TECHNICAL REPORT

ISO/IEC TR 19075-6

First edition
2017-03

Information technology — Database languages — SQL Technical Reports —

Part 6: SQL support for JavaScript Object Notation (JSON)

Technologies de l'information — Langages de base de données — SQL rapport techniques —

Partie 6: Support de SQL pour JavaScript Object Notation (JSON)

Reference number
ISO/IEC TR 19075-6:2017(E)

Contents

Tables

Figures

Figure **Page**

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

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

For an explanation on 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 the following URL: www.iso.org/iso/foreword.html.

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

A list of all parts in the ISO/IEC 19075 series can be found on the ISO website.

NOTE 1 — The individual parts of multi-part technical reports are not necessarily published together. New editions of one or more parts can be published without publication of new editions of other parts.

Introduction

The organization of this part of ISO/IEC 19075 is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ISO/IEC 19075.
2) Clause 2, “Normative references”, identifies additional standards that, through reference in this part of ISO/IEC 19075, constitute provisions of this part of ISO/IEC 19075.
3) Clause 3, “JavaScript Object Notation (JSON)”, introduces what is JSON.
4) Clause 4, “The SQL/JSON data model”, introduces the data model that is used by the SQL/JSON functions and the SQL/JSON path language.
5) Clause 5, “SQL/JSON functions”, introduces the SQL/JSON functions to query and construct JSON.
6) Clause 6, “SQL/JSON path language”, introduces the SQL/JSON path language.

Information technology — Database languages — SQL Technical Reports —

Part 6:
SQL support for JavaScript Object Notation (JSON)

1 Scope

This Technical Report describes the support in SQL for JavaScript Object Notation.

This Technical Report discusses the following features of the SQL language:

— Storing JSON data.

— Publishing JSON data.

— Querying JSON data.

— SQL/JSON data model and path language.

(Blank page)

2 Normative references

The following referenced documents are indispensable for the application 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.

2.1 ISO and IEC standards

[ISO9075-2] ISO/IEC 9075-2:2016, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

2.2 Other international standards

[ECMAscript] ISO/IEC 16262:2011, Information technology — Programming languages, their environments and system software interfaces — ECMAScript language specification; also available as *ECMAScript Language Specification*,
`http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf`

[Unicode] The Unicode Standard,
`http://unicode.org`

[RFC7159] Internet Engineering Task Force, RFC 7159, *The JavaScript Object Notation (JSON) Data Interchange Format*, March 2014,
`https://tools.ietf.org/rfc/rfc7159.txt`

(Blank page)

3 JavaScript Object Notation (JSON)

3.1 What is JSON?

JSON (an acronym for “JavaScript Object Notation”) is both a notation (that is, a syntax) for representing data and an implied data model. JSON is not an object-oriented data model; that is, it does not define sets of classes and methods, type inheritance, or data abstraction. Instead, JSON “objects” are simple data structures, including arrays. [RFC7159] says that JSON is a text format for the serialization of structured data. Its initial intended use was as a data transfer syntax.

The complete syntax of JSON is specified in [RFC7159].

The first-class components of the JSON data model are JSON values. A JSON value is one of the following: JSON object, JSON array, JSON string, JSON number, or one of the JSON literals: true, false, and null. A JSON object is zero or more name-value pairs and is enclosed in curly braces — `{ }`. A JSON array is an ordered sequence of zero or more values and is enclosed in square brackets — `[ ]`.

Here is an example of a JSON object:

```
{ "Name" : "Isaac Newton",
  "Weight" : 80,
  "Famous" : true,
  "Phone" : null }
```

The name-value pairs are separated by commas, and the names are separated from the values by colons. The names are always strings and are enclosed in (double) quotation marks.

Here is an example of a JSON array:

```
[ "Robert J. Oppenheimer", 98, false, "Beechwood 45789" ]
```

In a JSON array, the values are separated by commas. JSON arrays and objects are fully nestable. That is, values in both JSON objects and JSON arrays may be JSON strings, JSON numbers, JSON Booleans (represented by the JSON literals true and false), JSON nulls (represented by the JSON literal null), JSON objects, or JSON arrays.

JSON can be used to represent associative arrays — arrays whose elements are addressed by content, not by position. An associative array can be represented in JSON as a JSON object whose members are name-value pairs; the name is used as the “index” into the “array” — that is, to locate the appropriate member in the object — and the value is used as the content of the appropriate member. Here is such an associative array:

```
{ "Isaac Newton" : "apple harvester" ,
  "Robert J. Oppenheimer": "security risk" ,
  "Albert Einstein" : "patent clerk" ,
  "Stephen Hawking" : "inspiration" }
```

An extremely important part of JSON’s design is that it is inherently schema-less. Any JSON object can be modified by adding new name-value pairs, even with names that were never considered when the object was initially created or designed. Similarly, any JSON array can be modified by changing the number of values in the array.

3.2 Representations of JSON data

Before delving much deeper into the primary topic of this Technical Report, readers should understand that JSON data can be represented in several widely-acknowledged and -used forms. The most obvious and most easily recognizable is its "character string" representation, in which JSON data is represented in Unicode characters as plain text. More specifically, explicit characters such as the left square brace, comma, right curly brace, quotation mark, and letters and digits are all used in their native Unicode representation (UTF-8, UTF-16, UTF-32).

However, for a variety of reasons, JSON data is sometimes stored and exchanged in one of several binary representations. For example, a binary representation of JSON data may be significantly smaller (fewer octets) than the character representation of the same data, so a reduction in network bandwidth and or storage media can be achieved by transferring or storing that data in a binary representation.

Readers should note that there are no published standards, either from traditional *de jure* standards organizations nor from consortia or other *de facto* standards groups, for any binary representations of JSON. The two described in this Technical Report are frequently used and may be representative of binary JSON representations generally. The following discussion is intended only to illustrate the use of and issues with binary serializations of JSON. The SQL standard leaves it implementation-defined whether or not such representations are supported in any particular implementation.

3.2.1 Avro

Avro [Avro] is described as a "data serialization system". As such, its use is not limited to a binary representation or as a compression representation of JSON data. However, a number of JSON environments have chosen Avro as their preferred binary, compressed representation.

Avro has a number of important characteristics that affect its choice as a JSON representation.

— Data is represented in a variable-length, "shortest" form; *e.g.*, the integer 2 and the integer 2000 occupy a different number of octets in an Avro string.

— Numbers are represented using a rather arcane "zig-zag" encoding (this notation "moves" the sign bit from its normal position as the first bit to the last bit; doing so permits removing leading zeros from the numbers, thus making their representation occupy fewer octets).

— There is not a one-to-one mapping between JSON atomic types and Avro atomic types.

— Avro data is always associated with an Avro schema. An Avro schema describes the physical structure of the corresponding Avro data and is needed merely to "unpack" Avro data because of the variable-length fields and the various encoding rules. An Avro schema may accompany each individual Avro data string, or it may be specified separately and applied to all Avro data strings in, say, an Avro data file. Avro schemas tell almost nothing about the data other than how it is packed into a data string.

— Avro strings can be encoded using JSON notation (which sort of contradicts its use as a different representation for JSON data) or using a binary notation.

Readers should recognize that Avro is not a different kind of data at all. It is, instead, merely another way of representing the same data that the JSON character string format represents. (It should be noted that not all possible Avro strings can be treated as JSON data; similarly, not every character string is a valid bit of JSON data.) In this Technical Report, Avro is referenced as one possible *serialization* of JSON data; the character string format is another serialization of the same data.

3.2.2 BSON

BSON [BSON] (variously pronounced as though it were spelled "bison" or as "beeson") is another data serialization system. [BSON] says that it is "a binary format in which zero or more key/value pairs are stored as a single entity," called a "document." BSON is used by — and apparently was created for — a specific commercial product.

BSON strings are no less difficult for human beings to read than Avro strings are, but the design of BSON is significantly different than Avro's. A BSON document is roughly a sequence of elements, each of which is introduced by a single-octet code (for example, the hexadecimal value '01' identifies the element as a double precision floating point value, which is always eight octets in length, and '0D' identifies the element as a string containing JavaScript code), followed by an optional element name, followed by the (mandatory) element value.

BSON, like Avro, is not a different kind of data, but merely provides yet another way of representing JSON data. (Also like Avro, not all BSON strings represent valid JSON data.)

3.3 Schemas

3.3.1 JSON schemata and validity

Neither [RFC7159] nor [ECMAscript] provide any mechanism by which JSON values can be tested for validity other than strict syntactic validity.

JSON text is sufficiently "self-describing" that data encoded in JSON is easily parsed and can often be used in application components that have no specific knowledge of the data contained therein. This last fact explains why JSON is so successful in the broader data management community, in spite of the lack of a standard way to document its structure.

There is at least one effort to define a schema for JSON [JSONschema] that would describe the structure of JSON data to be considered "valid" for some given application. However, there does not appear to be any significant amount of interest from the JSON community for the rapid development of a schema definition language for JSON.

While a schema language for JSON data could be useful in some circumstances, it does not appear that such a language is widely used, and [ISO9075-2] neither relies on nor creates such a language.

[ISO9075-2] uses the word "valid" to describe data instances that satisfy all JSON syntactic requirements. [ISO9075-2] specifies a new SQL predicate, IS JSON, to test the (syntactic) validity of JSON data instances.

3.3.2 Avro schemata

Because Avro is a representation in which each “field” (a bit of JSON data) occupies no more octets than is required, using the particular encoding method for data of each type, it is not possible to simply index to a specific field in each Avro string. In fact, because of the way that Avro encodes its fields, it is not possible to scan an Avro string and identify the start of the second, third, or twenty-fifth field in that string.

Consequently, Avro specifies that each Avro value be described by an Avro schema. Avro schemata are expressed in the character representation of JSON. The schema that represents an entire Avro string is composed of “smaller” schemata that represent each field in the Avro string. The schema describes each field by its name, data type, and (if not already unambiguous) length. Thus, an application wishing to access the fields in an Avro string must first parse the schema of that string, then use that information to locate the desired fields in the string and to “decode” the field contents into a value of a JSON data type.

Because each JSON text can be of a different size or contain different components, one might wish to provide a different schema for each JSON text...a schema that uniquely describes that text and not (necessarily) any other JSON text. This approach necessarily creates an increase in size of the JSON texts associated with those schemata. For small JSON texts and/or JSON texts with a great many fields, the overhead (in octets) of providing a schema for each such text in the Avro representation becomes unacceptable very quickly. For very large JSON texts, particularly with small numbers of (very large) fields, the presence of a per-text schema may be perfectly reasonable.

Avro does not require that a schema be provided along with each JSON text, individually. It does permit that approach, but it also allows for a single schema to describe all of the Avro-represented JSON texts in a “container file.” As long as all of the texts in that container file are sufficiently alike, a single schema is adequate — and, of course, produces much-reduced overhead. In the context of this Technical Report, an entire column containing the Avro representation of JSON texts acts as the “container file.” The Avro schema associated with such a column is part of the metadata describing the column.

In the preceding paragraph, the phrase “sufficiently alike” was used. That phrase means that each object/array in the rows of a column contain the same number of members/elements, each having the same name (for objects) and data type. It also means that the lengths (number of octets) of each member/element must be respectively the same, and that’s not always easy to ensure.

NOTE 2 — Therefore, the JSON objects “**{ "name" : "Joe", "salary" : 85000 }**” and “**{ "name" : "Bob", "salary" : 78000 }**” are “sufficiently alike”, but “**{ "name" : "Ann", "bonus" : 85000 }**” is not.

3.3.3 BSON schemata

Because the BSON representation of JSON contains a one-octet code as the first octet of every field, it is possible to scan a BSON value and uniquely identify each field and its data type. Consequently, no sort of schema is required for BSON-represented JSON data.

However, BSON — like Avro — uses variable-length fields, so that corresponding members/elements in different objects/arrays can have significantly different lengths. Scanning BSON strings to locate those field codes (and thus the fields themselves) requires CPU cycles. BSON might benefit from having a schema capability somewhat similar to Avro’s, but it is certainly not necessary.

3.4 Why does JSON matter in the context of SQL? What is JSON's relationship to NoSQL?

It is unclear that JSON and SQL [ISO9075-2] have any inherent relationship, but it is equally clear that the technical, business, and government worlds are increasingly using both kinds of data in their environments. Individual applications are required to access and manipulate all of these kinds of data, often concurrently. There are great benefits when a single data management system can concurrently handle all of the data. Among the benefits are: reduced administrative costs, improved security and transaction management, better performance and greater optimizability, better resource allocation, and increased developer productivity.

Incorporation of JSON into the SQL umbrella offers implementers and users alike the benefits described above. That fact easily justifies the relatively small increase in size and complexity of the SQL standard, especially when the approach described in this Technical Report is used.

NoSQL database systems [NoSQLDB] are generally characterized by the following attributes:

— They do not use the relational model (they also do not use a number of other data models).

— They tend to be focused on "big data" and on applications for which "approximate" answers are often adequate.

— They are optimized for data retrieval, not for data creation or update, nor on the relationships between data.

— They usually do not use ACID [ACIDtxns] transactions; instead, they may offer transactional models that result in "eventual consistency".

— They tend to be designed using distributed, fault-tolerant, highly parallel hardware and software architectures.

NoSQL database systems come in a very wide variety of kinds, based on their targeted marketplaces, data models, and application requirements. They have been crudely taxonomized into:

— Key-value stores

— "BigTable" stores

— Document stores

— Graph databases

Key-value stores provide exactly the capability that the name implies: applications provide a key and are given a value in return. Key-value stores may manage only "flat" pairs, or they may manage hierarchical pairs.

BigTable stores implement multi-dimensional arrays such that applications provide one or more index values (often strings used as key values, instead of numeric indexes) that together provide the location of a cell, the value of which is returned to the application.

Document stores, contrary to what the name many suggest to many, do not necessarily store textual documents such as books, specifications, or magazines; instead, they store data that may be traditional textual documents or organized collections of structured (and semi-structured) data.

Graph databases provide a way to store data that is generally linked together into graphs (often directed graphs, sometimes trees in particular).

Many, but hardly all, NoSQL database systems manage data represented in JSON, especially key-value stores and document stores.

The rapidly increasing use of JSON to interchange data between Web applications has caught the attention of academics, technologists, developers, and even enterprise management. SQL database implementers are increasingly convinced that they must support JSON data "natively". This Technical Report describes the approach taken in [ISO9075-2], which allows such implementers to provide that support in a common manner.

3.5 JSON terminology

JSON is taken from JavaScript, which has been standardized under the name "ECMAScript" [ECMAscript]. [ECMAscript] defines the terminology for its objects, but [RFC7159] uses terminology that is significantly different. Other specifications associated with JSON use still different terminology.

[ISO9075-2] and this Technical Report stick as closely as possible to the notation in [RFC7159]. The following terms and their definitions are used:

Table 1 — Terms and definitions

Term	**Definition**
JSON text	A sequence of tokens, which must be encoded in Unicode [Unicode] (UTF-8 by default); insignificant white space may be used anywhere in JSON text except within strings (where all white space is significant), numbers, and literals — note that JSON text is a single object or array
Token	One of six structural characters ("{", "}", "[", "]", ":", ","), strings, numbers, and literals
Value	An object, array, number, string, or one of three literals
Type	A primitive type or a structured type
Primitive type	A string, a number, a Boolean, or a null
Primitive value	A value that is a string, number, Boolean, or null
Structured type	An object or an array
Structured value	A value that is an object or an array
String	A sequence of Unicode characters; some characters must be "escaped" by preceding them with a reverse solidus ("\"), while any or all characters can be represented in "Unicode notation" comprising the string "\u" followed by four hexadecimal digits or two such strings representing the UTF-16 surrogate pairs representing characters not on the Basic Multilingual Plane (strings are surrounded by double-quote characters, which are not part of the value of the strings)
Number	A sequence comprising an integer part, optionally followed by a fractional part and/or an exponent part (non-numeric values, such as infinity and NaN are not permitted)
Boolean	The literal "true" or the literal "false"

Term	Definition
Null	The literal “null”
Object	A structure represented by a “{”, zero or more members separated by “,”, and “}”
Member	A string followed by a colon followed by a value in an object (a member is also known as a “name-value pair”; the name is sometimes called a “key” and the second value is sometimes called a “bound value”)
Array	A structure represented by a “[”, zero or more elements separated by “,”, and “]”
Element	A value in an array
Field	A member in an object, an element in an array, a name in a member, or a value in a member
Data model (general)	A definition of what kinds of data belong to a particular universe of discourse, including the operations on those kinds of data
JSON data model	The (implicit) data model associated with JSON
SQL/JSON data model	The data model created for operating on JSON data within the SQL language

3.6 Use cases for JSON support in SQL

There are three primary use cases:

— JSON data ingestion and storage

— JSON data generation from relational data

— Querying JSON as persistent semi-structured data model instances

The following sections discuss these use cases in greater detail.

3.6.1 JSON data ingestion and storage

Defining a new native data type is both costly to implement for SQLimplementations and costly to adopt by database tools and applications alike. Thus, [ISO9075-2] took the approach to ingest JSON data as character strings or binary strings that are then stored in ordinary SQL columns of some existing string type. When such data is retrieved from those columns for use in JSON-based functions, it is transformed (parsed) into instances of an internal SQL/JSON data model that is never directly exposed to the application author.

3.6.2 JSON data generation from relational data

This use case asks “How can JSON data instances be (declaratively) generated from relational tables for data export?” Applications that are based on JSON data not only want to store and retrieve such data upon demand, but they typically want their queries against such data to provide results in the same form — JSON. Although it is trivial to provide procedural mechanisms by which applications can laboriously (and with many likely errors) construct JSON data, SQL’s declarative nature suggests that JSON objects and arrays should be generated instead of potentially lengthy character strings that represent such objects and arrays. (Readers are cautioned not to misinterpret this use case as requiring provision of a “bulk JSON data export” facility.)

[ISO9075-2] addresses this use case by providing several functions that transform the data stored in SQL tables into instances of the internal SQL/JSON data model that can, if needed, be serialized back into character string form. This Technical Report provides several examples for those functions.

3.6.3 Querying JSON as persistent semi-structured data model instances

This use case explores how JSON data that is stored directly in SQL tables can be queried. Direct mapping of entire SQL tables into single (or, necessarily, multiple) JSON objects or arrays has not been specified in [ISO9075-2], although support is provided for such mappings when needed by applications. Instead, JSON data is stored within an opaque data type (specifically an SQL string or Large Object) that can be manipulated through the functional interface specified in [ISO9075-2], as illustrated by examples in this Technical Report.

3.7 What features address those use cases?

The use cases are addressed as follows:

— SQL query expressions can access JSON data according to its structure (*e.g.*, using the names of key-value pairs in JSON objects, positions in JSON arrays, *etc.*).

— SQL queries can generate JSON data directly for return to invokers of those queries.

— SQL tables can store JSON data persistently.

In the next sections, each of these “macro-features” are explored in turn.

3.7.1 Storing JSON data in an SQL table

The approach taken by [ISO9075-2] and described in this Technical Report is to store JSON data into character string columns or binary string columns that are defined within ordinary SQL tables. That permits those JSON data to participate in SQL queries (and, importantly, SQL-transactions) in the same manner as the data stored in other columns of the same tables. By choosing to use columns whose declared types are string types, the standardization (and implementation) overhead of creating a new built-in SQL data type is avoided without losing any significant advantages of a built-in type.

Applications, however, are not expected to provide detailed code to manipulate JSON data in those strings directly in the form of character string operations. [ISO9075-2] provides a number of built-in SQL functions that access (query) JSON data stored in such columns. These functions are described in Clause 5, “SQL/JSON functions”.

3.7.2 Generating JSON in an SQL query

[ISO9075-2] provides built-in functions that generate JSON objects and JSON arrays as the results of SQL queries, whether the source of the data queried is JSON data or ordinary SQL data. These functions are also described in Clause 5, “SQL/JSON functions”.

3.7.3 Querying JSON data in SQL tables using SQL

Since there is no well-known, standardized, and universally accepted language that queries data represented in the JSON data model, [ISO9075-2] defines the SQL/JSON path language, which is embedded in SQL operators that address the use cases outlined in Subclause 3.6, “Use cases for JSON support in SQL”.

(Blank page)

4 The SQL/JSON data model

The SQL/JSON data model used by SQL/JSON path language can be summarized as “sequences of items”. The items are SQL scalar values with an additional SQL/JSON null value, and composite data structures using JSON arrays and objects.

To clearly distinguish between JSON values “outside” the SQL-environment and their analogs “inside” the SQL-environment, the following conventions are adopted:

— The modifier “JSON” refers to constructs within a character or binary string that conforms to [RFC7159].

— The modifier “SQL/JSON” refers to JSON constructs within the SQL-environment.

The relationship between “JSON” outside of and “SQL/JSON” within the SQL-environment is crudely illustrated in Figure 1, “Relationships between “JSON” and “SQL/JSON””.

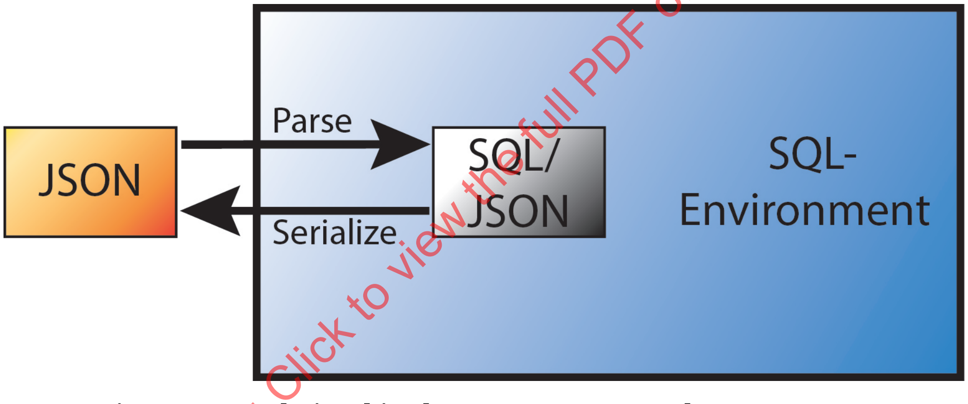

Figure 1 — Relationships between “JSON” and “SQL/JSON”

JSON data is parsed into SQL/JSON values, which can then be serialized back into JSON. All JSON data can be parsed into SQL/JSON values; however, not all SQL/JSON values can be serialized into JSON data. Parsing and serializing are addressed in more detail in Subclause 4.3, “Parsing JSON”, and Subclause 4.4, “Serializing JSON”, respectively.

Table 2, “JSON, SQL/JSON, and SQL (other than SQL/JSON values and counterparts)”, lists various terms used in [ISO9075-2] and this Technical Report related to JSON, SQL/JSON, and SQL (other than SQL/JSON values and their corresponding counterparts).

Table 2 — JSON, SQL/JSON, and SQL (other than SQL/JSON values and counterparts)

JSON	SQL/JSON	SQL (other than SQL/JSON)
JSON array	SQL/JSON array	
JSON object	SQL/JSON object	

JSON	SQL/JSON	SQL (other than SQL/JSON)
JSON member	SQL/JSON member	
JSON literal `null`	SQL/JSON null	typed nulls
JSON literal `true`	*True*	*True*
JSON literal `false`	*False*	*False*
JSON number	number	non-null number
JSON string	character string	non-null character string
	datetime	non-null datetime
	SQL/JSON item	
	SQL/JSON sequence	

NOTE 3 — SQL/JSON and SQL other than SQL/JSON have precisely the same value spaces in the case of non-null scalars of boolean, numeric, string, or datetime types.

4.1 SQL/JSON items

An SQL/JSON item is defined recursively as any of the following:

1) An SQL/JSON scalar, defined as a non-null value of any of the following predefined (SQL) types: character string with character set Unicode, numeric, boolean, or datetime.

2) An SQL/JSON null, defined as a value that is distinct from any value of any SQL type.

   NOTE 4 — An SQL/JSON null is distinct from the null value of any SQL type.

3) An SQL/JSON array, defined as an ordered list of zero or more SQL/JSON items, called the SQL/JSON elements of the SQL/JSON array.

4) An SQL/JSON object, defined as an unordered collection of zero or more SQL/JSON members, where an SQL/JSON member is a pair whose first value is a character string with character set Unicode and whose second value is an SQL/ JSON item. The first value of an SQL/JSON member is called the *key* and the second value is called the *bound value*.

   NOTE 5 — [RFC7159], section 2.2, “Objects”, says “The names within an object SHOULD be unique”. Thus, non-unique keys are permitted but not advised. The user may use the WITH UNIQUE KEYS clause in the <JSON predicate> to check for uniqueness if desired.

Two SQL/JSON items are comparable if one of them is an SQL/JSON null, or if both are in one of these types: character string, numeric, boolean, DATE, TIME, TIMESTAMP.

Two SQL/JSON items *SJI1* and *SJI2* are said to be equivalent, defined recursively as follows:

1) If *SJI1* and *SJI2* are non-null values of a predefined type, then *SJI1* and *SJI2* are equivalent if they are equal.

2) If *SJI1* and *SJI2* are the SQL/JSON null, then *SJI1* and *SJI2* are equivalent.

3) If *SJI1* and *SJI2* are both SQL/JSON arrays of the same length *N* and corresponding elements of *SJI1* and *SJI2* are equivalent, then *SJI1* and *SJI2* are equivalent.

4) If *SJI1* and *SJI2* are SQL/JSON objects with the same number of members, and there exists a bijection *B* from *SJI1* to *SJI2* mapping each SQL/JSON member *M* of *SJI1* to an SQL/JSON member *B*(*M*) of *SJI2* such that the key and bound value of *M* are equivalent to the key and bound value of *B*(*M*), respectively, for all members *M* of *SJI1* then SJI1 and SJI2 are equivalent.

An SQL/JSON sequence is an ordered list of zero or more SQL/JSON items.

NOTE 6 — There is no SQL <data type> whose value space is SQL/JSON items or SQL/JSON sequences.

SQL/JSON items are typed values in the following categories:

1) Atomic values:

   Atomic values are the non-null SQL values of the following types:

   a) Strings in a Unicode character set.

   b) Numeric values:

      i) Exact numeric values.

      ii) Approximate numeric values.

   c) Boolean values.

   d) Datetime values.

   e) The SQL/JSON null value, which is distinct from all SQL values, including the SQL null.

2) SQL/JSON arrays.

3) SQL/JSON objects.

These categories are discussed in the following subsections.

4.1.1 Atomic values

Atomic values in the SQL/JSON data model are virtually a subset of the values of <predefined type> in [ISO9075-2], Subclause 6.1, “<data type>”. To recap the choices in Foundation:

```
<predefined type> ::=
    <character string type> [ CHARACTER SET <character set specification> ]
        [ <collate clause> ]
  | <national character string type> [ <collate clause> ]
  | <binary string type>
  | <numeric type>
  | <boolean type>
  | <datetime type>
  | <interval type>
```

The SQL/JSON data model differs from Foundation predefined types as follows:

1) The data model does not support <binary string type> and <interval type>.

2) Only character strings of Unicode characters are supported. The only collation is the binary (codepoint) collation of Unicode.

3) The SQL/JSON null value is regarded as the sole value of its own type. That is, there is no null character string value, null numeric value, *etc*. In addition, JavaScript semantics are used for nulls in comparisons — the SQL/JSON null is equal to the SQL/JSON null. (See [ECMAscript], section 11.9.6, “The strict equality comparison algorithm”, and similar sections.)

In general, operations in the path language operate on atomic values in the SQL/JSON data model with the same semantics as the corresponding operation in SQL, but note that null semantics follows [ECMAscript].

Datetimes have no serialized representation in JSON. They are part of the SQL/JSON data model to support comparison predicates after converting JSON strings to datetime.

4.1.2 SQL/JSON arrays

An SQL/JSON array is an ordered list of zero or more SQL/JSON items in the SQL/JSON data model. When serialized, the list is separated by commas and enclosed in square brackets, for example:

```
[ 2.3, "bye bye" ]
```

SQL arrays are 1-relative, whereas [ECMAscript] arrays are 0-relative. SQL/JSON arrays are also 0-relative. This allows some path expression to be used in either the SQL/JSON path language or in JavaScript.

4.1.3 SQL/JSON objects

An SQL-JSON object is an unordered set of zero or more members.

NOTE 7 — Empty objects are permitted; see [RFC7159], section 2.2, “Objects”, and [ECMAscript], section 15.12.1.2, “The JSON syntactic grammar”.

A member is a pair of values:

1) The first value is a character string and is called the key of the member.

2) The second value is any SQL/JSON item in the SQL/JSON data model and is called the bound value of the member.

An SQL/JSON object is serialized enclosed in curly braces, with the members listed in a non-deterministic order separated by commas. Each member is serialized as its key (a character string, therefore enclosed in double quotes), a colon, and then the serialization of the (second) value of the member. For example:

```
{ "name": "Fido", "tag": 12345 }
```

4.2 SQL/JSON sequences

An SQL/JSON sequence is an ordered list of zero or more SQL/JSON items. SQL/JSON sequences do not nest; they can only be concatenated. One may think of an SQL/JSON item as equivalent to an SQL/JSON sequence of that one item, which is an acceptable mental model, but this Technical Report endeavors to always view an SQL/JSON sequence as a container of zero or more SQL/JSON items.

4.3 Parsing JSON

Parsing refers to the process of importing from some storage format into the SQL/JSON data model. The storage format may be JSON text stored in a Unicode character string, or it may be some binary format such as AVRO or BSON. Since no specifications of AVRO and BSON are formally referenceable, they have been left as implementation extensions in [ISO9075-2].

The conversion from JSON is straightforward, because SQL/JSON data model is basically a superset of JSON. In particular:

Table 3 — Parallels between JSON text and SQL/JSON data model

JSON text	**SQL/JSON data model**
true	*true*
false	*false*
null	null
string, e.g., "hello dolly"	string, e.g., 'hello dolly'
number	The format for numbers in [RFC7159] is the same as numeric literals in SQL. Consequently, a number from JSON can be transferred into SQL by simply parsing it as a <signed numeric literal>.
array	SQL/JSON array
object	SQL/JSON object

4.4 Serializing JSON

Serializing JSON refers to the process of exporting a value from the SQL/JSON data model back to some storage format. Serialization is specified only to JSON text; conversion to some other format, such as AVRO or BSON, is left as an implementation-defined extension in [ISO9075-2].

SQL/JSON datetimes cannot be serialized; neither can SQL/JSON sequences of length greater than one.

The result of serialization is an implementation-dependent string of Unicode characters that, if parsed, would restore the original value in the SQL/JSON data model.

The precise result of serialization is implementation-dependent because:

— Meaningless whitespace is not specified.

— Because of escape sequences, there are multiple ways to serialize a character string.

— [ECMAscript], Sections 5.1.3, “The numeric string grammar”, and 9.3.1, “ToNumber applied to the string type”, provide some discretion in the formatting of numbers.

— Members of an SQL/JSON object are unordered, so there are many possible permutations of the members of an object.

5 SQL/JSON functions

In [ISO9075-2], operations on JSON data are generally performed using a set of built-in functions specified explicitly for that purpose. These functions are called “SQL/JSON functions”. [ISO9075-2] also specifies some supporting SQL syntax, such as the IS JSON predicate.

5.1 Handle JSON using built-in functions

SQL/JSON functions are partitioned into two groups: constructor functions (JSON_OBJECT, JSON_OBJECTAGG, JSON_ARRAY, and JSON_ARRAYAGG) and query functions (JSON_VALUE, JSON_TABLE, JSON_EXISTS, and JSON_QUERY). Constructor functions use values of SQL types and produce JSON values (JSON objects or JSON arrays) represented in SQL character or binary string types. Query functions evaluate SQL/JSON path language expressions against JSON values, producing values of SQL/JSON types, which are converted to SQL types. The SQL/JSON path language is described in Clause 6, “SQL/JSON path language”.

5.2 JSON API common syntax

The SQL/JSON query functions all need a path specification, the JSON value to be input to that path specification for querying and processing, and optional parameter values passed to the path specification. They use a common syntax:

```
<JSON API common syntax> ::=
  <JSON context item> <comma> <JSON path specification> [ AS <JSON table path name> ]
      [ <JSON passing clause> ]

<JSON context item> ::=
  <JSON value expression>

<JSON path specification> ::=
  <character string literal>

<JSON passing clause> ::=
  PASSING <JSON argument> [ { <comma> <JSON argument> } ]

<JSON argument> ::=
  <JSON value expression> AS <identifier>
```

The type of the <value expression> contained in the <JSON value expression> immediately contained in the <JSON context item> is a string type. If the <JSON context item> does not implicitly or explicitly specify a <JSON input clause>, then FORMAT JSON is implicit.

If <JSON API common syntax> is not contained in <JSON table>, then <JSON table path name> is not required. The <JSON table path name> is optional; if <JSON table path name> is not specified, then an implementation-dependent <JSON table path name> is implicit. The <JSON context item> is the JSON input on which the SQL/JSON path expression operates.

5.2.1 JSON value expression

<JSON context item> is defined as a <JSON value expression>, which is in turn defined as:

```
<JSON value expression> ::=
  <value expression> [ <JSON input clause> ]

<JSON input clause> ::=
  FORMAT <JSON representation>

<JSON representation> ::=
    JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
  | <implementation-defined JSON representation option>
```

A <JSON value expression> may have an optional <JSON input clause>. This indicates that the values expression should be parsed as JSON. The standardized option is FORMAT JSON; implementations may also support syntax such as FORMAT AVRO or FORMAT BSON. When using the <JSON input clause>, the <value expression> may be either a character string or a binary string.

If the user does not specify the ENCODING for a <JSON value expression>, then the SQL-implementation will determine it as one of the three encoding alternatives: UTF8, UTF16, and UTF32.

5.2.2 Path expression

The context item is followed by a comma and the SQL/JSON path expression. <JSON path specification> is a character string literal. Requiring a literal enables the implementation to optimize the query by analyzing the SQL/JSON path expression and planning accordingly, for example, if there are indexes available on the JSON data.

The optional <JSON table path name> is syntax that is only used in JSON_TABLE, and then only if the user wants to write an explicit plan for processing nested COLUMNS clauses. Explicit plans for JSON_TABLE are explained later.

5.2.3 PASSING clause

SQL's JSON facilities include a PASSING clause that is used to pass parameters to the SQL/JSON path expression. For example, suppose that an SQL/PSM routine has parameters “upper” and “lower” and it is desired to find rows in which the JSON data has a member called “age” between these two values. The user might write:

```
SELECT *
FROM T
WHERE JSON_EXISTS (T.C,
   'lax $ ? ($lo <= @.age && @.age <= $up)'
   PASSING upper AS "up",
           lower AS "lo")
```

In this example there are two variables defined in the PASSING clause. In “SQL-land” they are the PSM variables `upper` and `lower`; in “SQL/JSON-land” they are `$up` and `$lo`.

Syntactically, the PASSING clause is a comma-separated list of <JSON argument>s, defined as:

```
<JSON argument> ::=
  <JSON value expression> AS <identifier>
```

The <JSON value expression> specifies the value to be passed into the SQL/JSON path engine. This may be of any type supported by the data model (Unicode character strings, numbers, booleans and datetimes) or any other type that can be cast to a Unicode character string (for example, almost all non-Unicode character strings, or user-defined types with a user-defined cast to a Unicode character string type). In the example above, the <value expression>s were SQL/PSM variables, but any <value expression> may be used. Thus, a join to another table can be constructed by passing a column from one table into path expression on an SQL/JSON value of another table.

The <identifier> in the <JSON argument> specifies the variable name by which the value can be referenced within the SQL/JSON path expression. In this example, these identifiers are “up” and “lo”. Within the path expression, these are referenced with a prefixed dollar sign (`$up` and `$lo`), since $ marks the variables in a path expression.

5.2.4 JSON output clause

Whenever JSON data is returned as the result of a function, the application author will normally wish to control the form in which that JSON data is returned. The syntax used to specify the data type, format, and encoding of the JSON text created by a JSON-returning function.

```
<JSON output clause> ::=
  RETURNING <data type>
      [ FORMAT <JSON representation> ]

<JSON representation> ::=
    JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
  | <implementation-defined JSON representation option>
```

If FORMAT is not specified, then FORMAT JSON is implicit. If the <JSON output clause> specifies or implies JSON, then the <data type> shall identify a string type. FORMAT JSON specifies the data format specified in [RFC7159]. FORMAT <implementation-defined JSON representation option> specifies an implementation-defined data format.

NOTE 8 — For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON representation option> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of “Parsing a JSON text”, and “Serializing an SQL/JSON item”, respectively.

5.2.5 ON ERROR and ON EMPTY syntax

The SQL/JSON query functions also have ON ERROR and ON EMPTY clauses in common. However, the details of these clauses vary depending on the query function, so they are not included in the <JSON API common syntax> shown above. They will be described individually for each query function.

5.3 Query functions

The SQL/JSON query functions are:

— JSON_EXISTS — determine whether an SQL/JSON value satisfies a path specification.

— JSON_VALUE — extract an SQL scalar value from an SQL/JSON value.

— JSON_QUERY — extract an SQL/JSON value from an SQL/JSON value.

— JSON_TABLE — extract a table from an SQL/JSON value.

5.3.1 JSON_EXISTS

JSON_EXISTS determines whether a JSON value satisfies a search criterion. The syntax for JSON_EXISTS is:

```
<JSON exists predicate> ::=
  JSON_EXISTS <left paren>
      <JSON API common syntax>
      [ <JSON exists error behavior> ON ERROR ]
  <right paren>

<JSON exists error behavior> ::=
  TRUE | FALSE | UNKNOWN | ERROR
```

The syntax for JSON_EXISTS is the shared syntax <JSON API common syntax> , plus an optional ON ERROR clause, defaulting to FALSE ON ERROR. If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of <JSON exists predicate> is *Unknown*.

JSON_EXISTS is a predicate that can be used to test whether an SQL/JSON path expression is fulfilled in an SQL/JSON value. JSON_EXISTS evaluates the SQL/JSON path expression; the result is *True* if the path expression finds one or more SQL/JSON items.

Consider the following sample data in the following table with two columns, *K* (the primary key) and *J* (a column containing JSON text):

Table 4 — JSON_EXISTS sample data

K	J
101	{ "who": "Fred", "where": "General Products", "friends": [{ "name": "Lili", "rank": 5 }, {"name": "Hank", "rank": 7}] }
102	{ "who": "Tom", "where": "MultiCorp", "friends": [{ "name": "Sharon", "rank": 2}, {"name": "Monty", "rank": 3}] }
103	{ "who": "Jack","friends": [{ "name": "Connie" }] }
104	{ "who": "Joe","friends": [{ "name": "Doris" }, {"rank": 1}] }
105	{ "who": "Mabel", "where": "Black Label","friends": [{ "name": "Buck", "rank": 6}] }

K	J
106	{ "who": "Louise", "where": "Iana" }

Looking at the sample data, one sees that each row contains a JSON object with one or more members. Observe that rows 103 and 104 have no “where” member. To find the rows that have a “where” member, one can write:

```
SELECT T.K
FROM T
WHERE JSON_EXISTS (T.J, 'lax $.where')
```

This is the simplest possible invocation of JSON_EXISTS. The first argument is the context item T.J, a JSON value to be queried. The second argument is the SQL/JSON path expression.

The SQL/JSON path expression begins with the keyword lax, the alternative being strict. The choice of lax or strict governs the behavior in certain error situations to be described later. In this particular example the choice is actually irrelevant.

After the mode declaration (lax or strict) one reaches the essence of the SQL/JSON path expression. In the SQL/JSON path expression, the dollar sign ($) represents the context item. The dot operator (`.`) is used to select a member of an object; in this case the member called “where” is selected.

If a row has a “where” member, then the result of the SQL/JSON path expression is the bound value of that “where” member. Thus, the SQL/JSON path expression returns a non-empty SQL/JSON sequence for rows 101, 102, 105, and 106, and JSON_EXISTS return *True* for these rows.

In this example, the path expression is in lax mode, which means that any “structural” errors are converted to the empty SQL/JSON sequence. A structural error is an attempt to access a non-existent member of an object or element of an array. Rows 103 and 104 have structural errors, because they lack the “where” member. In lax mode, such structural errors are handled by returning an empty SQL/JSON sequence. (The alternative, strict mode, treats a structural error as a “hard” error and returns that error to the invoking routine. Strict mode will be considered later.) On rows 103 and 104, the result of the SQL/JSON path expression is the empty SQL/JSON sequence, and for these rows JSON_EXISTS returns *False*.

Thus, the result of the sample query is in the following table:

Table 5 — Result of the sample query

K
101
102
105
106

Alternatively, the query could be run in strict mode:

```
SELECT T.K
```

```
FROM T
WHERE JSON_EXISTS (T.J, 'strict $.where')
```

In this case, rows 103 and 104 will have errors in the path expression, which might be presented to the user as exceptions. Users want the ability to handle exceptions so that the query can run to completion, rather than halting on an exception. Thus, ON ERROR clause is provided in each of the four query operators. For example, the user might write:

```
SELECT T.K
FROM T
WHERE JSON_EXISTS(T.J, 'strict $.where' FALSE ON ERROR)
```

Here, FALSE ON ERROR means that the result of JSON_EXISTS should be *False* if there is an error. The results of this query will therefore be as shown in Table 5, "Result of the sample query", matching the results of the lax mode example. FALSE ON ERROR is the default error behavior for JSON_EXISTS. Other choices are TRUE ON ERROR, UNKNOWN ON ERROR, or ERROR ON ERROR.

In the examples considered above, the SQL/JSON path expression `$.where` will find either zero or one SQL/JSON item. In general, an SQL/JSON path expression might result in more than one SQL/JSON item. For example, some rows have more than one "rank" member; however, row 103 has no rank member (and row 106 does not even have friends). To search for rows having "rank", one might use:

```
SELECT T.K
FROM T
WHERE JSON_EXISTS (T.J, 'strict $.friends[*].rank')
```

This example shows another accessor in the SQL/JSON path language, `[*]`, which selects all elements of an array. Look at how this SQL/JSON path expression is evaluated in row 101:

Table 6 — Accessor example

	Path step	**Result**
1	$	{ "who": "Fred", "where": "General Products", "friends": [{ "name": "Lili", "rank": 5 }, {"name": "Hank", "rank": 7}] }
2	$.friends	[{ "name": "Lili", "rank": 5 }, { "name": "Hank", "rank": 7}]
3	$.friends[*]	{ "name": "Lili", "rank": 5 }, { "name": "Hank", "rank": 7}
4	$.friends[*].rank	5, 7

Successive lines above show the evaluation of the SQL/JSON path expression $.friends[*].rank. On the first line, $, the value is the entire context item. The next line drills down to the "friends" member, which is an array. The next line drills down to the elements of the array. Notice that at this point the square bracket `[ ]` array wrappers are lost, and the result at this point is an SQL/JSON sequence of length two. The final line drills down to the "rank" member of each SQL/JSON item in the SQL/JSON sequence; the result is again an SQL/JSON sequence of length two.

Note in this example how the accessors in the SQL/JSON path language automatically iterate over all SQL/JSON items discovered by the previous step in the SQL/JSON path expression. In this example, this is seen in the transition from step 3 to step 4.

In some commercial products, it is customary for a member accessor such as `.rank` to automatically iterate over the elements of an array such as “friends”. Thus, the user of such a product would prefer to write `$.friends.rank`, as if there were an implicit `[*]` on “friends”. This convention is supported in lax mode. Thus, the path expression `lax $.friends.rank` finds the same rows as '`strict $.friends[*].rank` in this example. In lax mode, there are effectively two automatic iterations: first, any array in the sequence is unwrapped (as if modified by `[*]`) and then the possibly expanded sequence is iterated over. Precise details will be presented later when the path language is addressed.

The result of the SQL/JSON path expression in this example is 0 (zero), 1 (one), or 2 SQL/JSON items, depending on the row. JSON_EXISTS is *True* if the result is 1 (one) or more SQL/JSON items, *False* if the result is 0 (zero) SQL/JSON items, and governed by the ON ERROR clause if the result is an error.

5.3.2 JSON_VALUE

JSON_VALUE is an operator to extract an SQL scalar from a JSON value. The syntax of JSON_VALUE is

```
<JSON value function> ::=
  JSON_VALUE <left paren>
      <JSON API common syntax>
      [ <JSON returning clause> ]
      [ <JSON value empty behavior> ON EMPTY ]
      [ <JSON value error behavior> ON ERROR ]
  <right paren>

<JSON returning clause> ::=
  RETURNING <data type>

<JSON value empty behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>

<JSON value error behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>
```

<JSON value empty behavior> specifies what to do if the result of the SQL/JSON path expression is empty:

— NULL ON EMPTY means that the result of JSON_VALUE is the null value.

— ERROR ON EMPTY means that an exception is raised.

— DEFAULT <value expression> ON EMPTY means that the <value expression> is evaluated and cast to the target type.

<JSON value error behavior> specifies what to do if there is an unhandled error. Unhandled errors can arise if there is an input conversion error (for example, if the context item cannot be parsed), an error returned by the SQL/JSON path engine, or an output conversion error. The choices are the same as for <JSON value empty behavior>.

When using DEFAULT <value expression> for either the empty or error behavior, what happens if the <value expression> raises an exception? The answer is that an error during empty behavior “falls through” to the error behavior. If the error behavior itself has an error, there is no further recourse but to raise the exception.

If <JSON returning clause> is not specified, then an implementation-defined character string type is implicit. The <data type> contained in the explicit or implicit <JSON returning clause> is a <predefined type> that identifies a character string data type, numeric data type, boolean data type, or datetime data type. The declared type of <JSON value function> is the type specified by <data type>. If <JSON value empty behavior> is not specified, then NULL ON EMPTY is implicit. If <JSON value error behavior> is not specified, then NULL ON ERROR is implicit. If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of <JSON value function> is the null value of type <data type>.

After finding a desired row, the user might wish to extract an SQL scalar value from a JSON value. This is done using JSON_VALUE. For example, to extract the who member from each row:

```
SELECT T.K,
     JSON_VALUE (T.J, 'lax $.who') AS Who
FROM T
```

with the following result from the sample data:

Table 7 — Result 1

K	**WHO**
101	Fred
102	Tom
103	Jack
104	Joe
105	Mabel
106	Louise

Note that JSON_VALUE by default returns an implementation-defined character string type. The user can specify other types using a RETURNING clause, to be considered later.

The “where” member from each row can also be extracted. However, there is no “where” there in rows 103 and 104. This is a structural error when evaluating `$.where`. In lax mode, structural errors are converted to an empty SQL/JSON sequence. For those rows, the user may desire a default value, such as null. This use case is supported using the underlined syntax shown below:

```
SELECT T.K,
     JSON_VALUE (T.J, 'lax $.who') AS Who,
     JSON_VALUE (T.J, 'lax $.where'
     NULL ON EMPTY) AS Nali
FROM T
```

with the following result:

Table 8 — Result 2

K	**WHO**	**NALI**
101	Fred	General Products
102	Tom	MultiCorp
103	Jack	
104	Joe	
105	Mabel	Black Label
106	Louise	Iana

If the query is reformulated in strict mode, then the structural errors become "hard" errors, that is, errors that are reported out of the path engine and back to the API level. To control "hard" errors, JSON_VALUE has an ON ERROR clause. As one possibility of the ON ERROR clause, consider

```
SELECT T.K,
     JSON_VALUE (T.J, 'strict $.who') AS Who,
     JSON_VALUE (T.J, 'strict  $.where'
     DEFAULT 'no where there' ON ERROR )
     AS Nali
FROM T
```

The result of this example would be:

Table 9 — Result 3

K	**WHO**	**NALI**
101	Fred	General Products
102	Tom	MultiCorp
103	Jack	no where there
104	Joe	no where there
105	Mabel	Black Label
106	Louise	Iana

JSON_VALUE expects that the SQL/JSON path expression will return one SQL/JSON item; the ON EMPTY clause can be used to handle missing data (no SQL/JSON items) gracefully. More than one SQL/JSON item is an error. To avoid raising an exception on more than one SQL/JSON item, the ON ERROR clause can be used. For example, some rows have more than one "friend". Consider this query:

```
SELECT T.K,
        JSON_VALUE (T.J, 'lax $.who') AS Who,
        JSON_VALUE (T.J, 'lax $.where' NULL ON EMPTY) AS Nali,
        JSON_VALUE (T.J, 'lax $.friends.name' NULL ON EMPTY
                                            DEFAULT '*** error ***' ON ERROR)
        AS Friend
   FROM T
```

with the following result:

Table 10 — Result 4

K	WHO	NALI	FRIEND
101	Fred	General Products	*** error ***
102	Tom	MultiCorp	*** error ***
103	Jack		Connie
104	Joe		Doris
105	Mabel	Black Label	Buck
106	Louise	Iana	

Rows 101 and 102 have an error because the path expression `$.friends.name` returns more than one SQL/JSON item. Row 106, on the other hand, has no “friends”, so the NULL ON EMPTY clause determines the result.

Row 104 in the preceding example deserves a closer look. Actually the “friends” member in this row is

```
"friends": [ { "name": "Doris" }, {"rank": 1} ]
```

Thus, `$.friends` is an array of two objects. The member accessor `$.friends.name` will iterate over both objects, as if `$.friends[*].name` had been written. The first object has a “name” member, the second one does not. In lax mode, `$.friends.name` will quietly eliminate the SQL/JSON item in which there is no “name”, leaving one SQL/JSON item, and then JSON_VALUE can succeed with the result “Doris” without relying on either an ON EMPTY or ON ERROR clause.

The ON ERROR clause is useful in strict mode, where even structural errors are hard errors. For example, consider the following example, with a small rewrite to specify strict mode:

```
   SELECT T.K,
        JSON_VALUE (T.J, 'strict $.who') AS Who,
        JSON_VALUE (T.J, 'strict $.where' NULL ON EMPTY NULL ON ERROR) AS Nali,
        JSON_VALUE (T.J, 'strict $.friends[*].name' NULL ON EMPTY
                                            DEFAULT '*** error ***' ON ERROR)
        AS Friend
   FROM T
```

then the result changes to the following:

Table 11 — Result 5

K	WHO	NALI	FRIEND
101	Fred	General Products	*** error ***
102	Tom	MultiCorp	*** error ***
103	Jack		Connie
104	Joe		*** error ***
105	Mabel	Black Label	Buck
106	Louise	Iana	

Look especially at row 104. In lax mode, the result in the FRIEND column was “Doris”, because there was only one object in `$.friends[*]` with a “name”. In strict mode, this row has a path error, because `$.friends[*]` has two objects, and one of them has no “name”.

So far the examples have all returned character strings. This is the default; to extract other types, use the RETURNING clause. For example, the rank field is a number. A query to get the rank of the first friend is:

```
SELECT T.K,
     JSON_VALUE (T.J, 'lax $.who') AS Who,
     JSON_VALUE (T.J, 'lax $.friends[0].rank' RETURNING INTEGER NULL ON EMPTY)
     AS Rank
FROM T
```

Note in the underlined syntax the use of the subscript [0] to access the first element of the array. This follows the convention in [ECMAscript], section 5.1, that arrays begin at subscript 0 (zero). This is one instance where it was better to follow the conventions of the JSON community rather than SQL. The example has the following result:

Table 12 — Result 6

K	WHO	RANK
101	Fred	5
102	Tom	2
103	Jack	
104	Joe	
105	Mabel	6
106	Louise	

5.3.3 JSON_QUERY

The JSON_VALUE function can extract a scalar from an SQL/JSON value, but it cannot extract an SQL/JSON array or an SQL/JSON object from an SQL/JSON value. The JSON_QUERY function exists to extract SQL/JSON values from SQL/JSON values.

The syntax for JSON_QUERY is:

```
<JSON query> ::=
  JSON_QUERY <left paren>
      <JSON API common syntax>
      [ <JSON output clause> ]
      [ <JSON query wrapper behavior> WRAPPER ]
      [ <JSON query quotes behavior> QUOTES [ ON SCALAR STRING ] ]
      [ <JSON query empty behavior> ON EMPTY ]
      [ <JSON query error behavior> ON ERROR ]
      <right paren>

<JSON query wrapper behavior> ::=
    WITHOUT [ ARRAY ]
  | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]

<JSON query quotes behavior> ::=
    KEEP
  | OMIT

<JSON query empty behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<JSON query error behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT
```

The ON EMPTY and ON ERROR clauses are similar to JSON_VALUE, and handled essentially the same way. The novel wrinkle is that DEFAULT <value expression> options are not provided; instead, the user can specify an empty array or empty object as the result in the empty or error cases.

If <JSON output clause> is not specified, then RETURNING FORMAT JSON is implicit. The declared type of <JSON query> is the type specified by the <data type> contained in the explicit or implicit <JSON output clause>. If <JSON query empty behavior> is not specified, then NULL ON EMPTY is implicit. If <JSON query error behavior> is not specified, then NULL ON ERROR is implicit. If <JSON query wrapper behavior> is not specified, then WITHOUT ARRAY is implicit. If <JSON query wrapper behavior> specifies WITH, and if neither CONDITIONAL nor UNCONDITIONAL is specified, then UNCONDITIONAL is implicit. If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of <JSON query> is the null value.

Continuing with the example data from Table 4, “JSON_EXISTS sample data”, suppose the user wishes to retrieve the `who`, `where`, and `friends` members. `who` and `where` are scalars and can be extracted with JSON_VALUE, but `friends` is a JSON array, so JSON_QUERY is used:

```
SELECT T.K,
       JSON_VALUE (T.J, 'lax $.who') AS Who,
       JSON_VALUE (T.J, 'lax $.where' NULL ON EMPTY) AS Nali,
       JSON_QUERY (T.J, 'lax $.friends') AS Friends
FROM T
WHERE JSON_EXISTS (T.J, 'lax $.friends')
```

with the following result:

K	WHO	NALI	FRIENDS
101	Fred	General Products	[{ "name": "Lili", "rank" : 5 }] { "name": "Hank", "rank" : 7 }]
102	Tom	MultiCorp	[{ "name": "Sharon", "rank" : 2 }] { "name": "Monty", "rank" : 3 }]
103	Jack		[{ "name": "Connie" }]
104	Joe		[{ "name": "Doris", "rank" : 1 }]
105	Mabel	Black Label	[{ "name": "Buck", "rank" : 6 }]

In row 106, there is no `friends` member, so this row has been suppressed by the WHERE clause.

Now, consider the various special cases that can arise. One possibility is that the SQL/JSON path expression returns an empty SQL/JSON sequence. Similar to JSON_VALUE, the user can use an ON EMPTY clause to handle the empty case. Thus, to handle row 106, one might write:

```
SELECT T.K,
       JSON_VALUE (T.J, 'lax $.who') AS Who,
       JSON_VALUE (T.J, 'lax $.where' NULL ON EMPTY) AS Nali,
       JSON_QUERY (T.J, 'lax $.friends' NULL ON EMPTY) AS Friends
FROM T
```

with this result:

K	WHO	NALI	FRIENDS
101	Fred	General Products	[{ "name": "Lili", "rank" : 5 }] { "name": "Hank", "rank" : 7 }]
102	Tom	MultiCorp	[{ "name": "Sharon", "rank" : 2 }] { "name": "Monty", "rank" : 3 }]
103	Jack		[{ "name": "Connie" }]
104	Joe		[{ "name": "Doris", "rank" : 1 }]
105	Mabel	Black Label	[{ "name": "Buck", "rank" : 6 }]
106	Louise	Iana	

The example explicitly specified NULL ON EMPTY, which is also the default behavior. Other alternatives are ERROR ON EMPTY, EMPTY ARRAY ON EMPTY, and EMPTY OBJECT ON EMPTY. The latter two alternatives return the empty JSON forms “[]” and “{}”, respectively.

A possible error condition is that the SQL/JSON path expression may result in more than one SQL/JSON item, or the result may be a scalar rather than an SQL/JSON array or object. For example, the path expression `$.friends.name` (or `$.friends[*].name` in strict mode) may result in two names in rows 101 and 102. To handle this, the user may request that the results be wrapped in an array wrapper. Here is an example:

```
SELECT T.K,
       JSON_VALUE (T.J, 'lax $.who') AS Who,
       JSON_VALUE (T.J, 'lax $.where' NULL ON EMPTY) AS Nali,
       JSON_QUERY (T.J, 'lax $.friends.name' WITH ARRAY WRAPPER) AS FriendsNames
FROM T
```

with the result:

K	WHO	NALI	FRIENDSNAMES
101	Fred	General Products	["Lili", "Hank"]
102	Tom	MultiCorp	["Sharon", "Monty"]
103	Jack		["Connie"]
104	Joe		["Doris"]
105	Mabel	Black Label	["Buck"]
106	Louise	Iana	[ ]

Once again, row 106 is especially interesting. In this row, the result of the path expression is an empty SQL/JSON sequence. The array wrapper is applied to the empty SQL/JSON sequence, producing an empty array, so there is no need to resort to the NULL ON EMPTY behavior in this case (and in fact the ON EMPTY clause is prohibited if WITH ARRAY WRAPPER is specified).

The alternative to WITH ARRAY WRAPPER is WITHOUT ARRAY WRAPPER, the default shown in the initial examples. Actually, WITH ARRAY WRAPPER comes in two varieties, WITH UNCONDITIONAL ARRAY WRAPPER and WITH CONDITIONAL ARRAY WRAPPER, the default being UNCONDITIONAL. The difference is that CONDITIONAL only supplies the array wrapper if the path expression results in anything other than a singleton SQL/JSON array or object. UNCONDITIONAL always supplies the array wrapper, regardless of what the path expression results are.

What is the difference between JSON_VALUE returning a character string and JSON_QUERY? The difference can be seen with the following example data:

J2
{ a: "[1,2]", b: [1,2], c: "hi"}

Note in this data that members a and c have values that are character strings, whereas member b has a value that is an array. Here are the results of extracting each member, comparing JSON_VALUE with JSON_QUERY, and comparing the different wrapper options:

Table 13 — Comparison of wrapper options

Operator		**$.a**	**$.b**	**$.c**
JSON_VALUE		[1, 2]	error	hi
JSON_-QUERY	WITHOUT ARRAY WRAPPER	error	[1, 2]	error
	WITH UNCONDITIONAL ARRAY WRAPPER	["[1,2]"]	[[1,2]]	["hi"]
	WITH CONDITIONAL ARRAY WRAPPER	["[1,2]"]	[1,2]	["hi"]

There are three error cases in this example. In the case of JSON_VALUE, they will be handled by using the ON ERROR clause, as already discussed. As for JSON_QUERY, the possibilities for ON ERROR are the same as ON EMPTY, namely NULL ON ERROR, EMPTY ARRAY ON ERROR, EMPTY OBJECT ON ERROR, or ERROR ON ERROR.

5.3.4 JSON_TABLE

JSON_TABLE is a function that takes JSON data as input and generates relational data for valid input data. It has three parameters:

1) The JSON value on which to operate.

2) An SQL/JSON path expression to specify zero or more rows.

3) A COLUMNS clause to specify the shape of the output table.

The complete syntax for JSON_TABLE is complex, because of the support for nested COLUMNS clauses. Therefore the syntax will be presented in stages.

Consider the following sample data in a table, BOOKCLUB, that contains a JSON column, JCOL. Table 14, “JSON_TABLE sample data in a book recommendation table” will be used in this section to illustrate some examples of the JSON_TABLE function syntax:

Table 14 — JSON_TABLE sample data in a book recommendation table

ID	JCOL

111	`{ "Name" : "John Smith",`
	`  "address" : { "streetAddress": "21 2nd Street",`
	`                "city": "New York",`
	`                "state" : "NY",`
	`                "postalCode" : 10021 },`
	`  "phoneNumber" : [ { "type" : "home", "number" : "212 555-1234" },`
	`                    { "type" : "fax", "number" : "646 555-4567" } ]`
	`  "books" : [ { "title" : "The Talisman",`
	`                "authorList" : [ "Stephen King", "Peter Straub" ],`
	`                "category" : [ "SciFi", "Novel" ]`
	`              },`
	`              { "title" : "Far from the Madding Crowd",`
	`                "authorList" : [ "Thomas Hardy" ],`
	`                "category" : [ "Novel" ]`
	`              }`
	`            ]`
	`}`

222	`{ "Name" : "Peter Walker",`
	`  "address" : { "streetAddress": "111 Main Street",`
	`                "city": "San Jose",`
	`                "state" : "CA",`
	`                "postalCode" : 95111 },`
	`  "phoneNumber" : [ { "type" : "home", "number" : "408 555-9876" },`
	`                    { "type" : "office", "number" : "650 555-2468" } ]`
	`  "books" : [ { "title":"Good Omens",`
	`                "authorList" : [ "Neil Gaiman", "Terry Pratchett" ],`
	`                "category" : [ "Fantasy", "Novel" ] },`
	`              { "title" : "Smoke and Mirrors",`
	`                "authorList" : [ "Neil Gaiman" ],`
	`                "category" : ["Novel"] } ] }`

333	`{ "Name" : "James Lee" }`

5.3.4.1 COLUMNS clause that is not nested

The elementary case with no nested COLUMNS clause is supported by the following syntax:

```
<JSON table> ::=
  JSON_TABLE <left paren>
      <JSON API common syntax>
      <JSON table columns clause>
      [ <JSON table plan clause> ]
      [ <JSON table error behavior> ON ERROR ]
  <right paren>

<JSON table columns clause> ::=
  COLUMNS <left paren>
      <JSON table column definition>
      [ { <comma> <JSON table column definition> }... ]
      <right paren>

<JSON table column definition> ::=
    <JSON table ordinality column definition>
  | <JSON table regular column definition>
  | <JSON table formatted column definition>
  | <JSON table nested columns>

<JSON table ordinality column definition> ::=
  <column name> FOR ORDINALITY

<JSON table regular column definition> ::=
  <column name> <data type>
      [ PATH <JSON table column path specification> ]
      [ <JSON table column empty behavior> ON EMPTY ]
      [ <JSON table column error behavior> ON ERROR ]

<JSON table column empty behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>

<JSON table column error behavior> ::=
    ERROR
  | NULL
  | DEFAULT <value expression>

<JSON table column path specification> ::=
  <JSON path specification>

<JSON table formatted column definition> ::=
  <column name> <data type>
  FORMAT <JSON representation>
  [ PATH <JSON table column path specification> ]
  [ <JSON table formatted column wrapper behavior> WRAPPER ]
  [ <JSON table formatted column quotes behavior> QUOTES
      [ ON SCALAR STRING ] ]
  [ <JSON table formatted column empty behavior> ON EMPTY ]
  [ <JSON table formatted column error behavior> ON ERROR ]

<JSON table formatted column wrapper behavior> ::=
    WITHOUT [ ARRAY ]
  | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]

<JSON table formatted column quotes behavior> ::=
    KEEP
  | OMIT

<JSON table formatted column empty behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<JSON table formatted column error behavior> ::=
    ERROR
  | NULL
  | EMPTY ARRAY
  | EMPTY OBJECT

<JSON table error behavior> ::=
    ERROR
  | EMPTY
```

Like the other JSON querying operators, JSON_TABLE begins with <JSON API common syntax> to specify the context item, path expression and PASSING clause. The path expression in this case is more accurately called the row pattern path expression. This path expression is intended to produce an SQL/JSON sequence, with one SQL/JSON item for each row of the output table.

The COLUMNS clause can define two kinds of columns: ordinality columns and regular columns.

An ordinality column provides a sequential numbering of rows. Row numbering is 1-based.

A regular column supports columns of scalar type. The column is produced using the semantics of JSON_VALUE. The column has an optional path expression, called the column pattern, which can be defaulted from the column name. The column pattern is used to search for the column within the current SQL/JSON item produced by the row pattern. The column also has optional ON EMPTY and ON ERROR clauses, with the same choices and semantics as JSON_VALUE.

The final option for a <JSON table column definition> is <JSON table nested columns>, which is considered later.

This is followed by the PLAN clause as part of the <JSON table plan clause> option, which is used to express the processing of multiple nested paths.

This following example generates relational data containing a column, ROWSEQ, illustrating the use of the FOR ORDINALITY clause, a column, NAME, of type VARCHAR(30), and a column, ZIP, of type CHAR(5), extracted according to the specified path expressions applied to the JSON data contained in the column JCOL of the BOOKCLUB table.

```
SELECT jt.rowseq, jt.name, jt.zip
FROM bookclub,
     JSON_TABLE ( bookclub.jcol, "lax $"
                  COLUMNS ( rowSeq FOR ORDINALITY,
                            name VARCHAR(30) PATH 'lax $.Name',
                            zip CHAR(5) PATH 'lax $.address.postalCode'
                          )
                ) AS jt
```

The result of the query is shown in Table 15, “Query result”.

Table 15 — Query result

ROWSEQ	NAME	ZIP
1	John Smith	10021
2	Peter Walker	95111
3	James Lee	

5.3.4.2 Nested COLUMNS clause

The syntax for a nested COLUMNS clause is:

```
<JSON table nested columns> ::=
  NESTED [ PATH ] <JSON table nested path specification>
      [ AS <JSON table nested path name> ]
      <JSON table columns clause>

<JSON table nested path specification> ::=
  <JSON path specification>

<JSON table nested path name> ::=
  <JSON table path name>

<JSON table path name> ::=
  <identifier>
```

The nested COLUMNS clause begins with the keyword NESTED, followed by a path and an optional path name. The path provides a refined context for the nested columns. The primary use of the path name is if the user wishes to specify an explicit plan.

After the prolog to specify the path and path name, there is a COLUMNS clause, which has the same capabilities already considered.

The NESTED clause allows unnesting of (even deeply) nested JSON objects/arrays in one invocation rather than chaining several JSON_TABLE expressions in the SQL-statement.

The following example illustrates the use of an elementary nested COLUMNS clause in JSON_TABLE. Note the missing phoneNumber data in the last row of the result. This default LEFT OUTER JOIN-like semantic in this parent-child nested relationship is further explained in the next section.

```
SELECT bookclub.id, jt.name, jt.type, jt.number
FROM bookclub,
     JSON_TABLE ( bookclub.jcol, 'lax $'
                  COLUMNS ( name VARCHAR(30) PATH 'lax $.Name',
                            NESTED PATH 'lax $.phoneNumber[*]'
                            COLUMNS ( type VARCHAR(10) PATH 'lax $.type',
                                      number CHAR(12) PATH 'lax $.number' )
                ) AS jt;
```

The result of the query using the BOOKCLUB table sample data is shown in Table 16, “Query result”.

Table 16 — Query result

ID	NAME	TYPE	NUMBER
111	John Smith	Home	212 555-1234
111	John Smith	Fax	646 555-4567
222	Peter Walker	Home	408 555-9876
222	Peter Walker	Office	650 555-2468

ID	NAME	TYPE	NUMBER
333	James Lee		

5.3.4.3 PLAN clause

As seen above, every path may optionally be followed by a path name using an AS clause. Path names are identifiers and must be unique. Path names are used in the PLAN clause to express the desired output plan.

The syntax for the PLAN clause is:

```
<JSON table plan clause> ::=
    <JSON table specific plan>
  | <JSON table default plan>

<JSON table specific plan> ::=
  PLAN <left paren> <JSON table plan> <right paren>

<JSON table plan> ::=
    <JSON table path name>
  | <JSON table plan parent/child>
  | <JSON table plan sibling>

<JSON table plan parent/child> ::=
    <JSON table plan outer>
  | <JSON table plan inner>

<JSON table plan outer> ::=
  <JSON table path name> OUTER <JSON table plan primary>

<JSON table plan inner> ::=
  <JSON table path name> INNER <JSON table plan primary>

<JSON table plan sibling> ::=
    <JSON table plan union>
  | <JSON table plan cross>

<JSON table plan union> ::=
  <JSON table plan primary> UNION <JSON table plan primary>
      [ { UNION <JSON table plan primary> }... ]

<JSON table plan cross> ::=
  <JSON table plan primary> CROSS <JSON table plan primary>
      [ { CROSS <JSON table plan primary> }... ]

<JSON table plan primary> ::=
    <JSON table path name>
  | <left paren> <JSON table plan> <right paren>

<JSON table default plan> ::=
  PLAN DEFAULT <left paren> <JSON table default plan choices> <right paren>

<JSON table default plan choices> ::=
    <JSON table default plan inner/outer>
        [ <comma> <JSON table default plan union/cross> ]
  | <JSON table default plan union/cross>
      [ <comma> <JSON table default plan inner/outer> ]

<JSON table default plan inner/outer> ::=
    INNER
  | OUTER

<JSON table default plan union/cross> ::=
    UNION
  | CROSS
```

The INNER, OUTER, UNION, and CROSS concepts in the context of the PLAN clause have the following characteristics.

— INNER expresses INNER JOIN semantics.

— OUTER expresses LEFT OUTER JOIN semantics and is the default with parent/child relationships.

— The first operand of an INNER or OUTER (parent/child relationship) is necessarily a <JSON table path name> and must be an ancestor of all path names in the second operand.

— If there is an explicit PLAN clause, all path names must be explicit and appear in the PLAN clause exactly once.

— CROSS expresses CROSS JOIN semantics

— UNION expresses semantics of a FULL OUTER JOIN with an unsatisfiable predicate such as 1=0 and is the default with sibling relationships

— UNION is associative (no parentheses required for a list of paths to be unioned).

— CROSS is associative

— Parentheses are required to disambiguate complex expressions. In particular, there is no precedence between UNION and CROSS.

The following query illustrates the default OUTER semantics between the parent-child relationship of the outer context of ID and NAME with the nested columns of the book title and first and second author columns. The first and second author column values, if present, are explicitly placed in two separate columns in this example.

```
SELECT bookclub.id, jt.name, jt.title, jt.author1, jt.author2
FROM bookclub,
     JSON_TABLE ( bookclub.jcol, 'lax $'
                  COLUMNS ( name VARCHAR(30) PATH 'lax $.Name',
                            NESTED PATH 'lax $.books[*]'
                            COLUMNS ( title VARCHAR(60) PATH 'lax $.title',
                                      NESTED PATH 'lax $.authorList[*]' AS A
                                      COLUMNS ( author1 VARCHAR(30) PATH 'lax $[0]'
                                                author2 VARCHAR(30) PATH 'lax $[1]'
                                              )
                                    )
                          )
                ) AS jt;
```

The result of the query using the BOOKCLUB table sample data is shown in Table 17, “Query result”.

Table 17 — Query result

ID	NAME	TITLE	AUTHOR1	AUTHOR2
111	John Smith	The Talisman	Stephen King	Peter Straub
111	John Smith	Far From the Madding Crowd	Thomas Hardy	
222	Peter Walker	Good Omens	Neil Gaiman	Terry Pratchett
222	Peter Walker	Smoke and Mirrors	Neil Gaiman	
333	James Lee			

The following query illustrates the nested COLUMNS clause using the default UNION semantic with the sibling nested author and category columns.

```
SELECT bookclub.id, jt.name, jt.title, jt.author1, jt.category
FROM bookclub,
     JSON_TABLE ( bookclub.jcol, 'lax $'
                  COLUMNS ( name VARCHAR(30) PATH 'lax $.Name',
                            NESTED PATH 'lax $.books[*]'
                            COLUMNS ( title VARCHAR(60) PATH 'lax $.title',
                                      NESTED PATH 'lax $.authorList[*]' AS ATH
                                      COLUMNS ( author VARCHAR(30) PATH 'lax $' )
                                                NESTED PATH 'lax $.category[*]' AS CAT
                                                COLUMNS ( category VARCHAR(30)
                                                              PATH 'lax $' )
                                              )
                          ) AS jt;
```

The result of the query using the BOOKCLUB table sample data is shown in Table 18, “Query result”.

Table 18 — Query result

ID	NAME	TITLE	AUTHOR	CATEGORY
111	John Smith	The Talisman	Stephen King	
111	John Smith	The Talisman	Peter Straub	
111	John Smith	The Talisman		SciFi
111	John Smith	The Talisman		Novel
111	John Smith	Far From the Madding Crowd	Thomas Hardy	
111	John Smith	Far From the Madding Crowd		Novel
222	Peter Walker	Good Omens	Neil Gaiman	

ID	NAME	TITLE	AUTHOR	CATEGORY
222	Peter Walker	Good Omens	Terry Pratchett	
222	Peter Walker	Good Omens		Fantasy
222	Peter Walker	Good Omens		Novel
222	Peter Walker	Smoke and Mirrors	Neil Gaiman	
222	Peter Walker	Smoke and Mirrors		Fantasy
333	James Lee			

The following query illustrates the nested columns using the PLAN clause specifying the CROSS semantics between the sibling author and category columns. The rows with outer values that do not have any nested columns are not present in the result given the INNER semantics.

```
SELECT bookclub.id, jt.name, jt.title, jt.author1, jt.category
FROM bookclub,
     JSON_TABLE ( bookclub.jcol, 'lax $' AS PERSON
                  COLUMNS ( name VARCHAR(30) PATH 'lax $.Name',
                            NESTED PATH 'lax $.books[*]' AS BOOKS
                            COLUMNS ( title VARCHAR(60) PATH 'lax $.title',
                                      NESTED PATH 'lax $.authorList[*]' AS ATH
                                      COLUMNS ( author VARCHAR(30) PATH 'lax $' )
                                                NESTED PATH 'lax $.category[*]' AS CAT
                                                COLUMNS ( category VARCHAR(30)
                                                              PATH 'lax $' )
                                                )
                            )
                  PLAN ( PERSON INNER ( BOOKS INNER ( ATH CROSS CAT ) ) )
                ) AS jt;
```

The query above is equivalent to using the PLAN DEFAULT clause instead of the explicit PLAN clause as follows:

```
SELECT bookclub.id, jt.name, jt.title, jt.author1, jt.category
FROM bookclub,
     JSON_TABLE ( bookclub.jcol, 'lax $' AS PERSON
                  COLUMNS ( name VARCHAR(30) PATH 'lax $.Name',
                            NESTED PATH 'lax $.books[*]' AS BOOKS
                            COLUMNS ( title VARCHAR(60) PATH 'lax $.title',
                                      NESTED PATH 'lax $.authorList[*]' AS ATH
                                      COLUMNS ( author VARCHAR(30) PATH 'lax $' )
                                                NESTED PATH 'lax $.category[*]' AS CAT
                                                COLUMNS ( category VARCHAR(30)
                                                              PATH 'lax $' )
                                                )
                            )
                  PLAN DEFAULT ( INNER , CROSS )
                ) AS jt;
```

The result of this query using the BOOKCLUB table sample data is shown in Table 19, “Query result”.

Table 19 — Query result

ID	NAME	TITLE	AUTHOR	CATEGORY
111	John Smith	The Talisman	Stephen King	SciFi
111	John Smith	The Talisman	Stephen King	Novel
111	John Smith	The Talisman	Peter Straub	SciFi
111	John Smith	The Talisman	Peter Straub	Novel
111	John Smith	Far From the Madding Crowd	Thomas Hardy	Novel
222	Peter Walker	Good Omens	Neil Gaiman	Fantasy
222	Peter Walker	Good Omens	Neil Gaiman	Novel
222	Peter Walker	Good Omens	Terry Pratchett	Fantasy
222	Peter Walker	Good Omens	Terry Pratchett	Novel
222	Peter Walker	Smoke and Mirrors	Neil Gaiman	Fantasy

5.3.5 Conformance features for query operators

The following conformance features are defined for the SQL/JSON query operators:

— Feature T821, “Basic SQL/JSON query operators”, defined as the following:

- JSON_VALUE with no PASSING clause, no ON EMPTY, no ON ERROR clause.
  - — Without explicit PASSING syntax, no values other than the context item can be passed to an SQL/JSON path expression.
  - — Without explicit syntax, the default for ON EMPTY is NULL ON EMPTY.
  - — Without explicit syntax, the default for ON ERROR is NULL ON ERROR.
- JSON_TABLE: with no PASSING clause, no sibling NESTED COLUMNS, no PLAN, no table-level ON ERROR, and including same restrictions as JSON_VALUE for regular columns (*i.e.*, no ON EMPTY, no ON ERROR)
  - — Without explicit PASSING syntax, no values other than the context item can be passed to an SQL/JSON path expression.
  - — Without explicit PLAN syntax, the default for joining parent/ child columns has OUTER join semantics.
  - — Without sibling NESTED COLUMNS support, the default for joining sibling NESTED COLUMNS has UNION join semantics.

  - — Without explicit syntax, the default for the table-level ON ERROR is EMPTY ON ERROR.
- JSON_EXISTS with no PASSING clause and no ON ERROR clause.
  - — Without explicit PASSING syntax, no values other than the context item can be passed to an SQL/JSON path expression.
  - — Without explicit syntax, the default for ON ERROR is FALSE ON ERROR.
- IS JSON with no <JSON key uniqueness constraint>
  - — Without explicit UNIQUE KEYS syntax, the default is WITHOUT UNIQUE KEYS.
- Support for the SQL/JSON path language, except as listed as advanced features in Subclause 6.14, “Conformance features for SQL/JSON path language”.

NOTE 9 — In all of the preceding, the excluded syntax options become enabled by other features enumerated below.

— Feature T822, “SQL/JSON: IS JSON WITH UNIQUE KEYS predicate”

- Adds the WITH UNIQUE KEYS and WITHOUT UNIQUE KEYS syntax to the IS JSON predicate.

— Feature T823, “SQL/JSON: PASSING clause”

- Allows passing of additional values to the SQL/JSON path expression.

— Feature T824, “JSON_TABLE: specific PLAN clause”

- Allows explicit specification of a join plan for parent/child and sibling NESTED COLUMNS.

— Feature T825, “SQL/JSON: ON EMPTY and ON ERROR clauses”

- Allows to overwrite the default for ON EMPTY and ON ERROR options for JSON_VALUE, JSON_TABLE, JSON_QUERY, and JSON_EXISTS.

— Feature T826, “General value expression in ON ERROR or ON EMPTY clauses”

- Without this feature, the user specified value expression in the ON ERROR clause or ON EMPTY clause in JSON_VALUE or on a regular column definition in JSON_TABLE can only be a literal.

— Feature T827, “JSON_TABLE: sibling NESTED COLUMNS clauses”

- With support for this feature, the user can specify either UNION or CROSS join semantics for joining sibling NESTED COLUMNS.

— Feature T828, “JSON_QUERY”

- JSON_QUERY but no PASSING, ON EMPTY, ON ERROR, or wrapper clauses. These excluded syntax options are enabled in conjunction with other features.
  - — Without explicit PASSING syntax, no values other than the context item can be passed to an SQL/JSON path expression.
  - — Without explicit syntax, the default for ON EMPTY is NULL ON EMPTY.
  - — Without explicit syntax, the default for ON ERROR is NULL ON ERROR.
  - — Without explicit syntax, the default for the wrapper option is WITHOUT ARRAY.

— Feature T829, “JSON_QUERY: array wrapper options”

- With support for this feature, the user can specify whether none, all, or only scalar results should be enclosed in an SQL/JSON array.

— Feature T838, “JSON_TABLE: PLAN DEFAULT clause”

- This feature adds the PLAN DEFAULT syntax to JSON_TABLE.

5.4 Constructor functions and IS JSON predicate

To illustrate the use of the SQL/JSON constructor functions, consider the following two ordinary SQL tables and one additional table that incorporates a single column of JSON data:

```
CREATE TABLE depts (
  deptno  INTEGER,
  deptname CHARACTER VARYING(30) )

CREATE TABLE jobs (
  job_seq INTEGER,
  job_attrib CHARACTER(5),
  job_attval CHARACTER VARYING(64) )

CREATE TABLE employees (
   emp_id INTEGER,
   name CHARACTER VARYING(50),
   salary DECIMAL(7,2),
   dept_id INTEGER,
   json_emp CHARACTER VARYING(5000) )
```

Consider the DEPTS table with the following content:

Table 20 — DEPTS table

DEPTNO	DEPTNAME
314	Engineering
113	Architecture
12	Accounting
7	Sales
13	Executive

Consider also the JOBS table with the following content:

Table 21 — JOBS table

JOB_SEQ	JOB_ATTRIB	JOB_ATTVAL
101	Leader	155566
101	Duration	00:30:00
101	Description	Design the new table for the web site
234	Duration	01:00:00
234	Description	Load the tables with existing data
492	Leader	129596
17	Description	Design the look-and-feel of the web site

Finally, consider the EMPLOYEES table with the following content:

Table 22 — EMPLOYEES table

EMP_ID	NAME	SALARY	DEPT_ID	JSON_EMP
29334	Logan	10000	7	
29335	James	7000	7	
29336	Rachel	9000	7	

5.4.1 JSON_OBJECT

SQL applications working with JSON data will often need to construct new JSON objects, either for use within the applications themselves, for storage in the SQL database, or to return to the application program itself. This section of this Technical Report describes the built-in function, JSON_OBJECT, that constructs JSON objects from explicit name/value pairs.

```
<JSON object constructor> ::=
  JSON_OBJECT <left paren>
      [ <JSON name and value> [ { <comma> <JSON name and value> }... ]
      [ <JSON constructor null clause> ]
      [ <JSON key uniqueness constraint> ] ]
      [ <JSON output clause> ]
  <right paren>

<JSON name and value> ::=
    [ KEY ] <JSON name> VALUE <JSON value expression>
  | <JSON name> <colon> <JSON value expression>
```

```
<JSON name> ::=
  <character value expression>

<JSON constructor null clause> ::=
    NULL ON NULL
  | ABSENT ON NULL

<JSON key uniqueness constraint> ::=
    WITH UNIQUE [ KEYS ]
  | WITHOUT UNIQUE [ KEYS ]
```

The <JSON name>s may not be NULL. The <JSON value expression>s may be NULL, with the action taken controlled by the <JSON constructor null clause>. NULL ON NULL produces an SQL/JSON null, while ABSENT ON NULL omits that key:value pair from the resulting SQL/JSON object. The default if no <JSON constructor null clause> is given is NULL ON NULL.

JSON_OBJECT is typically used in <select list>s, as illustrated in the following example.

```
SELECT
  JSON_OBJECT( KEY 'deptno' VALUE d.deptno,
               KEY 'deptname' VALUE d.deptname ) AS D314
FROM depts AS d
WHERE d.deptno = 314
```

This query returns one row for department 314 recorded in the DEPTS table; that row contains a single column, which contains a serialization of a JSON object having the department number and name. The result column type is VARCHAR with an implementation-defined length. Visually, the returned JSON object would look something like this:

NOTE 10 — This and other examples in this Technical Report show insignificant whitespace in the result for readability only. For example, the spaces following the left curly brace, before and after the colons, *etc.*, in the output are insignificant whitespace. While a conforming SQL-implementation may add insignificant whitespace, no conforming SQL-implementation is required to do so.

Table 23 — The JSON object returned

D314
`{ "deptno" : 314, "deptname" : "Engineering" }`

If Feature T814, “Colon in JSON_OBJECT or JSON_OBJECTAGG” is implemented, the query could also be expressed using colon as a key-value separator:

```
SELECT  JSON_OBJECT( 'deptno' : d.deptno, 'deptname' : d.deptname ) AS D314
FROM depts AS d
WHERE d.deptno = 314
```

5.4.2 JSON_OBJECTAGG

Often, it is inappropriate or even impossible to construct a JSON object by explicitly specifying the names of the contained name/value pairs (*e.g.*, because the names are not known a priori). Instead, an application developer may wish to construct a JSON object as an aggregation of information in an SQL table. Presuming that the

SQL table actually contains a column with JSON names and another column with corresponding values, the built-in function JSON_OBJECTAGG (“object aggregate”) performs this function.

```
<JSON object aggregate constructor> ::=
  JSON_OBJECTAGG <left paren>
      <JSON name and value>
      [ <JSON constructor null clause> ]
      [ <JSON key uniqueness constraint> ]
  [ <JSON output clause> ]
  <right paren>
```

The default if no <JSON constructor null clause> is given is NULL ON NULL.

The following example will create a JSON object containing a sequence of name/value pairs in which the name is a department name and the value is the department number:

```
SELECT JSON_OBJECTAGG ( deptname VALUE deptno )
FROM depts
```

The result of this query is a table containing a single row of one column, which contains a serialization of a JSON object. That object would look something like this:

```
{ "Engineering" : 314, "Architecture" : 113, "Accounting" : 12,
  "Sales" : 7, "Executive" : 13 }
```

The reader will observe that this is actually a kind of “pivot” of the DEPTS table.

The JSON_OBJECTAGG function can also be used in grouped queries to good effect.

The SQL query:

```
SELECT j.job_seq,
       JSON_OBJECTAGG ( j.job_attrib, j.job_attval RETURNING VARCHAR(200) )
     AS attributes
FROM jobs AS j
GROUP BY j.job_seq
```

will produce a table containing four rows, each containing two columns. The first column of the table contains the job sequence numbers, while the second column contains a serialization of a JSON object that is a pivot of the information in all of the rows associated with the corresponding job sequence number. The result type of the JSON object is VARCHAR with a maximum length of 200, as specified in the query. An exception condition would be raised if this length is exceeded. The result will look something like this:

Table 24 — Returned JSON object with the corresponding job sequence number

JOB_SEQ	ATTRIBUTES
101	{ "Leader" : "155566", "Duration" : "00:30:00", "Description" : "Design the new tables for the web site" }
234	{ "Duration" : "01:00:00", "Description" : "Load the tables with existing data" }
492	{ "Leader" : "129596" }
17	{ "Description" : "Design the look-and-feel of the web site" }

5.4.3 JSON_ARRAY

Just as an application developer might wish to construct a JSON object from an explicit list of data, she might wish to construct a JSON array from a similar list of data. The built-in function JSON_ARRAY provides that capability.

```
<JSON array constructor> ::=
    <JSON array constructor by enumeration>
  | <JSON array constructor by query>

<JSON array constructor by enumeration> ::=
  JSON_ARRAY <left paren>
      [ <JSON value expression> [ { <comma> <JSON value expression> }... ]
      [ <JSON constructor null clause> ] ]
      [ <JSON output clause> ]
  <right paren>

<JSON array constructor by query> ::=
  JSON_ARRAY <left paren>
      <query expression>
      [ <JSON input clause> ]
      [ <JSON constructor null clause> ]
      [ <JSON output clause> ]
  <right paren>
```

JSON_ARRAY has two variants: One variant produces its result from an explicit list of SQL values (literals or computed values, including subqueries); the second variant produces its results from an SQL query expression invoked within the function. For the constructor-by-query form, the query expression must return exactly one column, and the array values are formed from the column values generated by the query expression.

For JSON_ARRAY, if not explicitly specified, the default ON NULL clause is ABSENT ON NULL (which is different from the default for JSON_OBJECT).

The following query illustrates the use of JSON_ARRAY:

```
SELECT
  JSON_ARRAY ( 'deptno', d.deptno, 'deptname', d.deptname )
  FROM depts AS d
  WHERE d.deptno = 314
```

This query returns one row for department 314 recorded in the DEPTS table; that row contains a single column, which contains a serialization of a JSON array containing two character string literals, the department number, and the department name. Visually, the returned JSON array would look something like this:

```
[ "deptno", 314, "deptname", "Engineering" ]
```

5.4.4 JSON_ARRAYAGG

Just as an SQL application might need to construct a JSON object as an aggregation of SQL data, so might it need to construct a JSON array as an aggregate.

```
<JSON array aggregate constructor> ::=
  JSON_ARRAYAGG <left paren>
      <JSON value expression>
      [ <JSON array aggregate order by clause> ]
      [ <JSON constructor null clause> ]
      [ <JSON output clause> ]
  <right paren>

<JSON array aggregate order by clause> ::=
  ORDER BY <sort specification list>
```

The default, if no ON NULL clause is given, is ABSENT ON NULL.

For example:

```
SELECT JSON_ARRAYAGG ( j.job_attval RETURNING CLOB(8K) ) AS attributes
FROM jobs AS j
```

The result of this query is a table of one row and one column, which would look something like this:

Table 25 — Query result

ATTRIBUTES
`[ "155566", "00:30:00", "Design the new tables for the web site", "01:00:00", "Load the tables with existing data", "129596", "Design the look-and-feel of the web site" ]`

JSON_ARRAYAGG supports an optional ORDER BY clause that allows the results of the query to be ordered before the selected data is extracted to be placed in the resulting JSON array. As an example, the SQL that might be used to create a JSON object for each department listing all employees and their salary in order of increasing salary looks like this:

```
SELECT JSON_OBJECT (
    'department' : d.name,
    'employees' : JSON_ARRAYAGG ( JSON_OBJECT ( 'employee' : e.name,
                                                'salary' : e.salary )
                                  ORDER BY e.salary ASC ) ) AS departments
FROM depts d, employees e
WHERE d.deptno = e.dept_id
GROUP BY d.deptno;
```

Table 26, “Query Results”, illustrates the results of this query:

Table 26 — Query Results

DEPARTMENTS
`{ "department" : "Sales", "employees" : [ { "employee" : "James", "salary" : 7000}, { "employee" : "Rachel", "salary" : 9000}, { "employee" : "Logan", "salary" : 10000} ] }`

In the following example, when there no employees in a department, a JSON null is output as the employee name and the salary in the result:

```
SELECT JSON_OBJECT (
    'department' : d.name,
    'employees' : JSON_ARRAYAGG ( JSON_OBJECT ( 'employee' : e.name,
                                                'salary' : e.salary )
                                        ORDER BY e.salary ASC ) ) AS departments
FROM depts d LEFT OUTER JOIN employees e
     ON ( d.deptno = e.dept_id )
GROUP BY d.deptno;
```

The result would be something like those shown in Table 27, "Query Results":

Table 27 — Query Results

DEPARTMENTS

`{ "department" : "Sales",`
`  "employees" : [ { "employee" : "James", "salary" : 7000},`
`                  { "employee" : "Rachel", "salary" : 9000},`
`                  { "employee" : "Logan", "salary" : 10000} ]`
`}`

`{ "department" : "Engineering",`
`  "employees" : [ { "employee" : null, "salary" : null } ]`
`}`

`{ "department" : "Architecture",`
`  "employees" : [ { "employee" : null, "salary" : null } ]`
`}`

`{ "department" : "Accounting",`
`  "employees" : [ { "employee" : null, "salary" : null } ]`
`}`

`{ "department" : "Executive",`
`  "employees" : [ { "employee" : null, "salary" : null } ]`
`}`

5.4.5 IS JSON predicate

Applications will frequently want to ensure that the data they expect to consume as JSON data is, indeed, JSON data. The IS JSON predicate determines whether the value of a specified string does or does not conform to the structural rules for JSON. The syntax of the IS JSON predicate is:

```
<JSON predicate> ::=
  <string value expression> [ <JSON input clause> ] IS [ NOT ] JSON
      [ <JSON predicate type constraint> ]
      [ <JSON key uniqueness constraint> ]
```

```
<JSON predicate type constraint> ::=
    VALUE
  | ARRAY
  | OBJECT
  | SCALAR
```

If <JSON input clause> is not specified, then FORMAT JSON is implicit. If <JSON key uniqueness constraint> is not specified, then WITHOUT UNIQUE KEYS is implicit.

5.4.6 Handling of JSON nulls and SQL nulls

SQL (correctly) distinguishes between data such as zero-length strings and the special pseudo-value known as "the null value". The semantics of those two things are quite different and those differences affect a great many SQL operations. The differences are an important part of the semantics of the SQL language.

The JSON null provides yet another related difference. In JSON, null is an actual value, represented by a JSON literal ("null"). It must be able to distinguish JSON nulls from SQL null values and that distinction is an important part of the semantics of JSON handling in the SQL context.

To illustrate the situation, consider the JSON object stored in a column of an SQL table:

```
{ "a" : null, "b" : "null", "c" : "" }
```

JSON_VALUE, evaluated against that JSON object, returning the result as an SQL scalar value, would return, for each respective name/value pair, the following: the SQL null value, the SQL character string comprising the four characters "`null`", and the SQL zero-length character string; if JSON_VALUE were used to retrieve the value associated with the name "d", it would return the SQL null value. Note that, when retrieving the value of the first name/value pair, the SQL/JSON null value is automatically transformed into an SQL null value.

The JSON constructor functions have to deal with situations in which the SQL data that is being queried are SQL null values. SQL/JSON supplies optional syntax to allow the application author to select whether SQL null values are included in the JSON object or JSON array being constructed, or whether object members or array elements whose (bound) values are SQL null values are omitted from the JSON object or JSON array being constructed.

5.4.7 Conformance features for constructor functions

There are five conformance features defined for the JSON constructor functions:

— Feature T811, "Basic SQL/JSON constructor functions", defined as the following:

  - JSON_OBJECT with no <JSON key uniqueness constraint>.
    — Without explicit UNIQUE KEYS syntax, the default is WITHOUT UNIQUE KEYS.
  - JSON_ARRAY.
  - JSON_ARRAYAGG without the ORDER BY option.
    — Without explicit ORDER BY syntax, the ordering of the elements in the SQL/JSON array is implementation-dependent.

NOTE 11 — In all of the preceding, the excluded syntax options become enabled by other features enumerated below.

— Feature T812, “SQL/JSON: JSON_OBJECTAGG” with no <JSON key uniqueness constraint>.

— Feature T813, “SQL/JSON: JSON_ARRAYAGG with ORDER BY”

- This feature allows the user to specify an order of the elements of the constructed SQL/JSON array.

— Feature T814, “Colon in JSON_OBJECT or JSON_OBJECTAGG”

- Subclause 5.4.1, “JSON_OBJECT”, contains examples that show the syntax of JSON_OBJECT with and without support for Feature T814, “Colon in JSON_OBJECT or JSON_OBJECTAGG”.

— Feature T830, “Enforcing unique keys in SQL/JSON constructor functions”

- Adds the WITH UNIQUE KEYS and WITHOUT UNIQUE KEYS syntax to JSON_OBJECT and JSON_OBJECTAGG.

6 SQL/JSON path language

6.1 Overview of SQL/JSON path language

The SQL/JSON path language is a query language used by certain SQL operators (JSON_VALUE, JSON_QUERY, JSON_TABLE and JSON_EXISTS, collectively known as the SQL/JSON query operators) to query JSON text. The SQL/JSON path language is not, strictly speaking, SQL, though it is embedded in these operators within SQL. Lexically and syntactically, the SQL/JSON path language adopts many features of [ECMAscript], though it is neither a subset nor a superset of [ECMAscript]. The semantics of the SQL/JSON path language are primarily SQL semantics.

The SQL/JSON path language is used by the SQL/JSON query operators in the architecture shown in this diagram:

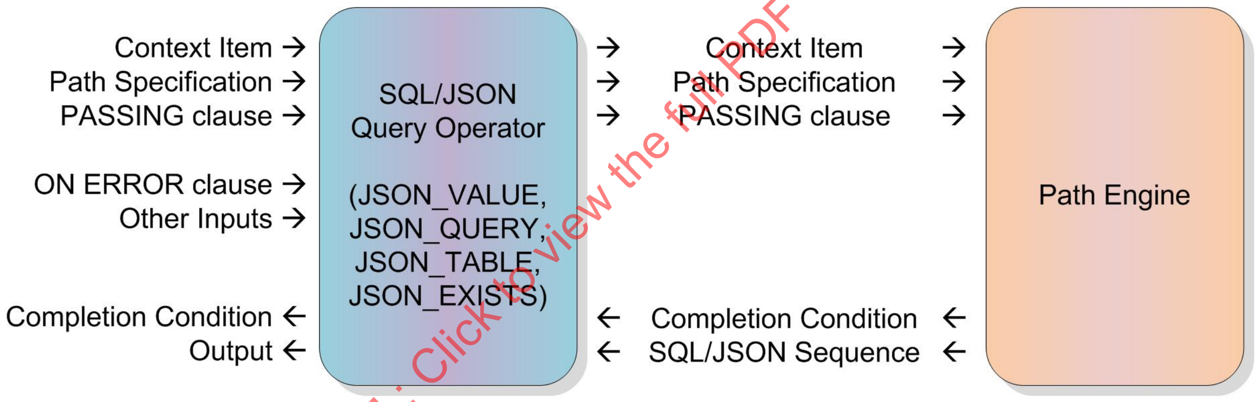

Figure 2 — The SQL/JSON path language architecture

The SQL/JSON query operators share the same first three lines in the diagram, which are expressed syntactically in the <JSON API common syntax> that is used by all SQL/JSON query operators. This framework provides the following inputs to an SQL/JSON query operator:

1) A context item (the JSON text to be queried).

2) A path specification (the query to perform on the context item; this query is expressed in the SQL/JSON path language specified in [ISO9075-2], Subclause 9.38, “SQL/JSON path language: lexical elements” and Subclause 9.39, “SQL/JSON path language: syntax and semantics”.

3) A PASSING clause (SQL values to be assigned to variables in the path specification, for example, as values used in predicates within the path specification).

The SQL/JSON operators effectively pass these inputs to a “path engine” that evaluates the path specification, using the context item and the PASSING clause to specify the values of variables in the path specification. The effective behavior of the path engine is specified in the General Rules of Subclause 9.39, “SQL/JSON path language: syntax and semantics”, in [ISO9075-2].

The result of evaluating a path specification on a context item and PASSING clause is a completion condition, and, if the completion condition is successful completion, an SQL/JSON sequence. The SQL/JSON query operators, in their General Rules, use the completion code and SQL/JSON sequence to complete the specific computation specified via the particular SQL/JSON query operator.

Errors can occur at the following junctures in this architecture:

1) An error can occur when converting an input. For example, if the context item does not parse as JSON text, then that is an input conversion error.

2) An error can occur while processing an SQL/JSON path expression. This category of errors is further subdivided as follows:

   a) A structural error occurs when an SQL/JSON path expression attempts to access a non-existent element of an SQL/JSON array or a non-existent member of a JSON object.

   b) A non-structural error is any other error during evaluation of an SQL/JSON path expression; for example, divide by zero.

3) An error can occur when converting an output.

The SQL operators JSON_VALUE, JSON_QUERY, JSON_TABLE, and JSON_EXISTS provide the following mechanisms to handle these errors:

1) The SQL/JSON path language traps any errors that occur during the evaluation of a <JSON filter expression>. Depending on the precise <JSON path predicate> contained in the <JSON filter expression>, the result may be *Unknown*, *True*, or *False*, depending on the outcome of non-error tests evaluated in the <JSON path predicate>.

2) The SQL/JSON path language has two modes, strict and lax, which govern structural errors, as follows:

   a) In lax mode:

      i) If an operation requires an SQL/JSON array but the operand is not an SQL/JSON array, then the operand is first wrapped in an SQL/JSON array prior to performing the operation.

      ii) If an operation requires something other than an SQL/JSON array, but the operand is an SQL/JSON array, then the operand is unwrapped by converting its elements into an SL/JSON sequence prior to performing the operation.

      iii) After applying the preceding resolutions to structural errors, if there is still a structural error, the result is an empty SQL/JSON sequence.

   b) In strict mode, if the structural error occurs within a <JSON filter expression>, then the error handling of <JSON filter expression> applies. Otherwise, a structural error is an unhandled error.

3) Non-structural errors outside of a <JSON path predicate> are always unhandled errors, resulting in an exception condition returned from the path engine to the SQL/JSON query operator.

4) The SQL/JSON query operators provide an ON ERROR clause to specify the behavior in case of an input conversion error, an unhandled structural error, an unhandled non-structural error, or an output conversion error.

6.2 Objectives for the SQL/JSON path language

The objectives for the SQL/JSON path language are:

1) Minimalism: a minimal language that meets a short list of use cases, leaving freedom to adapt the language to additional use cases in the future.

   The following decisions were made based on this objective:

   a) Rule out the following: union, intersection, difference, join, FLWOR expressions.

   b) Only a minimal set of predicates, such as the standard comparison operators, on atomic values only (no “deep equal”).

   c) JSON path expressions must be compile-time constants; this excludes dynamic JSON path expressions embedded in static SQL queries.

   d) Parameters to JSON queries must be passed by value, not by reference.

   e) No “reverse axes”.

2) SQL semantics: the language should be readily integratable into an SQL engine; therefore, the semantics of predicates, operators, *etc.* should generally follow SQL. “Push-down” optimizations are facilitated by making the semantics in SQL and in the path language the same.

   Here are some consequences of this objective:

   a) Functions in the path expression language should have SQL semantics.

   b) Predicates should have SQL semantics; this means three-valued logic, SQL comparison rules (such as trailing blank handling) and any other semantics issues should also be derived from SQL. However, JSON null is not the same as SQL null, so that `null == null` is *True* and there is no need for an `is null` predicate.

3) JavaScript-like: the language should evolve from JavaScript, because that is the language that users will find most appropriate for working on JSON. JavaScript has been standardized as [ECMAscript]. This means that lexical and syntactic issues generally follow JavaScript, while semantic issues follow SQL in case of conflict between the two.

   Here are some consequences of this objective:

   a) Dot (.) for member access and [] for array access. 0-relative arrays (JavaScript-like rather than SQL-like) were adopted.

   b) Lexical and syntactic design generally follow JavaScript.

4) Implementation accommodation: Some platforms for JSON have adopted some conventions that are laxer than JavaScript, particularly that in some contexts a singleton array can behave like a scalar, and conversely a scalar can behave like a singleton array. As a consequence of this objective, a “lax” mode is provided that avoids errors on certain path expressions that would be regarded as errors in JavaScript.

6.3 Modes

The path engine has two modes, strict and lax. The motivation for these modes is that strict mode will be used to examine data from a strict schema perspective, for example, to look for data that diverges from an expected schema. Therefore, strict mode raises an error if the data does not strictly adhere to the requirements of a path expression. Lax mode is intended to be more forgiving, so lax mode converts errors to empty SQL/JSON sequences.

In addition, lax mode adopts the convention that an array of size 1 is interchangeable with the singleton. This convention is supported with the following conventions:

1) If an operation requires an array but the operand is not an array, then the operand is implicitly “wrapped” in an array.

2) If an operation requires a non-array but the operand is an array, then the operand is implicitly “unwrapped” into an SQL/JSON sequence.

These modes govern three aspects of path evaluation, as shown in the following table:

Table 28 — Three aspects of path evaluation governed by modes

	lax	**strict**
Automatic unnesting of arrays	Certain path steps, such as the member accessor $.key, automatically iterate over SQL/JSON sequences. To make these iterative path steps friendlier for arrays, arrays are automatically unnested prior to performing the iterative path step. This means that the user does not need to use an explicit [*] to unnest an array prior to performing an iterative path step. This facilitates the use case where a field may be either an array or a scalar.	Arrays are not automatically unnested (the user can still write [*] to unnest an array explicitly).
Automatic wrapping within an array	Subscript path steps, such as $[0] or $[*], may be applied to a non-array. To do this, the non-array is implicitly wrapped in an array prior to applying the subscript operation. This also facilitates the use case where a field may be either an array or a scalar.	There is no automatic wrapping prior to subscript path steps.

	lax	strict
Error handling	Many errors related to whether data is or is not an array or scalar are handled by the two preceding features. The remaining errors are classified as either structural or non-structural. An example of a structural error is $.name if $ has no member whose key is name. Structural errors are converted to empty SQL/JSON sequences. An example of a non-structural error is divide by zero; such errors are not elided.	Errors are strictly defined in all cases

Note that the path language mode is orthogonal to the ON ERROR clause. There are numerous use cases for having any combination of ON ERROR clauses combined with either strict or lax modes.

6.3.1 Example of strict vs lax

Consider the following data, stored in a table called Data:

Table 29 — Example of strict *vs* lax

pk	col
1	`{ name: "Fred", phonetype: "work", "phone#": "650-506-2051" }`
2	`{ name: "Molly", phones: [ { phonetype: "work", "phone#": "650-506-7000" }, { phonetype: "cell", "phone#": "650-555-5555" } ] }`
3	`{ name: "Afu", phones: [ { phonetype: "cell", "phone#": "88-888-8888" } ] }`
4	`{ name: "Justin" }`
5	`{ name: "U La La", phones: [] }`

This data has been created with a sloppy schema. If a person has just one phone (row 1), then the phonetype and phone# are members of the JSON object. If a person has more than one phone (row 2), then there is a member called phones whose value is an array holding the phone information. But sometimes a person with just one phone still has a phones array (row 3). Also, some people have no phones, which can be indicated by an absence of the phonetype and phone# members (row 4), or by the presence of a phones array whose value is empty (row 5).

Now the question is how to use JSON_TABLE to display all the name and phone information. Suppose one wants to get a table with columns called name, phonetype, and phone#. If a person has multiple phones, the display should be denormalized, with the person's name repeated in multiple rows, in order to display each phone number in a separate row. If a person has no phones, the person name should appear in a single row, with nulls for the phone information.

Processing this data would be very difficult using strict mode. This is why lax mode is provided: to make it easier to deal with sloppy schemas such as this.

The solution to this use case is the following query:

```
SELECT D.pk, JT.name,
       COALESCE (JT."phone#", JT."phones.phone#") AS "phone#",
       COALESCE (JT."phonetype", JT."phones.phonetype#") AS "phonetype"
FROM Data AS D,
     JSON_TABLE (D.col, 'lax $'
         COLUMNS (
             name VARCHAR(30) PATH 'lax $.name',
             "phone#" VARCHAR(30) PATH 'lax $.phone#',
             "phonetype" VARCHAR(30) PATH 'lax $.phonetype',
             NESTED COLUMNS PATH 'lax $.phones[*]' (
                "phones.phone#" VARCHAR(30) PATH 'lax $.phone#',
                "phones.phonetype" VARCHAR(30) PATH 'lax $.phonetype'
             )
         )
     ) AS JT
```

Above, two output columns of the JSON_TABLE have been underlined, and two others are boxed. To understand this query, note the following:

1) Row 1 has phone# and phonetype as "bare" members of the outermost object. These two members will be picked up by the underlined columns called "phone#" and "phonetype". The NESTED COLUMNS clause has a path that will find no rows. The default plan for NESTED COLUMNS is an outer join. Thus, there will be effectively a dummy row created with null values for the boxed columns. In the SELECT list, each COALESCE operator is used to choose the non-null values from an underlined column and the corresponding boxed column.

2) Rows 2 and 3 do not have bare phone# and phonetype; instead they have an array called `phones`. In these rows, the underlined columns have paths that will find empty sequences, defaulting to the null value. The NESTED COLUMNS clause is used to iterate over the phones array, producing values for the boxed columns, and again, the COALESCE operators in the SELECT list retain the non-null values.

3) Row 4 has no phone data at all. In this case, the underlined columns have paths that will find nothing (defaulting to null values). The NESTED COLUMNS clause also has a path that finds an empty sequence. Using the default outer join logic, this means that the boxed columns will also be null. The COALESCE operators must coalesce two null values, resulting in null.

4) Row 5 has a phones array, but it is empty. This case is processed similarly to rows 2 and 3: the underlined columns are null because their paths are empty. The NESTED COLUMNS clause is used, but the array is empty, so this is an outer join with an empty table. Thus, the boxed columns also come up null, and the COALESCE operators combine these nulls to get null. The end result is the same as row 4.

6.4 Lexical issues

Lexically, the SQL/JSON path language generally follows the conventions of [ECMAscript] (with a few modifications detailed below). It follows that SQL/JSON path language is case-sensitive in both identifiers and key words. Unlike SQL, there are no “quoted” identifiers, and there is no automatic conversion of any identifiers to uppercase.

It was decided not to adopt the following lexical features of JavaScript into the SQL/JSON path language:

— Comments.

— Hex numeric literals.

— JavaScript regular expressions (instead the SQL predicate LIKE_REGEX is adopted, which uses XQuery regular expressions, written as JavaScript character string literals).

— Automatic semicolon insertion (this feature pertains to JavaScript statements; since we only have expressions and not statements, this is not relevant to the SQL/JSON path language).

The following lexical adjustments were made:

— Identifiers must not start with $.

— @ is an additional punctuator.

It turns out that no reserved words in SQL/JSON path language are required. The issue is how to determine in a lexical scanner whether an alphabetic string is a key word or an identifier. In the defined language, identifiers occur in only two contexts in the language:

— Beginning with a dollar sign, as a variable name.

— After a period, as a member name (never followed by a <left paren>).

Keywords never begin with a dollar sign, and, if they can come after a period, are always followed by a <left paren>. Thus, it is possible to determine if a token is an identifier or a key word purely from the lexical context.

The rules for nested quoted strings were particularly examined. An SQL/JSON path expression is required to be an SQL character string literal, so it will be enclosed in single quotes. Within this literal, the user may wish to write a character string literal; such a character string literal will be written using the JavaScript convention to enclose in double quotes. Within this character string literal, the user may wish to have a single quote. At this point the user must escape the single quote, which can be done using either the SQL convention of writing it twice, or using a JavaScript escape.

Here is an example. The user wishes to find names that start with “O'” such as “O'Connor”. The user writes this query:

```
'lax $.name ? (@ starts with "O''")'
```

The quotes in the preceding example are interpreted as follows:

— The outermost single quotes ' enclose an SQL character string literal.

— The double quotes " enclose a character string in the SQL/JSON path language.

— The inner single quotes ' are doubled in accordance with the SQL convention, because they are contained in an SQL character string literal. The pair actually denotes one instance of a single quote.

The example could also be written using JavaScript escapes to represent the single quote, although this is not a good option. The example would be written:

```
'lax $.name ? (@ starts with "O\''")'
```

Here the user is using the JavaScript escape for single quote, which is \'. However, the single quote in this must still survive the quoting rules of the outermost container, the SQL character string literal, so it is necessary to write \''. Thus, there is no benefit in using JavaScript escape here.

It would also be feasible to use the “\u” escape for single quote, which is “\u0027”, like this:

```
'lax $.name ? (@ starts with "O\u0027")'
```

Now let's look at double quotes. Suppose the user wants to search $.text for an initial substring:

```
"hello
```

The user might write:

```
'lax $.text ? (@ starts with "\"hello")'
```

In this example there is no problem with placing a double quote within the outermost single quotes which delimit an SQL character string literal. However, there is a problem placing a double quote within a JavaScript double-quoted literal; therefore the need to use the JavaScript escape `\"`. Alternatively, using `\u` escapes:

```
'lax $.text ? (@ starts with "\u0022hello")'
```

6.5 Syntax summary

The following table summarizes the features of the SQL/JSON path language:

Table 30 — Features of the SQL/JSON path language

Component	**Example**
literals	`“hello”, 1.5e3, true, false, null`
variables	`$` — context item `$frodo` — variable whose value is set in PASSING clause `@` — value of the current item in a filter
parentheses	($a + $b)*$c

Component	Example
accessors	member accessor: `$.phone` wildcard member accessor: `$.*` element accessor: `$[1, 2, 4 to 7]` wildcard element accessor: `$[*]`
filter	`$?( @.salary > 100000 )`

boolean	`&&`
	`||`
	`!`

comparison	`== != <> < <= > >=`

special predicates	`exists ($)`
	`($a == $b) is unknown`
	`$ like_regex "colou?r"`
	`$ starts with $a`

arithmetic	`+ - * / %`

item functions	`$.type()`
	`$.size()`
	`$.double()`
	`$.ceiling()`
	`$.floor()`
	`$.abs()`
	`$.datetime()`
	`$.keyvalue()`

6.6 Formal semantics

6.6.1 Notational conventions

Throughout the formal semantics, there are SQL/JSON sequences of SQL/JSON items. SQL/JSON sequences are generally denoted *S*, possibly with additional letters and possible with a subscript, and SQL/JSON items are generally denoted *I*, possibly with additional letters and possibly with a subscript. SQL/JSON sequences are shown enclosed in parentheses, like this: $S = (I_1, I_2, ..., I_n)$. Individual subscripts on SQL/JSON items are denoted with lowercase letters, such as *i*, *j*, or *k*.

Objects are represented as an unordered set of members $\{ M_1, ..., M_m \}$, where each member is a key/bound value pair: $M_j = K_j : V_j$. Or an object can be represented as $\{ K_1 : V_1, ..., K_m : V_m \}$.

Arrays are represented as an ordered list of elements: $[E_1, ..., E_s]$.

6.7 Primitive operations

The formal semantics uses the following primitive operations:

6.7.1 Concatenation

Concatenation of SQL/JSON sequences S_1, S_2, ..., S_n is denoted (S_1, S_2, ..., S_n). There is no nesting of SQL/JSON sequences and empty SQL/JSON sequences are removed.

6.7.2 unwrap()

The unwrap() operator expands all the arrays in an SQL/JSON sequence.

Let S = (I_1, I_2, ..., I_n); unwrap(S) is defined by these rules:

1) For each j between 1 and n, let $S2_j$ be the SQL/JSON sequence

   Case:

   a) If I_j is an array I_j = [E_1, ..., E_m], then let $S2_j$ = (E_1, ..., E_m).

   b) Otherwise, let $S2_j$ = (I_j).

2) The result of unwrap(S) is the concatenation of the SQL/JSON sequences ($S2_1$, $S2_2$, ..., $S2_n$).

The unwrap() operator is only used in lax mode. Its purpose is to support data that is sometimes a single object and sometimes an array of objects. If it is an array of objects, the user wants to ignore the array boundary and just drill down to the members of the objects. This user view is accommodated by converting the array into an SQL/JSON sequence prior to accessing the members of the nested objects. Example: $.phones.type using the data shown below:

Table 31 — Data used by unwrap() example

T.C
`{ name: "Babu", phones: { type: "cell", "090-0101" } }`

`{ name: "Fred", phones: [ { type: "home", number: "372-0453" },`
`                          { type: "work", number: "506-2051" } ] }`

In the first row, phones is just an object, so there is no problem performing $.phones.type.

In the second row, phones is an array of objects. In lax mode, $.phones will evaluate to an array, and then the next step to get type will use the unwrap operator to iterate over the array, so the end result is an SQL/JSON sequence with two values, “home” and “work”. This is equivalent to performing $.phones[*].type in either mode.

6.7.3 wrap()

wrap: wrap() converts any nonarray in an SQL/JSON sequence to an array of length 1.

Let $S = (I_1, I_2, ..., I_n)$; wrap(S) is defined by these rules:

1) For each j between 1 and n, let $I2_j$ be the SQL/JSON item

   a) If I_j is an array, then $I2_j = I_j$.

   b) Otherwise, $I2_j = [I_j]$.

2) The result of wrap (S) is the SQL/JSON sequence $(I2_1, I2_2, ..., I2_n)$

The wrap() operator is only used in lax mode. Its role is to handle data that is sometimes an array and sometimes not an array. This is similar to the unwrap() operator. The difference is that wrap() is used when the user's intended final outcome is a singleton. That is, if the data is an array, the user only wants to get a single element from the array, say $[0]. If the data is not an array, then the user wants the operation to act as if it were a singleton array. Example: $.phones[0] applied to the following data:

Table 32 — Data used by wrap() example

T.C
{ name: "Fred", phones: ["372-0453", "506-2051"] }
{ name: "Babu", phones: "090-0101" }

On the first row, the result is “372-0453”, on the second row the result is “090-0101”.

NOTE 12 — wrap() and unwrap() are not inverses in general. However, if $A = [E]$ is a singleton array and E is not an array, then

wrap (unwrap ([E])) = wrap ((E)) = [E]

Also if $S = (I_1, ..., I_n)$ is an SQL/JSON sequence that contains no arrays, then

unwrap (wrap ($I_1, ..., I_n$) = unwrap ([I_1], ..., [I_n]) = ($I_1, ..., I_n$)

6.8 Mode declaration

A <JSON path expression> begins with a declaration of either strict or lax mode:

```
<JSON path expression> ::=
  <JSON path mode> <JSON path wff>

<JSON path mode> ::=
    strict
  | lax
```

<JSON path wff> is the “meat” of an SQL/JSON path expression (“wff” stands for “well-formed formula”).

6.9 <JSON path primary>

In programming languages, a “primary” is a BNF non-terminal that is self-delimited, either because it is a single token, or because of matching delimiters such as parentheses. (For example, <value expression primary> and <table primary> in [ISO9075-2]). The primaries in the specified language are given by the BNF:

```
<JSON path primary> ::=
    <JSON path literal>
  | <JSON path variable>
  | <left paren> <JSON path wff> <right paren>
```

6.9.1 Literals

The atomic values in the SQL/JSON path language are written the same as in JSON, and are interpreted as if they were SQL values. Here are some examples:

Table 33 — Examples of atomic values in the SQL/JSON path language

As written	Interpreted as
`true`	boolean *True*
`false`	boolean *False*
`null`	SQL/JSON null
`123`	exact numeric scale 0 value 123
`12.3`	exact numeric scale 1 value 12.3
`12.3e0`	approximate numeric value 12.3
`"hello"`	Unicode character string, value 'hello' (without the delimiting quotes)

In character strings, the escaping rules of both SQL (as the outer language) and JavaScript apply. Here are some examples:

Table 34 — Examples of the escaping rules

Example	Explanation
`"O''Connor"`	The single quote character is escaped by doubling (SQL convention). The value is O'Connor
`"\"hello\""`	The double quote character is escaped with a backslash (JavaScript convention). The value is "hello"

6.9.2 Variables

The BNF for variables is:

```
<JSON path variable> ::=
    <JSON path context variable>
  | <JSON path named variable>
  | <at sign>
  | <JSON last subscript>

<JSON path context variable> ::=
  <dollar sign>

<JSON path named variable> ::=
  <dollar sign> <JSON path identifier>

<JSON last subscript> ::=
  last
```

Context variable: The SQL/JSON path language is always invoked with a context item. The context item is referenced using the symbol `$`. The context item is parsed as JSON; it is an error if the parsing fails.

Named variables: Optionally, additional values can be passed in to the path engine using the PASSING clause. Each value in the PASSING clause has an SQL identifier declared using AS. For example:

```
JSON_VALUE (T1.J, 'lax $.phone [$K]'
  PASSING T2.Huh AS K)
```

The preceding example passes in the computed value T2.Huh as a variable named K. Within the path expression, this value is referenced using the variable `$K`.

In the preceding example, the declared type of T2.Huh must be supported in the SQL/JSON data model. This means it must be a character string with character set Unicode, numeric, boolean, or datetime. It cannot be a binary string, interval, row type, user-defined type, reference type, or collection type.

It is also possible to pass JSON to a named variable. The (unnamed) context item is always parsed as JSON; to parse a named variable, the FORMAT clause is required, as in this example:

```
JSON_EXISTS (T1.J1, 'lax $ ? (@.name == $J2.name)'
  PASSING T2.J2 FORMAT JSON AS J2)
```

The preceding example compares the `name` field in two JSON values, T1.J1 and T2.J2. T1.J1 is chosen as the context item, whereas T2.J2 is passed in the PASSING clause. T1.J1 does not need a FORMAT clause, whereas T2.J2 does, because without it T2.J2 would not be parsed and would only be passed as a character string. The path expression tests whether the `name` member in T1.J1 is the same as the `name` member in T2.J2, using a filter expression (presented later). The result of the path expression is an empty SQL/JSON sequence if the `name` members are not equal, causing JSON_EXISTS to return *False*. If the `name` members are equal, then the result of path expression is a singleton SQL/JSON item, and the result of JSON_EXISTS is *True*.

Note that it is necessary to observe the identifier rules of both SQL and JavaScript. Going back to the first example, the SQL identifier K was coerced to uppercase since it is not a quoted identifier. JavaScript does not coerce its identifiers to either upper or lower case. Consequently the following would be an error:

```
JSON_VALUE (T1.J, 'lax $.phone [$k]'
  PASSING T2.Huh AS k)
```

In the erroneous rewrite, uppercase K has been replaced everywhere by lowercase k. In SQL, this is still coerced to uppercase, but in the path expression, $k is left in lowercase, so there is a mismatch. To get a variable with a lowercase name, it must be double-quoted in SQL, like this:

```
JSON_VALUE (T1.J, 'lax $.phone [$k]'
  PASSING T2.Huh AS "k")
```

Other variables: Two kinds of variables occur only in special contexts; these are:

1) The keyword `last` is a kind of variable, referencing the last subscript of an array; this will be considered with element accessors later.

2) An at-sign `@` is used in filter expressions to denote the value of the current SQL/JSON item; this will be considered with filter expressions later.

6.9.3 Parentheses

As in SQL and JavaScript, parentheses may be used to override precedence. For example:

```
$a * ($b + 4)
```

The parentheses override the usual precedence that performs multiplication before addition.

6.10 Accessors

The syntax for accessors is:

```
<JSON accessor expression> ::=
    <JSON path primary>
  | <JSON accessor expression> <JSON accessor op>

<JSON accessor op> ::=
    <JSON member accessor>
  | <JSON wildcard member accessor>
  | <JSON array accessor>
  | <JSON wildcard array accessor>
  | <JSON filter expression>
  | <JSON item method>
```

The first four choices are the accessors to be considered in this section. The last two are syntactically similar but will be treated separately for semantic reasons.

So, for present purposes, there are four accessors:

1) Member accessor.

2) Wildcard member accessor.

3) Element accessor.

4) Wildcard element accessor.

These accessors follow these general principles:

1) Accessors are postfix operators so they can be concatenated; they are evaluated from left to right.

2) The first operand of an accessor is evaluated to obtain an SQL/JSON sequence.

3) The second operand specifies which kind of access to perform.

4) The access is performed by iterating over all SQL/JSON items in the value of the first operand.

5) In strict mode, an accessor results in an error if any SQL/JSON item in the sequence fails the access (*e.g.*, member not found, subscript out of range, *etc.*).

6) Lax mode has three techniques to mitigate many errors:

    a) Automatically unwrapping arrays before performing member access.

    b) Automatically wrapping non-arrays in an array before performing element access.

    c) Converting structural errors to empty SQL/JSON sequence.

6.10.1 Member accessor

The syntax for member accessor is:

```
<JSON member accessor> ::=
    <period> <JSON path key name>
  | <period> <JSON path string literal>
```

A member accessor is used to access a member of an object by key name. There are two ways to specify the key name:

1) If the key name does not begin with a dollar sign and meets the JavaScript rules of an Identifier, then the member name can be written in clear text. For example,

```
$.name
$.firstName
$.Phone
```

2) Any key name can be written as a character string literal. This supports member names that begin with a dollar sign or contain special characters. For example:

```
$."name"
$."$price"
$."home address"
```

The semantics are as follows:

1) The first operand is evaluated, resulting in an SQL/JSON sequence of SQL/JSON items.

2) In strict mode, every SQL/JSON item in the SQL/JSON sequence must be an object with a member having the specified key name. If this condition is not met, the result is an error.

3) In lax mode, any SQL/JSON array in the SQL/JSON sequence is unwrapped. Unwrapping only goes one deep; that is, if there is an array of arrays, the outermost array is unwrapped, leaving the inner arrays alone.

4) Iterating over the SQL/JSON sequence, the bound value of each SQL/JSON item corresponding to the specified key name is extracted. (In lax mode, any missing members are passed over silently).

Example: Suppose the context item is:

```
$ = { phones: [ { type: "cell", number: "abc-defg" },
                { number: "pqr-wxyz" },
                { type: "home", number: "hij-klmn" } ] }
```

`$.phones.type` is evaluated in lax mode as follows:

Table 35 — Evaluation of '`$.phones.type`' in lax mode

	Step	Value

1	`$`	`{ phones: [`
		`  { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ] }`

2	`$.phones`	`[ { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ]`

3	`$.phones.type`	`"cell",`
		`"home"`

In the first step, the value is just an SQL/JSON sequence of length 1, the context item.

In the second step, the value is the bound value of the member named `phones`. This is still an SQL/JSON sequence of length 1; the only item is an array.

The third step tries to access the `type` member. However, the SQL/JSON item in the SQL/JSON sequence is an array, not an object. Since this is an array in lax mode, the member accessor first unwraps this SQL/JSON item, giving the following intermediate step:

Table 36 — Intermediate step

	Step	Value

2.1	*`Unwrap`*
	`($.phones)`
		`{ type: "cell", number: "abc-defg" },`
		`{               number: "pqr-wxyz" },`
		`{ type: "home", number: "hij-klmn" }`

(Note that there is no *unwrap* function in the path language; this is an implicit primitive used in lax mode.)

The result of the intermediate step 2.1 is to unwrap the array, producing an SQL/JSON sequence with three SQL/JSON items. Now, the member access for `type` is performed iteratively on each SQL/JSON item of the intermediate result. The first and third SQL/JSON items have a `type` member, but the second does not. The final result (step 3) only retains the bound values for those SQL/JSON items that have a `type` member. The second SQL/JSON item, which lacks a `type` member, is a structural error, which is converted to an empty SQL/JSON sequence in lax mode.

Now let's consider this example in strict mode. Step 1 is evaluated the same as in lax mode. In step 2, a structural error is seen, because the SQL/JSON item is an array rather than an object.

To get past this error, the strict mode user can use the wildcard element accessor presented later. The revised path expression is `$.phones[*].type`, and the evaluation is shown below:

Table 37 — Evaluation of `'$.phones[*].type'`

	Step	Value

1	`$`	`{ phones: [`
		`  { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ] }`

2	`$.phones`	`[ { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ]`

3	`$.phones[*]`	`{ type: "cell", number: "abc-defg" },`
		`{               number: "pqr-wxyz" },`
		`{ type: "home", number: "hij-klmn" }`

4	`$.phones[*]`
	`.type`
		error

The revised path expression still gets an error on step 4, because the second SQL/JSON item in the value of step 3 does not have a `type` member. The data has a loose schema that does not always provide a `type` member. Most likely, this data was not created with a strict mode application in mind. However, a strict mode user can surmount this hurdle by filtering out the SQL/JSON items that do not have a type member, using the path expression `$.phones[*] ? (exists (@.type)).type`. Filters are another capability to be presented later. This version of the path expression is evaluated as shown below:

Table 38 — Evaluation of `'$.phones[*] ? (exists (@.type)).type'`

	Step	Value

1	`$`	`{ phones: [`
		`  { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ] }`

2	`$.phones`	`[ { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ]`

3	`$.phones[*]`	`{ type: "cell", number: "abc-defg" },`
		`{               number: "pqr-wxyz" },`
		`{ type: "home", number: "hij-klmn" }`

	Step	Value
4	`$.phones[*] ? (exists (@.type))`	`{ type: "cell", number: "abc-defg" }, { type: "home", number: "hij-klmn" }`
5	`$.phones[*] ? (exists (@.type)) .type`	`"cell", "home"`

6.10.2 Member wildcard accessor

The BNF is:

```
<JSON wildcard member accessor> ::=
  <period> <asterisk>
```

The semantics are as follows:

1) The first operand is evaluated, resulting in an SQL/JSON sequence of SQL/JSON items.

2) In strict mode, every SQL/JSON item in the SQL/JSON sequence must be an object. If this condition is not met, the result is an error.

3) In lax mode, any SQL/JSON array in the SQL/JSON sequence is unwrapped.

4) Iterating over the SQL/JSON sequence, every bound value of each SQL/JSON object in the SQL/JSON sequence is extracted. (In lax mode, any SQL/JSON items that are not objects are passed over silently.) There is only an implementation-dependent order to members within an object, but the order of objects within the SQL/JSON sequence is preserved in the result.

For example, using the data in the last section, consider the path expression `$.phones.*` in lax mode. The evaluation is shown below:

Table 39 — Evaluation of `'$.phones.*'` in lax mode

	Step	Value
1	`$`	`{ phones: [ { type: "cell", number: "abc-defg" }, { number: "pqr-wxyz" }, { type: "home", number: "hij-klmn" } ] }`
2	`$.phones`	`[ { type: "cell", number: "abc-defg" }, { number: "pqr-wxyz" }, { type: "home", number: "hij-klmn" } ]`
2.1	*`Unwrap`* `($.phones)`	`{ type: "cell", number: "abc-defg" }, { number: "pqr-wxyz" }, { type: "home", number: "hij-klmn" }`

	Step	Value

3	`$.phones.*`	`"cell", "abc-defg", "pqr-wxyz", "home",`
		`"hij-klmn"`

Step 2.1 shows the intermediate step to unwrap the array because of lax mode.

In strict mode, the user must write `$.phones[*].*` to avoid raising an error. The computation is then:

Table 40 — Evaluation of `'$.phones[*].*'`

	Step	Value

1	`$`	`{ phones: [`
		`  { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ] }`

2	`$.phones`	`[ { type: "cell", number: "abc-defg" },`
		`  {               number: "pqr-wxyz" },`
		`  { type: "home", number: "hij-klmn" } ]`

2.1	`$.phones[*]`	`{ type: "cell", number: "abc-defg" },`
		`{               number: "pqr-wxyz" },`
		`{ type: "home", number: "hij-klmn" }`

3	`$.phones[*].*`	`"cell", "abc-defg", "pqr-wxyz", "home",`
		`"hij-klmn"`

6.10.3 Element accessor

The BNF is:

```
<JSON array accessor> ::=
  <left bracket> <JSON subscript list> <right bracket>

<JSON subscript list> ::=
  <JSON subscript> [ { <comma> <JSON subscript> }... ]

<JSON subscript> ::=
    <JSON path wff 1>
  | <JSON path wff 2> to <JSON path wff 3>

<JSON path wff 1> ::=
  <JSON path wff>

<JSON path wff 2> ::=
  <JSON path wff>

<JSON path wff 3> ::=
  <JSON path wff>
```

An element accessor uses square brackets to enclose a comma-separated list of subscripts. The subscripts can be specified in either of two forms:

1) A single numeric value.

2) A range between two numeric values (inclusive) indicated by the keyword `to`.

Following JavaScript conventions rather than SQL conventions, subscripts are 0-relative. Thus, [0] accesses is the first element in an array.

To handle arrays of unknown length, the special variable `last` may be used in a subscript. The value of `last` is the size of the array minus 1. For example, `$[last]` accesses the last element in array `$`; and `$last-1 to last]` accesses the last two elements. This variable can only be used within an array accessor, where it references the innermost array containing `last`.

For example:

```
$[0, last-1 to last, 5]
```

The preceding accesses the first element of `$`, the last two elements of `$`, and the sixth element of `$`. Subscripts can be specified in any order and may contain duplicates.

In strict mode, subscripts must be singleton numeric values between 0 and `last`; in lax mode, any subscripts that are out of bound are simply ignored. In both strict and lax mode, non-numeric subscripts such as `$["hello"]` are an error.

More precisely, the semantics are specified as follows:

1) The first operand is evaluated, yielding an SQL/JSON sequence of SQL/JSON items.

2) In lax mode, any SQL/JSON item in the SQL/JSON sequence that is not an array is wrapped in an array of size 1.

3) In strict mode, it is an error if any SQL/JSON item in the SQL/JSON sequence is not an array.

4) For every SQL/JSON item *I* in the SQL/JSON sequence:

   a) Every subscript is evaluated and subject to implementation-defined rounding or truncation. Note that `last` may have a different value on different arrays in the SQL/JSON sequence (which is why this step is not performed outside the loop on SQL/JSON items). It is an error if any subscript is not a singleton numeric item, even in lax mode.

   b) Each subscript specifies a set of integers (either a single integer, or all integers between the lower and upper bound inclusive).

   c) In strict mode, it is an error if any subscript is less than 0 or greater than `last`. It is also an error when using `to` to specify a range if the lower bound is greater than the upper bound.

   d) The sets of integers are concatenated in the order specified by the user to obtain the final set of subscripts.

   e) The result for *I* is the SQL/JSON sequence of elements in *I* at the positions specified by the final set of subscripts.

5) The overall result is the concatenation of the result for each SQL/JSON item *I* in the input SQL/JSON sequence.

Example: Let the context item be:

```
$ = { sensors:
      { SF: [ 10, 11, 12, 13, 15, 16, 17 ],
        FC: [ 20, 22, 24 ],
        SJ: [ 30, 33 ]
       }
    }
```

Consider the path expression `lax $.sensors.*[0, last, 2]`. The evaluation is:

Table 41 — Evaluation of 'lax $.sensors.*[0, last, 2]'

	Step	Value
1	`$`	`{ sensors: { SF: [10,11,12,13,15,16,17], FC: [20,22,24], SJ: [30,33] } }`
2	`$.sensors`	`{ SF:[10,11,12,13,15,16,17], FC: [20,22,24], SJ: [30,33] }`
3	`$.sensors.*`	`[10,11,12,13,15,16,17], [20,22,24], [30,33]`
4	`$.sensors.*[0,last,2]`	`10,17,12, 20,24,24, 30,33`

Note that in step 3, the second array has 3 elements, so that `last` and 2 select the same element. Thus, in step 4, the element whose value is 24 is selected twice.

Also, in step 3, the third array has 2 elements, so that 2 is out of bounds. In lax mode, this is passed over silently, and only subscript positions 0 and last appear in the final result.

If this was evaluated in strict mode, there would be an error because the third array has a subscript that is out of bounds. To avoid the error, the user might filter out arrays with less than three elements.

6.10.4 Element wildcard accessor

The BNF is:

```
<JSON wildcard array accessor> ::=
  <left bracket> <asterisk> <right bracket>
```

For example, the accessor `$[*]` converts an array into a sequence of all of its elements. In strict mode, the operand must be an array. In lax mode, if the operand is not an array, then one is provided by wrapping it in an array before unwrapping (effectively a no-op on non-array operands).

More precisely, the semantics are specified as follows:

1) The first operand is evaluated, yielding an SQL/JSON sequence of SQL/JSON items.
2) In lax mode, any SQL/JSON item in the SQL/JSON sequence that is not an array is wrapped in an array of size 1.
3) In strict mode, it is an error if any SQL/JSON item in the SQL/JSON sequence is not an array.
4) For every SQL/JSON item *I* in the SQL/JSON sequence: the result for *I* is the sequence of elements of *I*.
5) The overall result is the concatenation of the result for each SQL/JSON item *I* in the input SQL/JSON sequence.

In lax mode, `$[*]` is the same as `$[0 to last]`. In strict mode, there is a subtle difference: `$[0 to last]` actually requires that the array have at least one element (at subscripts `0` and `last`), whereas `$[*]` is not an error in strict mode if `$` is the empty array.

6.10.5 Sequence semantics of the accessors

In review, the input to an accessor is an SQL/JSON sequence. The accessor is applied to each SQL/JSON item in the SQL/JSON sequence in turn and the results are concatenated, preserving order. When applying an accessor to an SQL/JSON item, the result may be an error, or an SQL/JSON sequence of some length (possibly empty, possibly a singleton, possibly longer). Overall, this means that there may be no one-to-one correspondence between the input SQL/JSON items and the output SQL/JSON items.

For example, consider the path expression `$.*[1 to last]` applied in lax mode to the following JSON text:

```
$ = { "x": [ 12, 30 ],
      "y": [ 8 ],
      "z": [ "a", "b", "c" ] }
```

The result of `$.*` is the following SQL/JSON sequence:

```
[ 12, 30 ], [ 8 ], [ "a", "b", "c" ]
```

The next step in the evaluation is shown below:

Table 42 — The step in the evaluation

Input SQL/JSON sequence	**Output SQL/JSON sequence**
`[ 12, 30 ]`	`30`
`[ 8 ]`	
`[ "a", "b", "c" ]`	`"b", "c"`

In the first SQL/JSON item in the SQL/JSON sequence, `last` = 1 (SQL/JSON arrays are 0-relative), so the result is the singleton 30. (Note that the array accessor has removed the container). In the next SQL/JSON item, `last` = 0. The subscript expression `1 to 0` is a structural error but lax mode converts this to an empty

SQL/JSON sequence. In the last row, last = 2 and the result is the last 2 SQL/JSON items of the array. The final result is this SQL/JSON SQL/JSON sequence:

```
30, "b", "c"
```

6.11 Item methods

Item methods are functions that operate on an SQL/JSON item and return an SQL/JSON item. Item methods iterate over an SQL/JSON sequence; therefore they are written like methods as postfix operators on a path expression.

```
<JSON item method> ::=
  <period> <JSON method>

<JSON method> ::=
    type <left paren> <right paren>
  | size <left paren> <right paren>
  | double <left paren> <right paren>
  | ceiling <left paren> <right paren>
  | floor <left paren> <right paren>
  | abs <left paren> <right paren>
  | datetime <left paren> [ <JSON datetime template> ] <right paren>
  | keyvalue <left paren> <right paren>

<JSON datetime template> ::=
  <JSON path string literal>
```

The first two item methods, `type()` and `size()`, can be used to learn type information about the SQL/JSON items in an SQL/JSON sequence. Even in lax mode, these item methods do not unwrap arrays, because if they unwrapped arrays, it would be impossible to learn their type or size.

The other item methods automatically unwrap an array in lax mode.

6.11.1 type()

The `type()` method returns a character string that names the type of the SQL/JSON item. Let *I* be the SQL/JSON item, then *I*.type() is:

— If *I* is the SQL/JSON null, then “null”.

— If *I* is true or false, then “boolean”.

— If *I* is numeric, then “number”.

— If *I* is a character string, then “string”.

— If *I* is an SQL/JSON array, then “array”.

— If *I* is an SQL/JSON object, then “object”.

— If *I* is a datetime, then “date”, “time without time zone”, “time with time zone”, “timestamp without time zone”, or “timestamp with time zone”, as appropriate.

For example, to filter to retain only numeric SQL/JSON items, one might use:

```
lax $.* ? (@.type() == "number")
```

6.11.2 size()

The `size()` item method returns the size of an SQL/JSON item. The size is defined:

- The size of an SQL/JSON array is the number of elements in the array.
- The size of an SQL/JSON object or a scalar is 1.

For example, to filter retain only arrays of size 2 or more, one might use:

```
      strict $.* ? (@.type() == "array" &&  @.size() > 1)
```

Here, strict mode must be used, because the filter operator ? automatically unwraps arrays in lax mode.

6.11.3 Numeric item methods (double, ceiling, floor, abs)

The numeric item methods provide common numeric functions.

- `double()` converts a string or numeric to an approximate numeric value; this is primarily useful to handle character strings containing numbers.
- `ceiling()`, `floor()`, and `abs()` perform the same operations as CEILING, FLOOR, and ABS in SQL.

6.11.4 datetime()

JSON has no datetime types. Datetime values are most likely stored in character strings. [RFC7159] gives no guidance about how to format datetimes in character strings; therefore one can expect that user data exhibits a profusion of formats, including a preference for month/day/year versus a preference for day-month-year or the computer-friendly yearmonthday; and a preference for twelve-hour clock with am/pm vs a twenty-four hour clock.

One way to handle datetimes would be to pull the string out to the SQL level, where products already have functions to interpret datetime strings. However, this means that predicates on datetimes must be expressed in SQL rather than the path language.

It is useful to perform predicates close to the data in the path language. The solution is to augment the SQL/JSON path language with a modest datetime capability. The four ingredients are:

- The SQL/JSON data model is augmented with the SQL datetime types.
- `datetime()` method to convert a character string to an SQL datetime type, optionally using a conversion template.
- Variables passed in to the path engine may be of datetime type.

— Comparison predicates on datetimes are supported.

This functionality is not complete because it lacks datetime arithmetic. Nevertheless, it supports the critical use case of comparison predicates.

The only ingredient listed above that is not already present in SQL is the `datetime()` method. The `datetime()` method is used to convert a character string to a datetime type.

6.11.5 keyvalue()

The keyvalue() method is used to interrogate an SQL/JSON object of unknown schema, by transforming to an SQL/JSON sequence of objects with a known schema.

For example, suppose:

```
      $ = { who: "Fred", what: 64 }
```

Then:

```
      $.keyvalue() =
         ( { name: "who",  value: "Fred", id: 9045 },
           { name: "what", value: 64,     id: 9045 }
         )
```

Looking at this example, the input is a single SQL/JSON object having two members; the output is an SQL/JSON sequence of two SQL/JSON objects having three members. In the result SQL/JSON sequence, the members are:

— `name`, the key name of a member *M* in the input object.

— `value`, the bound value of *M*.

— `id`, an implementation-dependent integer that is a unique identifier for the input SQL/JSON object.

Since members of an object are unordered, the order of the result SQL/JSON sequence is implementation-dependent.

Using `keyvalue()`, a path expression to learn the key names in the input is:

```
      $.keyvalue().name
```

and the result is the SQL/JSON sequence:

```
      ("who", "what")
```

Note that in lax mode, `keyvalue()` unwraps its input. For example, suppose:

```
      $ = [ { who: "Fred", what: 64 },
            { who: "Moe", how: 22 } ]
```

Then:

```
      lax $.keyvalue() =
         ( { name: "who", value: "Fred", id: 8394 },
           { name: "what", value: 64, id: 8394 },
```