCD*).

2) Otherwise, let VL_j be V_j.

ii) The following <insert statement> is effectively executed without further Access Rule and constraint checking:

```
INSERT INTO TN VALUES (VL1, ..., VLd)
```

NOTE 714 — Constraint checking is performed at the end of triggering UPDATE statement.

d) If *BSTARTVAL* < *TOVAL* and *TOVAL* < *BENDVAL*, then:

i) For *k*, 1 (one) ≤ *k* ≤ *d*,

Case:

1) If the *k*-th field of *BR* corresponds to the *ATPN* period start column of *T*, then let VR_k be CAST (*TOVAL* AS *BCD*).

2) Otherwise, let VR_k be V_k.

ii) The following <insert statement> is effectively executed without further Access Rule and constraint checking:

```
INSERT INTO TN VALUES (VR1, ..., VRd)
```

NOTE 715 — Constraint checking is performed at the end of triggering UPDATE statement.

13) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.15 Effect of replacing some rows in a derived table

Function

Specify the effect of replacing some rows in a derived table.

Subclause Signature

```
"Effect of replacing some rows in a derived table" [General Rules] (
  Parameter: "TABLE",
  Parameter: "REPLACEMENT SET FOR TABLE"
)
```

TABLE — a table.

REPLACEMENT SET FOR TABLE — a set of rows to be used to replace rows in TABLE.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *QE* be the *TABLE* and let *RS* be the *REPLACEMENT SET FOR TABLE* in an application of the General Rules of this Subclause.

2) Let *T* be the result of evaluating *QE*. Let *CL* be the object columns of *QE*.

3) Case:

   a) If *QE* simply contains a <query primary> that immediately contains a <query expression body>, then let *QEB* be that <query expression body>. The General Rules of this Subclause are applied with *QEB* as *TABLE* and *RS* as *REPLACEMENT SET FOR TABLE*.

   b) If *QE* simply contains a <query expression body> *QEB* that specifies UNION ALL, let *LO* and *RO* be the <query expression body> and the <query term>, respectively, that are immediately contained in *QEB*. Let *T1* and *T2* be the tables identified by *LO* and *RO*, respectively. Let the object columns *CL1* and *CL2* be the columns of *T1* and *T2* that are underlying columns of the object columns of *CL*, respectively. Let *RS1* and *RS2* be the initially empty replacement sets for *T1* and *T2*, respectively.

      i) For every pair (*SR*, *CNR*) of *RS*,

         Case:

         1) If *SR* has been derived from a row of *T1*, then identify that row *SR1* for replacement by *CNR*; the pair (*SR1*, *CNR*) is effectively added to *RS1*.

         2) Otherwise, let *SR2* be the row of *T2* from which *SR* has been derived; identify that row for replacement by *CNR*; the pair (*SR2*, *CNR*) is effectively added to *RS2*.

ii) The General Rules of this Subclause are applied with *LO* as *TABLE* and *RS1* as *REPLACEMENT SET FOR TABLE*.

iii) The General Rules of this Subclause are applied with *RO* as *TABLE* and *RS2* as *REPLACEMENT SET FOR TABLE*.

c) Otherwise, let *QS* be the <query specification> simply contained in *QE*. Let *TE* be the <table expression> immediately contained in *QS*, and let *TREF* be the <table reference>s simply contained in the <from clause> of *TE*. Let *SL* be the <select list> immediately contained in *QS*, and let *n* be the number of <value expression>s VE_j, 1 (one) ≤ *j* ≤ *n*, simply contained in *SL*.

i) Case:

1) If *TREF* contains only one <table reference>, then let TR_1 be that <table reference>, and let *m* be 1 (one).

2) Otherwise, let *m* be the number of <table reference>s that identify tables with respect to which *QS* is one-to-one. Let TR_i, 1 (one) ≤ *i* ≤ *m*, be those <table reference>s.

ii) Let TT_i, 1 (one) ≤ *i* ≤ *m*, be the table identified by TR_i, let RS_i be an initially empty replacement set for TT_i, and let CL_i be the object column list of TT_i, such that every column of CL_i is an underlying column of *CL*.

iii) For every pair (*SR*, *CNR*) of *RS*, and for *i* ranging from 1 (one) to *m*:

1) Let *SRTI* be the row of TT_i from which *SR* has been derived.

2) A candidate row *CNRI* of TT_i is effectively created by copying *SRTI*. The candidate row includes every column of TT_i.

3) For *j* ranging from 1 (one) to *n*, let *C* be a column of some candidate row identified by VE_j, and let *SV* be the *j*-th value of *CNR*. The General Rules of Subclause 9.2, "Store assignment", are applied with *C* as *TARGET* and *SV* as *VALUE*

4) Identify *SRTI* for replacement by *CNRI*; the pair (*SRTI*, *CNRI*) is effectively added to RS_i.

iv) For *i* ranging from 1 (one) to *m*

Case:

1) If TT_i is a base table, then

Case:

A) If TR_i specifies ONLY, then TT_i is identified for replacement processing without subtables with respect to the object columns CL_i.

B) Otherwise, TT_i is identified for replacement processing with subtables with respect to the object columns CL_i.

2) If TT_i is a viewed table, then the General Rules of Subclause 15.16, "Effect of replacing some rows in a viewed table", are applied with the <table name> of TT_i as *VIEW NAME* and RS_i as *REPLACEMENT SET FOR VIEW NAME*.

3) If TT_i is a derived table, then the General Rules of this Subclause are applied with TR_i as *TABLE* and RS_i as *REPLACEMENT SET FOR TABLE*.

v) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

4) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.16 Effect of replacing some rows in a viewed table

Function

Specify the effect of replacing some rows in a viewed table.

Subclause Signature

```
"Effect of replacing some rows in a viewed table" [General Rules] (
  Parameter: "VIEW NAME",
  Parameter: "REPLACEMENT SET FOR VIEW NAME"
)
```

TABLE — a table.

REPLACEMENT SET FOR TABLE — a set of rows to be used to replace rows in TABLE.

VIEW NAME — the name of a view.

REPLACEMENT SET FOR VIEW NAME — a set of rows to be used to replace rows in the view identified by VIEW NAME.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *VN* be the *VIEW NAME* and let *RS* be the *REPLACEMENT SET FOR VIEW NAME* in an application of the General Rules of this Subclause.

2) Let *T* be the view identified by *VN*. Let *TD* be the view descriptor of *T*. If *VN* specifies ONLY, then let *QE* be the original <query expression> included in *TD*; otherwise, let *QE* be the hierarchical <query expression> included in *TD*.

3) Case:

   a) If *T* is trigger updatable, then:

      i) Let *TR* be the update INSTEAD OF trigger whose subject table is *T*.

      ii) A state change *SC* is created as follows:

         1) The set of transitions of *SC* consists of copies of row pairs in the replacement set for *T*.

         2) The trigger event of *SC* is UPDATE.

         3) The subject table of *SC* is *T*.

         4) The column list of *SC* is empty.

5) The set of statement-level triggers for which *SC* is considered as executed is empty.

6) The set of row-level triggers consists of *TR* paired with the empty set (of rows considered as executed).

iii) The General Rules of Subclause 15.21, "Execution of triggers", are applied with *TR* as *TRIGGER* and *SC* as *STATE CHANGE*.

b) Otherwise:

i) If *TD* indicates CHECK OPTION, then *TD* is added to the set of views to be checked in the current statement execution context.

ii) The General Rules of Subclause 15.15, "Effect of replacing some rows in a derived table", are applied with *QE* as *TABLE* and *RS* as *REPLACEMENT SET FOR TABLE*.

4) Let n be the number of target leaf generally underlying tables of *QE*. Let T_i, 1 (one) $\leq i \leq n$ be the target leaf generally underlying tables of *QE*. Let NT_i, 1 (one) $\leq i \leq n$ be the new delta table of update operation on T_i. Let *S* be the result of evaluating *QE* with every reference to T_i, 1 (one) $\leq i \leq n$, being replaced with a reference to NT_i. *S* is the *new delta table of update operation* on *T*.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.17 Checking of views that specify CHECK OPTION

Function

Check views that specify CHECK OPTION after an insert or update operation.

Subclause Signature

```
"Checking of views that specify CHECK OPTION" [General Rules] (
  Parameter: "OPERATION"
)
```

OPERATION — a data change operation (MERGE, INSERT, or UPDATE),

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *OP* be the *OPERATION* in an application of the General Rules of this Subclause.

2) Let *LOV* be the set of views to checked in the current execution context.

3) If *OP* is MERGE, then for every target leaf underlying table *TLUT* of every view in *LOV*, the new delta table of merge operation on *TLUT* is the multiset union of the new delta table of insert operation on *TLUT*, if any, with the new delta table of update operation on *TLUT*, if any.

NOTE 716 — If the <target table> of a <merge statement> identifies a view *V*, then the new delta table of merge operation of *V* has been defined in the General Rules of Subclause 14.12, "<merge statement>", but not for any of the base tables that the <merge statement> targets. If the operation is INSERT or UPDATE, then the relevant new delta table of the target table(s) has been defined in either Subclause 15.11, "Effect of inserting tables into base tables", or Subclause 15.14, "Effect of replacing rows in base tables".

4) For every view *V* in *LOV*:

a) Let *GGUTS* be the graph of generally underlying table specifications of the original <query expression> of *V*. Let *N* be one plus the number of generally underlying table specifications in *GGUTS* that are views. Let V_1 be *V*, and let V_2, ..., V_N be an enumeration of the views that are generally underlying table specifications of *V*.

b) For all *i*, 1 (one) ≤ *i* ≤ *N*,

i) Let *s*(*i*) be the number of subviews of V_i.

NOTE 717 — Every view is a subview of itself, therefore *s*(*i*) is at least 1 (one).

ii) For all *j*, 1 (one) ≤ *j* ≤ *s*(*i*), let $SV_{i,j}$ be an enumeration of the subviews of V_i, chosen so that $SV_{i,1}$ is V_i.

iii) For all *j*, 1 (one) ≤ *j* ≤ *s*(*i*),

1) Let $SVX_{i,j}$ be a view descriptor that is a copy of the view descriptor of $SV_{i,j}$.

2) The name of the view described by $SVX_{i,j}$ is set to a distinct effective view name $EVN_{i,j}$.

3) Case:

   A) If j = 1 (one), then $SVX_{i,j}$ is set to indicate that it has no direct superview.

      NOTE 718 — *V* can have a proper superview, but only subviews matter for checking of CHECK OPTION. Hence for each *i*, these rules construct a family of subviews whose maximal superview is $SVX_{i,1}$.

   B) Otherwise, if the direct superview of $SV_{i,j}$ is $SV_{i,k}$, then the direct superview of $SVX_{i,j}$ is set to $SVX_{i,k}$.

4) For every <table name> *TN* contained in the original <query expression> of $SVX_{i,j}$:

   Case:

   A) If *TN* references some view $SV_{f,g}$, then *TN* is replaced by $EVN_{f,g}$.

      NOTE 719 — That is, all view names in the copied view descriptors are renamed so that they reference views in the set of copied view descriptors.

   B) If *TN* references a base table *BT* and *BT* is a target leaf generally underlying table of the hierarchical <query expression> of *V*, then

      Case:

      I) If *OP* is INSERT and *BT* has a new delta table of insert operation *NDTI*, then *TN* is replaced by a distinct effective name for *NDTI*.

      II) If *OP* is UPDATE and *BT* has a new delta table of update operation *NDTU*, then *TN* is replaced by a distinct effective name for *NDTU*.

      III) If *OP* is MERGE and *BT* has a new delta table of merge operation *NDTM*, then *TN* is replaced by a distinct effective name for *NDTU*.

   C) Otherwise, *TN* is not replaced.

5) If *V* specifies LOCAL CHECK OPTION and *i* > 1 (one), then every <where clause> is removed from the original <query expression> of $SVX_{i,j}$.

   NOTE 720 — If a target leaf underlying table *TLUT* of *V* is a view that also specifies CHECK OPTION, then *TLUT* is in *LOV* and will be checked in its own right; there is no need to check it when checking *V*. If *V* specifies CASCADED CHECK OPTION, then no <where clause>s are removed, effectively checking all views that are target leaf underlying tables of *V*. Some views can be redundantly checked twice because of these considerations.

c) The General Rules of Subclause 9.38, "Generation of the hierarchical <query expression> of a view", are applied with $SVX_{1,1}$ as *VIEW*.

   NOTE 721 — This is the only hierarchical <query expression> that is required to check *V*. Note that $V = V_1 = SV_{1,1}$, which was copied and edited to form $SVX_{1,1}$. For *i* > 1 (one), only the original <query expression>s are required, though there would be no harm in forming their hierarchical <query expression>s.

d) Let *NVDX* be the number of rows in

```
SELECT *
FROM EVN1,1
```

e) Let *NDT* be the number of rows in

Case:

i) If *OP* is INSERT, then the new delta table of insert operation of the target leaf underlying table of the original <query expression> of *V*.

ii) If *OP* is UPDATE, then the new delta table of update operation of the target leaf underlying table of the original <query expression> of *V*.

iii) If *OP* is MERGE, then the new delta table of merge operation of the target leaf underlying table of the original <query expression> of *V*.

f) If *NVDX* does not equal *NDT*, then an exception condition is raised: *with check option violation (44000)*.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.18 Execution of referential actions

Function

Specify the effect of referential actions.

Subclause Signature

```
"Execution of referential actions" [General Rules] (
  Parameter: "CONSTRAINT"
)
```

CONSTRAINT — a referential constraint.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *RC* be the *CONSTRAINT* in an application of the General Rules of this Subclause.
2) *RC* is a referential constraint.
3) Let *M* be the <match type> of *RC* (either SIMPLE, PARTIAL, or FULL).
4) Let *UR* be the <update rule> of *RC* and let *DR* be the <delete rule> of *RC*.
5) Let *FF* be the referencing table of *RC*.

   Case:

   a) If *FF* is a system-versioned table, then let *F* be the result of

   ```
   SELECT *
   FROM FF FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP
   ```

   b) Otherwise, let *F* be *FF*.
6) Case:

   a) If *M* specifies SIMPLE or FULL, then for a given row in the referenced table, every row that is a subrow or a superrow of a row *R* in *F* such that the referencing column values equal the corresponding referenced column values in *R* for the referential constraint is a *matching row*.

   b) If *M* specifies PARTIAL, then:

      i) For a given row in the referenced table, every row that is a subrow or a superrow of a row *R* in *F* such that *R* has at least one non-null referencing column value and the non-null referencing column values of *R* equal the corresponding referenced column values for the referential constraint is a *matching row*.

ii) For a given row in the referenced table, every matching row for that given row that is a matching row only to the given row in the referenced table for the referential constraint is a *unique matching row*. For a given row in the referenced table, a matching row for that given row that is not a unique matching row for that given row for the referential constraint is a *non-unique matching row*.

7) For each row of the referenced table, its matching rows, unique matching rows, and non-unique matching rows are determined immediately prior to the execution of any <SQL procedure statement>. No new matching rows are added during the execution of that <SQL procedure statement>.

The association between a referenced row and a non-unique matching row is dropped during the execution of that SQL-statement if the referenced row is either marked for deletion or updated to a distinct value on any referenced column that corresponds to a non-null referencing column. This occurs immediately after such a mark for deletion or update of the referenced row. Unique matching rows and non-unique matching rows for a referenced row are evaluated immediately after dropping the association between that referenced row and a non-unique matching row.

8) Let *CTEC* be the most recent statement execution context. Let *SSC* be the set of state changes in *CTEC*. Let SC_i be a state change in *SSC*.

9) For every row of the referenced table that is marked for deletion and has not previously been marked for deletion,

Case:

a) If *M* specifies SIMPLE or FULL, then

Case:

i) If *DR* specifies CASCADE, then:

1) *F* is identified for deletion processing with subtables and every matching row in *F* is identified for deletion.

2) The General Rules of Subclause 15.8, "Effect of deleting rows from base tables", are applied.

ii) If *DR* specifies SET NULL, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the null value. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iii) If *DR* specifies SET DEFAULT, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting each referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iv) If *DR* specifies RESTRICT and there exists some matching row, then an exception condition is raised: *integrity constraint violation — restrict violation (23001)*.

NOTE 722 — If *DR* specifies NO ACTION, then no referential delete action is performed.

b) If *M* specifies PARTIAL, then

Case:

i) If *DR* specifies CASCADE, then:

1) *F* is identified for deletion processing with subtables and every unique matching row in *F* is identified for deletion.

2) The General Rules of Subclause 15.8, "Effect of deleting rows from base tables", are applied.

ii) If *DR* specifies SET NULL, then:

1) Each unique matching row *UMR* in *F* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting each referencing column in the copy to the null value. *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iii) If *DR* specifies SET DEFAULT, then:

1) Each unique matching row *UMR* in *F* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting each referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iv) If *DR* specifies RESTRICT and there exists some unique matching row, then an exception condition is raised: *integrity constraint violation — restrict violation (23001)*.

NOTE 723 — If *DR* specifies NO ACTION, then no referential delete action is performed.

10) If a non-null value of a referenced column *RC* in the referenced table is updated to a value that is distinct from the current value of *RC*, then,

Case:

a) If *M* specifies SIMPLE or FULL, then

Case:

i) If *UR* specifies CASCADE, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting in the copy each referencing column corresponding to an updated referenced column to the new value of that referenced column. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the modified referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

ii) If *UR* specifies SET NULL, then

Case:

1) If *M* specifies SIMPLE, then:

A) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting in the copy each referencing column corresponding to an updated referenced column to the null value. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

B) *F* is identified for replacement processing with subtables with respect to the modified referencing columns.

C) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

2) If *M* specifies FULL, then:

A) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting in the copy each referencing column of *RC* to the null value. *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

B) *F* is identified for replacement processing with subtables with respect to the referencing columns of *RC*.

C) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iii) If *UR* specifies SET DEFAULT, then:

1) Each matching row *MR* in *F* is paired with the candidate replacement row *NMR*, formed by copying *MR* and setting in the copy each referencing column corresponding to an updated referenced column to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *MR* is identified for replacement by *NMR* in *F*. The set of (*MR*, *NMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the modified referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iv) If *UR* specifies RESTRICT and there exists some matching row, then an exception condition is raised: *integrity constraint violation — restrict violation (23001)*.

NOTE 724 — If *UR* specifies NO ACTION, then no referential update action is performed.

b) If *M* specifies PARTIAL, then

Case:

i) If *UR* specifies CASCADE, then:

1) Each unique matching row *UMR* in *F* that contains a non-null value in the referencing column *C1* in *F* that corresponds to the updated referenced column *C2* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting *C1* in the copy to the new value *V* of *C2*, provided that, in all updated rows in the referenced table that formerly had, during the same execution of the same innermost SQL-statement, that unique matching row as a matching row, the values in *C2* have all been updated to a value that is not distinct from *V*. If this last condition is not satisfied, then an exception condition is raised: *triggered data change violation (27000)*. *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

NOTE 725 — Because of the Rules of Subclause 8.2, "<comparison predicate>", on which the definition of "distinct" relies, the values in *C2* can have been updated to values that are not distinct, yet are not identical. Which of these non-distinct values is used for the cascade operation is implementation-dependent.

2) *F* is identified for replacement processing with subtables with respect to the modified referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

ii) If *UR* specifies SET NULL, then:

1) Each unique matching row *UMR* in *F* that contains a non-null value in the referencing column in *F* is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting that referencing column in the copy to the null value. *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the modified referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iii) If *UR* specifies SET DEFAULT, then:

1) Each unique matching row *UMR* in *F* that contains a non-null value in the referencing column in *F* that corresponds with the updated referenced column is paired with the candidate replacement row *NUMR*, formed by copying *UMR* and setting that referencing column in the copy to the default value specified in the General Rules of Subclause 11.5, "<default clause>". *UMR* is identified for replacement by *NUMR* in *F*. The set of (*UMR*, *NUMR*) pairs is the replacement set for *F*.

2) *F* is identified for replacement processing with subtables with respect to the modified referencing columns.

3) The General Rules of Subclause 15.14, "Effect of replacing rows in base tables", are applied.

iv) If *UR* specifies RESTRICT and there exists some unique matching row, then an exception condition is raised: *integrity constraint violation — restrict violation (23001)*.

NOTE 726 — If *UR* specifies NO ACTION, then no referential update action is performed.

11) Let *ISS* be the innermost SQL-statement being executed.

12) If evaluation of these General Rules during the execution of *ISS* would cause an update of some site to a value that is distinct from the value to which that site was previously updated during the execution of *ISS*, then an exception condition is raised: *triggered data change violation (27000)*.

13) If evaluation of these General Rules during the execution of *ISS* would cause deletion of a row containing a site that is identified for replacement in that row, then an exception condition is raised: *triggered data change violation (27000)*.

14) If evaluation of these General Rules during the execution of *ISS* would cause either an attempt to update a row that has been deleted by any <delete statement: positioned>, <dynamic delete statement: positioned>, or <preparable dynamic delete statement: positioned> that identifies some open cursor *CR* or that has been updated by any <update statement: positioned>, <dynamic update statement: positioned>, or <preparable dynamic update statement: positioned> that identifies some open cursor *CR*, or an attempt to mark for deletion such a row, then a completion condition is raised: *warning — cursor operation conflict (01001)*.

15) If the subject table restriction flag of the current SQL-session context is set to *True*, and if evaluation of these General Rules during the execution of *ISS* would cause an insertion of some row, update of some row, or deletion of some row in a table whose name is included in the restricted subject table name list included in the current SQL-session context, then an exception condition is raised: *triggered data change violation — modify table modified by data change delta table (27001)*.

16) For every row *RMD* that is marked for deletion, every subrow of *RMD* and every superrow of *RMD* is marked for deletion.

17) If any table is the subject table of a state change in *SSC* that has been created or modified during evaluation of the preceding General Rules of this subclause, then, for every referential constraint descriptor *RC2* of an enforced referential constraint, the General Rules of this Subclause are applied with *RC2* as *CONSTRAINT*.

NOTE 727 — Thus these rules are repeatedly evaluated until no further transitions are generated.

18) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.19 Execution of BEFORE triggers

Function

Define the execution of BEFORE triggers.

Subclause Signature

```
"Execution of BEFORE triggers" [General Rules] (
  Parameter: "SET OF STATE CHANGES"
)
```

SET OF STATE CHANGES — a set of state changes.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *SCC* be the *SET OF STATE CHANGES* in an application of the General Rules of this Subclause.

2) Let *BT* be the set of BEFORE triggers that are activated by some state change in *SSC*.

NOTE 728 — Activation of triggers is defined in Subclause 4.46, "Triggers".

3) Let *NT* be the number of triggers in *BT* and let TR_k be the *k*-th such trigger, ordered according to their order of execution. Let SC_k be the state change in *SSC* that activated TR_k.

NOTE 729 — Ordering of triggers is defined in Subclause 4.46, "Triggers".

4) For *k* ranging from 1 (one) to *NT*, the General Rules of Subclause 15.21, "Execution of triggers", are applied with TR_k as *TRIGGER* and SC_k as *STATE CHANGE*.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.20 Execution of AFTER triggers

Function

Define the execution of AFTER triggers.

Subclause Signature

```
"Execution of AFTER triggers" [General Rules] (
  Parameter: "SET OF STATE CHANGES"
)
```

SET OF STATE CHANGES — a set of state changes.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *SSC* be the *SET OF STATE CHANGES* in an application of the General Rules of this Subclause.

2) Let *AT* be the set of AFTER triggers that are activated by some state change in *SSC*.

NOTE 730 — Activation of triggers is defined in Subclause 4.46, "Triggers".

3) Let *NT* be the number of triggers in *AT* and let TR_k be the *k*-th such trigger, ordered according to their order of execution. Let SC_k be the state change in *SSC* that activated TR_k.

NOTE 731 — Ordering of triggers is defined in Subclause 4.46, "Triggers".

4) For *k* ranging from 1 (one) to *NT*, the General Rules of Subclause 15.21, "Execution of triggers", are applied with TR_k as *TRIGGER* and SC_k as *STATE CHANGE*.

5) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

15.21 Execution of triggers

Function

Define the execution of triggers.

Subclause Signature

```
"Execution of triggers" [General Rules] (
  Parameter: "TRIGGER",
  Parameter: "STATE CHANGE"
)
```

TRIGGER — a trigger.

STATE CHANGE — a state change.

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *TR* be the *TRIGGER* and let *SC* be the *STATE CHANGE* in an application of the General Rules of this Subclause.

2) Except where explicitly specified, the General Rules of this Subclause are not terminated if an exception condition is raised.

3) Let *TA* be the triggered action included in the trigger descriptor of *TR*. Let *TSS* be the <triggered SQL statement> contained in *TA*. Let *TE* be the trigger event of *SC*. Let *ST* be the set of transitions in *SC*.

4) *TR* is executed as follows.

   Case:

   a) If *TR* is a row-level trigger, then, for each transition *T* in *ST* for which *TR* is not considered as executed, *TA* is invoked and *TR* is considered as executed for *T*. The order in which the transitions in *ST* are taken is implementation-dependent (US048).

   b) If *TR* is not considered as executed for *SC*, then *TA* is invoked once and *TR* is considered as executed for *SC*.

5) When *TA* is invoked:

   a) Case:

      i) If *TE* is DELETE, then the old transition table for the invocation of *TA* is *ST*. If *TR* is a row-level trigger, then the value of the old transition variable for the invocation of *TA* is *T*.

ii) If *TE* is INSERT, then the new transition table for the invocation of *TA* is *ST*. If *TR* is a row-level trigger, then the value of the new transition variable for the invocation of *TA* is *T*.

iii) If *TE* is UPDATE, then the old transition table for the invocation of *TA* is the multiset formed by taking the old rows of the transitions in *ST* and the new transition table for the invocation of *TA* is the multiset formed by taking the new rows of the transitions in *ST*. If *TR* is a row-level trigger, then the value of the old transition variable for the invocation of *TA* is the old row of *T* and the new transition variable for the invocation of *TA* is the new row of *T*.

b) Case:

i) If *TA* contains a <search condition> *TASC*, then *TASC* is evaluated.

Case:

1) If, during the evaluation of *TASC*, a <column reference> that references the system-time period start column or the system-period period end column of the transition table is encountered or an exception condition is raised, then an exception condition is raised: *triggered action exception (09000)*, and no further General Rules of this Subclause are applied.

2) If the result of evaluating *TASC* is *True*, then *TSS* is executed.

3) Otherwise, *TSS* is not executed.

ii) If *TA* does not contain a <search condition>, then *TSS* is executed.

6) When *TSS* is executed:

a) The General Rules of Subclause 23.2, "Pushing and popping the diagnostics area stack", are applied with "PUSH" as *OPERATION* and the diagnostics area stack in the current SQL-session context *CSC* as *STACK*.

b) The authorization identifier of the owner of the schema that includes the trigger descriptor of *TR* is pushed onto the authorization stack.

c) A new savepoint level is established.

d) Indicate that another triggered action is executing in the current SQL-session context.

e) Let *N* be the number of <SQL procedure statement>s simply contained in *TSS*. For *i* ranging from 1 (one) to *N*:

i) Let S_i be the *i*-th such <SQL procedure statement>.

ii) The General Rules of Subclause 9.17, "Executing an <SQL procedure statement>", are applied with S_i as *EXECUTING STATEMENT*.

iii) If, before the completion of S_i, an attempt is made to execute an SQL-schema statement, an SQL-connection statement, an SQL-transaction statement, an SQL-dynamic statement, or an SQL-session statement, then an exception condition is raised: *prohibited statement encountered during trigger execution (0W000)*.

iv) If the execution of S_i raises an exception, then evaluation of GR 6)e) is terminated immediately and evaluation continues with GR 6)f).

f) Indicate that this triggered action is no longer executing in the current SQL-session context.

g) If, before the completion of the execution of any <SQL procedure statement> simply contained in *TSS*, an attempt is made to evaluate a <column reference> that references the system-time period start column or the system-time period end column of the old transition table, then an exception condition is raised: *prohibited column reference encountered during trigger execution (11000)*.

h) If *TR* is a BEFORE trigger and if, before the completion of the execution of any <SQL procedure statement> simply contained in *TSS*, an attempt is made to execute an SQL-statement that possibly modifies SQL-data, then an exception condition is raised: *prohibited statement encountered during trigger execution (0W000)*.

i) If *TR* is an AFTER trigger and the subject table restriction flag of the current SQL-session context is set to *True*, and if during the execution of *TR* any attempt is made to insert a row, update a row, or delete a row in a table whose name is included in the restricted subject table name list included in the current SQL-session context, then an exception condition is raised: *prohibited statement encountered during trigger execution — modify table modified by data change delta table (0W001)*.

j) The current savepoint level is destroyed.

NOTE 732 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

k) The General Rules of Subclause 23.2, "Pushing and popping the diagnostics area stack", are applied with "POP" as *OPERATION* and the diagnostics area stack in *CSC* as *STACK*.

l) The top cell in the authorization stack is removed.

m) If the execution of *TSS* is not successful, then an exception condition is raised: *triggered action exception (09000)*. The exception condition that caused *TSS* to fail is raised. If *TR* is a row-level trigger, then no further transitions in *ST* are processed.

NOTE 733 — Raising the exception condition that caused *TSS* to fail enters the exception information into the diagnostics area that was pushed prior to the execution of *TSS*.

7) Evaluation of the General Rules is terminated and control is returned to the invoking Subclause.

Conformance Rules

None.

16 Control statements

This Clause is modified by Clause 15, "Control statements", in ISO/IEC 9075-4.
This Clause is modified by Clause 9, "Control statements", in ISO/IEC 9075-10.

16.1 <call statement>

This Subclause is modified by Subclause 9.1, "<call statement>", in ISO/IEC 9075-10.

Function

Invoke an SQL-invoked routine.

Format

```
<call statement> ::=
  CALL <routine invocation>
```

Syntax Rules

1) Let *RI* be the <routine invocation> immediately contained in the <call statement>.

2) The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *RI* as *ROUTINE INVOCATION*, an empty <schema name> list as *SQLPATH*, and the null value as *UDT*; let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* returned from the application of those Syntax Rules.

   NOTE 734 — The *SAL* returned is not used.

3) *SR* shall be an SQL-invoked procedure.

Access Rules

None.

General Rules

1) The General Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*; let *V* be the *VALUE* returned from the application of those General Rules.

   NOTE 735 — The *V* returned is not used.

Conformance Rules

None.

16.2 <return statement>

Function

Return a value from an SQL routine that is an SQL-invoked function.

Format

```
<return statement> ::=
  RETURN <return value>

<return value> ::=
    <value expression>
  | NULL
```

Syntax Rules

1) <return statement> shall be contained in an SQL routine body that is simply contained in the <routine body> of an <SQL-invoked function> *F*. Let *RDT* be the <returns data type> of the <returns clause> of *F*.

2) The <return value> <null specification> is equivalent to the <value expression>:

   ```
   CAST (NULL AS RDT)
   ```

3) Let *VE* be the <value expression> of the <return value> immediately contained in <return statement>.

4) The Syntax Rules of Subclause 9.2, "Store assignment", are applied with an item of the data type *RDT* as *TARGET* and *VE* as *VALUE*.

Access Rules

None.

General Rules

1) The value of *VE* is the *returned value* of the execution of the SQL routine body of *F*.

2) The execution of the SQL routine body of *F* is terminated immediately.

Conformance Rules

None.

17 Transaction management

This Clause is modified by Clause 10, "Transaction management", in ISO/IEC 9075-10.

17.1 <start transaction statement>

Function

Start an SQL-transaction and set its characteristics.

Format

```
<start transaction statement> ::=
  START TRANSACTION [ <transaction characteristics> ]
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction (25001)*.

2) Case:

   a) If <transaction characteristics> is omitted, then let *TC* be

      ECAM ISOLATION LEVEL *ECIL* DIAGNOSTICS SIZE *ECNC*

      where *ECAM*, *ECIL*, and *ECNC* are the transaction access mode, transaction isolation level and number of conditions, respectively, of the enduring transaction characteristics of the current SQL-session.

   b) Otherwise, let *TC* be the <transaction characteristics>.

3) If <number of conditions> is specified and is less than 1 (one), then an exception condition is raised: *invalid condition number (35000)*.

4) The <set transaction statement>

   SET TRANSACTION *TC*

   is effectively executed.

   NOTE 736 — The characteristics of a transaction begun by a <start transaction statement> are as specified here regardless of the characteristics specified by any preceding <set transaction statement>. That is, even if one or more characteristics are omitted by the <start transaction statement>, the defaults specified in the Syntax Rules of this

Subclause and of Subclause 17.3, "<transaction characteristics>", are effective and are not affected by any (preceding) <set transaction statement> in the same SQL-session.

5) An SQL-transaction is initiated.

Conformance Rules

1) Without Feature T241, "START TRANSACTION statement", conforming SQL language shall not contain a <start transaction statement>.

17.2 <set transaction statement>

This Subclause is modified by Subclause 10.1, "<set transaction statement>", in ISO/IEC 9075-10.

Function

Set the characteristics of the next SQL-transaction for the SQL-agent.

NOTE 737 — This statement has no effect on any SQL-transactions subsequent to the next SQL-transaction.

Format

```
<set transaction statement> ::=
  SET [ LOCAL ] TRANSACTION <transaction characteristics>
```

Syntax Rules

1) [10] If LOCAL is specified, then <transaction characteristics> shall not contain <number of conditions>.

Access Rules

None.

General Rules

1) Case:

   a) If a <set transaction statement> that does not specify LOCAL is executed, then

      Case:

      i) If an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction (25001)*.

      ii) If an SQL-transaction is not currently active, then if there are any holdable cursors remaining open from the previous SQL-transaction and the transaction isolation level of the previous SQL-transaction is not the same as the transaction isolation level determined by the <level of isolation>, then an exception condition is raised: *invalid transaction state — held cursor requires same isolation level (25008)*.

   b) If a <set transaction statement> that specifies LOCAL is executed, then:

      i) If the SQL-implementation does not support Feature T262, "Multiple server transactions", then an exception condition is raised: *feature not supported — multiple server transactions (0A001)*.

      ii) If there is no SQL-transaction that is currently active, then an exception condition is raised: *invalid transaction state — no active SQL-transaction for branch transaction (25005)*.

      iii) If there is an active SQL-transaction and there has been a transaction-initiating SQL-statement executed at the current SQL-connection in the context of the active SQL-transaction, then an exception condition is raised: *invalid transaction state — branch transaction already active (25002)*.

iv) If the transaction access mode of the SQL-transaction is read-only and <transaction access mode> specifies READ WRITE, then an exception condition is raised: *invalid transaction state — inappropriate access mode for branch transaction (25003)*.

v) If the transaction isolation level of the SQL-transaction is SERIALIZABLE and <level of isolation> specifies anything except SERIALIZABLE, then an exception condition is raised: *invalid transaction state — inappropriate isolation level for branch transaction (25004)*.

vi) If the transaction isolation level of the SQL-transaction is REPEATABLE READ and <level of isolation> specifies anything except REPEATABLE READ or SERIALIZABLE, then an exception condition is raised: *invalid transaction state — inappropriate isolation level for branch transaction (25004)*.

vii) If the transaction isolation level of the SQL-transaction is READ COMMITTED and <level of isolation> specifies READ UNCOMMITTED, then an exception condition is raised: *invalid transaction state — inappropriate isolation level for branch transaction (25004)*.

NOTE 738 — If the transaction isolation level of the SQL-transaction is READ UNCOMMITTED, then any <level of isolation> is permissible.

2) If <number of conditions> is specified and is less than 1 (one), then an exception condition is raised: *invalid condition number (35000)*.

3) If <number of conditions> is specified and is greater than the implementation-dependent (UL003) maximum value for <number of conditions> *IDMVNC*, then the implicit <number of conditions> is *IDMVNC*, and a completion condition is raised: *warning — invalid number of conditions (01012)*.

4) Let *TC* be <transaction characteristics>. Let *CSC* be the current SQL-session context.

5) If the explicit or implicit <transaction access mode> contains READ ONLY, then the current transaction access mode of *CSC* is set to *read-only*. Otherwise, the current transaction access mode of *CSC* is set to *read-write*.

6) The current transaction isolation level of *CSC* is set to an implementation-defined (IV222) transaction isolation level that will not exhibit any of the phenomena that the explicit or implicit <level of isolation> contained in *TC* would not exhibit, as specified in Table 11, "SQL-transaction isolation levels and the three phenomena".

7) The current condition area limit of *CSC* is set to the explicit or implicit <number of conditions> contained in *TC*.

Conformance Rules

1) Without Feature T251, "SET TRANSACTION statement: LOCAL option", conforming SQL language shall not contain a <set transaction statement> that immediately contains LOCAL.

17.3 <transaction characteristics>

Function

Specify transaction characteristics.

Format

```
<transaction characteristics> ::=
      [ <transaction mode> [ { <comma> <transaction mode> }... ] ]

<transaction mode> ::=
    <isolation level>
  | <transaction access mode>
  | <diagnostics size>

<transaction access mode> ::=
    READ ONLY
  | READ WRITE

<isolation level> ::=
  ISOLATION LEVEL <level of isolation>

<level of isolation> ::=
    READ UNCOMMITTED
  | READ COMMITTED
  | REPEATABLE READ
  | SERIALIZABLE

<diagnostics size> ::=
  DIAGNOSTICS SIZE <number of conditions>

<number of conditions> ::=
  <simple value specification>
```

Syntax Rules

1) Let TC be the <transaction characteristics>.

2) TC shall contain at most one <isolation level>, at most one <transaction access mode>, and at most one <diagnostics size>.

3) If TC does not contain an <isolation level>, then ISOLATION LEVEL SERIALIZABLE is implicit.

4) If <transaction access mode> is READ WRITE, then the <level of isolation> shall not be READ UNCOMMITTED.

5) If TC does not contain a <transaction access mode>, then

   Case:

   a) If <isolation level> contains READ UNCOMMITTED, then READ ONLY is implicit.

   b) Otherwise, READ WRITE is implicit.

6) The declared type of <number of conditions> shall be exact numeric with scale 0 (zero).

7) If *TC* does not contain a <diagnostics size>, then DIAGNOSTICS SIZE *n* is implicit, where *n* is an implementation-dependent (UL003) value not less than 1 (one).

Access Rules

None.

General Rules

None.

Conformance Rules

1) Without Feature F111, "Isolation levels other than SERIALIZABLE ", conforming SQL language shall not contain an <isolation level> that contains a <level of isolation> other than SERIALIZABLE.

2) Without Feature F112, "Isolation level READ UNCOMMITTED", conforming SQL language shall not contain an <isolation level> that contains a <level of isolation> of READ UNCOMMITTED.

3) Without Feature F113, "Isolation level READ COMMITTED", conforming SQL language shall not contain an <isolation level> that contains a <level of isolation> of READ COMMITTED.

4) Without Feature F114, "Isolation level REPEATABLE READ", conforming SQL language shall not contain an <isolation level> that contains a <level of isolation> of REPEATABLE READ.

5) Without Feature F124, "SET TRANSACTION statement: DIAGNOSTICS SIZE clause", conforming SQL language shall not contain a <diagnostics size>.

17.4 <set constraints mode statement>

Function

If an SQL-transaction is currently active, then set the constraint mode for that SQL-transaction in the current SQL-session. If no SQL-transaction is currently active, then set the constraint mode for the next SQL-transaction in the current SQL-session for the SQL-agent.

NOTE 739 — This statement has no effect on any SQL-transactions subsequent to this SQL-transaction.

Format

```
<set constraints mode statement> ::=
  SET CONSTRAINTS <constraint name list> { DEFERRED | IMMEDIATE }

<constraint name list> ::=
    ALL
  | <constraint name> [ { <comma> <constraint name> }... ]
```

Syntax Rules

1) If a <constraint name> is specified, then it shall identify a constraint.

2) The constraint identified by <constraint name> shall be deferrable.

Access Rules

None.

General Rules

1) Let *CSC* be the current SQL-session context.

2) If IMMEDIATE is specified, then

   Case:

   a) If ALL is specified, then the constraint mode in *CSC* of all constraints that are deferrable is set to immediate.

   b) Otherwise, the constraint mode in *CSC* for the constraints identified by the <constraint name>s in the <constraint name list> is set to immediate.

3) If DEFERRED is specified, then

   Case:

   a) If ALL is specified, then the constraint mode in *CSC* of all constraints that are deferrable is set to deferred.

   b) Otherwise, the constraint mode in *CSC* for the constraints identified by the <constraint name>s in the <constraint name list> is set to deferred.

Conformance Rules

1) Without Feature F721, “Deferrable constraints”, conforming SQL language shall not contain a <set constraints mode statement>.

17.5 <savepoint statement>

This Subclause is modified by Subclause 10.2, "<savepoint statement>", in ISO/IEC 9075-10.

Function

Establish a savepoint.

Format

```
<savepoint statement> ::=
  SAVEPOINT <savepoint specifier>

<savepoint specifier> ::=
  <savepoint name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the <savepoint name>.

2) If *S* identifies an existing savepoint established within the current savepoint level, then that savepoint is destroyed.

3) If the number of savepoints that now exist within the current SQL-transaction is equal to the implementation-defined (II208) maximum number of savepoints per SQL-transaction, then an exception condition is raised: *savepoint exception — too many (3B002)*.

4) A savepoint is established in the current savepoint level and at the current point in the current SQL-transaction. *S* is assigned as the identifier of that savepoint.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <savepoint statement>.

17.6 <release savepoint statement>

This Subclause is modified by Subclause 10.3, "<release savepoint statement>", in ISO/IEC 9075-10.

Function

Destroy a savepoint.

Format

```
<release savepoint statement> ::=
  RELEASE SAVEPOINT <savepoint specifier>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *S* be the <savepoint name>.

2) If *S* does not identify a savepoint established in the current savepoint level, then an exception condition is raised: *savepoint exception — invalid specification (3B001)*.

3) The savepoint identified by *S* and all savepoints established in the current savepoint level subsequent to the establishment of *S* are destroyed.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <release savepoint statement>.

17.7 <commit statement>

This Subclause is modified by Subclause 10.4, "<commit statement>", in ISO/IEC 9075-10.

Function

Terminate the current SQL-transaction with commit.

Format

```
<commit statement> ::=
  COMMIT [ WORK ] [ AND [ NO ] CHAIN ]
```

Syntax Rules

1) [10] If neither AND CHAIN nor AND NO CHAIN is specified, then AND NO CHAIN is implicit.

Access Rules

None.

General Rules

1) If the current SQL-transaction is part of an encompassing transaction that is controlled by an agent other than the SQL-agent, then an exception condition is raised: *invalid transaction termination (2D000)*.

2) If the current SQL-session has an atomic execution context, then an exception condition is raised: *invalid transaction termination (2D000)*.

3) For every open cursor *CR* that is not a holdable cursor in the stack of contexts of the current SQL-session, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

4) For every temporary table in any SQL-client module associated with the current SQL-transaction that specifies the ON COMMIT DELETE option and that was updated by the current SQL-transaction, the execution of the <commit statement> is effectively preceded by the execution of a <delete statement: searched> that specifies DELETE FROM *T*, where *T* is the <table name> of that temporary table.

5) The effects specified in the General Rules of Subclause 17.4, "<set constraints mode statement>", occur as if the statement SET CONSTRAINTS ALL IMMEDIATE were executed for each active SQL-connection.

6) Case:

   a) If any enforced constraint is not satisfied, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback — integrity constraint violation (40002)*.

   b) If any other error preventing commitment of the SQL-transaction has occurred, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback (40000)* with an implementation-defined (IC021) subclass value.

c) Otherwise, any changes to SQL-data or schemas that were made by the current SQL-transaction are eligible to be perceived by all concurrent and subsequent SQL-transactions.

7) All savepoint levels are destroyed and a new savepoint level is established.

NOTE 740 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

8) Every valid non-holdable locator value is marked invalid.

9) The current SQL-transaction is terminated.

10) Case:

a) If <commit statement> contains AND CHAIN, then an SQL-transaction is initiated. Any branch transactions of the SQL-transaction are initiated with the same transaction access mode, transaction isolation level, and condition area limit as the corresponding branch of the SQL-transaction just terminated.

b) Otherwise:

i) The current transaction access mode, current transaction isolation level, and current condition area limit of the current SQL-session context are set to the transaction access mode, transaction isolation level, and condition area limit, respectively, of the enduring transaction characteristics of the current SQL-session.

ii) For every constraint *C*, the constraint mode of *C* in the current SQL-session context is set to the initial constraint mode included in the constraint descriptor for *C*.

11) The prepared statement of every held cursor remains in existence. It is implementation-defined (IA169) whether or not any other prepared statement is deallocated.

Conformance Rules

1) Without Feature T261, “Chained transactions”, conforming SQL language shall not contain a <commit statement> that immediately contains CHAIN.

17.8 <rollback statement>

This Subclause is modified by Subclause 10.5, "<rollback statement>", in ISO/IEC 9075-10.

Function

Terminate the current SQL-transaction with rollback, or rollback all actions affecting SQL-data and/or schemas since the establishment of a savepoint.

Format

```
<rollback statement> ::=
  ROLLBACK [ WORK ] [ AND [ NO ] CHAIN ] [ <savepoint clause> ]

<savepoint clause> ::=
  TO SAVEPOINT <savepoint specifier>
```

Syntax Rules

1) If AND CHAIN is specified, then <savepoint clause> shall not be specified.

2) If neither AND CHAIN nor AND NO CHAIN is specified, then AND NO CHAIN is implicit.

Access Rules

None.

General Rules

1) If the current SQL-transaction is part of an encompassing transaction that is controlled by an agent other than the SQL-agent and the <rollback statement> is not being implicitly executed, then an exception condition is raised: *invalid transaction termination (2D000)*.

2) If a <savepoint clause> is not specified, then:

   a) If the current SQL-session has an atomic execution context, then an exception condition is raised: *invalid transaction termination (2D000)*.

   b) All changes to SQL-data or schemas that were made by the current SQL-transaction are canceled.

   c) All savepoint levels are destroyed and a new savepoint level is established.

      NOTE 741 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

   d) Every valid locator is marked invalid.

   e) For every open cursor *CR* in the stack of contexts of the current SQL-session, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

   f) It is implementation-defined (IA167) whether or not any prepared statement is deallocated.

   g) The current SQL-transaction is terminated.

   h) Case:

i) If <rollback statement> contains AND CHAIN, then an SQL-transaction is initiated. Any branch transactions of the SQL-transaction are initiated with the same transaction access mode, transaction isolation level, and condition area limit as the corresponding branch of the SQL-transaction just terminated.

ii) Otherwise:

1) The current transaction access mode, current transaction isolation level, and current condition area limit of the current SQL-session context are set to the transaction access mode, transaction isolation level, and condition area limit, respectively, of the enduring transaction characteristics of the current SQL-session.

2) For every constraint *C*, the constraint mode of *C* in the current SQL-session context is set to the initial constraint mode included in the constraint descriptor for *C*.

3) If a <savepoint clause> is specified, then:

a) Let *S* be the <savepoint name>.

b) If *S* does not specify a savepoint established within the current savepoint level, then an exception condition is raised: *savepoint exception — invalid specification (3B001)*.

c) If the current SQL-session has an atomic execution context, and *S* specifies a savepoint established before the beginning of the most recent atomic execution context, then an exception condition is raised: *savepoint exception — invalid specification (3B001)*.

d) All changes to SQL-data or schemas that were made by the current SQL-transaction subsequent to the establishment of *S* are canceled.

e) All savepoints established by the current SQL-transaction subsequent to the establishment of *S* are destroyed.

NOTE 742 — Destroying a savepoint level destroys all existing savepoints that are established at that level.

f) Every valid locator that was generated in the current SQL-transaction subsequent to the establishment of *S* is marked invalid.

g) For every open cursor *CR* that is not a holdable cursor in the stack of contexts of the current SQL-session, the General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

h) The status of any open cursors in the stack of contexts of the current SQL-session that were opened by the current SQL-transaction before the establishment of *S* is implementation-defined (IA168).

i) It is implementation-defined (IA167) whether or not any prepared statement that was prepared before the establishment of *S* is deallocated.

j) It is implementation-defined (IA167) whether or not any prepared statement that was prepared subsequent to the establishment of *S* is deallocated.

NOTE 743 — The current SQL-transaction is not terminated, and there is no other effect on the SQL-data or schemas.

Conformance Rules

1) Without Feature T271, "Savepoints", conforming SQL language shall not contain a <savepoint clause>.

2) Without Feature T261, "Chained transactions", conforming SQL language shall not contain a <rollback statement> that immediately contains CHAIN.

18 Connection management

18.1 <connect statement>

Function

Establish an SQL-session.

Format

```
<connect statement> ::=
  CONNECT TO <connection target>

<connection target> ::=
    <SQL-server name> [ AS <connection name> ] [ USER <connection user name> ]
  | DEFAULT
```

Syntax Rules

1) If <connection user name> is not specified, then an implementation-defined (ID238) <connection user name> for the SQL-connection is implicit.

Access Rules

None.

General Rules

1) If a <connect statement> is executed after the first transaction-initiating SQL-statement executed by the current SQL-transaction and the SQL-implementation does not support Feature T262, "Multiple server transactions",then an exception condition is raised: *feature not supported — multiple server transactions (0A001)*.

2) If <connection user name> is specified, then let *S* be <connection user name> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

3) If *V* does not conform to the Format and Syntax Rules of a <user identifier>, then an exception condition is raised: *invalid authorization specification (28000)*.

4) If the SQL-client module that contains the <externally-invoked procedure> that contains the <connect statement> specifies a <module authorization identifier>, then whether or not <connection user name> shall be identical to that <module authorization identifier> is implementation-defined (IA166), as are any other restrictions on the value of <connection user name>. Otherwise, any restrictions on the value of <connection user name> are implementation-defined (IA166).

5) If the value of <connection user name> does not conform to the implementation-defined (IA064) restrictions, then an exception condition is raised: *invalid authorization specification (28000)*.

6) If <connection name> was specified, then let *CV* be <simple value specification> immediately contained in <connection name>. If neither DEFAULT nor <connection name> were specified, then let *CV* be <SQL-server name>. Let *CN* be the result of

```
TRIM ( BOTH ' ' FROM CV )
```

If *CN* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid connection name (2E000)*.

7) If an SQL-connection with name *CN* has already been established by the current SQL-agent and has not been disconnected, or if DEFAULT is specified and a default SQL-connection has already been established by the current SQL-agent and has not been disconnected, then an exception condition is raised: *connection exception — connection name in use (08002)*.

8) Case:

   a) If DEFAULT is specified, then the default SQL-session is initiated and associated with the default SQL-server. The method by which the default SQL-server is determined is implementation-defined (IW154).

   b) Otherwise, an SQL-session is initiated and associated with the SQL-server identified by <SQL-server name>. The method by which <SQL-server name> is used to determine the appropriate SQL-server is implementation-defined (IW154).

9) If the <connect statement> successfully initiates an SQL-session, then:

   a) The current SQL-connection *CC* and current SQL-session, if any, become a dormant SQL-connection and a dormant SQL-session, respectively. The SQL-session context for *CC* is preserved and is not affected in any way by operations performed over the initiated SQL-connection.

      NOTE 744 — The SQL-session context is defined in Subclause 4.45, "SQL-sessions".

   b) The SQL-session initiated by the <connect statement> becomes the current SQL-session and the SQL-connection established to that SQL-session becomes the current SQL-connection.

   c) The transaction access mode, transaction isolation level, and condition area limit of the enduring transaction characteristics of the current SQL-session are read-write, SERIALIZABLE, and an implementation-dependent (UL003) value not less than 1 (one), respectively.

   NOTE 745 — If the <connect statement> fails to initiate an SQL-session, then the current SQL-connection and current SQL-session, if any, remain unchanged.

10) If the SQL-client cannot establish the SQL-connection, then an exception condition is raised: *connection exception — SQL-client unable to establish SQL-connection (08001)*.

11) If the SQL-server rejects the establishment of the SQL-connection, then an exception condition is raised: *connection exception — SQL-server rejected establishment of SQL-connection (08004)*.

12) The SQL-server for the subsequent execution of <externally-invoked procedure>s in any SQL-client modules associated with the SQL-agent is set to the SQL-server identified by <SQL-server name>.

13) The current SQL-session context of the current SQL-session is initialized as follows:

   a) The authorization stack is set to a single cell containing the user identifier <connection user name>.

   b) The current transaction access mode, current transaction isolation level, and current condition area limit are set to the transaction access mode, transaction isolation level, and condition area limit, respectively, of the enduring transaction characteristics of the current SQL-session.

   c) The SQL-session identifier is set to a unique implementation-dependent (UV109) value.

d) The SQL-session user identifier is set to the value of the implicit or explicit <connection user name>.

e) The condition area limit is set to an implementation-dependent (UL003) value that shall be at least 1 (one).

f) There are no collations, temporary table identities, open cursor instance descriptors, valid locators, prepared statements, SQL dynamic descriptor areas, or result set sequences.

g) The original and current default time zone displacements, SQL-path value and defining text, current default catalog name, current default unqualified schema name, current default character set name, text defining the default transform group name, and text defining the user-defined type name – transform group pair for each user-defined type explicitly set by the user are set to implementation-defined (IV088) values.

h) The subject table restriction flag is set to *False* and the restricted subject table name list is set to empty.

i) The triggered action indicator is set to indicate that no triggered actions are executing.

j) The statement timestamp is set to "not set".

k) The statement execution context is set to indicate non-atomic, with no state changes.

l) The routine execution context is set to indicate that no SQL-invoked routine is active (rendering the rest of the routine execution context irrelevant).

m) The diagnostic stack is set to "empty".

n) The constraint mode is set to "immediate".

14) A new savepoint level is established.

Conformance Rules

1) Without Feature F771, "Connection management", conforming SQL language shall not contain a <connect statement>.

18.2 <set connection statement>

Function

Select an SQL-connection from the available SQL-connections.

Format

```
<set connection statement> ::=
  SET CONNECTION <connection object>

<connection object> ::=
    DEFAULT
  | <connection name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If a <set connection statement> is executed after the first transaction-initiating SQL-statement executed by the current SQL-transaction and the SQL-implementation does not support Feature T262, "Multiple server transactions", then an exception condition is raised: *feature not supported — multiple server transactions (0A001)*.

2) Case:

   a) If DEFAULT is specified and there is no default SQL-connection that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist (08003)*.

   b) Otherwise, if <connection name> does not identify an SQL-session that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist (08003)*.

3) If the SQL-connection identified by <connection object> cannot be selected, then an exception condition is raised: *connection exception — connection failure (08006)*.

4) The current SQL-connection and current SQL-session become a dormant SQL-connection and a dormant SQL-session, respectively. The SQL-session context is preserved and is not affected in any way by operations performed over the selected SQL-connection.

   NOTE 746 — The SQL-session context is defined in Subclause 4.45, "SQL-sessions".

5) The SQL-connection identified by <connection object> becomes the current SQL-connection and the SQL-session associated with that SQL-connection becomes the current SQL-session. SQL-session context is restored to the same state as at the time the SQL-connection became dormant.

   NOTE 747 — The SQL-session context is defined in Subclause 4.45, "SQL-sessions".

6) The SQL-server for the subsequent execution of <externally-invoked procedure>s in any SQL-client modules associated with the SQL-agent are set to that of the current SQL-connection.

Conformance Rules

1) Without Feature F771, "Connection management", conforming SQL language shall not contain a <set connection statement>.

18.3 <disconnect statement>

Function

Terminate an SQL-connection.

Format

```
<disconnect statement> ::=
  DISCONNECT <disconnect object>

<disconnect object> ::=
    <connection object>
  | ALL
  | CURRENT
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If <connection name> is specified and <connection name> does not identify an SQL-connection that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist (08003)*.

2) If DEFAULT is specified and there is no default SQL-connection that is current or dormant for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist (08003)*.

3) If CURRENT is specified and there is no current SQL-connection for the current SQL-agent, then an exception condition is raised: *connection exception — connection does not exist (08003)*.

4) Let *C* be the current SQL-connection.

5) Let *L* be a list of SQL-connections. If a <connection name> is specified, then *L* is that SQL-connection. If CURRENT is specified, then *L* is the current SQL-connection. If ALL is specified, then *L* is a list representing every SQL-connection that is current or dormant for the current SQL-agent, in an implementation-dependent (US049) order. If DEFAULT is specified, then *L* is the default SQL-connection.

6) If any SQL-connection in *L* is active, then an exception condition is raised: *invalid transaction state — active SQL-transaction (25001)*.

7) For every SQL-connection *C1* in *L*, treating the SQL-session *S1* identified by *C1* as the current SQL-session, all of the actions that are required after the last call of an <externally-invoked procedure> by an SQL-agent, except for the execution of a <rollback statement> or a <commit statement>, are performed. *C1* is terminated, regardless of any exception condition that might occur during the disconnection process.

NOTE 748 — See the General Rules of Subclause 13.1, "<SQL-client module definition>", for the actions to be performed after the last call of an <externally-invoked procedure> by an SQL-agent.

8) If any error is detected during execution of a <disconnect statement>, then a completion condition is raised: *warning — disconnect error (01002)*.

9) If *C* is contained in *L*, then there is no current SQL-connection following the execution of the <disconnect statement>. Otherwise, *C* remains the current SQL-connection.

Conformance Rules

1) Without Feature F771, "Connection management", conforming SQL language shall not contain a <disconnect statement>.

19 Session management

This Clause is modified by Clause 14, "Session management", in ISO/IEC 9075-9.
This Clause is modified by Clause 16, "Session management", in ISO/IEC 9075-14.

19.1 <set session characteristics statement>

Function

Set one or more characteristics for the current SQL-session.

Format

```
<set session characteristics statement> ::=
  SET SESSION CHARACTERISTICS AS <session characteristic list>

<session characteristic list> ::=
  <session characteristic> [ { <comma> <session characteristic> }... ]

<session characteristic> ::=
  <session transaction characteristics>

<session transaction characteristics> ::=
  TRANSACTION <transaction mode> [ { <comma> <transaction mode> }... ]
```

Syntax Rules

1) <session transaction characteristics> shall contain at most one <isolation level>, at most one <transaction access mode>, and at most one <diagnostics size>.

Access Rules

None.

General Rules

1) Let *SCL* be the <session transaction characteristics>. Let *ESC* be the enduring session characteristics of the current SQL-session.

2) If *SCL* contains an <isolation level> *IL*, then the transaction isolation level of *ESC* is set to the <level of isolation> contained in *IL*.

3) If *SCL* contains a <transaction access mode> *AM*, then the transaction access mode of *ESC* is set to read-only or read-write, according to whether *AM* contains READ ONLY or READ WRITE, respectively.

4) If *SCL* contains a <diagnostics size> *DS*, then the condition area limit of *ESC* is set to the <number of conditions> contained in *DS*.

Conformance Rules

1) Without Feature F761, “Session management”, conforming SQL language shall not contain a <set session characteristics statement>.

19.2 <set session user identifier statement>

Function

Set the SQL-session user identifier and the current user identifier of the current SQL-session context.

Format

```
<set session user identifier statement> ::=
  SET SESSION AUTHORIZATION <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) If a <set session user identifier statement> is executed and an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction (25001)*.

2) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

3) If *V* does not conform to the Format and Syntax Rules of an <authorization identifier>, then an exception condition is raised: *invalid authorization specification (28000)*.

4) If *V* is not equal to the current value of the SQL-session user identifier of the current SQL-session context, then the restrictions on the permissible values for *V* are implementation-defined (IA165).

5) If the current user identifier and the current role name are restricted from setting the user identifier to *V*, then an exception condition is raised: *invalid authorization specification (28000)*.

6) The SQL-session user identifier of the current SQL-session context is set to *V*.

7) The current user identifier is set to *V*.

8) The current role name is removed.

   NOTE 749 — The current role name is also the SQL-session role name.

Conformance Rules

1) Without Feature F321, "User authorization", conforming SQL language shall not contain a <set session user identifier statement>.

19.3 <set role statement>

Function

Set the SQL-session role name and the current role name for the current SQL-session context.

Format

```
<set role statement> ::=
  SET ROLE <role specification>

<role specification> ::=
    <value specification>
  | NONE
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) If a <set role statement> is executed and an SQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active SQL-transaction (25001)*.

2) If there is no current user identifier, then an exception condition is raised: *invalid role specification (0P000)*.

3) If <role specification> contains a <value specification>, then:

   a) Let *S* be <value specification> and let *V* be the character string that is the value of

      ```
      TRIM ( BOTH ' ' FROM S )
      ```

   b) If *V* does not conform to the Format and Syntax Rules of a <role name>, then an exception condition is raised: *invalid role specification (0P000)*.

   c) If no role authorization descriptor exists that indicates that the role identified by *V* has been granted to either the current user identifier or to PUBLIC, then an exception condition is raised: *invalid role specification (0P000)*.

   d) The SQL-session role name and the current role name are set to *V*.

4) If NONE is specified, then the current role name is removed.

Conformance Rules

1) Without Feature T331, "Basic roles", conforming SQL language shall not contain a <set role statement>.

19.4 <set local time zone statement>

Function

Set the current default time zone displacement for the current SQL-session.

Format

```
<set local time zone statement> ::=
  SET TIME ZONE <set time zone value>

<set time zone value> ::=
    <interval value expression>
  | LOCAL
```

Syntax Rules

1) The declared type of the <interval value expression> immediately contained in the <set time zone value> shall be INTERVAL HOUR TO MINUTE.

Access Rules

None.

General Rules

1) Case:

   a) If LOCAL is specified, then the current default time zone displacement of the current SQL-session is set to the original time zone displacement of the current SQL-session.

   b) Otherwise,

      Case:

      i) If the value *V* of the <interval value expression> is not the null value and is between the minimum permitted negative time zone displacement and the maximum permitted negative time zone displacement, then the current default time zone displacement of the current SQL-session is set to *V*.

      ii) Otherwise, an exception condition is raised: *data exception — invalid time zone displacement value (22009)*.

Conformance Rules

1) Without Feature F411, "Time zone specification", conforming SQL language shall not contain a <set local time zone statement>.

19.5 <set catalog statement>

Function

Set the default catalog name for unqualified <schema name>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly.

Format

```
<set catalog statement> ::=
  SET <catalog name characteristic>

<catalog name characteristic> ::=
  CATALOG <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

2) If *V* does not conform to the Format and Syntax Rules of a <catalog name>, then an exception condition is raised: *invalid catalog name (3D000)*.

3) The default catalog name of the current SQL-session is set to *V*.

Conformance Rules

1) Without Feature F651, "Catalog name qualifiers", conforming SQL language shall not contain a <set catalog statement>.

2) Without Feature F761, "Session management", conforming SQL language shall not contain a <set catalog statement>.

19.6 <set schema statement>

Function

Set the default schema name for unqualified <schema qualified name>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly.

Format

```
<set schema statement> ::=
  SET <schema name characteristic>

<schema name characteristic> ::=
  SCHEMA <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

2) If *V* does not conform to the Format and Syntax Rules of a <schema name>, then an exception condition is raised: *invalid schema name (3F000)*.

3) Case:

   a) If *V* conforms to the Format and Syntax Rules for a <schema name> that contains a <catalog name>, then let *X* be the <catalog name> part and let *Y* be the <unqualified schema name> part of *V*. The following statement is implicitly executed:

      ```
      SET CATALOG 'X'
      ```

      and the <set schema statement> is effectively replaced by:

      ```
      SET SCHEMA 'Y'
      ```

   b) Otherwise, the default unqualified schema name of the current SQL-session is set to *V*.

Conformance Rules

1) Without Feature F761, "Session management", conforming SQL language shall not contain a <set schema statement>.

19.7 <set names statement>

Function

Set the default character set name for <character string literal>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly.

Format

```
<set names statement> ::=
  SET <character set name characteristic>

<character set name characteristic> ::=
  NAMES <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

2) If *V* does not conform to the Format and Syntax Rules of a <character set name>, then an exception condition is raised: *invalid character set name (2C000)*.

3) The default character set name of the current SQL-session is set to *V*.

Conformance Rules

1) Without Feature F461, "Named character sets", conforming SQL language shall not contain a <set names statement>.

2) Without Feature F761, "Session management", conforming SQL language shall not contain a <set names statement>.

19.8 <set path statement>

Function

Set the SQL-path used to determine the subject routine of <routine invocation>s with unqualified <routine name>s in <preparable statement>s that are prepared in the current SQL-session by an <execute immediate statement> or a <prepare statement> and in <direct SQL statement>s that are invoked directly. The SQL-path remains the current SQL-path of the SQL-session until another SQL-path is successfully set.

Format

```
<set path statement> ::=
  SET <SQL-path characteristic>

<SQL-path characteristic> ::=
  PATH <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   a) If *V* does not conform to the Format and Syntax Rules of a <schema name list>, then an exception condition is raised: *invalid schema name list specification (0E000)*.

   b) The SQL-path of the current SQL-session is set to *V*.

   NOTE 750 — A <set path statement> that is executed between a <prepare statement> and an <execute statement> has no effect on the prepared statement.

Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <set path statement>.

19.9 <set transform group statement>

Function

Set the group name that identifies the group of transform functions for mapping values of user-defined types to predefined data types.

Format

```
<set transform group statement> ::=
  SET <transform group characteristic>

<transform group characteristic> ::=
    DEFAULT TRANSFORM GROUP <value specification>
  | TRANSFORM GROUP FOR TYPE <path-resolved user-defined type name> <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

2) If <path-resolved user-defined type name> is specified, then let *UDT* be the user-defined type identified by that <path-resolved user-defined type name>.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   a) If *V* does not conform to the Format and Syntax Rules of a <group name>, then an exception condition is raised: *invalid transform group name specification (0S000)*.

   b) Case:

      i) If <path-resolved user-defined type name> is specified, then the transform group name corresponding to all subtypes of *UDT* for the current SQL-session is set to *V*.

      ii) Otherwise, the default transform group name for the current SQL-session is set to *V*.

   NOTE 751 — A <set transform group statement> that is executed after a <prepare statement> has no effect on the prepared statement.

Conformance Rules

1) Without Feature S241, "Transform functions", conforming SQL language shall not contain a <set transform group statement>.

19.10 <set session collation statement>

Function

Set the SQL-session collation of the SQL-session for one or more character sets. An SQL-session collation remains effective until another SQL-session collation for the same character set is successfully set.

Format

```
<set session collation statement> ::=
    SET COLLATION <collation specification> [ FOR <character set specification list> ]
  | SET NO COLLATION [ FOR <character set specification list> ]

<collation specification> ::=
  <value specification>
```

Syntax Rules

1) The declared type of the <value specification> shall be a character string type.

Access Rules

None.

General Rules

1) Let *S* be <value specification> and let *V* be the character string that is the value of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   a) If *V* does not conform to the Format and Syntax Rules of a <collation name>, then an exception condition is raised: *invalid collation name (2H000)*.

   b) Let *CO* be the collation identified by the <collation name> contained in *V*.

      Case:

      i) If <character set specification list> is specified, then

         Case:

         1) If the collation specified by *CO* is not applicable to any character set identified by a <character set specification>, then an exception condition is raised: *invalid collation name (2H000)*.

         2) Otherwise, for each character set specified, the SQL-session collation for that character set in the current SQL-session is set to *CO*.

      ii) Otherwise, the character sets for which the SQL-session collations are set to *CO* are implementation-defined (ID236).

2) If SET NO COLLATION is specified, then

   Case:

a) If <character set specification list> is specified, then, for each character set specified, the SQL-session collation for that character set in the current SQL-session is set to *none*.

b) Otherwise, the SQL-session collation for every character set in the current SQL-session is set to *none*.

Conformance Rules

1) Without Feature F693, "SQL-session and client module collations", conforming SQL language shall not contain a <set session collation statement>.

20 Dynamic SQL

This Clause is modified by Clause 9, "Dynamic SQL", in ISO/IEC 9075-3.
This Clause is modified by Clause 16, "Dynamic SQL", in ISO/IEC 9075-4.
This Clause is modified by Clause 15, "Dynamic SQL", in ISO/IEC 9075-9.
This Clause is modified by Clause 17, "Dynamic SQL", in ISO/IEC 9075-14.
This Clause is modified by Clause 15, "Dynamic SQL", in ISO/IEC 9075-15.

20.1 Description of SQL descriptor areas

This Subclause is modified by Subclause 15.1, "Description of SQL descriptor areas", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.1, "Description of SQL descriptor areas", in ISO/IEC 9075-14.
This Subclause is modified by Subclause 15.1, "Description of SQL descriptor areas", in ISO/IEC 9075-15.

Function

Specify the identifiers, data types, and codes used in SQL item descriptor areas.

Syntax Rules

1) An SQL item descriptor area comprises the components specified in Table 29, "Data types of <key word>s used in SQL item descriptor areas".

2) An SQL descriptor area comprises the components specified in Table 28, "Data types of <key word>s used in the header of SQL descriptor areas", and zero or more occurrences of an SQL item descriptor area.

3) Given an SQL item descriptor area *IDA* in which the value of LEVEL is *N*, the *immediately subordinate descriptor areas* of *IDA* are those SQL item descriptor areas in which the value of LEVEL is *N*+1 and whose position in the SQL descriptor area follows that of *IDA* and precedes that of any SQL item descriptor area in which the value of LEVEL is less than *N*+1.

   NOTE 752 — The value of LEVEL for the first SQL item descriptor area is always 0 (zero).

   The *subordinate descriptor areas* of *IDA* are those SQL item descriptor areas that are immediately subordinate descriptor areas of *IDA* or that are subordinate descriptor areas of an SQL item descriptor area that is immediately subordinate to *IDA*.

4) Given a data type *DT* and its descriptor *DE*, the *immediately subordinate descriptors* of *DE* are defined to be

   Case:

   a) If *DT* is a row type, then the field descriptors of the fields of *DT*. The *i*-th immediately subordinate descriptor is the descriptor of the *i*-th field of *DT*.

   b) If *DT* is a collection type, then the descriptor of the associated element type of *DT*.

   The *subordinate descriptors* of *DE* are those descriptors that are immediately subordinate descriptors of *DE* or that are subordinate descriptors of a descriptor that is immediately subordinate to *DE*.

5) Given a descriptor *DE*, let SDE_j represent its *j*-th immediately subordinate descriptor. There is an implied ordering of the subordinate descriptors of *DE*, such that:

a) SDE_1 is in the first ordinal position.

b) The ordinal position of SDE_{j+1} is $K+NS+1$, where K is the ordinal position of SDE_j and NS is the number of subordinate descriptors of SDE_j. The implicitly ordered subordinate descriptors of SDE_j occupy contiguous ordinal positions starting at position $K+1$.

6) An item descriptor area *IDA* is *valid* if and only if TYPE indicates a code defined in Table 30, "Codes used for SQL data types in Dynamic SQL", and exactly one of the following is true.

a) TYPE indicates CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, LENGTH is a valid length value for TYPE, and CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the fully qualified name of a character set that is valid for TYPE.

b) TYPE indicates CHARACTER LARGE OBJECT LOCATOR.

c) TYPE indicates BINARY, BINARY VARYING, or BINARY LARGE OBJECT and LENGTH is a valid length value for the TYPE.

d) TYPE indicates BINARY LARGE OBJECT LOCATOR.

e) TYPE indicates NUMERIC and PRECISION and SCALE are valid precision and scale values for the NUMERIC data type.

f) TYPE indicates DECIMAL and PRECISION and SCALE are valid precision and scale values for the DECIMAL data type.

g) TYPE indicates SMALLINT, INTEGER, BIGINT, REAL, or DOUBLE PRECISION.

h) TYPE indicates DECFLOAT and PRECISION is a valid precision value for the DECFLOAT data type.

i) TYPE indicates FLOAT and PRECISION is a valid precision value for the FLOAT data type.

j) TYPE indicates BOOLEAN.

k) TYPE indicates a <datetime type>, DATETIME_INTERVAL_CODE is a code specified in Table 31, "Codes associated with datetime data types in Dynamic SQL", and PRECISION is a valid value for the <time precision> or <timestamp precision> of the indicated datetime data type.

l) TYPE indicates an <interval type>, DATETIME_INTERVAL_CODE is a code specified in Table 32, "Codes used for <interval qualifier>s in Dynamic SQL", and DATETIME_INTERVAL_PRECISION and PRECISION are valid values for <interval leading field precision> and <interval fractional seconds precision> for an <interval qualifier>.

m) TYPE indicates JSON.

n) TYPE indicates USER-DEFINED TYPE LOCATOR and USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are the fully qualified name of a valid user-defined type.

o) TYPE indicates REF, LENGTH is the length in octets for the REF type, and USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are a valid fully qualified user-defined type name, and SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME are a valid fully qualified table name.

p) TYPE indicates ROW, the value N of DEGREE is a valid value for the degree of a row type, there are exactly N immediately subordinate descriptor areas of *IDA* and those SQL item descriptor areas are valid.

q) 15 TYPE indicates ARRAY or ARRAY LOCATOR, the value of CARDINALITY is a valid value for the cardinality of an array, there is exactly one immediately subordinate descriptor area of *IDA*, and that SQL item descriptor area is valid.

r) TYPE indicates MULTISET or MULTISET LOCATOR, there is exactly one immediately subordinate descriptor area of *IDA*, and that SQL item descriptor area is valid.

s) 09 TYPE indicates an implementation-defined (IV220) data type.

7) The declared type *T* of a <simple value specification> or a <simple target specification> *SVT* is said to *match* the data type specified by a valid item descriptor area *IDA* if and only if exactly one of the following is true.

a) TYPE indicates CHARACTER and *T* is specified by CHARACTER(*L*), where *L* is the value of LENGTH and the <character set specification> formed by the values of CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME identifies the character set of *SVT*.

b) Either TYPE indicates CHARACTER VARYING and *T* is specified by CHARACTER VARYING(*L*) or TYPE indicates CHARACTER LARGE OBJECT and *T* is specified by CHARACTER LARGE OBJECT(*L*), where the <character set specification> formed by the values of CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME identifies the character set of *SVT* and

Case:

i) *SVT* is a <simple value specification> and *L* is the value of LENGTH.

ii) *SVT* is a <simple target specification> and *L* is not less than the value of LENGTH.

c) TYPE indicates CHARACTER LARGE OBJECT LOCATOR and *T* is specified by CHARACTER LARGE OBJECT LOCATOR.

d) TYPE indicates BINARY and *T* is specified by BINARY(*L*), where *L* is the value of LENGTH.

e) Either TYPE indicates BINARY VARYING and *T* is specified by BINARY VARYING(*L*) or TYPE indicates BINARY LARGE OBJECT and *T* is specified by BINARY LARGE OBJECT(*L*), and

Case:

i) *STV* is a <simple value specification> and *L* is the value of LENGTH.

ii) *STV* is a <simple target specification> and *L* is not less than the value of LENGTH.

f) TYPE indicates BINARY LARGE OBJECT LOCATOR and *T* is specified by BINARY LARGE OBJECT LOCATOR.

g) TYPE indicates NUMERIC and *T* is specified by NUMERIC(*P*,*S*), where *P* is the value of PRECISION and *S* is the value of SCALE.

h) TYPE indicates DECIMAL and *T* is specified by DECIMAL(*P*,*S*), where *P* is the value of PRECISION and *S* is the value of SCALE.

i) TYPE indicates SMALLINT and *T* is specified by SMALLINT.

j) TYPE indicates INTEGER and *T* is specified by INTEGER.

k) TYPE indicates BIGINT and *T* is specified by BIGINT.

l) TYPE indicates DECFLOAT and *T* is specified by DECFLOAT(*P*), where *P* is the value of PRECISION.

m) TYPE indicates FLOAT and *T* is specified by FLOAT(*P*), where *P* is the value of PRECISION.

n) TYPE indicates REAL and *T* is specified by REAL.

o) TYPE indicates DOUBLE PRECISION and *T* is specified by DOUBLE PRECISION.

p) TYPE indicates BOOLEAN and *T* is specified by BOOLEAN.

q) TYPE indicates JSON and *T* is specified by JSON.

r) TYPE indicates USER-DEFINED TYPE LOCATOR and *T* is specified by USER-DEFINED TYPE LOCATOR, where the values of USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are the fully qualified name of the associated user-defined type of *SVT*.

s) TYPE indicates REF and *T* is specified by REF, where the <user-defined type name> formed by the values of USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME identifies the referenced type of *SVT*, and SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME identify the scope of the reference type.

t) TYPE indicates ROW, and *T* is a row type with degree *D*, where *D* is the value of DEGREE, and the data type of the *i*-th field of *SVT* matches the data type specified by the *i*-th immediately subordinate descriptor area of *IDA*.

u) TYPE indicates ARRAY and *T* is an array type with maximum cardinality *C* and the data type of the element type of *T* matches the data type specified by the immediately subordinate descriptor area of *IDA*, and

Case:

i) *SVT* is a <simple value specification> and *C* is the value of CARDINALITY.

ii) *SVT* is a <simple target specification> and *C* is not less than the value of CARDINALITY.

v) 15 TYPE indicates ARRAY LOCATOR and *T* is an array locator type whose associated array type has maximum cardinality *C* and the data type of the element type of the associated array type of *T* matches the data type specified by the immediately subordinate descriptor area of *IDA*, and

Case:

i) *SVT* is a <simple value specification> and *C* is the value of CARDINALITY.

ii) *SVT* is a <simple target specification> and *C* is not less than the value of CARDINALITY.

w) TYPE indicates MULTISET and *T* is a multiset type and the data type of the element type of *T* matches the data type specified by the immediately subordinate descriptor area of *IDA*.

x) TYPE indicates MULTISET LOCATOR and *T* is a multiset locator type and the data type of the element type of *T* matches the data type specified by the immediately subordinate descriptor area of *IDA*.

y) TYPE indicates a data type from Table 30, "Codes used for SQL data types in Dynamic SQL", other than an implementation-defined (IE002) data type and *T* satisfies the implementation-defined (IA080) rules for matching that data type.

z) 09 TYPE indicates an implementation-defined (IE002) data type and *T* satisfies the implementation-defined (IA080) rules for matching that data type.

8) A data type *DT* is said to be *represented* by an SQL item descriptor area if a <simple value specification> of type *DT* matches the SQL item descriptor area.

Table 28 — Data types of <key word>s used in the header of SQL descriptor areas

<key word>	Data Type
COUNT	exact numeric with scale 0 (zero)
DYNAMIC_FUNCTION	character string with character set SQL_IDENTIFIER and length not less than 128 characters
DYNAMIC_FUNCTION_CODE	exact numeric with scale 0 (zero)
KEY_TYPE	exact numeric with scale 0 (zero)
TOP_LEVEL_COUNT	exact numeric with scale 0 (zero)

Table 29 — Data types of <key word>s used in SQL item descriptor areas

<key word>	Data Type
CARDINALITY	exact numeric with scale 0 (zero)
CHARACTER_SET_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
CHARACTER_SET_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
CHARACTER_SET_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
COLLATION_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
COLLATION_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
COLLATION_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
DATA	matches the data type represented by the SQL item descriptor area
DATETIME_INTERVAL_CODE	exact numeric with scale 0 (zero)
DATETIME_INTERVAL_PRECISION	exact numeric with scale 0 (zero)
DEGREE	exact numeric with scale 0 (zero)
INDICATOR	exact numeric with scale 0 (zero)
KEY_MEMBER	exact numeric with scale 0 (zero)
LENGTH	exact numeric with scale 0 (zero)

<key word>	Data Type
LEVEL	exact numeric with scale 0 (zero)
NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
NULLABLE	exact numeric with scale 0 (zero)
NULL_ORDERING	exact numeric with scale 0 (zero)
OCTET_LENGTH	exact numeric with scale 0 (zero)
PARAMETER_MODE	exact numeric with scale 0 (zero)
PARAMETER_ORDINAL_POSITION	exact numeric with scale 0 (zero)
PARAMETER_SPECIFIC_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
PARAMETER_SPECIFIC_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
PARAMETER_SPECIFIC_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
PRECISION	exact numeric with scale 0 (zero)
RETURNED_CARDINALITY	exact numeric with scale 0 (zero)
RETURNED_LENGTH	exact numeric with scale 0 (zero)
RETURNED_OCTET_LENGTH	exact numeric with scale 0 (zero)
SCALE	exact numeric with scale 0 (zero)
SCOPE_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
SCOPE_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters
SCOPE_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
SORT_DIRECTION	exact numeric with scale 0 (zero)
TYPE	exact numeric with scale 0 (zero)
UNNAMED	exact numeric with scale 0 (zero)
USER_DEFINED_TYPE_CATALOG	character string with character set SQL_IDENTIFIER and length not less than 128 characters
USER_DEFINED_TYPE_NAME	character string with character set SQL_IDENTIFIER and length not less than 128 characters

<key word>	Data Type
USER_DEFINED_TYPE_SCHEMA	character string with character set SQL_IDENTIFIER and length not less than 128 characters
USER_DEFINED_TYPE_CODE	exact numeric with scale 0 (zero)

NOTE 753 — "Matches" and "represented by", as applied to the relationship between a data type and an SQL item descriptor area are defined in the Syntax Rules of this Subclause.

Access Rules

None.

General Rules

1) Table 30, "Codes used for SQL data types in Dynamic SQL", specifies the codes associated with the SQL data types.

Table 30 — Codes used for SQL data types in Dynamic SQL

Data Type	Code
implementation-defined (IV220) data types	< 0 (zero)
ARRAY	50
ARRAY LOCATOR	51
BIGINT	25
BINARY	60
BINARY VARYING	61
BINARY LARGE OBJECT	30
BINARY LARGE OBJECT LOCATOR	31
BOOLEAN	16
CHARACTER	1 (one)
CHARACTER VARYIN	12
CHARACTER LARGE OBJECT	40
CHARACTER LARGE OBJECT LOCATOR	41
DATE, TIME WITHOUT TIME ZONE, TIME WITH TIME ZONE, TIMESTAMP WITHOUT TIME ZONE, or TIMESTAMP WITH TIME ZONE	9

Data Type	Code
DECFLOAT	26
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
INTERVAL	10
JSON	65
MULTISET	55
MULTISET LOCATOR	56
NUMERIC	2
REAL	7
SMALLINT	5
ROW TYPE	19
REF	20
User-defined types	17
USER-DEFINED TYPE LOCATOR	18

2) Table 31, "Codes associated with datetime data types in Dynamic SQL", specifies the codes associated with the datetime data types.

Table 31 — Codes associated with datetime data types in Dynamic SQL

Datetime Data Type	Code
DATE	1 (one)
TIME WITH TIME ZONE	4
TIME WITHOUT TIME ZONE	2
TIMESTAMP WITH TIME ZONE	5
TIMESTAMP WITHOUT TIME ZONE	3

3) Table 32, "Codes used for <interval qualifier>s in Dynamic SQL", specifies the codes associated with <interval qualifier>s for interval data types.

Table 32 — Codes used for <interval qualifier>s in Dynamic SQL

Interval Qualifier	Code
DAY	3
DAY TO HOUR	8
DAY TO MINUTE	9
DAY TO SECOND	10
HOUR	4
HOUR TO MINUTE	11
HOUR TO SECOND	12
MINUTE	5
MINUTE TO SECOND	13
MONTH	2
SECOND	6
YEAR	1 (one)
YEAR TO MONTH	7

4) The value of DYNAMIC_FUNCTION is a character string that identifies the type of the prepared or executed SQL-statement. Table 39, "SQL-statement codes", specifies the identifier of the SQL-statements.

5) The value of DYNAMIC_FUNCTION_CODE is a number that identifies the type of the prepared or executed SQL-statement. Table 39, "SQL-statement codes", specifies the code of the SQL-statements.

6) Table 33, "Codes used for input/output SQL parameter modes in Dynamic SQL", specifies the codes used for the PARAMETER_MODE item descriptor component when describing a <call statement>.

Table 33 — Codes used for input/output SQL parameter modes in Dynamic SQL

Parameter mode	Code
PARAMETER_MODE_IN	1 (one)
PARAMETER_MODE_INOUT	2
PARAMETER_MODE_OUT	4

7) Table 34, "Codes associated with user-defined types in Dynamic SQL", specifies the codes associated with user-defined types.

Table 34 — Codes associated with user-defined types in Dynamic SQL

User-Defined Type	Code
DISTINCT	1 (one)
STRUCTURED	2

8) Table 35, "Codes associated with sort direction", specifies the codes used to describe sort directions in the SQL item descriptor areas of a PTF descriptor area of the ordering of a <table argument>.

Table 35 — Codes associated with sort direction

Sort direction	Code
SORT_ASC	1 (one)
SORT_DESC	−1 (negative one)

9) Table 36, "Codes associated with null ordering", specifies the codes used to describe whether nulls are sorted first or last in the SQL item descriptor areas of a PTF descriptor area of the ordering of a <table argument>.

Table 36 — Codes associated with null ordering

Null ordering	Code
NULLS_FIRST	1 (one)
NULLS_LAST	−1 (negative one)

Conformance Rules

None.

20.2 <allocate descriptor statement>

Function

Allocate an SQL descriptor area.

Format

```
<allocate descriptor statement> ::=
  ALLOCATE [ SQL ] DESCRIPTOR <conventional descriptor name> [ WITH MAX <occurrences> ]

<occurrences> ::=
  <simple value specification>
```

Syntax Rules

1) The declared type of <occurrences> shall be exact numeric with scale 0 (zero).

2) If WITH MAX <occurrences> is not specified, then an implementation-defined (ID237) default value for <occurrences> that is greater than 0 (zero) is implicit.

Access Rules

None.

General Rules

1) Case:

   a) If a <conventional descriptor name> is an <extended descriptor name>, then let *S* be the <simple value specification> that is immediately contained in <extended descriptor name> and let *V* be the character string that is the result of

      ```
      TRIM ( BOTH ' ' FROM S )
      ```

      Case:

      i) If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

      ii) Otherwise, let *DN* be the <extended descriptor name>. The value of *DN* is *V*.

   b) Otherwise, let *DN* be the <non-extended descriptor name>.

2) The maximum number of SQL descriptor areas that can be allocated at one time is implementation-defined (IL206).

3) If <occurrences> is less than 1 (one) or is greater than an implementation-defined (IL207) maximum value, then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

4) Case:

   a) If *DN* identifies an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

b) Otherwise, an SQL descriptor area is created that is identified by *DN*. The SQL descriptor area will have at least <occurrences> number of SQL item descriptor areas. The value of LEVEL in each of the item descriptor areas is set to 0 (zero). The value of every other component in the SQL descriptor area is implementation-dependent (UV110).

Conformance Rules

1) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain an <occurrences> that is not a <literal>.

2) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <allocate descriptor statement>.

20.3 <deallocate descriptor statement>

Function

Deallocate an SQL descriptor area.

Format

```
<deallocate descriptor statement> ::=
  DEALLOCATE [ SQL ] DESCRIPTOR <conventional descriptor name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Case:

   a) If <conventional descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

   b) Otherwise, the SQL descriptor area identified by <conventional descriptor name> is destroyed.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <deallocate descriptor statement>.

20.4 <get descriptor statement>

This Subclause is modified by Subclause 15.2, "<get descriptor statement>", in ISO/IEC 9075-15.

Function

Get information from an SQL descriptor area.

Format

```
<get descriptor statement> ::=
  GET [ SQL ] DESCRIPTOR <descriptor name> <get descriptor information>

<get descriptor information> ::=
    <get header information> [ { <comma> <get header information> }... ]
  | VALUE <item number> <get item information>
        [ { <comma> <get item information> }... ]

<get header information> ::=
  <get header information target specification> <equals operator> <header item name>

<header item name> ::=
    COUNT
  | KEY_TYPE
  | DYNAMIC_FUNCTION
  | DYNAMIC_FUNCTION_CODE
  | TOP_LEVEL_COUNT

<get item information> ::=
  <get item information target specification> <equals operator> <descriptor item name>

<item number> ::=
  <simple value specification>

<get header information target specification> ::=
  <simple target specification>

<get item information target specification> ::=
  <simple target specification>

<descriptor item name> ::=
    CARDINALITY
  | CHARACTER_SET_CATALOG
  | CHARACTER_SET_NAME
  | CHARACTER_SET_SCHEMA
  | COLLATION_CATALOG
  | COLLATION_NAME
  | COLLATION_SCHEMA
  | DATA
  | DATETIME_INTERVAL_CODE
  | DATETIME_INTERVAL_PRECISION
  | DEGREE
  | INDICATOR
  | KEY_MEMBER
  | LENGTH
  | LEVEL
  | NAME
  | NULLABLE
  | NULL_ORDERING
  | OCTET_LENGTH
```

```
| PARAMETER_MODE
| PARAMETER_ORDINAL_POSITION
| PARAMETER_SPECIFIC_CATALOG
| PARAMETER_SPECIFIC_NAME
| PARAMETER_SPECIFIC_SCHEMA
| PRECISION
| RETURNED_CARDINALITY
| RETURNED_LENGTH
| RETURNED_OCTET_LENGTH
| SCALE
| SCOPE_CATALOG
| SCOPE_NAME
| SCOPE_SCHEMA
| SORT_DIRECTION
| TYPE
| UNNAMED
| USER_DEFINED_TYPE_CATALOG
| USER_DEFINED_TYPE_NAME
| USER_DEFINED_TYPE_SCHEMA
| USER_DEFINED_TYPE_CODE
```

Syntax Rules

1) The declared type of <item number> shall be exact numeric with scale 0 (zero).

2) For each <get header information>, the declared type of <get header information target specification> shall be that shown in the Data Type column of the row in Table 28, "Data types of <key word>s used in the header of SQL descriptor areas", whose <key word> column value is equivalent to <header item name>.

3) For each <get item information>, the declared type of <get item information target specification> shall be that shown in the Data Type column of the row in Table 29, "Data types of <key word>s used in SQL item descriptor areas", whose <key word> column value is equivalent to <descriptor item name>.

Access Rules

None.

General Rules

1) If <descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

2) If the <item number> specified in a <get descriptor statement> is greater than the value of <occurrences> specified when the SQL descriptor area identified by the <descriptor name> was allocated or less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

3) If the <item number> specified in a <get descriptor statement> is greater than the value of COUNT, then a completion condition is raised: *no data (02000)*.

4) If the declared type of the <simple target specification> associated with the keyword DATA does not match the data type represented by the item descriptor area, then an exception condition is raised: *data exception — error in assignment (22005)*.

NOTE 754 — "Match" and "represented by" are defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

5) Let *i* be the value of the <item number> contained in <get descriptor information>. Let *IDA* be the *i*-th item descriptor area. If a <get item information> specifies DATA, then:

   a) If *IDA* is subordinate to an item descriptor area whose TYPE field indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — undefined DATA value (0700C)*.

   b) If TYPE in *IDA* indicates ROW, then an exception condition is raised: *dynamic SQL error — undefined DATA value (0700C)*.

   c) If the value of INDICATOR is negative and no <get item information> specifies INDICATOR, then an exception condition is raised: *data exception — null value, no indicator parameter (22002)*.

6) If an exception condition is raised in a <get descriptor statement>, then the values of all targets specified by <get header information target specification> and <get item information target specification> are implementation-dependent (UV111).

7) A <get descriptor statement> retrieves values from the SQL descriptor area specified by <descriptor name>. The values retrieved are specified by the <get descriptor information>. If <get descriptor information> specifies one or more <get header information>s, then the values retrieved are those identified by the <header item name>s contained in those <get header information>s. If <get descriptor information> specifies one or more <get item information>s, then the values retrieved are those identified by the <descriptor item name>s contained in those <get item information>s in the item descriptor area identified by <item number>. For each item, the value that is retrieved is the one established by the most recently executed <allocate descriptor statement>, <set descriptor statement>, <describe statement>, <copy descriptor statement>, General Rule of Subclause 9.24, "Compilation of an invocation of a polymorphic table function", or General Rule of Subclause 9.25, "Execution of an invocation of a polymorphic table function", that references the specified SQL descriptor area. The value retrieved by a <get descriptor statement> for any component whose value is undefined is implementation-dependent (UV112).

   Case:

   a) If <get descriptor information> contains one or more <get header information>s, then for each <get header information> specified, the General Rules of Subclause 9.2, "Store assignment", are applied with <get header information target specification> as *TARGET* and the value *V* in the SQL descriptor area of the field identified by the <header item name> as *VALUE*.

   b) If <get descriptor information> contains one or more <get item information>s, then:

      i) Let *i* be the value of the <item number> contained in the <get descriptor information>.

      ii) For each <get item information> specified, the General Rules of Subclause 9.2, "Store assignment", are applied with <get item information target specification> as *TARGET* and the value *V* in the *i*-th SQL item descriptor area of the component identified by the <descriptor item name> as *VALUE*.

Conformance Rules

1) Without at least one of Feature B031, "Basic dynamic SQL" or Feature B209, "PTF extended names", conforming SQL language shall not contain a <get descriptor statement>.

2) Without Feature T301, "Functional dependencies", conforming SQL language shall not contain a <descriptor item name> that contains KEY_MEMBER.

3) Without Feature B209, "PTF extended names", <descriptor name> shall not be a <PTF descriptor name>.

4) Without Feature B209, “PTF extended names”, conforming SQL language shall not contain a <descriptor item name> that contains NULL_ORDERING or SORT_DIRECTION.

20.5 <set descriptor statement>

Function

Set information in an SQL descriptor area.

Format

```
<set descriptor statement> ::=
  SET [ SQL ] DESCRIPTOR <descriptor name> <set descriptor information>

<set descriptor information> ::=
    <set header information> [ { <comma> <set header information> }... ]
  | VALUE <item number> <set item information>
        [ { <comma> <set item information> }... ]

<set header information> ::=
  <header item name> <equals operator> <set header information value specification>

<set item information> ::=
  <descriptor item name> <equals operator> <set item information value specification>

<set header information value specification> ::=
  <simple value specification>

<set item information value specification> ::=
  <simple value specification>
```

Syntax Rules

1) Let *DN* be the <descriptor name>.

2) For each <set header information>, <header item name> shall not be KEY_TYPE, TOP_LEVEL_COUNT, DYNAMIC_FUNCTION, or DYNAMIC_FUNCTION_CODE, and the declared type of <set header information value specification> shall be that shown in the Data Type column of the row of Table 28, "Data types of <key word>s used in the header of SQL descriptor areas", whose <key word> column value is equivalent to <header item name>.

3) For each <set item information>:

   a) <descriptor item name> shall not be RETURNED_LENGTH, RETURNED_OCTET_LENGTH, RETURNED_CARDINALITY, OCTET_LENGTH, NULLABLE, KEY_MEMBER, UNNAMED, PARAMETER_MODE, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, PARAMETER_SPECIFIC_NAME, or USER_DEFINED_TYPE_CODE.

   b) If *DN* is a <conventional descriptor name>, then <descriptor item name> shall not be COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, or NAME.

   c) No <descriptor item name> shall be specified more than once in a <set descriptor statement>.

   d) The declared type of <set item information value specification> shall be that shown in the Data Type column of the row in Table 29, "Data types of <key word>s used in SQL item descriptor areas", whose <key word> column value is equivalent to <descriptor item name>.

4) If *DN* is a <conventional descriptor name> and the <descriptor item name> specifies DATA, then <set item information value specification> shall not be a <literal>.

Access Rules

None.

General Rules

1) If *DN* does not identify an SQL descriptor area *SDA*, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

2) If the <item number> specified in a <set descriptor statement> is greater than the value of <occurrences> specified when *SDA* was allocated or less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

3) When more than one value is set in a single <set descriptor statement>, the values are effectively assigned in the following order: LEVEL, NAME, TYPE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, PRECISION, SCALE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, SCOPE_NAME, LENGTH, INDICATOR, DEGREE, CARDINALITY, and DATA.

   When any value other than DATA is set, the value of DATA becomes undefined.

4) For every <set item information> specified, let *DIN* be the <descriptor item name>, let *V* be the value of the <set item information value specification>, let *N* be the value of <item number>, and let *IDA* be the *N*-th item descriptor area of *SDA*.

   Case:

   a) If *DIN* is DATA, then:

      i) If *IDA* is subordinate to an item descriptor area whose TYPE component indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — invalid DATA target (0700D)*.

      ii) If TYPE in *IDA* indicates ROW, then an exception condition is raised: *dynamic SQL error — invalid DATA target (0700D)*.

      iii) If the most specific type of *V* does not match the data type specified by the item descriptor area, then an exception condition is raised: *data exception — error in assignment (22005)*.

         NOTE 755 — "Match" is defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

      iv) The value of DATA in *IDA* is set to *V*.

   b) If *DIN* is LEVEL, then:

      i) If *N* is 1 (one) and *V* is not 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid LEVEL value (0700E)*.

      ii) If *N* is greater than 1 (one), then let *PIDA* be *IDA*'s immediately preceding item descriptor area and let *K* be its LEVEL value.

         1) If *V* = *K*+1 and TYPE in *PIDA* does not indicate ROW, ARRAY, ARRAY LOCATOR, MULTISET, MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value (0700E)*.

2) If $V > K+1$, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value (0700E)*.

3) If $V < K+1$, then let $OIDA_i$ be the i-th item descriptor area to which *PIDA* is subordinate and whose TYPE component indicates ROW, let NS_i be the number of immediately subordinate descriptor areas of $OIDA_i$ between $OIDA_i$ and *IDA* and let D_i be the value of DEGREE in $OIDA_i$.

A) For each $OIDA_i$ whose LEVEL value is greater than V, if D_i is not equal to NS_i, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value (0700E)*.

B) If K is not 0 (zero), then let $OIDA_j$ be the $OIDA_i$ whose LEVEL value is K. If there exists no such $OIDA_j$ or D_j is not greater than NS_j, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value (0700E)*.

iii) The value of LEVEL in *IDA* is set to *V*.

c) If *DIN* is TYPE, then:

i) The value of TYPE in *IDA* is set to *V*.

ii) The value of all components other than TYPE and LEVEL in *IDA* are set to implementation-dependent (UV113) values.

iii) Case:

1) If *V* indicates CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME in *IDA* are set to the values for the default character set name for the SQL-session and LENGTH in *IDA* is set to 1 (one). If *SDA* is a PTF descriptor area, then COLLATION_CATALOG, COLLATION_SCHEMA, and COLLATION_NAME are set to the default collation name for the default character set of the SQL-session.

2) If *V* indicates CHARACTER LARGE OBJECT LOCATOR, then LENGTH in *IDA* is set to 1 (one).

3) If *V* indicates BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then LENGTH in *IDA* is set to 1 (one).

4) If *V* indicates BINARY LARGE OBJECT LOCATOR, then LENGTH in *IDA* is set to 1 (one).

5) If *V* indicates DATETIME, then PRECISION in *IDA* is set to 0 (zero).

6) If *V* indicates INTERVAL, then DATETIME_INTERVAL_PRECISION in *IDA* is set to 2.

7) If *V* indicates NUMERIC or DECIMAL, then SCALE in *IDA* is set to 0 (zero) and PRECISION in *IDA* is set to the implementation-defined (IA235) default value for the precision of NUMERIC or DECIMAL data types, respectively.

8) If *V* indicates DECFLOAT, then PRECISION in *IDA* is set to the implementation-defined (IA235) default value for the precision of the DECFLOAT data type.

9) If *V* indicates FLOAT, then PRECISION in *IDA* is set to the implementation-defined (IA235) default value for the precision of the FLOAT data type.

d) If *DIN* is DATETIME_INTERVAL_CODE, then

Case:

i) If TYPE in *IDA* indicates DATETIME, then

Case:

1) If *V* indicates DATE, TIME, or TIME WITH TIME ZONE, then PRECISION in *IDA* is set to 0 (zero) and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

2) If *V* indicates TIMESTAMP or TIMESTAMP WITH TIME ZONE, then PRECISION in *IDA* is set to 6 and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

3) Otherwise, an exception condition is raised: *dynamic SQL error — invalid DATETIME_INTERVAL_CODE (0700F)*.

ii) If TYPE in *IDA* indicates INTERVAL, then

Case:

1) If *V* indicates DAY TO SECOND, HOUR TO SECOND, MINUTE TO SECOND, or SECOND, then PRECISION in *IDA* is set to 6, DATETIME_INTERVAL_PRECISION in *IDA* is set to 2 and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

2) If *V* indicates YEAR, MONTH, DAY, HOUR, MINUTE, YEAR TO MONTH, DAY TO HOUR, DAY TO MINUTE, or HOUR TO MINUTE, then PRECISION in *IDA* is set to 0 (zero), DATETIME_INTERVAL_PRECISION in *IDA* is set to 2 and DATETIME_INTERVAL_CODE in *IDA* is set to *V*.

3) Otherwise, an exception condition is raised: *dynamic SQL error — invalid DATETIME_INTERVAL_CODE (0700F)*.

iii) Otherwise, an exception condition is raised: *dynamic SQL error — invalid DATETIME_INTERVAL_CODE (0700F)*.

e) Otherwise, the General Rules of Subclause 9.2, "Store assignment", are applied with the component of *IDA* identified by *DIN* as *TARGET* and *V* as *VALUE*.

5) For each <set header information> specified, the General Rules of Subclause 9.2, "Store assignment", are applied with the component identified by <header item name> as *TARGET* and the value of <set header information value specification> as *VALUE*.

6) If *SDA* is a PTF descriptor area, then the TOP_LEVEL_COUNT component of the header is set to the number of SQL item descriptors of *SDA* in which the LEVEL component is 0 (zero).

7) If an exception condition is raised in a <set descriptor statement>, then the values of all elements of the item descriptor area specified in the <set descriptor statement> are implementation-dependent (UV114).

8) Restrictions on changing TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME values resulting from the execution of a <describe statement> before execution of an <execute statement>, <dynamic open statement>, or <dynamic fetch statement> are implementation-defined (IA235).

Conformance Rules

1) Without at least one of Feature B031, "Basic dynamic SQL", or Feature B209, "PTF extended names", conforming SQL language shall not contain a <set descriptor statement>.

2) Without Feature B209, “PTF extended names”, <descriptor name> shall not be a <PTF descriptor name>.

20.6 <copy descriptor statement>

Function

Copy one SQL descriptor area, in whole or in part, to another SQL descriptor area.

Format

```
<copy descriptor statement> ::=
    <copy whole descriptor statement>
  | <copy item descriptor statement>

<copy whole descriptor statement> ::=
  COPY <source descriptor name> TO <target descriptor name>

<copy item descriptor statement> ::=
  COPY <source descriptor name> VALUE <item number 1>
      <left paren> <copy descriptor options> <right paren>
      TO <target descriptor name> VALUE <item number 2>

<source descriptor name> ::=
  <descriptor name>

<target descriptor name> ::=
  <PTF descriptor name>

<item number 1> ::=
  <simple value specification>

<item number 2> ::=
  <simple value specification>

<copy descriptor options> ::=
    NAME
  | TYPE
  | NAME <comma> TYPE
  | DATA
```

Syntax Rules

1) The declared types of <item number 1> and <item number 2> shall be exact numeric with scale 0 (zero).

Access Rules

None.

General Rules

1) Let *CDS* be the <copy descriptor statement>.

2) If <source descriptor name> *SDN* does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

3) Let *SDA* be the SQL descriptor area identified by *SDN*. Let *SIC* be the number of item descriptor areas of *SDA*.

4) If <target descriptor name> *TDN* does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

5) Let *TDA* be the SQL descriptor area identified by *TDN*. Let *TIC* be the number of item descriptor areas of *TDA*.

6) If <copy whole descriptor statement> is specified, then:

   a) If *SIC* is greater than the maximum number of SQL item descriptors permitted by *TDA*, then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

   b) The number of SQL item descriptors of *TDA* is set to *SIC*.

   c) The header and all SQL item descriptors of *SDA* are copied to the header and the corresponding SQL item descriptors of *TDA*.

7) If <copy item descriptor statement> is specified, then:

   a) Let *IN1* be the value of <item number 1>.

   b) If *IN1* is greater than *SIC* or less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

   c) Let *SI* be the *IN1*-th SQL item descriptor area of *SDA*. If the LEVEL component of *SI* is not 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

   d) Let *IN2* be the value of <item number 2>.

   e) If *IN2* is less than 1 (one) or greater than the maximum number of SQL item descriptors permitted by *TDA*, then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

   f) Case:

      i) If *IN2* ≤ *TIC*, then let *TI* be the *IN2*-th SQL item descriptor area of *TDA*. If the LEVEL component of *TI* is not 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

      ii) Otherwise, the number of SQL item descriptors in *TDA* is increased to *IN2*. The LEVEL component of each new SQL item descriptor is set to 0 (zero). The COUNT component of the header area of *TDA* is increased to *IN2*. The TOP_LEVEL_COUNT component of *TDA* is set to the number of SQL item descriptors in *SDA* whose LEVEL component is 0 (zero). Let *TI* be the *IN2*-th item descriptor area of *TDA*.

   g) Let *CDO* be the <copy descriptor options>.

   h) Case:

      i) If *CDO* is DATA, then the DATA component of *TI* is set to the value of the DATA component of *SI*.

      ii) If *CDO* is NAME, then the NAME component of *TI* is set to the value of the NAME component of *SI*.

      iii) Otherwise:

         1) If *CDO* contains NAME, then the NAME component of *TI* is set to the value of the NAME component of *SI*.

         2) Case:

A) If *SI* has no subordinate items, then the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME components of *TI* are set to the value of the corresponding components of *SI*.

B) Otherwise, let *K* be the number of subordinate items of *SI*.

I) If *IN2* + *K* is greater than the maximum number of SQL item descriptors permitted in *TDA*, then an exception condition is raised: *dynamic SQL error — invalid descriptor index (07009)*.

II) If *IN2* + *K* > *TIC*, then the number of SQL item descriptors in *TDA* is increased to *IN2* + *K*. The LEVEL component of each new SQL item descriptor is set to 0 (zero). The COUNT component of the header area of *TDA* is increased to *IN2* + *K*.

III) For all *k*, 0 (zero) ≤ *k* ≤ *K*:

1) Let *SIK* be the (*IN2* + *k*)-th item descriptor area of *SDA*.

2) Let *TIK* be the (*IN2* + *k*)-th item descriptor area of *TDA*.

3) The LEVEL, TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME components of *TIK* are set to the value of the corresponding components of *SIK*.

IV) The TOP_LEVEL_COUNT component of the header of *TDA* is set to the number of SQL item descriptors of *TDA* in which the LEVEL component is 0 (zero).

Conformance Rules

1) Without Feature B209, "PTF extended names", conforming SQL language shall not contain <copy descriptor statement>.

20.7 <prepare statement>

This Subclause is modified by Subclause 16.1, "<prepare statement>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 15.2, "<prepare statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.4, "<prepare statement>", in ISO/IEC 9075-14.

Function

Prepare a statement for execution.

Format

```
<prepare statement> ::=
  PREPARE <SQL statement name> [ <attributes specification> ]
      FROM <SQL statement variable>

<attributes specification> ::=
  ATTRIBUTES <attributes variable>

<attributes variable> ::=
  <simple value specification>

<SQL statement variable> ::=
  <simple value specification>

<preparable statement> ::=
    <preparable SQL data statement>
  | <preparable SQL schema statement>
  | <preparable SQL transaction statement>
  | <preparable SQL control statement>
  | <preparable SQL session statement>
  | <preparable implementation-defined statement>

<preparable SQL data statement> ::=
    <delete statement: searched>
  | <dynamic single row select statement>
  | <insert statement>
  | <dynamic select statement>
  | <update statement: searched>
  | <truncate table statement>
  | <merge statement>
  | <preparable dynamic delete statement: positioned>
  | <preparable dynamic update statement: positioned>
  | <hold locator statement>
  | <free locator statement>

<preparable SQL schema statement> ::=
  <SQL schema statement>

<preparable SQL transaction statement> ::=
  <SQL transaction statement>

<preparable SQL control statement> ::=
  <SQL control statement>

<preparable SQL session statement> ::=
  <SQL session statement>

<dynamic select statement> ::=
  <cursor specification>
```

```
<preparable implementation-defined statement> ::=
  !! See the Syntax Rules.
```

Syntax Rules

1) The <simple value specification> of <SQL statement variable> shall not be a <literal>.

2) The declared types of each of <SQL statement variable> and <attributes variable> shall be character string.

3) The Format and Syntax Rules for <preparable implementation-defined statement> are implementation-defined (IA164).

4) A <preparable SQL control statement> shall not contain an <SQL procedure statement> that is not a <preparable statement>, nor shall it contain a <dynamic single row select statement> or a <dynamic select statement>.

Access Rules

None.

General Rules

1) 09 Let *P* be the contents of the <SQL statement variable>. If *P* is an <SQL control statement>, then let *PS* be an <SQL procedure statement> contained in *P*.

2) Two subfields *SF1* and *SF2* of row types *RT1* and *RT2* are *corresponding subfields* if *SF1* and *SF2* are positionally corresponding fields of *RT1* and *RT2*, respectively, or if *SF1* and *SF2* are positionally corresponding fields of *RT1SF1* and *RT2SF2* and *RT1SF1* and *RT2SF2* are the declared types of corresponding subfields of *RT1* and *RT2*, respectively.

3) Let *DTGN* be the default transform group name and let *TFL* be the list of {user-defined type name — transform group name} pairs used to identify the group of transform functions for every user-defined type that is referenced in *P*. *DTGN* and *TFL* are not affected by the execution of a <set transform group statement> after *P* is prepared.

4) A data type is *undefined* if it is neither a data type defined in this standard nor a data type defined by the SQL-implementation.

5) 04 Let *MP* be the implementation-defined (IL009) maximum value of <precision> for the NUMERIC data type. Let *ML* be the implementation-defined (IL006) maximum length of variable-length character strings. A <value expression> that is either a <dynamic parameter specification> or a <dynamic parameter specification> immediately enclosed in any number of <left paren>-<right paren> pairs is called a *dynamic parameter marker*. For each dynamic parameter marker *DP* in *P* or *PS*, let *DT* denote the declared type of *DP*, which is undefined unless defined by these General Rules. The syntactic substitutions specified in Subclause 14.15, "<set clause list>", shall not be applied until the data types of <dynamic parameter specification>s are determined by this General Rule.

   a) Case:

      i) If *DP* is immediately followed by an <interval qualifier> *IQ*, then *DT* is INTERVAL *IQ*.

      ii) If *DP* is the <numeric value expression> simply contained in an <array element reference>, then *DT* is NUMERIC (*MP*, 0).

iii) If *DP* is the <string value expression> simply contained in a <char length expression> or an <octet length expression>, then *DT* is CHARACTER VARYING(*ML*) with an implementation-defined (IV219) character set.

iv) If *DP* is either the <numeric value expression dividend> *X1* or the <numeric value expression divisor> *X2* simply contained in a <modulus expression>, then *DT* is the declared type of *X2* or the declared type of *X1*, respectively.

v) If *DP* is either *X1* or *X2* in a <position expression> of the form "POSITION (*X1* IN *X2*)", then

Case:

1) If the declared type of *X2* or the declared type of *X1*, respectively, is CHARACTER or CHARACTER VARYING with character set *CS*, then *DT* is CHARACTER VARYING (*ML*) with character set *CS*.

2) Otherwise, *DT* is the declared type of *X2* or *X1*, respectively.

vi) If *DP* is either *X2* or *X3* in a <string value function> of the form "SUBSTRING (*X1* FROM *X2* FOR *X3*)" or "SUBSTRING (*X1* FROM *X2*)", then *DT* is NUMERIC (*MP*, 0).

vii) If *DP* is either *X1*, *X2*, or *X3* in a <string value function> of the form "SUBSTRING (*X1* SIMILAR *X2* ESCAPE *X3*)", then:

1) Case:

A) If the declared type of *X1* is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then let *CS* be the character set of *X1*.

B) If the declared type of *X2* is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then let *CS* be the character set of *X2*.

C) If the declared type of *X3* is CHARACTER, CHARACTER VARYING, or CHARACTER LARGE OBJECT, then let *CS* be the character set of *X3*.

D) Otherwise, *CS* is undefined.

2) If *CS* is defined, then:

A) If *DP* is *X1* or *X2*, then *DT* is CHARACTER VARYING(*ML*) with character set *CS*.

B) If *DP* is *X3*, then *DT* is CHARACTER(1) with character set *CS*.

viii) If *DP* is any of *X1*, *X2*, *X3*, or X4 in a <string value function> of the form "OVERLAY (*X1* PLACING *X2* FROM *X3* FOR X4)" or "OVERLAY (*X1* PLACING *X2* FROM *X3*)", then

Case:

1) If *DP* is *X1* or *X2*, then

Case:

A) If the declared type of *X2* or *X1*, respectively, is CHARACTER or CHARACTER VARYING with character set *CS*, *DT* is CHARACTER VARYING (*ML*) with character set *CS*.

B) Otherwise, *DT* is the declared type of *X2* or *X1*, respectively.

2) Otherwise, *DT* is NUMERIC (*MP*, 0).

ix) If *DP* is either *X1* or *X2* in a <value expression> of the form "*X1* || *X2*", then

Case:

1) If the declared type of *X2* or *X1*, respectively, is CHARACTER or CHARACTER VARYING with character set *CS*, then *DT* is CHARACTER VARYING (*ML*) with character set *CS*.

2) Otherwise, *DT* is the declared type of *X2* or *X1*, respectively.

x) If *DP* is either *X1* or *X2* in <value expression> of the form "*X1* * *X2*" or "*X1* / *X2*", then

Case:

1) If *DP* is *X1*, then *DT* is the declared type of *X2*.

2) Otherwise,

Case:

A) If the declared type of *X1* is an interval type, then *DT* is NUMERIC (*MP*, 0).

B) Otherwise, *DT* is the declared type of *X2* or *X1*, respectively.

xi) If *DP* is either *X1* or *X2* in a <value expression> of the form "*X1* + *X2*" or "*X1* - *X2*", then

Case:

1) If *DP* is *X1* in an expression of the form "*X1* - *X2*", then *DT* is the declared type of *X2*.

2) Otherwise,

Case:

A) If the declared type of *X2* or *X1*, respectively, is date, then *DT* is INTERVAL YEAR (*PR*) TO MONTH, where *PR* is the implementation-defined (IL059) maximum <interval leading field precision>.

B) If the declared type of *X2* or *X1*, respectively, is time or timestamp, then *DT* is INTERVAL DAY (*PR*) TO SECOND(*FR*), where *PR* and *FR* are the implementation-defined (IL059) maximum <interval leading field precision> and implementation-defined (IL060) maximum <interval fractional seconds precision>, respectively.

C) Otherwise, *DT* is the declared type of *X2* or *X1*, respectively.

xii) If *DP* is the <value expression primary> simply contained in a <boolean primary>, then *DT* is BOOLEAN.

xiii) If *DP* is an <array element> simply contained in an <array element list> *AEL* or *DP* represents the value of a subfield *SF* of the declared type of an <array element> simply contained in an <array element list> *AEL*, then the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <array element>s simply contained in *AEL* as *DTSET*; let *ET* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

1) If *DP* is an <array element> of *AEL*, then *DT* is *ET*.

2) Otherwise, *DT* is the declared type of the subfield of *ET* that corresponds to *SF*.

xiv) If *DP* is a <multiset element> simply contained in a <multiset element list> *MEL* or *DP* represents the value of a subfield *SF* of the declared type of a <multiset element> simply

contained in a <multiset element list> *MEL*, then the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <multiset element>s simply contained in *MEL* as *DTSET*; let *ET* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

1) If *DP* is a <multiset element> of *MEL*, then *DT* is *ET*.

2) Otherwise, *DT* is the declared type of the subfield of *ET* that corresponds to *SF*.

xv) If *DP* is the <cast operand> simply contained in a <cast specification> *CS* or *DP* represents the value of a subfield *SF* of the declared type of the <cast operand> simply contained in a <cast specification> *CS*, then let *CT* be the simply contained <cast target> of *CS*.

1) Let *RT* be a data type determined as follows.

Case:

A) If *CT* immediately contains ARRAY or MULTISET, then *RT* is undefined.

B) If *CT* immediately contains <data type>, then *RT* is that data type.

C) If *CT* simply contains <domain name> *D*, then *RT* is the declared type of the domain identified by *D*.

2) Case:

A) If *DP* is the <cast operand> of *CS*, *DT* is *RT*.

B) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xvi) If *DP* is a <value expression> simply contained in a <case abbreviation> *CA* or *DP* represents the value of a subfield *SF* of the declared type of such a <value expression>, then the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <value expression>s simply contained in *CA* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

1) If *DP* is a <value expression> simply contained in *CA*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xvii) If *DP* is a <result expression> simply contained in a <case specification> *CE* or *DP* represents the value of a subfield *SF* of the declared type of such a <result expression>, then Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <result expression>s simply contained in *CE* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

1) If *DP* is a <result expression> simply contained in *CE*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xviii) If *DP* is a <case operand> or <when operand> simply contained in a <simple case> *CE* or *DP* represents the value of a subfield *SF* of the declared type of such a <case operand> or <when operand>, then:

1) Let *SDT* be the set union of the following sets of declared types:

A) The set consisting of the declared type of the <case operand>.

B) The set consisting of the declared type of any <when operand> of *CE* that is a <row value predicand>.

C) The set consisting of the declared types of any <row value predicand> simply contained in any <comparison predicate part 2>, <between predicate part 2>, <character like predicate part 2>, <octet like predicate part 2>, <similar predicate part 2>, <regex like predicate part 2>, <overlaps predicate part 2>, <distinct predicate part 2>, or <member predicate part 2> that is simply contained in *CE*.

D) The set consisting of the declared row type of a <table subquery> simply contained in an <in predicate part 2>, <match predicate part 2>, or <quantified comparison predicate part 2> simply contained in *CE*.

E) [14] The set consisting of the declared types of the <row value expression>s simply contained in an <in value list> simply contained in an <in predicate part 2> simply contained in *CE*.

NOTE 756 — The following "part 2" predicates do not have any value expressions with declared type information: <null predicate part 2>, <normalized predicate part 2>, <set predicate part 2>, <type predicate part 2>.

2) The Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with *SDT* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

3) Case:

A) If *DP* is a <case operand> or <when operand> simply contained in *CE*, then *DT* is *RT*.

B) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xix) If *DP* is a <row value expression> or <contextually typed row value expression> simply contained in a <table value constructor> or <contextually typed table value constructor> *TVC*, or if *DP* represents the value of a subfield *SF* of the declared type of such a <row value expression> or <contextually typed row value expression>, then

Case:

1) Let *RT* be a data type determined as follows.

Case:

A) If *TVC* is simply contained in a <query expression> that is simply contained in an <insert statement> *IS* or if *TVC* is immediately contained in the <insert columns and source> of an <insert statement> *IS*, then *RT* is a row type in which the declared type of the *i*-th field is the declared type of the *i*-th column in the explicit or implicit <insert column list> of *IS* and the degree of *RT* is equal to the number of columns in the explicit or implicit <insert column list> of *IS*.

B) Otherwise, the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <row value expression>s or <contextually typed row value expression>s simply contained in *TVC* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

2) Case:

A) If *DP* is a <row value expression> or <contextually typed row value expression> simply contained in *TVC*, then *DT* is *RT*.

B) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xx) If *DP* is the <value expression> simply contained in a <merge insert value list> of a <merge insert specification> *MIS* of a <merge statement> or if *DP* represents the value of a subfield *SF* of the declared type of such a <value expression>, then let *RT* be the data type indicated in the column descriptor for the positionally corresponding column in the explicit or implicit <insert column list> contained in *MIS*.

Case:

1) If *DP* is the <value expression> simply contained in *MIS*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxi) If *DP* is a <row value predicand> simply contained in a <comparison predicate>, <distinct predicate>, or <between predicate> *PR* or if *DP* represents the value of a subfield *SF* of the declared type of such a <row value predicand>, then Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <row value predicand>s simply contained in *PR* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

1) If *DP* is a <row value predicand> simply contained in *PR*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxii) If *DP* is a <row value predicand> simply contained in a <quantified comparison predicate> or <match predicate> *PR* or *DP* represents the value of a subfield *SF* of the declared type of such a <row value predicand>, then let *RT* be the declared type of the <table subquery> simply contained in *PR*.

Case:

1) If *DP* is a <row value predicand> simply contained in *PR*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxiii) If *DP* is a <row value predicand> simply contained in an <in predicate> *PR* or if *DP* represents the value of a subfield *SF* of the declared type of such a <row value predicand>, then the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of the <row value predicand>s simply contained in *PR* and the declared row type of the <table subquery> (if any) simply contained in *PR* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

1) If *DP* is a <row value predicand> simply contained in *PR*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxiv) If *DP* is the first <row value constructor element> simply contained in either <row value predicand 1> *RV1* or <row value predicand 2> *RV2* in an <overlaps predicate> *PR*, then

Case:

1) If both *RV1* and *RV2* simply contain a <row value constructor predicand> whose first <row value constructor element> is a dynamic parameter marker, then *DT* is TIMESTAMP WITH TIME ZONE.

2) Otherwise, if *DP* is simply contained in *RV1* or *RV2*, then *DT* is the declared type of the first field of *RV2* or *RV1*, respectively.

xxv) If *DP* is simply contained in a <character like predicate>, <octet like predicate>, or <similar predicate> *PR*, then let *X1* represent the <row value predicand> immediately contained in *PR*, let *X2* represent the <character pattern>, the <octet pattern> or the <similar pattern>, and let *X3* represent the <escape character> or the <escape octet>.

Case:

1) If all *X1*, *X2* and *X3* are dynamic parameter markers, then *DT* is CHARACTER VARYING (*ML*) with an implementation-defined (IV219) character set.

2) Otherwise, the Syntax Rules of Subclause 9.5, "Result of data type combinations", are applied with the declared types of *X1*, *X2* and *X3* as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those Syntax Rules.

Case:

A) If *RT* is CHARACTER or CHARACTER VARYING with character set *CS*, then *DT* is CHARACTER VARYING(*ML*) with character set *CS*.

B) Otherwise, *DT* is *RT*.

xxvi) If *DP* is the <value expression> simply contained in an <update source> of a <set clause> *SC* or if *DP* represents the value of a subfield *SF* of the declared type of such a <value expression>, then let *RT* be the declared type of the <update target> or <mutated set clause> specified in *SC*.

Case:

1) If *DP* is the <value expression> simply contained in *SC*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxvii) 04 If *DP* is a <contextually typed row value expression> simply contained in a <multiple column assignment> *MCA* of a <set clause> *SC* or if *DP* represents the value of a subfield *SF* of the declared type of such a <contextually typed row value expression>, then let *RT* be a row type in which the declared type of the *i*-th field is the declared type of the <update target> or <mutated set clause> immediately contained in the *i*-th <set target> contained in the <set target list> of *MCA*.

Case:

1) If *DP* is a <contextually typed row value expression> simply contained in *MCA*, then *DT* is *RT*.

2) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxviii) If *DP* is the <value specification> immediately contained in a <catalog name characteristic>, <schema name characteristic>, <character set name characteristic>, <SQL-path characteristic>, <transform group characteristic>, <role specification> or <set session user identifier statement>, then *DT* is CHARACTER VARYING (*ML*) with an implementation-defined (IV219) character set.

xxix) If *DP* is the <interval value expression> immediately contained in a <set local time zone statement>, then *DT* is INTERVAL HOUR TO MINUTE.

xxx) 04 If *DP* is an <SQL argument> of a <routine invocation> *RI* or if *DP* represents the value of a subfield *SF* of the declared type of a <value expression> immediately contained in such an <SQL argument>, and if *DP* is the *i*-th <SQL argument> of *RI* or is contained in the *i*-th <SQL argument> of *RI*, then:

1) The Syntax Rules of Subclause 9.18, "Invoking an SQL-invoked routine", are applied with *RI* as *ROUTINE INVOCATION*, an empty <schema name> list as *SQLPATH*, and the null value as *UDT*; let *SR* be the *SUBJECT ROUTINE* and let *SAL* be the *STATIC SQL ARG LIST* returned from the application of those Syntax Rules.

NOTE 757 — The *SAL* returned is not used.

2) Let *RT* denote the declared type of the *i*-th SQL parameter of *SR*.

Case:

A) If *DP* is the *i*-th <SQL argument> of *RI*, then *DT* is *RT*.

B) Otherwise, *DT* is the declared type of the subfield of *RT* that corresponds to *SF*.

xxxi) If *DP* is contained in a <window frame preceding> or a <window frame following> contained in a <window specification> *WS*, then

Case:

1) If *WS* specifies ROWS or GROUPS, then *DT* is NUMERIC(*MP*, 0).

2) Otherwise, let *SDT* be the data type of the single <sort key> contained in *WS*.

Case:

A) If *SDT* is a numeric type, then *DT* is *SDT*.

B) If *SDT* is DATE, then *DT* is INTERVAL DAY.

C) If *SDT* is TIME(*P*) WITHOUT TIME ZONE or TIME(*P*) WITH TIME ZONE, then *DT* is INTERVAL HOUR TO SECOND(*P*).

D) If *SDT* is TIMESTAMP(*P*) WITHOUT TIME ZONE or TIMESTAMP(*P*) WITH TIME ZONE, then *DT* is INTERVAL DAY TO SECOND(*P*).

E) If *SDT* is an interval type, then *DT* is *SDT*.

xxxii) If *DP* is a <number of tiles> simply contained in an <ntile function>, or if *DP* is an <nth row> simply contained in an <nth value function>, then *DT* is INTEGER.

xxxiii) If *DP* is <row marker offset>, then *DT* is NUMERIC(*MP*, 0).

xxxiv) If *DP* is <value_of default value> simply contained in a <value_of expression at row> *VOF*, then *DT* is the declared type of the <value expression> immediately contained in *VOF*.

xxxv) If *DP* is a <locator reference> simply contained in a <hold locator statement> or a <free locator statement>, then *DT* is INTEGER.

xxxvi) If *DP* is an <XQuery pattern>, <XQuery option flag>, <regex subject string>, or <XQuery replacement string> immediately contained in a <regex occurrences function>, <regex position expression>, <regex substring function>, <regex transliteration>, or <regex like predicate> *FUN* and if there exists an <XQuery pattern>, <XQuery option flag>, <regex subject string>, or <XQuery replacement string> immediately contained in *FUN*, then let *CS* be the character set of the <XQuery pattern>, <XQuery option flag>, <regex

subject string>, or <XQuery replacement string> that is not a <dynamic parameter specification>. *DT* is CHARACTER VARYING (*ML*) CHARACTER SET *CS*.

xxxvii) [14] If *DP* is a <start position>, <regex occurrence>, or <regex capture group> immediately contained in a <regex occurrences function>, <regex position expression>, <regex substring function>, or <regex transliteration>, then *DT* is NUMERIC(*MP*).

xxxviii) If *DP* is a <physical offset> or <logical offset> simply contained in a <row pattern navigation operation>, then *DT* is NUMERIC(*MP*).

b) If *DT* is undefined, then an exception condition is raised: *syntax error or access rule violation (42000)*.

6) If *P* does not conform to the Format, Syntax Rules, and Access Rules of a <preparable statement>, or if *P* contains a <simple comment>, then

Case:

a) If *P* contains a <preparable dynamic cursor name> that is ambiguous, then an exception condition is raised: *ambiguous cursor name (3C000)*.

b) If *P* contains a <preparable dynamic cursor name> that is invalid, then an exception condition is raised: *invalid cursor name (34000)*.

c) Otherwise, an exception condition is raised: *syntax error or access rule violation (42000)*.

7) Whether a <dynamic parameter specification> is an input argument, an output argument, or both an input and an output argument is determined as follows.

Case:

a) If *P* is a <call statement>, then:

i) Let *SR* be the subject routine of the <routine invocation> *RI* immediately contained in *P*. Let *n* be the number of <SQL argument>s in the <SQL argument list> immediately contained in *RI*.

ii) Let A_y, 1 (one) $\leq y \leq n$, be the *y*-th <SQL argument> of the <SQL argument list> immediately contained in *RI*.

iii) For each <dynamic parameter specification> *D* contained in some <SQL argument> A_k, 1 (one) $\leq k \leq n$:

1) *D* is an input <dynamic parameter specification> if the <parameter mode> of the *k*-th SQL parameter of *SR* is IN or INOUT.

2) *D* is an output <dynamic parameter specification> if the <parameter mode> of the *k*-th SQL parameter of *SR* is OUT or INOUT.

b) Otherwise:

i) If a <dynamic parameter specification> is contained in a <target specification>, then it is an output <dynamic parameter specification>.

ii) If a <dynamic parameter specification> is contained in a <value specification>, then it is an input <dynamic parameter specification>.

8) If <extended statement name> is specified for the <SQL statement name>, then let *S* be <simple value specification> and let *V* be the character string that is the result of

```
TRIM ( BOTH ' ' FROM S )
```

Case:

a) If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid SQL statement identifier (30000)*.

b) Otherwise, let *ESN* be the <extended statement name>. The value of *ESN* is *V*.

9) If <SQL statement name> identifies a prepared statement *PS*, then an implicit

```
DEALLOCATE PREPARE SSN
```

is executed, where *SSN* is an <SQL statement name> that identifies *PS*.

10) *P* is prepared for execution, resulting in a prepared statement *PRP*.

Case:

a) If the <prepare statement> is contained in an <SQL-invoked routine> *R*, then

Case:

i) If the security characteristic of *R* is DEFINER, then the owner of *PRP* is set to the owner of *R*.

ii) Otherwise, *PRP* has no owner.

b) If the <prepare statement> is contained in a triggered action, then the owner of *PRP* is set to the owner of the trigger.

c) Otherwise,

NOTE 758 — If the <prepare statement> is in neither of the above, then it must necessarily be immediately contained in an externally-invoked procedure.

Case:

i) If the SQL-client module that includes the <prepare statement> has a <module authorization identifier> *MAI* and FOR STATIC ONLY was not specified in the <SQL-client module definition>, then the owner of *PRP* is *MAI*.

ii) Otherwise, *PRP* has no owner.

11) If <attributes specification> is specified, then let *ATV* be the contents of the <attributes variable>. If *ATV* is not the zero-length character string and if *ATV* does not conform to the Format and Syntax Rules of Subclause 20.8, "<cursor attributes>", then an exception condition is raised: *syntax error or access rule violation (42000)*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <prepare statement>.

2) Without Feature B034, "Dynamic specification of cursor attributes", conforming SQL language shall not contain an <attributes specification>.

20.8 <cursor attributes>

Function

Specify a list of cursor attributes.

Format

```
<cursor attributes> ::=
  <cursor attribute>...

<cursor attribute> ::=
    <cursor sensitivity>
  | <cursor scrollability>
  | <cursor holdability>
  | <cursor returnability>
```

Syntax Rules

1) Each of <cursor sensitivity>, <cursor scrollability>, <cursor holdability> and <cursor returnability> shall be specified at most once.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

20.9 <deallocate prepared statement>

This Subclause is modified by Subclause 15.3, "<deallocate prepared statement>", in ISO/IEC 9075-9.

Function

Deallocate SQL-statements that have been prepared with a <prepare statement>.

Format

```
<deallocate prepared statement> ::=
  DEALLOCATE PREPARE <SQL statement name>
```

Syntax Rules

1) If <SQL statement name> is a <statement name>, then

   Case:

   a) If the <deallocate prepared statement> is contained in an <SQL-invoked routine>, then the innermost containing <SQL-invoked routine> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <deallocate prepared statement>.

   b) Otherwise, the containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <deallocate prepared statement>.

Access Rules

None.

General Rules

1) 09 If <SQL statement name> does not identify a prepared statement, then an exception condition is raised: *invalid SQL statement name (26000)*.

2) If <SQL statement name> identifies a prepared statement that is the <cursor specification> of an open cursor, then an exception condition is raised: *invalid cursor state (24000)*.

3) The prepared statement identified by the <SQL statement name> is destroyed. The cursor instance descriptor of any declared dynamic cursor that is associated with the prepared statement is destroyed. The cursor declaration descriptor and cursor instance descriptor of any extended dynamic cursor that is associated with the prepared statement identified by the <SQL statement name> is destroyed. If the value of the <SQL statement name> identifies an existing prepared statement that is a <cursor specification>, then any prepared statements that reference that cursor are destroyed.

Conformance Rules

1) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain a <deallocate prepared statement>.

20.10 <describe statement>

This Subclause is modified by Subclause 15.4, "<describe statement>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 15.3, "<describe statement>", in ISO/IEC 9075-15.

Function

Obtain information about the <select list> columns or <dynamic parameter specification>s contained in a prepared statement or about the columns of the result set associated with a cursor.

Format

```
<describe statement> ::=
    <describe input statement>
  | <describe output statement>

<describe input statement> ::=
  DESCRIBE INPUT <SQL statement name> <using descriptor> [ <nesting option> ]

<describe output statement> ::=
  DESCRIBE [ OUTPUT ] <described object> <using descriptor> [ <nesting option> ]

<nesting option> ::=
    WITH NESTING
  | WITHOUT NESTING

<using descriptor> ::=
  USING [ SQL ] DESCRIPTOR <descriptor name>

<described object> ::=
    <SQL statement name>
  | CURSOR <cursor name> STRUCTURE
```

Syntax Rules

1) If <SQL statement name> is a <statement name>, then

   Case:

   a) If the <describe statement> is contained in an <SQL-invoked routine>, then the innermost containing <SQL-invoked routine> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <describe statement>.

   b) Otherwise, the containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <describe statement>.

2) If <nesting option> is not specified, then WITHOUT NESTING is implicit.

3) If <described object> simply contains a <cursor name> *CN*, then *CN* shall identify a received cursor.

4) The <descriptor name> contained in a <describe input statement> shall not be a <PTF descriptor name>.

Access Rules

None.

General Rules

1) 09 If <SQL statement name> is specified and does not identify a prepared statement *PS*, then an exception condition is raised: *invalid SQL statement name (26000)*.

2) If <cursor name> *CN* is specified, then:

   a) Let *CR* be the received cursor identified by *CN*. If *CR* is not in the open state, then an exception condition is raised: *invalid cursor state (24000)*.

   b) Let *CS* be the <cursor specification> contained in the result set descriptor of *CR*. Let *ISSN* be an implementation-dependent (UV116) <SQL statement name> distinct from any other <SQL statement name>s in the SQL-session. The following statement is executed:

      ```
      PREPARE ISSN FROM CS
      ```

      resulting in a prepared statement *PS*.

3) If <descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

4) Let *DA* be the descriptor area identified by <descriptor name>. Let *N* be the <occurrences> specified when *DA* was allocated.

5) Case:

   a) If the statement being executed is a <describe input statement>, then a descriptor for the input <dynamic parameter specification>s for *PS* is stored in *DA*. Let *D* be the number of input <dynamic parameter specification>s in *PS* prior to any syntactic transformations specified in any Syntax Rules of the ISO/IEC 9075 series. If WITH NESTING is specified, then let NS_i, 1 (one) ≤ *i* ≤ *D*, be the number of subordinate descriptors of the descriptor for the *i*-th input dynamic parameter; otherwise, let NS_i be 0 (zero).

      NOTE 759 — If a syntactic transformation has the apparent effect of replicating a <dynamic parameter specification>, then it is understood that each replication of the <dynamic parameter specification> represents the same dynamic parameter and does not constitute a new dynamic parameter. For example:

      ```
      ? BETWEEN 1 AND 3
      ```

      is transformed by the Syntax Rules of Subclause 8.3, "<between predicate>", to:

      ```
      (? >= 1 AND ? <= 3)
      ```

      but there is still only one dynamic parameter, not two, as a result of this transformation.

   b) If the statement being executed is a <describe output statement> and *PS* is a <dynamic select statement> or a <dynamic single row select statement>, then a descriptor for the <select list> columns for *PS* is stored in *DA*. Let *T* be the table defined by *PS* and let *D* be the degree of *T*. If WITH NESTING is specified, then let NS_i, 1 (one) ≤ *i* ≤ *D*, be the number of subordinate descriptors of the descriptor for the *i*-th column of *T*; otherwise, let NS_i be 0 (zero).

   c) Otherwise, a descriptor for the output <dynamic parameter specification>s for *PS* is stored in *DA*. Let *D* be the number of output <dynamic parameter specification>s in *PS*. If WITH NESTING is specified, then let NS_i, 1 (one) ≤ *i* ≤ *D*, be the number of subordinate descriptors of the descriptor for the *i*-th output dynamic parameter; otherwise, let NS_i be 0 (zero).

6) *DA* is set as follows:

   a) Let *TD* be the value of $D+NS_1+NS_2+...+NS_D$. COUNT is set to *TD*.

   b) TOP_LEVEL_COUNT is set to *D*.

c) DYNAMIC_FUNCTION and DYNAMIC_FUNCTION_CODE are set to the identifier and code, respectively, for *PS* as shown in Table 39, "SQL-statement codes". It is implementation-defined (IA163) whether the identifier and code from Table 39, "SQL-statement codes", for <dynamic select statement> or <dynamic single row select statement> is used to describe a <dynamic select statement> or <dynamic single row select statement> that has been prepared but has not yet been executed dynamically.

d) If the statement being executed is a <describe output statement> and *PS* is a <dynamic select statement> or a <dynamic single row select statement>, then

Case:

i) If some subset of the columns of *T* is the primary key of *T*, then KEY_TYPE is set to 1 (one).

ii) If some subset of the columns of *T* is the preferred candidate key of *T*, then KEY_TYPE is set to 2.

iii) Otherwise, KEY_TYPE is set to 0 (zero).

NOTE 760 — Primary keys and preferred candidate keys are defined in Subclause 4.26, "Functional dependencies".

e) If *TD* is greater than *N*, then a completion condition is raised: *warning — insufficient item descriptor areas (01005)*.

f) If *TD* is 0 (zero) or *TD* is greater than *N*, then no item descriptor areas are set. Otherwise:

i) The first *TD* item descriptor areas are set with values from the descriptors and, optionally, subordinate descriptors for

Case:

1) If the statement being executed is a <describe input statement>, then the input <dynamic parameter specification>s.

2) If the statement being executed is a <describe output statement> and the statement being described is a <dynamic select statement> or a <dynamic single row select statement>, then the columns of *T*.

3) Otherwise, the output <dynamic parameter specification>s.

ii) The descriptor for the first such column or <dynamic parameter specification> is assigned to the first item descriptor area.

iii) If the descriptor for the *j*-th column or <dynamic parameter specification> is assigned to the *k*-th item descriptor area, then:

1) The descriptor for the (*j*+1)-th column or <dynamic parameter specification> is assigned to the (*k*+NS_j)+1-th item descriptor area.

2) If WITH NESTING is specified, then the implicitly ordered subordinate descriptors for the *j*-th column or <dynamic parameter specification> are assigned to contiguous item descriptor areas starting at the (*k*+1)-th item descriptor area.

7) An SQL item descriptor area, if set, consists of values for LEVEL, TYPE, NULLABLE, NAME, UNNAMED, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, PARAMETER_SPECIFIC_NAME, and other components depending on the value of TYPE as described below. The DATA and INDICATOR components are not relevant. Those components and components that are not applicable for a particular value of TYPE are set to implementation-dependent (UV115) values.

a) If the SQL item descriptor area is set to a descriptor that is immediately subordinate to another whose LEVEL value is *K*, then LEVEL is set to *K*+1; otherwise, LEVEL is set to 0 (zero).

b) TYPE is set to a code, as shown in Table 30, "Codes used for SQL data types in Dynamic SQL", indicating the declared type of the column, <dynamic parameter specification>, or subordinate descriptor.

c) Case:

  i) If the value of LEVEL is 0 (zero) and the item descriptor area describes a column, then:

    1) If the column is possibly nullable, then NULLABLE is set to 1 (one); otherwise, NULLABLE is set to 0 (zero).

    2) If the column name is implementation-dependent, then NAME is set to the name of the column and UNNAMED is set to 1 (one); otherwise, NAME is set to the <derived column> name for the column and UNNAMED is set to 0 (zero).

    3) If the column is a member of the primary key of *T* and KEY_TYPE was set to 1 (one) or if the column is a member of the preferred candidate key of *T* and KEY_TYPE was set to 2, then KEY_MEMBER is set to 1 (one); otherwise, KEY_MEMBER is set to 0 (zero).

  ii) If the value of LEVEL is 0 (zero) and the item descriptor area describes a <dynamic parameter specification>, then:

    1) NULLABLE is set to 1 (one).

      NOTE 761 — This indicates that the <dynamic parameter specification> can have the null value.

    2) UNNAMED is set to 1 (one) and NAME is set to an implementation-dependent (UV134) name.

    3) KEY_MEMBER is set to 0 (zero).

  iii) Otherwise:

    1) NULLABLE is set to 1 (one).

    2) Case:

      A) If the item descriptor area describes a field of a row, then

        Case:

          I) If the name of the field is implementation-dependent, then NAME is set to the name of the field and UNNAMED is set to 1 (one).

          II) Otherwise, NAME is set to the name of the field and UNNAMED is set to 0 (zero).

      B) Otherwise, UNNAMED is set to 1 (one) and NAME is set to an implementation-dependent (UV135) name.

    3) KEY_MEMBER is set to 0 (zero).

d) Case:

  i) If TYPE indicates a <character string type>, then:

    1) LENGTH is set to the length or maximum length in characters of the character string type.

2) OCTET_LENGTH is set to the maximum possible length in octets of the character string type.

3) CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are set to the fully qualified name of the character string type's character set.

4) COLLATION_CATALOG, COLLATION_SCHEMA and COLLATION_NAME are set to the fully qualified name of the character string type's declared type collation, if any, and otherwise to the zero-length character string.

If the subject <language clause> specifies C, then the lengths specified in LENGTH and OCTET_LENGTH do not include the implementation-defined (IV030) null character that terminates a C character string.

ii) If TYPE indicates a <binary string type>, then LENGTH and OCTET_LENGTH are set to the length or maximum length in octets of the binary string.

iii) If TYPE indicates an <exact numeric type>, then PRECISION and SCALE are set to the precision and scale of the exact numeric.

iv) If TYPE indicates an <approximate numeric type>, then PRECISION is set to the precision of the approximate numeric.

v) If TYPE indicates DECFLOAT, then PRECISION is set to the precision of the decimal floating-point type.

vi) If TYPE indicates a <datetime type>, then LENGTH is set to the length in positions of the datetime type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 31, "Codes associated with datetime data types in Dynamic SQL", to indicate the specific datetime data type and PRECISION is set to the <time precision> or <timestamp precision>, if either is applicable.

vii) If TYPE indicates an <interval type>, then LENGTH is set to the length in positions of the interval type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 32, "Codes used for <interval qualifier>s in Dynamic SQL", to indicate the <interval qualifier> of the interval data type, DATETIME_INTERVAL_PRECISION is set to the <interval leading field precision> and PRECISION is set to the <interval fractional seconds precision>, if applicable.

viii) If TYPE indicates a user-defined type, then USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the fully qualified name of the user-defined type, and USER_DEFINED_TYPE_CODE is set to a code as specified in Table 34, "Codes associated with user-defined types in Dynamic SQL", to indicate the category of the user-defined type.

ix) If TYPE indicates a <reference type>, then:

1) USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the fully qualified name of the referenced type.

2) SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME are set to the fully qualified name of the referenceable table.

3) LENGTH and OCTET_LENGTH are set to the length in octets of the <reference type>.

x) If TYPE indicates ROW, then DEGREE is set to the degree of the row type.

xi) 09 15 If TYPE indicates ARRAY, then CARDINALITY is set to the maximum cardinality of the array type.

e) If LEVEL is 0 (zero) and *PS* is a <call statement>, then:

i) Let *SR* be the subject routine for the <routine invocation> of the <call statement>.

ii) Let D_x be the *x*-th <dynamic parameter specification> simply contained in an SQL argument A_y of the <call statement>.

iii) Let P_y be the *y*-th SQL parameter of *SR*.

NOTE 762 — A *P* whose <parameter mode> is IN can be a <value expression> that contains zero, one, or more <dynamic parameter specification>s. Thus:

— Every D_x maps to one and only one P_y.

— Several D_x instances can map to the same P_y.

— There can be P_y instances that have no D_x instances that map to them.

iv) The PARAMETER_MODE value in the descriptor for each D_x is set to the value from Table 33, "Codes used for input/output SQL parameter modes in Dynamic SQL", that indicates the <parameter mode> of P_y.

v) The PARAMETER_ORDINAL_POSITION value in the descriptor for each D_x is set to the ordinal position of P_y.

vi) The PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, and PARAMETER_SPECIFIC_NAME values in the descriptor for each D_x are set to the values that identify the catalog, schema, and specific name of *SR*.

Conformance Rules

1) Without Feature B036, "Describe input statement", conforming SQL language shall not contain a <describe input statement>.

2) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <describe output statement> that contains a <described object> that is an <SQL statement name>.

3) Without Feature T472, "DESCRIBE CURSOR", conforming SQL language shall not contain a <describe output statement> that contains a <described object> that contains a <cursor name>.

4) Without Feature B209, "PTF extended names", the <descriptor name> contained in a <describe output statement> shall not be a <PTF descriptor name>.

20.11 <input using clause>

This Subclause is modified by Subclause 16.2, "<input using clause>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 15.5, "<input using clause>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.2, "<input using clause>", in ISO/IEC 9075-14.

Function

Supply input values for an <SQL dynamic statement>.

Format

```
<input using clause> ::=
    <using arguments>
  | <using input descriptor>

<using arguments> ::=
  USING <using argument> [ { <comma> <using argument> }... ]

<using argument> ::=
  <general value specification>

<using input descriptor> ::=
  <using descriptor>
```

Syntax Rules

1) 04 The <general value specification> immediately contained in <using argument> shall be either a <host parameter specification>, an <SQL parameter reference>, or an <embedded variable specification>.

2) The <descriptor name> contained in a <using input descriptor> shall not be a <PTF descriptor name>.

Access Rules

None.

General Rules

1) 09 If <descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

2) When an <input using clause> is used in a <dynamic open statement> or as the <parameter using clause> in an <execute statement>, the <input using clause> describes the input <dynamic parameter specification> values for the <dynamic open statement> or the <execute statement>, respectively. Let *PS* be the prepared <dynamic select statement> referenced by the <dynamic open statement> or the prepared statement referenced by the <execute statement>, respectively.

3) Let *D* be the number of input <dynamic parameter specification>s in *PS*, prior to the application of any syntactic transformations specified in the Syntax Rules of this document.

4) If <using arguments> is specified and the number of <using argument>s is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications (07001)*.

5) If <using input descriptor> is specified, then:

a) Let *N* be the value of COUNT.

b) If *N* is greater than the value of <occurrences> specified when the SQL descriptor area identified by <descriptor name> was allocated or is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count (07008)*.

c) If the first *N* item descriptor areas are not valid as specified in Subclause 20.1, "Description of SQL descriptor areas", then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications (07001)*.

d) In the first *N* item descriptor areas:

i) If the number of item descriptor areas in which the value of LEVEL is 0 (zero) is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications (07001)*.

ii) If the value of INDICATOR is not negative, TYPE does not indicate ROW, and the item descriptor area is not subordinate to an item descriptor area whose INDICATOR value is negative or whose TYPE field indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, and if the value of DATA is not a valid value of the data type represented by the item descriptor area, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications (07001)*.

6) For 1 (one) ≤ *i* ≤ *D*:

a) Let *TDT* be the effective declared type of the *i*-th input <dynamic parameter specification>, defined to be the type represented by the item descriptor area and its subordinate descriptor areas that would be set by a <describe input statement> to reflect the description of the *i*-th input <dynamic parameter specification> of *PS*.

NOTE 763 — See the General Rules of Subclause 20.10, "<describe statement>".

NOTE 764 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

b) Case:

i) If <using input descriptor> is specified, then:

1) Let *IDA* be the *i*-th item descriptor area whose LEVEL value is 0 (zero).

2) Let *SDT* be the effective declared type represented by *IDA*.

NOTE 765 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

3) Let *SV* be the *associated value* of *IDA*.

Case:

A) If the value of INDICATOR is negative, then *SV* is the null value.

B) Otherwise,

Case:

I) If TYPE indicates ROW, then *SV* is a row whose type is *SDT* and whose field values are the associated values of the immediately subordinate descriptor areas of *IDA*.

II) Otherwise, *SV* is the value of DATA with data type *SDT*.

ii) If <using arguments> is specified, then let *SDT* and *SV* be the declared type and value, respectively, of the *i*-th <using argument>.

c) Case:

i) If *SDT* is a locator type, then

Case:

1) If *SV* is not the null value, then let the value of the *i*-th dynamic parameter be the value of *SV*.

2) Otherwise, let the value of the *i*-th dynamic parameter be the null value.

ii) [14] If *SDT* and *TDT* are predefined data types, then

Case:

1) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.13, "<cast specification>", and there is an implementation-defined conversion from type *STD* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th input dynamic parameter.

2) Otherwise:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.13, "<cast specification>", then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation (07006)*.

B) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed and is the value of the *i*-th input dynamic parameter.

iii) If *SDT* is a predefined data type and *TDT* is a user-defined type, then:

1) Let *DT* be the data type identified by *TDT*.

2) If the current SQL-session has a group name corresponding to the user-defined type name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

3) The Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with *DT* as *TYPE* and *GN* as *GROUP*; let *TSF* be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules.

Case:

A) If there is an applicable to-sql function *TSF* and *TSF* is an SQL-invoked method, then let *TSFPT* be the declared type of the second SQL parameter of *TSF*; otherwise, let *TSFPT* be the declared type of the first SQL parameter of *TSF*.

Case:

I) If *TSFPT* is compatible with *SDT*, then

Case:

1) If *TSF* is an SQL-invoked method, then *TSF* is effectively invoked with the value returned by the function invocation:

```
DT()
```

as the first parameter and *SV* as the second parameter. The <return value> is the value of the *i*-th input dynamic parameter.

2) Otherwise, *TSF* is effectively invoked with *SV* as the first parameter. The <return value> is the value of the *i*-th input dynamic parameter.

II) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation (07006)*.

B) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation (0700B)*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <input using clause>.

20.12 <output using clause>

This Subclause is modified by Subclause 16.3, "<output using clause>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 15.6, "<output using clause>", in ISO/IEC 9075-9.
This Subclause is modified by Subclause 17.3, "<output using clause>", in ISO/IEC 9075-14.

Function

Supply output variables for an <SQL dynamic statement>.

Format

```
<output using clause> ::=
    <into arguments>
  | <into descriptor>

<into arguments> ::=
  INTO <into argument> [ { <comma> <into argument> }... ]

<into argument> ::=
  <target specification>

<into descriptor> ::=
  INTO [ SQL ] DESCRIPTOR <descriptor name>
```

Syntax Rules

1) 04 The <target specification> immediately contained in <into argument> shall be either a <host parameter specification>, an <SQL parameter reference>, or an <embedded variable specification>.

Access Rules

None.

General Rules

1) 09 When an <output using clause> is used in a <dynamic fetch statement> or as the <result using clause> of an <execute statement>, let *PS* be the prepared <dynamic select statement> referenced by the <dynamic fetch statement> or the prepared <dynamic single row select statement> referenced by the <execute statement>, respectively.

2) Case:

   a) If *PS* is a <dynamic select statement> or a <dynamic single row select statement>, then let *D* be the degree of the table specified by *PS*.

   b) Otherwise, let *D* be the number of output <dynamic parameter specification>s contained in *PS*.

3) If <into arguments> is specified and the number of <into argument>s is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications (07002)*.

4) If <into descriptor> is specified, then:

a) If <descriptor name> does not identify an SQL descriptor area, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

b) Let *N* be the value of COUNT.

c) If *N* is greater than the value of <occurrences> specified when the SQL descriptor area identified by <descriptor name> was allocated or less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count (07008)*.

d) If the first *N* item descriptor areas are not valid as specified in Subclause 20.1, "Description of SQL descriptor areas", then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications (07002)*.

e) In the first *N* item descriptor areas, if the number of item descriptor areas in which the value of LEVEL is 0 (zero) is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications (07002)*.

5) For 1 (one) ≤ *i* ≤ *D*:

a) Let *SDT* be the effective declared type of the *i*-th <select list> column or output <dynamic parameter specification>, defined to be the type represented by the item descriptor area and its subordinate descriptor areas that would be set by

Case:

i) If *PS* is a <dynamic select statement> or a <dynamic single row select statement>, then a <describe output statement> to reflect the description of the *i*-th <select list> column; let *SV* be the value of that <select list> column, with data type *SDT*.

ii) Otherwise, a <describe output statement> to reflect the description of the *i*-th output <dynamic parameter specification>; let *SV* be the value of that <dynamic parameter specification>, with data type *SDT*.

NOTE 766 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

b) Case:

i) If <into descriptor> is specified, then let *TDT* be the declared type of the *i*-th <target specification> as represented by the *i*-th item descriptor area *IDA* whose LEVEL value is 0 (zero).

NOTE 767 — "Represented by", as applied to the relationship between a data type and an item descriptor area, is defined in the Syntax Rules of Subclause 20.1, "Description of SQL descriptor areas".

ii) If <into arguments> is specified, then let *TDT* be the declared type of the *i*-th <into argument>.

c) If the <output using clause> is directly contained in a <dynamic fetch statement>, then let *CR* be the dynamic cursor identified by the <dynamic fetch statement>, and let *LTDT* be the most specific type of the *i*-th <target specification> or <into argument> of the most recently executed <dynamic fetch statement> prior to the current execution, if any, for the cursor *CR*. It is implementation-defined (IA162) whether or not an exception condition is raised: *dynamic SQL error — restricted data type attribute violation (07006)* if any of the following are true:

i) *LTDT* and *TDT* both identify a binary large object type and only one of *LTDT* and *TDT* is a binary large object locator.

ii) *LTDT* and *TDT* both identify a character large object type and only one of *LTDT* and *TDT* is a character large object locator.

iii) *LTDT* and *TDT* both identify an array type and only one of *LTDT* and *TDT* is an array locator.

iv) *LTDT* and *TDT* both identify a multiset type and only one of *LTDT* and *TDT* is a multiset locator.

v) *LTDT* and *TDT* both identify a user-defined type and only one of *LTDT* and *TDT* is a user-defined type locator.

d) Case:

i) If *TDT* is a locator type, then

Case:

1) If *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and is the value *TV* of the *i*-th <target specification>.

2) Otherwise, the value *TV* of the *i*-th <target specification> is the null value.

ii) [14] If *STD* and *TDT* are predefined data types, then

Case:

1) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.13, "<cast specification>", and there is an implementation-defined conversion of type *STD* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th <target specification>.

2) Otherwise:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.13, "<cast specification>", then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation (07006)*.

B) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed, and is the value *TV* of the *i*-th <target specification>.

iii) If *SDT* is a user-defined type and *TDT* is a predefined data type, then:

1) Let *DT* be the data type identified by *SDT*.

2) If the current SQL-session has a group name corresponding to the user-defined type name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

3) The Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with *DT* as *TYPE* and *GN* as *GROUP*; let *FSF* be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules.

Case:

A) If there is an applicable from-sql function *FSF*, then let *FSFRT* be the <returns data type> of *FSF*.

Case:

I) If *FSFRT* is compatible with *TDT*, then the from-sql function *FSF* is effectively invoked with *SV* as its input SQL parameter and the <return value> is the value *TV* of the *i*-th <target specification>.

II) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation (07006)*.

B) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation (0700B)*.

e) Case:

i) If <into descriptor> is specified, then *IDA* is set to reflect the value of *TV* as follows.

Case:

1) If TYPE indicates ROW, then

Case:

A) If *TV* is the null value, then the value of INDICATOR in *IDA* and in all subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR is set to –1.

B) Otherwise, the *i*-th subordinate descriptor area of *IDA* is set to reflect the value of the *i*-th field of *TV* by applying this subrule (beginning with the outermost "Case") to the *i*-th subordinate descriptor area of *IDA* as *IDA*, the value of the *i*-th field of *TV* as *TV*, the value of the *i*-th field of *SV* as *SV*, and the data type of the *i*-th field of *SV* as *SDT*.

2) Otherwise,

Case:

A) If *TV* is the null value, then the value of INDICATOR is set to –1.

B) If *TV* is not the null value, then:

I) The value of INDICATOR is set to 0 (zero).

II) Case:

1) If TYPE indicates a locator type, then a locator *L* that uniquely identifies *TV* is generated and the value of DATA is set to an implementation-dependent (UV043) four-octet integer value that represents *L*.

2) Otherwise, the value of DATA is set to *TV*.

III) Case:

1) If TYPE indicates CHARACTER VARYING, CHARACTER LARGE OBJECT, BINARY VARYING, or BINARY LARGE OBJECT, then RETURNED_LENGTH is set to the length in characters or octets, respectively, of *TV*, and RETURNED_OCTET_LENGTH is set to the length in octets of *TV*.

2) If *SDT* is CHARACTER VARYING, CHARACTER LARGE OBJECT, BINARY VARYING, BINARY LARGE OBJECT, then RETURNED_LENGTH is set to the length in characters or octets, respectively, of *SV*, and RETURNED_OCTET_LENGTH is set to the length in octets of *SV*.

3) If TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then RETURNED_CARDINALITY is set to the cardinality of *TV*.

ii) If <into arguments> is specified, then the General Rules of Subclause 9.1, "Retrieval assignment", are applied with the *i*-th <into argument> as *TARGET* and *TV* as *VALUE*.

NOTE 768 — All other values of the SQL descriptor area are unchanged.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <output using clause>.

20.13 <execute statement>

This Subclause is modified by Subclause 15.7, "<execute statement>", in ISO/IEC 9075-9.

Function

Associate input SQL parameters and output targets with a prepared statement and execute the statement.

Format

```
<execute statement> ::=
  EXECUTE <SQL statement name> [ <result using clause> ] [ <parameter using clause> ]

<result using clause> ::=
  <output using clause>

<parameter using clause> ::=
  <input using clause>
```

Syntax Rules

1) If <SQL statement name> is a <statement name>, then

   Case:

   a) If the <execute statement> is contained in an <SQL-invoked routine>, then the innermost containing <SQL-invoked routine> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <execute statement>.

   b) Otherwise, the containing <SQL-client module definition> shall contain a <prepare statement> whose <statement name> is equivalent to the <statement name> of the <execute statement>.

2) A <descriptor name> contained in either a <result using clause> or a <parameter using clause> shall be a <conventional descriptor name>.

Access Rules

None.

General Rules

1) 09 If the <SQL statement name> does not identify a prepared statement *P*, then an exception condition is raised: *invalid SQL statement name (26000)*.

2) Let *PS* be the statement previously prepared using <SQL statement name>.

3) If *PS* is a <dynamic select statement>, then

   Case:

   a) If *PS* does not conform to the Format and Syntax Rules of a <dynamic single row select statement>, then an exception condition is raised: *dynamic SQL error — cursor specification cannot be executed (07003)*.

   b) Otherwise, *PS* is treated as a <dynamic single row select statement>.

4) If *PS* contains the <table name> of a created or declared local temporary table and if the <execute statement> is not in the same <SQL-client module definition> as the <prepare statement> that prepared the prepared statement, then an exception condition is raised: *syntax error or access rule violation (42000)*.

5) If *PS* contains input <dynamic parameter specification>s and a <parameter using clause> is not specified, then an exception condition is raised: *dynamic SQL error — using clause required for dynamic parameters (07004)*.

6) If *PS* is a <dynamic single row select statement> or it contains output <dynamic parameter specification>s and a <result using clause> is not specified, then an exception condition is raised: *dynamic SQL error — using clause required for result fields (07007)*.

7) If a <parameter using clause> is specified, then the General Rules specified in Subclause 20.11, "<input using clause>", for a <parameter using clause> in an <execute statement> are applied.

8) A copy of the top cell is pushed onto the authorization stack. If *PS* has an owner, then the top cell of the authorization stack is set to contain only the authorization identifier of the owner of *PS*.

9) The General Rules of Subclause 9.17, "Executing an <SQL procedure statement>", are applied with *PS* as *EXECUTING STATEMENT*. During this evaluation, for each <dynamic parameter specification> *DPS* contained in the original SQL-statement, let *I* be the ordinal position of DPS within the collection of <dynamic parameter specification>s in the original SQL-statement. If *DPS* is replicated as a result of any syntactic transformation specified in any Syntax Rule in this document, then the value of all such replicated input dynamic parameters are set identically to the value of the *I*-th input dynamic parameter.

10) If a <result using clause> is specified, then the General Rules specified in Subclause 20.12, "<output using clause>", for a <result using clause> in an <execute statement> are applied.

11) Upon completion of execution, the top cell in the authorization stack is removed.

Conformance Rules

1) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain a <result using clause>.

2) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <execute statement>.

20.14 <execute immediate statement>

Function

Dynamically prepare and execute a preparable statement.

Format

```
<execute immediate statement> ::=
  EXECUTE IMMEDIATE <SQL statement variable>
```

Syntax Rules

1) The declared type of <SQL statement variable> shall be character string.

Access Rules

None.

General Rules

1) Let *P* be the contents of the <SQL statement variable>.

2) If at least one of the following are true, then an exception condition is raised: *syntax error or access rule violation (42000)*.

   a) *P* is a <dynamic select statement> or a <dynamic single row select statement>.

   b) *P* contains a <dynamic parameter specification>.

3) Let *SV* be <SQL statement variable>. <execute immediate statement> is equivalent to the following:

```
PREPARE IMMEDIATE_STMT FROM SV ;
EXECUTE IMMEDIATE_STMT ;
DEALLOCATE PREPARE IMMEDIATE_STMT ;
```

   where *IMMEDIATE_STMT* is an implementation-dependent (UV117) <statement name> that does not identify any existing prepared statement.

   NOTE 769 — Exception condition or completion condition information resulting from the PREPARE or EXECUTE is reflected in the diagnostics area.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain an <execute immediate statement>.

20.15 <dynamic declare cursor>

This Subclause is modified by Subclause 15.8, "<dynamic declare cursor>", in ISO/IEC 9075-9.

Function

Declare a declared dynamic cursor to be associated with a <statement name>, which may in turn be associated with a <cursor specification>.

Format

```
<dynamic declare cursor> ::=
  DECLARE <cursor name>
      <cursor properties>
      FOR <statement name>
```

Syntax Rules

1) The <cursor name> shall not be identical to the <cursor name> specified in any other <declare cursor>, <dynamic declare cursor>, or <allocate received cursor statement> in the same <SQL-client module definition> *M*. The scope of the <cursor name> is *M* with the exception of any <SQL schema statement> contained in *M*.

2) Let *SN* be the <statement name> simply contained in the <dynamic declare cursor>. The containing <SQL-client module definition> shall contain, without an intervening <SQL schema statement>, a <prepare statement> whose <statement name> is equivalent to *SN*.

Access Rules

None.

General Rules

1) 09 A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

   a) The kind of cursor is a declared dynamic cursor.

   b) The provenance of the cursor is an indication of the SQL-client module whose <SQL-client module definition> contains the <dynamic declare cursor>.

   c) The name of the cursor is the <cursor name>.

   d) The cursor's origin is *SN*.

   e) The cursor's declared properties are as determined by the <cursor properties>.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic declare cursor>.

20.16 <descriptor value constructor>

Function

Construct a PTF descriptor.

Format

```
<descriptor value constructor> ::=
  DESCRIPTOR <left paren> <descriptor column list> <right paren>

<descriptor column list> ::=
  <descriptor column specification> [ { <comma> <descriptor column specification> }... ]

<descriptor column specification> ::=
  <column name> [ <data type> ]
```

Syntax Rules

1) No two <column name>s in a <descriptor column list> shall be equivalent.

Access Rules

None.

General Rules

1) Let *DVE* be the <descriptor value constructor>. Let *N* be the number of <descriptor column specification>s contained in *DVE*. For *i* ranging from 1 (one) through *N*, let DCS_i be an enumeration of the <descriptor column specification>s contained in *DVE*.

2) An SQL descriptor area *USDA* is created as follows:

   a) For *i* ranging from 1 (one) through *N*, one or more SQL item descriptors are appended to *USDA* corresponding to DCS_i with the following components:

      i) In the first appended SQL item descriptor, the value of LEVEL is 0 (zero) and the value of NAME is the <column name> simply contained in DCS_i.

      ii) Case:

         1) If DCS_i contains <data type> *DT*, then the values of TYPE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, LENGTH, PRECISION, SCALE, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, SCOPE_NAME, and DEGREE are set so as to create a valid SQL item descriptor that represents *DT*. If *DT* identifies a character type, then the values of COLLATION_CATALOG, COLLATION_SCHEMA, and COLLATION_NAME identify the collation of *DT*. If the representation of *DT* requires subordinate descriptor areas, these are appended as well.

NOTE 770 — The Syntax Rule of Subclause 20.1, "Description of SQL descriptor areas", contain definitions of how to form a valid SQL item descriptor (or set of SQL item descriptors, including subordinate SQL item descriptors) that represents a <data type>.

2) Otherwise, the value of TYPE is 0 (zero), the values of CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, COLLATION_CATALOG, COLLATION_SCHEMA, COLLATION_NAME, LENGTH, PRECISION, SCALE, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, SCOPE_NAME, and DEGREE are undefined, and no subordinate SQL item descriptors are appended.

b) The header of *USDA* has the following components:

i) The value of the COUNT component is the total number of SQL item descriptors, including subordinate SQL item descriptors.

ii) The value of TOP_LEVEL_COUNT field is *N*.

iii) All other components in the header of *USDA* are undefined.

Conformance Rules

1) Without Feature B206, "PTF descriptor parameters", conforming SQL language shall not contain <descriptor value constructor>.

20.17 <allocate extended dynamic cursor statement>

This Subclause is modified by Subclause 15.9, "<allocate extended dynamic cursor statement>", in ISO/IEC 9075-9.

Function

Define a cursor based on a prepared statement for a <cursor specification>.

Format

```
<allocate extended dynamic cursor statement> ::=
  ALLOCATE <extended cursor name>
      <cursor properties>
      FOR <extended statement name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) 09 Let *S* be the <simple value specification> immediately contained in <extended cursor name>. Let *V* be the character string that is the result of

   ```
   TRIM ( BOTH ' ' FROM S )
   ```

   Case:

   a) If *V* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid cursor name (34000)*.

   b) Otherwise, let *ECN* be the <extended cursor name>. The value of *ECN* is *V*.

2) If *ECN* identifies a cursor, then an exception condition is raised: *invalid cursor name (34000)*.

3) If <extended statement name> does not identify a prepared statement, then an exception condition is raised: *invalid SQL statement name (26000)*.

4) If the prepared statement *P* identified by <extended statement name> is not a <cursor specification>, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification (07005)*.

5) *P* is re-prepared as follows: if *P* does not conform to the Syntax Rules of Subclause 14.3, "<cursor specification>", then an exception condition is raised: *syntax error or access rule violation (42000)*.

6) A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

   a) The kind of cursor is an extended dynamic cursor.

   b) The provenance of the cursor is

Case:

i) If the <scope option> contained in <extended cursor name> is GLOBAL, then the current SQL-session identifier.

ii) Otherwise, an indication of the SQL-client module whose <SQL-client module definition> contains the <allocate extended dynamic cursor statement>.

c) The name of the cursor is the extended name *V* and the explicit or implicit <scope option> of the <extended cursor name>.

d) The cursor origin is *P*.

e) The cursor's declared properties are as determined by the <cursor properties>.

7) A cursor instance descriptor *CID* is created. *CID* includes indications that:

a) The cursor declaration descriptor is *CDD*.

b) The SQL-session identifier is the current SQL-session identifier.

c) The cursor's state is closed.

8) An association is made between the value of the <extended cursor name> and the prepared statement in the scope of the <extended cursor name>. The association is preserved until the prepared statement is destroyed, at which time the cursor declaration descriptor and the cursor instance descriptor of the cursor identified by <extended cursor name> are also destroyed.

Conformance Rules

1) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain an <allocate extended dynamic cursor statement>.

20.18 <allocate received cursor statement>

This Subclause is modified by Subclause 15.10, "<allocate received cursor statement>", in ISO/IEC 9075-9.

Function

Assign a cursor to the result set sequence returned from an SQL-invoked procedure.

Format

```
<allocate received cursor statement> ::=
  ALLOCATE <cursor name>
      [ CURSOR ] FOR <specific routine designator>
```

Syntax Rules

1) The <cursor name> shall not be identical to the <cursor name> specified in any other <declare cursor>, <dynamic declare cursor>, or <allocate received cursor statement> in the same <SQL-client module definition> *M*. The scope of the <cursor name> is *M* with the exception of any <SQL schema statement> contained in *M*.

2) The SQL-invoked routine identified by <specific routine designator> shall be an SQL-invoked procedure.

Access Rules

None.

General Rules

1) 09 Let *SIP* be the SQL-invoked procedure identified by <specific routine designator>. Let *INV* be the active SQL-invoked routine of the current routine execution context.

2) If the SQL-session context of the current SQL-session does not include a result set sequence *RSS* brought into existence by an invocation of *SIP* by *INV*, then an exception condition is raised: *invalid SQL-invoked procedure reference (0M000)*.

3) If *RSS* is empty, then an exception condition is raised: *no data — no additional result sets returned (02001)*.

4) A cursor declaration descriptor *CDD* is created. *CDD* includes indications that:

   a) The kind of cursor is a received cursor.

   b) The provenance of the cursor is an indication of the SQL-client module whose <SQL-client module definition> contains the <allocate received cursor statement>.

   c) The name of the cursor is the <cursor name>.

   d) The cursor origin is the <specific routine designator>.

   e) The cursor's declared properties are as follows:

      i) The cursor's declared sensitivity is ASENSITIVE.

      ii) The cursor's declared scrollability is NO SCROLL.

iii) The cursor's declared holdability is WITHOUT HOLD.

iv) The cursor's declared returnability is WITHOUT RETURN.

5) A cursor instance descriptor *CID* is created. *CID* includes:

a) The cursor declaration descriptor is *CDD*.

b) The current SQL-session identifier.

c) The cursor's state is open.

6) The General Rules of Subclause 15.2, "Effect of receiving a result set", are applied with *CID* as *CURSOR* and *RSS* as *RESULT SET SEQUENCE*.

Conformance Rules

1) Without Feature T471, "Result sets return value", conforming SQL language shall not contain an <allocate received cursor statement>.

20.19 <dynamic open statement>

This Subclause is modified by Subclause 15.11, "<dynamic open statement>", in ISO/IEC 9075-9.

Function

Associate input dynamic parameters with a <cursor specification> and open the dynamic cursor.

Format

```
<dynamic open statement> ::=
  OPEN <conventional dynamic cursor name> [ <input using clause> ]
```

Syntax Rules

1) If <conventional dynamic cursor name> *DCN* is a <cursor name> *CN*, then *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor. Let *CDD* be the cursor declaration descriptor identified by *CN*.

Access Rules

1) The Access Rules for the <query expression> simply contained in the prepared statement associated with the <conventional dynamic cursor name> are applied.

General Rules

1) Case:

   a) 09 If *DCN* is a <cursor name> *CN*, then

      Case:

      i) If the <statement name> *SN* contained in *CDD* does not identify a prepared statement, then an exception condition is raised: *invalid SQL statement name (26000)*.

      ii) If *SN* does not identify a prepared statement that is a <cursor specification>, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification (07005)*.

   b) Otherwise, if *DCN* does not identify an extended dynamic cursor, then an exception condition is raised: *invalid cursor name (34000)*.

2) Let *CR* be the cursor identified by *DCN*.

3) If the prepared statement *P* associated with the <conventional dynamic cursor name> contains <dynamic parameter specification>s and an <input using clause> is not specified, then an exception condition is raised: *dynamic SQL error — using clause required for dynamic parameters (07004)*.

4) If the <conventional dynamic cursor name> is a <cursor name>, then *P* is re-prepared as follows: if *P* does not conform to the Syntax Rules of Subclause 14.3, "<cursor specification>", then an exception condition is raised: *syntax error or access rule violation (42000)*.

5) *CR* is updatable if and only if the <cursor specification> included in the result set descriptor of *CR* is updatable.

6) If an <input using clause> is specified, then the General Rules specified in Subclause 20.11, "<input using clause>", for <dynamic open statement> are applied.

7) The General Rules of Subclause 15.1, "Effect of opening a cursor", are applied with *CR* as *CURSOR*. For each <dynamic parameter specification> *DPS* contained in the original SQL-statement, let *I* be the ordinal position of DPS within the collection of <dynamic parameter specification>s in the original SQL-statement. If *DPS* is replicated as a result of any syntactic transformation specified in any Syntax Rule in this document, then the value of all such replicated input dynamic parameters are set identically to the value of the *I*-th input dynamic parameter.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic open statement>.

20.20 <dynamic fetch statement>

This Subclause is modified by Subclause 15.12, "<dynamic fetch statement>", in ISO/IEC 9075-9.

Function

Fetch a row for a dynamic cursor.

Format

```
<dynamic fetch statement> ::=
  FETCH [ [ <fetch orientation> ] FROM ] <dynamic cursor name> <output using clause>
```

Syntax Rules

1) If <fetch orientation> is omitted, then NEXT is implicit.

2) If <dynamic cursor name> *DCN* is a <cursor name> *CN*, then *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

Access Rules

None.

General Rules

1) 09 If *DCN* does not identify a cursor, then an exception condition is raised: *invalid cursor name (34000)*.

2) Let *CR* be the cursor identified by *DCN*.

3) The General Rules of Subclause 15.3, "Determination of the current row of a cursor", are applied with *CR* as *CURSOR* and <fetch orientation> as *FETCH ORIENTATION*.

4) If a completion condition *no data (02000)* is raised, then no further General Rules of this Subclause are applied.

5) The General Rules specified in Subclause 20.12, "<output using clause>", for an <output using clause> in a <dynamic fetch statement> are applied.

6) If an exception condition is raised during the assignment of a value to a target, then the values of all targets are implementation-dependent (UV048).

NOTE 771 — It is implementation-dependent whether *CR* remains positioned on the current row when an exception condition is raised during the derivation of any <derived column>.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic fetch statement>.

20.21 <dynamic single row select statement>

Function

Retrieve values from a dynamically-specified row of a table.

Format

```
<dynamic single row select statement> ::=
  <query specification>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *Q* be the result of the <query specification>.

2) Case:

   a) If the cardinality of *Q* is greater than 1 (one), then an exception condition is raised: *cardinality violation (21000)*.

   b) If *Q* is empty, then a completion condition is raised: *no data (02000)*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic single row select statement>.

20.22 <dynamic close statement>

This Subclause is modified by Subclause 15.13, "<dynamic close statement>", in ISO/IEC 9075-9.

Function

Close a dynamic cursor.

Format

```
<dynamic close statement> ::=
  CLOSE <conventional dynamic cursor name>
```

Syntax Rules

1) If <conventional dynamic cursor name> *DCN* is a <cursor name> *CN*, then *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

Access Rules

None.

General Rules

1) If *DCN* does not identify a cursor, then an exception condition is raised: *invalid cursor name (34000)*.

2) Let *CR* be the cursor identified by *DCN*.

3) 09 The General Rules of Subclause 15.4, "Effect of closing a cursor", are applied with *CR* as *CURSOR* and DESTROY as *DISPOSITION*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic close statement>.

20.23 <dynamic delete statement: positioned>

Function

Delete a row of a table.

Format

```
<dynamic delete statement: positioned> ::=
  DELETE FROM <target table> WHERE CURRENT OF <conventional dynamic cursor name>
```

Syntax Rules

1) Let *DDSP* be the <dynamic delete statement: positioned>, let *TT* be the <target table>, and let *DCN* be the <conventional dynamic cursor name>.

2) If *DCN* is a <cursor name> *CN*, then *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

3) Let *TN* be the <table name> contained in *TT*.

Access Rules

1) Case:

   a) If *DDSP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the authorization identifier that owns that schema. The applicable privileges for *A* shall include DELETE for *TN*.

   b) Otherwise, the current privileges shall include DELETE for *TN*.

General Rules

1) If *DCN* does not identify a dynamic cursor, then an exception condition is raised: *invalid cursor name (34000)*.

2) Let *CR* be the cursor identified by *DCN*.

3) If *CR* is not an updatable cursor, then an exception condition is raised: *invalid cursor name (34000)*.

4) Let *T* be the simply underlying table of *CR*. Let *LUT* be the target leaf underlying table of *T*.

5) If *TN* does not identify *LUT*, or if ONLY is specified and the <table reference> in *T* that references *LUT* does not specify ONLY, or if ONLY is not specified and the <table reference> in *T* that references *LUT* does specify ONLY, then an exception condition is raised: *target table disagrees with cursor specification (0T000)*.

6) The General Rules of Subclause 15.6, "Effect of a positioned delete", are applied with *CR* as *CURSOR*, *DDSP* as *STATEMENT*, and *TT* as *TARGET*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic delete statement: positioned>.

20.24 <dynamic update statement: positioned>

Function

Update a row of a table.

Format

```
<dynamic update statement: positioned> ::=
  UPDATE <target table> SET <set clause list>
      WHERE CURRENT OF <conventional dynamic cursor name>
```

Syntax Rules

1) Let *DUSP* be the <dynamic update statement: positioned>, let *TT* be the <target table>, let *SCL* be the <set clause list>, and let *DCN* be the <conventional dynamic cursor name>.

2) If *DCN* is a <cursor name> *CN*, then *CN* shall be contained within the scope of a <cursor name> that is equivalent to *CN*. *CN* shall identify a declared dynamic cursor.

3) Let *TN* be the <table name> contained in *TT*.

4) The scope of *TN* is *DUSP*.

Access Rules

1) Case:

   a) If *DUSP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* shall include UPDATE for each <object column>.

   b) Otherwise, the current privileges shall include UPDATE for each <object column>.

General Rules

1) If *DCN* does not identify a dynamic cursor, then an exception condition is raised: *invalid cursor name (34000)*.

2) Let *CR* be the cursor identified by *DCN*.

3) If *CR* is not an updatable cursor, then an exception condition is raised: *invalid cursor name (34000)*.

4) Let *T* be the simply underlying table of *CR*. Let *LUT* be the target leaf underlying table of *T*.

5) If *TN* does not identify *LUT*, or if ONLY is specified and the <table reference> in *T* that references *LUT* does not specify ONLY, or if ONLY is not specified and the <table reference> in *T* that references *LUT* does specify ONLY, then an exception condition is raised: *target table disagrees with cursor specification (0T000)*.

6) If any object column is directly or indirectly referenced in the <order by clause> simply contained in the <cursor specification> for *CR*, then an exception condition is raised: *attempt to assign to ordering column (0V000)*.

7) If any object column identifies a column that is not identified by a <column name> contained in the explicit or implicit <column name list> of the explicit or implicit <updatability clause> of the <cursor specification> included in the result set descriptor of *CR*, then an exception condition is raised: *attempt to assign to non-updatable column (0U000)*.

8) The General Rules of Subclause 15.7, "Effect of a positioned update", are applied with *CR* as *CURSOR*, *SCL* as *SET CLAUSE LIST*, *DUSP* as *STATEMENT*, and *TT* as *TARGET*.

Conformance Rules

1) Without Feature B031, "Basic dynamic SQL", conforming SQL language shall not contain a <dynamic update statement: positioned>.

20.25 <preparable dynamic delete statement: positioned>

Function

Delete a row of a table through a dynamic cursor.

Format

```
<preparable dynamic delete statement: positioned> ::=
  DELETE [ FROM <target table> ]
      WHERE CURRENT OF <preparable dynamic cursor name>
```

Syntax Rules

1) Let *PDDSP* be the <preparable dynamic delete statement: positioned>. Let *PDCN* be the <preparable dynamic cursor name>.

2) If *PDCN* is not ambiguous or invalid, then:

   a) Let *CR* be the cursor identified by *PDCN*. *CR* shall be an updatable cursor.

   b) Let *QE* be the <query expression> simply contained in the <cursor specification> of the result set descriptor of *CR*. Let *LUT* be the target leaf underlying table of *QE*.

   c) Case:

      i) If <target table> is not specified, then let *TN* be the name of *LUT*.

         Case:

         1) If the <table reference> that references *LUT* specifies ONLY, then the <target table>

            ```
            ONLY ( TN )
            ```

            is implicit.

         2) Otherwise, the <target table>

            ```
            TN
            ```

            is implicit.

      ii) Otherwise, let *TN* be the <table name> contained in <target table>. *TN* shall identify *LUT*.

   d) Let *TT* be the explicit or implicit <target table>.

   e) *LUT* shall not be an old transition table or a new transition table.

   f) If *TT* immediately contains ONLY and *LUT* is not a typed table, then *TT* is equivalent to *TN*.

   g) *TT* shall specify ONLY if and only if the <table reference> contained in *T* that references *LUT* specifies ONLY.

   h) The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *LUT*.

Access Rules

1) If *PDCN* is not ambiguous or invalid, then

   Case:

   a) If *PDDSP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the authorization identifier that owns that schema. The applicable privileges for *A* shall include DELETE for *TN*.

   b) Otherwise, the current privileges shall include DELETE for *TN*.

General Rules

1) The General Rules of Subclause 15.6, "Effect of a positioned delete", are applied with *CR* as *CURSOR*, *PDDSP* as *STATEMENT*, and *TT* as *TARGET*.

   NOTE 772 — If the General Rules are reached, then *PDCN* cannot be ambiguous or invalid.

Conformance Rules

1) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain a <preparable dynamic delete statement: positioned>.

20.26 <preparable dynamic cursor name>

This Subclause is modified by Subclause 9.1, "<preparable dynamic cursor name>", in ISO/IEC 9075-3.

Function

Specify the cursor of a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>.

Format

```
<preparable dynamic cursor name> ::=
  [ <scope option> ] <cursor name>
```

Syntax Rules

1) Let *PDCN* be the <preparable dynamic cursor name>, let *CN* be the <cursor name> contained in *PDCN*, and let *P* be the <preparable dynamic delete statement: positioned> or <preparable dynamic update statement: positioned> that contains *PDCN*.

   Case:

   a) If *PDCN* contains a <scope option> that specifies GLOBAL, then

      Case:

      i) If there exists in the current SQL-session context an extended dynamic cursor *EDC* with a <conventional dynamic cursor name> having a global scope and a <cursor name> that is equivalent to *CN*, then *EDC* is the cursor referenced by *PDCN*.

      ii) Otherwise, *PDCN* is said to be *invalid*.

   b) If *PDCN* contains a <scope option> that specifies LOCAL, or if no <scope option> is specified, then:

      i) The *potentially referenced cursors* of *PDCN* include:

         1) Every declared dynamic cursor whose <cursor name> is equivalent to *CN* and whose scope is the containing SQL-client module (minus any <SQL schema statement>s contained in the SQL-client module).

         2) 03 Every extended dynamic cursor having a <conventional dynamic cursor name> that has a scope of the containing SQL-client module (minus any <SQL schema statement>s contained in the SQL-client module) and whose <cursor name> is equivalent to *CN*.

      ii) Case:

         1) If the number of potentially referenced cursors is greater than 1 (one), then *PDCN* is said to be *ambiguous*.

         2) If the number of potentially referenced cursors is less than 1 (one), then *PDCN* is said to be *invalid*.

         3) Otherwise, *PDCN* refers to the single potentially referenced cursor of *P*.

Access Rules

None.

General Rules

None.

Conformance Rules

None.

20.27 <preparable dynamic update statement: positioned>

Function

Update a row of a table through a dynamic cursor.

Format

```
<preparable dynamic update statement: positioned> ::=
  UPDATE [ <target table> ] SET <set clause list>
      WHERE CURRENT OF <preparable dynamic cursor name>
```

Syntax Rules

1) Let *PDUSP* be the <preparable dynamic update statement: positioned>. Let *SCL* be the <set clause list>. Let *PDCN* be the <preparable dynamic cursor name>.

2) If *PDCN* is not ambiguous or invalid, then:

   a) Let *CR* be the cursor identified by *PDCN*. *CR* shall be an updatable cursor.

   b) Let *QE* be the <query expression> simply contained in the <cursor specification> of the result set descriptor of *CR*. Let *LUT* be the target leaf underlying table of *QE*.

   c) Case:

      i) If <target table> is not specified, then let *TN* be the name of *LUT*.

         Case:

         1) If the <table reference> that references *LUT* specifies ONLY, then the <target table>

            ```
            ONLY ( TN )
            ```

            is implicit.

         2) Otherwise, the <target table>

            ```
            TN
            ```

            is implicit.

      ii) Otherwise, let *TN* be the <table name> contained in <target table>. *TN* shall identify *LUT*.

   d) Let *TT* be the explicit or implicit <target table>.

   e) *LUT* shall not be an old transition table or a new transition table.

   f) If *TT* immediately contains ONLY and *LUT* is not a typed table, then *TT* is equivalent to *TN*.

   g) *TT* shall specify ONLY if and only if the <table reference> contained in *TU* that references *LUT* specifies ONLY.

   h) The schema identified by the explicit or implicit <schema name> of *TN* shall include the descriptor of *LUT*.

   i) Let *CN* be *TN*. *CN* is an exposed <table or query name>.

j) The scope of *CN* is *SCL*.

k) If *CR* is an ordered cursor, then for each <object column> *OC* contained in *SCL*, no generally underlying column of a <sort key> in the <order by clause> simply contained in the <query expression> of the <cursor specification> for *CR* shall be *OC* or a generally underlying column of *OC*.

l) Each <column name> specified as an <object column> shall identify a column in the explicit or implicit <column name list> contained in the explicit or implicit <updatability clause> of the <cursor specification> included in the result set descriptor of *CR*.

Access Rules

1) If *PDCN* is not ambiguous or invalid, then

Case:

a) If *PDUSP* is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, in an <SQL schema statement>, then let *A* be the <authorization identifier> that owns that schema. The applicable privileges for *A* shall include UPDATE for each <object column>.

b) Otherwise, the current privileges shall include UPDATE for each <object column>.

General Rules

1) The General Rules of Subclause 15.7, "Effect of a positioned update", are applied with *CR* as *CURSOR*, *SCL* as *SET CLAUSE LIST*, *PDUSP* as *STATEMENT*, and *TT* as *TARGET*

NOTE 773 — If the General Rules are reached, then *PDCN* cannot be ambiguous or invalid.

Conformance Rules

1) Without Feature B030, "Enhanced dynamic SQL", conforming SQL language shall not contain a <preparable dynamic update statement: positioned>.

20.28 <pipe row statement>

Function

Output a row from a polymorphic table function.

Format

```
<pipe row statement> ::=
  PIPE ROW <PTF descriptor name>
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) If <pipe row statement> is not executed during the execution of a PTF start component procedure, PTF fulfill component procedure, or PTF finish component procedure, then an exception condition is raised: *dynamic SQL error — PIPE ROW not during PTF execution (0700H)*.

2) Let *PTFR* be the current innermost executing PTF start component procedure, PTF fulfill component procedure, or PTF finish component procedure.

3) Let *VP* be the virtual processor that is executing *PTFR*.

4) Let *PTFDT* be the <PTF derived table> that is invoking *PTFR*, let *PTF* be the polymorphic table function that is the subject routine of *PTFDT*, and let *PTFDA* be the PTF data area of *PTFDT* on *VP*.

5) Let *PDN* be the <PTF descriptor name>. Let *N* be the value of the <simple value specification> simply contained in *PDN*.

6) If *N* does not identify the intermediate result row type descriptor area in *PTFDA*, then an exception condition is raised: *invalid SQL descriptor name (33000)*.

7) Let *R* be a row whose row type is the result row type of *PTFDT*.

8) If *PTFDT* has any partitioned <table argument>s, then every column of *R* corresponding to a partitioning column is initialized to the common value of the partitioning column in *VP*.

9) Let *PDA* be the PTF descriptor area identified by *N*.

10) Let *m* be the number of SQL descriptor item areas in *PDA*. Let I_1, ..., I_m be an enumeration of the SQL item descriptor areas.

11) For all *i*, 1 (one) ≤ *i* ≤ *m*, if the LEVEL component of I_i is 0 (zero), then

    Case:

    a) If I_i describes a proper result column of *PTF*, then the value of the DATA component of I_i is copied from I_i to the corresponding column of *R*.

b) If I_i is a pass-through output surrogate column, then let *GTA* be the <table argument> of *PTFDT* such that I_i is the pass-through output surrogate column of *GTA*. Let *T* be the table of *GTA*. Let *D* be the value of the DATA component of I_i.

Case:

i) If *D* is the null value, then every column of *R* corresponding to a non-partitioning column of *T* is set to the null value.

ii) If *D* does not represent a row of *T*, then an exception condition is raised: *dynamic SQL error — invalid pass-through surrogate value (0700G)*.

iii) Otherwise, let *SR* be the row of *T* that is represented by *D*. Every non-partitioning column of *SR* is copied to the corresponding column of *R*.

12) *R* is inserted in the partial result table of *VP*.

Conformance Rules

1) Without Feature B209, "PTF extended names", conforming SQL language shall not contain <pipe row statement>.

21 Embedded SQL

This Clause is modified by Clause 17, "Embedded SQL", in ISO/IEC 9075-4.
This Clause is modified by Clause 16, "Embedded SQL", in ISO/IEC 9075-9.
This Clause is modified by Clause 11, "Embedded SQL", in ISO/IEC 9075-10.
This Clause is modified by Clause 18, "Embedded SQL", in ISO/IEC 9075-14.
This Clause is modified by Clause 16, "Embedded SQL", in ISO/IEC 9075-15.

21.1 <embedded SQL host program>

This Subclause is modified by Subclause 17.1, "<embedded SQL host program>", in ISO/IEC 9075-4.
This Subclause is modified by Subclause 11.1, "<embedded SQL host program>", in ISO/IEC 9075-10.
This Subclause is modified by Subclause 18.1, "<embedded SQL host program>", in ISO/IEC 9075-14.

Function

Specify an <embedded SQL host program>.

Format

```
10 <embedded SQL host program> ::=
     <embedded SQL Ada program>
   | <embedded SQL C program>
   | <embedded SQL COBOL program>
   | <embedded SQL Fortran program>
   | <embedded SQL MUMPS program>
   | <embedded SQL Pascal program>
   | <embedded SQL PL/I program>

<embedded SQL statement> ::=
  <SQL prefix> <statement or declaration> [ <SQL terminator> ]

10 <statement or declaration> ::=
     <declare cursor>
   | <dynamic declare cursor>
   | <temporary table declaration>
   | <embedded authorization declaration>
   | <embedded path specification>
   | <embedded transform group specification>
   | <embedded collation specification>
   | <embedded exception declaration>
   | <SQL procedure statement>

10 <SQL prefix> ::=
     EXEC SQL
   | <ampersand> SQL <left paren>

<SQL terminator> ::=
     END-EXEC
   | <semicolon>
   | <right paren>

<embedded authorization declaration> ::=
  DECLARE <embedded authorization clause>
```

```
<embedded authorization clause> ::=
    SCHEMA <schema name>
  | AUTHORIZATION <embedded authorization identifier>
        [ FOR STATIC { ONLY | AND DYNAMIC } ]
  | SCHEMA <schema name> AUTHORIZATION <embedded authorization identifier>
        [ FOR STATIC { ONLY | AND DYNAMIC } ]

<embedded authorization identifier> ::=
  <module authorization identifier>

<embedded path specification> ::=
  <path specification>

<embedded transform group specification> ::=
  <transform group specification>

<embedded collation specification> ::=
  <module collations>

<embedded SQL declare section> ::=
    <embedded SQL begin declare>
        [ <embedded character set declaration> ]
        [ <host variable definition>... ]
        <embedded SQL end declare>
  | <embedded SQL MUMPS declare>

<embedded character set declaration> ::=
  SQL NAMES ARE <character set specification>

<embedded SQL begin declare> ::=
  <SQL prefix> BEGIN DECLARE SECTION [ <SQL terminator> ]

<embedded SQL end declare> ::=
  <SQL prefix> END DECLARE SECTION [ <SQL terminator> ]

<embedded SQL MUMPS declare> ::=
  <SQL prefix>
      BEGIN DECLARE SECTION
      [ <embedded character set declaration> ]
      [ <host variable definition>... ]
      END DECLARE SECTION
      <SQL terminator>

<host variable definition> ::=
    <Ada variable definition>
  | <C variable definition>
  | <COBOL variable definition>
  | <Fortran variable definition>
  | <MUMPS variable definition>
  | <Pascal variable definition>
  | <PL/I variable definition>

10 <embedded variable name> ::=
  <colon> <host identifier>

<host identifier> ::=
    <Ada host identifier>
  | <C host identifier>
  | <COBOL host identifier>
  | <Fortran host identifier>
  | <MUMPS host identifier>
  | <Pascal host identifier>
  | <PL/I host identifier>
```

Syntax Rules

1) An <embedded SQL host program> is a compilation unit that consists of programming language text and SQL text. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s, as defined in this document. The programming language text shall conform to the requirements of a specific programming language, the *host language*. When <embedded SQL Ada program>, <embedded SQL C program>, <embedded SQL COBOL program>, <embedded SQL Fortran program>, <embedded SQL MUMPS program>, <embedded SQL Pascal program>, or <embedded SQL PL/I program> is specified, the host language is Ada, C, COBOL, Fortran, M, Pascal, or PL/I, respectively.

   NOTE 774 — "Compilation unit" is defined in Subclause 4.30, "SQL-client modules".

2) Case:

   a) If an <embedded SQL statement> or <embedded SQL MUMPS declare> is contained in an <embedded SQL MUMPS program>, then it shall contain an <SQL prefix> that is "<ampersand>SQL<left paren>". There shall be no <separator> between the <ampersand> and "SQL" nor between "SQL" and the <left paren>.

   b) [10] If an <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> is not contained in an <embedded SQL MUMPS program>, then it shall contain an <SQL prefix> that is "EXEC SQL".

3) [10] Case:

   a) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> contained in an <embedded SQL COBOL program> shall contain an <SQL terminator> that is END-EXEC.

   b) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> contained in an <embedded SQL Fortran program> shall not contain an <SQL terminator>.

   c) [10] An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> contained in an <embedded SQL Ada program>, <embedded SQL C program>, <embedded SQL Pascal program>, or <embedded SQL PL/I program> shall contain an <SQL terminator> that is a <semicolon>.

   d) An <embedded SQL statement> or <embedded SQL MUMPS declare> that is contained in an <embedded SQL MUMPS program> shall contain an <SQL terminator> that is a <right paren>.

4) Case:

   a) An <embedded SQL declare section> that is contained in an <embedded SQL MUMPS program> shall be an <embedded SQL MUMPS declare>.

   b) An <embedded SQL declare section> that is not contained in an <embedded SQL MUMPS program> shall not be an <embedded SQL MUMPS declare>.

5) The <token>s comprising an <SQL prefix>, <embedded SQL begin declare>, or <embedded SQL end declare> shall be separated by <space> characters and shall be specified on one line. Otherwise, the rules for the continuation of lines and tokens from one line to the next and for the placement of host language comments are those of the programming language of the containing <embedded SQL host program>.

6) If an <embedded authorization declaration> appears in an <embedded SQL host program>, then it shall be contained in the first <embedded SQL statement> of that <embedded SQL host program>.

7) An <embedded SQL host program> shall not contain more than one <embedded path specification>.

8) An <embedded SQL host program> shall not contain more than one <embedded transform group specification>.

9) An <embedded SQL host program> shall not contain more than one <embedded collation specification>.

10) Case:

a) If <embedded transform group specification> is not specified, then an <embedded transform group specification> containing a <multiple group specification> with a <group specification> *GS* for each <host variable definition> that has an associated user-defined type *UDT*, but is not a user-defined type locator variable is implicit. The <group name> of *GS* is implementation-defined (ID177) and its <path-resolved user-defined type name> is the <user-defined type name> of *UDT*.

b) If <embedded transform group specification> contains a <single group specification> with a <group name> *GN*, then an <embedded transform group specification> containing a <multiple group specification> with a <group specification> *GS* for each <host variable definition> that has an associated user-defined type *UDT*, but is not a user-defined type locator variable is implicit. The <group name> of *GS* is *GN* and its <path-resolved user-defined type name> is the <user-defined type name> of *UDT*.

c) If <embedded transform group specification> contains a <multiple group specification> *MGS*, then an <embedded transform group specification> containing a <multiple group specification> that contains *MGS* extended with a <group specification> *GS* for each <host variable definition> that has an associated user-defined type *UDT*, but is not a user-defined type locator variable and no equivalent of *UDT* is contained in any <group specification> contained in *MGS* is implicit. The <group name> of *GS* is implementation-defined (ID177) and its <path-resolved user-defined type name> is the <user-defined type name> of *UDT*.

11) In the text of the <embedded SQL host program>, the implicit or explicit <embedded transform group specification> shall precede every <host variable definition>.

12) An <embedded SQL host program> shall contain no more than one <embedded character set declaration>. If an <embedded character set declaration> is not specified, then an <embedded character set declaration> that specifies an implementation-defined (ID235) character set that contains at least every character that is in <SQL language character> is implicit.

13) A <temporary table declaration> that is contained in an <embedded SQL host program> shall precede in the text of that <embedded SQL host program> any SQL-statement or <declare cursor> that references the <table name> of the <temporary table declaration>.

14) A <declare cursor> that is contained in an <embedded SQL host program> shall precede in the text of that <embedded SQL host program> any SQL-statement that references the <cursor name> of the <declare cursor>.

15) A <dynamic declare cursor> that is contained in an <embedded SQL host program> shall precede in the text of that <embedded SQL host program> any SQL-statement that references the <cursor name> of the <dynamic declare cursor>.

16) [10] Any <host identifier> that is contained in an <embedded SQL statement> in an <embedded SQL host program> shall be defined in exactly one <host variable definition> contained in that <embedded SQL host program>. In programming languages that support <host variable definition>s in subprograms, two <host variable definition>s with different, non-overlapping scope in the host language are to be regarded as defining different host variables, even if they specify the same variable name. That <host variable definition> shall appear in the text of the <embedded SQL host program> prior to any <embedded SQL statement> that references the <host identifier>. The <host variable definition> shall be such that a host language reference to the <host identifier> is valid at every <embedded SQL statement> that contains the <host identifier>.

17) The *operative embedded language Subclause* is

Case:

a) If the host language is Ada, then Subclause 21.3, "<embedded SQL Ada program>".

b) If the host language is C, then Subclause 21.4, "<embedded SQL C program>".

c) If the host language is COBOL, then Subclause 21.5, "<embedded SQL COBOL program>".

d) If the host language is Fortran, then Subclause 21.6, "<embedded SQL Fortran program>".

e) If the host language is M, then Subclause 21.7, "<embedded SQL MUMPS program>".

f) If the host language is Pascal, then Subclause 21.8, "<embedded SQL Pascal program>".

g) If the host language is PL/I, then Subclause 21.9, "<embedded SQL PL/I program>".

18) 10 A <host variable definition> defines the host language data type of the <host identifier> and the equivalent <host parameter data type>, as specified in the operative embedded language Subclause.

19) 10 An <embedded SQL host program> shall contain a <host variable definition> that specifies SQLSTATE as the <host identifier>.

NOTE 775 — The host language data type of the SQLSTATE host variable must satisfy the Syntax Rules of Subclause 13.3, "<externally-invoked procedure>".

20) If one or more <host variable definition>s that specify SQLSTATE appear in an <embedded SQL host program>, then the <host variable definition>s shall be such that a host language reference to SQLSTATE is valid at every <embedded SQL statement>, including <embedded SQL statement>s that appear in any subprograms contained in that <embedded SQL host program>. The first such <host variable definition> of SQLSTATE shall appear in the text of the <embedded SQL host program> prior to any <embedded SQL statement>.

21) 10 Given an <embedded SQL host program> *H*, there is an implied standard-conforming <SQL-client module definition> *M* and an implied host program *P* derived from *H*. The derivation of the implied program *P* and the implied <SQL-client module definition> *M* of an <embedded SQL host program> *H* effectively precedes the processing of any host language program text manipulation commands such as inclusion or copying of text.

Before *H* can be executed, *M* is processed by an implementation-defined (IW210) mechanism to produce an SQL-client module. An SQL-implementation may combine this mechanism with the processing of the <embedded SQL host program>, in which the existence of *M* is purely hypothetical.

Given an <embedded SQL host program> *H* with an implied <SQL-client module definition> *M* and an implied program *P* defined as above:

a) The implied <SQL-client module definition> *M* of *H* shall be a standard-conforming <SQL-client module definition>.

b) The implied program *P* shall conform to the specification of the host language.

22) 10 *M* is derived from *H* as follows:

a) *M* contains a <module name clause> whose <SQL-client module name> is either implementation-dependent (UV118) or is omitted.

b) *M* contains a <module character set specification> that is identical to the explicit or implicit <embedded character set declaration> with the keyword "SQL" removed.

c) *M* contains a <language clause> that specifies either ADA, C, COBOL, FORTRAN, M, PASCAL, or PLI, where *H* is respectively an <embedded SQL Ada program>, an <embedded SQL C program>, an <embedded SQL COBOL program>, an <embedded SQL Fortran program>, an

<embedded SQL MUMPS program>, an <embedded SQL Pascal program>, or an <embedded SQL PL/I program>.

d) Case:

i) If *H* contains an <embedded authorization declaration> *EAD*, then let *EAC* be the <embedded authorization clause>contained in *EAD*; *M* contains a <module authorization clause> that specifies *EAC*.

ii) Otherwise, let *SN* be an implementation-defined (ID234) <schema name>; *M* contains a <module authorization clause> that specifies "SCHEMA *SN*".

e) Case:

i) If *H* contains an <embedded path specification> *EPS*, then *M* contains the <module path specification> *EPS*.

ii) Otherwise, *M* contains an implementation-defined (ID233) <module path specification>.

f) *M* contains a <module transform group specification> that is identical to the explicit or implicit <embedded transform group specification>.

g) If an <embedded collation specification> *ECS* is specified, then *M* contains a <module collations> that is identical to the <module collations> contained in *ECS*.

h) For every <declare cursor> *EC* contained in *H*, *M* contains one <declare cursor> *PC* and one <externally-invoked procedure> *PS* that contains an <open statement> that references *PC*.

i) The <procedure name> of *PS* is implementation-dependent (UV119). *PS* contains a <host parameter declaration> *PD* for each distinct <embedded variable name> *EVN* contained in *PC* with an implementation-dependent (UV120) <host parameter name> *PN* and the <host parameter data type> *PT*, determined by the Syntax Rules of the operative embedded language Subclause.

ii) *PS* contains a <host parameter declaration> that specifies SQLSTATE. The order of <host parameter declaration>s in *PS* is implementation-dependent (US051). *PC* is a copy of *EC* in which each *EVN* has been replaced as follows.

Case:

1) If *EVN* does not identify user-defined type locator variable, but *EVN* identifies a host variable that has an associated user-defined type *UT*, then:

A) Let *GN* be the <group name> corresponding to the <user-defined type name> of *UT* contained in <group specification> contained in <embedded transform group specification>.

B) The Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with *DT* as *TYPE* and *GN* as *GROUP*; let *TSF* be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules.

C) Let *TPT* be the declared type of the single SQL parameter of *TSF*. *PT* shall be assignable to *TPT*.

D) *EVN* is replaced by:

```
TSFN(CAST(PN AS TPT))
```

2) [14]Otherwise, *EVN* is replaced by:

```
PN
```

i) For every <dynamic declare cursor> *EC* in *H*, *M* contains one <dynamic declare cursor> *PC* that is a copy of *EC*.

j) *M* contains one <temporary table declaration> for each <temporary table declaration> contained in *H*. Each <temporary table declaration> of *M* is a copy of the corresponding <temporary table declaration> of *H*.

k) 04 *M* contains one <embedded exception declaration> for each <embedded exception declaration> contained in *H*. Each <embedded exception declaration> of *M* is a copy of the corresponding <embedded exception declaration> of *H*.

l) 04 *M* contains an <externally-invoked procedure> for each <SQL procedure statement> contained in *H*. The <externally-invoked procedure> *PS* of *M* corresponding with an <SQL procedure statement> *ES* of *H* is defined as follows.

Case:

i) If *ES* is not an <open statement>, then:

1) The <procedure name> of *PS* is implementation-dependent (UV119).

2) Let *n* be the number of distinct <embedded variable name>s contained in *ES*. Let HVN_i, 1 (one) ≤ *i* ≤ *n*, be the *i*-th such <embedded variable name> and let HV_i be the host variable identified by HVN_i.

3) For each HVN_i, 1 (one) ≤ *i* ≤ *n*, *PS* contains a <host parameter declaration> PD_i defining a host parameter P_i such that:

A) The <host parameter name> PN_i of PD_i is implementation-dependent (UV120).

B) The <host parameter data type> PT_i of PD_i is determined by the Syntax Rules of the operative embedded language Subclause.

4) *PS* contains a <host parameter declaration> that specifies SQLSTATE.

5) The order of the <host parameter declaration>s PD_i, 1 (one) ≤ *i* ≤ *n*, is implementation-dependent (US051).

6) For each HVN_i, 1 (one) ≤ *i* ≤ *n*, that identifies some HV_i that has an associated user-defined type, but is not a user-defined type locator variable, the Syntax Rules of Subclause 9.8, "Host parameter mode determination", are applied with PD_i corresponding to HVN_i as *HOST PARAM DECL* and *ES* as *SQL PROC STMT*; let $PMODE_i$ be the *PARAMETER MODE* returned from the application of those Syntax Rules to determine whether the corresponding P_i is an input host parameter, an output host parameter, or both an input host parameter and an output host parameter.

A) Among P_i, 1 (one) ≤ *i* ≤ *n*, let *a* be the number of host parameters, where $PMODE_i$ is equal to "IN", *b* be the number of host parameters, where $PMODE_i$ is equal to "OUT", and let *c* be the number of host parameters, where $PMODE_i$ is equal to "INOUT".

B) Among P_i, 1 (one) ≤ *i* ≤ *n*, let PI_j, 1 (one) ≤ *j* ≤ *a*, be the host parameters, where $PMODE_i$ is equal to "IN", let PO_k, 1 (one) ≤ *k* ≤ *b*, be the host parameters, where $PMODE_i$ is equal to "OUT", and let PIO_l, 1 (one) ≤ *l* ≤ *c*, be the host parameters, where $PMODE_i$ is equal to "INOUT".

C) Let PNI_j, 1 (one) ≤ j ≤ a, be the <host parameter name> of PI_j. Let PNO_k, 1 (one) ≤ k ≤ b, be the <host parameter name> of PO_k. Let $PNIO_l$, 1 (one) ≤ l ≤ c, be the <host parameter name> of PIO_l.

D) Let HVI_j, 1 (one) ≤ j ≤ a, be the host variable corresponding to PI_j. Let HVO_k, 1 (one) ≤ k ≤ b, be the host variable corresponding to PO_k. Let $HVIO_l$, 1 (one) ≤ l ≤ c, be the host variable corresponding to PIO_l.

E) Let TSI_j, 1 (one) ≤ j ≤ a, be the associated SQL data type of HVI_j. Let TSO_k, 1 (one) ≤ k ≤ b, be the associated SQL data type of HVO_k. Let $TSIO_l$, 1 (one) ≤ l ≤ c, be the associated SQL data type of $HVIO_l$.

F) Let TUI_j, 1 (one) ≤ j ≤ a, be the associated user-defined type of HVI_j. Let TUO_k, 1 (one) ≤ k ≤ b, be the associated user-defined type of HVO_k. Let $TUIO_l$, 1 (one) ≤ l ≤ c, be the associated user-defined type of $HVIO_l$.

G) Let GNI_j, 1 (one) ≤ j ≤ a, be the <group name> corresponding to the <user-defined type name> of TUI_j contained in the <group specification> contained in <embedded transform group specification>. Let GNO_k, 1 (one) ≤ k ≤ b, be the <group name> corresponding to the <user-defined type name> of TUO_k contained in the <group specification> contained in <embedded transform group specification>. Let $GNIO_l$, 1 (one) ≤ l ≤ c, be the <group name> corresponding to the <user-defined type name> of $TUIO_l$ contained in the <group specification> contained in <embedded transform group specification>.

H) For every j, 1 (one) ≤ j ≤ a, the Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with TUI_j as *TYPE* and GNI_j as *GROUP*; let $TSFI_j$ be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable to-sql function $TSFI_j$ identified by <routine name> $TSIN_j$. Let TTI_j be the data type of the single SQL parameter of $TSFI_j$. TSI_j shall be assignable to TTI_j.

I) For every l, 1 (one) ≤ l ≤ c, the Syntax Rules of Subclause 9.33, "Determination of a to-sql function", are applied with $TUIO_l$ as *TYPE* and $GNIO_l$ as *GROUP*; let $TSFIO_l$ be the *TO-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable to-sql function $TSFIO_l$ identified by <routine name> $TSION_l$. Let $TTIO_l$ be the data type of the single SQL parameter of $TSFIO_l$. $TSIO_l$ shall be assignable to $TTIO_l$.

J) For every k, 1 (one) ≤ k ≤ b, the Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with TUO_k as *TYPE* and GNO_k as *GROUP*; let $FSFO_k$ be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable from-sql function $FSFO_k$ identified by <routine name> $FSON_k$. Let TRO_k be the result data type of $FSFO_k$. TSO_k shall be assignable to TRO_k.

K) For every l, 1 (one) ≤ l ≤ c, the Syntax Rules of Subclause 9.31, "Determination of a from-sql function", are applied with $TUIO_l$ as *TYPE* and $GNIO_l$ as *GROUP*; let $FSFIO_l$ be the *FROM-SQL FUNCTION* returned from the application of those Syntax Rules. There shall be an applicable from-sql function $FSFIO_l$

identified by <routine name> $FSION_l$. Let $TRIO_l$ be the result data type of $FSFIO_l$. $TSIO_l$ shall be assignable to $TRIO_l$.

L) Let SVI_j, 1 (one) ≤ j ≤ a, SVO_k, 1 (one) ≤ k ≤ b, and $SVIO_l$, 1 (one) ≤ l ≤ c, be implementation-defined <SQL variable name>s, each of which is not equivalent to any other <SQL variable name> contained in *ES*, to any <SQL parameter name> contained in *ES*, or to any <column name> contained in *ES*.

7) [14]Let *NES* be an <SQL procedure statement> that is a copy of *ES* in which every HVN_i, 1 (one) ≤ i ≤ n, is replaced as follows.

Case:

A) If HV_i has an associated user-defined type but is not a user-defined type locator variable, then

Case:

I) If $PMODE_i$ is equal to "IN", then let PI_j, 1 (one) ≤ j ≤ a, be the host parameter that corresponds to P_i; HVN_i is replaced by SVI_j.

II) If $PMODE_i$ is equal to "OUT", then let PO_k, 1 (one) ≤ k ≤ b, be the host parameter that corresponds to P_i; HVN_i is replaced by SVO_k.

III) Otherwise, let PIO_l, 1 (one) ≤ l ≤ c, be the host parameter that corresponds to P_i; HVN_i is replaced by $SVIO_l$.

B) [14]Otherwise, HVN_i is replaced by PN_i.

8) [14]The <SQL procedure statement> of *PS* is:

```
BEGIN ATOMIC
    DECLARE SVI1 TUI1;
    ...
    DECLARE SVIa TUIa;
    DECLARE SVO1 TUO1;
    ...
    DECLARE SVOb TUOb;
    DECLARE SVIO1 TUIO1;
    ...
    DECLARE SVIOc TUIOc;
    SET SVI1 = TSIN1 (CAST (PNI1 AS TTI1));
    ...
    SET SVIa = TSINa (CAST (PNIa AS TTIa));
    SET SVIO1 = TSION1 (CAST (PNIO1 AS TTIO1));
    ...
    SET SVIOc = TSIONc (CAST (PNIOc AS TTIOc));
    NES;
    SET PNO1 = CAST ( FSON1 (SVO1) AS TSO1);
    ...
    SET PNOb = CAST ( FSONb (SVOb) AS TSOb);
    SET PNIO1 = CAST ( FSION1 (SVIO1) AS TSIO1);
    ...
    SET PNIOc = CAST ( FSIONc (SVIOc) AS TSIOc);
END;
```

9) Whether one <externally-invoked procedure> of *M* can correspond to more than one <SQL procedure statement> of *H* is implementation-dependent (UA071).

ii) If *ES* is an <open statement>, then:

1) Let *EC* be the <declare cursor> in *H* referenced by *ES*.

2) *PS* is the <externally-invoked procedure> in *M* that contains an <open statement> that references the <declare cursor> in *M* corresponding to *EC*.

23) 04 10 *P* is derived from *H* as follows:

a) Each <embedded SQL begin declare>, <embedded SQL end declare>, and <embedded character set declaration> has been deleted. If the embedded host language is M, then each <embedded SQL MUMPS declare> has been deleted.

b) Each <host variable definition> in an <embedded SQL declare section> has been replaced by a valid data definition in the target host language according to the Syntax Rules of the operative embedded language Subclause.

c) 04 Each <embedded SQL statement> that contains a <declare cursor>, a <dynamic declare cursor>, an <SQL-invoked routine>, or a <temporary table declaration> has been deleted, and every <embedded SQL statement> that contains an <embedded exception declaration> has been replaced with statements of the host language that will have the effect specified by the General Rules of Subclause 21.2, "<embedded exception declaration>".

d) Each <embedded SQL statement> that contains an <SQL procedure statement> has been replaced by host language statements that perform the following actions:

i) A host language procedure or subroutine call of the <externally-invoked procedure> of the implied <SQL-client module definition> *M* of *H* that corresponds with the <SQL procedure statement>.

If the <SQL procedure statement> is not an <open statement>, then the arguments of the call include each distinct <host identifier> contained in the <SQL procedure statement> together with the SQLSTATE <host identifier>. If the <SQL procedure statement> is an <open statement>, then the arguments of the call include each distinct <host identifier> contained in the corresponding <declare cursor> of *H* together with the SQLSTATE <host identifier>.

The order of the arguments in the call corresponds with the order of the corresponding <host parameter declaration>s in the corresponding <externally-invoked procedure>.

NOTE 776 — In an <embedded SQL Fortran program>, the "SQLSTATE" variable can be abbreviated to "SQLSTA". See the Syntax Rules of Subclause 21.6, "<embedded SQL Fortran program>".

ii) Exception actions, as specified in Subclause 21.2, "<embedded exception declaration>".

e) Each <statement or declaration> that contains an <embedded authorization declaration> is deleted.

Access Rules

1) For every host variable whose <embedded variable name> is contained in <statement or declaration> and has an associated user-defined type, the current privileges shall include EXECUTE privilege on all from-sql functions (if any) and all to-sql functions (if any) referenced in the corresponding SQL-client module.

General Rules

1) The interpretation of an <embedded SQL host program> *H* is defined to be equivalent to the interpretation of the implied program *P* of *H* and the implied <SQL-client module definition> *M* of *H*.