Couchbase

Comparing Two SQL-Based Approaches for Querying JSON: SQL++ and SQL:2016

Don Chamberlin

08/2019

Introduction

According to GitHub's Octoverse 2018 report, JavaScript is the most widely-used programming language in the world, and has occupied that position for more than five years. JavaScript is now used by more than 95% of websites, according to W3techs.com. JavaScript Object Notation, abbreviated JSON, is the native data format for storing and manipulating data generated by JavaScript applications. The large amount of data generated by websites in JSON format has made clear the importance of a query capability for JSON within databases.

In addition to the popularity of JavaScript, there are other good reasons for the importance of JSON as a data storage format. Since JSON data is self-describing, it's not necessary for all the objects in a JSON collection to have the same structure. Each object can have the members that are pertinent to it, identified by their respective names. These names occupy some space in storage, but in today's world, that's a good tradeoff. JSON data is often schemaless, but even when a schema is present, the self-describing nature of JSON makes schema evolution easy. JSON also makes it possible to store objects with complex internal structure, which would need to be spread across multiple tables in a relational database.

Several approaches have been proposed for querying JSON data. Some of these proposals are based on extensions to the SQL query language. The purpose of these extensions is to build on the database industry's investment in SQL knowledge, tools, and data. SQL is the world's most widely used database query language. Ideally, JSON extensions to SQL should preserve the declarative nature of the language, and should enable application developers to transition from SQL to the extended language with minimal training. The extended language should be suitable both for simple queries against JSON data and for complex JSON-to-JSON transforms.

Extending SQL for JSON data involves several challenges. SQL was designed for data that is stored in flat tables with well-defined schemas. JSON data is more flexible, permitting nested data structures, and does not always have a well-defined schema, or any schema at all. At first glance, SQL and JSON do not seem to be a good match. But research and experience have shown that SQL queries and JSON data are more closely aligned than they appear.

This paper assumes that its readers have a basic working knowledge of JSON and SQL. Its purpose is to examine and compare two quite different SQL-based approaches to querying JSON data. Both of these approaches are well defined and are available in the marketplace today.

The first approach is to create a new SQL-based language that is specifically designed for JSON data. Rather than operating on tables, this language operates on JSON collections, typically arrays of JSON objects. The goal of this approach is to find the minimum set of SQL extensions that are sufficient for JSON queries and transforms. In 2015, Dr. Yannis Papakonstantinou and others at the University of California, San Diego proposed an SQL-based JSON query language called SQL++ [1, 2]. Versions of SQL++ have been implemented by the AsterixDB project at the University of California, Irvine [3, 4], and by Couchbase, Inc. [5, 6]. The Couchbase implementation is called N1QL, a name that suggests “not first normal form” because N1QL is not limited to querying flat (first normal form) tables. An open-source language called PartiQL, similar to SQL++, is supported by Amazon Web Services [7]. The SQL++ example queries that you are about to read are written in N1QL and have run successfully on Couchbase Server and (with minor syntactic variations) on Amazon’s PartiQL reference interpreter.

The second approach is to extend SQL to allow JSON documents to be stored in columns of tables. This approach has been adopted by the latest version of the international SQL language standard, released in 2016 and informally referred to as SQL:2016. In this approach, JSON documents are stored in the form of strings, and a set of SQL functions are provided for processing the strings with JSON semantics. The current version of the SQL standard, at several thousand pages, is daunting to read (or to pick up, for that matter), and you have to pay to get a copy. Fortunately, the JSON-related features of SQL:2016 have been summarized in a smaller and more readable technical report that is available for free [8].

In the following sections, we’ll introduce SQL++ and SQL:2016, take a look at some example queries written in both languages, and conclude with an analysis and comparison of the two languages. Each example query will be labeled with the language in which it is written. One note on terminology: various terms have been used for the name-value pairs that are found inside JSON objects. Following the convention in [8], we’ll use the term "members" for these name-value pairs.

The Languages

In comparing SQL++ and SQL:2016, the first thing to notice is that the two languages are designed to operate in quite different environments. An SQL++ query operates on JSON inputs and generates a JSON output. An SQL:2016 query operates on tables as inputs and generates a table as an output. As far as SQL:2016 is concerned, JSON exists only in a column of a table. We’ll need to take these differences into account when we construct example queries to compare the two languages.

SQL++

SQL++ is based on the observation that a JSON object is very similar to a row of a table. Both consist of (name, value) pairs (in a row, the name is a column-name). Furthermore, an array of JSON objects can be similar to a table if the objects have a common structure. Based on these similarities, SQL++ applies the familiar operations of SQL to JSON arrays of objects. The goal is to remain as close to SQL as possible. The keywords of SQL including SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, and JOIN are all present in SQL++, and have pretty much the same meaning that they have in SQL. For example, FROM is used to iterate over the objects in an array, WHERE is used to choose a subset of these objects based on their values, GROUP BY forms a list of objects into groups, and SELECT is used to specify the desired output.

SQL++ has made a small extension to the JSON data model by introducing the concept of an unordered collection of objects. JSON arrays are always ordered. In SQL++, an unordered collection behaves exactly like an array, except that its order is not significant. This provides a query optimizer with opportunities to optimize execution plans for parts of a query where order is not significant. Any desired ordering can be imposed by ORDER BY as the last step in query processing before the result is returned in the form of a JSON array.

We don't need to spend any more space here explaining the syntax of SQL++, since it is the same as SQL with a few small extensions. We'll show by a series of examples how the operators of SQL have been adapted to operate on JSON.

SQL:2016

Like all versions of SQL, SQL:2016 operates exclusively on tables. In SQL:2016, JSON exists only as a string (usually a character string) in a column of a table. As far as the operators of SQL (JOIN, GROUP BY, etc.) are concerned, a JSON instance is just a character string. JSON semantics are exposed only by a set of special functions. These functions are organized into two categories called SQL/JSON Query Functions and SQL/JSON Constructor Functions.

SQL/JSON Query Functions

The SQL/JSON Query Functions of SQL:2016 are as follows:

- JSON_EXISTS( ) follows a "path" through a JSON instance and returns `true` if the path leads to an actual data value (i.e., if the path exists and is valid in the JSON instance).

- JSON_VALUE( ), like JSON_EXISTS( ), follows a path through a JSON instance. But instead of simply returning `true` or `false`, it returns the scalar value at the end of the path, converted into one of the SQL data types (if no return type is specified, the value is returned as a character string).

- JSON_QUERY( ) also follows a path through a JSON instance. But in this case the result is not constrained to be a scalar value and is not converted to an SQL data type. The result of JSON_QUERY( ) can be any JSON instance, returned as a character string in JSON format.

- JSON_TABLE( ) is a function whose purpose is to convert a JSON instance into a table. To do this, JSON_TABLE( ) takes as input a JSON instance and possibly many path expressions, separated by keywords like NESTED PATH and COLUMNS. Any query that involves joining or grouping the internal parts of a JSON instance will require use of JSON_TABLE( ). This is because the joining and grouping operators of SQL cannot see these internal parts; to SQL the whole JSON instance is just a character string. In order to do joining or grouping, the JSON instance must be first converted into a table so that the SQL operators can be used. We will soon see some example queries that require use of JSON_TABLE( ).

One more observation: JSON_TABLE( ) is a very complex function, and is capable of converting pretty much any JSON instance into a single table, even when this conversion is unwise. Consider this JSON instance:

```
{"persons":
  [ { "id": 101,
      "name": "Bill",
      "location": "Chicago",
      "skills":
            [ { "skill": "welding", "level": 5},
              { "skill": "cooking", "level": 2}
            ],
      "hobbies":
            [ { "hobby": "bird watching", "years": 3 },
              { "hobby": "stamp collecting", "years": 4}
            ]
    }, . . .
  ]
}
```

The right way to convert this instance into relational form is to convert it into three normalized tables that have the following structure (of course, the names could be changed):

```
      persons ( id, name, location )
      skills ( id, skill, level )
      hobbies ( id, hobby, years )
```

This conversion could be accomplished by three calls to JSON_TABLE( ). But JSON_TABLE( ) also provides options to convert this instance into a single table in which the (dubious) relationship between skills and hobbies is governed by a PLAN clause with options like INNER, OUTER, UNION, and CROSS.

Path Language

All four of the SQL/JSON Query Functions take at least one "path" as an argument. As far as the SQL grammar is concerned, a path is just a character string constant, enclosed in single quotes like all character string constants. But the SQL/JSON Query Functions interpret these strings as expressions in a "path language," influenced by (but not identical to) the path expressions used in other languages such as JavaScript, XPath, and XQuery. Here's an example of a typical path:

```
persons [ * ] ? (@.location == "Chicago") . hobbies [ * ] ? (@.years >= 3) . hobby
```

A path represents a traversal of a JSON data structure, from an outer level of nesting to an inner level, returning the value(s) found at the end of the path. The path expression above can be interpreted as follows:

`persons`	is the starting point of the path. In this path, persons is an array of objects.
`[ * ]`	means "iterate over all the objects in the array." The path now has multiple branches, all of which are followed independently.
`? (@.location == "Chicago")`	means "keep only those objects whose location value is equal to Chicago."
`. hobbies`	means, "for each surviving object, find the hobbies member, which in this path is another array of objects."
`[ * ]`	means "iterate over all the objects in the array." The path has now branched out again.
`? (@.years >= 3)`	means "keep only those objects whose years value is at least 3."
`. hobby`	means "for each surviving object, return the hobby value."

The result of this path is a set of hobbies that have been pursued by persons in Chicago for at least three years. If you want to know the names of the people who have these hobbies, the path expression will not help you.

In addition to comparison predicates like @.location == "Chicago", the path language provides some other filtering expressions that can be used in the steps of a path. For example, a path step can test whether its value starts with a certain string (example: @.name starts with "Mc") or matches a regular expression (example: @.name like_regex "%student%").

Since path expressions are used in all the SQL/JSON Query Functions, you might say that path expressions do the "heavy lifting" of searching for things inside JSON hierarchies. In this sense, path expressions represent a second language that is embedded inside SQL:2016. The path language has its own syntax and semantics that are different from those of SQL. The path language uses [*] for iteration, whereas SQL uses FROM. The path language uses ? ( ) for filtering, whereas SQL uses WHERE. The path language encloses character strings in double quotes, whereas SQL uses single quotes. Since an entire SQL:2016 path is enclosed in single quotes, any single quotes occurring in the path must be "escaped" (usually denoted by two adjacent single quotes). The equality operator in the path language is ==, whereas in SQL it is =. The path language uses the Boolean operators &&, ||, ! of JavaScript rather than the operators AND, OR, NOT of SQL. The comparison operator = in SQL compares two scalar values, whereas the comparison operator == in the path language can compare two sets, returning `true` if any value in one set is equal to any value in the other set (this is called "existential semantics"). Each of these inconsistencies may seem minor, but in the aggregate they place a significant cognitive burden on the query writer and can be a source of errors.

SQL/JSON Constructor Functions

In SQL, a SELECT clause takes a list of expressions and returns their values as a row of a table. In order for query results to be returned in JSON format, SQL:2016 provides four constructor functions, which are typically used in SELECT clauses. The functions are as follows:

- JSON_OBJECT( ) takes a sequence of (name, value) pairs and assembles them into a JSON object, represented as a character string.

- JSON_OBJECTAGG( ) is an aggregation function that takes a set of table-rows as input and converts them to a JSON object, taking the member-names from one column and the member-values from another column.

- JSON_ARRAY( ) takes a sequence of JSON values and assembles them into a JSON array.

- JSON_ARRAYAGG( ) is an aggregation function. Its input is the sequence of values that are generated by an SQL query block (FROM, WHERE, GROUP BY etc.). JSON-ARRAYAGG( ) assembles this sequence of values into a JSON array. JSON_ARRAYAGG( ) has an optional ORDER BY clause that can impose an ordering on the resulting array. JSON_ARRAYAGG ( ) and JSON_OBJECT( ) are often used together to create a JSON array of objects. We will see this pattern repeated in our example queries.

In contrast with SQL:2016, SQL++ has no need for constructor functions. SQL++ automatically returns query results in JSON notation. Inside a query, to construct a JSON object or array, SQL++ simply uses JSON notation. The following is a valid SQL++ literal expression for an array of two objects:
[{ "one": 1 }, { "two": 2 }]

Examples

We'll begin by examining how JSON can be used to represent a table as an array of similar objects. Table 1 is an ordinary table containing some data about sales, listed by state.

sales	
state	**units**
CA	7500
NY	4000

Table 1

Here's an ordinary SQL query on this table that returns the number of units sold in California (result is 7500).

Q1 (ordinary SQL):

```
    SELECT units
    FROM sales
    WHERE state = 'CA';
```

The data in Table 1 can be represented in JSON using an array of objects, like this:

```
[ { "state": "CA", "units": 7500 }, { "state": "NY", "units": 4000 } ]
```

In a real database, since JSON does not require a schema, some of the objects might have no `units` member, and some might have additional members not shown here. That's not a problem for SQL++ or SQL:2016.

Here's how Q1 would be written in SQL++, operating on the JSON object array shown above:

Q2 (SQL++):

```
    SELECT units
    FROM sales
    WHERE state = 'CA';
```

As you can see, an SQL++ query on an array of objects looks just like an SQL query on a table containing the same data. That's by design.

In many of the following examples, I'm going to use the word AS in the FROM clause to give a short name (sometimes called an "alias") to a table. Then I'll use the alias whenever I refer to a column-value in that table. Aliases are a standard feature of SQL, often used to clarify column references. Applied to Q2, this convention would look like this:

Q3 (SQL++):

```
    SELECT s.units
    FROM sales AS s
    WHERE s.state = 'CA';
```

Aliases are sometimes helpful in understanding a query. In this case, you can think of `sales` as representing the table, and the alias s as representing one row of the table as the query iterates over the rows. Aliases are also helpful in avoiding ambiguities in certain cases, such as when a table is joined to itself. Q3 is valid in SQL++ and also in standard SQL.

In SQL++, the result of a SELECT query is always an array of JSON objects. Q2 and Q3 both return the same result, which looks like this:

```
[ { "units": 7500 } ]
```

Now let's look at how SQL:2016 would handle this query if the data were represented in JSON. Since SQL operates on tables, we'll need to put the data in a table with a JSON column, like this:

sales
data
[{ "state": "CA", "units": 7500 }, { "state": "NY", "units": 4000 }]

Table 2

Of course, in this very simple example we could have made the data look more "relational"—for example, we could have put each of the JSON objects in a separate row. But our goal here is to show how SQL:2016 operates on stored JSON instances, which in general can be complex, often much more complex than this introductory example. This example was chosen to show how SQL:2016 handles arrays of objects, which are commonly found in JSON data.

Here's an SQL:2016 query against the sales data in Table 2:

Q4 (SQL:2016):

```
SELECT JSON_VALUE(s.data,
       'lax $[ * ] ? (@.state == "CA").units'
       RETURNING INTEGER)
FROM sales AS s;
```

Q4 returns the value 7500, wrapped in a table with a single column.

Let's take a closer look at Q4. The SELECT clause contains a call to the JSON_VALUE function, which evaluates a path through a JSON instance and returns a scalar value of type INTEGER. The first operand of JSON_VALUE is `s.data`, which is the starting point of the path (in this case, it references the value of the `data` column, which is a JSON array). The second operand is the path itself, consisting of the following parts:

`lax`	means "if this path is looking for something that isn't there, just return null."
`$ [ * ]`	means "from the starting point of the path, iterate over the array of objects." (In SQL++, this would be a FROM clause.)
`? ( @.state == "CA" )`	means "keep only those objects whose state value is "CA". (In SQL++, this would be a WHERE clause.)
`.units`	means "in each surviving object, return the units value." (In SQL++, this would be a SELECT clause.)

The results of Q3 (the SQL++ version) and Q4 (the SQL:2016 version) are not exactly the same because Q3 returns a JSON object and Q4 returns a scalar value. If a JSON object is desired, it's easy enough to modify Q4 to return one. We just call the function JSON_OBJECT, which constructs an object having a member named `units` with the value computed by JSON_VALUE, as in Q5.

Q5 (SQL:2016):

```
      SELECT JSON_OBJECT ('units':
            JSON_VALUE(s.data,
                  'lax $[ * ] ? (@.state == "CA").units'
                  RETURNING INTEGER))
      FROM sales AS s;
```

Q5 returns its results in JSON notation (much like Q3):

```
{ "units": 7500 }
```

The examples we have seen so far have given a flavor of the two languages, but to compare the languages in depth we will need some more complex examples.

One of the strengths of JSON is its ability to represent hierarchies of nested objects and arrays of objects. Next we'll examine how SQL++ and SQL:2016 can be used to query nested JSON data. To do this, we'll use an example in which sales data is nested inside a larger JSON instance.

Suppose that you are an economist, and that each year you receive, from some research agency, a report on sales of cars and trucks in the United States, organized by brand and state, in JSON format. You're storing these reports in an array with one JSON object for each year. Here's a fragment of this data to illustrate its format.

vehicle_sales

```
[ { "year": 2016,
    "units_sold":
        [ { "brand": "Ford",
              "car_sales": [
                 { "state": "CA", "units": 7500 },
                 { "state": "NY", "units": 4000 }
              ],
              "truck_sales": [
                 { "state": "CA", "units": 2800 },
                 { "state": "NY", "units": 4700 }
              ]
          },
          { "brand": "Honda",
              "car_sales": [
                 { "state": "CA", "units": 3500 },
                 { "state": "NY", "units": 5200 }
              ],
              "truck_sales": [
                 { "state": "CA", "units": 2100 },
                 { "state": "NY", "units": 1900 }
              ]
          }
        ]
  },
  { "year": 2017,
    "units_sold": [...]
  },
  ...
]
```

If we think of a JSON array of objects as representing a table, we might visualize the data shown above as a table named `vehicle_sales`, containing a nested table named `units_sold`, which in turn contains two more nested tables named `car_sales` and `truck_sales`. This way of visualizing the `vehicle_sales` data is illustrated in Table 3, which shows only one row at each level of nesting (of course, an actual database would contain many more rows at each level).

vehicle_sales

year	units_sold				
	brand	car_sales		truck_sales	
		state	units	state	units
2016	Ford	CA	7500	CA	2800

Table 3

Here’s the first query that we’ll write against `vehicle_sales`:

Big-Car-Sales
For the year 2016, find brand and state combinations where at least 5000 cars were sold, and list them in decreasing order by car sales.

Big-Car-Sales is a typical selection-and-projection query that applies filter conditions at two levels: `year` must be 2016, and `units` (inside `car_sales`) must be at least 5000. The query also returns results taken from two different levels: `brand` (inside `units_sold`) and `state` (inside `car_sales`).

To handle queries like this one, which calls for a "flat" result, SQL++ effectively brings the nested rows up to the top level. This is done by a new keyword, UNNEST, that is used in the FROM clause, like this:

```
FROM vehicle_sales AS vs
      UNNEST vs.units_sold AS us
      UNNEST us.car_sales AS cs
```

Here’s how to read this FROM clause: vs represents a row of `vehicle_sales.` us represents a row of `units_sold` that is nested inside row `vs.` cs represents a row of `car_sales` that is nested inside row us. You might think of UNNEST as a kind of join, in which the inner rows are joined to the outer rows in which they are nested. Now all three levels of nested rows have names, and we can apply any filter conditions we like. Here’s the SQL++ version of the Big-Car-Sales query:

Q6 (SQL++):
```
SELECT us.brand, cs.state, cs.units AS cars
FROM vehicle_sales AS vs
      UNNEST vs.units_sold AS us
      UNNEST us.car_sales AS cs
WHERE vs.year = 2016 AND cs.units >= 5000
ORDER BY cars DESC;
```

In Q6, the SELECT clause references variables us and cs that are defined in the FROM clause, which comes later in the query. This is standard SQL usage; however, Q6 might be easier to understand if the SELECT clause appeared in the logical order of query execution, near the end of the query-block, just before ORDER BY. In this case, variables would always be defined before they are used. SQL++ allows (but does not require) you to reorder the SELECT clause in this way. After reordering the clauses, Q6 would look like this:

Q7 (SQL++):
```
      FROM vehicle_sales AS vs
            UNNEST vs.units_sold AS us
            UNNEST us.car_sales AS cs
      WHERE vs.year = 2016 AND cs.units >= 5000
      SELECT us.brand, cs.state, cs.units AS cars
      ORDER BY cars DESC;
```

In Q7, the FROM clause unnests the JSON arrays so we can refer to the three levels of data as `vs`, `us`, and `cs`. The WHERE clause applies our filter conditions: `vs.year` must be 2016 and `cs.units` must be at least 5000. The SELECT clause specifies the output values that we want: `brand`, `state`, and `units`, renaming `units` as cars. The output is delivered as an array of JSON objects, each containing members named `brand`, `state`, and `cars`. The ORDER BY clause imposes an ordering on the resulting array of objects. Here's what the result looks like (in part):

```
[ { "brand": "Ford", "state": "CA", "cars": 7500},
  { "brand": "Honda", "state": "NY", "cars": 5200}
]
```

Using the UNNEST keyword, we have written a query against a JSON hierarchy with three levels of data, filtering at multiple levels and ordering the output, with minimal extensions to SQL. In fact, since UNNEST can be thought of as a kind of join, the UNNEST keyword can be omitted (it's implied whenever an item in a FROM clause references a name (alias) defined earlier in the same FROM clause). The FROM clause of Q7 could have been written like this:

```
      FROM vehicle_sales AS vs, vs.units_sold AS us, us.car_sales AS cs
```

You can also use the keyword FLATTEN rather than UNNEST if you like it better. In the remaining SQL++ examples, we'll use the explicit keyword UNNEST as a kind of comment to remind us what is going on.

Now let's look at how the Big-Car-Sales query might be handled in SQL:2016.

Since SQL:2016 operates only on tables, we'll need to put the `vehicle_sales data` into a real table. We'll do this by storing each year's data in a row with two columns named `year` and `units_sold`. (Of course, there are many ways to put this data into a table—once again, we have chosen a way that preserves most of the JSON structure rather than converting it to relational format). The data that we'll query using SQL:2016 is shown in Table 4.

vehicle_sales	
year	units_sold
2016	[{ "brand": "Ford", "car_sales": [{ "state": "CA", "units": 7500 }, { "state": "NY", "units": 4000 }], "truck_sales": [{ "state": "CA", "units": 2800 }, { "state": "NY", "units": 4700 }] }, { "brand": "Honda", "car_sales": [...], "truck_sales": [...] }]
2017	. . .

Table 4

The path language of SQL:2016 is capable only of tracing a linear path through a JSON hierarchy and returning the data at the end of the path. If you need to do anything more interesting, such as joining, grouping, or returning data from multiple levels, SQL:2016 requires you to turn the JSON hierarchy into a flat table and then operate on the table. SQL:2016 provides a function called JSON_TABLE for this purpose. Here's how JSON_TABLE might be used in the FROM clause of the Big-Car-Sales query:

Q8, partial (SQL:2016):

```
FROM vehicle_sales AS vs,
            JSON_TABLE(vs.units_sold, 'lax $[ * ]'
                  COLUMNS (
                        brand VARCHAR(20) PATH 'lax $.brand',
                        NESTED PATH 'lax $.car_sales[ * ]'
                        COLUMNS (
                              state CHAR(2) PATH 'lax $.state',
                              cars INTEGER PATH 'lax $.units'
                        )
                  )
            ) AS flat
```

The FROM clause of Q8 converts the `units_sold` value in each row of `vehicle_sales` into one row in an SQL table named `flat` that looks like this:

flat		
brand	**state**	**cars**
Ford	CA	7500

Table 5

You might observe that the FROM clause in Q8 is doing essentially the same thing as the FROM clause in Q7 (that's why we named the intermediate table "flat"). However, Q8's FROM clause is more than three times as long, and it depends on the special JSON_TABLE function, whereas the syntax of Q7 should be more familiar to a typical SQL user.

Now that the JSON_QUERY function has converted our JSON data into a flat table, the rest of the Big-Car-Sales query is straightforward. The complete query is shown in Q8, which delivers the result in the form of a table:

Q8 (SQL:2016):

```
SELECT flat.brand, flat.state, flat.cars
FROM vehicle_sales AS vs,
      JSON_TABLE(vs.units_sold, 'lax $[ * ]'
            COLUMNS (
                  brand VARCHAR(20) PATH 'lax $.brand',
                  NESTED PATH 'lax $.car_sales[ * ]'
                  COLUMNS (
                        state CHAR(2) PATH 'lax $.state',
                        cars INTEGER PATH 'lax $.units'
                  )
            )
      ) AS flat
WHERE vs.year = 2016 AND flat.cars >= 5000
ORDER BY cars DESC;
```

The result of Q8 looks just like Table 5, containing a row for each brand and state that has big car sales. But what if we want our results to be delivered in JSON format? In this case, we need to do some more work in the SELECT clause.

The function JSON_OBJECT converts a set of values into a JSON object, and the function JSON_ARRAYAGG can turn a set of objects into an array. JSON_ARRAYAGG and JSON_OBJECT can be used together to convert the rows returned by a query like Q8 into a JSON array of objects. Q9 shows how this technique might be used.

Q9 (SQL:2016):
```
      SELECT JSON_ARRAYAGG (
            JSON_OBJECT (
                 'brand': flat.brand,
                 'state': flat.state,
                 'cars': flat.cars)
            ORDER BY cars DESC )
      FROM vehicle_sales AS vs,
            JSON_TABLE (vs.units_sold, 'lax $[ * ]'
                 COLUMNS (
                      brand VARCHAR(20) PATH 'lax $.brand',
                      NESTED PATH 'lax $.car_sales[ * ]'
                      COLUMNS (
                           state CHAR(2) PATH 'lax $.state',
                           cars INTEGER PATH 'lax $.units'
                      )
                 )
            ) AS flat
      WHERE vs.year = 2016 AND flat.cars >= 5000;
```

Q7 (the SQL++ version) and Q9 (the SQL:2016 version) form an interesting comparison. Q7 looks like pretty much like an ordinary SQL query, with the word UNNEST appearing where a JOIN might be expected, whereas Q9 is about four times as long and depends on three new complex functions (JSON_TABLE, JSON_OBJECT, and JSON_ARRAYAGG). The results of Q7 and Q9 are exactly the same, repeated here:

```
[ { "brand": "Ford", "state": "CA", "cars": 7500},
  { "brand": "Honda", "state": "NY", "cars": 5200}
]
```

Next, we'll examine a slightly more complex query that takes advantage of the fact that our `vehicle_sales` data contains separate values for car sales and truck sales.

Total-California-Units:
List all the brands of vehicles sold in California in 2016, with a total unit count for each brand, including both cars and trucks, in descending order by total units.

We want the result of the Total-California-Units query to be returned in JSON format as shown here:

```
[ { "brand": "Ford", "total_units": 10300 },
  { "brand": "Honda", "total_units": 5600 }
]
```

In SQL++, the Total-California-Units query looks very much like the Big-Car-Sales query. We use a FROM clause that unnests `vehicle_sales` and all its nested tables into one big flat table, and a WHERE clause that identifies the desired subset of data: year is 2016 and state is California. Then we use a SELECT clause to return the brand and total units sold. It's easy to add together the car sales and truck sales because they have been flattened into the same level. The resulting query is shown in Q10, which returns the output shown above. Once again, we have put the SELECT clause at the end of the query-block.

Q10 (SQL++):

```
      FROM vehicle_sales AS vs
            UNNEST vs.units_sold AS us
            UNNEST us.car_sales AS cs
            UNNEST us.truck_sales AS ts
      WHERE vs.year = 2016 AND cs.state = 'CA' AND ts.state = 'CA'
      SELECT us.brand, cs.units + ts.units AS total_units
      ORDER BY total_units DESC;
```

Now we'll take a look at how the Total-California-Units query might be written in SQL:2016. The SQL:2016 version is shown in Q11.

Q11 (SQL:2016):

```
      SELECT JSON_ARRAYAGG (
            JSON_OBJECT (
                  'brand': flat_cars.brand,
                  'total_units': flat_cars.units + flat_trucks.units )
            ORDER BY flat_cars.units + flat_trucks.units DESC)
      FROM vehicle_sales AS vs,
            JSON_TABLE (vs.units_sold, 'lax $[ * ]'
                  COLUMNS ( brand VARCHAR(20) PATH 'lax $.brand',
                        NESTED PATH 'lax $.car_sales[*] ?(@.state == "CA")'
                              COLUMNS ( units INTEGER PATH 'lax $.units' ) )
            ) as flat_cars
      JOIN
            JSON_TABLE (vs.units_sold, 'lax $[ * ]'
                  COLUMNS (brand VARCHAR(20) PATH 'lax $.brand',
                        NESTED PATH 'lax $.truck_sales[*] ?(@.state == "CA")'
                              COLUMNS ( units INTEGER PATH 'lax $.units' ) )
            ) as flat_trucks
      ON flat_cars.brand = flat_trucks.brand
      WHERE vs.year = 2016;
```

We'll explain this query by working from the inside out. Q11 contains two calls to the JSON_TABLE function. One of them flattens the `car_sales` data into an intermediate table named `flat_cars` containing columns `brand` and `units`, and including only car sales in California. The other JSON_TABLE call flattens `truck_sales` data into an intermediate table named `flat_trucks`, also containing columns `brand` and `units`, restricted to truck sales in California. The query then uses an SQL JOIN operation to join the two intermediate tables on their brand columns to make sure that each row of the joined table applies to a specific brand. The WHERE clause limits the query to data from 2016. Finally the SELECT clause adds the car and truck sales together to get `total_units`, and returns the result as an array of JSON objects. The result of Q11 (the SQL:2016 version) is exactly the same as that of Q10 (the SQL++ version).

Comparing the two versions of the Total-California-Units query, we see that the SQL++ version is similar to an ordinary SQL query, whereas the SQL:2016 version is about three times as long and contains several calls to special functions, and some path expressions that are quite unlike ordinary SQL.

In addition to UNNEST, SQL++ provides the following small syntactic extensions to SQL:

- SELECT VALUE is used to return a value that is not wrapped in a JSON object, as in SELECT VALUE 2 + 2, which returns the value 4.

- GROUP AS is a feature that provides access to values both before and after grouping. GROUP AS is used in queries that perform complex transforms such as inverting a JSON hierarchy.

- A LET clause can be used in any query-block to bind a variable to a computed value. This is often helpful to avoid repeating common subexpressions, as in LET pay = salary + bonus.

A complete discussion of these syntactic extensions, with examples, can be found in [9].

Analysis

As we've seen, SQL++ and SQL:2016 are designed for quite different environments. SQL++ is "JSON in, JSON out" whereas SQL:2016 is "tables in, table out." Nevertheless, both languages have a common purpose: to query or transform JSON instances. As a result, the two languages have some common aspects and some differences.

Common aspects between SQL++ and SQL:2016 include these:

- Since JSON data may be schemaless, both languages must deal with the possibility that a query may search for a data value that is not present. SQL:2016 allows each path expression to specify a "strict" or "lax" mode. Strict mode generates errors when searched-for values are not present, whereas lax mode simply returns null. Lax mode also suppresses certain other structural errors, for example by treating a singleton array the same as a scalar value. SQL++, on the other hand, is a parameterized language [1, 2] that permits a particular implementation of SQL++ to specify how it deals with various structural errors or missing items. N1QL sets these parameters in a way that is similar (but not identical) to the lax mode of SQL:2016.

- Following JavaScript convention, both languages use zero-based arrays (unlike SQL arrays, which are one-based.)

- Since JSON has no date/time type, both languages provide functions that assign date/time semantics to character strings in a designated format (e.g, `datetime()` in SQL:2016, `timestamp()` in SQL++). In addition, N1QL provides about three dozen functions for manipulating dates and times (extracting a component from a date, finding the difference between two dates, etc.).

The most important difference between SQL++ and SQL:2016 is that SQL++ is one uniform, consistent language, syntactically very similar to SQL. SQL++ brings the operations of SQL (joining, grouping, filtering, etc.) to bear directly on JSON data. SQL:2016, on the other hand, is not one language but two languages operating at different levels of abstraction: SQL for tables and a separate path language for JSON, invoked by special functions. As far as the SQL operations are concerned, JSON instances, and the path expressions that access them, are simply character strings.

As we have seen in the language descriptions, the syntax and semantics of the SQL:2016 path language are quite different from those of SQL, in the handling of iteration, filter predicates, quoted strings, comparison operators, Boolean operators, and other aspects. More importantly, the query power of a path expression is very limited. A path expression can only navigate inward through nested JSON arrays, possibly applying a filter condition at each level. It can return only values found at the end of the path, not intermediate values. Because of this limitation, almost every nontrivial SQL:2016 query must first convert the JSON data into one or more tables, then query it using SQL functionality, then (if desired) convert it back into JSON again. These conversion steps make queries difficult to write and maintain, and may interfere with global optimization.

One important difference between SQL++ and SQL:2016 lies in their handling of missing data. Since JSON data often lacks a schema, it is quite common for a query expression to return an empty value (for example, by trying to access a member that does not exist in a particular object.) In SQL:2016, in lax mode, cases like this are quickly converted into null values. SQL++, on the other hand, distinguishes between MISSING values and NULL values, and carefully propagates this distinction throughout query processing. In generating the final output of a query, object members whose values are MISSING simply disappear rather than being assigned a null value. In this respect, SQL++ correctly reflects the semantics of JSON, in which a null value is not equivalent to a missing value. This point is illustrated by the following example.

Suppose that I have an array of objects named t that looks like this:

```
[ { "a": 1, "b": 1, "c": 1}, {"a": 2, "b": null, "c": 2}, {"a": 3, "c": 3} ]
```

Here's an SQL++ query that performs a simple projection of these objects onto their "a" and "b" members:

Q12 (SQL++):
```
      SELECT a, b
      FROM t;
```

Here's the result of Q12:

```
[ {"a": 1, "b": 1}, {"a": 2, "b": null}, {"a": 3} ]
```

Note that the "b" member of the second object has the value null, and the "b" member of the third object does not exist. This is correct according to JSON semantics.

To see how SQL:2016 would handle an analogous query, suppose that the same data is organized into a three-row table, as shown in Table 6:

t
j
{ "a": 1, "b": 1, "c": 1}
{"a": 2, "b": null, "c": 2}
{"a": 3, "c": 3}

Table 6

Here's an SQL:2016 query that does the same projection (compare the length and clarity of this query with the SQL++ version):

Q13 (SQL:2016):

```
SELECT JSON_OBJECT (
      'a' : JSON_VALUE (t.j, ‘lax $.a' RETURNING INTEGER),
      'b' : JSON_VALUE (t.j, 'lax $.b' RETURNING INTEGER) )
FROM t;
```

The result of Q13 is shown in Table 7:

t
j
{ "a": 1, "b": 1}
{"a": 2, "b": null}
{"a": 3, "b": null}

Table 7

SQL:2016 converts the missing b-value to a null. This result does not conform to JSON semantics, because null is a value in JSON. The output of SQL:2016 for this query does not distinguish between a member that does not exist and a member that has the value null.

The above analysis is based on SQL:2016 "lax" mode with the default JSON_VALUE behavior, NULL ON EMPTY. JSON_VALUE also provides another option in which missing data is replaced by some non-null value. For example, the JSON_VALUE functions could specify "empty" ON EMPTY. But in this case there would be no way to tell whether "empty" in a query result denotes a member that is absent or a member that exists and has the value "empty." Of course, this issue would apply regardless of the value chosen to represent absent data.

Additional differences between SQL++ and SQL:2016 include these:

- SQL:2016 uses functions like JSON_OBJECT and JSON_ARRAY to construct JSON instances, whereas SQL++ simply uses native JSON notation.

- SQL:2016 treats all JSON arrays as having a fixed order, whereas SQL++ has a concept of an unordered JSON collection, which provides additional opportunities for optimization.

- SQL:2016 has no definition for equality of JSON objects or arrays (SQL++ uses a deep-equals definition in each case).

In summary, let's consider two scenarios in which the advantages of SQL++ and SQL:2016 might be compared.

In the first scenario, you have an existing application with lots of data stored in tables and lots of SQL code for maintaining and querying your data. You need to extend your application to include some data in JSON format. You need to store JSON instances, and sometimes you need to extract simple values from the JSON instances, to compare or combine them with your relational data values. You may think of the JSON instances as "documents associated with your data." In this scenario, SQL:2016 provides a way to add JSON data with minimum impact on your existing tables. Your existing SQL queries will run on the tables as before. To access your JSON data, you will need to learn a new and different "path" language and you will need to get familiar with some functions like JSON_TABLE and JSON_ARRAYAGG whose purpose is to bridge the gap between the SQL world and the JSON world. Simple queries on JSON data will be fairly straightforward, although there may be some confusion when missing values are converted to nulls. Complex queries or transforms on JSON data will give you a headache, but they will not be impossible.

In the second scenario, you are developing a new web-based application, written in JavaScript. Your application does not require enforcement of a rigid schema. JSON is the natural data format for your application. In this scenario, you do not think of JSON as "documents associated with your data"—instead, JSON *is* your data. SQL++ provides you with a single, consistent language that operates on JSON data directly without converting it to or from relational data. At the same time, it leverages your knowledge and experience with SQL, applying familiar SQL syntax to a more flexible data format. Your queries will be shorter by about a factor of three compared with the equivalent queries in SQL:2016, and they will be faster to develop, easier to maintain, and possibly more efficient to execute. For these reasons, SQL++ offers some important advantages over SQL:2016 in this scenario.

Acknowledgment

I am grateful to Jan-Eike Michels, a member of the working group that designed SQL:2016, for answering questions related to SQL/JSON and for his helpful review of an early draft of this paper.

References

[1] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ Query Language: Configurable, Unifying and Semi-structured. See https://arxiv.org/abs/1405.3631.

[2] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL, and NewSQL Databases. See http://db.ucsd.edu/wp-content/uploads/pdfs/375.pdf.

[3] Sattam Alsubaiee et al. AsterixDB: A Scalable, Open Source BDMS. PVLDB Vol. 7, No. 14, October 2014, pp. 1905-1916. See also http://asterixdb.apache.org/

[4] Michael Carey. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. Proc. of the 35th IEEE Int'l. Conf. On Data Engineering, Macau, China, pp. 1–12, April 8-11, 2019.

[5] N1QL For Analytics Language Reference. See https://docs.couchbase.com/server/current/analytics/1_intro.html

[6] Murtadha Al Hubail et al. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. PVLDB, Vol. 12, No. 12, August 2019, to appear.

[7] Amazon, Inc. PartiQL: SQL-compatible access to relational, semi-structured, and nested data. See https://partiql.org.

[8] Information technology - Database languages - SQL Technical Reports - Part 6: SQL Support for JavaScript Object Notation (JSON). ISO/IEC TR 19075-6:2017. Available at https://standards.iso.org/ittf/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip.

[9] Don Chamberlin. SQL++ for SQL Users: A Tutorial. ISBN 978-0-692-18450-9, published by Couchbase, Inc. Available at https://www.amazon.com/dp/0692184503.