The new and improved SQL:2016 standard

Article *in* ACM SIGMOD Record · December 2018

DOI: 10.1145/3299887.3299897

7 authors, including:

Calisto Zuzarte
IBM
78 PUBLICATIONS **597** CITATIONS
SEE PROFILE

Zhen Hua Liu
Oracle Corporation
72 PUBLICATIONS **541** CITATIONS
SEE PROFILE

Fred Zemke
Oracle Corporation
9 PUBLICATIONS **131** CITATIONS
SEE PROFILE

The New and Improved SQL:2016 Standard

Jan Michels
Oracle Corporation
jan.michels@oracle.com

Keith Hare
JCC Consulting
keith@jcc.com

Krishna Kulkarni
SQL Consultant
krishna7277@gmail.com

Calisto Zuzarte
IBM Corporation
calisto@ca.ibm.com

Zhen Hua Liu
Oracle Corporation
zhen.liu@oracle.com

Beda Hammerschmidt
Oracle Corporation
beda.Hammerschmidt@oracle.com

Fred Zemke
Oracle Corporation
fred.zemke@oracle.com

ABSTRACT

SQL:2016 (officially called ISO/IEC 9075:2016, *Information technology — Database languages — SQL*) was published in December of 2016, replacing SQL:2011 as the most recent revision of the SQL standard. This paper gives an overview of the most important new features in SQL:2016.

1. INTRODUCTION

The database query language SQL has been around for more than 30 years. SQL was first standardized in 1986 by ANSI as a US standard and a year later by ISO as an international standard, with the first implementations of SQL preceding the standard by a few years. Ever since first published, the SQL standard has been a tremendous success. This is evidenced not only by the many relational database systems (RDBMSs) that implement SQL as their primary query[1] language but also by the so-called "NoSQL" databases that more and more see the requirement (and value) to add an SQL interface to their systems. One of the success factors of SQL and the standard is that it evolves as new requirements and technologies emerge. Be it the procedural [23], active database, or object-relational extensions that were added in the 1990s [22], the XML capabilities in the 2000s [19], temporal tables in the early 2010s [17], or the many other features described in previous papers [18], [20], [21], and not the least the features described in this paper, the SQL standard has always kept up with the latest trends in the database world.

SQL:2016 consists of nine parts [1]-[9], all of which were published together. However, with the exception of Part 2, "Foundation" [2], the other parts did not significantly change from their previous versions, containing mostly bug fixes and changes required to align with the new functionality in Part 2. As with the previous revisions, SQL:2016 is available for purchase from the ANSI[2] and ISO[3] web stores.

A high-level theme in SQL:2016 is expanding the SQL language to support new data storage and retrieval paradigms that are emerging from the NoSQL and Big Data worlds. The major new features in SQL:2016 are:

- Support for Java Script Object Notation (JSON) data
- Polymorphic Table Functions
- Row Pattern Recognition

SQL:2016 also includes a number of smaller features, such as additional built-in functions.

In addition to the formal SQL standard, the SQL committee has developed a series of Technical Reports (TRs). TRs, while non-normative, contain information that is useful for understanding how the SQL standard works. The SQL Technical Report series contains seven TRs [10] – [16] that are available at no charge from the ISO/IEC JTC1 "Freely available standards" web page[4].

The remainder of this paper is structured as follows: section 2 discusses the support for JSON data, section 3 discusses polymorphic table functions, section 4 discusses the row pattern recognition functionality, and section 5 showcases a select few of the smaller enhancements.

2. SUPPORT FOR JSON DATA

JSON [24] is a simple, semi-structured data format that is popular in developer communities because it is well suited as a data serialization format for data interchange. JSON data is annotated and the format allows nesting making it easy-to-read by both humans and machines. Many database applications that would benefit from JSON also need to access "traditional" tabular data. Thus, there is great value in storing, querying, and manipulating of JSON data inside an RDBMS as well as providing bi-directional conversion between relational data and JSON data.

To minimize the overhead of a new SQL data type, SQL:2016 uses the existing SQL string types (*i.e.*, either character strings like VARCHAR and CLOB, or

[1] "Query" in this context is not restricted to retrieval-only operations but also includes, among others, DML and DDL statements.

[2] http://webstore.ansi.org/

[3] https://www.iso.org/store.html

[4] http://standards.iso.org/ittf/PubliclyAvailableStandards/

binary strings like BLOB) to carry JSON values. Since there is no standard JSON query language yet, the SQL standard defines a path language for navigation within JSON data and a set of SQL built-in functions and predicates for querying within a JSON document.

We will illustrate the SQL/JSON[5] path language and SQL/JSON operators in the following sections. Due to space restrictions, we cannot cover all SQL/JSON features. For a detailed description of the SQL/JSON functionality the interested reader is referred to [15].

2.1 Querying JSON in SQL

2.1.1 Sample data

Our examples will use the table `T` shown below:

ID	JCOL
111	{ "Name" : "John Smith", "address" : { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }, "phoneNumber" : [{ "type" : "home", "number" : "212 555-1234" }, { "type" : "fax", "number" : "646 555-4567" }] }
222	{ "Name" : "Peter Walker", "address" : { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }, "phoneNumber" : [{ "type" : "home", "number" : "408 555-9876" }, { "type" : "office", "number" : "650 555-2468" }] }
333	{ "Name" : "James Lee" }

In `T`, the column `JCOL` contains JSON data stored in a character string.

Curly braces { } enclose JSON objects. A JSON object has zero or more comma-separated key/value pairs, called members. The key is a character string before a colon; the value is a JSON value placed after a colon. For example, in each row, the outermost JSON object has a key called "Name" with varying values in each row.

[5] The support for JSON is specified in Foundation [2] and not in a separate part (as is done, *e.g.*, for SQL/XML [9]). Still, the moniker SQL/JSON is associated with the JSON-specific functionality in SQL.

Square brackets [ ] enclose JSON arrays. A JSON array is an ordered, comma-separated list of JSON values. In the first and second rows, the key called "phoneNumber" has a value which is a JSON array. This illustrates how JSON objects and arrays can be nested arbitrarily.

Scalar JSON values are character strings, numbers, and the literals `true`, `false` and `null`.

The sample data is fairly homogeneous, but this is not a requirement of JSON. For example, the elements of an array do not need to be of the same type, and objects in different rows do not have to have the same keys.

2.1.2 IS JSON predicate

The `IS JSON` predicate is used to verify that an SQL value contains a syntactically correct JSON value. For example, this predicate can be used in a column check constraint, like this:

```
CREATE TABLE T (
  Id INTEGER PRIMARY KEY,
  Jcol CHARACTER VARYING ( 5000 )
           CHECK ( Jcol IS JSON ) )
```

The preceding might have been used to create the table `T`, insuring that the value of `Jcol` is valid JSON in all rows of `T`.

In the absence of such a constraint, one could use `IS JSON` as a filter to locate valid JSON data, like this:

```
SELECT * FROM T WHERE Jcol IS JSON
```

2.1.3 SQL/JSON path expressions

The remaining SQL/JSON operators to query JSON use the SQL/JSON path language. It is used to navigate within a JSON value to its components. It is similar to XPath for XML and also somewhat similar to object/array navigation in the JavaScript language. A path expression is composed of a sequence of path steps; each step can be associated with a set of predicates.

2.1.4 JSON_EXISTS predicate

`JSON_EXISTS` is used to determine if an SQL/JSON path expression has any matches in a JSON document. For example, this query finds the IDs of the rows with a key called "address":

```
SELECT Id
FROM T
WHERE JSON_EXISTS ( Jcol,
         'strict $.address' )
```

The example works as follows. The first argument to `JSON_EXISTS`, `Jcol`, specifies the context item (JSON value) on which `JSON_EXISTS` operates. The keyword `strict` selects the strict mode; the alternative is `lax`. As its name implies, strict mode expects that the JSON document conforms strictly to the path expression, whereas lax mode relaxes some of these expectations, as will be seen in later examples. `$.address` is the path expression that is applied to the context item. In the path expression, `$` is a variable referencing the context item, the period is an operator used to navigate to a key/value pair within a JSON object, and `address` is the name of the desired key. The `JSON_EXISTS` predicate will be true if this path expression successfully finds one or more such key/value pairs. With the sample data, the query will find the IDs 111 and 222 but not 333.

2.1.5 JSON_VALUE function

The `JSON_VALUE` function is used to extract a scalar value from a given JSON value. For example, to find the value of the "Name" key/value pair in each row, one could use this query:

```
SELECT JSON_VALUE ( Jcol,
   'lax $.Name' ) AS Name
FROM T
```

This example uses lax mode, which is more forgiving than strict mode. For example, it is common to use a singleton JSON value interchangeably with an array of length one. To accommodate that convention, in lax mode, if a path step requires an array but does not find one, the data is implicitly wrapped in a JSON array. Conversely, if a path step expects a non-array but encounters an array, the array is unwrapped into a sequence of items, and the path step operates on each item in the sequence.

The following query might be used to find the first phone number in each row.

```
SELECT Id, JSON_VALUE ( Jcol,
   'lax $.phoneNumber[0].number' )
   AS Firstphone
FROM T
```

JSON arrays are 0-relative, so the first element is addressed `[0]`. The last row of sample data has no such data; in that case, lax mode produces an empty sequence (instead of an error) and `JSON_VALUE` will return a null value. The result of the query is:

ID	FIRSTPHONE
111	212 555-1234
222	408 555-9876
333	

In the last row above, the `FIRSTPHONE` cell is blank, indicating an SQL null value, a convention we will use throughout this paper.

Or suppose the task is to find all fax phone numbers. The query to solve this is

```
SELECT Id, JSON_VALUE ( Jcol,
   'lax $.phoneNumber
      ? ( @.type == "fax" ).number' )
    AS Fax
FROM T
```

This query illustrates a filter, introduced by a question mark and enclosed within parentheses. The filter is processed as follows: since the query is in lax mode, the array `$.phoneNumber` is unwrapped into a sequence of items. Each item is tested against the predicate within the parentheses. In this predicate, the at-sign `@` is a variable bound to the item being tested. The predicate `@.type == "fax"` is true if the value of the `"type"` member equals `"fax"`. The result of the filter is the sequence of just those items that satisfied the predicate. Finally, the member accessor `.number` is applied, to obtain the value of the member whose key is `"number"`. The result of the query is:

ID	FAX
111	646 555-4567
222	
333	

All of these examples returned character string data, the default return type of `JSON_VALUE`. Optional syntax can be used to specify other return types, as well as various options to handle empty or error results.

2.1.6 JSON_QUERY function

`JSON_VALUE` can only extract scalars from a JSON value. The `JSON_QUERY` function, on the other hand, is used to extract a fragment (*i.e.*, an SQL/JSON object, array, or scalar, possibly wrapped in an SQL/JSON array, if the user specifies this) from a given JSON value. For example, to obtain the complete value of the "address" key, this query might be used:

```
SELECT Id, JSON_QUERY ( Jcol,
   'lax $.address' ) AS Address
FROM T
```

With the following results[6]:

[6] The result shows some insignificant pretty-printing whitespace. The SQL standard does not prescribe this. A conforming implementation is allowed to either add or omit insignificant whitespace. Here it is only shown for readability.

ID	ADDRESS
111	{ "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }
222	{ "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }
333	

In the last row, `ADDRESS` is null because the data does not match the path expression. There are options to obtain other behaviors for empty and error conditions.

2.1.7 JSON_TABLE function

`JSON_TABLE` is a table function that is invoked in the `FROM` clause of a query to generate a relational table from a JSON value. As a simple example, to extract the scalar values of name and ZIP code from each JSON document, the following query can be used:

```
SELECT T.Id, Jt.Name, Jt.Zip
FROM T,
  JSON_TABLE ( T.Jcol, 'lax $'
    COLUMNS (
      Name VARCHAR ( 30 )
           PATH 'lax $.Name'
      Zip  VARCHAR ( 5 ) PATH
           'lax $.address.postalCode'
      )
   ) AS Jt
```

In this example, the first path expression `'lax $'` is the "row pattern" used to locate rows. The path expression here is the simplest possible, just $, meaning that there is no navigation within a JSON document; however, if the row data were deeply nested within a JSON document, then the row pattern would be more complicated.

The example defines two output columns, NAME and ZIP. Each output column has its own PATH clause, specifying the "column pattern" that is used to navigate within a row to locate the data for a column.

The example has the following results on the sample data:

ID	NAME	ZIP
111	John Smith	10021
222	Peter Walker	95111
333	James Lee	

`JSON_TABLE` also allows unnesting of (even deeply) nested JSON objects/arrays in one invocation by using a nested `COLUMNS` clause, as the next example illustrates. This query will return the name and phone number and type for each person:

```
SELECT T.Id, Jt.Name, Jt.Type,
       Jt.Number
FROM T,
  JSON_TABLE ( T.Jcol, 'lax $'
    COLUMNS
      ( Name VARCHAR ( 30 )
             PATH 'lax $.Name',
        NESTED PATH
            'lax $.phoneNumber[*]'
        COLUMNS
          ( Type VARCHAR ( 10 )
                PATH 'lax $.type',
            Number VARCHAR ( 12 )
                PATH 'lax $.number' )
        )
    ) AS Jt
```

The preceding example has an outer row pattern `'lax $'` and within that, a nested row pattern `'lax $.phoneNumber[*]'`. The nested row pattern uses the wildcard array element accessor `[*]` to iterate over all elements of the `phoneNumber` array. Thus it is possible to flatten hierarchical data. The first column `Name` is found at the outer level of the hierarchy, whereas the nested columns `Type` and `Number` are found in the inner level of the hierarchy. The result on the sample data is:

ID	NAME	TYPE	NUMBER
111	John Smith	home	212 555-1234
111	John Smith	fax	646 555-4567
222	Peter Walker	home	408 555-9876
222	Peter Walker	office	650 555-2468
333	James Lee		

The last row has no phone number. If one wanted to include only those JSON documents that have a phoneNumber member, the following `WHERE` clause could be appended to the previous query:

```
WHERE JSON_EXISTS ( T.Jcol,
                'lax $.phoneNumber' )
```

In the sample data, this `WHERE` clause would filter out the row whose ID is 333.

2.1.8 Structural-inspection methods

Because the structure of JSON data may not be known a priori and/or vary from one JSON value to the next, the SQL/JSON path language provides methods that allow for the inspection of the structural aspects of a JSON value. These methods are:

— `keyvalue`, which returns an SQL/JSON object containing three members for the key, the bound value, and an ID uniquely identifying the containing input object for each member of the input SQL/JSON object.

— `type`, which returns “object”, “array”, “string”, “number”, etc. corresponding to the actual type of the SQL/JSON item.

— `size`, which returns the number of elements, if the input SQL/JSON item is an array; otherwise it returns 1.

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

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

2.2 SQL/JSON constructor functions

SQL/JSON constructor functions use values of SQL types and produce JSON (either JSON objects or JSON arrays) represented in SQL character or binary string types.

The functions are: `JSON_OBJECT`, `JSON_ARRAY`, `JSON_OBJECTAGG`, and `JSON_ARRAYAGG`. The first two are scalar functions, whereas the latter two are aggregate functions. As with other SQL functions/expressions, these can be arbitrarily nested. This supports the construction of JSON data of arbitrary complexity.

For example, given the well-known `Employees` and `Departments` tables, one can construct a JSON object for each department that contains all employees and their salary, sorted by increasing salary using this query:

```
SELECT JSON_OBJECT
 ( KEY 'department' VALUE D.Name,
   KEY 'employees'
    VALUE JSON_ARRAYAGG
          ( JSON_OBJECT
            ( KEY 'employee'
               VALUE E.Name,
              KEY 'salary'
               VALUE E.Salary )
             ORDER BY E.Salary ASC )
 ) AS Department
FROM Departments D, Employees E
WHERE D.Dept_id = E.Dept_id
GROUP BY D.Name
```

with results that might look like this:

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

3. POLYMORPHIC TABLE FUNCTIONS

The SQL standard prior to the 2016 release had only support for monomorphic table functions, *i.e.*, the definition of both the output table and the set of input parameters were fixed at function creation time. With the specification of polymorphic table functions (PTFs), SQL:2016 includes a very powerful enhancement to table functions. With this feature, the RDBMS is able to evaluate custom functionality closer to the data. For example, the MapReduce paradigm could be implemented using PTFs.

A polymorphic table function is a function that may have generic table input parameter(s) whose row type(s) may be unknown at creation time. The PTF may return a table whose row type also may be unknown when the function is created. The row type of the result may depend on the function arguments or on the row type(s) of the input table(s) in the invocation of the PTF. When a PTF is invoked in a query, the RDBMS and the PTF interact through a family of one to four SQL-invoked procedures. These procedures are called the PTF component procedures[7].

In the next sections, we describe these four component procedures and give two examples of user-defined PTFs. Due to space restrictions, we cannot cover all PTF features. For a detailed description of all aspects of PTFs (including the different perspectives of query author, PTF author, and RDBMS developer) the interested reader is referred to [16].

3.1 PTF Component Procedures

There are one to four PTF component procedures:

1. “describe”: The PTF describe component procedure is called once during compilation of the query that invokes the PTF. The primary task of the PTF describe component procedure is to determine the row type of the output table. This component procedure receives a description of the input tables and their ordering (if any) as well as any scalar input arguments that are compile-time constants. This component procedure is optional if all result columns are defined statically in the `CREATE FUNCTION` statement or if the PTF has only result columns that are passed through unchanged from the input table(s); otherwise it is mandatory.

2. “start”: The PTF start component procedure is called at the start of the execution of the PTF to allocate any resources that the RDBMS does not provide. This procedure is optional.

[7] SQL:2016 uses the four PTF component procedures as a specification vehicle. A conforming implementation may substitute this interface with an implementation-defined API.

3. "fulfill": The PTF fulfill component procedure is called during the execution to deliver the output table by "piping" rows to the RDBMS. This is the component procedure that reads the contents of the input table(s) and generates the output table. This procedure is mandatory.

4. "finish": The PTF finish component procedure is called at the end of the execution to deallocate any resources allocated by the PTF start component procedure. This procedure is optional.

3.2 Execution model

The SQL standard defines the run-time execution of a PTF using an abstraction called a virtual processor, defined as a processing unit capable of executing a sequential algorithm. Using techniques such as multiprocessing, a single physical processor might host several virtual processors. Virtual processors may execute independently and concurrently, either on a single physical processor or distributed across multiple physical processors. There is no communication between virtual processors. The RDBMS is responsible for collecting the output on each virtual processor; the union of the output from all virtual processors is the result of the PTF. The virtual processor abstraction is the standard's way of permitting but not requiring parallelization of PTF execution.

3.3 Examples

This section illustrates the value of PTFs using a couple of examples. The first one is a simple example introducing the polymorphic nature of a PTF. The second one illustrates a variety of options including multiple input tables with different input semantics.

3.3.1 CSV reader table function

Consider a file with a comma-separated list of values (CSV file). The first line of the file contains a list of column names, and subsequent lines of the file contain the actual data. A PTF called `CSVreader` was created to read a CSV file and provide its data as a table in the `FROM` clause of a query. The effective signature of the PTF is:

```
FUNCTION CSVreader (
     File VARCHAR ( 1000 ),
     Floats DESCRIPTOR DEFAULT NULL,
     Dates DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
NOT DETERMINISTIC
CONTAINS SQL
```

This signature has two parameter types that are distinctive to PTFs. (a) `DESCRIPTOR` is a type that is capable of describing a list of column names, and optionally for each column name, a data type. (b) `TABLE` denotes the generic table type, a type whose value is a table. The row type of the table is not specified, and may vary depending on the invocation of the PTF. Here, the `TABLE` specification specifies a generic table output of `CSVreader`. The row type is unknown at creation time and this is characteristic of a PTF.

A user reference guide accompanying a PTF will need to describe the semantics of the input parameters and what the output will be. For example, here `File` is the name of a file that contains the comma-separated values which are to be converted to a table. The first line of the file contains the names of the resulting columns. Succeeding lines contain the data. Each line after the first will result in one row of output, with column names as determined by the first line of the input. In the example above, `Floats` and `Dates` are PTF descriptor areas, which provide a list of the column names that are to be interpreted numerically and as dates, respectively; the data types of all other columns will be `VARCHAR`. With that information a query such as the following can be written:

```
SELECT *
FROM TABLE
    ( CSVreader (
         File => 'abc.csv',
         Floats => DESCRIPTOR
          ( "principal", "interest" )
         Dates => DESCRIPTOR
          ( "due_date" ) ) ) AS S
```

In the `FROM` clause, the `TABLE` operator introduces the invocation of a table function. A table function might be either a conventional (monomorphic) table function or a PTF. In this case, because `CSVreader` is declared with return type `TABLE`, this is a PTF invocation. This invocation says that `CSVreader` should open the file called abc.csv. The list of output column names is found in the first line of the file. Among these column names, there must be columns named `principal` and `interest`, which should be interpreted as numeric values, and a column named `due_date` which should be interpreted as a date. During the compilation of this query, the RDBMS will call the PTF describe component procedure and provide this information to the component procedure. In return, the component procedure will provide the RDBMS with the row type of the result table.

The component procedures are SQL procedures that are specified as a part of the definition of a PTF. For example:

```
CREATE FUNCTION CSVreader (
      File VARCHAR(1000),
      Floats DESCRIPTOR DEFAULT NULL,
      Dates DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
```

```
NOT DETERMINISTIC CONTAINS SQL
PRIVATE DATA ( FileHandle INTEGER )
DESCRIBE WITH PROCEDURE
         CSVreader_describe
START WITH PROCEDURE
         CSVreader_start
FULFILL WITH PROCEDURE
         CSVreader_fulfill
FINISH WITH PROCEDURE
         CSVreader_finish
```

The procedures `CSVreader_describe`, `CSVreader_start`, `CSVreader_fulfill`, and `CSVreader_finish` are SQL stored procedures and can take advantage of the existing procedural language, dynamic SQL, and other existing SQL capabilities.

3.3.2 User Defined Join table function

The following example demonstrates a variety of options that can be specified in a PTF. It is a function that has input tables and also introduces options related to input table semantics (row or set semantics, keep or prune when empty) and the use of pass-through columns to flow data unaltered to the result table.

The PTF `UDJoin` performs a custom user-defined join. It takes two input tables, `T1` and `T2`, and matches rows according to some user defined join criterion that may not be built into the database system. The PTF has the following signature:

```
CREATE FUNCTION UDJoin
 ( T1 TABLE PASS THROUGH
            WITH SET SEMANTICS
            PRUNE WHEN EMPTY,
   T2 TABLE PASS THROUGH
            WITH SET SEMANTICS
            KEEP WHEN EMPTY
 ) RETURNS ONLY PASS THROUGH
```

The `RETURNS ONLY PASS THROUGH` syntax declares that the PTF does not generate any columns of its own; instead the only output columns are passed through from input columns.

`WITH SET SEMANTICS` is specified when the outcome of the function depends on how the data is partitioned. A table should be given set semantics if all rows of a partition should be processed on the same virtual processor. In this example, the entire table T2 is sent to the virtual processors.

`WITH ROW SEMANTICS` specified on an input table means that the result of the PTF is decided on a row-by-row basis for this input table. This is specified if the PTF does not care how rows are assigned to virtual processors. Only tables with set semantics may be partitioned and/or ordered.

The `KEEP WHEN EMPTY` option implies that the PTF could generate result rows even if the input table (in this example, `T2`), is empty. The result rows are based on rows from the other input table `T1`. `T1` is specified with `PRUNE WHEN EMPTY`, meaning that there is no output when the input table is empty. This example is analogous to a left outer join.

The `UDJoin` PTF can be invoked in a query like this:

```
SELECT E.*, D.*
FROM TABLE
     ( UDJoin (
           T1 => TABLE (Emp) AS E
                 PARTITION BY Deptno,
           T2 => TABLE (Dept) AS D
                 PARTITION BY Deptno
                 ORDER BY Tstamp ) )
```

In this example, both input tables have set semantics, which permits the use of `PARTITION BY` and `ORDER BY` clauses. `PARTITION BY` says that the input table is partitioned on a list of columns; each partition must be processed on a separate virtual processor. In this example, since there are two partitioned tables, the RDBMS must in fact create the cross product of the partitions of the two tables, with a virtual processor for each combination of partitions. (In the absence of `PARTITION BY`, a table with set semantics constitutes a single partition.) The second input table is also ordered; the RDBMS must sort the rows of each partition prior to passing them to the fulfill component procedure executing on any virtual processor.

Consider the following variation of the same query:

```
SELECT E.*, D.*
FROM TABLE
    ( UDJoin
     ( T1 => TABLE (Emp) AS E
             PARTITION BY Deptno,
       T2 => TABLE (Dept) AS D
             PARTITION BY Deptno
             ORDER BY Tstamp
             COPARTITION (Emp,Dept)))
```

Here, the `COPARTITION` clause allows each virtual processor to avoid the cross product as in the earlier example and collocates the corresponding values in the `Deptno` columns from the two tables in the same virtual processor.

4. ROW PATTERN RECOGNITION

Row Pattern Recognition (RPR) can be used to search an ordered partition of rows for matches to a regular expression. RPR can be supported in either the `FROM` clause or the `WINDOW` clause. This article will discuss

RPR in the FROM clause; RPR in the WINDOW clause uses much the same syntax and semantics.

RPR in the FROM clause uses the keyword MATCH_RECOGNIZE as a postfix operator on a table, called the *row pattern input table*. MATCH_RECOGNIZE operates on the row pattern input table and produces the *row pattern output table* describing the matches to the pattern that are discovered in the row pattern input table. There are two principal variants of MATCH_RECOGNIZE:

— ONE ROW PER MATCH, which returns a single summary row for each match of the pattern (the default).

— ALL ROWS PER MATCH, which returns one row for each row of each match.

The following example illustrates MATCH_RECOGNIZE with the ONE ROW PER MATCH option. Let Ticker (Symbol, Tradeday, Price) be a table with three columns representing historical stock prices. Symbol is a character column, Tradeday is a date column and Price is a numeric column. It is desired to partition the data by Symbol, sort it into increasing Tradeday order, and then detect maximal "V" patterns in Price: a strictly falling price, followed by a strictly increasing price. For each match to a V pattern, it is desired to report the starting price, the price at the bottom of the V, the ending price, and the average price across the entire pattern. The following query may be used to perform this pattern matching problem:

```
SELECT
   M.Symbol,  /* ticker symbol */
   M.Matchno, /* match number */
   M.Startp,  /* starting price */
   M.Bottomp, /* bottom price */
   M.Endp,    /* ending price */
   M.Avgp     /* average price */
FROM Ticker
  MATCH_RECOGNIZE (
    PARTITION BY Symbol
    ORDER BY Tradeday
    MEASURES
      MATCH_NUMBER() AS Matchno,
      A.Price        AS Startp,
      LAST (B.Price) AS Bottomp,
      LAST (C.Price) AS Endp,
      AVG (U.Price)  AS Avgp
    ONE ROW PER MATCH
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (A B+ C+)
    SUBSET U = (A, B, C)
    DEFINE
    /* A defaults to True,
         matches any row */
     B AS B.Price < PREV (B.Price),
     C AS C.Price > PREV (C.Price)
   ) AS M
```

In this example:

— Ticker is the row pattern input table.

— MATCH_RECOGNIZE introduces the syntax for row pattern recognition.

— PARTITION BY specifies how to partition the row pattern input table. If omitted, the entire row pattern input table constitutes a single partition.

— ORDER BY specifies how to order the rows within partitions.

— MEASURES specifies *measure columns*, whose values are calculated by evaluating expressions related to the match. The first measure column uses the nullary function MATCH_NUMBER(), whose value is the sequential number of a match within a partition. The third and fourth measure columns use the LAST operation, which obtains the value of an expression in the last row that is mapped by a row pattern match to a row pattern variable. LAST is one of many row pattern navigation operations, which may be used to navigate to specific rows of interest within a match.

— ONE ROW PER MATCH specifies that the result, the *row pattern output table,* will have a single row for each match that is found in the row pattern input table; each output row has one column for each partitioning column and one column for each measure column.

— AFTER MATCH SKIP specifies where to resume looking for the next row pattern match after successfully finding a match. In this example, PAST LAST ROW specifies that pattern matching will resume after the last row of a successful match.

— PATTERN specifies the row pattern that is sought in the row pattern input table. A row pattern is a regular expression using primary row pattern variables. In this example, the row pattern has three primary row pattern variables (A, B, and C). The pattern is specified as (A B+ C+) which will match a single A followed by one or more Bs followed by one or more Cs. An extensive set of regular expression specifications are supported.

— SUBSET defines the union row pattern variable U as the union of A, B, and C.

— DEFINE specifies the Boolean condition that defines a primary row pattern variable; a row must satisfy the Boolean condition in order to be mapped to a particular primary row pattern variable. This example uses PREV, a row pattern navigation operation that evaluates an expression in the previous row. In this example, the row pattern variable A is undefined, in which case any row can be mapped to A.

— AS M defines the range variable M to associate with the row pattern output table.

Here is some sample data for Ticker, having one match to the pattern in the example (the mapping of rows to primary row pattern variables is shown in the last column):

Symbol	Tradeday	Price	Mapped to
XYZ	2009-06-08	50	
XYZ	2009-06-09	60	A
XYZ	2009-06-10	49	B
XYZ	2009-06-11	40	B
XYZ	2009-06-12	35	B
XYZ	2009-06-13	45	C
XYZ	2009-06-14	45	

Here is the row of the row pattern output table generated by the match shown above:

Symbol	Matchno	Startp	Bottomp	Endp	Avgp
XYZ	1	60	35	45	45.8

Due to space restrictions, we cannot cover all RPR features. For a detailed description of the RPR functionality the interested reader is referred to [14].

5. ADDITIONAL FUNCTIONALITY

5.1 Default values and named arguments for SQL-invoked functions

SQL:2011 allowed for a parameter of an SQL-invoked procedure to have a default value and thus be optional when invoking the procedure. A companion enhancement is invoking a procedure using named arguments [18]. SQL:2016 extends this functionality to cover SQL-invoked functions as well. For example, a function that computes the total compensation as the sum of the base salary and the bonus (where the bonus by default is 1000) can be defined like this:

```
CREATE FUNCTION Total_comp (
   Base_sal DECIMAL(7,2),
   Bonus DECIMAL(7,2) DEFAULT 1000.00
   ) RETURNS DECIMAL(8,2)
     LANGUAGE SQL
     RETURN Base_sal + Bonus;
```

This function can now be invoked in different ways:

— Passing all arguments by position:

```
      Total_comp(9000.00, 1000.00)
```

— Passing all non-defaulted arguments by position:

```
      Total_comp(9000.00)
```

— Passing all arguments by name (in this case the order of the arguments does not need to match the order of the parameters in the function signature):

```
      Total_comp(Bonus=>1000.00,
                 Base_sal=>9000.00)
```

— Passing all non-defaulted arguments by name:

```
      Total_comp(Base_sal=>9000.00)
```

All of these invocations return the same result (10000.00). It should be clear that the greatest benefit of named and defaulted arguments can be realized when the parameter list is long and many parameters have useful defaults. Specifying some argument values by position and other arguments by name within the same invocation is not supported.

5.2 Additional built-in functions

SQL:2016 adds support for additional scalar mathematical built-in functions including trigonometric and logarithm functions. The trigonometric functions are sine, cosine, tangent, and their hyperbolic and inverse counterparts. Besides the existing natural logarithm function, SQL:2016 now supports a general logarithm function (where the user can specify an arbitrary value for the base) and a common logarithm function (with the base fixed at 10).

LISTAGG is a new aggregate function that allows concatenating character strings over a group of rows.

6. FUTURES

The SQL standards committee is currently working on additional expansions in three areas, support for multi-dimensional arrays, support for streaming data, and support for property graphs.

The work on multi-dimensional arrays (aka SQL/MDA) is well under way and will be completed by the end of 2018. This new incremental part adds a multi-dimensional array type so that instances of a multi-dimensional array can be stored in a column of a table and operations can be executed close to the data in the RDBMS.

The SQL committee has begun investigating requirements for streaming data and property graphs in the context of SQL.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge the following people, who at one point or another in one form or another have contributed to the standardization of the functionality presented in this paper: Jim Melton, Fatma Ozcan, Hamid Pirahesh, Doug McMahon, Shashaanka Agrawal, Ivan Bowman, Karl Schendel, Janhavi Rajagopal, Kathy McKnight, Hans Zeller, Dan

Pasco, Andy Witkowski, Chun-chieh Lin, Lei Sheng, and Taoufik Abdellatif.

The authors would like to thank the following people for reviewing earlier drafts of this paper and providing valuable feedback: Jim Melton, Hermann Baer, Mark Anderson, Andy Witkowski, and Claire McFeely.

8. REFERENCES

[1] ISO/IEC 9075-1:2016, *Information technology — Database languages — SQL* — Part 1: Framework (SQL/Framework)

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

[3] ISO/IEC 9075-3:2016, *Information technology — Database languages — SQL* — Part 3: Call-Level Interface (SQL/CLI)

[4] ISO/IEC 9075-4:2016, *Information technology — Database languages — SQL* — Part 4: Persistent stored modules (SQL/PSM)

[5] ISO/IEC 9075-9:2016, *Information technology — Database languages — SQL* — Part 9: Management of External Data (SQL/MED)

[6] ISO/IEC 9075-10:2016, *Information technology — Database languages — SQL* — Part 10: Object language bindings (SQL/OLB)

[7] ISO/IEC 9075-11:2016, *Information technology — Database languages — SQL* — Part 11: Information and definition schemas (SQL/Schemata)

[8] ISO/IEC 9075-13:2016, *Information technology — Database languages — SQL* — Part 13: SQL Routines and types using the Java programming language (SQL/JRT)

[9] ISO/IEC 9075-14:2016, *Information technology — Database languages — SQL* — Part 14: XML-Related Specifications (SQL/XML)

[10] ISO/IEC TR 19075-1:2011, *Information technology — Database languages — SQL Technical Reports* — Part 1: XQuery Regular Expression Support in SQL, http://standards.iso.org/ittf/PubliclyAvailableStandards/

[11] ISO/IEC TR 19075-2:2015, *Information technology — Database languages — SQL Technical Reports* — Part 2: SQL Support for Time-Related Information, http://standards.iso.org/ittf/PubliclyAvailableStandards/

[12] ISO/IEC TR 19075-3:2015, *Information technology — Database languages — SQL Technical Reports* — Part 3: SQL Embedded in Programs using the Java^{TM} programming language, http://standards.iso.org/ittf/PubliclyAvailableStandards/

[13] ISO/IEC TR 19075-4:2015, *Information technology — Database languages — SQL Technical Reports* — Part 4: SQL with Routines and types using the Java^{TM} programming language, http://standards.iso.org/ittf/PubliclyAvailableStandards/

[14] ISO/IEC TR 19075-5:2016, *Information technology — Database languages — SQL Technical Reports* — Part 5: Row Pattern Recognition in SQL, http://standards.iso.org/ittf/PubliclyAvailableStandards/

[15] ISO/IEC TR 19075-6:2017, *Information technology — Database languages — SQL Technical Reports* — Part 6: SQL support for JavaScript Object Notation (JSON), http://standards.iso.org/ittf/PubliclyAvailableStandards/

[16] ISO/IEC TR 19075-7:2017, *Information technology — Database languages — SQL Technical Reports* — Part 7: Polymorphic table functions in SQL, http://standards.iso.org/ittf/PubliclyAvailableStandards/

[17] Krishna Kulkarni and Jan-Eike Michels, “Temporal features in SQL:2011”, SIGMOD Record Vol. 41 No. 3, September 2012, https://sigmodrecord.org/publications/sigmodRecord/1209/pdfs/07.industry.kulkarni.pdf

[18] Fred Zemke, “What’s new in SQL:2011”, SIGMOD Record, Vol. 41, No. 1, March 2012, https://sigmodrecord.org/publications/sigmodRecord/1203/pdfs/10.industry.zemke.pdf

[19] Andrew Eisenberg and Jim Melton, “Advancements in SQL/XML”, SIGMOD Record Vol. 33 No. 3, September 2004, https://sigmodrecord.org/publications/sigmodRecord/0409/11.JimMelton.pdf

[20] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke, “SQL:2003 has been published”, SIGMOD Record Vol. 33 No. 1, March 2004, https://sigmodrecord.org/publications/sigmodRecord/0403/E.JimAndrew-standard.pdf

[21] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna Kulkarni, Peter Schwarz, Kathy Zeidenstein, “SQL and Management of External Data”, SIGMOD Record Vol. 30 No. 1, March 2001, https://sigmodrecord.org/publications/sigmodRecord/0103/JM-Sta.pdf

[22] Andrew Eisenberg and Jim Melton, “SQL:1999, formerly known as SQL3”, SIGMOD Record Vol. 28 No. 1, March 1999, https://sigmodrecord.org/publications/sigmodRecord/9903/standards.pdf.gz

[23] Andrew Eisenberg, “New Standard for Stored Procedures in SQL”, SIGMOD Record Vol 25 No. 4, Dec.1996, https://sigmodrecord.org/issues/96-12/sqlpsm.ps

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