International Standard

ISO/IEC 19075-10

First edition
2024-10

Information technology — Guidance for the use of database language SQL —

Part 10: SQL model (Guide/Model)

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

Reference number
ISO/IEC 19075-10:2024(en)

© ISO/IEC 2024

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Tables

Table

Page

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Figures

Figure

Page

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Examples

Example

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

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 https://www.iso.org/directives or https://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 had 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 https://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 https://www.iso.org/iso/foreword.html. In the IEC, see https://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 document is intended to be used in conjunction with the following editions of the parts of the ISO/IEC 9075 series:

— ISO/IEC 9075-1, sixth edition or later;

— ISO/IEC 9075-2, sixth edition or later;

— ISO/IEC 9075-3, sixth edition or later;

— ISO/IEC 9075-4, seventh edition or later;

— ISO/IEC 9075-9, fifth edition or later;

— ISO/IEC 9075-10, fifth edition or later;

— ISO/IEC 9075-11, fifth edition or later;

— ISO/IEC 9075-13, fifth edition or later;

— ISO/IEC 9075-14, sixth edition or later;

— ISO/IEC 9075-15, second edition or later;

— ISO/IEC 9075-16, first edition or later.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

A list of all parts in the ISO/IEC 19075 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 https://www.iso.org/members.html and https://www.iec.ch/national-committees.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Introduction

This document describes the model of database language SQL, as defined primarily in ISO/IEC 9075-1 and ISO/IEC 9075-2. It supplies some background in the form of a brief history of the relational data model, plus a brief overview of some SQL features relevant to the SQL model.

The SQL model, of course, existed from the initial definition of the language, but was not explicitly recognized until the language became sufficiently complex that appropriate vocabulary and definitions were required to accurately specify all capabilities required.

This document summarizes the result of hundreds of hours of discussion and analysis that were required in order to create a coherent and useful definition of the SQL model of operation. This document briefly outlines the Relational model popularized by E.F. Codd. However, the fundamental structure and content of this document addresses SQL language facilities such as tables, column, relationships, and operations, not the Relational model itself.

Since SQL's adoption in 1986 as a database language standard, many extensions have been proposed, defined, reviewed, modified, and adopted by members of SQL-related standards committees across the world. While the SQL model herein addresses only a small subset of the SQL language's data management facilities, SQL's overall scope includes non-relational data types, object classes, multi-level data structures, and even DBMS-generated pointers to rows that appear as if they were user-based values. Supporting all all these SQL capabilities, there exists extensive and formally-defined language constructs that (when correctly implemented by SQL database implementers) provide a data definition and processing environment that is reliable and repeatable across all SQL standard-adherent implementations.

NOTE 1 — In this document, the terms "SQL" and "SQL language" are used as synonyms. The terms "SQL standard", "ISO/IEC 9075", and "the ISO/IEC 9075 series" are used as synonyms and refer to the standard for SQL [language], which is the ISO/IEC 9075 series of standards.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

Information technology — Guidance for the use of database language SQL —

Part 10:
SQL model (Guide/Model)

1 Scope

This document describes the model of database language SQL as defined in ISO/IEC 9075-1, ISO/IEC 9075-2, and ISO/IEC 9075-11. The meanings of and the relationships between various concepts of that model are described in text and illustrated graphically. Background in the form of some historical review and a brief overview of key SQL features is included.

NOTE 2 — In spite of the fact that the names of the ISO/IEC 9075 series of standards contain the phrase "database language", the standards do not use the word "database" to describe the *thing* that SQL creates and on which it operates. The word "database" is used in many different contexts and has meanings wholly unrelated to the intent of the ISO/IEC 9075 series. Consequently, a variety of other terms are defined and used by the ISO/IEC 9075 series. The word "database" is frequently used in this document informally to mean "a collection of data managed by an SQL-implementation at any given time".

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

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.

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

ISO/IEC 9075-2, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9075-1, ISO/IEC 9075-2, and ISO/IEC 9075-11 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/

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

4 SQL model: origin and components

4.1 Relational model

In 1970, Dr. E.F. Codd published Relational Model [32]. Dr. Codd's premise was that "users of large data banks must be protected from having to know how the data is organized in the machine". The computer industry had years of experience with database management systems such as:

— IBM's Information Management System (IMS)™ that used a hierarchical model to organize data;

— The Conference/Committee on Data Systems Languages and its CODASYL data model that managed relationships between data records using chains of "pointers" (often called a "network database system");

— The American National Standard ANSI X3.133:1986 [2] and its International Standard companion ISO 8907:1987 [1], which was a standardized specification of a CODASYL-like language;

— Cincom System, Inc.'s TOTAL™, which was a database management system that combined hierarchical and network data management techniques.

Such systems offered high-performance access to data, particularly data that was closely related to data currently accessed, provided there were predefined, relevant links (or pointers) to that related data. Unfortunately, those links were *built into* the data as stored in the database system and it was not possible to easily reorganize them.

Codd's belief was that it was not necessary for application programs and other users to have intimate knowledge of the manner in which data was organized in persistent storage, nor to suffer the performance problems caused by a change in application requirements that rendered existing data relationships unhelpful.

The *relational model of data*, as Codd's work is universally known, separated the logical organization of data from its physical organization and used straightforward mathematical definitions to identify relationships among data on demand by application requirements.

The relational model defines collections of closely-related data as "relations" (sets of tuples); each "tuple" (set of elements) is therefore unique within its containing relation. Every relation is described by a metadata structure called a "relation scheme", which is a set of "attributes". Each attribute is identified by a unique name or by its ordinal position within the scheme, and has a datatype. In the relational model, each element of a tuple is mapped to a scheme attribute, and every tuple has the same number of elements as the scheme has attributes. Each element is therefore an "attribute value" that is defined by the data type of its corresponding attribute.

The mathematical underpinnings of the relational model are based on the definition of a handful of operations. There are two broad categories of these operations, commonly called "relational algebra" and "relational calculus".

The relational algebra specifies a number of simple operators, each of which transforms one or more source relations to a result relation. The operators include:

— *selection*, which transforms a source relation into a result relation by eliminating tuples that do not satisfy a specified predicate;

— *projection*, which transforms the source relation into a result relation by eliminating one or more attributes (more precisely, the result relation incorporates only the attributes named by the projection operator);

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

— *join* (also known as "Cartesian product"), which combines two relations into a result relation by combining every tuple in one relation with every tuple in the other relation.

There are numerous variations of join operation. Some of those variations are discussed briefly in Subclause 6.3, "Query expressions and query specifications".

Most resources, including Codd's original paper, add a few more basic operators to those already listed. They are:

— *rename*, which transforms the source relation into a result relation by changing the names of one or more attributes to a specified name;

— *set union*, which combines two relations into a result relation by copying all tuples that are found in either or both relations;

— *set difference*, which combines two relations into a result relation by copying all tuples of one relation *except* those tuples that are found in the other relation.

Both set union and set difference require that the two relations be "union compatible", meaning that they must have the same number of attributes and the names of the respective attributes must be the same. (Unstated, but implicit, the respective attributes must also be of the same or a compatible, data type.)

Relational algebra has a kind of procedural aspect to it. Most application operations on a relational system that use a relational algebraic approach will first combine relations (joins, unions, differences) to create a new relation, then filter that new relation to eliminate tuples that fail to satisfy one or more predicates to create yet another new relation, then project that relation onto the desired set of attributes and create a final, result relation.

Relational calculus, by contrast, provides the ability for applications to express the desired result in a non-procedural manner, allowing the underlying software to determine the most effective (and, presumably, efficient) mechanisms to achieve that result.

Codd proved that relational algebra and relational calculus are logically equivalent and one can be transformed into the other. The primary justification for the ostensibly less-precise structure of relational calculus is to promote automatic *optimization* of database operations. That is, a database system, when presented with a relational calculus expression, can transform it into many possible equivalent relational algebra expressions based on minimizing data access times, numbers of operations, and so forth.

4.2 The SQL model

SQL is based on the relational model. However, SQL uses different terminology for its basic data structures, for at least two reasons: first, those basic data structures have minor but important differences from their relational model analogs; second, it was felt that the users of the language would prefer a less "technical-sounding" nomenclature than the mathematical terminology used in the relational model.

SQL's analog of the relational model's "relation" is called a *table*. A table is made up of *column*s and *row*s, corresponding respectively to the relational model's attributes and tuples. Figure 1, "Illustration of an SQL table", illustrates an SQL table.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

EMPLOYEES					
EMP_ID	**EMP_NAME**	**DEPT_ID**	**CATEGORY**	**AMOUNT**	**HIRE_DATE**
E00147	John Philips	ENG	S	185300.00	2012-08-25
S00023	Stefan Schultz	MFG	H	23.00	2020-04-13
M00052	Marie Lacroix	MGT	S	224000.00	2008-11-10
F01598	Bill Hanson	ENG	S	106500.00	2014-04-30
…	…	…	…	…	…
…	…	…	…	…	…
…	…	…	…	…	…
J00215	Willem Peshma	MNT	H	16.25	2021-02-15
D01284	Mike Johnston	DES	S	96000.00	2018-06-17
S00007	Juan Muñez	ENG	S	88750.00	2020-01-28
E02853	Kohji Fujita	FAC	H	14.80	2019-12-02

Figure 1 — Illustration of an SQL table

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

The most important — indeed, fundamental — differences between the relational model of data and the SQL model are:

1) In the relational model, every tuple in a relation is unique; that is, no two tuples are permitted to have the same value in every respective attribute. Relations are mathematical *sets*. By contrast, SQL permits *duplicate rows*, in which any number of rows are permitted to have the same values in respective columns.

2) The relational model requires that every attribute of a tuple have a (single) value of a specified data type; if the value of an attribute for a given tuple is not yet known, unavailable, or irrelevant (consider how the "year of marriage" of an unmarried person's is recorded on a paper form), then some indication, such as a "default" value, must be provided. However, there are situations in which no such default value is sensible. Consequently, SQL defines a special kind of "value", called a *null value*, that can be assigned to columns of any data type in any row of a table[1] to represent information that is unknown for any reason.

3) Comparisons of data in the relational model always results in a known result, true or false; that is, values are either equal or not equal, greater than or less than, one another. This is a direct result of the fact that every data object has to have a known value of a specified data type. SQL accommodates its null values by defining *three-valued logic*, in which the results of a comparison can be either true or false — or unknown. In SQL, if one compares a (non-null) value with a null value, the result is always unknown. Three-valued logic sometimes leads to results that some people find counter-intuitive, but it is mathematically provable to be correct; it also offers considerable computational power for applications that use it properly.

In the relational model, there may be no sequence (or order) to the attributes of a relation, and the relation, as a set of tuples, is by definition unordered. By contrast, in the SQL model there are circumstances in which the sequence of columns in a table is important, and there are operations that will form an application-specified sequence of the rows of a table.

SQL is a language that combines aspects of both relational algebra and relational calculus.

1 In SQL, columns can be defined to prohibit null values, if desired.

© ISO/IEC 2024 – All rights reserved

As described in Clause 6, "Queries, expressions, and statements", SQL defines *query expressions* that formulate requests to the database system to identify information of interest to applications. Just as operations in the relational model accept one or more relations as sources and transform those operations to a new, result relation, SQL's operations accept one or more tables as sources and transform them to new, result tables. Query expressions contain syntactic elements that correspond to the relational algebra operations select, project, join, union, difference, and others; they also contain elements that more closely resemble relational calculus.

In SQL, *persistent base tables* are tables that are "stored" in persistent storage. SQL also provides *virtual tables* that are "computed" upon demand by the evaluation of query expressions. Persistent base tables and virtual tables are both usable as sources to relational operations, usually without significant distinction; the most important distinction between the two are that persistent base tables are known by persistent names (table names), while virtual tables are often computed, used, and discarded without ever having been given a name.

If a particular virtual table's definition is used frequently — that is, there is frequent need to compute the virtual table using an identical query expression — the definition is made persistent so that the virtual table can be referenced by name in the same manner as a persistent base table. A virtual tables with a persistent definition is called a *view*.

The SQL standard defines both the syntax and the semantics of the SQL language. The syntax is specified in a particular dialect of "extended BNF"[2] selected by the technical committees that defined the standard. The dialect of BNF used is described in ISO/IEC 9075-1.

In the SQL standard's specification of the semantics of the language, "general" rules describe an algorithm (possibly not the most efficient algorithm) that, if implemented, would produce the desired results. SQL-implementations are not required to implement that algorithm, as more efficient implementations are usually possible. The standard uses the word "effectively" to indicate that the intent of the algorithm is not to specify the actual implementation, but only the result that an implementation must achieve.

NOTE 3 — A brief history of the SQL language is provided in Annex A, "History of the SQL standard".

4.3 The name "SQL"

The name "SQL", in the context of the ISO/IEC 9075 series, is not an acronym, but is properly pronounced "ess cue ell". Some products that implement the standard pronounce that sequence of letters in their products's names "sequel"; others use the phrase "structured query language" to describe their products. The original developers of the language that eventually became the SQL standard named their language "Structured English Query Language" and planned to use the shorter name "SEQUEL", but discovered that term had already been trademarked.

4.4 Syntax conventions in SQL

The syntax of SQL, like that of most programming languages, includes "keywords", special characters ("punctuation"), and identifiers (which are the names of data objects used in the language). Some keywords are reserved, meaning that they cannot be used as the names of data objects; other keywords are not reserved and are sometimes referred to as "noise" words (meaning that their use is not necessarily required for proper analysis of the intent, but whose presence can improve the readability for users).

In SQL, all keywords (reserved or not) may be "spelled" using any combination of upper-case and lower-case characters. For example, the reserved keyword "`SELECT`" may be written as "`Select`", "`select`", or even "`sElEcT`" without affecting its meaning or the ability of the SQL implementation to determine its usage. By convention in this document, keywords are usually spelled using all upper-case characters.

An identifier in SQL is either a "regular identifier" or a "delimited identifier".

2 "BNF" is an initialism for "Backus Naur Form", named for its inventors, John Backus and Peter Naur.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

An ordinary identifier is required to start with a character that belongs to one of the Unicode character categories "Lu", "Ll", "Lt", "Lm", "Lo", or "Nl". Those categories identify characters that are, respectively, upper-case letters, lower-case letters, title-case letters, modifier letters, other letters, and "letter numbers". The remainder of an ordinary identifier is permitted to be one of the categories allowed for the initial character, as well as characters that belong to one of the additional Unicode character categories "Mn", "Mc", "Nd", and "Pc", which respectively identify characters that are non-spacing marks, spacing combining marks, decimal numbers, and connector punctuations. In the SQL standard, lower-case alphabetic characters contained in ordinary identifiers are implicitly transformed to their corresponding upper-case characters.

For example, the ordinary identifier "`street_name`" may be written as "`Street_Name`", "`Street_name`", or "`street_Name`"; all variations correctly identify the same SQL object, the canonical spelling of which is "`STREET_NAME`". All such spellings are interpreted in the SQL standard as representing the same SQL object. In this document, identifiers are often spelled using only lower-case characters, but some text uses mixed-case or only upper-case characters; the choice is usually determined by what concept the text discusses and what it is desired to emphasize.

NOTE 4 — The transformation of lower-case alphabetic characters to their corresponding upper-case characters is not necessarily a one-to-one transformation. For example, an identifier written by a German programmer that is spelled "`straßen_name`" has a canonical spelling "`STRASSEN_NAME`".

By contrast, a delimited identifier is a syntax element that begins and ends with a "double quote" character "`"`" (equivalent to Unicode character U+0022 and contains a sequence of characters (other than a double quote character); no character in that sequence is permitted to belong to any of the Unicode character categories "Cc", "Cf", "Cn", "Cs", "Zl", or "Zp". (Those categories are, respectively, control characters, formatting characters, unassigned characters, surrogates, line separators, and paragraph separators.) If the name of an SQL object is required to incorporate a double quote character, that is represented as "`""`" (two consecutive double quote characters).

NOTE 5 — In the SQL standard, identifiers must not exceed a maximum length (number of characters) that is determined by the SQL-implementation. SQL-implementations must support identifiers of at least a minimum number defined by the standard. That minimum number, in editions of the SQL standard from 1986 through 2023, is 18 characters. In future editions of the SQL standard, it is possible that the minimum length of identifiers that implementations are required to support will be changed to some other value, such as 128 characters.

The names of SQL objects (identifiers) have additional restrictions placed on them. One restriction is that certain names are "qualified" by the names of other objects. For example, the names of the columns of an SQL table are implicitly qualified by the name of the table: the `salary` column of the `employees` table is, in its qualified form, `employees.salary`. Table names are further qualified by the name of the "container" in which tables are defined, called the "schema". If the `employees` table is contained in a schema named `headquarters`, then the name of the table is effectively `headquarters.employees` and the name of the `salary` column is effectively `headquarters.employees.salary`.

NOTE 6 — In the SQL standard, schemata are specified to exist in the context of another kind of SQL object, called a "catalog". Few SQL-implementations explicitly support catalogs; in practice, that means that the catalog name cannot be specified as a qualifier for schema names (in effect, only a single, unnamed, catalog is implied).

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

5 Data integrity

5.1 Introduction to integrity

In any database, including an SQL database, it is highly likely that much of the data relates closely to other of the data. The value of that data is highly dependent on it being correct, or at least as correct as possible. When two or more pieces of data are related to one another, it is imperative that their relationships and correctness are correlated.

SQL provides a number of facilities to help ensure that data is correct, that it remains correct, and that relationships between data remain correct. Among these capabilities are the implementation of transactions and the provision of integrity constraints on data and their relationships.

5.2 Transactions

A transaction is defined in computer science to be an atomic unit of work that either succeeds entirely or fails entirely. If any operation performed as part of a transaction fails, then all previous operations performed as part of the same transaction are reversed, the failure is reported to the application, and the data is restored to the values it had before the transaction was initiated.

The semantics of the SQL standard require that all changes made to data managed by an SQL-implementation are done as part of a transaction. It also assumes that every transaction completes before another transaction is initiated. When transactions behave in this manner, they are referred to as "serialized" transactions, meaning that they are initiated and terminated in a sequential fashion, without overlap.

Database systems normally do not actually serialize transactions, but provide *serializable* transactions — transactions that behave as though they were executed sequentially, but that overlap in time. In order for transactions that overlap to behave as though they were executed sequentially, their operations must be *isolated* from those of other, concurrent transactions. There are many implementation mechanisms that can be used to provide transaction isolation, but those are beyond the scope of this document. The SQL standard permits SQL-implementations to provide varying levels of transaction isolation, ranging from fully serializable to a point where the data "seen" by one transaction can be affected by changes caused by another transaction *even if that other transaction eventually fails*. Transactions and isolation levels are discussed in greater detail in Clause 11, "Transaction model".

In order to ensure the highest level of data accuracy and consistency, application programs start transactions as serializable transactions. Applications that have lower data accuracy and consistency requirements are able to start transactions with lower levels of isolation.

5.3 Constraints

SQL provides capabilities that allow implementations to continuously validate that the data they manage satisfy defined value restrictions, including specific values or ranges of values and relationships between multiple values. Those capabilities are generically known as *integrity constraints*.

SQL defines several kinds of integrity constraint: unique constraints, including primary key constraints; referential constraints; and table check constraints. Most table check constraints are defined in the context of a single table, but SQL provides a variation, called an *assertion*, that is defined with a larger scope than a single table.

Unique constraints are used in the definition of a table to specify that the value of a specified column or group of columns of each row of the table is required to be unique among all rows of the table. Ordinary unique constraints do not prohibit values in the specified column or group of columns from being the

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

null value; all such null values are ignored for the purposes of determining unique values. SQL permits multiple unique constraints to be defined for tables, but does not require that every table have a unique constraint.

A *candidate key* is a unique constraint that identifies a column or group of columns that is defined to prohibit null values. Special syntax (PRIMARY KEY) is used to define a particular kind of candidate key that both requires the value of the designated column, or the combination of values of the designated group of columns, must be unique among all rows of the table and are also prohibited from having the null value. No table is permitted to have more than one primary key, but tables may have no, one, or multiple candidate keys.

Frequently, data in one table is required to be closely associated with data in a second table. That close association is often indicated by requiring that the value of one column or group of columns in every row of the first table be equal to the value of some column or group of columns in some row of the second table. If the second table — called the *referenced table* — has been defined to have a unique constraint comprising that column or group of columns, then the first table — called the *referencing table* — can be defined to have a *foreign key* that identifies the referenced table. See Figure 2, "Primary key — Foreign key relationships", for an illustration of this relationship.

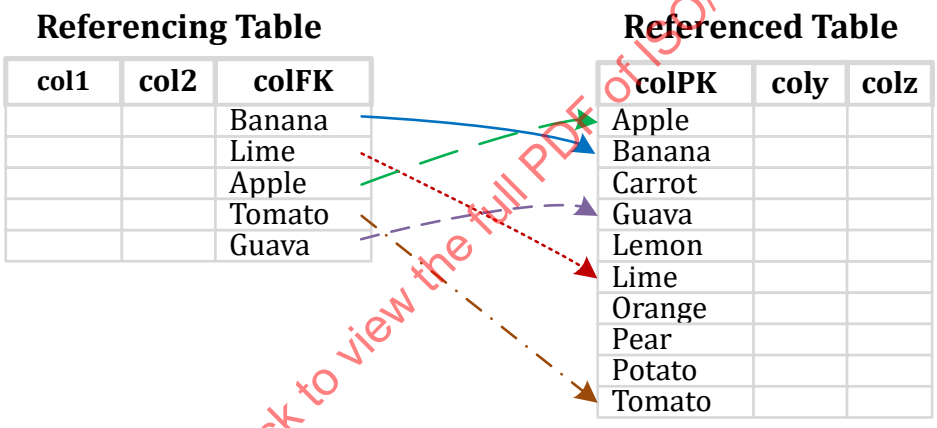

Figure 2 — Primary key — Foreign key relationships

The referenced table is sometimes informally called the "parent table" and the referencing table the "child table".

5.4 Predicates

The SQL standard provides facilities that allow applications to compare values to other values and to test a variety of situations. Those facilities are known as "predicates". A predicate is an expression that can take at most three values: *True*, *False*, and (under some conditions) *Unknown*.

In SQL, predicates are most often used in the `WHERE` clause of a query specification (see Subclause 6.3, "Query expressions and query specifications"), but are used in other places as well (such as the `FROM` clause and in constraints). The most common form of predicate is the *comparison predicate*, which allows comparison of values based on their equality or lack thereof, as well as based on one value being less than or greater than the other. Other predicates permit testing whether a value is the SQL null value or not, whether a virtual table has any rows or not, whether a specific value is unique among a collection of values, and numerous other tests.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Predicates are often combined into complex tests using logical operators such as `AND`, `OR`, and `NOT`. Such combinations are known as *search conditions* in the SQL standard. Example 1, "Search condition", provides an example of such a combination.

Example 1 — Search condition

```
employees.dept_id = departments.dept_id AND
( employee.salary > 100000 OR
  employee.salary IS NULL )
```

5.5 Functional dependencies

A *functional dependency* is a relationship between identifying columns of a table and sets of columns of that table. Such sets are concerned with "base table unique constraint sets" and "base table primary key sets". In SQL terms, it is a set of *known not null* columns of a unique constraint (of a base table), or a *primary key constraint* (of a base table), respectively.

The logic of inferring known functional dependencies is beyond the scope of this document, but described in detail in Subclause 4.26, "Functional dependencies", of ISO/IEC 9075-2:2023 [14].

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

6 Queries, expressions, and statements

6.1 Sets and multisets

Subclause 4.1, "Relational model", discusses the fact that the mathematics of the relational model is based on the definition of a relation as a set of tuples. Subclause 4.2, "The SQL model", observes that the SQL model differs from the relational model by allowing tables to contain duplicate rows. SQL terminology for a table containing rows that are allowed to duplicate other rows in the same table is *multiset* (other programming languages sometimes refer to collections that are permitted to contain duplicates as "bags"). The possibility of duplicate rows — that is, tables for which no unique or primary key constraints have been defined — complicates the mathematics, which in turn requires different definitions of the various operations defined in the relational model: select, project, join, union, and difference. The SQL standard provides detailed semantics of those operators to accommodate the multiset nature of tables.

6.2 Queries

Intuitively, a *query* is a question. The SQL standard does not define the word "query", but implies that queries are the mechanism by which information is requested from a collection of data (a "database").

SQL, of course, provides the ability to retrieve data from a database. It does so through the use of *query expressions*. It also provides the ability to add new data into a database, to modify data already in a database, and to remove data from a database. Information is added, modified, and removed through SQL-statements (see Subclause 6.7, "SQL-statements"). SQL-statements frequently use query expressions to determine precisely what information is to be added, modified, or removed.

6.3 Query expressions and query specifications

Arguably, the most central concept in the SQL language is its query expression. The SQL standard defines a BNF non-terminal symbol, <query expression>, that forms the root syntactic element of the language's query capabilities. See ISO/IEC 9075-2:2023 [14], for the syntax of query expressions and their components (particularly <query specification>s).

A concrete example, seen in Example 2, "Query specification", reveals the most important aspects of the syntax of SQL queries.

Example 2 — Query specification

```
SELECT emp_name, amount
FROM employees
WHERE dept_id = 'ENG'
```

That query instructs the SQL-implementation to retrieve from the EMPLOYEES table the rows corresponding to employees working in the (presumably) Engineering department, and then to return only the employees' names and payment amounts.

In its simplest form, the structure of a query is often stated to be: SELECT ... FROM ... WHERE ... GROUP BY ... HAVING .... The details are rather complex, particularly in the FROM clause, in which tables are joined in various ways, combined with union, except, and intersect operators, and constructed from a variety of data sources (not all of which are necessarily SQL-environments — ISO/IEC 9075-9 [17] supports SQL operations in non-SQL environments). Some of those syntactic elements form clauses that cause the *shape* (see Subclause 6.4, "Table "shapes"") of the result (or output) virtual table to be different from the shape of any source (or input) virtual table.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

NOTE 7 — In this document, the virtual table that the various relational operators use as data sources are called source tables or input tables; the virtual tables produced by the evaluation of such operators are called result tables or output tables.

Each clause has a particular purpose. The order of evaluation of those clauses is different from the syntactic order in which they are specified. The purpose of each clause, shown in "effective" evaluation order, is:

— FROM: If a single table is specified in the FROM clause, the evaluation of the clause transforms the collection of rows contained in that table to a virtual table with the same number of columns where each column has the same name as the corresponding column in the specified table. If multiple tables are specified using any syntax that specifies a join, evaluation of the clause combines the columns and rows of the tables in the manner specified by that join syntax. For example, if two tables's names are separated by a comma, the tables are joined in a Cartesian product (sometimes called a "cross product"), which means that the resulting virtual table has a column created from each column of both tables and that each row of the resulting table is created from concatenating one row from the first table with a row of the second table. Figure 3, "Cartesian product", illustrates a Cartesian product of two (rather small) tables. The number of columns of the result virtual table is the sum of the number of columns of the two source tables, and the number of rows of the result is the product of the number of rows of the two source tables.

Products

Name	Cost	Color
Apple	0.40	Red
Banana	0.35	Yellow
Lemon	0.12	Yellow
Lime	0.15	Green
Tomato	0.65	Red

Inventory

Supplier	Name	Qty
FruitCo	Banana	125
SourInc	Lime	487

```
FROM PRODUCTS, INVENTORY
```

Cross product

Name	Cost	Color	Supplier	Name	Qty
Apple	0.40	Red	FruitCo	Banana	125
Banana	0.35	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	FruitCo	Banana	125
Lime	0.15	Green	FruitCo	Banana	125
Tomato	0.65	Red	FruitCo	Banana	125
Apple	0.40	Red	SourInc	Lime	487
Banana	0.35	Yellow	SourInc	Lime	487
Lemon	0.12	Yellow	SourInc	Lime	487
Lime	0.15	Green	SourInc	Lime	487
Tomato	0.65	Red	SourInc	Lime	487

Figure 3 — Cartesian product

NOTE 8 — The FROM clause, which is required, is an example of an operation whose result table has a different shape than any of its source tables, as discussed in Subclause 6.4, "Table "shapes"".

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Cross products of two tables are rarely useful to SQL applications. Instead, applications usually require tables to be joined according to one or more criteria, such as equality of values in a column of one table to the values in a column of the second table.

For example, of an application requires information about the salaries of employees working for various departments, the code shown in Example 3, "Join based on equality" is permitted.

Example 3 — Join based on equality

```
FROM employees, departments USING dept_name
```

In Example 3, "Join based on equality", the "USING" clause informs the SQL processor that all rows in the `employees` table are to be matched to all rows in the `departments` table based on the equality of values in the `dept_name` column (that must be a column of both tables). That example is equivalent to

```
FROM employees, departments
WHERE employees.dept_name = departments.dept_name
```

There are several ways to express join operations in SQL. The syntax details are beyond the scope of this document. Interested readers are referred to ISO/IEC 9075-2.

— WHERE: The WHERE clause applies a *filter*, expressed as a predicate, to the rows of the source table, generating a result table whose rows are the rows of the source table that satisfy the predicate. (See Subclause 5.4, "Predicates", for information about predicates.)

Although SQL is based on three-valued logic (*True*, *False*, and *Unknown*, as discussed in Subclause 4.2, "The SQL model"), a predicate in the WHERE clause is *satisfied* only by those rows for which the predicate evaluates to *True*.

The result of the WHERE clause is a virtual table of the same shape as the source table, as demonstrated in Figure 4, "Filtering". The WHERE clause is optional; if not specified, the result table is identical to the source table.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

```
FROM PRODUCTS, INVENTORY
```

Cross product

Name	Cost	Color	Supplier	Name	Qty
Apple	0.40	Red	FruitCo	Banana	125
Banana	0.35	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	FruitCo	Banana	125
Lime	0.15	Green	FruitCo	Banana	125
Tomato	0.65	Red	FruitCo	Banana	125
Apple	0.40	Red	SourInc	Lime	487
Banana	0.35	Yellow	SourInc	Lime	487
Lemon	0.12	Yellow	SourInc	Lime	487
Lime	0.15	Green	SourInc	Lime	487
Tomato	0.65	Red	SourInc	Lime	487

```
WHERE COLOR = 'Yellow' OR COST > 0.40
```

Filtered

Name	Cost	Color	Supplier	Name	Qty
Banana	0.35	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	FruitCo	Banana	125
Tomato	0.65	Red	FruitCo	Banana	125
Banana	0.35	Yellow	SourInc	Lime	487
Lemon	0.12	Yellow	SourInc	Lime	487
Tomato	0.65	Red	SourInc	Lime	487

Figure 4 — Filtering

— GROUP BY: The result of a GROUP BY clause is a *grouped table*. A grouped table is a set of groups, and a group is a collection of rows in which the value of each *grouping column* of every row in that group is "not distinct" from (roughly meaning "either equal to one another or both null") the value of the grouping column in every other row in that group. This action, though awkward to express in natural language, is illustrated in Figure 5, "Grouping".

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

```
WHERE COLOR = 'Yellow' OR COST > 0.40
```

Filtered

Name	Cost	Color	Supplier	Name	Qty
Banana	0.35	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	FruitCo	Banana	125
Tomato	0.65	Red	FruitCo	Banana	125
Banana	0.35	Yellow	SourInc	Lime	487
Lemon	0.12	Yellow	SourInc	Lime	487
Tomato	0.65	Red	SourInc	Lime	487

```
GROUP BY COLOR
```

Grouped

Name	Cost	Color	Supplier	Name	Qty
Banana	0.35	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	SourInc	Lime	487
Banana	0.35	Yellow	SourInc	Lime	487
Tomato	0.65	Red	SourInc	Lime	487
Tomato	0.65	Red	FruitCo	Banana	125

Figure 5 — Grouping

The GROUP BY clause is optional. If it is omitted and grouping is requested (by the inclusion of a HAVING clause of by the specification of a set function in the SELECT clause), the result table has a single group, the shape and content of which are the same as those of the source table. The principle purpose of grouping is to permit the projection operation (which is syntactically specified in the SELECT clause) to use *set functions* — functions that compute a single result by aggregating information from all rows in a group.

NOTE 9 — Set functions are also known as “aggregate functions” in SQL.

— HAVING: The purpose of the HAVING clause is to cause entire groups of the source table for which the predicate of the clause is not *True* to be omitted from the result table. The clause is optional; if omitted, no groups are omitted and the result table is identical to the source table.

— SELECT: Although specified first in the syntax of a query specification, the SELECT clause is the last clause to be evaluated (among the “simplest form” clauses). As with other clauses, the SELECT clause operates on the source table that results from the evaluation of the preceding clauses (those that are specified). Because this clause specifies result columns, it can (and normally does) generate a result table with a different shape than the source table. Figure 6, “Projection”, illustrates the behavior of the SELECT clause.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

```
GROUP BY COLOR
```

Grouped

Name	Cost	Color	Supplier	Name	Qty
Banana	0.35	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	FruitCo	Banana	125
Lemon	0.12	Yellow	SourInc	Lime	487
Banana	0.35	Yellow	SourInc	Lime	487
Tomato	0.65	Red	SourInc	Lime	487
Tomato	0.65	Red	FruitCo	Banana	125

```
SELECT COLOR, SUM(QTY), AVG(COST)
```

Projected

Color	*sum(qty)*	*avg(cost)*
Yellow	1224	0.235
Red	612	0.65

Figure 6 — Projection

The complete query expression illustrated by Figure 3, "Cartesian product", Figure 4, "Filtering", Figure 5, "Grouping", and Figure 6, "Projection", is:

```
SELECT COLOR, SUM(QTY), AVG(COST)
FROM PRODUCTS, INVENTORY
WHERE COLOR = 'Yellow' OR COST > 40
GROUP BY COLOR
```

Other syntax elements are also possible to use as part of a query. These include the following:

— The ORDER BY clause, which causes the result of the query to be ordered ("sorted") by the values in specified columns of the result or other specified criteria.

— The OFFSET clause, which forces the result of the query to omit all rows whose sequential position in the result is less than the value specified as an argument to the clause.

— The FETCH FIRST clause, which instructs the SQL implementation to return only rows of the result whose sequential position is less than or equal to the value specified as an argument to the clause.

The ORDER BY clause is useful primarily when specified in certain situations, such as when defining a cursor (an SQL object that provides the ability to process results of queries one row at a time). When specified, it syntactically follows the last clause of the remainder of the "simplest" query expression clauses. Its result table has the same shape as its source table.

The syntax of SQL's query expressions and query specifications is, of course, formally specified in ISO/IEC 9075-2.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

6.4 Table “shapes”

The SQL standard neither defines nor uses the word “shape” with respect to tables. Nonetheless, the word provides a convenient name for the concept of a table having a specific number of columns that each have specific names (or, in a few cases, are known to be unnamed) and specific data types.

The *shape* of a table is just that: the number of columns, the names of the columns, and the data types of the columns. The shape does not include the name of the table itself, nor does it have anything to do with the number of rows contained in the table or the values in the columns of those rows.

6.5 Subqueries

A *subquery* is a query expression that is used within another query expression or within an SQL-statement. There are three kinds of subquery: scalar subqueries, row subqueries, and table subqueries. As the name suggests, the result of a scalar subquery is a single (scalar) value of some SQL data type. Similarly, the result of a row subquery is a single row value such as the row of a virtual table. And, finally, the result of a table subquery is a virtual table.

Example 4, “Subquery use”, illustrates the use of a scalar subquery. In that example, the subquery is the parenthesized query specification (including the parentheses).

Example 4 — Subquery use

```
SELECT emp_name, hire_date
FROM employees
WHERE dept_id =
      ( SELECT dept_id
        FROM departments
        WHERE dept_name = 'Engineering' );
```

The syntax of subqueries is formally specified in ISO/IEC 9075-2.

6.6 Views

It is common for a query to make use of a virtual table in some manner. For instance, a query sometimes requires a table subquery in order to calculate a necessary intermediate result. Consider the following query to compute the average number of employees per department:

```
SELECT AVG(emps_in_dept)
FROM ( SELECT dept_id, COUNT(*) AS emps_in_dept
       FROM employees
       GROUP BY dept_id ) AS e;
```

This is a very simple example and is easily written with a subquery. However, it is sometimes the case that the necessary subquery is very common, or extremely complex, or error prone (perhaps relies on some subtlety of the data model that is not necessarily obvious to query writers). For these situations, it is possible to define the virtual table persistently. A virtual table with a named persistent definition is called a *view*.

The same query written using a view would look like this:

```
CREATE VIEW emps_in_dept_view AS
    SELECT dept_id, COUNT(*) AS emps_in_dept
    FROM employees
    GROUP BY dept_id;

SELECT AVG(emps_in_dept)
FROM emps_in_dept_view;
```

When a view is referenced in a query, the effect is as if the view is evaluated into a (virtual) table, and that table then used in the query.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

There are a number of reasons to use views:

— The view query is long, complex, very frequently used, or difficult to write correctly.

— It is possible that a view designer wants to use the view to "hide" query or data model logic that view users do not want to or need to know.

— Access control: users can be granted access to the view even if they do not have access to the underlying table or tables. This, of course, depends on the view creator having appropriate access to the underlying table or tables; see Subclause 12.4, "Special considerations for views".

— Implementations have an opportunity to optimize or pre-evaluate the view query for performance reasons, as long as the end result is not changed.

Views are most often used to define a virtual table for reading, such as in a SELECT query. It is also possible to modify data through a view; for instance, INSERT INTO a view; as long as the view is appropriately constructed. The SQL standard defines the conditions which must be met for a view to be an updateable view. The CHECK OPTION clause of the CREATE VIEW statement permits view definers to indicate whether or not rows inserted or updated through the view are required to still be visible via the view.

6.7 SQL-statements

SQL specifies a number of statements with which applications are able to make changes to an SQL database. There are a number of such statements; four that are of particular relevance are: INSERT, UPDATE, DELETE, and MERGE. The INSERT statement can create new data in a table. The UPDATE statement modifies data already contained in a table. The DELETE statement deletes data from a table. The MERGE statement can do one or more of the following: insert new data into a table, update existing data in a table, or delete existing data from a table.

The UPDATE statement has two forms. One form is capable of changing multiple rows in a single operation, while the other (as discussed in Subclause 8.14, "Cursors") modifies exactly one row. The DELETE statement similarly has two forms, one capable of deleting multiple rows in a single operation and the other that deletes exactly one row.

The SQL standard defines a number of other SQL-statements that are used to manage transactions, user sessions, privileges, and other aspects of SQL database use, but those statements do not affect the data contained in SQL databases. There is also no SQL-statement to retrieve a table from an SQL database, although there are mechanisms that allow applications to specify a virtual table for retrieval one row at a time; see Subclause 8.14, "Cursors", for additional information.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

7 Data structure and metadata

7.1 Metadata

The word "metadata" is defined to mean "data about data". Metadata describes one or more aspects of the data with which it is associated. There are numerous kinds of metadata (for example, describing the author and publication date of the data, the semantics of the data, or relationships between different sets of data) that are not used by SQL and are thus beyond the scope of this document. SQL uses a specific type of metadata, often described as "structural" metadata.

Structural metadata means that the metadata describes how data is logically organized (such as the names of tables containing data, the names and data types of the columns of those tables, and so forth). SQL's metadata also describes all persistent SQL objects (such as views, stored routines, constraints, privileges, and triggers).

7.2 Metadata and SQL

Metadata describes data, but there exists much data without any (explicit) metadata to describe it. Most data used within computer systems is described by some form of metadata, even if that form is inherent in the program logic of the applications that process the data.

SQL-data literally cannot exist without metadata. Every SQL database includes metadata describing the tables, views, and other objects contained in the database. This sometimes frustrates the desire to "quickly" prototype some sort of application without having to first painstakingly define the structures used to contain and organize the data. Ultimately, however, users of SQL systems are better served by the requirement that they describe the data to be used.

SQL's metadata is both persistent and transient. Persistent metadata is effectively stored within an SQL database in the form of tables and other objects belonging to a "definition schema", which is described in Subclause 7.4, "Definition schema". Persistent metadata is accessible by SQL-implementations for use by expression evaluation and statement execution in all transactions and for all applications.

7.3 SQL descriptors

SQL-implementations use transient metadata to describe SQL objects and context that are required for expression evaluation and statement execution in a specific transaction at a specific point in time. The SQL standard defines a number of "descriptors" to capture transient metadata, such as information about an SQL-connection between the SQL-agent and the SQL-implementation, tables and columns referenced in an expression, and the privileges available to the current user. Descriptors are specific to the thing that they describe and are never made available for use by concurrent transactions or applications.

There are many kinds of descriptors specified in the SQL standard. For example, table descriptors are used to describe all tables (virtual or physical) used or referenced during the evaluation of an expression or the execution of a statement. Table descriptors include a number of pieces of data, including the name (if any) of the table, the number of columns and their names and data types, and other pertinent information.

7.4 Definition schema

ISO/IEC 9075-11 provides the specifications of two assortments of SQL metadata. One is the definition schema mentioned in Subclause 7.2, "Metadata and SQL", and is named DEFINITION_SCHEMA. SQL-implementations are not required to implement DEFINITION_SCHEMA or any component of it. In fact,

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

rules in the standard prohibit application code from referencing the schema and its contents directly. (Instead, applications are able to perform privilege-based access to certain information through views of the *information schema* described in Subclause 7.5, "Information schema".)

DEFINITION_SCHEMA contains specifications of a number of tables and a small number of multi-table integrity constraints (called *assertions*). Those tables describe every schema that is part of the "database" that DEFINITION_SCHEMA describes, every table and view in each of those schemata, every column and constraint of every such table, and every other SQL object that can be contained in a schema.

7.5 Information schema

The other SQL metadata specified by ISO/IEC 9075-11 is INFORMATION_SCHEMA, which is an assortment of views that present information from DEFINITION_SCHEMA that the "current" user has privileges to access, transformed in some cases to a form more convenient for application use than the DEFINITION_SCHEMA tables themselves. An SQL-implementation that does not implement the DEFINITION_SCHEMA as defined in ISO/IEC 9075-11 is permitted to define an INFORMATION_SCHEMA that presents the same information, using views against that SQL-implementation's version of the equivalent metadata.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

8 The SQL-environment and its components

8.1 Nomenclature of the components of the SQL model

In ISO/IEC 9075-1 and ISO/IEC 9075-2, many components of the SQL model are named using a convention that prefixes a descriptive word or phrase with the string "SQL-". For example, the term "SQL-environment" is used to identify, not merely an environment of some sort, but specifically the environment in which SQL concepts are meaningful and within which the syntax and semantics of the SQL standard are applicable.

8.2 Components of the SQL model

The SQL model has many different components, as described in this Clause. Many, if not all, of those components are represented in the SQL standard itself as *descriptors*. For example, a *table descriptor* describes a particular table in the context of Syntax Rules and General Rules that manipulate that table. The use of such descriptors provides a more precise vocabulary to identify important aspects of, in this case, tables than the otherwise somewhat vague terms used in ordinary conversation. These descriptors are described in detail in Subclause 7.3, "SQL descriptors".

Subclause 7.1, "Metadata", presents the concept of metadata describing a set of data and the fact that, in SQL, metadata is represented in the form of schemata. Clause 7, "Data structure and metadata", discusses the structure of SQL schemata and their use in governing the processing of SQL expressions and statements.

Subclause 7.4, "Definition schema", and Subclause 7.5, "Information schema", describe the representation of schemata in SQL, particularly how the Definition Schema and the Information Schema (which are specified in ISO/IEC 9075-11) describe the content of what is often thought of as a "database".

8.3 SQL-environment

As its name suggests, an *SQL-environment* is an environment within which the SQL standard is defined and within which an implementation of the SQL standard operates. An SQL-environment incorporates (or includes) numerous objects, many of which are defined using the formal "SQL-" terminology defined in this document.

Among the components of an SQL-environment are:

— One *SQL-agent* (see Subclause 8.4, "SQL-agents");

— One *SQL-implementation* (see Subclause 8.5, "SQL-implementations");

— Zero or more *SQL-client modules*, each one containing one or more *externally-invoked procedures* that are available to the SQL-agent (see Subclause 8.6, "SQL-clients" and Subclause 8.10, "SQL-client modules, externally-invoked procedures, and embedded SQL");

— Zero or more *authorization identifiers* (see Subclause 12.1, "User identification");

— Zero or more *catalogs*, each of which contains one or more *SQL-schemas*, zero or more *foreign server descriptors*, and zero or more *foreign-data wrapper descriptors* (see Clause 7, "Data structure and metadata" and Subclause 9.7, "Foreign servers and foreign-data wrappers");

— Zero or more *user mappings* (see Subclause 9.7, "Foreign servers and foreign-data wrappers");

— Zero or more *routine mappings* (see Subclause 9.7, "Foreign servers and foreign-data wrappers");

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

— The *sites*, principally base tables, that contain SQL-data, as described by the contents of the schemata. This data may be thought of as "the database", but the term is not used in the ISO/IEC 9075 series, because it has different meanings in the broader context;

The SQL standard defines the syntax and semantics of database language SQL. In doing so, it necessarily makes assumptions that are not necessarily valid in actual implementations of the standard. Without those assumptions, the specifications of the standard would be unreasonably complex, likely to the point that the standard itself would be unreadable and unusable.

Among the most important of those assumptions are:

— Only a single "user" (e.g., application program, person typing instructions into a remote terminal) uses the SQL-implementation at any instant in time.

— All actions performed in the context of a single transaction are completed (successfully or not) before any actions are performed in the context of another transaction. (This is known as *serialization* of transactions.)

— Each SQL-statement is submitted in the form of a single SQL-statement contained in an externally-invoked procedure contained in an SQL-client module.

— Authorization identifiers (and users themselves) exist without SQL standard facilities to define or eliminate them. SQL-implementations often provide syntax or other mechanisms to create, manage, and destroy authorization identifiers.

— Schemata are created and destroyed by applications using SQL-statements intended for that purpose; the SQL standard recognizes the possibility that schemata can also be created and destroyed by implementation-defined means. Catalogs come into existence (and, perhaps, go out of existence) without the use of syntax defined by the SQL standard.

Of course, SQL-implementations usually allow multiple users to perform operations "at the same time" and provide transaction facilities that permit multiple transactions to operate concurrently. Many SQL-implementations also provide facilities for the creation (and destruction) of schemata; some even permit creation (and destruction) of catalogs. SQL-implementations often provide mechanisms for creating (and destroying) authorization identifiers and for identification and "registration" of users.

The remaining Subclauses in this Clause define the concepts cited in this Subclause, as well as additional concepts cited in those remaining Subclauses.

8.4 SQL-agents

ISO/IEC 9075-1:2023 [13], in Subclause 4.4.2, "SQL-agents", defines an SQL-agent thus: "An SQL-agent is that which causes the execution of SQL-statements."

That phrase evokes the notion of an application program, an interactive command-line tool, or perhaps some GUI environment that takes user input (often human input, possibly in the form of a textual representation of an SQL-statement; however, other kinds of input are possible that result in the creation of an SQL-statement) and then transfers a textual SQL-statement to the SQL-implementation for execution. The manner in which the SQL standard is written presumes a particular paradigm of statement execution, called "module language." Module language is examined in Subclause 8.10, "SQL-client modules, externally-invoked procedures, and embedded SQL".

8.5 SQL-implementations

The term "SQL-implementation" is defined in ISO/IEC 9075-1 to mean a "processor that processes SQL-statements". While true, that definition leaves a lot to the imagination. A more helpful definition, found in ISO/IEC 9075-1:2023 [13], Subclause 4.4.3.1, "Introduction to SQL-implementations", states: "An SQL-implementation is a processor that executes SQL-statements, as required by the SQL-agent. An SQL-

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

implementation, as perceived by the SQL-agent, includes one SQL-client, to which that SQL-agent is bound, and one or more SQL-servers."

This text specifies that an SQL-implementation includes both a single SQL-client and at least one SQL-server. An SQL-implementation is permitted, but not required, to incorporate multiple SQL-servers. Furthermore, the (single) SQL-client and the SQL-server (or SQL-servers) are not required to have been acquired from the same source. The SQL standard defines only the logical interface between the various components.

The word "implementation" implies a software program (or, more generally, a collection of such programs) that performs specific functions defined by a technical specification (of which a standard is one example). Therefore, an SQL-implementation comprises software that implements the SQL standard. Most SQL-implementations do not, in practice, implement the SQL standard exactly as written, without extensions, omissions, and variations. Those differences occur either because of resource limitations caused by the standard's complexity and size or because they are deemed necessary for the intended consumers of the SQL-implementation.

In fact, the SQL standard does not — and cannot — specify every facility that a practical SQL-implementation must provide to fulfill the needs of users of that SQL-implementation. For example, the SQL standard does not make any specifications regarding the performance or capacity of an SQL-implementation. Such aspects of implementations are necessarily left to the discretion of the individuals or organizations producing the SQL-implementations, based on their knowledge of the requirements of the intended user community and of the resources available to create the SQL-implementation.

Figure 7, "First approximation of SQL-environment", illustrates the concepts discussed to this point.

Figure 7 — First approximation of SQL-environment

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

8.6 SQL-clients

ISO/IEC 9075-1:2023 [13], Subclause 4.4.3.2, "SQL-clients", says that "An SQL-client is a processor, perceived by the SQL-agent as part of the SQL-implementation, that establishes SQL-connections between itself and SQL-servers and maintains a diagnostics area and other state data relating to interactions between itself, the SQL-agent, and the SQL-servers."

SQL-clients communicate with SQL-agents, accepting SQL-statements from those SQL-agents, and returning the results of the execution of those SQL-statements to the SQL-agents. SQL-clients also communicate with SQL-servers and thus act as a kind of intermediary between SQL-agents and SQL-servers.

8.7 SQL-servers

According to ISO/IEC 9075-1:2023 [13], Subclause 4.4.3.3, "SQL-servers", "Each SQL-server is a processor, perceived by the SQL-agent as part of the SQL-implementation, that manages SQL-data."

© ISO/IEC 2024 – All rights reserved

Thus, an SQL-server is exactly what it sounds like: it is the software component that is responsible for executing the SQL-statements that are given to it.

NOTE 10 — SQL-clients and SQL-servers correspond to the two components of the well-known client-server model of software systems.

8.8 SQL-connections

The definition in ISO/IEC 9075-1 of an SQL-connection states that it is an "association between an SQL-client and an SQL-server." The nature of that association is to provide a channel for communication between an SQL-client and an SQL-server. SQL-connections are usually considered to exist for a specified period, meaning that an SQL-connection is established between an SQL-client and an SQL-server at some point in time, utilized during the execution of SQL-statements on behalf of some user or application, and then destroyed when the user or application terminates its activities.

8.9 SQL-sessions

An SQL-session is defined in ISO/IEC 9075-1 to be the "context within which a single user, from a single SQL-agent, executes a sequence of consecutive SQL-statements over a single SQL-connection."

That is, an SQL-session is created when an SQL-connection is established between an SQL-client and an SQL-server, and is then destroyed when the SQL-connection is severed (or destroyed) once the user or application no longer requires its services.

Figure 8, "Second approximation of SQL-environment", extends the visualization started in Figure 7, "First approximation of SQL-environment", adding the additional concepts of SQL-clients, SQL-servers, SQL-connections, and SQL-sessions. That visualization includes an object called a "Diagnostics Area", which is described in Subclause 13.2, "Diagnostics area".

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

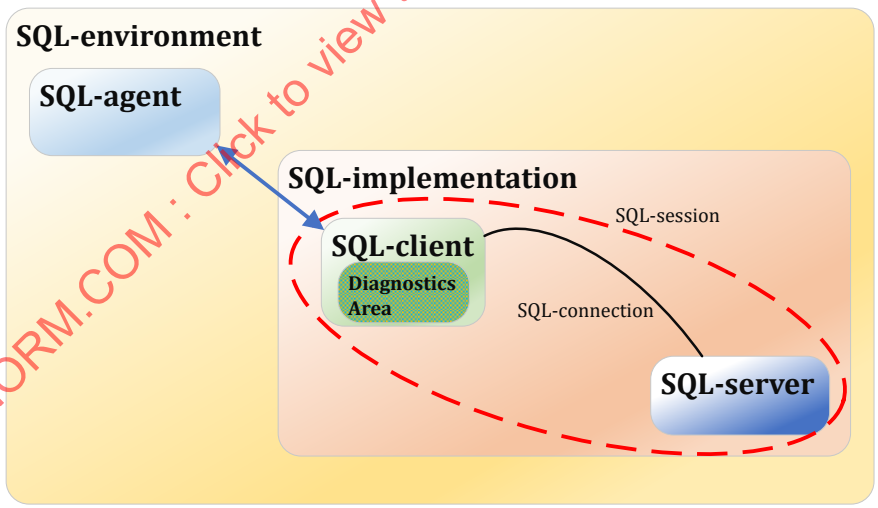

Figure 8 — Second approximation of SQL-environment

NOTE 11 — The line indicating the communication between the SQL-agent and the SQL-client is not given a specific name in the SQL model.

8.10 SQL-client modules, externally-invoked procedures, and embedded SQL

NOTE 12 — This subject is discussed with additional details in Clause 9, "Accessing SQL-data".

© ISO/IEC 2024 – All rights reserved

Subclause 4.4.4, "SQL-client modules", in ISO/IEC 9075-1:2023 [13] defines an SQL-client module to be "a module that is explicitly created and dropped[3] by implementation-defined mechanisms." The text goes on to clarify that SQL-client modules are permitted to have names and that permitted names are implementation-defined. Every SQL-client module contains one or more externally-invoked procedures, each of which contains a single SQL-statement. The text also specifies that, at any instant in time, there is exactly one SQL-client module associated with the SQL-agent. The specific nature of that "association" is not specified.

NOTE 13 — If an application comprises more than one SQL-client module, those SQL-client modules are associated with the SQL-agent one at a time.

Figure 9, "Relationship between SQL-client, SQL-client module, and SQL-statement", illustrates the relationships between an SQL-client, the SQL-client module that it "contains", and the single SQL-statement that each SQL-client module contains.

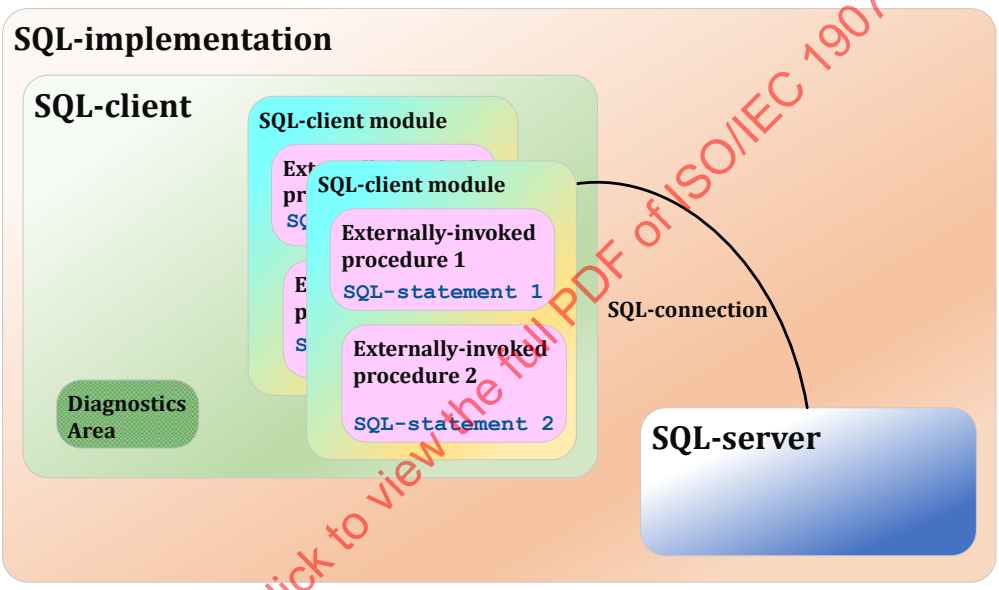

Figure 9 — Relationship between SQL-client, SQL-client module, and SQL-statement

Not all implementations of the SQL standard provide explicit SQL-client module capabilities. Instead, many SQL-implementations depend on applications to be written using SQL-statements embedded into host language programs (programs written in conventional programming languages, such as Fortran, C, or PL/I). These SQL-implementations transform such *embedded SQL programs* as though they are "compiled" into SQL-client modules containing "externally-invoked procedures"[4] that in turn contain a single SQL-statement each and "ordinary" programs that "call" the resulting externally-invoked procedures. Embedded SQL-statements contained in programs written in most programming languages are indicated by the use of "`EXEC SQL`" as a prefix to the remaining SQL syntax.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

3 "dropped" is used in ISO/IEC 9075 series [12] as a synonym for "destroyed".
4 The use of the phrase "externally-invoked" implies that those procedures are called from outside of the SQL-implementation — that is, from the SQL-agent.

© ISO/IEC 2024 – All rights reserved

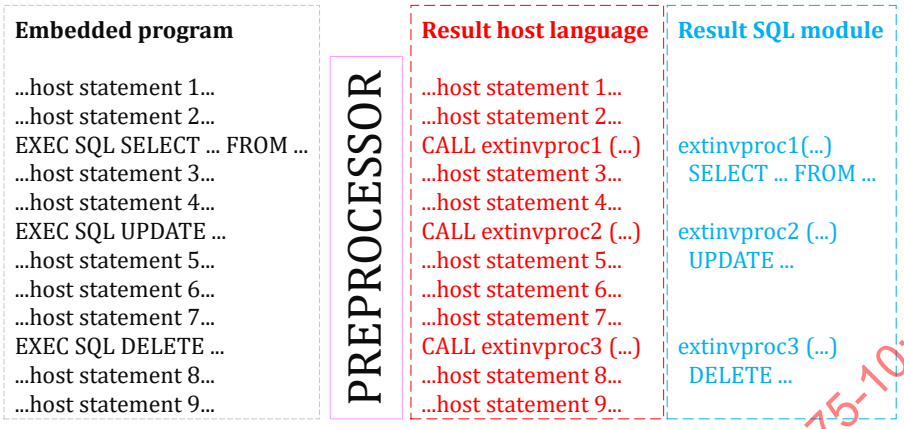

Figure 10 — Transformation from embedded SQL to externally-invoked procedure

Figure 10, "Transformation from embedded SQL to externally-invoked procedure", shows the relationship between an embedded SQL program and the (literal or metaphoric) SQL-client module, externally-invoked procedures, and contained SQL-statements that are eventually executed by an SQL-server.

8.11 Declarative and procedural statements

Database language SQL is, in many ways, a traditional programming language. It specifies a number of *statements* that are executed by an SQL-implementation.

However, the "query"-only component of SQL (which some call the "read-only" component, meaning those features of the language that do not cause changes to SQL-data) can be considered to be a functional language. That is, the greater fraction of the SQL standard concerns itself with retrieving data, specifically through the use of "non-procedural" (*declarative*) expressions that can be deeply nested and evaluated without causing any changes to underlying data.

In fact, the relational model of data (defined in Relational Model [32], of which SQL is an approximate implementation), derives its power explicitly from the ability to specify — through the use of a declarative syntax — the desired results of a query without having to resort to step-by-step instructions telling the underlying code precisely how to derive those results. In this sense, SQL is a declarative language and not a procedural language. Even those SQL-statements that do cause changes in the underlying data are declarative in nature.

SQL exhibits procedural characteristics only in the sense that real applications normally require the evaluation of multiple expressions and execution of multiple statements one after the other. Without the facilities of ISO/IEC 9075-4 [16], SQL gains its procedural capabilities only through the use of ordinary host programming language capabilities and calls to externally-invoked procedures (or the equivalent embedded SQL).

SQL, when the facilities of ISO/IEC 9075-4 [16] are included, is a "Turing-complete" programming language. That, taken literally, means that SQL has the necessary facilities to create, implement, or simulate an arbitrary Turing machine. The implication is that SQL can perform all computations that ordinary (non-quantum) computers can perform. That, in turn, means that every function provided by programming languages and their libraries can be written in SQL.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

8.12 Client-server model

8.12.1 Introduction to the client-server model

Although the earliest computers (and, later, the simplest computers) tended to be operated using individual programs that did not interact with other programs, modern complex systems depend entirely on interactions between many different pieces of software. Application programs interact with computers' operating system code, but also with other distinct programs, even other application programs. The details of those interactions with other distinct programs vary greatly, but they are frequently (if not always) in the form of one program acting as a "client" of the other "server" program, with the client submitting requests to the server and the server responding with the results of the request. This has long been known as the "client-server model" of computing.

Virtually all meaningful applications require the evaluation of numerous expressions and the execution of numerous statements, often sequentially. SQL's externally-invoked procedures (and the equivalent embedded SQL) provides the ability to specify sequential execution of statements and evaluation of expressions by using the underlying ordinary programming language facilities. The externally-invoked procedure (and embedded SQL) approach has a significant disadvantage: the execution of two consecutive SQL-statements requires that the host program issue a "call" to an externally-invoked procedure, which causes a change of context (from the SQL-client to the SQL-server), followed by results being returned to the host program, causing another change in context (from SQL-server to SQL-client), then the host program issues a second call, resulting in two more context changes. Context changes — whether changing processes within a single computer environment or issuing network messages to communicate between a client computer and a server computer — are expensive in terms of computer cycles and time.

8.12.2 SQL-server modules

One part of the SQL standard, ISO/IEC 9075-4 [16], permits an application to be written entirely in SQL (that is, no ordinary programming language statements or syntax is required) and stored persistently at the SQL-server. When SQL-statements are executed (and expressions evaluated) within such application code, there are no context changes required between the client (application program) and the server (SQL-implementation). The SQL-implementation evaluates all such expressions and executes such statements entirely within the SQL-server, returning results to the SQL-client only when the SQL code completes. This can result in significant performance improvements.

The use of such stored application code does not change the fact that SQL is inherently based on a client-server model. Stored application code is written in SQL (and not in a different programming language). Stored SQL code is contained within *SQL routines*, meaning routines (e.g., functions and procedures) that are written in SQL. SQL routines can be contained directly in schemata and are called *schema routines*. They can also be grouped within *SQL-server modules* that are stored directly in schemata.

Storing SQL code within an SQL-server does not change the semantics of that code. Figure 11, "Minimizing client-server context changes", illustrates this; contrast with the model illustrated in Figure 10, "Transformation from embedded SQL to externally-invoked procedure".

NOTE 14 — The "stored routine" approach improves the client-server performance only when all of the application logic depicted in Figure 10, "Transformation from embedded SQL to externally-invoked procedure", is incorporated into the code that is stored and invoked at the SQL-server.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

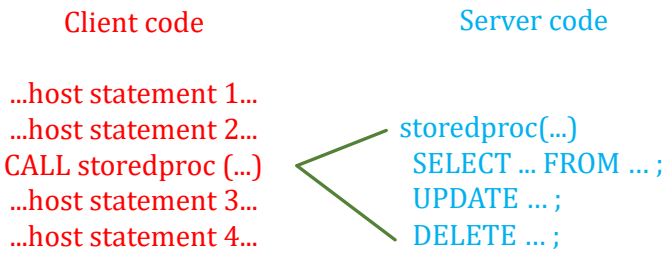

Figure 11 — Minimizing client-server context changes

8.13 SQL-statements

As described in earlier Subclauses, database language SQL has characteristics of a functional language — its <query expression>s are composed from other "lower-level" expressions and can be nested. (It is important to note that SQL expressions are, from the viewpoint of the standard, free from side effects; however, SQL expressions are allowed to invoke routines written in other programming languages, and the SQL standard does not have the ability to guarantee that such routines are free from side effects. For this reason, SQL is not formally claimed to be a functional programming language.) SQL also has characteristics of an ordinary procedural programming language — it provides a number of SQL-statements and, in the context of ISO/IEC 9075-4 [16] (in addition to a few other situations, like triggers), the ability to compose sequences of statements to be executed sequentially, to perform conditional execution of statements, and to create loops.

SQL-statements can be executed as a result of being specified within an <externally-invoked procedure> that is itself contained in an <SQL-client module definition> when the <externally-invoked procedure> is called from a host program written in an ordinary programming language (or, equivalently, by being contained in an embedded SQL program). In addition, most SQL-statements can be dynamically prepared and executed as described in Subclause 9.5, "Dynamic SQL". Furthermore, a number of SQL-statements can be executed "directly", which is also described in Subclause 9.2, ""Interactive" SQL". (The term "directly" implies that a user enters SQL-statements *directly* into a command-line interface or a GUI tool of some sort.)

Additionally, ISO/IEC 9075-2 defines six ways in which SQL-statements can be classified:

1) According to their effect on SQL objects, whether persistent objects (i.e., SQL-data, SQL-client modules, and schemata) or transient objects (such as SQL-sessions and other SQL-statements).

2) According to whether or not they start an SQL-transaction if there exists no SQL-transaction, or can, or shall, be executed when no SQL-transaction is active.

3) According to whether they possibly read SQL-data or possibly modify SQL-data.

4) According to whether or not they can be embedded in a program written in an ordinary programming language.

5) According to whether they can be dynamically prepared and executed.

6) According to whether or not they can be directly executed.

The main categories of SQL-statements defined by the ISO/IEC 9075 series are:

— SQL-schema statements — these statements create, alter, and destroy schema objects (e.g., tables, views, columns, constraints).

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

— SQL-data statements — these include all SQL-statements that access or make changes to SQL-data, including the declaration and use of cursors.

— SQL-transaction statements — explicit initiation and termination of SQL-transactions are performed by these statements.

— SQL-control statements — these statements (more are defined by other parts of the ISO/IEC 9075 series, such as ISO/IEC 9075-4 [16]) are used to invoke SQL-invoked procedures, return results from SQL-invoked functions, control the flow of SQL-statements, and so forth.

— SQL-connection statements — the creation and destruction of SQL-connections are handled by these statements.

— SQL-session statements — these statements set a number of characteristics associated with SQL-sessions.

— SQL-diagnostics statements — these statements retrieves information about the results of executing SQL-statements by accessing the diagnostics area.

— SQL-dynamic statements — these statements are used to prepare and execute SQL-statements dynamically and to process the results of those executions.

— SQL embedded exception declaration — this is a special-purpose declaration (i.e., not executable) statement used only in embedded SQL.

8.14 Cursors

Subclause 4.1, "Relational model", states that relations are sets of tuples. Subclause 4.2, "The SQL model", clarifies that SQL tables are multisets of rows. SQL provides many facilities (e.g., SQL-statements and query expressions) to process sets or multisets of rows at a time. However, there are occasions when an application must retrieve a single row at a time, performing program logic based on the values of the columns in that row, and possibly modifying or deleting the row itself.

Subclause 9.2, ""Interactive" SQL", observes that only a <direct select statement: multiple rows> can be used to retrieve more than a single row from SQL-data, and that the statement was available for use *only* in direct invocation.

SQL applications written using module language and its equivalent mechanism, embedded SQL, use a different technique for retrieving (and sometimes processing) SQL-data. That mechanism requires specifying the query expression that identifies the data that the application wishes to process as a *cursor specification* within a *cursor declaration*. A *cursor* behaves like a "pointer" to the rows of a virtual table. Applications can cause that pointer to move forward or backward, normally one row at a time (but it is possible to move the pointer by increments other than one). Those applications can also update the contents of the row identified by the cursor or delete the row.

Example 8, "Embedded SQL example", illustrated the declaration of two cursors (`EXEC SQL DECLARE CURSOR ...`) and the use of those cursors. When an application uses a cursor, it must first *open* the cursor (using an <open statement>), which effectively "computes" the virtual table defined by the cursor specification and positions the cursor on the first row of the cursor. (The rows of the virtual table defined by a cursor are ordered in some implementation-dependent order unless the cursor specification includes an ORDER BY clause. If that ORDER BY clause does not specify a complete ordering, then the rows are partially ordered.) Rows are then retrieved, one at a time, by the use of <fetch statement>s. The values of fetched rows may be modified by the use of <update statement: positioned>s and deleted by the use of <delete statement: positioned>s. When an application has completed processing the rows of a cursor, the cursor can be closed using a <close statement>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

9 Accessing SQL-data

9.1 Introduction to SQL-data access

SQL-data is accessed by using the SQL language. The SQL language, in the form of SQL-statements, is presented by an application or user to an SQL-implementation in one (or more) of several mechanisms.

— Interactively.

— SQL module language and externally-invoked procedures.

— SQL statements embedded in programs written in another programming language.

— SQL statements submitted for run-time preparation and execution using dynamic SQL statements.

— SQL statements submitted for run-time preparation and execution using a functional ("call-level") interface.

— SQL semantics invoked by application programs accessing foreign-data servers through a functional interface.

9.2 "Interactive" SQL

Most SQL-implementations provide some kind of application that permits users to enter SQL-statements directly through some user interface. The SQL-statements entered through such an application are analyzed and executed, with the results (perhaps only an indication of success or an error report) displayed through the same interface.

In the SQL standard, this kind of capability is called *direct invocation*. Direct invocation of SQL is the only environment in which the SQL standard provides a "SELECT statement" — that is, a query expression that is permitted to directly return more than a single row as a result. The standard provides the <direct select statement: multiple rows> for this purpose. Direct invocation is illustrated in Example 5, "Example of direct invocation".

Example 5 — Example of direct invocation

```
SQL-statement: SELECT EMPID, JOB_TITLE FROM EMPS

Results:
45231 Engineer
00427 Senior Vice President
.
.
13420 Secretary
```

NOTE 15 — As described in Subclause 8.14, "Cursors", other facilities are required when potentially retrieving more than one row of data from SQL-data using mechanisms other than direct invocation.

9.3 Module language

The fundamental syntax and semantics of the SQL standard are specified using the concept of *externally-invoked procedures*, each containing a single <SQL procedure statement>, that are collected together into <SQL-client module definition>s.

In this "pure SQL" paradigm, an application program comprises an SQL-client module (in the form of an <SQL-client module definition>) that contains some number of <externally-invoked procedure>s. A

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

separate piece of code, written in some conventional programming language (such as C, COBOL, or PL/I) contains programming logic as well as "call statement"s that invoke those <externally-invoked procedure>s, passing arguments to the procedures' parameters.

Example 6, "Module language example: SQL code fragment", and Example 7, "Module language example: COBOL code fragment", illustrate an <SQL-client module definition> and an application program that invokes the module's <externally-invoked procedure>s.

Example 6 — Module language example: SQL code fragment

```
MODULE data_input
  NAMES ARE LATIN1
  LANGUAGE COBOL
  SCHEMA video_and_music

-- By not specifying AUTHORIZATION, we are requiring that the authID
-- of the user executing the procedures in the module be the one used
-- for privilege checking.

-- Insert a new movie title into MOVIE_TITLES, and star information into
-- MOVIES_STARS.

-- First, insert into MOVIE_TITLES

PROCEDURE insert_movie_titles
  ( SQLSTATE,
    :title      CHARACTER (50),
    :year       CHARACTER (10),
    :cost       DECIMAL (5,2),
    :reg_rental DECIMAL (5,2),
    :cur_rental DECIMAL (5,2),
    :reg_sale   DECIMAL (5,2),
    :cur_sale   DECIMAL (5,2),
    :series     CHARACTER (3),
    :ser_ind    INTEGER,
    :type       CHARACTER (10),
    :vhs_owned  INTEGER,
    :beta_owned INTEGER,
    :vhs_stock  INTEGER,
    :beta_stock INTEGER );

INSERT INTO movie_titles
  VALUES (:title, CAST(:year AS DATE), :cost,
    :reg_rental, :cur_rental, :reg_sale, :cur_sale,
    :series INDICATOR :ser_ind, :type,
    :vhs_owned, :beta_owned, :vhs_stock, :beta_stock,
    :0, 0, 0, 0, 0.00, 0.00, 0.00, 0.00);


-- Insert corresponding information into MOVIES_STARS.

PROCEDURE insert_movies_stars
  . . .
```

Example 7 — Module language example: COBOL code fragment

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
  DATA-INPUT.

ENVIRONMENT DIVISION.
. . .

DATA DIVISION.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

```
FILE SECTION.
. . .

WORKING-STORAGE SECTION.
* Variables for interfacing with SQL procedures
01 SQLSTATE     PICTURE X(5).
01 TITLE        PICTURE X(50).
01 YEAR         PICTURE X(10).
01 COST         PICTURE S999V99 USAGE DISPLAY LEADING SEPARATE.
01 RENTAL-REG   PICTURE S999V99 USAGE DISPLAY LEADING SEPARATE.
01 RENTAL-CUR   PICTURE S999V99 USAGE DISPLAY LEADING SEPARATE.
01 SALE-REG     PICTURE S999V99 USAGE DISPLAY LEADING SEPARATE.
01 SALE-CUR     PICTURE S999V99 USAGE DISPLAY LEADING SEPARATE.
01 SERIES       PICTURE X(3).
01 SER-IND      PICTURE S9(5) USAGE BINARY.
01 TYPE         PICTURE X(10).
01 VHS-OWNED    PICTURE S9(5) USAGE BINARY.
01 BETA-OWNED   PICTURE S9(5) USAGE BINARY.
01 VHS-STOCK    PICTURE S9(5) USAGE BINARY.
01 BETA-STOCK   PICTURE S9(5) USAGE BINARY.
01 FIRST-NAME   PICTURE X(50).
01 LAST-NAME    PICTURE X(50).
01 LAST-NAME-I  PICTURE S9(5) USAGE BINARY.
01 ADDRESS      PICTURE X(50).
01 ADDRESS-IND  PICTURE S9(5) USAGE BINARY.
01 CITY         PICTURE X(50).
01 CITY-IND     PICTURE S9(5) USAGE BINARY.
01 STATE        PICTURE X(50).
01 STATE-IND    PICTURE S9(5) USAGE BINARY.
01 ZIP-CODE     PICTURE X(50).
01 ZIP-CODE-IND PICTURE S9(5) USAGE BINARY.
01 PHONE-NUMBER PICTURE X(50).
01 PHONE-IND    PICTURE S9(5) USAGE BINARY.
01 CREDIT       PICTURE X(50).
01 CREDIT-IND   PICTURE S9(5) USAGE BINARY.

*        Variable definitions

01 MESSAGE      PICTURE X(128).

PROCEDURE DIVISION.
MAIN-ROUTINE.
*
* Here would be some code to get input of some sort to determine
* which function to perform and what data to use for that function.
* Rather than bore you (and fill pages) with code that has little
* or nothing to do with the purpose of this example, we leave
* these details as an exercise for you to do on a slow weekend.
*
* We will assume that one of the following COBOL paragraphs will
* be PERFORMed by the preceding code, though.
*

INSERT-NEW-MOVIE.
  CALL "INSERT_MOVIE_TITLES"
    USING SQLSTATE, TITLE, YEAR, COST, RENTAL-REG, RENTAL-CUR,
      SALE-REG, SALE-CUR, SERIES, SER-IND, TYPE,
      VHS-OWNED, BETA-OWNED, VHS-STOCK, BETA-STOCK.
  IF SQLSTATE IS NOT '00000' THEN
    DISPLAY 'Error INSERTing MOVIE_TITLES; SQLSTATE = ', SQLSTATE.
    CALL "GET_DIAGNOSTICS"
      USING SQLSTATE, MESSAGE.
  DISPLAY 'Message in Diagnostics Area is: ', MESSAGE.
END-IF.


CALL "INSERT_MOVIES_STARS"
  . . .
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

```
CALL "COMMIT_TRANSACTION"
  USING SQLSTATE.
IF SQLSTATE IS NOT '00000' THEN
  DISPLAY 'Error INSERTing MOVIE_TITLES; SQLSTATE = ', SQLSTATE.
  CALL "GET_DIAGNOSTICS"
    USING SQLSTATE, MESSAGE.
  DISPLAY 'Message in Diagnostics Area is: ', MESSAGE.
END-IF.
```

NOTE 16 — The SQL standard's module language has been implemented by only a few SQL-implementations. Most implementations provide only embedded SQL (see Subclause 9.4, "Embedded SQL") capabilities that emulate the module language facility.

9.4 Embedded SQL

The SQL standard specifies a mechanism for "embedding" SQL language into the text of ordinary programming languages. The languages for which such embedding is defined in ISO/IEC 9075-2 are Ada, C, COBOL, Fortran, M (formerly known as MUMPS), Pascal, and PL/I. In addition, ISO/IEC 9075-10 [18] specifies syntax for embedding SQL language in the Java™ programming language.

In the SQL standard, embedded SQL is defined to be transformed into two components: module language that contains "pure" SQL (that is, no other programming language), and host language code that contains "call" statements to invoke the module language SQL. Actual implementations of embedded SQL typically implement embedded SQL by transforming the embedded SQL statements into executable code instead of actual module language; the semantics are identical using either mechanism.

Example 8, "Embedded SQL example", illustrates the use of embedded SQL (in this case, SQL code is embedded in code that is otherwise written in the C programming language).

Example 8 — Embedded SQL example

```
int cust_assist ()

{

  char   title[51], year[11], result[102], star_name[51];

  EXEC SQL DECLARE CURSOR movie_cursor FOR
    SELECT title, CAST (year_released AS CHARACTER(10))
    FROM movie_titles;

  EXEC SQL OPEN movie_cursor;

  if ( strcmp(SQLSTATE, "00000") != 0 )
    { /* Print, display, or otherwise record SQLSTATE value */
      EXEC SQL GET DIAGNOSTICS EXCEPTION 1 MESSAGE;
      /* Print, display, or otherwise record MESSAGE text */
    }
  while (1==1) {

    EXEC SQL FETCH NEXT FROM movie_cursor
      INTO :title, :year;

    if ( strcmp(SQLSTATE, "00000") != 0 )
      { /* Print, display, or otherwise record SQLSTATE value */
      EXEC SQL GET DIAGNOSTICS EXCEPTION 1 MESSAGE;
      /* Print, display, or otherwise record MESSAGE text */
      }

    /* Check SQLSTATE for '02000'; if so, then do not display values */


    if ( strcmp(SQLSTATE, "02000") != 0 )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

```
      /* Print, display, or otherwise output movie title */

    EXEC SQL CLOSE movie_cursor;

    if ( strcmp(SQLSTATE, "00000") != 0 )
      { /* Print, display, or otherwise record SQLSTATE value */
        EXEC SQL GET DIAGNOSTICS EXCEPTION 1 MESSAGE;
        /* Print, display, or otherwise record MESSAGE text */
      }

}
```

9.5 Dynamic SQL

Both module language and embedded SQL require that the actual text of all SQL-statements and expressions (such as query expressions) be fully known when the application program (e.g., the SQL module definition or the embedded SQL host program) is written. Many applications construct SQL-statements and expressions during the program execution, so neither the module language mechanism nor the embedded SQL mechanism can be used.

NOTE 17 — Because the actual text of SQL-statements and expressions are required to be fully known when the application program is written, module language and embedded SQL are sometimes called “static SQL”.

When application programs construct the SQL-statements and expressions during the programs’ execution, they can use a mechanism known as *dynamic SQL*. When dynamic SQL is used, the SQL language text of an SQL-statement or a query declaration is constructed as an ordinary character string. The application program passes that character string as an argument to a <prepare statement>, which performs a syntax analysis on the character string. If the character string satisfies the syntactic specifications of an SQL-statement, the statement is *prepared* for execution and the <prepare statement> returns a *statement handle* to the application program.

The application then passes that statement handle to an <execute statement> that causes the prepared statement to be executed.

Many times, a statement that is prepared for dynamic execution incorporates one or more *dynamic parameters*, either used to pass information needed for statement execution or to return information from the statement upon its completion. When this is the case, the application typically requests the SQL-implementation to *describe* those dynamic parameters. The prepared statement’s statement handle is passed to a <describe statement> that analyzes each of the dynamic parameters referenced in the prepared statement. That action places a description of each parameter into an *SQL descriptor area* that the application had previously allocated (using an <allocate descriptor statement>). The description of each dynamic parameter includes such information as its parameter name, its data type, and so forth. Applications sometimes “know” the appropriate descriptions of the dynamic parameters and can thus avoid requesting the SQL-implementation to provide the descriptions.

Each SQL descriptor area contains one or more *item descriptor areas*, each of which describes a single dynamic parameter. The contents of item descriptor areas can be retrieved by the application program (using a <get descriptor statement>) and can be modified by the program (using a <set descriptor statement>).

When SQL-statements that are to be executed dynamically do not have dynamic parameters, the application program can submit the statement text (the character string) to an <execute immediate statement>, which performs the syntax analysis on the character string and, if it satisfies the syntactic requirements of an SQL-statement, automatically executes it.

Not every SQL-statement can be handled using dynamic SQL. In particular, the <fetch statement> cannot be processed using dynamic preparation and execution; the <dynamic fetch statement> is used to dynamically process the rows of (dynamic) cursors.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Application programs using dynamic SQL do so through the use of SQL-statements such as <prepare statement>, <execute statement>, <describe statement>, and other statements described in this Subclause. Those statements are executed through either the module language mechanism or the embedded SQL mechanism and *not* through the dynamic SQL mechanism. For example, the <prepare statement> cannot be dynamically prepared and executed.

9.6 Other mechanisms to access SQL-data: Call-Level Interface (CLI)

There are other mechanisms in the SQL standard for accessing SQL-data: one of those is a *functional* (or *call-level*) interface. ISO/IEC 9075-3 [15] defines such an interface for SQL, commonly called "CLI" or "SQL/CLI". (The best-known example of an SQL/CLI implementation is the Microsoft interface named "ODBC". Other implementations, if they exist, are not widely known or used.) Applications that use SQL/CLI are not required to use either module language or embedded SQL-statements. Instead, all access to SQL-data is done by the invocation of CLI functions that are designed to perform tasks very similar to those of dynamic SQL-statements. For example, SQL/CLI defines functions to prepare an SQL-statement for execution, to execute a prepared statement, to prepare and execute a statement in one step, to allocate descriptor areas, and to retrieve values from and change values in descriptor areas. CLI replicates most capabilities of dynamic SQL and provides a few facilities that dynamic SQL does not. The key difference between SQL/CLI and dynamic SQL is that applications using CLI do not have to explicitly use any SQL-statements at all other than those being used dynamically through the CLI interface.

9.7 Foreign servers and foreign-data wrappers

ISO/IEC 9075-9 [17] specifies still another mechanism for accessing data — but not SQL-data. Instead, facilities are defined that allows for the identification of *foreign servers* containing data that is wanted for access by SQL applications and *foreign-data wrappers* that allow those SQL applications to access that data almost as if it were actually SQL-data.

ISO/IEC 9075-9 [17] (commonly known as "MED" for "Management of External Data") defines a number of functions that can be invoked from SQL-servers as they perform SQL-statements and evaluate query expressions that reference foreign tables. Those functions allow applications to specify information about the foreign (that is, non-SQL) servers they wish to access, to connect to and disconnect from those servers, and to retrieve data from and manipulate data stored by those servers using ordinary SQL syntax. The foreign-data wrappers implement a subset of those functions and provide a mapping between SQL concepts (such as query expressions and cursors) and the corresponding concepts used by the software implementing those foreign servers.

Data managed by foreign servers is often protected against unauthorized access. Foreign server user identifiers, if they exist at all, are not necessarily compatible with SQL syntax, so SQL/MED provides the ability to map SQL authorization identifiers to foreign servers.

Similarly, foreign servers often provide a variety of subroutines that are invoked as functions, but whose routines names are not necessarily compatible with SQL syntax. SQL/MED also provides the ability to map SQL routine names to foreign server routines, thus allowing SQL applications to invoke those routines from SQL expressions or statements.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

10 Active databases and SQL

10.1 Introduction to active databases

When a database management system performs additional actions that are not implicit in the syntax of the SQL-statements being executed — for example, maintenance operations used to keep values in different tables consistent when some of those values are changed by the execution of SQL-statements — that system is commonly known as an *active database* system.

SQL defines two facilities that relate to active database mechanisms. The first of these is specified in the form of *referential actions* that are defined as part of SQL's referential integrity capabilities. The second is a facility commonly known as *trigger*s.

10.2 Referential integrity and referential actions

Subclause 5.3, "Constraints", described several kinds of integrity constraint, including referential integrity constraints. A *referential integrity constraint* is a constraint defined on a table that specifies that the value of a specified column (or the values of a specified group of columns), called the *referencing column*(s), in every row (*referencing row*) of that table is required to be found as the value of a column (or group of columns), called the *referenced column*(s) in some row (the *referenced row*) of a specified table. This relationship is illustrated in Figure 2, "Primary key — Foreign key relationships".

SQL permits referential integrity constraints to be specified with optional syntax known as a *referential triggered action* (more commonly, *referential action*). A referential action specifies what actions the SQL-implementation takes whenever an application instructs the modification or deletion of one or more rows contained in the table that the referential constraint identifies as the referenced table (the "parent table", as defined in Subclause 5.3, "Constraints").

Possible referential actions include "NO ACTION", which means that no action is to be taken except completion of the modification or deletion of the identified rows. Because no additional action is taken, referencing rows (rows that reference the rows that are modified or deleted) are not updated to reflect the changes to the referenced rows. As a consequence, the execution of the SQL statement (modifying or deleting referenced rows) fail if there are rows for which the referential constraint is not satisfied.

A more common action is "RESTRICT". When an application executes an UPDATE or a DELETE statement that would cause the modification or deletion of a row in a table on which a referential constraint is defined (a "parent table") that has a referential action of RESTRICT, the database checks whether there are any referencing tables that contain rows (referencing rows) that "match" the row or rows being modified in or deleted from the referenced table. If there are matching referencing rows, the UPDATE or DELETE statement fails with a status that indicates the reason: the presence of referencing rows.

Other possible actions are:

— "CASCADE" (meaning to update all referencing columns of referencing rows to match the modified columns in the referenced rows, or to delete all referencing rows, depending on whether the application is updating or deleting the referenced rows);

— "SET NULL" (meaning to set the values of all referencing columns of referencing rows to the SQL null value);

— "SET DEFAULT" (meaning to set the values of all referencing columns of referencing rows to their default values, or to the null value if there is no explicit default value).

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Matching rows of referencing tables with rows of referenced tables can be simple equality, or can be required to account for null values in some or all referencing columns. The details of such matching are beyond the scope of this document.

10.3 Triggers

SQL provides a second active database facility with greater generality than referential actions; this second facility is the *trigger*. "A trigger is a specification for a given action to take place every time a given operation takes place on a given object."[5]

That action, the *triggered action*, is an SQL-statement (possibly a compound statement) and the object is either a table or a view, called the *subject table* of the trigger. The operation that causes the action to take place is the insertion, deletion, or replacement of a row or collection or rows in the table or view. The trigger is therefore called an *insert trigger*, a *delete trigger*, or an *update trigger*.

The triggered action is specified to be taken either before the operation itself, after the operation, or even instead of the operation. The trigger is thus known as either a *before trigger*, an *after trigger*, or an *instead of trigger*. The terms can be combined: "instead of insert trigger".

Before and after triggers are permitted to be specified only on base tables, while instead of triggers are permitted to be specified only on views. An instead of insert trigger can only be specified on a view that is not otherwise capable of supporting an insert operation; similarly, an instead of update trigger or an instead of delete trigger can only be specified on a view that is not otherwise capable of supporting an update statement or a delete statement, respectively.

Triggers are caused to be invoked ("triggered" or "fired") by the insertion of a new row in the subject table, by the modification of a specified column or columns of an existing row in the subject table, or by the deletion of an existing row from the subject table. That change of SQL-data is often caused by the application's explicit invocation of an INSERT, UPDATE, or DELETE statement. However, another potential cause is the evaluation of a referential integrity constraint and its contained referential action. Thus, there is an interaction between the referential actions of referential constraints and triggered actions of triggers.

Triggers are defined to be statement-level triggers or row-level triggers. Statement-level triggers are executed once per SQL-statement that causes the trigger to be "fired", regardless of how many rows (including no rows at all) are affected by that SQL-statement. Row-level triggers are executed once per row that is affected by the triggering event; if no rows are affected by the triggering event, then no row-level triggers are executed at all.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

5 ISO/IEC 9075-2:2023 [14], Subclause 4.46, "Triggers", first paragraph.

© ISO/IEC 2024 – All rights reserved

11 Transaction model

Subclause 5.2, "Transactions", states that a transaction is "an atomic unit of work that either succeeds entirely or fails entirely." The ISO/IEC 9075 series specifies a transaction model that presumes, for specification purposes only, that only a single transaction is ever in progress at one time. However, an SQL-transaction has a number of characteristics, one of which is its *isolation level*.

An SQL-transaction that is specified to be *serializable* behaves as though it is the only transaction being processed by its SQL-implementation at the time of its processing. If multiple serializable transactions are being processed concurrently, their effects are, in most ways, invisible to one another. However, SQL-implementations of serializable transactions sometimes encounter conflicts in which one transaction has modified or deleted a row that a different concurrent transactions tries to access; in such cases, the SQL-implementation is free to cancel the actions of one or both transactions and raising an exception to inform the application programs of the cancellations. Other levels of isolation are defined in terms of specific phenomena that are either permitted or prohibited with respect to the execution of operations on SQL-data under the control of a specified SQL-transaction. There are three such phenomena defined in ISO/IEC 9075-2:

— "Dirty read" — SQL-transaction *T1* modifies a row. SQL-transaction *T2* then reads that row before *T1* performs a COMMIT. If, instead, *T1* performs a ROLLBACK, *T2* will have read a row that was never committed and that may thus be considered to have never existed.

— "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 either receives the modified value or discovers that the row has been deleted.

— "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 following examples illustrate these phenomena:

Table 1 — Dirty read

Time	Transaction 1	Transaction 2
T0	START TRANSACTION;	START TRANSACTION;

T1	SELECT c1, unique_id FROM mytbl WHERE unique_id = 123;
	`c1 unique_id`
	`----- -----------`
	`XYZ 123`	

T2	UPDATE mytbl SET c1 = 'ABC' WHERE unique_id = 123;	

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Time	Transaction 1	Transaction 2
T3		SELECT c1, unique_id FROM mytbl WHERE unique_id = 123; c1 unique_id ----- ----------- ABC 123
T4	ROLLBACK;	

In Table 1, "Dirty read", at time T4, Transaction 2 has read a value of "ABC" but Transaction 1 has rolled back the update so that value of c1 in the data is "XYZ". Transaction 2 is operating on data that Transaction T1 decided was a mistake.

Table 2 — Non-repeatable read

Time	Transaction 1	Transaction 2
T0	START TRANSACTION;	START TRANSACTION;
T1	SELECT c1, unique_id FROM mytbl WHERE unique_id = 123; c1 unique_id ----- ----------- XYZ 123	
T2		UPDATE mytbl SET c1 = 'ABC' WHERE unique_id = 123;
T3		COMMIT;
T4	SELECT c1, unique_id FROM mytbl WHERE unique_id = 123; c1 unique_id ----- ----------- ABC 123	

In Table 2, "Non-repeatable read", at time T1, Transaction 1 retrieves c1 with a value of "XYZ". At time T4, Transaction 1 repeats the query but gets a different value for c1: "ABC".

Table 3 — Phantom read

Time	Transaction 1	Transaction 2
T0	START TRANSACTION;	START TRANSACTION;
T1	SELECT c1, unique_id FROM mytbl WHERE unique_id = 123; c1 unique_id ----- ----------- XYZ 123	

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Time	Transaction 1	Transaction 2
T2		INSERT INTO mytbl (c1, unique_id) VALUES ('XYZ', 456);
T3		COMMIT;
T4	SELECT c1, unique_id FROM mytbl WHERE unique_id = 123; c1 unique_id ----- ----------- ABC 123 XYZ 456	

In Table 3, "Phantom read", Transaction 1 gets a single row for the query. At time T4, Transaction 1 repeats the query but gets two rows.

SQL-transactions having an isolation level of SERIALIZABLE never encounter any of those phenomena. SQL-transactions with isolation level REPEATABLE READ may encounter the "Phantom" phenomenon, but neither of the other two phenomena. SQL-transactions declared to have READ COMMITTED as their isolation level may encounter the "Phantom" and "Non-repeatable read" phenomena, but will not encounter the "Dirty read" phenomenon. Of course, SQL-transactions having isolation level READ UNCOMMITTED may encounter any or all of those three phenomena. However, regardless of the SQL-transaction's isolation level, none of the phenomena are ever experienced during the implicit reading of schema definitions on behalf of executing an SQL-statement, or while checking integrity constraints, or while performing referential actions.

NOTE 18 — There are numerous phenomena that theoretically exist in the context of transactions. The SQL standard specifies these three phenomena based on perceptions of transaction theory, implementation capabilities, and application requirements. Future editions of the SQL standard can, if demand exists, augment or entirely replace the specification of transaction isolations levels and phenomena.

Every access to SQL-data by execution of SQL-statements or evaluation of SQL expressions takes place in the context of an SQL-transaction. Applications are permitted to explicitly initiate an SQL-transaction by execution of a START TRANSACTION statement that specifies the desired transaction isolation level, whether the transaction is intended to be READ ONLY or READ WRITE, and how large a diagnostics area is desired. Applications that do not explicitly initiate an SQL-transaction cause the automatic initiation of an SQL-transaction (with default characteristics) whenever they perform some access to (even retrieval of) SQL-data. An SQL-statement that performs some access to SQL-data is called a *transaction-initiating* SQL-statement.

If execution of an SQL-statement fails (for example, because one of its operations resulted in an exception condition being raised), the entire effect of that SQL-statement is reversed; the database is left in such a condition as though the statement had never been attempted. This all-or-nothing characteristic is referred to as "atomic statement" execution. If the raising of the exception condition is not *handled* (a facility in ISO/IEC 9075-4 [16] permits handling of completion conditions and exception conditions), then the SQL-transaction within which the exception condition is raised is automatically rolled back ("undone") and the database is left in such condition as though no actions had been performed under control of that SQL-transaction. Applications are allowed to cause their transactions to be rolled back, by execution of a ROLLBACK statement. Under normal conditions, however, applications will terminate their SQL-transactions by executing a COMMIT statement.

NOTE 19 — A facility (<embedded exception declaration>) in ISO/IEC 9075-2:2023 [14] provides a mechanism for programs written using embedded SQL to detect exception conditions, as well as warning conditions and no-data conditions, and to alter program flow upon detection. The embedded SQL mechanism is not as generalized as the exception handling capabilities defined in ISO/IEC 9075-4 [16].

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

SQL does not support a transaction model that allows “nested” transactions (or subtransactions). It does, however, support a “savepoint” mechanism in which an application executes a SAVEPOINT statement that established a “marker” in the course of the transaction. Applications are then able to undo *part* of an SQL-transaction by rolling the SQL-transaction back only as far as the savepoint that was established. Multiple savepoints can be established by the specification of a savepoint name that can be specified in a ROLLBACK TO SAVEPOINT statement. Applications can also explicitly “release” savepoints, indicating that they do not continue to require the ability to rollback to those savepoints.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

12 Security model

12.1 User identification

Data has value and must be protected. Sometimes, the protection is merely against unauthorized changes, including unauthorized deletion and creation. In many environments, the value of data is such that all access, including viewing the data, must be restricted. For example, data related to corporate finances, individual persons' medical records, and national security has such high value that access must be strictly controlled.

In an SQL-environment, all access to SQL-data and to its metadata can be controlled so that only specified users have access to certain data and metadata. Users are identified to the SQL-implementations by *authorization identifier*s, which ordinarily identify individual users. Users may also be identified by a *role identifier*s, which may identify multiple users in terms of the role(s) that they hold in the organization that owns the SQL-data. The keyword PUBLIC is used to specify that all users, both existing and those created in the future, are being referenced.

When a user becomes known to the SQL-implementation, an authorization identifier is the name by which the user is known. When necessary, a suitably authorized user *grant*s a specific role (in the form of a role identifier) to some authorization identifier, which causes the SQL-implementation to recognize the privileges of that authorization identifier *and* of the granted role when determining what access is permitted. Roles are also *revoked* when no longer needed.

12.2 Ownership of SQL-data

In an SQL-environment, a schema (see Clause 15, "Schema creation and manipulation") is nominally "owned" by the user that caused its creation. The owner of a schema is also the owner of most of the objects contained in that schema (which includes tables, constraints, and several other objects). The owner of the schema and its contained objects usually has complete access to the schema objects and the SQL-data contained in the tables. (An exception is described in Subclause 12.4, "Special considerations for views".)

The owner of a schema is always permitted to create new schema objects, although there can be restrictions on the creation of objects that reference other objects (for example, objects in a schema owned by a different user) to which the owner does not have appropriate access. The owner of a schema is also able to grant access to other users and to later revoke that access. The creator of a schema object (as well as the schema itself) is also usually able to delete the schema object (and schema), possibly limited because of the existence of other objects that reference the object to be deleted. SQL provides abilities to restrict deletion of objects that are referenced by other objects, as well as to "cascade" the deletion such that referencing objects are also deleted.

12.3 Privileges and privileged objects

The rights to perform particular kinds of access to schema objects are described by *privilege*s. The privileges that can be granted for certain kinds of access to schema objects of a particular kind are specific to the kind of schema object. For example, access can be granted to SQL tables that allow the *grantee* to view the contents of the table or the contents of specific columns of a table (SELECT privilege), to create new rows in the table (INSERT privilege), to modify the content of rows that already exist in the table or the contents of specific columns of the table (UPDATE privilege), and to delete existing rows from the table (DELETE privilege). Other table-related privileges exist, including permission to create a foreign key that references the table (REFERENCES privilege).

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

Other schema objects on which privileges can be granted by the owner to other users include domains, character sets, collations, user-defined types, and user-defined routines. Privileges can be granted to specific users, to roles, or to PUBLIC. A privilege that has been granted to PUBLIC is one that can be used by every user known to the SQL-implementation, including those that become known to the SQL-implementation after the privilege was granted.

The *grantor* of a privilege on a schema object is an authorization identifier that identifies the user or role that either owns the object or possesses the privilege with permission to further grant that privilege to others. That permission to further grant a privilege to others is indicated by having been granted the privilege WITH GRANT OPTION. Under certain circumstances (the details are beyond the scope of this document), the action of granting a privilege to a user can be done as though the grantor were a different authorization identifier (or role) than the grantor (GRANTED BY "authid-or-role"). (Generally, such circumstances are limited to database administrators and other “super users”, but that is not addressed by the SQL standard.)

NOTE 20 — All privilege descriptors identify both the grantee of the privilege being described and the grantor of that privilege. The owner of a database object is identified as the grantee of the privileges on that object, and the grantor of those privileges is a special value that does not correspond to any actual database user. That special value is “_SYSTEM”.

12.4 Special considerations for views

As described in Subclause 12.2, “Ownership of SQL-data”, the creator of a schema object is normally granted all privileges on that object (by the SQL-implementation); the major exceptions to that general rule are views.

There exist special considerations for privileges related to views. Views are, of course, tables, the values of which are “computed” upon demand (instead of being directly stored as persistent data). When a user creates a view, the view references one or more columns of one or more tables (or other views). The view creator must have at least some privilege on every column in every table that the view references. The privileges that the view creator has on the view are limited by the privileges that the creator has on those underlying tables and columns.

One consequence of that limitation is that the view creator cannot grant to other users any privileges that the creator did not possess WITH GRANT OPTION. In particular, the view creator cannot grant to another user any privileges on the view unless the creator has privileges on every column referenced by the view definition. This restriction protects SQL-data in tables owned by one user from unauthorized access by other owners through chains of views created by users that do not have such authorization.

12.5 Definer’s and invoker’s rights

Functions and procedures (*routines*) that are written in SQL and persistently stored in an SQL database (as specified in Subclause 11.60, “<SQL-invoked routine>”, in ISO/IEC 9075-2:2023 [14], and in ISO/IEC 9075-4 [16]) can be invoked, or “called”, by any user to which the EXECUTE privilege has been granted. Such routines are able to access schema objects (such as tables) only when the appropriate privileges are available. Those privileges can be implicitly granted to the routines themselves if the creator of the routines has been granted the privileges. In this case, the routines are created to execute with *definer’s rights*, meaning that the privileges used to allow or deny access to schema objects by SQL-statements contained in the routine are restricted to the privileges that the routine’s creator has.

Alternatively, some routines are created with *invoker’s rights*, which means that the privileges of the user that invoked the routine are used to determine what access to schema objects is permitted.

In SQL, when a routine is being created with definer’s rights, the creation syntax contains the keywords SQL SECURITY DEFINER. Similarly, when a routine is being created with invoker’s rights, the creation syntax contains the keywords SQL SECURITY INVOKER.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

13 Diagnostics model

13.1 SQLSTATE

The execution of every SQL-statement — if successful — can cause a change to SQL-data, a change to a schema and/or one or more of its contained objects, or a change to the SQL-environment. That statement execution *always* produces a status to indicate whether the execution succeeded or failed. The SQL standard uses the terminology "a[n] XXX condition is raised" to indicate the status that results from the statement execution. Status results are reported in a kind of "global variable" named SQLSTATE. When a condition is raised, it is always reported by setting the value of SQLSTATE to indicate the condition. Applications are able to query the value of SQLSTATE when desired.

When an SQL-statement is executed, one result of that execution is determination of the SQL-statement's success or failure. There are several possible ways in which an SQL-statement can terminate:

— If execution of the SQL-statement results in an error that prevents the statement from accomplishing its intent, an "exception condition" is raised. There are many exception conditions defined in the ISO/IEC 9075 series, including such errors as "cardinality violation", "data exception", and "integrity constraint violation". Many defined exception conditions ("data exception" is a notable example) have so many possible causes that the SQL standard has defined "subclasses" of the condition codes; for example, "data exception — division by zero" is the exception condition raised if an SQL-statement attempts to divide a value by zero.

— If some minor issue occurred that did not prevent the SQL-statement from completing its intent, the SQL-implementation is permitted to issue a *warning*, which is a kind of completion condition.

— If the SQL-statement was intended to view, create, modify, or insert SQL-data, but there were no data viewed, created, modified, or deleted, a third kind of completion condition, the *no data* condition, is raised.

— Otherwise, the SQL-statement was executed without error or other condition that requires attention from the application; that is said to be a "completion condition", specifically "successful completion".

When execution of an SQL-statement terminates with an exception condition, any changes made in whole or in part by that SQL-statement are reversed so the state of SQL-data is unchanged from its state before the statement was attempted.

13.2 Diagnostics area

The ISO/IEC 9075 series defines, in addition to SQLSTATE, a *diagnostics area*, which is a special data structure in which information is recorded about the results of execution of an SQL-statement. Diagnostics areas have two primary components: a *statement information* area and a *condition information* area. The statement information area is populated with information that pertains to the entire SQL-statement and its execution attempt. That includes codes to indicate exactly what the SQL-statement was, whether it was executed statically or using dynamic SQL, how many rows were affected by the statement's execution, and whether the transaction within which its execution occurred was committed, rolled back, or left active.

When an attempt is made to execute an SQL-statement, the SQL-implementation is permitted to terminate execution upon discovery of any exception condition. In that case, the condition information area of the diagnostics area will contain information about exactly that single exception condition. However, some SQL-implementations choose to continue statement execution as far as possible, gathering information

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

about multiple exception conditions that are encountered; in such situations, the condition information area may contain information about a large number of conditions.

Condition information areas contain information about the reasons why conditions (usually exception conditions) occurred, and that information relates closely to the purpose(s) of the SQL-statement and the SQL-data or other schema objects that it would have affected. For example, if an exception condition were raised during the execution of an SQL-invoked routine that was invoked by the SQL-statement, the name of the routine would be recorded in a condition information area. A field in the condition information area named RETURNED_SQLSTATE contains the SQLSTATE value associated with the condition recorded in the particular condition information area.

Applications use the GET DIAGNOSTICS statement to query the diagnostics area, and can query both the statement information area and any or all of the condition information areas.

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved

14 Data types

14.1 Built-in data types

14.1.1 Atomic data types

All data have a data type. The SQL standard defines a number of data types, often called *predefined* data types or *built-in* data types, that are used to specify the data types of SQL-data. SQL predefined data types include *atomic* data types. SQL provides two additional categories of data types: *constructed* data types, and *user-defined* data types.

Atomic data types include those used to represent numeric data, character string data, binary string data, Boolean data, date and time data, foreign-data link data, XML data, and JSON data.

The data types predefined in SQL to represent numeric data include types that represent *exact numeric* data and others that represent *approximate numeric* data. The exact numeric data types include INTEGER, SMALLINT, and BIGINT, which each represent integer data (data without a fractional component) and are usually implemented by binary data hardware types. Exact numeric types also include NUMERIC and DECIMAL, used to represent *scaled* data (with a precise fractional component permitted) and are often implemented by decimal data hardware types. Approximate numeric types, often called *floating point* data types, are named FLOAT, REAL, and DOUBLE PRECISION, which are usually implemented by binary floating point hardware types. The other numeric type is the decimal floating point type, DECFLOAT, which is sometimes implemented by a decimal floating point hardware type.

Character string data in SQL are characterized by the CHARACTER type, which is defined to have a fixed length; CHARACTER(20) is a data type that always stores exactly 20 characters — if an application stores fewer characters into a location of that data type, the SQL-implementation *pads* the value to fill 20 characters. SQL also provides the CHARACTER VARYING (which can be abbreviated VARCHAR) data type, which is specified with a *maximum* length; if an application stores 12 characters in a location declared to be CHARACTER VARYING(20), the value is not padded with spaces or anything else — the SQL-implementation simply notes that the location has exactly 12 characters stored. SQL also provides a data type called CHARACTER LARGE OBJECT (which can be abbreviated as CLOB) that permits applications to store potentially very large character strings. Values of the CHARACTER LARGE OBJECT type behave much like values of the CHARACTER VARYING type. The primary difference between the two types is that most SQL-implementations support much larger CHARACTER LARGE OBJECT values than CHARACTER VARYING values and optimize storage differently for the two types of values.

SQL provides data types for storing binary string data; those types are analogous to the character string data types, except that each “unit” is a single byte (the SQL standard uses the word “octet”) instead of a character. The binary string data types are named BINARY, BINARY VARYING, and BINARY LARGE OBJECT.

When applications deal with date and time information, SQL provides both datetime data types and interval data types. The datetime data types are named DATE, TIME, and TIMESTAMP; the third of those options is used to store data that both date and time components. Both the TIME and the TIMESTAMP types accept an optional time zone component (which allows specification of the number of hours and minutes the specified time zone is offset from UTC. When sites are declared to have a data type of TIME or TIMESTAMP, the syntax WITH TIME ZONE or WITHOUT TIME ZONE is specified to indicate whether the site accepts the time zone component or not. WITHOUT TIME ZONE is the default if neither is specified.

The interval data type is named INTERVAL; that type is given an “interval qualifier” that determines whether the data is a year-month interval or a day-time interval. Year-month intervals comprise either one or two components: integral numbers of years, integral numbers of months, or both (that is, they cannot represent days, hours, minutes, or seconds). Day-time intervals comprise up to four components:

IECNORM.COM : Click to view the full PDF of ISO/IEC 19075-10:2024

© ISO/IEC 2024 – All rights reserved