# Title: SQL/JSON part 2 - Querying JSON

Author: Fred Zemke, Beda Hammerschmidt, Krishna Kulkarni, Zhen Hua Liu, Doug McMahon, Jim Melton, Jan-Eike Michels, Fatma Özcan, Hamid Pirahesh

Source: U.S.A.
Status: Change proposal
Date: March 4, 2014

## Abstract

This paper addresses Comment #257, P02-USA-950, by proposing facilities for querying JSON using SQL.

## References

| | |
|---|---|
| [SQL-92] | Database language SQL", ANSI X3.135-1992 |
| [Foundation:1999] | Jim Melton (ed), "ISO International Standard (IS) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC 9075-2:1999 |
| [Foundation 7CD1] | Jim Melton (ed), "Committee Draft (CD) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC JTC1/SC32 WG3:USN-003 = ANSI INCITS DM32.2-2012-0Onnn |
| [Foundation 7IWD4] | Jim Melton (ed), "Informal Working Draft (IWD) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC JTC1/SC32 WG3:PEK-003 = ANSI INCITS DM32.2-2013-200 |
| [SQL/XML:2011] | Jim Melton (ed), "ISO International Standard (IS) Database Language SQL - Part 14: XML-Related Specifications (SQL/XML)", ISO/IEC 9075-14:2011 |
| [XQuery 1.0.2] | "XQuery 1.0: An XML Query Language (Second Edition)", http://www.w3.org/TR/xquery/ |
| [SQL/JSON part 1] | Jim Melton, "SQL/JSON part 1", ANSI INCITS DM32.2-2014-00024 = ISO/IEC JTC1/SC32 WG3:PEK-nnn |
| [RFC 2119] | Internet Engineering Task Force, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119 https://www.ietf.org/rfc/rfc2119.txt |
| [RFC 4627] | Internet Engineering Task Force, *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC 4627 http://tools.ietf.org/html/rfc4627 |

| | |
|---|---|
| [ECMA-262 5.1] | "ECMAscript language specification 5.1", June 2011, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf |
| [JSON.org] | http://www.json.org/ |
| [JPath] | http://projects.plural.cc/projects/jsonij/wiki/JPath |
| [JSONPath] | http://goessner.net/articles/JsonPath/ |
| [JAQL] | http://code.google.com/p/jaql/wiki/JaqlOverview |
| [JSONiq] | http://www.jsoniq.org/ |
| [Mongo] | http://docs.mongodb.org/manual/ |
| [AVRO] | http://avro.apache.org/ |
| [AVRO spec] | http://avro.apache.org/docs/current/spec.html |
| [BSON spec] | http://bsonspec.org/#/specification |

# 1. Introduction

This paper addresses Comment #257, P02-USA-950, which says

> 1-Major Technical
>
> P02-No specific location
>
> JavaScript Object Notation (popularly known as JSON) [RFC4627] is becoming increasingly important in Web- and Cloud-based communities for exchanging data between Web-based applications. This has in turn led to the demands for storage, retrieval, querying, and manipulation of JSON data in the context of SQL It is important that SQL respond to these requirements by providing appropriate extensions for handling JSON data.

The companion paper [SQL/JSON part 1] provides an introduction to JSON and proposes SQL operators to construct JSON values stored in either character or binary strings. This paper provides support for querying JSON data in SQL.

The discussion sections in this paper are organized as follows:

1. Section 2. "Overview of the API" on page 3.
2. Section 3. "Formal discussion of the API" on page 26
3. Section 4. "Data model" on page 39
4. Section 5. "Path language" on page 43

# 2. Overview of the API

This section presents an overview of the proposed features for querying JSON using SQL. This is an informal treatment, in which syntax and concepts are introduced as they are encountered, without attempting to be complete and thorough. After the overview, we will provide a formal treatment of all proposed features.

We propose five operators to query JSON using SQL:

| | |
|---|---|
| IS JSON | — predicate to test that a string contains JSON |
| JSON_VALUE | — extract an SQL value from a JSON value |
| JSON_TABLE | — convert a JSON value to an SQL table |
| JSON_EXISTS | — predicate to determine if a JSON value contains some information |
| JSON_QUERY | — extract a JSON value from a JSON value |

## 2.1 Validation — IS JSON predicate

The first step in handling JSON will frequently be to validate that a string (character or binary) conforms to the JSON specification [RFC 4627]; this is satisfied by the IS JSON predicate, exemplified:

```
WHERE T.C IS JSON
```

In general terms, this predicate is ***True*** if T.C is a string (character or binary) that conforms to [RFC 4627]. The encoding is deduced from the data (from the character set or from the first four bytes of the binary string).

There are two important issues to understand:

— unique key names

— SQL null values

### 2.1.1 Unique key names

The first issue is that [RFC 4627] section 2.2 “Objects” says

> The names within an object SHOULD be unique.

The all-caps word SHOULD is defined in [RFC 2119]

> 3. SHOULD This word, or the adjective "RECOMMENDED", mean that there
> may exist valid reasons in particular circumstances to ignore a particular item,
> but the full implications must be understood and carefully weighed before
> choosing a different course

Thus it is recommended that JSON objects should have unique key name, but it is not required. We decided that there may exist JSON texts that do not enforce this uniqueness constraint, and also that users may be interested in enforcing the uniqueness constraint. Therefore we propose two options on the IS JSON predicate, to either enforce the constraint or not. The two choices are illustrated as follows:

```
T.C IS JSON WITH UNIQUE KEYS
T.C IS JSON WITHOUT UNIQUE KEYS
```

Since enforcing a constraint is costly, the default is not to check, that is, T.C IS JSON is equivalent to T.C IS JSON WITHOUT UNIQUE KEYS.

### 2.1.2 SQL null values

The other issue is how to handle a null input to the IS JSON predicate, such as

```
CAST (NULL AS CLOB) IS JSON
```

A null value is not a string conforming to [RFC 4627], so one might think that the result should be *False*. However, there is a long precedent in SQL that most predicates return *Unknown* on null input, and we propose that behavior for the IS JSON predicate.

There are two important use cases to consider:

— table constraint to insure that only conformant JSON is inserted in a column

— WHERE clause to filter rows to insure that only conformant JSON is selected

**Constraints:** a user may wish to insure that a column contains only JSON. The most natural approach seems to be

```
CREATE TABLE T (
  D INTEGER CHECK (D > 0),
  J CLOB CHECK (J IS JSON)
)
```

Question: What does the user do if the user wishes to insert a row and currently knows the value of D but does not know the value of J? The usual approach to this situation is to insert a null value as a place holder, indicating that the correct value is not currently known, but may be supplied later. So the user performs

```
INSERT INTO T (D, J) VALUES (1, NULL)
```

This operation will succeed, because the check constraint will evaluate to *Unknown,* which is sufficient; only *False* rows are rejected by a check constraint.

The check constraint on column D is analogous; it requires a positive integer. Consider this insertion:

```
INSERT INTO T (D, J) VALUES (NULL, '[]')
```

In this instance, the null that is inserted into D is not a positive integer. The check constraint on D evaluates to *Unknown* so the row can be inserted.

At this point our table is the following contents

| D | J |
|---|---|
| 1 | |
| | [] |

and both rows pass the constraints, because constraints allow *Unknown* predicate values.

**WHERE clause:** Continuing the same example, the user wishes to find the rows that contain JSON. The user writes

```
SELECT D, J
FROM T
WHERE J IS JSON
```

The WHERE clause only passes rows in which the <search condition> is *True*. Thus the result of this query is

| D | J |
|---|---|
| | [] |

The summary is that the proposed behavior for IS JSON is consistent with existing SQL predicates and meets user expectations for both constraints and the WHERE clause.

## 2.2 SQL/JSON path language

The remaining SQL/JSON query operators require a query language to interrogate JSON values. The basic design has much in common with [SQL/XML:2011]. We propose JSON_TABLE, analogous to XMLTABLE, as probably the most important operator for querying JSON, since this operator converts JSON into a relational table in the FROM clause. Like XMLTABLE, JSON_TABLE will use a row pattern to describe the rows that are selected from a JSON value, and column patterns to describe the columns.

When designing XMLTABLE, the query language for the row pattern and column pattern clearly had to be XQuery, since this was a referenceable query language with considerable “buy-in” in the market. However, with JSON, things are not so simple. While JSON itself is referenceable [RFC 4627], there is no standards-setting organziation that has assumed resonsibility for querying JSON. Thus we immediately confronted the problem of what path language to use for JSON.

We began by reviewing available query languages for JSON. We found the following:

| | |
|---|---|
| • JPath | [JPath] |
| • JSONPath | [JSONPath] |
| • JAQL | [JAQL] |
| • JSONiq | [JSONiq] |

- Mongo [Mongo]

Although we found interesting ideas in each of these, we do not believe that any of them has emerged as a “winner” in the marketplace. Consequently, we decided to define our own place-holder query language, which is called the SQL/JSON path language in the proposal. The design allows future JSON query languages as alternatives to our “home brew” language.

The examples in this informal overview will include examples of the SQL/JSON path language. The path language examples will be explained informally as they are encountered. The complete SQL/JSON path language will be presented in a later section.

At the outset, it will be helpful to understand a few basics about the SQL/JSON path language. This language uses a data model similar to XQuery: a data model that supports ordered sequences. The elements of an SQL/JSON sequence are called SQL/JSON items. SQL/JSON items can be non-null SQL scalar values (of character, numeric, boolean and datetime types), a distinct SQL/JSON null value (standing alone in its own type), and combinations of these primitives built up using the SQL/JSON arrays and object consturctors. An SQL/JSON sequence may be empty or of any positive length. The SQL/JSON items of an SQL/JSON sequence need not conform to any common schema. We provide a formal description of the data model later.

## 2.3 Search — JSON_EXISTS

A common task when querying a database is to locate rows that have a certain property. If say a table has a column containing JSON data, then the user may be interested in finding the rows in which the JSON data satisfies a particular predicate. We propose JSON_EXISTS for this task.

JSON_EXISTS is a predicate that can be used to test whether an SQL/JSON path expression is fulfilled in a 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 a table T with two columns, K (the primary key) and J (a column containing JSON text)

**T**

| K | J |
|---|---|
| 101 | { "who": "Fred", "where": "Oracle",<br>"friends": [ { "name": "Lili", "rank": 5 }, {"name": "Hank", "rank": 7} ] } |
| 102 | { "who": "Tom", "where": "IBM",<br>"friends": [ { "name": "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] } |
| 103 | { "who": "Jack",<br>"friends": [ { "name": "Connie" } ] } |
| 104 | { "who": "Joe",<br>"friends": [ { "name": "Doris" }, {"rank": 1} ] } |

**T**

| K | J |
|---|---|
| 105 | { "who": "Mabel", "where": "Black Label", "friends": [ { "name": "Buck", "rank": 6} ] } |
| 106 | { "who": "Louise", "where": "Iana" } |

Looking at the sample data, we see 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. Imlementations are free to choose either of these as their default.

After the mode declaration (`lax` or `strict`) we reach the meat 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 it to the invoking routine. We will consider strict mode 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

| K |
|---|
| 101 |

| K |
|---|
| 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 which case rows 103 and 104 will have errors in the path expression, which might be presented to the user as exceptions. We believe that the user will want the ability to handle exceptions so that the query can run to completion rather than halting on an exception. Thus we provide an ON ERROR clause 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. This is the default error behavior for JSON_EXISTS. Other choices are TRUE ON ERROR, UNKNOWN ON ERROR or ERROR ON ERROR.

The keywords `lax` and `strict` are mandatory syntax in the proposal; this will permit implementations to choose their own default.

In the examples considered above, the SQL/JSON path expression `$.where` will find either 0 or 1 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, 103 has no `rank` member (and 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. Let us look at how this SQL/JSON path expression is evaluated in row 101:

| | path step | result |
|---|---|---|
| 1 | `$` | `{ "who": "Fred", "where": "Oracle", "friends": [ { "name": "Lili", "rank": 5 }, {"name": "Hank", "rank": 7} ] }` |

| | path step | result |
|---|---|---|
| 2 | `$.friends` | `[ { "name": "Lili", "rank": 5 },`<br>`{ "name": "Hank", "rank": 7} ]` |
| 3 | `$.friends[*]` | `{ "name": "Lili", "rank": 5 },`<br>`{ "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 we lose the square bracket `[ ]` array wrappers, and the result at this point is an SQL/JSON sequence of length 2. 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 2.

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 Mongo it is customary for a member accessor such as `.rank` to automatically iterate over the elements of an array such as `friends`. Thus the Mongo user would prefer to write `$.friends.rank`, as if there were an implicit `[*]` on `friends`. We propose to support this convention in lax mode. Thus the path expression `'lax $.friends.rank'` finds the same rows as `'strict $.friends[*].rank'` in this example. Thus, 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 we cover the path language.

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

## 2.4 Scalar extraction — JSON_VALUE

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 `name` member from each row:

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

with the following result from our sample data:

| 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.

We can also extract the `where` member from each row. 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:

| K | WHO | NALI |
|---|---|---|
| 101 | Fred | Oracle |
| 102 | Tom | IBM |
| 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

| K | WHO | NALI |
|---|---|---|
| 101 | Fred | Oracle |
| 102 | Tom | IBM |
| 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 1 SQL/JSON item; the ON EMPTY clause can be used to handle missing data (0 SQL/JSON items) gracefully. More than 1 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:

| K | WHO | NALI | FRIEND |
|---|---|---|---|
| 101 | Fred | Oracle | *** error *** |
| 102 | Tom | IBM | *** error *** |
| 103 | Jack | | Connie |
| 104 | Joe | | Doris |

| K | WHO | NALI | FRIEND |
|---|---|---|---|
| 105 | Mabel | Black Label | Buck |
| 106 | Louise | Iana | |

Rows 101 and 102 have an error because the path expression `$.friends.name` returns more than 1 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:

| K | WHO | NALI | FRIEND |
|---|---|---|---|
| 101 | Fred | Oracle | *** error *** |
| 102 | Tom | IBM | *** error *** |
| 103 | Jack | | Connie |
| 104 | Joe | | *** error *** |
| 105 | Mabel | Black Label | Buck |

| K | WHO | NALI | FRIEND |
|---|---|---|---|
| 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 [ECMA-262 5.1] that arrays begin at subscript 0. This is one instance where we decided it was better to follow the conventions of the JSON community rather than SQL. The example has the following result:

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

## 2.5 Extracting JSON from JSON — JSON_QUERY

Instead of extracting an SQL scalar value, the user may wish to extract a JSON value. For example, the friends member is an array, which cannot be retrieved in full using JSON_VALUE. To start with, let's use a JSON_EXISTS clause to filter out the annoying error cases when friends does not exist:

```
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 | Oracle | [ { "name": "Lili", "rank": 5 }, {"name": "Hank", "rank": 7} ] |
| 102 | Tom | IBM | [ { "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 let us consider the various error cases that can arise. One possibility is that the SQL/JSON path expression returns an empty SQL/JSON sequence. Similar to JSON_VALUE, we propose an ON EMPTY clause to handle the empty case. Thus to handle row 106, we 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 | Oracle | [ { "name": "Lili", "rank": 5 }, {"name": "Hank", "rank": 7} ] |
| 102 | Tom | IBM | [ { "name": "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] |
| 103 | Jack | | [ { "name": "Connie" } ] |
| 104 | Joe | | [ { "name": "Doris" }, {"rank": 1} ] |

| K | WHO | NALI | FRIENDS | |
|---|---|---|---|---|
| 105 | Mabel | Black Label | [ { "name": "Buck", "rank": 6} ] } | |
| 106 | Louise | Iana | | ← |

NULL ON EMPTY is the default behavior (so the example shows explicit syntax for the default). Another alternative is ERROR ON EMPTY. We did not think it would be useful for the user to provide literals for JSON output; instead we propose EMPTY ARRAY ON EMPTY and EMPTY OBJECT ON EMPTY.

Another 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 results in two names in rows 101 and 102. For this kind of situation, [JSONPath] provides a useful solution, which is to wrap the results 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 | Oracle | [ "Lili", "Hank" ] | |
| 102 | Tom | IBM | [ "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 the SRs actually prohibit the ON EMPTY clause 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 supplies the array wrapper if the path expression results in anything other than a singleton SQL/JSON array or object.

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

**T2**

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

Note in this data that member `a` has a value that is a character string, whereas member `b` has a value that is an array. Here are some key combinations of SQL/JSON path expression and API syntax:

| 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. They will be handled by using the ON ERROR clause, already discussed in the case of JSON_VALUE. 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.

## 2.6 JSON_TABLE

The examples above show that it is possible to write simple reports with JSON_EXISTS to search for specific rows and JSON_VALUE to report specific scalar values in those rows. JSON_TABLE provides a convenient shorthand for such reports, as well as more complicated reports of nested data structures.

### 2.6.1 Example data

The examples for JSON_TABLE have the following overall schema:

```
libraries [
    branch,
    books [
        title
        authors [
            name ]
        topics [ ]
        ]
    phones [
        type,
        number ]
    librarians [
        name ]
]
```

Here is the sample data:

```
{ libraries:
  [ { branch: "FC",
      books:
      [ { title: "abc",
          authors:
          [ {name: "Y"}, {name: "Z"} ]
          topics: ["love", "death", "taxes"]
        },
        { title: "def",
          authors:
          [ {name: "A"}, {name: "B"} ]
        }
      ],
      phones:
      [ { type:"desk", number:"rtyu" },
        { type:"fax", number:"yuio" }
      ],
      librarians:
      [ { name:"iop" }, { name:"cvb" }
      ]
    },
    { branch: "SF",
      books:
      [ { title: "pqr",
          authors:
          [ {name: "P"}, {name: "Q"} ]
        },
        { title: "stu",
          authors:
          [ {name: "S"}, {name: "T"} ],
          topics: ["war", "salami"]
        },
        { title: "xxx” }
      ],
      librarians:
      [ { name:"asd" }, { name:"bnm" }
      ]
    },
    { branch: "XX",
      phones:
      [ {type: "voice", number:"dfgh" }
  ]
}
```

There are several places where arrays are missing. Branch FC has books, phones and librarians; branch SF has books and librarians but no phones; branch XX only has a phone. Every book has a title, but authors and topics are spotty.

NOTE: [RFC 4627] does not permit member keys without enclosing quotes. Many JSON processors accept such data, and I followed that relaxed convention in the preceding data.

### 2.6.2 Simple report with no nesting

At the outermost level, the data only has branch name. To get a list of branch names, the user could write this report:

```
SELECT branch
FROM T,
JSON_TABLE (T.C,
  'lax $.libraries[*]' COLUMNS
  ( idx FOR ORDINALITY,
    branch VARCHAR(20) PATH '$.branch',
  )
) AS JT
```

with the following result:

| IDX | BRANCH |
|---|---|
| 1 | FC |
| 2 | SF |
| 3 | XX |

The ordinality column is a sequential numbering of rows, which is 1-based, in keeping with the precedent set by UNNEST, ROW_NUMBER windown function and XMLTABLE.

### 2.6.3 Nested reports: design philosophy

With this data, the user cannot drill very deeply because most of the data is found in nested data structures. The user will want some way to get at this nested data.

In [SQL/XML:2011], the XMLTABLE operator supports this use case through chaining: an XML value can be exported from one XMLTABLE, to serve as input for another XMLTABLE which can then "drill down" deeper in a hierarchy.

We considered this approach for JSON_TABLE. However, there were two issues:

1. We have heard feedback from users of XMLTABLE that chaining is hard to do, especially if an outer join is desired, which is actually the most common desire.
2. Chaining from one JSON_TABLE to another would potentiatlly expose the SQL/JSON data model to the user. This was not a problem in [SQL/XML:2011] because of the availability of the XML(SEQUENCE) type. However, with JSON, we decided to avoid adding SQL types.

Therefore, we decided to take a different approach, which we will now consider.

Let's suppose the user wishes to obtain a complete dump of the data (without ordinality information this time for conciseness). Here is the example to do this:

```
SELECT branch, title, aname, topic, type, number, lname
FROM T,
JSON_TABLE (T.C,
  'lax $.libraries[*]' COLUMNS
  ( branch VARCHAR(20) PATH 'lax $.branch',
    NESTED PATH 'lax $.books[*]' COLUMNS
    ( title VARCHAR(20) PATH 'lax $.title',
      NESTED PATH 'lax $.authors[*]' COLUMNS
      ( aname VARCHAR(20) PATH 'lax $.name'
      ),
      NESTED PATH 'lax $.topics[*]' COLUMNS
      ( topic VARCHAR(20) PATH 'lax $'
      )
    ),
    NESTED PATH 'lax $.phones[*]' COLUMNS
    ( type VARCHAR(20) PATH 'lax $.type',
      number VARCHAR(20) PATH 'lax $.number'
    ),
    NESTED PATH 'lax $.librarians' COLUMNS
    ( lname VARCHAR(20) PATH 'lax $.name'
    )
  )
) AS JT
```

The new feature in this example is the nested COLUMNS clause. In this example, the user has chosen to explode every level of the hierarchy in the data.

For hierarchical reporting, we set ourselves the following objectives:

1) Support nested COLUMNS clause to mirror the nesting within the data.

2) Provide syntax to support both inner and outer join cases between parent and child COLUMNS clauses

2) Permit more than one nested COLUMNS clause at any depth. Sibling nested COLUMNS can be related either as cross product of union join

The difference between cross product and union join is illustrated below:

**T1**

| C1 |
|---|
| A |
| B |

**T2**

| C2 |
|---|
| X |
| Y |

**CROSS PRODUCT**

| C1 | C2 |
|---|---|
| A | X |
| A | Y |
| B | X |
| B | Y |

**UNION JOIN**

| C1 | C2 |
|---|---|
| A | |
| B | |
| | X |
| | Y |

(UNION JOIN was a variety of <joined table> syntax supported in [SQL-92]. It was dropped from [Foundation:1999] because it is equivalent to a FULL OUTER JOIN with an unsatisfiable condition such as ON 0=1.)

To recap, our objective is to support:

a) either INNER or LEFT OUTER JOIN semantics in parent/child relationships

b) either CROSS or UNION in sibling relationships.

Now, what syntax shall we use to support these output plans? We could conceivably add syntactic decorations to the nested COLUMNS clauses. However, when we tried to sketch it out, the syntax looked unwieldy, confusing, and therefore not very user friendly.

Instead, our proposal is as follows:

A) we choose OUTER as the default parent/child relationship

B) we choose UNION as the default sibling relatonship

C) to override the defaults, we provide an optional PLAN clause.

### 2.6.4 Nested report with default plan

Once again, the example with the default plan is written:

```
SELECT branch, title, aname, topic, type, number, lname
FROM T,
JSON_TABLE (T.C,
  'lax $.libraries[*]' COLUMNS
  ( branch VARCHAR(20) PATH 'lax $.branch',
    NESTED PATH 'lax $.books[*]' COLUMNS
    ( title VARCHAR(20) PATH 'lax $.title',
      NESTED PATH 'lax $.authors[*]' COLUMNS
      ( aname VARCHAR(20) PATH 'lax $.name'
      ),
      NESTED PATH 'lax $.topics[*]' COLUMNS
      ( topic VARCHAR(20) PATH 'lax $'
      )
    ),
    NESTED PATH 'lax $.phones[*]' COLUMNS
    ( type VARCHAR(20) PATH 'lax $.type',
      number VARCHAR(20) PATH 'lax $.number'
    ),
    NESTED PATH 'lax $.librarians' COLUMNS
    ( lname VARCHAR(20) PATH 'lax $.name'
    )
  )
) AS JT
```

With the proposed defaults, we have outer joins in all parent/child relationships, and union joins in all siblings. The results with the sample data are:

| branch | title | aname | topic | type | number | lname |
|---|---|---|---|---|---|---|
| FC | abc | Y | | | | |
| FC | abc | Z | | | | |
| FC | abc | | love | | | |
| FC | abc | | death | | | |
| FC | abc | | taxes | | | |
| FC | def | A | | | | |
| FC | def | B | | | | |
| FC | | | | desk | rtyu | |
| FC | | | | fax | yuio | |
| FC | | | | | | iop |
| FC | | | | | | cvb |
| SF | pqr | P | | | | |
| SF | pqr | Q | | | | |
| SF | stu | S | | | | |
| SF | stu | T | | | | |
| SF | stu | | war | | | |
| SF | stu | | salami | | | |
| SF | | | | | | asd |
| SF | | | | | | bnm |
| XX | | | | voice | dfgh | |

### 2.6.5 Explicit plans

To get other output plans, we propose to give the row pattern path and every nested COLUMNS path a path name, and then describe the plan for joining them using a PLAN clause. In this example, let us name the paths L (libraries), B (books), A (authors), T (topics), P (phones) and E (librarians = employees). The default choice above is equivalent to the following explicit plan (additional syntax is underlined):

```
    SELECT branch, title, aname, topic, type, number, lname
```

```
FROM T,
JSON_TABLE (T.C,
  'lax $.libraries[*]' AS L COLUMNS
  ( branch VARCHAR(20) PATH 'lax $.branch',
    NESTED PATH 'lax $.books[*]' AS B COLUMNS
    ( title VARCHAR(20) PATH 'lax $.title',
      NESTED PATH 'lax $.authors[*]' AS A COLUMNS
      ( aname VARCHAR(20) PATH 'lax $.name'
      ),
      NESTED PATH 'lax $.topics[*]' AS T COLUMNS
      ( topic VARCHAR(20) PATH 'lax $'
      )
    ),
    NESTED PATH 'lax $.phones[*]' AS P COLUMNS
    ( type VARCHAR(20) PATH 'lax $.type',
      number VARCHAR(20) PATH 'lax $.number'
    ),
    NESTED PATH 'lax $.librarians' AS E COLUMNS
    ( lname VARCHAR(20) PATH 'lax $.name'
    )
  )
  PLAN (L OUTER ((B OUTER (A UNION T)) UNION P UNION E))
) AS JT
```

Admittedly the PLAN clause takes a little work to decode. However, we think it would be far harder to understand if this information was sprinkled around on the NESTED PATH clauses.

### 2.6.6 CROSS instead of UNION

Now let's look at some variations on the default plan. The first variation is to use CROSS instead of UNION when joining authors and topics of books. The revised query is

```
SELECT branch, title, aname, topic, type, number, lname
FROM T,
JSON_TABLE (T.C,
  'lax $.libraries[*]' AS L COLUMNS
  ( branch VARCHAR(20) PATH 'lax $.branch',
    NESTED PATH 'lax $.books[*]' AS B COLUMNS
    ( title VARCHAR(20) PATH 'lax $.title',
      NESTED PATH 'lax $.authors[*]' AS A COLUMNS
      ( aname VARCHAR(20) PATH 'lax $.name'
      ),
      NESTED PATH 'lax $.topics[*]' AS T COLUMNS
      ( topic VARCHAR(20) PATH 'lax $'
      )
    ),
    NESTED PATH 'lax $.phones[*]' AS P COLUMNS
    ( type VARCHAR(20) PATH 'lax $.type',
      number VARCHAR(20) PATH 'lax $.number'
    ),
    NESTED PATH 'lax $.librarians' AS E COLUMNS
    ( lname VARCHAR(20) PATH 'lax $.name'
    )
  )
  PLAN (L OUTER ((B OUTER (A CROSS T)) UNION P UNION E))
) AS JT
```

The resulting output is

| branch | title | aname | topic | type | number | lname |
|---|---|---|---|---|---|---|
| FC | abc | Y | love | | | |
| FC | abc | Y | death | | | |
| FC | abc | Y | taxes | | | |
| FC | abc | Z | love | | | |
| FC | abc | Z | death | | | |
| FC | abc | Z | taxes | | | |
| FC | def | | | | | |
| FC | | | | desk | rtyu | |
| FC | | | | fax | yuio | |
| FC | | | | | | iop |
| FC | | | | | | cvb |
| SF | pqr | | | | | |
| SF | stu | S | war | | | |
| SF | stu | S | salami | | | |
| SF | stu | T | war | | | |
| SF | stu | T | salami | | | |
| SF | | | | | | asd |
| SF | | | | | | bnm |
| XX | | | | voice | dfgh | |

Compared to the prior results, observe that (branch: FC title: def) is still in the output, but without author name or topic. This books has authors but no topics, so the cross product of authors and topics is empty. Nevertheless, the book title itself still appears, because B (books) is outer joined to A CROSS T (authors cross topics). Similarly, (branch: SF title: pqr) appears with no author or topics for the same reason.

### 2.6.7 INNER instead of OUTER

As the next variation, we might ask for an inner join between B (books) and A CROSS T (authors cross topics), like this:

```
SELECT branch, title, aname, topic, type, number, lname
FROM T,
JSON_TABLE (T.C,
  'lax $.libraries[*]' AS L COLUMNS
  ( branch VARCHAR(20) PATH 'lax $.branch',
    NESTED PATH 'lax $.books[*]' AS B COLUMNS
    ( title VARCHAR(20) PATH 'lax $.title',
      NESTED PATH 'lax $.authors[*]' AS A COLUMNS
      ( aname VARCHAR(20) PATH 'lax $.name'
      ),
      NESTED PATH 'lax $.topics[*]' AS T COLUMNS
      ( topic VARCHAR(20) PATH 'lax $'
      )
    ),
    NESTED PATH 'lax $.phones[*]' AS P COLUMNS
    ( type VARCHAR(20) PATH 'lax $.type',
      number VARCHAR(20) PATH 'lax $.number'
    ),
    NESTED PATH 'lax $.librarians' AS E COLUMNS
    ( lname VARCHAR(20) PATH 'lax $.name'
    )
  )
  PLAN (L OUTER ((B INNER (A CROSS T)) UNION P UNION E))
) AS JT
```

This will now remove (FC, def) and (SF, pqr) from the result, like this

| branch | title | aname | topic | type | number | lname |
|---|---|---|---|---|---|---|
| FC | abc | Y | love | | | |
| FC | abc | Y | death | | | |
| FC | abc | Y | taxes | | | |
| FC | abc | Z | love | | | |

| branch | title | aname | topic | type | number | lname |
|---|---|---|---|---|---|---|
| FC | abc | Z | death | | | |
| FC | abc | Z | taxes | | | |
| FC | | | | desk | rtyu | |
| FC | | | | fax | yuio | |
| FC | | | | | | iop |
| FC | | | | | | cvb |
| SF | stu | S | war | | | |
| SF | stu | S | salami | | | |
| SF | stu | T | war | | | |
| SF | stu | T | salami | | | |
| SF | | | | | | asd |
| SF | | | | | | bnm |
| XX | | | | voice | dfgh | |

# 3. Formal discussion of the API

## 3.1 No new types

As already stated, we do not propose a JSON type for SQL, nor any other metadata to indicate a JSON value. Instead, JSON values may be stored in either character or binary strings, in either the original Unicode encoding specified in [RFC 4627] or in other encodings such as [AVRO] or [BSON spec]. Since the latter are not referenceable from a standards perspective, we leave them as implementation-defined extensions.

Since there is no metadata to indicate a JSON value, the fact that a value is JSON is sometimes indicated from context (e.g., positionally in an argument list) or else via an explicit declaration using the <JSON format clause> introduced in [SQL/JSON part 1].

## 3.2 Common API syntax

As outlined above, we propose four operators for querying JSON:

JSON_EXISTS — determine if a JSON value satisfies a search criterion

JSON_VALUE — extract an SQL scalar from a JSON value

JSON_QUERY — extract a JSON value from a JSON value

JSON_TABLE — extract an SQL table from a JSON value

These four operators share the need to specify their inputs and the SQL/JSON path expression to be evaluated. All four operators use the same syntax at the beginning of their parameter list:

```
<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>
```

### 3.2.1 <JSON context item>

The <JSON context item> is the JSON input that the SQL/JSON path expression will operate on. In the introductory examples already seen, the context item is usually T.J.

### 3.2.2 <JSON value expression>

<JSON context item> is defined as a <JSON value expression>. <JSON value expression> was introduced in [SQL/JSON part 1]. It is defined as

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

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

<JSON input representation> ::=
       JSON
     | <implementation-defined JSON representation option>
```

The role of <JSON value expression> is to handle a <value expression> which needs to be passed into the SQL/JSON data model. We present the data model later in Section 4. "Data model" on page 39, but for now, it will suffice to say that the main issue is whether a <value expression> needs to be parsed as JSON or perhaps some implementation-defined format such as BSON or AVRO. Recall that we have not extended the type system with a JSON type, therefore the declared type of a <value expression> does not indicate whether it contains JSON. Instead, we need either explicit syntax or contextual clues to make that determination.

The <JSON input clause> syntax is available to specify the format of a <value expression>. We propose to support JSON text stored either in character strings or in binary strings. [RFC 4627] section 3 "Encoding" specifies how the Unicode encoding can be deduced from the first four bytes when the data is stored in a binary string, so no SQL syntax is required to handle this detail.

There are other storage formats in wide-spread use, such as AVRO and BSON. However, these other storage formats are not defined by standards-defining organizations, so they are not referenceable in the SQL standard. Nevertheless, we believe that implementations will want to support such formats. The workaround is that <JSON input representation> is defined as

```
<JSON input representation> ::=
       JSON
     | <implementation-defined JSON representation option>
```

The keyword JSON is used in syntax to mark data that is formatted according to [RFC 4627]. In all of the examples presented so far, this choice has been the implicit default. Implementations can define their own alternatives, such as keywords AVRO or BSON, to mark data with explicit syntax to indicate non-standard formats.

The <JSON input clause> provides explicit syntax to designate that a <value expression> must be parsed as JSON (or some implementation-defined format such as BSON).

If the <JSON input clause> is omitted, this usually indicates a non-JSON input which is passed into the SQL/JSON model without parsing. However, there are two contexts in which a <JSON input clause> can be inferred:

1. If the <value expression> is a JSON-returning function *JRF* (defined as a JSON consructor function from [SQL/JSON part 1] or JSON_QUERY in this paper) then the <JSON input clause> can be inferred from the RETURNING clause of *JRF*. This simplifies nested funtion invocations, since the format does not need to be stated both in the RETURNING clause of an inner JSON-returning function and again in a FORMAT clause for input argument to the surrounding function invocation.
2. The <JSON context item> defaults to JSON, unless it is already handled by the preceding inference.

### 3.2.3 Path expression

The context item is followed by a comma and the SQL/JSON path expression. Note that <JSON path specification> is a character string literal. Requiring a literal will enable 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 considered later.

### 3.2.4 PASSING clause

Similar to the SQL/XML query functions, we propose a PASSING clause. None of the examples considered so far have used a PASSING clause. The PASSING clause is used to pass additional parameters to the SQL/JSON path expression. We anticipate that the main use for the PASSING clause will be to specify filtering. For example, suppose that a 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,
  '$ ? ($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>

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

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 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 a 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 the examples, 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.

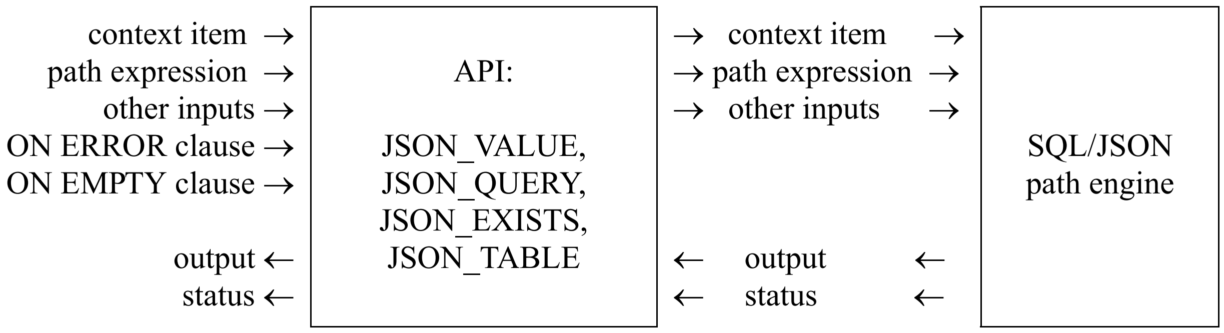
A <JSON value expression> may have an optional <JSON input clause>. This indicates that the values expression should be parsed as JSON. The standardized option is FORMAT JSON; implementations may also support syntax such as FORMAT AVRO or FORMAT BSON. When using the <JSON input clause>, the <value expression>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

### 3.2.5 ON ERROR and ON EMPTY syntax

The API entry points also have ON ERROR and ON EMPTY clauses in common. However, the details of these clauses vary depending on the operator, so they are not included in the <JSON API common syntax> shown above. They will be considered individually for each API entry point later.

## 3.3 Processing architecture

The four JSON query operators will conceptually evaluate an SQL/JSON path expression in the following processing architecture:



The inputs to a JSON query operator are shown in the upper left of the diagram. The context item, SQL/JSON path expression, and any other inputs (PASSING clause) are specified by the <JSON API common syntax> on the first three lines. The API entry points can merely pass these inputs along to the SQL/JSON path engine for evaluation.

The SQL/JSON path engine evaluates the path expression on the inputs, using a language described later in this paper. The language uses a data model similar to XQuery, having ordered sequences of SQL/JSON items.

The result of evaluating a path expression is a status code, and, if the status is *successful completion,* then some value in the SQL/JSON data model, i.e., an SQL/JSON sequence of SQL/JSON items.

The status and output value return to the JSON query operator. The JSON query operator examines the status and the output to decide the final outcome that is returned to the SQL engine, which can either be an exception condition, or some successful result.

Note that the SQL/JSON data model is completely inaccessible to the user. There is no SQL type for the SQL/JSON data model, and none of the four operators can return a value from the data model.

As stated previously, we are proposing a specific SQL/JSON path language because there is no referenceable language at this time. In the future perhaps the JSON community, or the SQL community, may converge on a different path language. It is our hope that the architecture diagram above can accomodate such evolution with mininal changes, perhaps by adding a LANGUAGE clause to specify the path language.

## 3.4 Error handling

Error handling is an integral part of the design. The architecture diagram has two flows labeled "status". This reflects the fact that errors can occur at both the API level and the path engine level.

Errors can be broadly classified as follows:

1. Input conversion errors, especially malformed input JSON.
2. Errors returned by the path engine. We subdivide these errors as follows:

   a) structural errors, for example, accessing an array element or an object member that does not exist

   b) non-structural, for example, divide by zero

   Errors in the path engine have an orthogonal distinction:

   i) errors within a filter

   b) errors outside a filter
3. Output conversion errors

Input and output conversion errors are usually elementary user errors, correctable by editing the input and/or the API invocations. Path engine errors, on the other hand, are typically more complex. We discuss the facilities for identifying and handling path engine errors later.

## 3.5 JSON null, SQL/JSON null, and SQL nulls

This topic is properly part of the data model; however we consider it here because it is a likely source of confusion.

JSON has a value that is spelled `null`. Unlike SQL nulls, this value stands alone in its own type. It is not a number, it is not a character string, it is not a boolean, it is just `null`

When a JSON text is parsed to move it into the SQL/JSON data model, a JSON `null` is converted to an SQL/JSON null. This is a value lodged within the SQL engine, but it is not an SQL null, because there is no SQL type information associated with it.

Even though a JSON `null` or its internal representation as an SQL/JSON null is not an SQL null, there are circumstances in the API in which they are converted from one to another, because there is no better alternative. We consider these scenarios next.

### 3.5.1 Scalar nulls

For scalar (non-JSON) data, these circumstances include the following:

1. In a PASSING clause, if an SQL null is passed into the SQL/JSON path engine, it becomes an SQL/JSON null (i.e., it loses its SQL type information).

2. In JSON_VALUE, if the result of a path expression is an SQL/JSON null, then JSON_VALUE will return an SQL null. In effect, the data acquires SQL type information at this point. The SQL type is the one declared as the return type of JSON_VALUE.
3. In JSON_TABLE, columns use JSON_VALUE logic, so an SQL./JSON null can result in an SQL null value in an output column.

Note that this implies that transitions such as the following are possible:

| SQL value | PASSING clause | SQL/JSON path engine | JSON_VALUE | SQL value |
|---|---|---|---|---|
| integer null | → | SQL/JSON null | → | character string null |

### 3.5.2 Null string values parsed as JSON

The topic here is a string value whose intended use is to hold JSON. For example, a character string column may be intended to hold JSON, or a character string variable in a PSM procedure might be intended to hold JSON. (we say “intended to hold JSON” because there is no type declaration that a column or variable holds JSON; it is the user’s responsibility to insure the semantics of the column or variable.)

We desitinguish two cases:

1. A null string passed as the context item. The context item is the first syntactic element in JSON_EXISTS, JSON_VALUE, JSON_QUERY and JSON_TABLE. For this case we adopt the common SQL principle of “null in, null out”. We also handle the only operand of the IS JSON predicate with this rule. The specifics are:

| operator | result on null context item |
|---|---|
| IS JSON | *Unknown* |
| JSON_EXISTS | *Unknown* |
| JSON_VALUE | null of the result data type |
| JSON_QUERY | null of the result data type |
| JSON_TABLE | empty table |

2. A null string passed in the PASSING clause modified by FORMAT JSON indicating that the string should be parsed as JSON. In this situation, we convert to an empty SQL/JSON sequence in the data model.

## 3.6 JSON_EXISTS

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> already discussed, plus an optional ON ERROR clause, defaulting to FALSE ON ERROR.

## 3.7 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 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.

## 3.8 JSON_QUERY

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 empty behavior> ON EMPTY ]
     [ <JSON query error behavior> ON ERROR ]
     <right paren>

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

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

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

<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 we do not provide DEFAULT <value expression> options; instead, the user can specify an empty array or empty object as the result in the empty or error cases.

## 3.9 JSON_TABLE

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

### 3.9.1 Unnested case

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 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>
```

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, as in XMLTABLE.

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 patern, 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 we consider later. But first, let's look at an example

### 3.9.2 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.

### 3.9.3 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 proposed syntax for the PLAN clause is

```
<JSON table plan clause> ::=
     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>
```

Notes:

1. The first operand of an INNER or OUTER (parent/child relationship) is necessarily a <JSON table path name>.
2. UNION is associative (no parentheses required for a list of paths to be unioned)
3. CROSS is associative.
4. Otherwise parentheses are required to disambiguate complex expressions. In particular, there is no precedence between UNION or CROSS.

Besides the BNF, we need the following syntactic requirements:

5. The first operand of INNER or OUTER must be an ancestor of all path names in the second operand,
6. All path names must appear exactly once in the plan.

## 3.10 Conformance features

We propose the following conformance features for the SQL/JSON query operators (Tx2n denotes a block of conformance feature numbers):

Tx21 SQL/JSON: Basic query operators

defined as the following:

IS JSON but no <JSON predicate uniqueness constraint>, so it defaults to WITHOUT UNIQUE KEYS

no PASSING clause in JSON_EXISTS, JSON_VALUE, JSON_TABLE, JSON_QUERY

JSON_EXISTS, no PASSING clause, no ON ERROR clause.

JSON_VALUE with no PASSING clause, no ON EMPTY, no ON ERROR clause.

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 (ie, no ON EMPTY, no ON ERROR)

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

Tx22 SQL/JSON: IS JSON WITH UNIQUE KEYS predicate

Tx23 SQL/JSON: PASSING clause

Tx24: JSON_TABLE: PLAN clause

Tx25: SQL/JSON: ON EMPTY and ON ERROR clauses in query functions

Tx26: SQL/JSON: General <value expression> in ON EMPTY or ON ERROR clauses

Tx27: JSON_TABLE: sibling NESTED COLUMNS clause

Tx28: JSON_QUERY

JSON_QUERY but no PASSING, NO EMPTY, ON ERROR or wrapper clauses. These excluded syntax options are enabled in conjunction with other features.

Tx29: JSON_QUERY: array wrapper options

Note that we will present a list of conformance features for the proposed SQL/JSON path language later.

# 4. Data model

This section considers the data model that is used by our proposed SQL/JSON path language. Our data model can be summarized as "sequences of items", which is similar to [XQuery 1.0.2]. Unlike XQuery, our items are SQL scalar values with an additional SQL/JSON null value, and composite data structures using JSON arrays and objects.

It is necessary to clearly distinguish between JSON values "outside" the DBMS and their analogs "inside" the DBMS. We adopt the following convention:

— the modifier "JSON" refers to constructs within a character or binary string that conforms to [RFC 4627].

— the modifier "SQL/JSON" refers to constructs within the data model SQL engine.

In addition, we believe that users will want to parameterize their queries with values from traditional SQL, particularly to parameterize predicates. Thus we need some ability to move values from classical SQL intot the SQL/JSON data model.

The relationship between the three worlds of "JSON", "SQL/JSON" and "SQL" is crudely illustrated in the following diagam:

$$\begin{tikzcd}
\boxed{\text{JSON}} \arrow[r, shift left, "\text{parse}"] & \boxed{\text{SQL/JSON}} \arrow[l, shift left, "\text{serialize}"] \arrow[r, shift right, "\text{RETURNING}"'] & \boxed{\text{SQL}} \arrow[l, shift right, "\text{PASSING}"']
\end{tikzcd}$$

The following table lists some of the parallels between these worlds:

| JSON | SQL/JSON | classical SQL |
|---|---|---|
| JSON array | SQL/JSON array | |
| JSON object | SQL/JSON object | |
| 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 | SQL/JSON | classical SQL |
|---|---|---|
| JSON string | character string | non-null character string |
| | datetime | non-null datetime |
| | SQL/JSON item | |
| | SQL/JSON sequence | |

With reference to "JSON" and "SQL/JSON", all JSON can be parsed into SQL/JSON; however, not all SQL/JSON can be serialized into JSON.

With reference to "SQL/JSON" and "SQL", the two worlds overlap (and have precisely the same values) in the case of non-null scalars of boolean, numeric, string or datetime types.

## 4.1 SQL/JSON items

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

   — plus:

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

2. arrays
3. objects

These categories are discussed in the following subsections

### 4.1.1 Atomic values

Atomic values in the data model are virtually a subset of the values of <predefined type> in [Foundation 7CD1] 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 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 binary 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, we decided to follow JavaScript semantics for null in comparisons, which says that the SQL/JSON null is equal to the SQL/JSON null. (see [ECMA-262 5.1] 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 data model with the same semantics as the corresponding operation in SQL, but note that null semantics follows [ECMA-262 5.1].

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

### 4.1.2 SQL/JSON arrays

An array is an ordered list of zero or more SQL/JSON items in the 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 [ECMA-262 5.1] arrays are 0-relative. We decided it was more important to be consistent with the latter, for example, so that some path expressions could be used in either the SQL/JSON path language or JavaScript. Therefore SQL/JSON arrays are 0-relative.

### 4.1.3 SQL/JSON objects

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

Note: Empty objects are permitted; see [RFC 4627] section 2.2 "Objects" and [ECMA-262 5.1] section 15.12.1.2 "The JSON syntactic grammar".

A member is a pair of values:

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

b) the second value is any SQL/JSON item in the data model and is called the bound value of the member. (See [SQL/JSON part 1] section 1.2.5 “JSON terminology”.)

An object is serialized enclosed in curly braces, with the members listed in a non-determinstic 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 (similar to [XQuery 1.0.2]). People 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 specification will endeavor 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 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 AVRO and BSON are not refereneceable, we leave them as an implementation extensions.

The conversion from JSON is pretty obvious, because our data model is basically a superset of JSON. In particular

| JSON text | data model |
|---|---|
| true | *True* |
| false | *False* |
| null | null |
| string, e.g., "hello dolly" | string, e.g., "hello dolly" |
| number | The format for numbers in [RFC 4627] is the same as numeric literals in SQL. Consequently we can port a number from JSON to 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 data model back to some storage format. We only specify serialization to JSON text; conversion to some other format, such as AVRO or BSON, is left to other specifications.

SQL datetimes cannot be serialized; neither can SQL/JSON sequences of length greater than 1.

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

The precise result of serialization is implementation-dependent because

1. we do not specify meaningless whitespace;
2. Because of escape sequences, there are multiple ways to serialize a character string.
3. [ECMA-262 5.1] 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.
4. Members of an object are unordered, so there are many possible permutations of the members of an object.

# 5. Path language

## 5.1 Objectives for the SQL/JSON path language

Our objectives for the SQL/JSON path language are

1. Minimalism: we looked for a minimal language that will meet a short list of use cases, leaving ourselves more freedom to adapt the language to additional use cases in the future.

   We made the following decisions 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. We hope to facilitate "push-down" optimizations 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 we believe that is the language that our customers will fnd most appropriate for working on JSON. JavaS-cript has been standardized as [ECMA-262 5.1]. 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. We adopted 0-relative arrays (JavaScript-like rather than SQL-like)

   b) lexical and syntactic design generally follow JavaScript

4. Mongo accomodation: Mongo is a platform for JSON which has 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.

   Here are some consequences of this objective:

   a) We provide a "lax" mode which avoids errors on certain path expressions that would be regarded as errors in JavaScript.

## 5.2 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.

In addition, lax mode adopts the Mongo 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:

| | 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 $[*], may be applied to a non-array. To do this, the non-array is implicitly wrapped in an array prior to applying the subscript operation. This also facilitates the use case where a field may be either an array or a scalar. | there is no automatic wrapping prior to subscript path steps. |
| error handling | Many errors related to whether data is or is not an array or scalar are handled by the two preceding features. The remaining errors are classified as either structural or non-structural. An example of a structural error is `$.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 |

The mode is specified by a mandatory key word, leaving implementations free to adopt a default.

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

### 5.2.1 Example of strict vs. lax

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

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

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. Let's say we want 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 we provide lax mode, 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 have been doubly underlined. 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 singly 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 valules for the doubly underlined columns. In the SELECT list, each COALESCE operator is used to choose the non-null values from a singly underlined column and the corresponding doubly underlined column.
2. Rows 2 and 3 do not have bare phone# and phonetype; isntead they have an array called phones. In these rows, the singly 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 doubly-underlined 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 singly 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 doubly underlined 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 singly 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 doubly underlined columns also come up null, and the COALESCE operatlors combine these nulls to get null. The end result is the same as row 4.

## 5.3 Lexical issues

Lexically, the SQL/JSON path language generally follows the conventions of [ECMA-262 5.1] (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.

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

— comments

— hex numeric literals

— JavaScript regular expressions (instead we adopt the SQL predicate LIKE_REGEX, 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)

We also made the following lexical adjustments

— identifiers may not start with $

— @ is an additional punctuator

It turns out that we do not need any reserved words in SQL/JSON path language. The issue is how to determine in a lexical scanner whether an alphabetic string is a key word or an identifier. In the proposed language, identifiers occur in only two contexts in our 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.

We should look particularly at the rules for nested quoted strings. 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")'
```

## 5.4 Syntax summary

The following table summarizes the features of the proposed language:

| ccomponent | example |
|---|---|
| literals | `"hello"`<br>`1.5e3`<br>`true`<br>`false`<br>`null` |

| ccomponent | example |
|---|---|
| variables | `$` — context item<br>`$frodo` — variable whose value is set in PASSING clause<br>`@` — value of the current item in a filter |
| parentheses | ($a + $b)*$c |
| accessors | member accessor: `$.phone`<br>wildcard member accessor: `$.*`<br>element accessor: `$[1, 2, 4 to 7]`<br>wildcard element accessor: `$[*]` |
| filter | `$?( @.salary > 100000 )` |
| boolean | `&&`<br>`\|\|`<br>`!` |
| comparison | `== != < <= > >=` |
| special predicates | `exists ($)`<br>`($a == $b) is unknown`<br>`$ like_regex "colou?r"`<br>`$ starts with $a` |
| arithmetic | `+ - * / %` |
| item functions | `$.type()`<br>`$.size()`<br>`$.double()`<br>`$.ceiling()`<br>`$.floor()`<br>`$.abs()`<br>`$.datetime()`<br>`$.keyvalue()` |

## 5.5 Formal semantics

### 5.5.1 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: $S = ( I_1, I_2, \ldots I_n )$. Individual subscripts on SQL/JSON items are *j, k.*

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

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

## 5.6 Primitive operations

The formal semantics will use the following primitive operations:

### 5.6.1 concatenation

concatenation of SQL/JSON sequences $S_1$ , $S_2$ , . . . , $S_n$ is denoted ( $S_1$ , $S_2$ , . . . , $S_n$ ). SQL/JSON sequences follow XQuery rules: any empty SQL/JSON sequences are removed and there is no nesting of SQL/JSON sequences.

### 5.6.2 unwrap

unwrap: unwrap() expands all the arrays in an SQL/JSON sequence.
Let $S$ = ( $I_1, I_2, \ldots I_n$ ); then unwrap($S$) is defined by these rules:

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

case:

i) If $I_j$ is an array $I_j$ = [ $E_1, \ldots E_m$ ], then let $S2_j$ = ( $E_1, \ldots E_m$ )

ii) Otherwise, let $S2_j$ = ( $I_j$ )

b) 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 accomodated 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:

| T.C |
|---|
| { name: "Babu", phones: { type: "cell", "090-0101" }} |
| { name: "Fred", phones: [ { type: "home", number: "372-0453" },<br>{ 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.

### 5.6.3 wrap

wrap: wrap() converts any nonarray in an SQL/JSON sequence to an array of length 1. Let $S = ( I_1, I_2, \ldots I_n )$; then wrap($S$) is defined by these rules:

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

i) If $I_j$ is an an array, then $I2_j = I_j$

ii) Otherwise, $I2_j = [ I_j ]$

b) The result of wrap ($S$) is the SQL/JSON sequence

$( I2_1 , I2_2 , \ldots , I2_n )$

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 sounds very like 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:

| 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: 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 $S = (I_1, \ldots , I_n)$ is an SQL/JSON sequence that contains no arrays, then

unwrap (wrap ( $(I_1, \ldots , I_n)$ ) ) = unwrap ( $[I_1], \ldots , [I_n]$ ) = $(I_1, \ldots , I_n)$

## 5.7 Mode declaration

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

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

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

The mode declaration is mandatory syntax; an implementation is free to pick a default.

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

## 5.8 <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 Foundation.). The primaries in the proposed language are given by the BNF

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

### 5.8.1 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:

| as written | interpreted as |
|---|---|
| `true` | boolean <u>*True*</u> |
| `false` | boolean <u>*False*</u> |
| `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:

| example | explanation |
|---|---|
| `"O''Connor"` | The single quote character is escaped by doubling (SQL convention). the value is O'Connor |

| example | explanation |
|---|---|
| `"\"hello\""` | The double quote character is escaped with a backslash (JavaScript convention). The value is "hello" |

### 5.8.2 Variables

The BNF for variables is

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

**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 may not 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, it 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:

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

— 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.

### 5.8.3 Parentheses

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

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

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

## 5.9 Accessors

The syntax for accesors 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 our present purposes, there are four accessors:

member accessor

wildcard member accessor

element accessor

wildcard element accessor

Accessors are postfix operators, so it is possible to concatenate them (examples to follow); evaluation is from left to right.

**Overview:** 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

### 5.9.1 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" } ] }
```

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

| | step | value |
|---|---|---|
| 1 | `$` | `{ phones: [`<br>`  { type: "cell", number: "abc-defg" },`<br>`  {               number: "pqr-wxyz" },`<br>`  { type: "home", number: "hij-klmn" } ] }` |
| 2 | `$.phones` | `[ { type: "cell", number: "abc-defg" },`<br>`  {               number: "pqr-wxyz" },`<br>`  { type: "home", number: "hij-klmn" } ]` |
| 3 | `$.phones.type` | `"cell",`<br>`"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::

| | step | value |
|---|---|---|
| 2.1 | *`Unwrap`* `($.phones)` | `{ type: "cell", number: "abc-defg" },`<br>`{               number: "pqr-wxyz" },`<br>`{ 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, we have a structural error, 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:

| | step | value |
|---|---|---|
| 1 | `$` | `{ phones: [`<br>`  { type: "cell", number: "abc-defg" },`<br>`  {               number: "pqr-wxyz" },`<br>`  { type: "home", number: "hij-klmn" } ] }` |
| 2 | `$.phones` | `[ { type: "cell", number: "abc-defg" },`<br>`  {               number: "pqr-wxyz" },`<br>`  { type: "home", number: "hij-klmn" } ]` |
| 3 | `$.phones[*]` | `{ type: "cell", number: "abc-defg" },`<br>`{               number: "pqr-wxyz" },`<br>`{ type: "home", number: "hij-klmn" }` |
| 4 | `$.phones[*].`<br>`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::

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

### 5.9.2 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 SQLan 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:

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

| | 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"` |

### 5.9.3 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 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 fourth element of $. Although subscripts can be specified in any order and may contain duplicates, the results are always returned in document order without 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 unioned, removing duplicates, to obtain the final set of subscripts.

   e) The result for *I* is the SQL/JSON sequence of elements in *I* having 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

| | step | value |
|---|---|---|
| 1 | `$` | `{ sensors:`<br>`  { SF: [10,11,12,13,15,16,17],`<br>`    FC: [20,22,24],`<br>`    SJ: [30,33]`<br>`  } }` |

| | step | value |
|---|---|---|
| 2 | `$.sensors` | `{ SF:[10,11,12,13,15,16,17],`<br>`FC: [20,22,24],`<br>`SJ: [30,33]`<br>`}` |
| 3 | `$.sensors.*` | `[10,11,12,13,15,16,17],`<br>`[20,22,24],`<br>`[30,33]` |
| 4 | `$.sensors.*[0,last,2]` | `10,13,17,`<br>`20,24,`<br>`30,33` |

Note that in step 3, the second array has 3 elements, so that `last` and 2 are redundant subscripts. In step 4, the element whose value is 24 is only selected once in spite of the redundant subscripts.

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 evalulated 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

### 5.9.4 Element wildcard accessor

The BNF is

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

For example, `$[*]`. This 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 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.

Most of the examples of JSON_TABLE have used `[*]` in the row pattern and also in any NESTED COLUMN patterns.

### 5.9.5 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:

| 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 (arrays are 0-based), 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"
```

## 5.10 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 pattern> ]
       <right paren>
     | keyvalue <left paren> <right paren>
```

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 automaticlly unwrap an array in lax mode.

### 5.10.1 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 charcter 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")`

### 5.10.2 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 is the number of members in the object

— the size of 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.

### 5.10.3 Numeric item methods (double, ceiling, floor, abs)

The numeric item methods provide common numeric functions

— doube() 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

### 5.10.4 datetime()

JSON has no datetime types. Users are probably storing datetimes in character strings. Unlike XPath and XQuery, the JSON specification gives no guidance about how to format datetimes in character strings; therefore we can expect that user data exhibits a profusion of formats, including the American preference for month/day/year versus the alterantive preference for day-month-year and the computer-friendly yearmonthday; and the American preference for twelve-hour clock with am/pm vs the 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.

We believe it will be useful to perform predicates close to the data in the path language. Our solution is to augment the SQL/JSON path language with a modest datetime capability. The four ingredients are

— the data model is augmented with the SQL datetime types

— `datetime()` method to convert a character string to an SQL datetime type

— 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 will support the critical use case of comparison predicates. If users demand datetime arithmetic, that can be added later.

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.

There is concurrently comment #252, P02-USA-900, on the subject of converting character strings to datetimes:

> P02-USA-900
>
> No specific location
>
> Currently, using the <cast specification> is the only way to convert a TIMESTAMP value into a string representation and vice versa. However, the <cast specification> is limited in that it accepts exactly one input format and produces one output format. Thus, there is a need for additional expressions/functions in the standard that allow the user to specify the format of the result (if transforming a timestamp to string) or the format of the input (if transforming a string to a timestamp). For example, all of the following could be valid string representations of the same timestamp (but just the first one is recognized by the SQL standard):
>
> - '2009-03-13 23:05:00'
>
> - '03-13-2009 11.05.00 PM'
>
> - '13.03.2009-23.05.00'
>
> A function that would convert any of the three alternatives above to a SQL TIMESTAMP value would take two input values: the string representing the timestamp and another string representing the format of the first input string. As an example,
>
> ```
> TIMESTAMP_FORMAT('13.3.2009-23.05.00',
>                  'DD.MM.YYYYHH24.MI.SS')
> ```
>
> will return a timestamp value equivalent to TIMESTAMP '2009-03-13 23:05:00'
>
> Conversely,
>
> ```
> VARCHAR_FORMAT(TIMESTAMP '2009-03-13 23:05:00',
>                'MM/DD/YYYY-HH12.MI.SS A.M.')
> ```
>
> will return a string of the following form:
>
> ```
> '03/13/2009-11.05.00 P.M.'
> ```

We hope to present a solution to this comment in a later paper. Once we have SQL-standard syntax for converting character strings to datetimes, it will be feasible to reference that functionality in SQL/JSON. In the meantime, we propose the minimal capability of converting character strings that are formatted so that CAST to character string succeeds. This means that the character string must be formatted as an <unquoted date string>, <unquoted time string> or <unquoted timestamp string>. These are rather rigid formatting conventions, which we hope to overcome with a general solution to comment #252.

### 5.10.5 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 },
     { name: "who",  value: "Moe",  id: 5372 },
     { name: "how",  value: 22,     id: 5372 }
   )
```

This example illustrates the use of the id member in the result to distinguish separate objects in the input. The data can be viewed using JSON_TABLE with the following query:

```
SELECT ID, NAME, SVALUE, IVALUE
FROM T,
  JSON_TABLE (T.J, 'lax $.keyvalue()'
    COLUMNS (
      NAME VARCHAR(30) PATH 'lax $.name',
      SVALUE VARCHAR(30) PATH
        'lax $.value ? (@.type() == "string")',
      IVALUE INTEGER VARCHAR(30) PATH
        'lax $.value ? (@.type() == "number")',
      ID INTEGER PATH 'lax $.id'
    ) ) AS JT
ORDER BY ID, NAME
```

Note how the patterns for SVALUE and IVALUE filter the `value` member based on its `type()` as either `string` or `number`.

The result of the query with the sample data is

| ID | NAME | SVALUE | IVALUE |
|---|---|---|---|
| 5372 | how | | 22 |
| 5372 | who | Moe | |
| 8394 | what | | 64 |
| 8394 | who | Fred | |

## 5.11 Arithmetic expressions

The SQL/JSON path language uses the same arithmetic operators as [ECMA-262 5.1]:

— unary + or −

— binary + `-` `*` `/` `%`
for addition, subtraction multiplcation, division, and modulus, respectively.

### 5.11.1 Unary plus and minus

The unary plus and minus operations iterate over the SQL/JSON sequence that is their operand. Every SQL/JSON item in the SQL/JSON sequence must be numeric (else an error is raised, even in lax mode). Otherwise, the only error is the corner case of overflow when taking the unary minus of certain numbers at the boundary of their range.

The unary operations are prefix operations whereas the accessors are postfix operations. The precedence binds the accessors more tightly (the same precedence as [ECMA-262 5.1]). For example, suppose

```
$ = { readings: [15.2, -22.3, 45.9] }
```

Then `lax -$.readings.floor()` is equivalent to
`lax -($.readings.floor())`:

| step | expression | value |
|---|---|---|
| 1 | `$` | `{ readings: [15.2, -22.3, 45.9] }` |
| 2 | `$.readings` | `[15.2, -22.3, 45.9 ]` |
| 3 | `$.readings.floor()` | `15, -23, 45` |
| 4 | `-$.readings.floor()` | `-15, 23, -45` |

To get a different order of evaluation, parentheses are required, as in
`lax (-$.readings).floor()`:

| step | expression | value |
|---|---|---|
| 1 | `$` | `{ readings: [15.2, -22.3, 45.9] }` |
| 2 | `$.readings` | `[15.2, -22.3, 45.9 ]` |
| 3 | `-$.readings` | `-15.2, 22.3, -45.9` |
| 4 | `(-$.readings)` | `-15.2, 22.3, -45.9` |
| 5 | `(-$.readings).floor()` | `-15, 22, -45` |

Notice that the two expressions are identical for positive inputs but not identical for negative inputs.

In strict mode, these examples require an explicit `[*]` to unnest the array, either
`strict -$.readings[*].floor()`, or
`strict (-$.readings[*]).floor()`

### 5.11.2 Binary operations

The binary operations do not iterate over an SQL/JSON sequence (such iteration would require a cross product). Instead, they expect their operand to be a singleton numeric, otherwise the result is an error, even in lax mode.

The binary operators have the same precedence as in most computer languges, including [ECMA-262 5.1] and SQL. As usual, parentheses may be used to override the precedence.

Modulus, indicated by % as in [ECMA-262 5.1], uses the same algorithm as the SQL `mod` function.

## 5.12 Filter expression

A filter expression is similar to a WHERE clause in SQL — it is used to remove SQL/JSON items from an SQL/JSON sequence if they do not satisfy a predicate. The syntax uses a question mark followed by a parenthesized predicate:

```
<JSON filter expression> ::=
     <question mark> <left paren> <JSON path predicate>
     <right paren>
```

To evaluate a filter, the following steps are performed:

1. In lax mode, any SQL/JSON arrays in the operand are unwrapped.
2. The predicate is evaluated for each SQL/JSON item in the SQL/JSON sequence.
3. The result is those SQL/JSON items for which the predicate resulted in *True*.

Within a filter, the special variable `@` is used to reference the current SQL/JSON item in the SQL/JSON sequence. There have been several examples of this variable earlier in this discussion. A Syntax Rule stipulates that `@` is only permitted in the predicate of a filter. The value of `@` is the current SQL/JSON item of the first operand of the innermost filter containing the `@`.

The SQL/JSON path language has the following predicates:

— `exists` predicate, to test if a path expression has a non-empty result

— comparison predicates == `!= >= > < <=`

— `like_regex` for string pattern matching

— `starts with` to test for an initial substring

— `is unknown` to test for *Unknown* results.

The result of evaluating a predicate is an SQL truth value (*True, False, Unknown*) rather than an SQL/JSON sequence in the data model.

Like [SQL-92] and unlike [ECMA-262 5.1], predicates are not expressions; instead, predicates constitute a sublanguage that is only permitted within the filter expression.

The result of a predicate is an SQL truth value (*True, False, Unknown*).

### 5.12.1 true/false and ***True/False***

JSON has literals `true` and `false` , which are parsed into the SQL/JSON model as the SQL boolean values *True* and *False*. However, there is no syntax to treat an SQL/JSON item as a boolean value. For example, suppose

```
$ = { name: "Portia", skilled: true }
```

then `$.skilled` is *True*, but there is no syntax such as

```
$ ? ( @.skilled )
```

which would presumably treat `$.skilled` as a predicate. We decided against such syntax because then the truth value of an arbitrary SQL/JSON sequence must be defined, which leads to the complexity seen in XQuery's notion of effective boolean value ([XQuery 1.0.2] section 2.4.3 "Effective boolean value").

Instead, to test whether `$.skilled` is *True*, a comparison predicate such as

```
$ ? (@.skilled == true)
```

should be used.

### 5.12.2 null and *Unknown*

JSON has literal `null`, which is parsed into the SQL/JSON data model as a special value called the SQL/JSON null. The SQL/JSON null is not the same as the SQLnull. Here are some differences to be aware of:

1. The SQL truth value *Unknown* is the same as the null value of boolean type. In the SQL/JSON data model, there are no SQL nulls so *Unknown* is not part of the SQL/JSON data model. However, SQL/JSON predicates have results that are SQL truth values *True, False* and *Unknown* Recall that the SQL/JSON path language distinguishes predicates from expressions; predicates result in truth values whereas expressions result in values in the data model.
2. An SQL predicate operating on an SQL null usually returns *Unknown*. In contrast, the SQL/JSON null value is equal to itself; the result of `null == null` is *True*.

### 5.12.3 Error handling in filters

Errors can arise in filters in two ways:

1. A predicates must evaluate its operand(s) which are expression(s), which may result in an error condition. (Lax mode converts structural errors to an empty SQL/JSON sequence. All non-structural errors in lax mode, as well as all errors of any sort in strict mode, are regarded as unhandled errors at the expression level.)
2. After evaluating its operand(s), the predicate may find that the values are not appropriate. For example, `10 == "ten"` there are no errors in the operands, but they are not comparable, so this predicate still has an error.

With either kind of error, the predicate results in *Unknown*.

Here are some examples. First let us consider a nonstructural error such as divide by a non-numeric value. Consider a table T with two rows:

| K | J |
|---|---|
| 101 | `{ pay: 100, hours: 10 }` |
| 102 | `{ pay: 100, hours: "ten" }` |

The user wishes to find rows in which the average of pay/hours is greater than 9. The user writes this query:

```
SELECT K
FROM T
WHERE JSON_EXISTS (T.J, 'lax $ ? (@.pay/@.hours > 9)'
```

Let us consider how this query is evaluated on each row of T.

In row K=101, the computation proceeds without incident. In the filter, `@.pay` = 100 and `@.hours` = 10, so the quotient `@.pay/@.hours` is 10, which is greater than 9. Therefore the filter succeeds and the overall path expression results in a nonempty SQL/JSON sequence, the JSON_EXISTS is *True*, and 101 appears in the result of the query.

As for row K=102, we have the following computation:

| step | expression | value |
|---|---|---|
| 1 | `$` | `{ pay: 100, hours: "ten" }` |
| 2 | `@` | `{ pay: 100, hours: "ten" }` |
| 3 | `@.pay` | `100` |
| 4 | `@.hours` | `"ten"` |
| 5 | `@.pay/@.hours` | *error* |
| 6 | `@.pay/@.hours > 9` | *Unknown* |
| 7 | `$ ? (@.pay/@.hours > 9)` | *empty SQL/JSON sequence* |

Since the result of the path expression is the empty SQL/JSON sequence, JSON_EXISTS is *False*, and 102 does not appear in the result of the query.

This example would behave precisely the same in strict mode, because nonstructural errors are unhandled errors in both modes.

Next let us consider a structural error. We modify the table slightly, like this:

| K | J |
|---|---|
| 101 | `{ pay: 100, hours: 10 }` |
| 102 | `{ pay: 100, horas: 10 }` |

The difference is that row K=102 does not have an `hours` member. With the same query as before (still in lax mode), row K=101 will process just the same and appear in the final result. As for row K=102, we have the following computation:

| step | expression | value |
|---|---|---|
| 1 | `$` | `{ pay: 100, horas: 10 }` |
| 2 | `@` | `{ pay: 100, horas: 10 }` |
| 3 | `@.pay` | `100` |
| 4 | `@.hours` | *empty sequence* |
| 5 | `@.pay/@.hours` | *error* |
| 6 | `@.pay/@.hours > 9` | <u>*Unknown*</u> |
| 7 | `$ ? (@.pay/@.hours > 9)` | *empty sequence* |

With the revised data, step 4 is now an empty sequence, but step 5 is still an unhandled error, and thereafter the computation is the same, with the same ultimate result, K=102 is omitted from the final result. Strict mode would have the identical value at each step of the computation.

As another example, with the same data, consider the query `'lax $ ? (@.hours > 9)'`. The computation for row K=101 is uninteresting; let's look at K=102:

| step | expression | value |
|---|---|---|
| 1 | `$` | `{ pay: 100, horas: 10 }` |
| 2 | `@` | `{ pay: 100, horas: 10 }` |
| 3 | `@.hours` | *empty SQL/JSON sequence* |
| 4 | `@.hours > 9` | <u>*False*</u> |
| 5 | `'lax $ ? (@.hours > 9)'` | *empty SQL/JSON sequence* |

In this example, step 4, the comparison of an empty SQL/JSON sequence with 9, is not an error. Instead, as we will discuss later, comparison predicates are performed with existential semantics. Essentially, the comparison predicate forms the cross product of all SQL/JSON items in the first operand with all SQL/JSON items in the second operand. The comparison predicate is ***True*** if any of these comparisons in the cross product is true. In this example, one of the operands is empty, so the cross product is empty, and therefore the predicate is ***False***.

Now let's consider the last example in strict mode, `'lax $ ? (@.hours > 9)'` :

| step | expression | value |
|---|---|---|
| 1 | `$` | `{ pay: 100, horas: 10 }` |
| 2 | `@` | `{ pay: 100, horas: 10 }` |
| 3 | `@.hours` | *error* |
| 4 | `@.hours > 9` | ***Unknown*** |
| 5 | `'lax $ ? (@.hours > 9)'` | *empty SQL/JSON sequence* |

Step 3 is a structural error, which is an unhandled error in strict mode. The unhandled error comes into the predicate, where it causes a result of *Unknown*. The final result in step 5 is the same as in lax mode, because a filter rejects both ***False*** and ***Unknown***.

### 5.12.4 Truth tables

By design, the boolean operators `&&` `||` `!` always have singleton operands which are truth values (***True***, ***False***, ***Unknown***). There is no way to treat a JSON atomic value `true` or `false` as an operand of a boolean operator. The user should write, e.g., $a == true to test if $a is the JSON literal true.

The operands of && (boolean AND) may be evaluated in either order. If the first tested operand's value is enough to determine the result there is no need to evaluate the other operand. Otherwise the result is given by this table

| | True | False | Unknown |
|---|---|---|---|
| True | True | False | Unknown |
| False | False | False | False |
| Unkinown | Unknown | False | Unknown |

Similarly, the operands of `||` may be tested in either order, with short circuit logic if the first to be tested is enough to determine the result. The result is given by

| | True | False | Unknown |
|---|---|---|---|
| True | True | True | True |
| False | True | False | Unknown |

| | True | False | Unknown |
|---|---|---|---|
| Unkinown | True | Unknown | Unknown |

The truth table for ! (boolean NOT) is

| P | NOT P |
|---|---|
| True | False |
| False | True |
| Unknown | Unknown |

### 5.12.5 Comparison predicates

Following [ECMA-262 5.1], the comparison operators are denoted == != < <= > >=.

```
<JSON comparison predicate> ::=
     <JSON path wff> <JSON comp op> <JSON path wff>

<JSON comp op> ::=
       <double equals>
     | <not equals operator>
     | <less than operator>
     | <greater than operator>
     | <less than or equals operator>
     | <greater than or equals operator>
```

Although we adopt the symbols from [ECMA-262 5.1], the syntax and semantics are quite different from [ECMA-262 5.1]. Here are some of the key differences:

— Comparison operators are not left associative, unlike [ECMAScript].

— Equality operators have the same precedence as inequality comparision operators, unlike [ECMAScript].

— We do not provide automatic casting rules to support comparisons across types. (For example, in [ECMA-262 5.1], booleans can be compared to numbers, with `true == 1` and `false == 0`.)

— We do not support comparison of arrays or objects to anything, even themselves. There is no “deep equals”.

— In lax mode, comparison operators automatically unwrap their operands.

The following table summarizes which comparisons are supported:

| | SQL/JSON null | SQL/JSON scalar | SQL/JSON array | SQL/JSON object |
|---|---|---|---|---|
| SQL/JSON null | comparable | comparable | NOT | NOT |
| SQL/JSON scalar | comparable | string vs. string<br>number vs. number<br>date vs. date<br>time vs. time<br>timestamp vs timestamp | NOT | NOT |
| SQL/JSON array | NOT | NOT | NOT | NOT |
| SQL/JSON object | NOT | NOT | NOT | NOT |

Thus comparisons to SQL/JSON arrays and objects are not supported. Note though that in lax mode, arrays are unwrapped prior to comparison, which might mitigate this

SQL/JSON nulls and scalars may be compared. When comparing two scalars, they must be comparable as SQL scalars. Comparison is decided using SQL semantics, with this additional rule: SQL/JSON null is equal to SQL/JSON null, and is not greater than or less than anything.

Comparison predicates have existential semantics, similar to XQuery. This means that the two operands may be SQL/JSON sequences. The cross product of these SQL/JSON sequences is formed. Each SQL/JSON item in one SQL/JSON sequence is compared to each ittem in the other SQL/JSON sequence. The predicate is _Unknown_ if there any pair of SQL/JSON items in the cross product is not comparable; the predicate is _True_ if any pair is comparable and satisfies the comparison operator. In lax mode, the path engine is permitted to stop evaluation early if it detects either an error or a success (whichever one is found first is the "winner").

The semantics of comparison predicate can be summarized with these rules:

1. Evaluate both operands, giving two SQL/JSON sequences *A* and *B*. If there is an error in either evaluation, the result is _Unknown_ and no further rules are performed.
2. In lax mode, unwrap each SQL/JSON *A* and *B*.
3. Let *ERR* and *FOUND* be flags that are initially _False_.
4. Form the cross product of *A* and *B*. Compare the pairs in this cross product in an implementation-dependent order. If any pair is not comparable, set *ERR* to _True_. If any pair satisfies the comparison, set *FOUND* to _True_.

5. The final result is determined by this table:

| | *ERR* = *True* | *ERR* = *False* |
|---|---|---|
| *FOUND* = *True* | strict mode: *Unknown*<br>lax mode: either *True* or *Unknown*<br>(implementation-dependent) | *True* |
| *FOUND* = *False* | *Unknown* | *False* |

Let's look at some examples. Suppose

To evaluate a comparison predicate

Note: there are paradoxical consequences, for example [1,2] == [1,2] will result in *Unknown*, and so will [1,2] != [1,2].

### 5.12.6 like_regex predicate

For pattern matching, we borrowed from SQL the LIKE_REGEX predicate rather than the LIKE predicate, because LIKE_REGEX provides a rich regular expression capability. To facilitate pre-compilation, we only support character string literals in the pattern and flags. The syntax is

```
<JSON like_regex predicate> ::=
     <JSON path wff> like_regex
     <JSON like_regex pattern>
     [ flag <JSON like_regex flags> ]

<JSON like_regex pattern> ::= <JSON path string literal>

<JSON like_regex flag> ::= <JSON path string literal>
```

Like comparison predicates, the like_regex predicate uses existential semantics.

### 5.12.7 starts with predicate

The `starts with` predicate tests whether its second operand is an initial substring of its first operand. While this could be done using like_regex, the benefit to the user is that there is no need to check the second operand for the presence of special characters such as quanitifiers which must be escaped when using regular expressions.

The `starts with` predicate can be seen as a kind of range comparison. For example,

```
@.name starts with "Mc"
```

is equivalent to

```
@.name >= "Mc" && @.name < "Md"
```

With this analogy in mind, we permit the second operand to be an SQL/JSON sequence and support existential semantics.

```
<JSON starts with predicate> ::=
     <JSON starts with whole> starts with
     <JSON starts with initial>

<JSON starts with whole> ::= <JSON path wff>

<JSON starts with initial> ::= <JSON path wff>
```

### 5.12.8 Exists predicate

The `exists` predicate tests whether a path expression has at least one SQL/JSON item. The BNF is:

```
<JSON exists predicate> ::=
     exists <left paren> <JSON path wff> <right paren>
```

The rules for this predicate are simple:

1. The path expression <JSON path wff> is evaluated.
2. Case:
    a) If the result of the path expression is an error, then <JSON exists predicate> is *Unknown*.
    b) If the result of the path expression is an empty SQL/JSON sequence, then <JSON exists predicate> is *False*
    c) Otherwise, <JSON exists predicate> is *True*.

The `exists` predicate can be used to probe for a member or element in advance of accessing it. This is especially useful in strict mode, which would otherwise raise an error.

For example, consider a table with JSON column in which some rows have a name member and other rows do, as in this sample data:

| K | J |
|---|---|
| 201 | `{ name: { first: "Manny", last: "Moe" }, points: 123 }` |
| 202 | `{ points: 41 }` |

In strict mode, an expression such as $.name.first would raise an error on row K=202. To avoid this, one could write `'strict $ ? (exists (@.name)) . name'`. The evaluation of this on row K=201 is as follows:

| step | expression | value |
|---|---|---|
| | `$` | `{ name: { first: "Manny", last: "Moe" }, points: 123 }` |
| | `@` | `{ name: { first: "Manny", last: "Moe" }, points: 123 }` |
| | `@.name` | `{ first: "Manny", last: "Moe" }` |
| | `exists (@.name)` | <u>*True*</u> |
| | `$?(exists (@.name))` | `{ name: { first: "Manny", last: "Moe" }, points: 123 }` |
| | `$?(exists (@.name)).name` | `{ first: "Manny", last: "Moe" }` |

The evaluation on row K=202 is as follows:

| step | expression | value |
|---|---|---|
| | `$` | `{ points: 41 }` |
| | `@` | `{ points: 41 }` |
| | `@.name` | *`error`* |
| | `exists (@.name)` | <u>*Unknown*</u> |
| | `$?(exists (@.name))` | *empty SQL/JSON sequence* |
| | `$?(exists (@.name)).name` | *empty SQL/JSON sequence* |

Thus the exists predicate enables the user in strict mode to achieve lax semantics (conversion of structural error to empty SQL/JSON sequence) on a selective basis.

### 5.12.9 Unknown predicate

The `is unknown` predicate tests if a boolean condition is <u>*Unknown*</u>. This is provided because without it, it would be very difficult to find data that causes an <u>*Unknown*</u> predicate result. The syntax is:

```
<JSON unknown predicate> ::=
     <right paren> <JSON path predicte> <left paren>
```

```
        is unknown
```

The boolean condition to be tested is enclosed in parentheses for clarity. The `is unknown` predicate has no error conditions in its own right; instead, it is used to test for error conditions in the parenthesized path predicate (since that is the only way to arrive at an *Unknown* re

For example, a data set may have a member called `sex` which should have values "M" or "F". To find the rows of males, one might write

```
SELECT *
FROM T
WHERE JSON_EXISTS (T.J, 'lax $ ? (@.sex == "M")')
```

and similarly to find the rows of females, one might use `"F"` in the filter.

Perhaps though the data also contains a few anomalous rows in which sex has been encoded numerically as 0 or 1. In that case, predicates such as @.sex == "M" will generate errors, which result in ***Unknown***. By simply counting rows, the user may find they have 50 rows with sex = "M", 50 rows with sex = "F", and two other rows.

Finding these other rows in three-valued logic can be tricky. In this example, you could find them with

```
SELECT *
FROM T
WHERE JSON_EXISTS (T.J,
  'lax $ ? (@.sex.type() == "number")')
```

We are blessed with the insight to know that the problem is that the rows contain numeric `sex`, but there could be other reasons that a `sex` is neither "M" nor "F". To conveniently find the rows that are neither ***True*** nor ***False*** under some predicate, what is required is the `is unknown` predicate. Using this predicate, the user writes

```
SELECT *
FROM T
WHERE JSON_EXISTS (T.J,
  'lax $ ? ( ((@.sex=="M") || (@.sex=="F")) is unknown')
```

The predicate nows selects precisely the troublesome rows that are neither "M" nor "F" for whatever reason.

## 5.13 Conformance features

We propose the following features for the SQL/JSON path language (Tx3n denotes a block of conformance feature numbers):

Tx31 SQL/JSON path language: strict mode

note: there is no feature for lax mode; this means that lax mode support is mandatory for all the SQL/JSON query functions.

Tx32 SQL/JSON path language: item methods

Tx33 SQL/JSON path language: multiple subscripts

Tx34 SQL/JSON path language: wildcard member accessor

Tx35 SQL/JSON path language: filter expressions

Tx36 SQL/JSON path language: starts with predicate

Tx37 SQL/JSON path language: regex_like predicate

# 6. Proposal for [Foundation 7IWD4]

## 6.1 Changes to 2.2, Other international standards

*as if we have the authority to make changes to other international standards*

1. ADD THE FOLLOWING REFERENCE (EDITOR PLEASE MAKE THIS CORRECT AND HARMONIZE WITH [SQL/JSON PART 1]):

**[RFC4627] Internet Engineering Task Force, *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC 4627 http://tools.ietf.org/html/rfc4627**

**[ECMAScript] European Computer Manufacturers Association, "ECMAScript Language Specification 3rd Edition", December 1999, http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf**

## 6.2 New Subclause 4.x.2 "Implied JSON data model"

*[NOTE to the proposal reader: this section is similar to material in [SQL/JSON part 1]; the version in that paper is to be preferred; this preliminary draft is repeated here because it is integral to this proposal as well.]*

1. ADD THE FOLLOWING SUBCLAUSE; CORRESPONDING MATERIAL IN [SQL/JSON PART 1] IS DEFINITIVE FOR THIS SUBCLAUSE:

**4.x.2 Implied JSON data model**

**A *JSON text* is a character string or binary string that conforms to the definition of "JSON-text" in [RFC4627]. A *JSON text fragment* is a substring of a JSON text that conforms to any BNF non-terminal in [RFC4627]. A *JSON literal* is a JSON text fragment that is any of the key words `false`, `null`, or `true`. A *JSON member* is a JSON text fragment that conforms to the definition of member in [RFC4627] section 2.2 "Objects". If *M* is a JSON member, then *M* matches the BNF** `member = string name-separator value`**; the *key* of *M* is the JSON fragment matching**

`string` **in this production, and the *bound value* of *M* is the JSON fragment matching** `value` **in this production. A *JSON object* is a JSON text fragment that conforms to the definition of object in [RFC4627] section 2.2 “Objects”. A *JSON array* is a JSON text fragment that conforms to the definition of array in [RFC4627] section 2.3 “Arrays”. A *JSON number* is a JSON text fragment that conforms to the definition of number in [RFC4627] section 2.4 “Numbers”. A *JSON string* is a JSON text fragment that conforms to the definition of string in [RFC4627] section 2.5 “Strings”. The *value* of a *JSON string* is the Unicode character string enclosed in the delimiting <double quote>s of a JSON string, after replacing all escapes with their equivalent Unicode values.**

**A *JSON value* is a JSON object, JSON array, JSON number, JSON string, or one of three JSON literals.**

## 6.3 New Subclause 4.x.3, SQL/JSON data model

1. ADD THE FOLLOWING SUBCLAUSE (THE CORRESPONDING TEXT IN [SQL/JSON PART 1] IS DEFINITIVE FOR THIS SUBCLAUSE):

### 4.x.3 SQL/JSON data model

**The SQL/JSON data model comprises SQL/JSON items and SQL/JSON sequences. The components of the SQL/JSON data model are:**

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

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

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

**NOTE nnn: An SQL/JSON null is distinct from the null value of any SQL type.**

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

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

**NOTE nnn: [RFC 4627] section 2.2 “Objects” says “The names within an object SHOULD be unique”. Thus non-unique keys are permitted but not 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:**

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

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

**— If *SJI1* and *SJI2* are SQL/JSON arrays, then *SJI1* and *SJI2* are equivalent if they are of the same length *N*, and corresponding elements of *SJI1* and *SJI2* are equivalent.**

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

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

**NOTE nnn: there is no SQL <data type> whose value space is SQL/JSON items, or SQL/JSON sequences.**

## 6.4 New Subclause 4.x.5 “Overview of SQL/JSON path language”

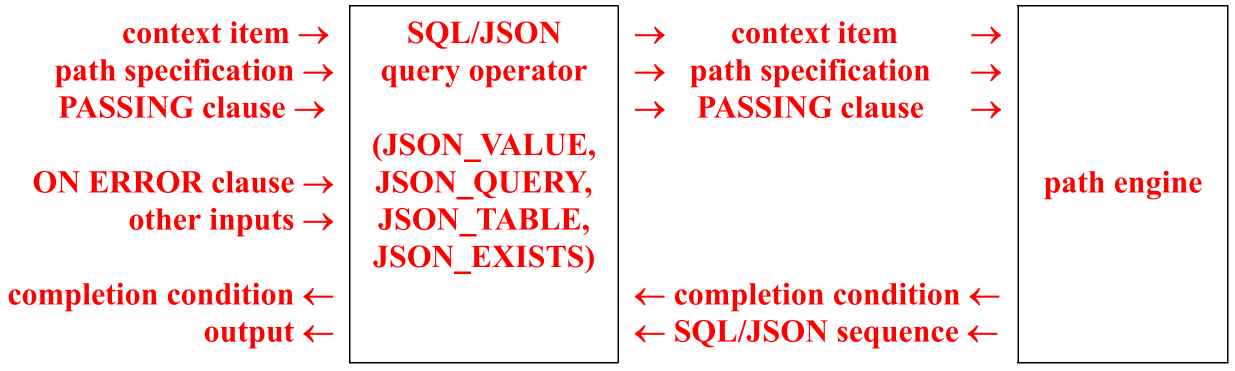
1. APPEND THE FOLLOWING SUBCLAUSE AFTER SUBCLAUSE 4.X.4 “SQL/JSON FUNCTIONS” PROPOSED IN [SQL/JSON PART 1].

### 4.x.5 Overview of SQL/JSON path language

**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 opeators*) to query JSON text. SQL./JSON path language is not, strictly speaking, SQL, though it is embedded in these operators within SQL. Lexically and syntactically, SQL/JSON path language adopts many features of [ECMAScript], though it is neither a subset nor a superset of [ECMAScript]. The semantics of 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:**

2. NOTE TO THE EDITOR: A HANDY WAY TO CREATE THE FOLLOWING DIAGRAM IS TO MAKE A TABLE WITH ONE ROW AND FOUR COLUMNS, AND SUPPRESS THE BORDERS ON TWO OF THE COLUMNS



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

**— a context item (the JSON text to be queried)**

**— a path specification (the query to perform on the context item; this query is expressed in the SQL/JSON path language specified in Subclause N.n “SQL/JSON path language: lexical elements” and in the Format and Syntax Rules of Subclause N.n “SQL/JSON path language: syntax and semantics”)**

**— a PASSING clause (SQL values to be assigned to variables in the path specification, for example, as values used in predicates within the path specification)**

**The SQL/JSON operators effectively pass these inputs to a “path engine” which evaluates the path specification, using the context item and the PASSING clause to specify the value of variables in the path specification. The effective behavior of the path engine is specified in the General Rules of Subclause N.n “SQL/JSON path language: syntax and semantics”.**

**The result of evaluating a path specification on a context item and PASSING clause is a completion condition, and, if the completion condition is *successful completion*, 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.**

## 6.5 Changes to 5.2, <token> and <separator>

1. ADD THE FOLLOWING <NON-RESERVED WORD>S:

```
CHAINING | COLUMNS | CONDITIONAL | ENCODING | ERROR |
FORMAT | JSON | KEYS | NESTED | PLAN |
UNCONDITIONAL | UTF8 | UTF16 | UTF32 | WRAPPER
```

*[NOTE to the proposal reader: already have ON, OBJECT, ORDINALITY, PATH*

2. MOVE THE FOLLOWING <NON-RESERVED WORD> FROM SQL/XML TO FOUNDATION:

```
PASSING | RETURNING
```

3. DELETE THE FOLLOWING <NON-RESERVED WORD>S FROM SQL/XML

```
EMPTY
```

*[NOTE to the proposal reader: this is already a <resreved word> in Foundation.]*

4. ADD THE FOLLOWING <RESERVED WORD>S:

```
JSON_EXISTS | JSON_QUERY | JSON_TABLE |
JSON_TABLE_PRIMITIVE | JSON_VALUE
```

*[NOTE to the proposal reader: already have ARRAY, CROSS, DEFAULT, EMPTY, FALSE, FOR, INNER TUBE, NULL, OUTER, TRUE, UNION, UNIQUE,UNKNOWN, WITH, WITHOUT,*

## 6.6 Changes to 6.3 <value expression primary>

1. APPEND TO THE LIST OF <NONPARENTHESIZED VALUE EXPRESSION>S:

```
14 <nonparenthesized value expression primary> ::=
       <unsigned value specification>
     | <column reference>
     | <set function specification>
     | <window function>
     | <nested window function>
     | <scalar subquery>
     | <case expression>
     | <cast specification>
     | <field reference>
     | <subtype treatment>
     | <method invocation>
     | <static method invocation>
     | <new specification>
     | <attribute or method reference>
     | <reference resolution>
     | <collection value constructor>
     | <array element reference>
     | <multiset element reference>
     | <next value expression>
     | <routine invocation>
     | <row pattern navigation operation>
     | <JSON value function>
```

2. EDIT SYNTAX RULE 1) AS FOLLOWS:

   1) 14 The declared type of a <value expression primary> is the declared type of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <nested window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, or <next value expression>, <row pattern navigation operation>, **<JSON value function>,** or the effective returns type of the simply contained <routine invocation>, respectively.

3. EDIT GENERAL RULE 1) AS FOLLOWS:

   1) 14 The value of a <value expression primary> is the value of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <nested window function>, <scalar subquery>, <case expression>, <cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, <next value expression>, <row pattern navigation operation>, **<JSON value function>,** or <routine invocation>.

## 6.7 New subclause 6.n <JSON value function>

1. ADD THIS SUBCLAUSE (AFTER 6.26 <ROW PATTERN NAVIGATION OPERATION> LOOKS APPROPRIATE):

**6.n <JSON value function>**

**Function**

**Extract an SQL value of a predefined type from a JSON value.**

**Format**

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

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

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

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

## Syntax Rules

1) If <JSON returning clause> is not specified, then an implementation-defined character string type is implicit.

1.1) The <data type> *DT* contained in the explicit or implicit <JSON returning clause> shall be a <predefined type> that identifies a character string data type, numeric data type, boolean data type or datetime data type.

2) The declared type of <JSON value function> is the type specified by *DT*.

3) If <JSON value empty behavior> is not specified, then NULL ON EMPTY is implicit.

4) If <JSON value error behavior> is not specified, then NULL ON ERROR is implicit.

## Access Rules

None

## General Rules

0) 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 *DT* and no further General Rules are performed.

**1) The General Rules of Subclause 10.n "<SQL/JSON common syntax>" are applied. Let *SEQ* be the *SQL/JSON SEQUENCE* and let *ST1* be the *STATUS* that are returned by the Subclause.**

**2) Let *ZB* be the explicit or implicit <JSON value empty behavior>, and let *EB* be the explicit or implicit <JSON value error behavior> contained in the <JSON value function>.**

**3) The General Rules of Subclause N.n "Casting an SQL/JSON sequence to an SQL type" are applied, with *ST1* as the *STAUTS IN, SEQ* as the *SQL/JSON SEQUENCE, ZB* as the *EMPTY BEHAVIOR, EB* as the *ERROR BEHAVIOR, DT* as the *DATA TYPE.* Let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* that are returned by that Subclause.**

**4) If *ST2* is an exception condition, then the exception condition *ST2* is raised. Otherwise, *V* is the result of the <JSON value function>.**

**Conformance Rules**

2. EDITOR SHALL CHOOSE THREE AVAILABLE BLOCKS OF CONFORMANCE FEATURES, DENOTED HERE AS TX1N FOR THE JSON CONSTRUCTION FUNCTIONS IN [SQL/JSON PART 1], TX2N FOR THE SQL/JSON QUERYING OPERATORS, AND TX3N FOR THE SQL/JSON PATH LANGUAGE

**1) Without Feature Tx21, "Basic SQL/JSON", conforming SQL language shall not contain <JSON value function>.**

**2) Without Feature Tx25, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON value function> shall not contain <JSON value empty behavior>.**

**3) Without Feature Tx25, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON value function> shall not contain <JSON value empty behavior>.**

**4) Without Feature Tx26, "General <value expression> in ON ERROR or ON EMPTY clauses", the <value expression> contained in <JSON value empty behavior> or <JSON value error behavior> shall be a <literal> that can be cast to the data type specified by the explicit or implicit <JSON returning clause> without raising an exception condition according to the General Rules of Subclause 6.13 <cast specification>.**

## 6.8 Changes to 6.27 <value expression>

1. EDIT SYNTAXRULE 7) AS FOLLOWS:

7) A <value expression> or <nonparenthesized value expression primary> is *possibly non-deterministic* if it generally contains any of the following:

a) ...

**x) <JSON value function>, <JSON query>, <JSON table> or <JSON exists predicate>**

## 6.9 Changes to 6.31 <string value function>

1. EDIT THE BNF FOR <STRING VALUE FUNCTION> AS FOLLOWS:

```
09 <string value function> ::=
       <character value function>
     | <binary value function>
     ~~| <classifier function>~~
     | <JSON query>
```

2. EDIT THE BNF FOR <CHARACTER VALUE FUNCTION> AS FOLLOWS:

```
14 <character value function> ::=
       <character substring function>
     | <regular expression substring function>
     | <regex substring function>
     | <fold>
     | <transcoding>
     | <character transliteration>
     | <regex transliteration>
     | <trim function>
     | <character overlay function>
     | <normalize function>
     | <specific type method>
     | <classifier function>
```

*[NOTE to the proposal reader: incidental improvement in passing. Since <classifier function> always returns a character string, we can place it within the more specific <character value function>. SR 2) already assumes this placement.]*

3. EDIT SYNTAX RULE 1) AS FOLLOWS. THE EDITOR WILL PLEASE FIX PART 9 AS WELL:

1) 09 The declared type of <string value function> is the declared type of the immediately contained <character value function>**,** ~~or~~ <binary value function> **or <JSON query>**.

4. EDIT GENERAL RULE 1) AS FOLLOWS. THE EDITOR WILL PLEASE EXAMINE PART 9 TO SEE IF IT NEEDS A FIX AS WELL

1) The result of <string value function> is the result of the immediately contained <character value function>**,** ~~or~~ <binary value function> **or <JSON query>**.

## 6.10 New Subclause 6.n <JSON query>

1. ADD THE FOLLOWING SUBCLAUSE AFTER 6.31 <STRING VALUE FUNCTION>.

**Function**

**Extract a JSON text from a JSON text using an SQL/JSON path expression.**

**Format**

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

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

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

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

**Syntax Rules**

**1) If <JSON output clause> is not specified, then RETURNING *SDT* FORMAT JSON is implicit, where *SDT* is an implementation-defined string type.**

**2) The declared type *DECT* of <JSON query> is the type specified by the <data type> *DT* contained in the explicit or implicit <JSON output clause>.**

**3) If <JSON query empty behavior> is not specified, then NULL ON EMPTY is implicit.**

**4) If <JSON query error behavior> is not specified, then NULL ON ERROR is implicit.**

**5) If <JSON query wrapper behavior> is not specified, then WITHOUT ARRAY is implicit**

**6) If <JSON query wrapper behavior> specifies WITH, then**

**a) <JSON query empty behavior> shall not be specified.**

**b) If neither CONDITIONAL nor UNCONDITIONAL is specified, then UNCONDITIONAL is implicit,**

**General Rules**

**1) If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of <JSON query> is the null value of type *DECT* and no further General Rules are performed.**

**2) The General Rules of Subclause 10.n "<SQL/JSON common syntax>" are applied. Let *SEQ* be the *SQL/JSON SEQUENCE* and let *ST* be the *STATUS* that are returned by the Subclause.**

**3) If *ST* is *successful completion*, then**

**a) Case:**

**i) If WITHOUT ARRAY WRAPPER is specified or implied, then let *WRAPIT* be *False***

**ii) If WITH UNCONDITIONAL ARRAY WRAPPER is specified or implied, then let *WRAPIT* be *True.***

**iii) If WITH CONDITIONAL ARRAY WRAPPER is specified, then**

**Case:**

**1) If *SEQ* has a single SQL/JSON item, and that item is an SQL/JSON array or SQL/JSON object, then let *WRAPIT* be *False.***

**2) Otherwise, let *WRAPIT* be *True.***

**b) Case:**

**i) If *WRAPIT* is *False*, then let *SEQ2* be *SEQ.***

**ii) If *WRAPIT* is *True*, then let *SEQ2* be an SQL/JSON sequence with a single SQL/JSON item, that item being an SQL/JSON array whose elements are the items of *SEQ*, preserving the order of the items in the resulting array.**

**c) Case:**

**i) If the length of *SEQ2* is 0 (zero) then**

**Case:**

**1) If the <JSON query empty behavior> is ERROR, then let *ST* be the exception condition *data exception: no SQL/JSON item.***

**2) If the explicit or implicit <JSON query empty behavior> is NULL, then the result of <JSON query> is the null value of type *DECT,* and no further General Rules are performed.**

**3) If the <JSON query empty behavior> is EMPTY ARRAY, then let *V* be an SQL/JSON array with no SQL/JSON elements.**

**4) If the <JSON query empty behavior> is EMPTY OBJECT, then let *V* be an SQL/JSON object with no SQL/JSON members.**

**ii) If the length of *SEQ2* is 1 (uno) then let *V* be the only SQL/JSON item in *SEQ*.**

**iii) Otherwise, let *ST* be the exception condition *data exception: more than one SQL/JSON item.***

**4) Let *FO* be the explicit or implicit <JSON output representation> simply contained in the <JSON output clause>.**

**5) If *ST* is *successful completion,* then the General Rules of Subclause N.4 "Serializing an SQL/JSON item" are applied, with *V* as the *SQL/JSON ITEM, FO* as the *FORMAT OPTION,* and *DECT* as the *TARGET TYPE.* Let *JT* be the *JSON TEXT* and let *ST* be the *STATUS* that are returned by the Subclause.**

**6) Case:**

**a) If *ST* is an exception condition, then**

**Case:**

**i) If <JSON query error behavior> is ERROR, then the exception condition *ST* is raised.**

**ii) If <JSON query error behavior> is NULL, then the result of <JSON query> is the null value of type *DECT.***

**iii) If <JSON query error behavior> is EMPTY ARRAY or EMPTY OBJECT, then:**

**1) Case:**

**A) If <JSON query error behavior> is EMPTY ARRAY, then let *X* be an SQL/JSON array that has no SQL/JSON elements.**

**B) If <JSON query error behavior> is EMPTY OBJECT, then let *X* be an SQL/JSON object that has no SQL/JSON members.**

**2) The General Rules of Subclause N.4 "Serializing an SQL/JSON item" are applied, with *X* as the *SQL/JSON ITEM, FO* as the *FORMAT OPTION,* and *DECT* as the *TARGET TYPE.* Let *JTE* be the *JSON TEXT* and let *STE* be the *STATUS* that are returned by the Subclause.**

**3) Case:**

**A) If *STE* is an exception condition, then the exception condition *STE* is raised.**

**NOTE nnn: for example, if the target type is too small to accomodate an empty array or empty object.**

**B) Otherwise, the result of <JSON query> is *JTE.***

**b) Otherwise, *JT* is the result of <JSON query>.**

**Conformance Rules**

**1) Without Feature Tx28, "Basic SQL/JSON", conforming SQL language shall not contain <JSON query>.**

**2) Without Feature Tx25, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON query> shall not contain <JSON query empty behavior>.**

**3) Without Feature Tx25, "SQL/JSON: ON EMPTY and ON ERROR clauses", <JSON query> shall not contain <JSON query error behavior>.**

**4) Without Feature Tx29, "JSON_QUERY: array wrapper behavior", <JSON query> shall not contain <JSON query wrapper behavior>.**

## 6.11 Changes to 7.6 <table reference>

1. EDIT THE BNF FOR <TABLE PRIMARY> AS FOLLOWS:

```
14 <table primary> ::=
        <table or query name>
          [ <query system time period specification> ]
          [ <correlation or recognition> ]
      | <derived table> <correlation or recognition>
```

```
          | <lateral derived table> <correlation or recognition>
          | <collection derived table>
            <correlation or recognition>
          | <table function derived table>
            <correlation or recognition>
          | <only spec> [ <correlation or recognition> ]
          | <data change delta table>
            [ <correlation or recognition> ]
          | <JSON table> <correlation or recognition>
          | <JSON table primitive> AS <correlation name>
          | <parenthesized joined table>
```

2. Edit Syntax Rule 10) as follows:

   10) Let *TPTI* be the table specified by the <table or query name>, <derived table>, <lateral derived table>, <data change delta table>, **<JSON table primitive>** or the <joined table> simply contained in *TP*. The degree of *TPT* and the column descriptor of each of the columns of *TPT* are determined as follows.

   Case: . . .

## 6.12 New Subclause 7.n <JSON table>

1. Add the following Subclause after 7.6 <table reference>

**7.n <JSON table>**

**Function**

**Query a JSON text and present it as a relational table.**

**Format**

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

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

```
<JSON table column definition> ::=
        <JSON table ordinality column definition>
      | <JSON table regular column definition>
      | <JSON table 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 nested columns> ::=
      NESTED [ PATH ] <JSON table nested path specification>
      [ AS <JSON table nested path name> ]
      <JSON table columns clause>

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

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

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

<JSON table plan clause> ::=
      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 error behavior> ::=
            ERROR
          | EMPTY

     <JSON table primitive> ::=
          JSON_TABLE_PRIMITIVE <left paren>
          <JSON API common syntax>
          <JSON table primitive columns clause>
          <JSON table error behavior> ON ERROR
          <right paren>

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

     <JSON table primitive column definition> ::=
            <JSON table ordinality column definition>
          | <JSON table regular column definition>
          | <JSON table primitive chaining column>
```

```
<JSON table primitive chaining column> ::=
     <column name> FOR CHAINING
```

**Syntax Rules**

**1) If <JSON table> *JTAB* is specified:**

**a) Let *JACS* be the <JSON API common syntax> simply contained in *JTAB*.**

**i) Let *JACSCI* be the <JSON context item> contained in *JACS.***

**iii) Let *JACSPATH* be the <JSON path specification> simply contained in *JACS.***

**iv) Let *JACSPN* be the explicit or implicit <JSON table path name> simply contained in *JACS*.**

**v) If *JACS* simply contains <JSON passing clause>, then let *JACSPC* be that <JSON passing clause>; otherwise let *JACSPC* be the empty string.**

**a.1) If <JSON table error behavior> is not specified, then EMPTY ON ERROR is implicit. Let *JTEB* be the explicit or implicit <JSON table error behavior>.**

**b) Let *JTABCOLS* be the <JSON table columns clause> simply contained in *JTAB.***

**c) Throughout *JTAB*, there is a three-way association between <JSON path name>s, <JSON path specification>s and <JSON table columns clause>s. At the outermost level, *JACSPN, JACSPATH,* and *JTABCOLS* are associated. Additional associations are defined later for each <JSON table nested columns> contained in *JTAB.* The <JSON path name> and the <JSON table columns clause> in an association uniquely determines the other members of the association.**

**d) For every <JSON table regular column definition> *JTRCD* contained in *JTAB:***

**i) Let *JTRCN* be the <column name> simply contained in *JTRCD.***

**ii) The <data type> *JTRCDT* contained in *JTRCD* shall be a <predefined type> that identifies a character string data type, numeric data type, boolean data type or datetime data type.**

**iii) If *JTRCD* does not contain <JSON table column empty behavior> then**

**Case:**

**1) If *JTEB* is ERROR ON ERROR, then the implicit <JSON table column empty behavior> of *JTRCD* is ERROR ON EMPTY.**

**1) Otherwise, the implicit <JSON table column empty behavior> of *JTRCD* is NULL ON EMPTY.**

**iii.2) If *JTRCD* does not contain <JSON table column error behavior> then**

**Case:**

**1) If *JTEB* is ERROR ON ERROR, then the implicit <JSON table column error behavior> of *JTRCD* is ERROR ON ERROR.**

**2) Otherwise, the implicit <JSON table column error behavior> of *JTRCD* is NULL ON ERROR.**

**v) Case:**

**1) If *JTRCD*  does contains <JSON column path specification>, then let *JTRCPATH* be the <JSON column path specification> contained in *JTRCD.***

**2) Otherwise, let  *JTRCPATH* be a <character string literal> consisting of:**

**A) the characters <dollar sign> <period>, followed by**

**B) A JSON string whose value is the same characters as in *JTRCN* in order.**

**vi)  The Syntax Rules of Subclause N.n "SQL/JSON path language: syntax and semantics" are applied, with *JTRCPATH* as the *PATH SPECIFICATION* and an empty character string as the *PASSING CLAUSE.***

**e) For every <JSON table nested columns> *JTNC* contained in *JTAB*:**

**i) If *JTAB* contains an explicit <JSON plan clause> then *JTNC* shall contain <JSON table nested path name>.**

**ii) If <JSON table path name> is not specified, then an implementation-dependent <JSON table path name> is implicit.**

**iii) The scope of an explicit or implicit <JSON table path name> *JTNCPN* is the explicit or implicit <JSON table plan clause>.**

**iv) Let *JTNCPATH* be the <JSON table nested path specification> of *JTNC.***

**iv.1)  The Syntax Rules of Subclause N.n "SQL/JSON path language: syntax and semantics" are applied, with *JTNCPATH* as the *PATH SPECIFICATION* and an empty character string as the *PASSING CLAUSE.***

**v) Let *JTNCCOLS* be the <JSON table columns clause> simply cotnained in *JTNC.***

**vi) *JTNCPN*, *JTNCPATH*, and *JTNCCOLS* are associated.**

**f) Within *JTAB*, the following shall be distinct from one another: *JACSPN,* every <JSON table path name> contained in a <JSON table nested columns>, and every <column name>.**

**NOTE nnn: in the syntactic transformation defined later, <JSON table path name>s are used as column names, hence <JSON table path name>s must be distinct from the <column name>s.**

**g) If <JSON table plan clause> is not specified, then an implicit <JSON table plan clause> is constructed as follows:**

**i) Let *COLTREE* be a tree whose nodes are the <JSON table columns clause>s contained in *JTAB* without an intervening <JSON table>, arranged according to syntactic containment, that is, for every interior node *N*, the children of *N* are the <JSON table column clause>s that are simply contained in *N*.**

**ii) Let the nodes of *COLTREE* be $CNODE_1$, . . ., $CNODE_{CN}$, enumerated in order of occurrence of the keyword COLUMNS in *JTAB*.**

**iii) For all *c*, 1 (one) <= *c* <= *CN*, let $CPN_c$ be the <JSON path name> associated with $CNODE_c$.**

**iv) For all *c*, 1 (one) <= *c* <= *CN*, let $CHILDREN_c$ be the set of child nodes of $CNODE_c$. Let $NCHILD_c$ be the number of nodes in $CHILDREN_c$.**

**v) for all *c*, *CN* >= *c* >= 1 (one) in descending order,**

**Case:**

**1) If $NCHILD_c$ is 0 (zero), let $CPLAN_c$ be $CPN_c$.**

**NOTE nnn: these are the leaves of *COLTREE*.**

**2) If $NCHILD_c$ is 1 (one), let $NCHILD_d$ be the only descendant node of $NCHILD_c$. Let $CPLAN_c$ be**

```
( CPNc OUTER CPLANd )
```

**NOTE nnn: this means that a left outer join will connect the evaluation of the parent <JSON table columns clause> to the evaluation of the child <JSON table columns clause>.**

**3) Otherwise,**

**A) Let $CHILDREN_c$ = { $CNODE_{d1}$, . . ., $CNODE_{dn}$ }.**

**NOTE nnn: the subscripts *d1* through *dn* of the child nodes are not necessarily consecutive integers.**

**B) Let $\mathit{CPLAN}_c$ be**

$\texttt{( } \mathit{CPN}_c \texttt{ OUTER ( } \mathit{CPLAN}_{d1} \texttt{ UNION ... UNION } \mathit{CPLAN}_{dn} \texttt{ )}$

**NOTE nnn: this means that the child <JSON table column clause>s will be connected using "union join" (the same as a full otuer join with a condition that is never satisfied). A left outer join will connect the parent <JSON table columns clause> to the union join of all the children.**

**vi) The implicit <JSON table plan clause> of *JTAB* is**

$\texttt{PLAN ( } \mathit{CPLAN}_1 \texttt{ )}$

**NOTE nnn: For example, consider the folllowing schematic <JSON table>:**

```
JSON_TABLE ( JACSCI, 'JACSPATH' AS JACSPN
  COLUMNS (
    NESTED PATH '...' AS $CPN_2$ COLUMNS (...),
    NESTED PATH '...' AS $CPN_3$ COLUMNS (...)
  ) )
```

**Then *COLTREE* may be diagrammed as shown below, using <JSON path name>s to label the nodes:**

$$\begin{tikzcd}
 & \mathit{CPN}_1 = \mathit{JACSPN} \arrow[dl] \arrow[dr] & \\
\mathit{CPN}_2 & & \mathit{CPN}_3
\end{tikzcd}$$

**The implicit <JSON table plan clause> is**

$\texttt{PLAN ( } \mathit{JACSPN} \texttt{ OUTER ( } \mathit{CPN}_2 \texttt{ UNION } \mathit{CPN}_3 \texttt{ ) )}$

**h) Let *JTPLAN* be the implicit or explicit <JSON table plan> simply contained in *JTAB*.**

**i) Every implicit or explicit <JSON table path name> contained in *JTAB* shall appear in *JTPLAN* exactly once.**

**ii) For every <JSON table plan parent/child> *JTPPC* contained in *JTPLAN*, let *LEFTOP* be the first <JSON table path name> contained in *JTPPC*, and let *OTHEROP* be any other <JSON table path name> contained in *JTPPC*. Let *LEFTCOLS* be the <JSON table columns clause> that is**

associated with *LEFTOP*, and let *OTHERCOLS* be the <JSON table columns clause> associated with *OTHEROP*. Then *OTHERCOLS* shall be contained in *LEFTCOLS*.

j) The plan tree *JTTREE* of *JTAB* is a tree whose leaves are the <JSON table path name>s contained in *JTPLAN* and whose interior nodes are the BNF non-terminals contained in *JTPLAN* that contain a <JSON table path name>. The nodes of *JTTREE* are arranged according to syntactic containment, that is, for every interior node *N*, the children of *N* are the BNF non-terminals immediately contained in *N*, excluding <left paren> and <right paren>.

NOTE nnn: continuing the prior example, given the plan

```
PLAN ( JACSPN OUTER ( CPN2 UNION CPN3 ) )
```

then the plan tree may be diagrammed (using OUTER to label a <JSON table plan outer> node and UNION to label a <JSON table plan union> node):

$$\begin{tikzcd}
& \text{OUTER} \arrow[dl] \arrow[dr] & & \\
\mathit{JACSPN} & & \text{UNION} \arrow[dl] \arrow[dr] & \\
& \mathit{CPN}_2 & & \mathit{CPN}_3
\end{tikzcd}$$

i) Let the nodes of *JTTREE* be $JTNODE_1, \ldots, JTNODE_{NN}$, enumerated in an implementation-dependent order such that for all *i1*, 1 (one) <= *i1* <= *NN* and for all *i2*,1 (one) <= *i* <= *NN*, if node $N_{i1}$ is contained in node $N_{i2}$, then *i1* < *i2*.

NOTE nnn: this can be done using a depth-first traversal of *JTTREE*.

ii) For all *i*, 1 (one) <= *i* <= *NN*, let $JTCORR_i$ be a <correlation name> defined as follows:

1) If $JTNODE_i$ is a <JSON table path name> *JTPN*, then let $JTCORR_i$ be *JTPN*.

2) If $JTNODE_i$ has only one child $JTNODE_c$, then let $JTCORR_i$ be the same as $JTCORR_c$.

NOTE nnn: this handles the cases where $JTNODE_i$ is a <JSON table plan>, <JSON table plan parent.child>, <JSON table plan sibling>, and <JSON table plan primary>.

**3) Otherwise, let $JTCORR_i$ be an implementation-dependent <corrrelation name> that is distinct from any <JSON table path name> and from any other <correlation name> chosen by this rule.**

**iii) For all *i*, 1 (one) <= *i* <= *NN*, a <select list> $JTSL_i$ and a <table primary> $JTPRIM_i$ are defined as follows.**

**Case:**

**1) If $JTNODE_i$ has exactly one child $JTNODE_c$, then let $JTSL_i$ be the same as $JTSL_c$ and let $JTPRIM_i$ be the same as $JTPRIM_c$, .**

**2) If $JTNODE_i$ is a <JSON table plan outer>, then let $JTNODE_a$ be the <JSON table path name> that is the first operand of $JTNODE_i$ and let $JTNODE_b$ be the <JSON table plan primary> that is the second operand of $JTNODE_i$. Then let $JTSL_i$ be**

```
        JTSL_a, JTSL_b
```

**and let $JTPRIM_i$ be**

```
LATERAL ( SELECT JTSL_a, JTSL_b
          FROM JTPRIM_a LEFT OUTER JOIN JTPRIM_b
                ON 0=0
         ) AS JTCORR_i
```

**2) If $JTNODE_i$ is a <JSON table plan inner>, then let $JTNODE_a$ be the <JSON table path name> that is the first operand of $JTNODE_i$ and let $TNODE_b$ be the <JSON table plan primary> that is the second operand of $JTNODE_i$. Then let $JTSL_i$ be**

```
        JTSL_a, JTSL_b
```

**and let $JTPRIM_i$ be**

```
LATERAL ( SELECT JTSL_a, JTSL_b
          FROM JTPRIM_a , JTPRIM_b
         ) AS JTCORR_i
```

2. NOTE TO EDITOR: IN THE NEXT RULE, SUBSCRIPTS *A, B, ... Z* ARE NOT NECESSARILY CONTIGUOUS AND *Z* IS NOT THE NUMBER OF ITEMS IN THE LIST, SO IT WILL NOT BE CORRECT TO SAY "LET *Z* BE THE NUMBER OF <JSON TABLE PLAN PRIMARY>S THAT ARE CHILDREN OF $JTNODE_I$". IF YOU WANT TO DO THAT, WHAT WE NEED IS "LET *NC* BE THE NUMBER OF <JSON TABLE PLAN PRIMARY>S THAT ARE CHILDREN OF $JTNODE_I$ AND LET *F* BE A FUNCTION THAT ENUMERATES THE SUBSCRIPTS OF THOSE CHILDREN, IE, $JTNODE_{F(1)}$ IS THE FIRST CHILD OF $JTNODE_I$. AND $JTNODE_{F(NC)}$ IS THE LAST CHILD OF $JTNODE_I$"

**2) If *JTNODE*$_i$ is a <JSON table plan union>, then let *JTNODE*$_a$, *JTNODE*$_b$, . . ., *JTNODE*$_z$ be the <JSON table plan primary>s that are the children of *JTNODE*$_i$ . Then let *JTSL*$_i$ be**

```
    JTSLa, ..., JTSLz
```

**and let *JTPRIM*$_i$ be**

```
LATERAL ( SELECT JTSLa, ..., JTSLz
          FROM
     JTPRIMa FULL OUTER JOIN JTPRIMb ON 0=1
             FULL OUTER JOIN JTPRIMc ON 0=1
               . . .
             FULL OUTER JOIN JTPRIMz ON 0=1
        ) AS JTCORRi
```

3. PREVIOUS NOTE TO THE EDITOR APPLIES TO THE NEXT RULE AS WELL:

**2) If *JTNODE*$_i$ is a <JSON table plan cross>, then let *JTNODE*$_a$, *JTNODE*$_b$, . . ., *JTNODE*$_z$ be the <JSON table plan primary>s that are the children of *JTNODE*$_i$ . Then let *JTSL*$_i$ be**

```
    JTSLa, ..., JTSLz
```

**and let *JTPRIM*$_i$ be**

```
LATERAL ( SELECT JTSLa, ..., JTSLz
          FROM JTPRIMa, JTPRIMb, ... JTPRIMz
        ) AS JTCORRi
```

**3) If *JTNODE*$_i$ is a <JSON table path name> *JTPN*$_i$, then:**

**A) Let *JTPATH*$_i$ be the <JSON path specification> associated with *JTPN*$_i$ and let *JTCOLS*$_i$ be the <JSON table columns clause> associated with *JTPN*$_i$.**

**B) Case:**

**I) If *JTPN*$_i$ is *JACSPN,* then let *JTCI*$_i$ be *JACSCI.***

**II) Otherwise, let *ANCESTOR* be the <JSON table plan inner> or <JSON table plan outer> that simply contains *JTPN*$_i$ in the second (<JSON table plan primary>) argument. Let *ANCESTORPN* be the <JSON table path name> that is the first operand of *ANCESTOR*. Let *JTCI*$_i$ be**

```
        ANCESTORPN.JTPNi
```

**NOTE nnn: <JSON path name>s are used both as correlation names and column names. Here *ANCESTORPN* is the correlation name of the query and $JTPN_i$ is the column name of the column that generates the context item for $JTPRIM_i$.**

**B.1) Let $JTCD_1$, . . ., $JTCD_{ND}$ be the <JSON table column definition>s simply contained in $JTCOLS_i$.**

**For all *d*, 1 (one) <= *d* <= *ND*,**

**Case:**

**I) If $JTCD_d$ is a <JSON table nested columns> *JTNC*, then let *JTCC* be the <JSON table columns clause> simply contained in *JTNC*, let $COLN_d$ be the <JSON table path name> associated with *JTCC*, and let $JTCDNEW_d$ be**

```
        COLN_d FOR CHAINING
```

**II) If $JTCD_d$ is a <JSON table orinality column definition>, then let $JTCDNEW_d$ be $JTCD_d$.**

**III) Otherwise, let $COLN_d$ be the <column name> contained in $JTCD_d$ and let $JTCDNEW_d$ be $JTCD_d$, with any implicit <JSON table column empty behavior> or <JSON table column error behavior> made explicit.**

**C) Let $JTSL_i$ be**

```
    COLN_1, . . ., COL_ND
```

**and let $JTPRIM_i$ be**

```
LATERAL ( SELECT COLN_1, . . ., COL_ND
          FROM JSON_TABLE_PRIMITIVE (
                JTCI_i,
                JTPATH_i AS JACSPN
                JACSPC
                COLUMNS (
                JTCDNEW_1, ... JTCDNEW_ND )
                JTEB ON ERROR ) AS JTPN_i
```

**k) Let $CN_1$, . . . , $CN_{NC}$ be the list of all <column name>s contained in *JTAB*, in order.**

**NOTE nnn: this includes all <column name>s in any <JSON table nested columns> at any depth of nesting, but excludes the columns whose names are**

**<JSON table path name>s created for chaining between JSON_TABLE_PRIMITIVE invocations.**

**l) *JTAB* is equivalent to**

```
LATERAL ( SELECT CN1, . . . , CNNC
          FROM JTPRIMNN )
```

**NOTE nnn: the <correlation or recognition> is suffixed to the preceding <lateral derived table>.**

**2) <JSON table primitive> shall not be specified.**

**NOTE nnn: <JSON table primitive> is a specification device used to define the semantics of nested tables in <JSON table> and is not syntax available to the user.**

**3) The degree of the table specified by <JSON table primitive> *JTP* is the number of <JSON table primitive column definition>s simply contained in *JTP*. For each <JSON table primitive column definition> *JTPCD* contained in *JTP*, a column descriptor is created, as follows:**

**a) The column name of the column is the <column name> simply contained in *JTPCD***

**b) Case:**

**i) If *JTPCD* is a <JSON table ordinality column definition> then the declared type of the column is an implementation-defined exact numeric type with scale 0 (zero).**

**ii) If *JTPCD* is a <JSON table regular column definition> then the declared type of the column is the type defined by the <data type> simply contained in *JTPCD*.**

**iii) If *JTPCD* is a <JSON table primitive chaining column> then the declared type of the column is an implementation-dependent string type.**

## General Rules

**1) If a <table primary> simply contains a <JSON table primitive> *JTP*, then**

**a) If the value of the <JSON context item> simply contained in the <JSON API common syntax> is the null value, then the result of <JSON table primitive> is an empty table and no further General Rules are performed.**

**b) Let *JACS* be the <JSON API common syntax> simply contained in *JTP*.**

**c) Let *JTEB* be the <JSON table error behavior> simply contained in *JTP*.**

**d) The General Rules of Subclause 10.n+1 "<JSON API common syntax>" are performed. Let *ROWST* be the *STATUS* and let *ROWSEQ* be the *SQL/JSON SEQUENCE* returned by that Subclause.**

**e) Case:**

**i) If *ROWST* is an exception condition, then**

**Case:**

**1) If *JTEB* is ERROR, then the exception condition *ST* is raised.**

**2) Otherwise, the result of *JTP* is an empty table.**

**ii) Otherwise, let $I_1, \ldots, I_{NI}$ be the SQL/JSON items in *ROWSEQ* in order, and let $JTCD_1, \ldots, JTCD_{NCD}$ be the <JSON table primitive column definition>s contained in *JTP*.**

**For all *j*, 1 (one) <= *j* <= *NI*, and for all *i*, 1 (one) <= *i* <= *NCD*, the value of the *i*-th column of the *j*-th row in the result of *JTP* is determined as follows:**

**Case:**

**1) If $JTCD_i$ is a <JSON table ordinality column definition> then the value of the *i*-th column of *j*-th row is *j*.**

**2) If $JTCD_i$ is a <JSON table regular column definition>, then**

**A) Let $JTCDPATH_i$ be the <JSON table column path specification>, let $ZB_i$ be the <JSON table column empty behavior>, let $EB_i$ be the <JSON table column error behavior> contained in $JTCD_i$ and let $DT_i$ be the <data type> simply contained in $JTCD_i$.**

**B) The General Rules of Subclause N.n "SQL/JSON path language: syntax and semantics" are applied, with $JTCDPATH_i$ as the *PATH SPECIFICATION*, $I_j$ as the *CONTEXT ITEM*, *True* as the *ALREADY PARSED* and an empty character string as the *PASSING CLAUSE*. Let *SEQ* be the *SQL/JSON SEQUENCE* and let *ST1* be the *STATUS* that are returned by the Subclause.**

**C) The General Rules of Subclause N.n "Casting an SQL/JSON sequence to an SQL type" are applied, with *ST1* as the *STAUTS IN*, *SEQ* as the *SQL/JSON SEQUENCE*, $ZB_i$ as the *EMPTY BEHAVIOR*, $EB_i$ as the *ERROR BEHAVIOR*, and $DT_i$ as the *DATA TYPE*. Let *ST2* be the *STATUS OUT* and let *V* be the *VALUE* that are returned by that Subclause.**

**D) If *ST2* is an exception condition, then the exception condition *ST2* is raised. Otherwise, the value of the *i*-th column of the *j*-th row is *V*.**

**3) If *$JTCD_i$* is a <JSON table primitive chaining column>, then the General Rules of Subclause N.n “Serializing an SQL/JSON item” are executed, with *$I_j$* as the *ITEM*, an implementation-dependent <JSON output representation> as the *FORMAT OPTION*, and an implementation-dependent data type as the *TARGET TYPE*. Let *JT* be the *JSON TEXT* and let *CHAINST* be the *STATUS* that are returned by those rules.**

**Case:**

**A) If *ST* is an exception condition, then**

**Case:**

**I) If *JTEB* is ERROR, then the exception condition *ST* is raised.**

**II) Otherwise, let *JT* be the null value.**

**NOTE nnn: Because of the syntactic transformation in the Syntax Rules, this value will be the context item of a different <JSON table primitive> invocation. The null value cannot be parsed, so that query will result in an empty table (not an error, since *JTEB* is EMPTY).**

**B) The value of the *i*-th column of the *j*-th row of the result of *JTP* is *JT*.**

**Conformance Rules:**

**1) Without Feature Tx21, “Basic SQL/JSON”, conforming SQL language shall not contain <JSON table>.**

**2) Without Feature Tx24, “JSON_TABLE: PLAN clause”, <JSON table> shall not contain <JSON table plan clause>.**

**3) Without Feature Tx25, “SQL/JSON: ON EMPTY and ON ERROR clauses”, <JSON table> shall not contain <JSON table error behavior>.**

**4) Without Feature Tx25, “SQL/JSON: ON EMPTY and ON ERROR clauses”, <JSON table> shall not contain <JSON table column empty behavior>.**

**5) Without Feature Tx25, “SQL/JSON: ON EMPTY and ON ERROR clauses”, <JSON table> shall not contain <JSON table column error behavior>.**

**6) Without Feature Tx26, “General <value expression> in ON ERROR or ON EMPTY clauses”, the <value expression> contained in <JSON table column empty behavior> or <JSON table column error behavior> contained in a <JSON**

**table regular column definition> *JTRCD* shall be a <literal> that can be cast to the data type specified by the <data type> contained in *JTRCD* without raising an exception condition according to the General Rules of Subclause 6.13 <cast specification>.**

**7) Without Feature Tx27, "JSON_TABLE: sibling NESTED COLUMNS clauses", an explicit or implicit <JSON table plan> shall not contain <JSON table plan sibling>.**

## 6.13 NO changes to 7.15 <query specification>

*[NOTE to the proposal reader: The definition of "known not null column" does not need to chang because none of the JSON query operators are ever known not null.]*

## 6.14 Changes to 8.1 <predicate>

1. APPEND <JSON EXISTS PREDICATE> TO <PREDICATE>:

```
14 <predicate> ::=
       <comparison predicate>
     | <between predicate>
     | <in predicate>
     | <like predicate>
     | <similar predicate>
     | <regex like predicate>
     | <null predicate>
     | <quantified comparison predicate>
     | <exists predicate>
     | <unique predicate>
     | <normalized predicate>
     | <match predicate>
     | <overlaps predicate>
     | <distinct predicate>
     | <member predicate>
     | <submultiset predicate>
     | <set predicate>
     | <type predicate>
     | <period predicate>
     | <JSON predicate>
     | <JSON exists predicate>
```

2. EDIT GENERAL RULE 1) AS FOLLOWS:

1) 14 The result of a <predicate> is the truth value of the immediately contained <comparison predicate>, <between predicate>, <in predicate>, <like predicate>, <similar predicate>, <regex like predicate>, <null predicate>, <quantified comparison

predicate>, <exists predicate>, <unique predicate>, <normalized predicate>, <match predicate>, <overlaps predicate>, <distinct predicate>, <member predicate>, <submultiset predicate>, <set predicate>, <type predicate>, ~~or~~ <period predicate>**, <JSON predicate> or <JSON exists predicate>**.

## 6.15 New Subclause 8.n+0 <JSON predicate>

1. ADD THE FOLLOWING SUBCLAUSE TO THE END OF CLAUSE 8 "PREDICATES"

**8.n+0 <JSON predicate>**

**Function**

**Test whether a string value is a JSON text.**

**Format**

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

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

**Syntax Rules**

**1) If <JSON input clause> is not specified, then FORMAT JSON is implicit.**

**1.1) If <JSON predicate uniqueness constraint> is not specified, then WITHOUT UNIQUE KEYS is implicit.**

**2) Let *SVE* be the <string value expression>, *FO* be the explicit or implicit <JSON input clause>, and *JPUC* be the explicit or implicit <JSON predicate uniqueness constraint>. Then**

```
SVE FO IS NOT JSON JPUC
```

**is equivalent to**

```
NOT (SVE FO IS JSON JPUC)
```

**Access Rules**

**None**

**General Rules**

1) Let *SVE* be the value of the <string value expression>. Let *FO* be the explicit or implicit <JSON input clause>

2) Case:

a) If *SVE* is the null value, then the value of the <JSON predicate>

```
SVE FO IS JSON
```

is *Unknown*.

b) Otherwise, the General Rules of Subclause N.3 "Parsing a JSON text" are performed, with *SVE* as the *JSON TEXT*, *FO* as the *FORMAT OPTION*, and *JPUC* as the *UNIQUENESS CONSTRAINT*. Let *SJI* be the *SQL/JSON ITEM* and let *ST* be the *STATUS* that are returned from the Subclause.

The value of the <JSON predicate>

```
SVE FO IS JSON
```

is

Case:

i) If *ST* is *successful completion*, then *True*.

ii) Otherwise, *False*.

**Conformance Rules**

1) Without Feature Tx21, "Basic SQL/JSON", conforming SQL language shall not contain <JSON predicate>.

2) Without Feature Tx22, "SQL/JSON: IS JSON WITH UNIQUE KEYS predicate"

## 6.16 New Subclause 8.n+1 <JSON exists predicate>

1. ADD THE FOLLOWING SUBCLAUSE TO THE END OF CLAUSE 8 "PREDICATES"

**8.n <JSON exists predicate>**

**Function**

Test whether a JSON path expression returns any SQL/JSON items.

**Format**

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

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

## Syntax Rules

1) If <JSON exists error behavior> is not specified, then the default is FALSE ON ERROR.

## Access Rules

None

## General Rules

0) 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*** and no further General Rules are performed.

1) Let *JEEB* be the explicit or implicit <JSON exists error behavior>.

2) The General Rules of Subclause 10.n "<SQL/JSON common syntax>" are applied. Let *SEQ* be the *SQL/JSON SEQUENCE* and let *ST* be the *STATUS* that are returned by the Subclause.

3) The result of <JSON exists predicate> is

Case:

a) If *ST* is *successful completion*, then

Case:

a) If the length of *SEQ* is 0 (zero), then ***False***.

b) Otherwise, ***True***.

b) Otherwise,

Case:

i) If *JEEB* is ERROR, then the exception condition *ST* is raised.

ii) If *JEEB* is TRUE, then ***True***.

ii) If *JEEB* is FALSE, then ***False***.

ii) If *JEEB* is UNKNOWN, then ***Unknown***.

**Conformance Rules**

**1) Without Feature Tx21, “Basic SQL/JSON”, conforming SQL language shall not contain <JSON exists predicate>.**

**2) Without Feature Tx25, “SQL/JSON: ON EMPTY and ON ERROR clauses”, <JSON exists predicate> shall not contain <JSON exists error behavior>.**

## 6.17 New Subclause 9.n+0, Parsing a JSON text

*[NOTE to the proposal reader: this section duplicates material in [SQL/JSON part 1]; it is repeated here because it is integral to this proposal as well.]*

1. ADD THE FOLLOWING SUBCLAUSE TO CLAUSE 9 “ADDITIONAL COMMON RULES”

**Parsing a JSON text**

**Subclause Signature**

```
"Parsing a JSON text" [Syntax Rules] (
)

"Parsing a JSON text" [General Rules] (
  Paramter: "JSON TEXT",
  Parameter: "FORMAT OPTION",
  Parameter: "UNIQUENESS CONSTRAINT"
) Returns: "SQL/JSON ITEM", "STATUS"
```

**Funtion**

**Convert a JSON text to an SQL/JSON item**

**Syntax Rules**

**None**

**Access Rules**

**None**

**General Rules**

**1) Let *JV* be the *JSON TEXT*, let *FO* be the *FORMAT OPTION* and let *UC* be the *UNIQUENESS CONSTRAINT* in an application of this Subclause. The result of the application of this Subclause is *SJI*, returned as *SQL/JSON ITEM,* and *ST*, returned as *STATUS.***

**2) Let *ST* be the condition: *successful completion.***

**3) Case:**

**a) If *FO* is JSON, then:**

**i) Case:**

**1) If *JV* is a character string, then let *ENC* be the Unicode encoding of *JV*.**

**2) Otherwise, let *ENC* be the encoding determined by Section 3 “Encoding” in [RFC4627].**

**ii) *JV* is parsed according to the grammar of Section 2 “JSON grammar” in [RFC4627], using the encoding *ENC*.**

**iii) Case:**

**1) If *JV* is not a JSON text, then let *ST* be the exception condition: *data exception — invalid JSON text.***

**1.1) If *UC* is WITH UNIQUE KEYS and *JV* contains a JSON object that has two JSON members whose keys are equivalent Unicode character strings, then let *ST* be the exception condition: *data exception — non-unique keys in a JSON object***

**2) Otherwise:**

**A) The function *F* transforming a JSON text fragment *J* to an SQL/JSON item is defined recursively according to the grammar of Section 2 “JSON grammar” in [RFC4627], as follows:**

**I) If *J* is the JSON literal `false`, then *F*(*J*) is the truth value *False*.**

**II) If *J* is the JSON literal `true`, then *F*(*J*) is the truth value *True*.**

**III) If *J* is the JSON literal `null`, then *F*(*J*) is the SQL/JSON null value.**

**IV) If *J* is a JSON number, then *F*(*J*) is the value of the <signed numeric literal> whose characters are identical *J*.**

**V) If *J* is a JSON string, then *F*(*J*) is an SQL character string whose character set is Unicode and whose characters are the ones enclosed by quotation marks in *J* after replacing any escape sequences by their unescaped equivalents.**

**VI) If *J* is a JSON array, then *F*(*J*) is the SQL/JSON array whose elements are obtained by applying the transform *F* to each element of *J* in turn.**

**VIII) If *J* is a JSON object, then:**

**1) Let $M_1, \ldots, M_m$ be the members of *J*, enumerated in an implementation-dependent order. For all *i*, 1 (one) <= *i* <= *m*, let $K_i$ be the key and let $V_i$ be the bound value of $M_i$. Let $SJM_i$ be the SQL/JSON member whose key is $F(K_i)$ and whose bound value is $F(BV_i)$.**

**1.1) The following rule is performed an implementation-dependent number of times:**

**If any pair of keys $K_i$ and $K_j$ are equal, where $i \neq j$, then one of $M_i$ or $M_j$ is deleted from the list of members of *J*.**

**2) $F(J)$ is the SQL/JSON object whose members are $SJM_1, \ldots, SJM_m$.**

**B) Let *SJI* be *F*(*JV*)**

**b) Otherwise, let *SJI* be the SQL/JSON object or SQL./JSON array obtained using implementation-defined rules for parsing *JV* according to format *FO* and uniqueness constraint *UC*. If there is an error during this conversion, let *ST* be an implementation-defined exception condition.**

**4) *SJI* is the *SQL/JSON ITEM* and *ST* is the *STATUS* that is the result of the application of this Subclause.**

**Conformance Rules**

**None**

## 6.18 New Subclause 9.n+1, Serializing an SQL/JSON item

*[NOTE to the proposal reader: this section duplicates material in [SQL/JSON part 1]; it is repeated here because it is integral to this proposal as well.]*

1. ADD THE FOLLOWING SUBCLAUSE TO NEW CLAUSE 9 "ADDITIONAL COMMON RULES"

**Subclause Signature**

```
"Serializing an SQL/JSON item" [Syntax Rules] (
  Parameter: "FORMAT OPTION",
  Parameter: "TARGET TYPE"
)

"Serializing an SQL/JSON item" [General Rules] (
  Parameter: "SQL/JSON ITEM",
  Parameter: "FORMAT OPTION",
  Parameter: "TARGET TYPE"
) Returns: "JSON TEXT", "STATUS"
```

**Function**

**Serialize an SQL/JSON item as a JSON text.**

**Syntax Rules**

**1) Let *FO* be the *FORMAT OPTION* and let *TT* be the *TARGET TYPE* in an application of this Subclause.**

**2) Case:**

**a) If *FO* contains JSON, then *TT* shall be either a character string type or a binary string type. If *TT* is a character string type, then the character set of *TT* shall be a Universal Character Set.**

**b) Otherwise, *TT* shall be an implementation-defined data type appropriate to the format identified by *FO*.**

**Access Rules**

**None**

**General Rules**

**1) Let *SJI* be the *SQL/JSON ITEM,* let *FO* be the *FORMAT OPTION* and let *TT* be the *TARGET TYPE* in an application of this Subclause. The result of this Subclause is a JSON text *JV* of type *TT* returned as *JSON TEXT,* and a completion condition *ST* returned as *STATUS.***

**2) Case:**

**a) If *FO* contains JSON then:**

**i) Case:**

**1) If *TT* is a character string type, then let *ENC* be the Unicode encoding of *TT.***

**2) If *TT* is a binary string type, then let *ENC* be UTF8, UTF16 or UTF32 as sepcified in the <JSON output representation> contained in *FO.***

**ii) Let *JV* be an implementation-dependent value of type *TT* and encoding *ENC* such that these two conditions hold:**

**1) *JV* is a JSON text.**

**NOTE nnn: It follows that it is an error if *SJI* is not an SQL/JSON array or SQL/JSON object.**

**2) When applying the General Rules of Subclause 9.n+0 "Parsing a JSON text" with *JV* as the *JSON TEXT* , *FO* as the *FORMAT OPTION* and WITHOUT UNIQUE KEYS as the *UNIQUENESS CONSTRAINT,* the returned *STATUS* is *successful completion* and the returned *SQL/JSON ITEM* is an SQL/JSON item that is equivalent to *SJI*.**

**If there is no such *JV*, then let *ST* be the exception condition: *data exception — invalid JSON text.***

**iii) If *JV* is longer than the length or maximum length of *TT*, then an exception condition is raised: *data exception — string data, right truncation.***

**b) Otherwise, let *JV* be an implementation-dependent value such that, when applying the General Rules of Subclause 9.n+0 "Parsing a JSON text" with *JV* as the *JSON TEXT* , an implementation-defined <JSON predicate uniqueness constraint> as the *UNIQUE CONSTRAINT,* and *FO* as the *FORMAT OPTION,* the returned *STATUS* is *successful completion* and the returned *SQL/JSON ITEM* is an SQL/JSON item that is equivalent to *SJI* according to an implementation-defined definition of this equivalence. If there is no such *JV*, then let *ST* be the exception condition: *data exception — invalid JSON text.***

**3) *JV* is the *JSON TEXT* and *ST* is the *STATUS* that are the result of the application of this Subclause.**

**Conformance Rules**

**None**

## 6.19 New Subclause 9.n+2, SQL/JSON path language: lexical elements

1. ADD THE FOLLOWING TO CLAUSE 9 "ADDITIONAL COMMON RULES"

**N.n SQL/JSON path language: lexical elements**

**Function**

**Specify the lexical analysis of the SQL/JSON path language.**

**Format**

```
<SQL/JSON special symbol> ::=
     | <asterisk>
     | <at sign>
     | <comma>
     | <dollar sign>
```

```
        | <double ampersand>
        | <double equals>
        | <double vertical bar>
        | <exclamation mark>
        | <greater than operator>
        | <greater than or equals operator>
        | <left bracket>
        | <left paren>
        | <less than operator>
        | <less than or equals operator>
        | <minus sign>
        | <not equals operator>
        | <percent>
        | <period>
        | <plus sign>
        | <question mark>
        | <right bracket>
        | <right paren>
        | <solidus>

<at sign> ::= @

<double ampersand> ::= &&

<double equals> ::= ==

<double vertical bar> ::= ||

<exclamation mark> ::= !
```

**NOTE nnn: other <SQL/JSON special symbol>s have the same definition as in Subclause 5.1 <SQL special symbol>**

```
<SQL/JSON key word> ::=
          abs
        | ceiling
        | datetime
        | double
        | exists
        | false
        | flag
        | floor
        | is
        | keyvalue
        | last
        | lax
        | like_regex
        | null
```

```
          | size
          | starts
          | strict
          | to
          | true
          | type
          | unknown
          | with

<JSON path literal> ::= !! see Syntax Rules

<JSON path string literal> ::= !! see Syntax Rules

<JSON path numeric literal> ::= !! see Syntax Rules

<JSON path identifier> ::= !! See the Syntax Rules

<JSON path context variable> ::= !! See the Syntax Rules

<JSON path named variable> ::= !! See the Syntax Rules

<JSON path key name> ::= !! See the Syntax Rules
```

## Syntax Rules

1) SQL/JSON path language adopts the conventions in [ECMAScript] section 6 "Source text" regarding the source text of an SQL/JSON path expression, and escape sequences used in that source text.

NOTE nnn: thus an SQL/JSON path expression is assumed to be in Unicode. An implementation may provide a conversion to Unicode if the SQL/JSON path expression is written in a character string of some other character set; any such conversion is outside the scope of this standard.

2) SQL/JSON path language adopts the lexical rules of [ECMAScript] section 5.1.2 "Lexical and RegExp grammars" and section 7 "Lexical conventions", with the following modifications:

a) The only goal symbol is *InputElementDiv* (modifies section 7 "Lexical convertions" paragraph 2).

b) there are no comments (modifies section 7.4 "Comments")

d) there are no reserved words (modifies section 7.6.1 "Reserved words")

NOTE nnn: lexcially, the only issue is whether a token that matches an <SQL/JSON key word> is a <JSON path member name> or in fact a key word. This can be decided from the observation that a <JSON path member name> cannot be followed by a <left paren>.

**e) the following are additional punctuators: @ (modifies section 7.7 "Punctuators")**

**f) there are no *HexIntegerLiteral* (modifies 7.8.3 "Numeric literals")**

**f) there are no *RegularExpressionLiteral*. (modifies section 7.8.5 "Regualr expression literals")**

**NOTE nnn: SQL/JSON path language uses SQL regular expressions in the <JSON like_regex predicate>, not [ECMAScript] regular expressions.**

**g) There is no automatic semicolon insertion (modifies section 7.9 "Automatic semicolon insertion")**

**NOTE nnn: 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.**

**3) SQL/JSON grammar is stated with BNF non-terminals enclosed in angle brackets < >. The following corrrespondences between SQL/JSON BNF non-terminals and [ECMAScript] BNF non-terminals applies:**

| **SQL/JSON path language** | **[ECMAScript]** |
|---|---|
| **<JSON path literal>** | ***Literal*** |
| **<JSON path numeric literal>** | ***NumericLiteral*** |
| **<JSON path string literal>** | ***StringLiteral*** |
| **<JSON path identifier>** | ***Identifier*** |

**4) A <JSON path identifier> is classified as follows:**

**Case:**

**a) A <JSON path identifier> that is a <dollar sign> is a <JSON path context variable>.**

**a) A <JSON path identifier> that begins with a <dollar sign> is a <JSON path named variable>.**

**b) Otherwise, a <JSON path identifier> is a <JSON path key name>.**

## General Rules

**1) The value of a <JSON path literal> is determined as follows:**

**a) The value of a <JSON path numeric literal> *JPNL* is the value of the <signed numeric literal> whose characters are identical to *JPNL*.**

**b) The value of a <JSON path string literal> *JPSL* is an SQL character string whose character set is Unicode and whose characters are the ones enclosed by single or double quotation marks (but excluding these delimiters) in *JPSL* after replacing any escape sequences by their unescaped equivalents.**

**c) The value of `null` is the SQL/JSON null.**

**d) The value of `true` is *True*.**

**e) The value of `false` is *False*.**

**Conformance Rules**

**None**

## 6.20 New Subclause 9.n+5 SQL/JSON path language: syntax and semantics

1. ADD THE FOLLOWING SUBCLAUSE TO CLAUSE 9 "ADDITIONAL COMMON RULES"

**N.n SQL/JSON path language: syntax and semantics**

**Subclause Signature**

```
"SQL/JSON path language blah blah blah" [Syntax Rules] (
  Parameter: "PATH SPECIFICATION",
  Parameter: "PASSING CLAUSE"
)

"SQL/JSON path language blah blah blah" [General Rules] (
  Parameter: "PATH SPECIFICATION",
  Parameter: "CONTEXT ITEM",
  Parameter: "ALREADY PARSED",
  Parameter: "PASSING CLAUSE"
) Returns: "STATUS", "SQL/JSON SEQUENCE"
```

**Function**

**Specify the syntax and semantics of SQL/JSON path language.**

**Format**

**NOTE nnn: The grammar in this Subclause is not SQL grammar; it is the grammar of JSON path expressions passed in the parameter *PATH* in both the Syntax Rules and the General Rules. The grammar parallels (but does not duplicate) [ECMAScript] Section 11 "Expressions". The grammar is neither a strict subset nor a strict superset of [ECMAScript] expressions.**

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

<JSON path mode> ::=
       strict
     | lax

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

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

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

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

<JSON last subscript> ::=
     last

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

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

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

**NOTE nnn: Unlike [ECMAScript], SQL/JSON path language does not provide a member accessor using brackets that enclose a character string**

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

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

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

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

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

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

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

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

<JSON filter expression> ::=
     <question mark> <left paren> <JSON path predicate>
     <right paren>
```

**NOTE nnn: unlike [ECMAScript], predicates are not expressions; instead they form a separate language that can only be invoked within a <JSON filter expression>.**

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

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

<JSON unary expression> ::=
       <JSON accessor expression>
     | <plus sign> <JSON unary expression>
     | <minus sign> <JSON unary expression>

<JSON multiplicative expression> ::=
       <JSON unary expression>
     | <JSON multiplicative expression> <asterisk>
       <JSON unary expression>
     | <JSON multiplicative expression> <solidus>
       <JSON unary expression>
```

```
      | <JSON multiplicative expression> <percent>
        <JSON unary expression>

<JSON additive expression> ::=
        <JSON multiplicative expression>
      | <JSON additive expression> <plus sign>
        <JSON multiplicative expression>
      | <JSON additive expression> <minus sign>
        <JSON multiplicative expression>

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

**NOTE nnn: this concludes the main language for JSON path expressions. Next comes the language for predicates, used only in <JSON filter expression>.**

```
<JSON predicate primary> ::=
        <JSON delimited predicate>
      | <JSON non-delimited predicate>

<JSON delimited predicate> ::=
        <JSON exists path predicate>
      | <left paren> <JSON path predicate> <right paren>

<JSON non-delimited predicate> ::=
        <JSON comparison predicate>
      | <JSON like_regex predicate>
      | <JSON starts with predicate>
      | <JSON unknown predicate>

<JSON exists path predicate> ::=
      exists <left paren> <JSON path wff> <right paren>

<JSON comparison predicate> ::=
      <JSON path wff> <JSON comp op> <JSON path wff>
```

**NOTE nnn: comparison operators are not left associative, unlike [ECMAScript].**

```
<JSON comp op> ::=
        <double equals>
      | <not equals operator>
      | <less than operator>
      | <greater than operator>
      | <less than or equals operator>
      | <greater than or equals operator>
```

**NOTE nnn: equality operators have the same precedence as inequality comparision operators, unlike [ECMAScript].**

```
<JSON like_regex predicate> ::=
      <JSON path wff> like_regex
```

```
     <JSON like_regex pattern>
     [ flag <JSON like_regex flags> ]

<JSON like_regex pattern> ::= <JSON path string literal>

<JSON like_regex flag> ::= <JSON path string literal>

<JSON starts with predicate> ::=
     <JSON starts with whole> starts with
     <JSON starts with initial>

<JSON starts with whole> ::= <JSON path wff>

<JSON starts with initial> ::=
       <JSON path string literal>
     | <JSON path named variable>

<JSON unknown predicate> ::=
     <right paren> <JSON path predicte> <left paren>
     is unknown

<JSON boolean negation> ::=
       <JSON predicate primary>
     | <exclamation mark> <JSON delimited predicate>

<JSON boolean conjunction> ::=
       <JSON boolean negation>
     | <JSON boolean conjunction> <double ampersand>
       <JSON boolean negation>

<JSON boolean disjunction> ::=
       <JSON boolean conjunction>
     | <JSON boolean disjunction> <double vertical bar>
       <JSON boolean conjunction>

<JSON path predicate> ::=
     <JSON boolean disjunction>
```

**Syntax Rules**

**1) Let *P* be the *PATH SPECIFICATION* and let *PC* be the *PASSING CLAUSE* in an application of this Subclause.**

**2) Let *N* be the number of <JSON argument>s contained in *PC*, and let $JA_1$, . . ., $JA_N$ be those <JSON argument>s. For all *i*, 1 (one) <= *i* <= *N*, let $VE_i$ be the <value expression> simply contained in $JA_i$, and let $ID_i$ be the <identifier> immediately contained in *JA*.**

**3) *P* shall conform to <JSON path expression>, using the lexical rules specified in Subclause N.n “SQL/JSON path language: lexical elements”.**

**4) If *JPNV* is a <JSON path named variable> contained in *P,* then the <JSON path identifier> contained in *JPNV* shall be equivalent to $ID_i$ for some *i,* 1 <= *i* <= *N.***

**5) A <JSON path primary> that is an <at sign> shall be contained in a <JSON path predicate>.**

**6) A <JSON last subscript> shall be contained in a <JSON subscript>.**

**7.1) If <JSON like_regex predicate> *JLP* is specified, then the value of <JSON like_regex pattern> shall be an XQuery regular expression. If *JLP* simply contains <JSON like_regex flag> *JLF*, then the value of *JLF* shall be an XQuery option flag.**

**7.2) If <JSON starts with initial> is a <JSON path named variable> *JPNV,* then *JPNV* shall be equivalent to some $ID_i$ such that the declared type of $VE_i$ is a character string type and $JA_i$ does not have an implicit or explicit <JSON input clause>.**

**8’) *P* shall not be such that an implementation-defined syntactic analysis of *P* determines that the evaluation of *P* according to the General Rules of this Subclause would raise an exception condition.**

**NOTE nnn: For example, if *P* is 'strict $[last+1]', this will certainly raise an exception condition according to the General Rules. The preceding rule allows the implementation to treat this as a violation of the Syntax Rules.**

## Access Rules

None

## General Rules

**1) Let *P* be the *PATH SPECIFICATION*, *CI* be the *CONTEXT ITEM, PARSED* be the *ALREADY PARSED, PC* be the *PASSING CLAUSE* in an application of this Subclause. The result is a completion condition *ST* returned as *STATUS,* and an SQL/JSON sequence *SJS*, returned the *SQL/JSON SEQUENCE.***

**1.1) Let *ST* be the completion condition *successful completion.***

**2) Let *JT* be the value of the <string value expression> and let *FO* be the explicit or implicit <JSON input clause> contained in *CI.***

**3) IF *PARSED* is *False*, then it is implementation-defined whether the following rules are executed:**

a) The General Rules of Subclause N.n "Parsing a JSON text" are performed, with *JT* as the *JSON TEXT* , an implementation-defined <JSON predicate uniquess constraint> as the *UNIQUENESS CONSTRAINT* and *FO* as the *FORMAT OPTION*. Let *CISJI* be the *SQL/JSON ITEM* and let *ST* be the *STATUS* returned by this Subclause.

b) If *ST* is not *successful completion,* then *ST* is returned as the *STATUS* of this application of these General Rules, and no further General Rules are executed.

4) It is implementation-defined whether the following rules are executed:

a) Let *N* be the number of <JSON argument>s contained in *PC*, and let $JA_1$, . . ., $JA_N$ be those <JSON argument>s. For all *i*, 1 (one) <= *i* <= *N*:

If $JA_i$ contains an implicit or explicit <JSON input clause> $JAFO_i$, then let $VE_i$ be the value of the <value expression> contained in $JA_i$.

i) The General Rules of Subclause N.n "Parsing a JSON text" are performed, with $VE_i$ as the *JSON TEXT* , an implementation-defined <JSON predicate uniquess constraint> as the *UNIQUENESS CONSTRAINT* and $JAFO_i$ as the *FORMAT OPTION*. Let $VV_i$ be the *SQL/JSON ITEM* and let $ST_i$ be the *STATUS* returned by this Subclause.

ii) If $ST_i$ *is not successful completion,* then $ST_i$ is returned as the *STATUS* of these General Rules, and no further General Rules are executed.

5) Let *MODE* be the <JSON path mode> contained in *P*, and let *WFF* be the <JSON path wff> contained in *P*.

6) If *SEQ* is an SQL/JSON sequence, then let the function *Wrap*(*SEQ*) be defined as follows.

a) Let the input SQL/JSON sequence *SEQ* be the list of SQL/JSON items $I_1$, . . . , $I_n$ in order.

b) For each *j*, 1 (one) <= *j* <= *n*,

i) If $I_j$ is an SQL/JSON array, then let $I2_j$ be $I_j$

ii) Otherwise let $I2_j$ be an SQL/JSON array whose only element is $I_j$

c) The result of *Wrap*(*SEQ*) is the SQL/JSON sequence $I2_1$, . . . , $I2_n$ .

NOTE nnn: Thus *Wrap*() wraps any non-SQL/JSON array in an SQL/JSON array. This function is used in lax mode for certain operations that require an array as input.

**7) If *SEQ* is an SQL/JSON sequence, then let the function *Unwrap*(*SEQ*) be defined as follows.**

**a) Let the input SQL/JSON sequence *SEQ* be the list of SQL/JSON items $I_1, \dots, I_n$ in order.**

**b) For each *j*, 1 (one) <= *j* <= *n*,**

**i) If $I_j$ is an SQL/JSON array, then let the elements of $I_j$ be $E_1, \dots, E_m$ in order. Then let $I2_j$ be the SQL/JSON sequence $E_1, \dots, E_m$.**

**ii) Otherwise let $I2_j$ be $I_j$.**

**c) The result of *Unwrap*(*SEQ*) is the SQL/JSON sequence that is the concatenation of $I2_1, \dots, I2_n$ .**

**NOTE nnn: Thus *Unwrap*() unwraps any SQL/JSON array into its elements. This function is used in lax mode for certain operations that require a SQL/JSON sequence of non-SQL/JSON arrays as input.**

**8) The result of evualating *P* is determined recursively by the following definitions to evaluate all BNF non-terminals contained in *P*.**

**9) For any BNF production in the Format of this Subclause of the form**

```
<JSON BNF non-terminal 1> ::=
      <JSON BNF non-terminal 2>
```

**the result of evaluating <JSON BNF non-terminal 1> is the same as the result of evaluating <JSON BNF non-terminal 2>.**

**10) The result of evaluating a <JSON path wff> is a completion condition, and, if that completion condition is *successful completion*, then an SQL/JSON sequence. For conciseness the result will be stated either as an exception condition or as an SQL/JSON sequence (in the latter case, the completion condition *successful completion* is implicit). Unsuccessful completion conditions are not automatically raised and do not terminate evaluation of the General Rules in this Subclause.**

**a) If <JSON path context variable> *JPCV* is specified, then**

**Case:**

**i) If *PARSED* is *True*, then the result of evaluating *JPCV* is *JT*.**

**ii) Otherwise:**

**1) The General Rules of Subclause N.n "Parsing a JSON text" are performed, with *JT* as the *JSON TEXT* , an implementation-defined <JSON predicate uniquess constraint> as the *UNIQUENESS CONSTRAINT* and *FO* as the *FORMAT OPTION*. Let *CISJI* be the**

***SQL/JSON ITEM*** **and let** ***ST*** **be the** ***STATUS*** **returned by this Subclause.**

**2) Case:**

**A) If** ***ST*** **is not** ***successful completion,*** **then the result of evaluating** ***JPCV*** **is** ***ST.***

**B) Otherwise, the result of evaluating** ***JPCV*** **is** ***CISJI*** **.**

**b) If** ***JPNV*** **is a <JSON path named variable>, then let** ***JPI*** **be the <JSON path identifier> contained in** ***JPNV.*** **Let** ***JA*** **be the <JSON argument> contained in** ***PC*** **that contains an <identifier> that is equivalent to** ***JPI.*** **Let** ***VE*** **be the value of the <value expression> simply contained in** ***JA*****.**

**i) Case:**

**1) If** ***JA*** **contains an implicit or explicit <JSON input clause>** ***JAFO*** **then**

**Case:**

**A) If** ***VE*** **is the null value, then let** ***ST*** **be the completion condition** ***successful completion*** **and let** ***VV*** **be the empty SQL/JSON sequence.**

**B) Otherwise, the General Rules of Subclause N.n “Parsing a JSON text” are performed, with** ***VE*** **as the** ***JSON TEXT*** **, an implementation-defined <JSON predicate uniquess constraint> as the** ***UNIQUENESS CONSTRAINT*** **and** ***JAFO*** **as the** ***FORMAT OPTION.*** **Let** ***VV*** **be the** ***SQL/JSON ITEM*** **and let** ***ST*** **be the** ***STATUS*** **returned by this Subclause.**

**2) Otherwise:**

**A) If the declared type of** ***VE*** **is character string with character set Unicode, numric, boolean or datetime type, then let** ***ST*** **be the completion condition** ***successful completion*** **and let** ***VV*** **be** ***VE*****.**

**B) Otherwise, let** ***CDT*** **be an implementation-defined character string type with character set Unicode. Let** ***VV*** **be the result of**

```
CAST (VE AS CDT)
```

**Let** ***ST*** **be the completion code that results from this <cast specification>.**

**ii) Case:**

**1) If** ***ST*** **is not** ***successful completion,*** **then the result of** ***JPNV*** **is** ***ST.***

**2) If** ***VV*** **is a null value, then the result of** ***JPNV*** **is the SQL/JSON null.**

**3) Otherwise, the result of *JPNV* is *VV.***

**c) If <at sign> *AS* is specified, then let *JFE* be the innermost <JSON filter expression> containing *AS*. The value of *AS* is the current SQL/JSON item in the evaluation of *JFE.***

**d) If <JSON last subscript> *LAST* is specified, then let *JAA* be the innermost <JSON array accessor> that contains *LAST.* Let *JAE* be the <JSON accessor expression> that is the prefixed argument to *JAA*. Let *JAER* be the result of evaluating *JAE.* *JAER* is an SQL/JSON sequence of arrays (any errors are already handled by the General Rules relevant to *JAE*). Let *ARR* be the currrent SQL/JSON array in *JAER*. Let *N* be the number of elements in *ARR*. The result of *LAST* is *N*-1.**

**NOTE nnn: subscripts are 0-relative, therefore the last subscript of an array of size *N* is *N*-1.**

**e) The result of evaluating a <JSON path literal> is the value of the <JSON path literal>, as specified in Subclause N.n "SQL/JSON path language: lexical elements".**

**g) The result of evaluating <left paren> <JSON path wff> <right paren> is the result of evaluating the <JSON path wff>.**

**h) If a <JSON accessor expression> *JAE* that contains a <JSON accessor op> *JAOP* is specified, then let *BASE* be the result of the <JSON accessor expression> contained in *JAE.***

**Case:**

**i) If *BASE* is an exception condition, then *BASE* is the result of evaluating *JAE.***

**ii) Otherwise, let *ST* be the condition *successful completion.***

**Case:**

**1) If *JAOP* is <JSON member accessor>, then**

**A) Case:**

**I) If *JAOP* contains <JSON path identifier> *JPI*, then let *JPSL* be a Unicode character string whose value is the character string that composes *JPI.***

**II) Otherwise, let *JPSL* be the value of <JSON path string literal>.**

**B) Case:**

**I) If *MODE* is `lax`, then replace *BASE* by *Unwrap*(*BASE*).**

II) If *MODE* is `strict`, and if any SQL/JSON item in *BASE* is not an SQL/JSON object that contains a member whose key is *JPSL*, then let *ST* be the condition *data exception — SQL/JSON member not found*

III) If any SQL/JSON object in *BASE* has two SQL/JSON members with the same key, then it is implementation-defined whether *ST* is set to the condition *data exception — nonunique keys in a JSON object*

C) If *ST* is *successful completion*, then let $I_1, . . . , I_n$ be the list of SQL/JSON items in *BASE*. For all *j*, 1 (one) <= *j* <= *n*,

Case:

I) If $I_j$ is an SQL/JSON object containing a member *M* whose key is *JPSL*, then let $V_j$ be the bound value of *M*. If there is more than one member whose key is *JPSL*, then it is implementation-dependent which one is chosen.

II) Otherwise, let $V_j$ be the empty SQL/JSON sequence.

D) The result of *JAE* is

Case:

I) If *ST* is an exception condition, then *ST*

II) Otherwise, the SQL/JSON sequence that is the concatenation of $V_1, . . . , V_n$ in order.

2) If *JAOP* is <JSON wildcard member accessor>, then

A) Case:

I) If *MODE* is `lax`, then replace *BASE* by *Unwrap*(*BASE*).

II) If *MODE* is `strict`, and if any SQL/JSON item in *BASE* is not an SQL/JSON object that contains a member whose key is *JPSL*, then let *ST* be the condition *data exception — SQL/JSON member not found*

III) If any SQL/JSON object in *BASE* has two SQL/JSON members with the same key, then it is implementation-defined whether *ST* is set to the condition *data exception — nonunique keys in a JSON object*

B) If *ST* is *successful completion*, then let $I_1, . . . , I_n$ be the list of SQL/JSON items in *BASE*. For all *j*, 1 (one) <= *j* <= *n*,

Case:

**I) If $I_j$ is an SQL/JSON object, then:**

**1) Let $M_1, \ldots, M_m$ be the SQL/JSON members of $I_j$, enumerated in an implementation-depenent order. For all *i*, 1 (one) <= *i* <= *m*, let $K_i$ be the key and let $V_i$ be the bound value of $M_i$.**

**2) The following rule is performed an implementation-dependent number of times:**

**If any pair of keys $K_a$ and $K_b$ are equal, where $a \neq b$ , then one of $M_a$ or $M_b$ is deleted from the list of members of $I_j$.**

**3) Let $BV_1, \ldots, BV_m$ be the bound values of the remaining members of $I_j$ in an implementation-dependent order.. Let $V_j$ be the SQL/JSON sequence $BV_1, \ldots, BV_m$.**

**II) Otherwise, let $V_j$ be the empty SQL/JSON sequence.**

**C) The result of *JAE* is**

**Case:**

**I) If *ST* is an exception condition, then *ST***

**II) Otherwise, the SQL/JSON sequence that is the concatenation of $V_1, \ldots, V_n$ in order.**

**NOTE nnn: the order of members within an SQL/JSON object is implementation-dependent, but the order of $V_1, \ldots, V_n$ is not.**

**3) If *JAOP* is <JSON array accessor>, then**

**A) If *MODE* is `lax`, then replace *BASE* by *Wrap*(*BASE*).**

**B) If any SQL/JSON item in *BASE* is not an SQL/JSON array, then let *ST* be the condition *data exception — SQL/JSON array not found***

**C) Let *JSL* be the <JSON subscript list> simply contained in *JAOP.* Let $JS_1, \ldots, JS_s$ be the list of <JSON subscript>s simply contained in *JSL* For all *i, 1 (one)* <= *i* <= *s,***

**Case:**

**I) If $JS_i$ is <JSON path wff 1>, then let $JSFROM_i$ and $JSTO_i$ be $JS_i$.**

**II Otherwise, let $JSFROM_i$ be the <JSON path wff 2> contained in $JS_i$ and let $JSTO_i$ be the <JSON path wff 3> contained in $JS_i$.**

**D) If *ST* is *successful completion*, then let $I_1$, . . . , $I_n$ be the list of SQL/JSON arrays in *BASE*. For all *j*, 1(one) <= *j* <= *n*,**

**I) For all *i*, 1 (one) <= *i* <= *s*,**

**1) Let $RJSFROM_i$ be the result of evaluating $JSFROM_i$ and let $RJSTO_i$ be the result of evaluating $JSTO_i$ with $I_j$ as the current SQL/JSON array.**

**NOTE nnn: the current SQL/JSON array detemines the value of `last` (<JSON last subscript>).**

**2) If $TJSFROM_i$ is not a singleton numeric value or $TJSTO_i$ is not a singleton numeric value, then let *ST* be the exception condition *data exception — invalid SQL/JSON subscript***

**E) If *ST* is *successful completion*, then for all *j*, 1 (one) <= *j* <= *n*,**

**I) Let $N_j$ be the number of elements of $I_j$.**

**II) For all *i*, 1 (one) <= *i* <= *s*,**

**1) Let $TJSFROM_i$ be the result of implementation-defined truncation or rounding of $RJSFROM_i$ and let $TJSTO_i$ be the result of implemenation-defined truncation or rounding of $TJSTO_i$.**

**2) Any of the following are potential errors: $TJSFROM_i$ is negative, $TJSFROM_i$ is greater than or equal to $N_j$, $TJSTO_i$ is negative, $TJSTO_i$ is greater than or equal to $N_j$, or $TJSFROM_i$ is greater than $TJSTO_j$.**

**Case:**

**a) If there is a potential error and *MODE* is strict, then let *ST* be the condition *data exception — invalid SQL/JSON subscript***

**c) Otherwise, let $RJS_i$ be the set of all integers between $TJSFROM_i$ and $TJSTO_i$ inclusive, intersected with the set of integers from 0 (zero) to $N_j$ inclusive.**

**III) If *ST* is *successful completion*, then**

**1) let *RJSU* be the set union of all $RJS_i$, eliminating duplicates.**

**2) Let $E_1, \ldots, E_e$ be the elements of $I_j$ whose subscripts are in *RJSU*, using 0-relative subscripting, in increasing order of subscript.**

**3) Let $V_j$ be the SQL/JSON sequence $E_1, \ldots, E_e$ in order.**

**E) The result of *JAE* is**

**Case:**

**I) If *ST* is an exception condition, then *ST***

**II) Otherwise, the SQL/JSON sequence that is the concatenation of $V_1, \ldots, V_n$ in order.**

**4) If *JAOP* is <JSON wildcard array accessor>, then**

**A) If *MODE* is `lax`, then replace *BASE* by *Wrap*(*BASE*).**

**B) If any SQL/JSON item in *BASE* is not an SQL/JSON array, then let *ST* be the condition *data exception — SQL/JSON array not found***

**C) If *ST* is *successful completion*, then let $I_1, \ldots, I_n$ be the list of SQL/JSON items in *BASE*. For all *j*, 1 (one) <= *j* <= *n*,**

**I) Let $E_1, \ldots, E_t$ be the list of elements of $I_j$ in order.**

**II) Let $V_j$ be the SQL/JSON sequence $E_1, \ldots, E_t$ in order.**

**D) The result of *JAE* is**

**Case:**

**I) If *ST* is an exception condition, then *ST***

**II) Otherwise, the SQL/JSON sequence that is the concatenation of $V_1, \ldots, V_n$ in order.**

**5) If *JAOP* is <JSON filter expression>, then**

**A.0) If *MODE* is `lax`, then replace *BASE* by *Unwrap*(*BASE*).**

**A) Let *JPP* be the <JSON path predicate> simply contained in *JAOP*.**

**B) Let $I_1, \ldots, I_n$ be the SQL/JSON items in *BASE* in order. For all *j*, 1 (one) <= *j* <= *n:***

**I) Let *TV* be the result of evaluating *JPP* with $I_j$ as the current SQL/JSON item of *JPP*.**

**NOTE nnn: the current SQL/JSON item of *JPP* determines the value of <at sign> contained in *JPP*.**

**II) Case:**

**1) If *TV* is *True*, then let $V_j$ be $I_j$**

**2) Otherwise, let $V_j$ be the empty SQL/JSON sequence.**

**C) The result of *JAE* is the SQL/JSON sequence that is the concatenation of $V_1$, . . ., $V_n$ in order.**

**6) If *JAOP* is <JSON item method>, then let *JM* be the <JSON method>.**

**A) If *MODE* is `lax` and <JSON method> is not `type` or `size`, then replace *BASE* with *Unwrap*(*BASE*).**

**B) Let $I_1$, . . ., $I_n$ be the SQL/JSON items in *BASE* in order.**

**C) Case:**

**I) If *JM* specifies the <SQL/JSON key word> `type`, then**

**1) For all *j*, 1 (one) <= *j* <= *n*, let $V_j$ be**

**Case:**

**a) If $I_j$ is an SQL/JSON null, then the Unicode character string "null"**

**b) If $I_j$ is numeric, then the Unicode character string "number"**

**c) If $I_j$ is a character string, then the Unicode character string "string"**

**c.1) If $I_j$ is a boolean, then the Unicode character string "boolean"**

**d) If $I_j$ is a date, then the Unicode character string "date"**

**d.1) If $I_j$ is a time without time zone, then the Unicode character string "time without time zone"**

**d.2) If $I_j$ is a time with time zone, then the Unicode character string "time with time zone"**

**d.3) If $I_j$ is a timestamp without time zone, then the Unicode character string "timestamp without time zone"**

**d.4) If $I_j$ is a timestamp with time zone, then the Unicode character string "timestamp with time zone"**

**e) If $I_j$ is an SQL/JSON array, then the Unicode character string “array”**

**f) If $I_j$ is an SQL/JSON object, then the Unicode character string “object”**

**2) The result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$.**

**I.1) If *JM* specifies the <SQL/JSON key word> `size`, then**

**Case:**

**1) If *MODE* is `strict` and any $V_j$ is not an array, then the result of *JAOP* is the exception condition *data exception — SQL/JSON array not found***

**2) Otherwise,**

**a) For all *j*, 1 (one) <= *j* <= *n*, let $V_j$ be**

**Case:**

**i) If *Ij* is an SQL/JSON array, then the number of SQL/JSON elements of *Ij*.**

**ii) Otherwise, 1 (one).**

**b) The result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$.**

**II) If *JM* specifies the <SQL/JSON key word> `double`, then**

**1) For all *j*, 1 (one) <= *j* <= *n*,**

**Case:**

**a) If $I_j$ is not a number or character string, then let *ST* be *data exception — non-numeric SQL/JSON item.***

**b) Otherwise, let *X* be an SQL variable whose value is $I_j$. Let $V_j$ be the result**

```
CAST (X AS DOUBLE PRECISION)
```

**If this conversion results in an exception condition, then let *ST* be that exception condition.**

**2) Case:**

**a) If *ST* is not *successful completion,* then the result of *JAOP* is *ST.***

b) Otherwise, the result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$.

III) If *JM* specifies the <SQL/JSON key word> `ceiling`, then

1) For all *j*, 1 (one) <= *j* <= *n*,

Case:

a) If $I_j$ is not a number, then let *ST* be *data exception — non-numeric SQL/JSON item.*

c) Otherwise, let *X* be an SQL variable whose value is $I_j$. Let $V_j$ be the result

```
CEILING (X)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not *successful completion,* then the result of *JAOP* is *ST.*

b) Otherwise, the result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$.

IV) If *JM* specifies the <SQL/JSON key word> `floor`, then

1) For all *j*, 1 (one) <= *j* <= *n*,

Case:

a) If $I_j$ is not a number, then let *ST* be *data exception — non-numeric SQL/JSON item.*

c) Otherwise, let *X* be an SQL variable whose value is $I_j$. Let $V_j$ be the result

```
FLOOR (X)
```

If this conversion results in an exception condition, then let *ST* be that exception condition.

2) Case:

a) If *ST* is not *successful completion,* then the result of *JAOP* is *ST.*

b) Otherwise, the result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$.

**V) If *JM* specifies the <SQL/JSON key word> `abs`, then**

**1) For all *j*, 1 (one) <= *j* <= *n*,**

**Case:**

**a) If $I_j$ is not a number, then let *ST* be *data exception — non-numeric SQL/JSON item.***

**c) Otherwise, let *X* be an SQL variable whose value is $I_j$. Let $V_j$ be the result**

```
ABS (X)
```

**If this conversion results in an exception condition, then let *ST* be that exception condition.**

**2) Case:**

**a) If *ST* is not *successful completion*, then the result of *JAOP* is *ST*.**

**b) Otherwise, the result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$.**

**VI) If *JM* specifies the <SQL/JSON key word> `datetime`, then**

**1) For all *j*, 1 (one) <= *j* <= *n*,**

**Case:**

**a) If $I_j$ is not a character string, then let $ST_j$ be the exception condition *data exception — invalid argument for SQL/JSON datetime function***

**b) If the rules for <unquoted date string> in Subclause 5.3, "<literal>", can be applied to $I_j$ to determine a valid value of the data type DATE, then let $V_j$ be that value.**

**c) If the rules for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to $I_j$ to determine a valid value of the data type TIME WITHOUT TIME ZONE, then let $V_j$ be that value.**

**d) If the rules for <unquoted time string> in Subclause 5.3, "<literal>", can be applied to $I_j$ to determine a valid value of the data type TIME WITH TIME ZONE, then let $V_j$ be that value.**

e) If the rules for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to $I_j$ to determine a valid value of the data type TIMESTAMP WITHOUT TIME ZONE, then let $V_j$ be that value.

f) If the rules for <unquoted timestamp string> in Subclause 5.3, "<literal>", can be applied to $I_j$ to determine a valid value of the data type TIMESTAMP WITH TIME ZONE, then let $V_j$ be that value.

g) Otherwise, then let $ST_j$ be the exception condition *data exception — invalid argument for SQL/JSON datetime function*

2) Case:

a) If any $ST_j$ is not *successful completion,* then the result of *JAOP* is $ST_j$. If more than one $ST_j$ is an exception condition, it is implementation-dependent which one is raised.

b) Otherwise, the result of *JAOP* is the SQL/JSON sequence $V_1, \ldots, V_n$ .

VII) If *JM* specifies the <SQL/JSON key word> `keyvalue`, then

1) For all *j*, 1 (one) <= *j* <= *n*,

Case:

a) If $I_j$ is not an SQL/JSON object, then let *ST* be *data exception — SQL/JSON object not found.*

a.1) If $I_j$ has two SQL/JSON members with the same key, then it is implementation-defined whether *ST* is set to the condition *data exception — nonunique keys in a JSON object*

b) Otherwise:

i) Let $ID_j$ be an implementation-dependent exact numeric value of scale 0 (zero) that uniquely identifies $I_j$.

ii) Let $M_1, \ldots, M_m$ be the members of $I_j$ , enumerated in an implementation-dependent order. For all *i*, 1 (one) <= *i* <= *m*, let $K_i$ be the key of $M_i$ and let $BV_1, \ldots,$ $BV_m$ be the bound values of $M_i$.

**iii) The following rule is performed an implementation-dependent number of times:**

**If any pair of keys *$K_a$* and *$K_b$* are equal, where *$a \neq b$* , then one of *$M_a$* or *$M_b$* is deleted from the list of members of *$I_j$*.**

**iv) For each *$M_i$* in the remaining members of *$I_j$*, let *$OBJ_i$* be an SQL/JSON object with three members, one member with key "key" and with bound value *$K_i$* , another member with key "value" and with bound value *$BV_i$*, and another member with key "id" and with bound value *$ID_j$*.**

**v) Let *$V_j$* be the SQL/JSON sequence *$OBJ_1$, . . . , $OBJ_m$*.**

**2) Case:**

**a) If *ST* is not *successful completion,* then the result of *JAOP* is *ST.***

**b) Otherwise, the result of *JAOP* is the SQL/JSON sequence *$V_1$, . . ., $V_n$* .**

**i) If <JSON unary expression> *JUE* simply contains <plus sign> or <minus sign>, then let *BASE* be the result of evaluating the <JSON unary expression> simply contained in *JUE.* If *MODE* is `lax`, then *BASE* is replaced by *Unwrap*(*BASE*).**

**Case:**

**i) If *BASE* is an exception condition, then *BASE* is the result of evaluating *JUE.***

**ii) If any item in *BASE* is not a number, then the result of evaluating *JUE* is the exception condition *data exception — SQL/JSON number not found***

**iii) Otherwise, let *ST* be the condition *successful completion.***

**Case:**

**1) If *JUE* is <plus sign>, then the result of *JUE* is *BASE.***

**2) Otherwise, let *$I_1$, . . ., $I_n$* be the list of SQL/JSON items contained in *BASE.***

**A) For all *j*, 1 (one) <= *j* <= *n*, let *$X_j$* be an SQL variable whose value is *$I_j$*. Let *$V_j$* be the result of**

```
-Xj
```

**If an exception condition is raised when evaluating -*$X_j$*, then let *ST* be that exception condition.**

**B) If *ST* is an exception condition, then the result of *JUE* is *ST*. Otherwise, the result of *JUE* is the SQL/JSON sequence *$V_1$, . . ., $V_j$*.**

**j) If <JSON multiplicative expression> *JME* simply contains <asterisk>, <solidus> or <percent>, then**

**i) Let *OP1* be the result of the <JSON multiplicative expression> simply contained in *JME* and let *OP2* be the <JSON unary expression> simply contained in *JME*.**

**i.1) If *MODE* is lax, then *OP1* is replaced by *Unwrap*(*OP1*) and *OP2* is replaced by *Unwrap*(*OP2*) and**

**ii) Case:**

**1) If either *OP1* or *OP2* is not a singleton numeric value, then let *ST* be the exception condition *data exception — singleton SQL/JSON item required***

**2) Otherwise, let *X* be an SQL variable whose value is *OP1* and let *Y* be an SQL variable whose value is *OP2*.**

**Case:**

**A) If *JME* simply contains <asterisk>, then let *Z* be the result of the <numeric value expression>**

```
X * Y
```

**If this calculation raises an exception condition, then let *ST* be that exception condition.**

**B) If *JME* simply contains <solidus>, then let *Z* be the result of the <numeric value expression>**

```
X / Y
```

**If this calculation raises an exception condition, then let *ST* be that exception condition.**

**C) If *JME* simply contains <percent>, then let *Z* be the result of the <modulus expression>**

```
MOD (X , Y)
```

**If this calculation raises an exception condition, then let *ST* be that exception condition.**

**iii) Case:**

**1) If *ST* is an exception condition, then the result of *JME* is *ST***

**2) Otherwise, the result of *JME* is *Z***

**k) If <JSON additive expression> *JAE* simply contains <plus sign> or <minus sign>, then**

**i) Let *OP1* be the result of the <JSON additive expression> simply contained in *JME* and let *OP2* be the <JSON multiplicative expression> simply contained in *JME*.**

**ii) Case:**

**1) If either *OP1* or *OP2* is not a singleton numeric value, then let *ST* be the exception condition *data exception — singleton SQL/JSON item required.***

**2) Otherwise, let *X* be an SQL variable whose value is *OP1* and let *Y* be an SQL variable whose value is *OP2*.**

**Case:**

**A) If *JME* simply contains <plus sign>, then let *Z* be the result of the <numeric value expression>**

```
X + Y
```

**If this calculation raises an exception condition, then let *ST* be that exception condition.**

**B) If *JME* simply contains <minus sign>, then let *Z* be the result of the <numeric value expression>**

```
X - Y
```

**If this calculation raises an exception condition, then let *ST* be that exception condition.**

**iii) Case:**

**1) If *ST* is an exception condition, then the result of *JAE* is *ST***

**2) Otherwise, the result of *JAE* is *Z***

**11) The result of evaluating a <JSON path predicate> is a truth value *True*, *False* or *Unknown*.**

**a) If <JSON exists path predicate> *JEP* is specified, then**

**i) Let *WFF* be the result of evaluating the <JSON path wff> simply contained in *JEP*.**

**ii) Case:**

**1) If *WFF* is an exception condition, then the result of *JEP* is *Unknown*.**

**2) If *WFF* is an empty SQL/JSON sequence, then the result of *JEP* is *False*.**

**3) Otherwise, the result of *JEP* is *True*.**

**b) If <JSON delimited predicate> *JDP* specifies <left paren> <JSON path predicate> <right paren>, then the result of evaluation *JDP* is the result of evaluating the <JSON path predicate> simply contained in *JDP*.**

**c) If <JSON comparison predicate> *JCP* is specified, then**

**i) Let *JCO* be the <JSON comp op> simply contained in *JCP*.**

**ii) Let *A* be the result of evaluating the first <JSON path wff> contained in *JCP*, and let *B* be the result of evaluating the second <JSON path wff> contained in *JCP*.**

**iii) Case:**

**1) If either *A* or *B* is an exception condition, then the result of *JCP* is *Unknown*.**

**2) Otherwise,**

**A) If *MODE* is `lax`, then replace *A* with *Unwrap*(*A*) and replace *B* with *Unwrap*(*B*).**

**B) Let *A* be the SQL/JSON sequence $AI_1, \ldots, AI_n$ and let *B* be the SQL/JSON sequence $BI_1, \ldots, BI_m$.**

**C) Let *ERR* be initially *False* and let *FOUND* be initially *False*.**

**D) For all *i*, 1(one) <= *i* <= *n*, and all *j*, 1 (one) <= *j* <= *m*, $AI_i$ is compared to $BI_j$ using *JCO*, according to the following rules. When comparing character strings, the codepoint collation of Unicode is used.**

**Case:**

**I) If $AI_i$ and $BI_j$ are not comparable, then let *ERR* be *True*.**

**NOTE nnn: otherwise, $AI_i$ and $BI_j$ are either the SQL/JSON null or comparable SQL/JSON scalars.**

**II) If *JCO* is <double equals> then**

**Case**

**1) If $AI_i$ and $BI_j$ are both the SQL/JSON null value, then let *FOUND* be *True*.**

**2) If $AI_i$ and $BI_j$ are equal SQL/JSON scalars, then let *FOUND* be *True*.**

**III) If *JCO* is <not equals operator>, then**

**Case:**

**1) If $AI_i$ is the SQL/JSON null value, and $BI_j$ is not the SQL/JSON null then let *FOUND* be *True*.**

**2) If $AI_i$ is not the SQL/JSON null value, and $BI_j$ is the SQL/JSON null then let *FOUND* be *True*.**

**3) If $AI_i$ and $BI_j$ are unequal SQL/JSON scalars, then let *FOUND* be *True*.**

**IV) If *JCO* is <less than operator>, then if $AI_i$ and $BI_j$ are SQL/JSON scalars and $AI_i$ is less than $BI_j$, then let *FOUND* be *True*.**

**V) If *JCO* is <greater then operator>, then if $AI_i$ and $BI_j$ are SQL/JSON scalars and $AI_i$ is greater than $BI_j$, then let *FOUND* be *True*.**

**VI) If *JCO* is <less than or equals operator> then**

**Case:**

**1) If $AI_i$ and $BI_j$ are both the SQL/JSON null value, then let *FOUND* be *True*.**

**2) If $AI_i$ and $BI_j$ are SQL/JSON scalars and $AI_i$ is less than or equal to $BI_j$, then let *FOUND* be *True*.**

**VII) If *JCO* is <greater than or equals operator> then**

**Case:**

**1) If $AI_i$ and $BI_j$ are both the SQL/JSON null value, then let *FOUND* be *True*.**

**2) If $AI_i$ and $BI_j$ are SQL/JSON scalars and $AI_i$ is greater than or equal to $BI_j$, then let *FOUND* be *True*.**

**D) Case:**

**I) If *MODE* is `strict` and *ERR* is *True*, then the result of *JCP* is *Unknown***

**II) If *ERR* is *True* and *FOUND* is *True*, then it is implementation-dependent whether the result of *JCP* is *True* or *Unknown*.**

**NOTE nnn: this means that in lax mode the implementation can stop on the first error or the first success, in an implementation-dependent order of evaluation.**

**III) If *FOUND* is *True*, then the result of *JCP* is *True*.**

**IV) If *ERR* is *True*, then the result of *JCP* is *Unknown***

**V) Otherwise, the result of *JCP* is *False*.**

**d) If <JSON like_regex predicate> *JLP* is specified, then**

**i) Let *SEQ* be the result of evaluating the <JSON path wff> simply contained in *JLP*.**

**ii) Case:**

**1) If *SEQ* is an exception condition, then the result of *JLP* is *Unknown*.**

**2) Otherwise:**

**A) If *MODE* is lax, then replace *SEQ* with *Unwrap*(*SEQ*).**

**B) Let *PATTERN* be a <character string literal> whose value is the value of the <JSON like_regex pattern> simply contained in *JLP*.**

**C) If <JSON like_regex flags> *JLF* is specified, then let *FLAGS* be a <character string literal> whose value is the value of *JLF;* otherwise let *FLAGS* be a zero-length character string.**

**D) Let $I_1, \ldots, I_n$ be the SQL/JSON items in *SEQ*.**

**E) Let *ERR* be initially *False* and let *FOUND* be initially *False*.**

**F) For all *j*, 1 (one) <= *j* <= *n***

**Case:**

**I) If $I_j$ is not a character string, then let *ERR* be *True*.**

**II) Otherwise, let $X_j$ be an SQL variable whose value is $I_j$. Let $TV_j$ be the result of evaluating**

```
Xj LIKE_REGEX PATTERN FLAG FLAGS
```

**If $TV_j$ is *True*, then let *FOUND* be *True*.**

**NOTE nnn: The LIKE_REGEX predicate cannot raise an exception because the Syntax Rules require that *PATTERN* is**

**an XQUERY regular expression and *FLAGS* is an XQuery option flag.**

**G) Case:**

**I) If *MODE* is `strict` and *ERR* is *True*, then the result of *JLP* is *Unknown***

**II) If *ERR* is *True* and *FOUND* is *True*, then it is implementation-dependent whether the result of *JLP* is *True* or *Unknown*.**

**III) If *FOUND* is *True*, then the result of *JLP* is *True*.**

**IV) If *ERR* is *True*, then the result of *JLP* is *Unknown***

**V) Otherwise, the result of *JLP* is *False*.**

**e) If <JSON starts with predicate> *JSWP* is specified, then**

**i) Let *A* be the result of evaluating the <JSON starts with whole> simply contained in *JSWP*, and let *B* be the result of evaluating the <JSON starts with initial> simply contained in *JSWP*.**

**ii) Case:**

**1) If *A* or *B* is an exception condition, then the result of *JSWP* is *Unknown*.**

**1.1) If *B* is not a character string, then the result of *JSWP* is *Unknown*.**

**2) Otherwise:**

**A) If *MODE* is lax, then replace *A* with *Unwrap*(*A*).**

**A) Let $AI_1, \ldots, AI_n$ be the SQL/JSON items in *A*.**

**B) Let *ERR* be initially *False* and let *FOUND* be initially *False*.**

**C) For all *i*, 1 (one) <= *i* <= *n***

**Case:**

**I) If $AI_j$ is not a character string, then let *ERR* be *True*.**

**II) If *B* is an initial substring of $AI_i$, then let *FOUND* be *True*.**

**D) Case:**

**I) If *MODE* is `strict` and *ERR* is *True*, then the result of *JSWP* is *Unknown***

**II) If *ERR* is *True* and *FOUND* is *True*, then it is implementation-dependent whether the result of *JSWP* is *True* or *Unknown*.**

**III) If *FOUND* is *True*, then the result of *JSWP* is *True*.**

**IV) If *ERR* is *True*, then the result of *JSWP* is *Unknown***

**V) Otherwise, the result of *JSWP* is *False*.**

**f) If <JSON unknown predicate> *JUP* is specified, then**

**i) Let *JPP* be the result of evaluating the <JSON path predicate> simply contained in *JUP*.**

**ii) Case:**

**1) If *JPP* is *Unknown*, then the result of *JUP* is *True*.**

**2) Otherwise, the result of *JUP* is *False*.**

**g) If <JSON boolean negative> *JBN* simply contains <exclamation mark>, then**

**i) Let *JDP* be the result of evaluating the <JSON delimited predicate> simply contained in *JBN*.**

**ii) The result of evaluating *JBN* is**

**Case:**

**1) If *JDP* is *True*, then *False*.**

**2) If *JDP* is *Unknown*, then *Unknown*.**

**1) If *JDP* is *False*, then *True*.**

**h) If <JSON boolean conjunction> *JBC* simply contains <double ampersand> then**

**i) let *OP1* be the result of evaluating the <JSON boolean conjunction> simply contained in *JBC*, and let *OP2* be the result of evaluating the <JSON boolean negation> simply contained in *JBC*.**

**ii) The result of *JBC* is**

**Case:**

**1) If either *OP1* or *OP2* is *False*, then *False*.**

**3) If either *OP1* or *OP2* is *Unknown*, then *Unknown*.**

**3) Otherwise, *True***

**i) If <JSON boolean disjunction> *JBD* simply contains <double vertical bar> then**

**i) let *OP1* be the result of evaluating the <JSON boolean disjunction> simply contained in *JBD*, and let *OP2* be the result of evaluating the <JSON boolean conjunction> simply contained in *JBD*.**

**ii) The result of *JBC* is**

**Case:**

**1) If either *OP1* or *OP2* is *True*, then *True*.**

**3) If either *OP1* or *OP2* is *Unknown*, then *Unknown*.**

**3) Otherwise, *False***

**Conformance Rules**

2. THE EDITOR SHALL CHOOSE A BLOCK OF CONFORMANCE FEATURES FOR THE SQL/JSON PATH LANGUAGE, HERE DENOTED TX3N:

**1) Without Feature Tx31, "SQL/JSON path language: strict mode", a <JSON path expression> shall not contain a <JSON path mode> that is `strict`.**

**2) Without Feature Tx32, "SQL/JSON path language: item methods", a <JSON path expression> shall not contain a <JSON item method>.**

**3) Without Feature Tx33 "SQL/JSON path language: multiple subscripts", a <JSON path expression> shall not contain a <JSON subscript list> that contains more than one <JSON subscript>.**

**4) Without Feature Tx34, "SQL/JSON path language: wildcard member accessor", a <JSON path expression> shall not contain a <JSON wildcard member accessor>.**

**5) Without Feature Tx35, "SQL/JSON path language: filter expressions", a <JSON path expression> shall not contain a <JSON filter expression>.**

**6) Without Feature Tx36, "SQL/JSON path language: starts with predicate", a <JSON path expression> shall not contain a <JSON starts with predicate>.**

**7) Without Feature Tx37, "SQL/JSON path language: regex_like predicate", a <JSON path expression> shall not contain a <JSON regex_like predicate>.**

## 6.21 New Subclause 9.n+6, Casting an SQL/JSON sequence to an SQL type

1. ADD THE FOLLOWING SUBCLAUSE TO CLAUSE 9 "ADDITIONAL COMMON RULES"

**N.n Casting an SQL/JSON sequence to an SQL type**

**Subclause Signature**

```
"Casting an SQL/JSON sequence to an SQL type"
[General Rules] (
  Parameter: "STATUS IN"
  Parameter: "SQL/JSON SEQUENCE",
  Parameter: "EMPTY BEHAVIOR",
  Parameter: "ERROR BEHAVIOR",
  Parameter: "DATA TYPE"
) Returns: "STATUS OUT", "VALUE"
```

**Function**

**Cast an SQL/JSON sequence to a <predefined type>.**

**Syntax Rules**

**None**

**Acces Rules**

**None**

**General Rules**

**1) Let *INST* be the *STATUS IN*, let *SEQ* be the *SQL/JSON SEQUENCE,* let *ONEMPTY* be the *EMPTY BEHAVIOR,* let *ONERROR* be the *ERROR BEHAVIOR* and let *DT* be the *DATA TYPE* in an application of this Subclause. The result of this Subclause is a completion condition *OUTST* returned as *STATUS OUT* and a value *V* returned as *VALUE.***

**2) The General Rules of this Subclause are not terminated if an exception condition is raised.**

**3) Let *TEMPST* be *INST.***

**4) If *TEMPST* is *successful completion,* then:**

**Case:**

**a) If the length of *SEQ* is more than 1 (one), then let *TEMPST* be *data exception — more than one SQL/JSON item.***

**b) If the length of *SEQ* is 1 (one), then let *I* be the SQL/JSON item in *SEQ.***

**Case:**

**i) If *I* is an SQL/JSON array or SQL/JSON object, then let *TEMPST* be *data exception — SQL/JSON scalar required.***

**ii) If *I* is the SQL/JSON null, then let *V* be the null value of type *DT.***

**iii) Otherwise, let *IDT* be the data type of *I*.**

**Case:**

**1) If *IDT* cannot be cast to target type *DT* according to the Syntax Rules of Subclause 6.13 <cast specification>, then let *TEMPST* be *data exception — SQL/JSON item cannot be cast to target type***

**2) Otherwise, let *X* be an SQL variable whose value is *I*. Let *V* be the value of**

```
CAST (X AS DT)
```

**If an exception condition is raised by this <cast specification>, then let *TEMPST* be that exception condition.**

**c) If the length of *SEQ* is 0 (zero), then**

**Case:**

**i) If *ONEMPTY* is ERROR, then let *TEMPST* be *data exception — no SQL/JSON item.***

**ii) If *ONEMPTY* is NULL, then let *V* be the null value of type *DT.***

**iii) If *ONEMPTY* immediately contains DEFAULT, then let *VE* be the <value expression> immediately contained in *ONEMPTY.* Let *V* be the value of**

```
CAST (VE AS DT)
```

**If an exception condition is raised by this <cast specification>, then let *TEMPST* be that exception condition.**

**5) Case:**

**a) If *TEMPST* is *successful completion*, then let *OUTST* be *successful completion.***

**b) If *ONERROR* is ERROR, then let *OUTST* be *TEMPST.***

**c) If *ONERROR* is NULL, then let *V* be the null value of type *DT* and let *OUTST* be *successful completion.***

**d) If *ONERROR* immediately contains DEFAULT, then let *VE* be the <value expression> immediately contained in *ONERROR.* Let *V* be the value of**

```
CAST (VE AS DT)
```

**Case:**

**i) If an exception condition is raised by this <cast specification>, then let *OUTST* be that exception condition.**

**ii) Otherwise, let *OUTST* be *successful completion.***

**6) *OUTST* is the *STATUS OUT* and *V* is the *VALUE* that are returned by this Subclause.**

## Conformance Rules

**Zero (0)**

## 6.22 New Subclause 10.n+1, <JSON value expression>

*[NOTE to the proposal reader: this section duplicates material in [SQL/JSON part 1]; it is repeated here because it is integral to this proposal as well.]*

1. Add a Subclause to Clause 10 "Additional common elements" as follows:

**<JSON value expression>**

**Function**

**Specify a value to be used as input by an SQL/JSON function.**

**Format**

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

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

<JSON input representation> ::=
       JSON
     | <implementation-defined JSON representation option>
```

**Syntax Rules**

**1) FORMAT JSON specifies the data format specified in [RFC4627].**

**2) FORMAT <implementation-defined JSON representation option> specifies an implementation-defined data format.**

**NOTE nnn: For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON format> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of Subclause N.3 "Parsing a JSON text" and Subclause N.4 "Serializing an SQL/JSON item", respectively.**

**3) The declared type *DT* of the <value expression> *VE* simply contained in <JSON value expression> *JVE* either shall be a character string type, a binary string type, a numeric type, a datetime type, or Boolean, or shall be a <data type> the values of which can be cast to a character string type according to the Syntax Rules of Subclause 6.13, "<cast specification>".**

**4) If *VE* is a JSON-returning function *JRF*, and <JSON input clause> is not specified, then**

**Case:**

**a) If the explicit or implicit <JSON output clause> simply contained in *JRF* contains FORMAT JSON, then the implicit <JSON input clause> of *JVE* is FORMAT JSON.**

**b) Otherwise, the implicit <JSON input clause> of *JVE* is implementation-defined.**

**5) If an explicit or implicit <JSON input clause> is specified, then *DT* shall be a string type.**

**6) If *DT* is a binary string type, then an explicit or implicit <JSON input clause> shall be specified.**

**Access Rules**

**None**

**General Rules**

**None**

**Conformance Rules**

**None**

## 6.23 New Subclause 10.n+1a, <JSON output clause>

*[NOTE to the proposal reader: this section duplicates material in [SQL/JSON part 1]; it is repeated here because it is integral to this proposal as well.]*

1. APPEND TO CLAUSE 10 “ADDITIONAL COMMON ELEMENTS” THE FOLLOWING:

**<JSON output clause>**

**Function**

**Specify the data type, format, and encoding of the JSON text created by a JSON-returning function.**

**Format**

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

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

**Syntax Rules**

**0.1) If FORMAT is not specified, then FORMAT JSON is implicit.**

**1) If the <JSON output clause> specifies or implies JSON, then the <data type> shall identify a string type *ST*.**

**Case:**

**a) If *ST* identifies a character string type, then *ST* shall have a Universal Character Set, ENCODING shall not be specified, and an implicit choice of UTF8, UTF16 or UTF16 is determined by the character encoding form of *ST* (i.e., the keywords UTF8, UTF16 and UTF32 denote the UTF8, UTF16 and UTF32 character encoding forms, respectively).**

**b) If *ST* is a binary string type and ENCODING is not specified, then it is implementation-defined whether UTF8, UTF16 or UTF32 is implicit.**

**4) FORMAT JSON specifies the data format specified in [RFC4627].**

**5) FORMAT <implementation-defined JSON representation option> specifies an implementation-defined data format.**

**NOTE nnn: For example, BSON or AVRO; see Bibliography. An <implementation-defined JSON representation option> implies an ability to parse a string into the SQL/JSON data model, and an ability to serialize an SQL/JSON array or SQL/JSON object to a string, similar to the capabilities of Subclause 9.x, “Parsing a JSON text”, and Subclause 9.y, “Serializing an SQL/JSON item”, respectively.**

**Access Rules**

**None.**

**General Rules**

**None.**

**Conformance Rules**

**None.**

## 6.24 New Subclause 10.n+2, <JSON API common syntax>

1. Append to Clause 10 "Additional common elements" the following:

**<JSON API common syntax>**

**Sublause Signature**

```
"<JSON API common syntax>" [General Rules] (
  Parameter: JSON API COMMON SYNTAX
) Returns STATUS, SQL/JSON SEQUENCE
```

**Function**

**Define the inputs to JSON_VALUE, JSON_QUERY, JSON_TABLE and JSON_EXISTS**

**Format**

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

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

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

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

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

**Syntax Rules**

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

**2) Case:**

**a) If <JSON API common syntax> is not contained in <JSON table>, then <JSON table path name> shall not be specified.**

**b) Otherwise, if <JSON table path name> is not specified, then an implementation-dependent <JSON table path name> is implicit.**

**4) Let *P* be the <JSON path specification>.**

**5) If the <JSON API common syntax> simply contains a <JSON passing clause>, then let *PC* be that <JSON passing clause>; otherwise, let *PC* be the empty string.**

**6) The Syntax Rules of Subclause N.n "SQL/JSON path language: syntax and semantics" are applied, with *P* as the *PATH SPECIFICATION* and *PC* as the *PASSING CLAUSE.***

## Access Rules

**None**

## General Rules

**1) Let *JACS* be the *JSON API COMMON SYNTAX* in an application of this Subclause. *JACS* is a <JSON API common syntax>.**

**2) Let *P* be the <JSON path specification> simply contained in *JACS*, and let *C* be the <JSON context item> simply contained in *JACS*. If *JACS* simply contains a <JSON passing clause, then let *PC* be that <JSON passing clause>; otherwise, let *PC* be the empty string.**

**3) The General Rules of Subclause N.n "SQL/JSON path language: syntax and semantics" are applied, with *P* as the *PATH SPECIFICATION, C* as the *CONTEXT ITEM, <u>False</u>* as the *ALREADY PARSED,* and *PC* as the *PASSING CLAUSE.* Let *SEQ* be the *SQL/JSON SEQUENCE* and let *ST* be the *STATUS* that are returned from the Subclause.**

**4) *SEQ* is the *SQL/JSON SEQUENCE* and *ST* is the *STATUS* that are returned by this Subclause.**

## Conformance Rules

**1) Without Feature Tx23, "SQL/JSON: PASSING clause", <JSON API common syntax> shall not contain <JSON passing clause>"**

## 6.25 Changes to 24.1, SQLSTATE

1. ADD THE FOLLOWING CONDITIONS TO TABLE 33 “SQLSTATE CLASS AND SUBCLASS VALUES”

| Category | Condition | Class | Subcondition | Subclass |
|---|---|---|---|---|
| X | data exception | 22 | **invalid JSON text** | [assigned by editor] |
| | | | **no SQL/JSON item** | |
| | | | **more than one SQL/JSON item** | |
| | | | **non-numeric SQL/JSON item** | |
| | | | **SQL/JSON member not found** | |
| | | | **SQL/JSON array not found** | |
| | | | **invalid SQL/JSON subscript** | |
| | | | **invalid argument for SQL/JSON datetime function** | |
| | | | **SQL/JSON object not found** | |
| | | | **SQL/JSON number not found** | |
| | | | **singleton SQL/JSON item required** | |
| | | | **nonunique keys in a JSON object** | |

## 6.26 Changes to 25.3, Implied feature relationships of SQL/Foundation

1. ADD THE FOLLOWING ROWS TO TABLE 35 “IMPLIED FEATURE RELATIONSHIPS OF SQL/FOUNDATION”:

| Feature ID | Feature Name | Implied Feature ID | Implied Feature Name |
|---|---|---|---|
| **Tx22** | **SQL/JSON: IS JSON WITH UNIQUE KEYS predicate** | **Tx21** | **Basic SQL/JSON** |
| **Tx23** | **SQL/JSON: PASSING clause** | **Tx21** | **Basic SQL/JSON** |
| **Tx24** | **JSON_TABLE: PLAN clause** | **Tx21** | **Basic SQL/JSON** |

| Feature ID | Feature Name | Implied Feature ID | Implied Feature Name |
|---|---|---|---|
| **Tx25** | **SQL/JSON: ON EMPTY and ON ERROR clauses** | **Tx21** | **Basic SQL/JSON** |
| **Tx27** | **JSON_TABLE: sibling NESTED COLUMNS clauses** | **Tx21** | **Basic SQL/JSON** |
| **Tx26** | **SQL/JSON: General <value expression> in ON EMPTY or ON ERROR clauses** | **Tx25** | **SQL/JSON: ON EMPTY and ON ERROR clauses** |
| **Tx36** | **SQL/JSON path language: starts with predicate** | **Tx35** | **SQL/JSON path language: filter expression** |
| **Tx37** | **SQL/JSON path language: regex_like predicate** | **Tx35** | **SQL/JSON path language: filter expression** |

## 6.27 New Clause or Annex, Bibliography

1. ADD THE FOLLOWING CLAUSE OR ANNEX (EDITOR'S DECISION)

**Bibliography**

**The following are non-normative documents that are relevant to this standard.**

**[Avro]** http://avro.apache.org/

**[BSON]** http://bsonspec.org/

## 6.28 Changes to Annex A, SQL conformance summary

Automatically generated

## 6.29 Changes to Annex B, Implementation-defined elements

1. ADD THE FOLLOWING ITEMS TO THIS ANNEX:

**n) Subclause 6.n, <JSON value function>**

**a) If <JSON returning clause> is not specified, then an implementation-defined character string type is implicit.**

**n) Subclause 6.n, <JSON query>**

**a) If <JSON output clause> is not specified, then RETURNING *SDT* FORMAT JSON is implicit, where *SDT* is an implementation-defined string type.**

**n) Subclause 7.n, <JSON table>**

**a) The declared type of an ORDINALITY column is an implementation-defined numeric type with scale 0 (zero).**

**n) Subclause 9.n, "Parsing a JSON text"**

**a) Parsing an implementation-defined format is implementation-defined.**

**n) Subclause 9.n, "Serializing an SQL/JSON item"**

**a) The permissible target types when serializing to an implementation-defined format is implementation-defined.**

**b) The result of serializing an SQL/JSON item is implementation-dependent**

**n) Subclause 9.n, "SQL/JSON path language: syntax and semantics"**

**a) It is implementation-defined whether syntactic analysis (in the Syntax Rules) may be used to detect errors that would necessarily occur in General Rules.**

**b) It is implementation-defined whether the context item or a named variable whose format is JSON is parsed even if the context item or named variable is never referenced in the SQL/JSON path expression.**

**c) When parsing a context item or named variable whose format is JSON, it is implementation-defined whether an exception is raised if the input contains a JSON member with duplicate keys.**

**d) It is implementation-defined whether a <JSON member accessor>, a <JSON wildcard member accessor>, or a `keyalue` <JSON item method> raises an exception if its argument is an SQL/JSON object with duplicate keys.**

**e) <JSON subscript>s use implementation-defined truncation or rounding to an exact numeric value of scale 0.**

**n) Subclause 10.n, "<JSON value expression>"**

**a) The syntax and semantics of an <implementation-defined JSON format> are— you guesed it! — implementation-defined.**

**b) The implicit <JSON input clause> to be used when a JSON-returning function returns a format other than JSON is implementation-defined.**

**n) Subclause 10.n, "<JSON output clause>"**

**a) The syntax and semantics of an <implementation-defined JSON representation option> is implementation-defined.**

**b) If <JSON output clause> specifies JSON for a binary string but does not specify ENCODING, then the encoding is implementation-defined.**

## 6.30 Changes to Annex C, Implementation-dependent elements

1. ADD THE FOLLOWING ITEMS TO THIS ANNEX:

**n) Subclause 7.n, <JSON table>**

**a) Missing <JSON table path name>s are implementation-dependent.**

**b) The syntactic transformation of <JSON table> into <JSON table primitive> involves a number of implementation-dependencies which are not significant to the final result.**

**n) Subclause 9.n, "Parsing a JSON text"**

**a) Removal of JSON members having duplicate keys is implementation-dependent.**

**n) Subclause 9.n, "Serializing an SQL/JSON item"**

**b) The result of serializing an SQL/JSON item is implementation-dependent**

**n) Subclause 9.n, "SQL/JSON path language: syntax and semantics"**

**a) If a <JSON member accessor> has an argument that is an SQL/JSON object with duplicate keys, then it is implementation-dependent which bound value is chosen by the <JSON member accessor>.**

**b) If a <JSON wildcard member accessor> or a `keyvalue` <JSON item method> has an argument that is an SQL/JSON object with duplicate keys, then it is implementation-dependent which SQL/JSON members are chosen by the operator.**

**c) The order of bond values in the result of a <JSON wildcard member accessor> is implementation-dependent.**

**d) If more than one exception is encountered during the evaluation of the `datetime` <JSON item method>, it is implementation-dependent which one is raised.**

**e) The bound value of an SQL/JSON member whose key is "id" in the result of a `keyvalue` <JSON item method> is implementation-dependent exact numeric with scale 0.**

**f) In lax mode, if a <JSON comparison predicate>, <JSON like_regex predicate> or <JSON starts with predicate> performs one test that is an error and another test that is a success, it is implementation-dependent whether the result is *Unknown* or *True*.**

**n) Subclause 10.n, <JSON API common syntax>**

**a) Missing <JSON table path name>s are implementation-dependent.**

## 6.31 Changes to Annex F, SQL feature taxonomy

1. ADD THE FOLLOWING ROWS TO TABLE 39, "FEATURE TAXONOMY FOR OPTIONAL FEATURES"

| Feature ID | Feature Name |
|---|---|
| **Tx21** | **Basic SQL/JSON** |
| **Tx22** | **SQL/JSON: IS JSON WITH UNIQUE KEYS predicate** |
| **Tx23** | **SQL/JSON: PASSING clause** |
| **Tx24** | **JSON-TABLE: PLAN clause** |
| **Tx25** | **SQL/JSON: ON EMPTY and ON ERROR clauses** |
| **Tx26** | **General <value expression> in ON ERROR or ON EMPTY clauses** |
| **Tx27** | **JSON_TABLE: sibling NESTED COLUMNS clauses** |
| **Tx28** | **JSON_QUERY** |
| **Tx29** | **JSON_QUERY: array wrapper options** |
| **Tx31** | **SQL/JSON path language: strict mode** |
| **Tx32** | **SQL/JSON path language: item method** |
| **Tx33** | **SQL/JSON path language: multiple subscripts** |
| **Tx34** | **SQL/JSON path language: wildcard member accesso** |
| **Tx35** | **SQL/JSON path language: filter expressions** |
| **Tx36** | **SQL/JSON path language: starts with predicate** |
| **Tx37** | **SQL/JSON path language: regex_like predicate** |

# 7. Comment addressed

1. MARK COMMENT #257, P02-USA-950, AS PARTIALLY ADDRESSED.

# 8. Checklist

| | |
|---|---|
| Concepts | done |
| Access Rules | none |
| Conformance Rules | done |
| Lists of SQL-statements by category | no new statements |
| Table of identifiers used by diagnostics statements | |
| Collation derivation for character strings | |
| Closing Possible Problems | one comment partially addressed |
| Any new Possible Problems clearly identified | none |
| Reserved and non-reserved keywords | |
| SQLSTATE tables and Ada package | |
| Information and Definition Schemas, including short-name views | no new schema objects |
| Implementation-defined and –dependent Annexes | |
| Incompatibilities Annex | |
| Embedded SQL and  host language implications | |
| Dynamic SQL issues: including descriptor areas | |
| CLI issues | none |

***- End of paper -***