Datenbank Spektrum (2017) 17:277–287
https://doi.org/10.1007/s13222-017-0267-4

FACHBEITRAG

SQL/JSON Standard: Properties and Deficiencies

Dušan Petković[1]

Received: 27 June 2017 / Accepted: 9 September 2017 / Published online: 24 October 2017

Abstract Recently, a new era of application development is emerging, which is based upon the ease of access to modern compute resources, such as mobile devices. This access can be supported using JSON (Java Script Object Notation). Therefore, the support of storage and query access for JSON documents in the context of relational DBMSs is necessary. For this reason, the SQL standardization committee published a proposal called SQL/JSON. In this paper we discuss the JSON features specified in the proposal and show to what extent different relational database systems have integrated them.

At the end of the paper we describe the main drawbacks of the proposal and the ways to solve them. From our point of view, the following should be specified in one of the future proposals of SQL/JSON: JSON documents should be first-class objects in SQL (native storage). Handling JSON documents as first-class objects in SQL would provide the potential for greater capability for users and for better performance. The support of modification of parts of a JSON document using the SQL UPDATE statement is necessary. Direct access of external JSON data should be supported, too.

1 Introduction

JSON (JavaScript Object Notation) is a simple data format used for data interchange. The structure of JSON content follows the syntax structure for JavaScript. Generally, a JSON string contains either an array of values or an object, which is a list of name/value pairs. An array is surrounded by a pair of square brackets and contains a comma-separated list of values. An object is surrounded by a pair of curly brackets and contains a comma-separated list of name/value pairs. A name/value pair consists of a field name (in double quotes), followed by a colon (:), followed by the field value (in double quotes).

Fig. 1 shows a group of JSON documents with the name *info*, which describe persons. The name of the person, her affiliation and her friends are described. The corresponding names ("who", "friends" and "where") are called properties. For instance, a single person, Fred, works for Microsoft and has friends Lili and Hank.

1.1 SQL/JSON Standard

The SQL/JSON standard is published as "change proposal" in March, 2014 in two documents. The first part of the change proposal [14] provides an introduction to JSON and discusses SQL operators, which can be used to construct JSON documents stored as character or binary strings. The second part specifies how JSON data can be queried in SQL [15].

The members of the committee based their work on the following objectives:

- The language should be minimal
- JSON should be handled using built-in functions
- The query language should be exclusively designed for this purpose

✉ Dušan Petković
petkovic@fh-rosenheim.de

[1] University of Applied Sciences,
Hochschulstr. 1, 83024 Rosenheim, Germany

```
{"info":{"who": "Fred" ,"where": "Microsoft" ,
   "friends":[{"name":"Lili","rank":5}, {"name":"Hank","rank":7}]}}
{"info":{"who": "Tom", "where": "IBM", "friends": [ { "name":
   "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] }}
{"info":{"who":"Jack", "friends": [ { "name": "Connie" } ] }}
{"info":{"who":"Joe","friends":[{"name":"Doris"},{"rank":1}]}}
{"info": {"who":"Mabel","where":"PostgresSQL","friends":[{"name":"Buck","rank": 6}]}}
{"info":{"who": "Louise", "where": "Hanna" }}
```

Fig. 1 A group of JSON documents (stored in the *ora_json_table* table)

1.2 Why Integration of JSON in RDBMSs is Necessary?

There are several reasons why it is necessary to integrate JSON in relational database systems:

- Storage of semi-structured data
- Databases provide reduced administrative costs
- Increased developer productivity

Relational tables contain structured data. The advantage of JSON is that it can contain both structured and semi-structured data. By supporting storage of JSON documents, a relational database extends its capabilities and integrates structured and semi-structured data together.

When JSON objects are stored individually and are used for separate programs, each program has to administrate its own data. In case of JSON support through a relational database system, the system manages all data. The same is true for security and transaction management, because the system takes over the management of security and transaction processing.

Integrating JSON in relational DBMSs increases productivity, because the database system supports a lot of tasks that otherwise must be implemented by programmers.

1.3 Integration Features Discussed

The grouping of the SQL/JSON features in this paper is different than in the SQL/JSON standard. The reason is that integration of data exchange formats, such as JSON and XML in relational DBMSs can be generally discussed in relation to the following questions:

- How data are stored in tables?
- How data are presented in relational form?
- How relational data can be published in the particular format?
- How data is queried?
- Which indexing techniques can be applied to that data?

Note that indexing techniques are not an issue in the SQL standardization process. For this reason, we will not discuss it further.

1.4 Roadmap

The rest of the paper is organized as follows. Sect. 2 discusses general features described in the SQL/JSON standard. Sect. 3 describes the specification of the storage of JSON data in relational tables, while Sect. 4 explains how JSON data can be presented in relational form. The next section discusses how the standard solved a problem of publishing of content of relational tables as JSON. Sect. 6 describes the SQL/JSON query functions and gives several examples, how they can be used. Sect. 7 gives a concise description of the support of SQL/JSON in IBM DB2, Oracle and SQL Server. At the end, in the summary, we discuss the deficiencies of the proposal, and the ways how they could be improved in one of future versions of the SQL/JSON standard.

1.5 Related Work

The SQL/JSON proposal is published in two papers [14, 15]. The former describes general properties of the SQL/JSON specification, while the latter discusses SQL/JSON query functions.

Besides IBM DB2, Oracle and SQL Server, which are described in Sect. 7, PostgreSQL also supports integration of JSON functions in its database server, but the entire implementation does not have any resemblance to the SQL/JSON standard [10, 11].

Vendors of NoSQL data stores have used JSON as a data format for the document object model. Another approach is to add a middleware between JSON and the corresponding database system. An example of such a system is Sinew [13].

The topic of several articles is JSON integration in RDBMSs. Liu et al. [5] present three architectural principles for schema-less development within RDBMSs. The

same author investigates, how JSON data model can be added to a RDBMS [6]. The similar idea, but discussed from the different aspect is shown in [4].

2 SQL/JSON: General Features

In this section we will describe several general topics from the SQL/JSON standard, which are applied to all (or several) SQL/JSON functions described in the rest of this paper:

- *Lax* and *strict* modes
- JSON input and output clauses
- IS JSON predicate
- SQL NULLs, JSON nulls and SQL/JSON nulls

2.1 Lax and Strict Modes

JSON documents can have a "sloppy" structure, meaning that several JSON values could be omitted in a document. (JSON value could be an object, array, number, string, or one of the literals.) For this reason, SQL/JSON introduces two modes: *lax* and *strict*. The motivation to introduce these modes is that the *strict* mode should be used to examine data from a strict schema perspective, i. e. when the document diverges from the schema. Therefore, the *strict* mode raises an error if the data does not strictly adhere to the requirements of the corresponding path expression. (See also the description of error handling at the beginning of Sect. 6.) On the other hand, the *lax* mode converts errors to an empty SQL/JSON sequence. (see Examples 9 and 10 show the use of both modes.)

Note that the standard does not specify which mode is the default mode. In other words, the keywords *lax* and *strict* are mandatory syntax in the specification. This allows implementations to choose their own default value.

In addition, the *lax* mode handles a JSON array of size one as a singleton, and vice versa. In other words, if an operation requires an array but the operand is not an array, then the operand is implicitly "wrapped" in an array. Also, if an operation requires a non-array, but the operand is an array, then the operand is implicitly "unwrapped" into a sequence. For this reason, SQL/JSON introduces several options, which handle wrapping and unwrapping of an array. The corresponding options will be discussed in Sect. 6.3.

2.2 JSON Input and Output Clauses

To use JSON meaningfully, it is necessary to include syntax for specifying that input arguments of an SQL/JSON function are JSON values (instead of ordinary values of a relational table). In the same way, the control of the output of JSON data should be supported. For this reason, SQL/JSON introduces two clauses: JSON input and JSON output clause.

The syntax for the input clause (see also Example 3) is as following:

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

where <JSON input representation> is implementation defined representation of JSON. (Two well-known JSON representation are BSON [3] and Avro [1]).

The syntax for the output clause (see also Example 9) is:

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

where <JSON output representation> specifies the encoding format for the output value.

2.3 IS JSON Predicate

Before JSON data are integrated in a relational database, it is recommended to check whether the data in the document fulfills all syntax rules. The SQL/JSON standard specifies the IS JSON predicate to check this constraint. (The JSON document, which fulfills the given syntax rules, is called *well-formed*.)

Additionally, SQL/JSON does not specify whether field names of a JSON object must be unique for that object. This means that a well-formed JSON object can have multiple members with the same field name. For this reason, the standard introduces the WITH UNIQUE KEYS constraint, to determine that object fields must be unique for the particular JSON object. That way, JSON data will be considered as well-formed only if none of its objects have duplicate field names.

On the other hand, the specification of the keywords WITHOUT UNIQUE KEYS means that objects of the corresponding JSON document can have duplicate field names and still be considered as well-formed.

2.4 SQL NULLs, JSON nulls and SQL/JSON nulls

JSON has a value "null". Unlike SQL NULLs, this value stands alone in its own type. When a JSON text is parsed to move it into the SQL/JSON model, a JSON null is converted to an "SQL/JSON null". This is a value used by the database server, but it is not an SQL NULL, because there is no SQL data type information associated with it. Even though a JSON null or its internal representation ("SQL/JSON null") is not an SQL NULL, there are cases, in which

they are converted from one to another. In this paper, we will use the phrases, "SQL NULLs", "JSON nulls" and "SQL/JSON nulls", depending on the context.

3 Storing JSON Documents

Generally, there are three different ways in which data presented in a particular format can be stored in relational form:

- As raw documents
- Decomposed into relational columns
- Using native storage

In case that JSON documents are stored using either VARCHAR, CLOB or BLOB data type, an exact copy of the data is stored. In this case, JSON documents are stored "raw"—that is, in their character or binary string form. The raw form allows insertion of JSON documents in an easy way. The retrieval of such a document is very efficient if the entire document is retrieved. To retrieve parts of the documents, special types of indices are helpful.

To decompose a document into separate columns of one or more tables, its schema is used. In this case, the hierarchical structure of the document is preserved, while order among elements is ignored. Note that storing JSON documents in decomposed form can be applied in rare cases, where the corresponding schema exists.

Native storage means that JSON documents are stored in their parsed form. In other words, the document is stored in an internal representation that preserves the content of the data. Using native storage makes it easy to query information based on the structure of the document. On the other hand, reconstructing the original form of the document is difficult, because the created content may not be an exact copy of the document.

The specified solution in the proposal is a "light-weight" one, meaning that JSON documents are specified in a relational table using SQL standard data types, such as VARCHAR, BLOB and CLOB. Therefore, this solution corresponds to the form of storing documents in a "row" form. As the SQL standardization committee members quote, the primary reason for taking this approach is to improve the chances of its quick adoption into the SQL standard, as well as the rapid implementation by vendors of relational database systems.

4 Projecting JSON Documents in Relational Form

There are two different issues concerning presentation of JSON: JSON documents can be projected in relational form and relational data stored in a table can be published as JSON documents. (The former will be discussed in this section, while the latter is topic of Sect. 5.) One common reason for projecting JSON documents in relational form is that existing legacy applications, packaged business applications or reporting software do not always support JSON format. In that case, it is useful to convert JSON documents into rows and columns of relational tables.

The SQL/JSON standard introduces the JSON_TABLE function to project JSON data in relational form. As all other SQL/JSON functions, this one uses SQL/JSON path language, to specify the part of the document, which should be projected. This means that the function uses a row pattern to describe the rows that are selected from a JSON value, and column patterns to describe the columns.

The proposal supports different clauses, which allow users to project JSON documents in the simple form (i. e. with no nesting) as well as with nesting.

Example 1

```
SELECT jt.* FROM   ora_json_table,
     JSON_TABLE(person_and_friends, 'lax $.info'
       COLUMNS("Person"    VARCHAR2(20) PATH 'lax $.who',
                "Affiliation" VARCHAR2(20) PATH 'lax $.where')) AS jt;

PERSON               AFFILIATION
-------------------- --------------------
Fred                 Microsoft
Tom                  IBM
Jack
Joe
Mabel                PostgresSQL
Louise               Hanna
```

The COLUMNS clause of the JSON_TABLE function allows users to explicitly define how the output table looks like. The names of particular columns can be specified, too. The path expression in the PATH option uses the SQL/JSON path language to specify which part of the JSON document should be projected to the given column name. (The syntax of the SQL/JSON path language can be found in [15].)

The previous example shows the way to project simple data with no nesting. If the nested data should be displayed, the NESTED PATH clause must be used.

Example 2

```
SELECT jt.* FROM   ora_json_table,
     JSON_TABLE(person_and_friends, 'lax $.info'
       COLUMNS ( "Person" VARCHAR(20) PATH 'lax $.who',
       NESTED PATH 'lax $.friends[*]'
       COLUMNS  ( "name" VARCHAR2(20) PATH 'lax $.name',
              "rank" VARCHAR2(20) PATH 'lax $.rank'))) as jt;
```

PERSON	NAME	RANK
Fred	Lili	5
Fred	Hank	7
Tom	Sharon	2
Tom	Monty	3
Jack	Connie	
Joe	Doris	
Joe		1
Mabel	Jack	6
Louise		

Generally, the NESTED PATH clause:

- Supports the nested COLUMNS clause to allow the nesting within the data
- Provides syntax to support both inner and outer join cases between parent and child COLUMNS clauses
- Allows more than one nested COLUMNS clause at any depth

The proposal contains two other clauses:

- FOR ORDINALITY clause
- PLAN clause

The FOR ORDINALITY clause specifies the ordinality column. The ordinality column allows sequential numbering of rows, starting with 1. Therefore, this clause is similar to the SQL analytic function called ROW_NUMBER. The PLAN clause is used to express the desired output plan.

5 Publishing Relational Data as JSON

The main reason to publish relational data as JSON documents is Internet. JSON is a format, which is generally used in Web applications. If available data is given in relational form, but should be used for Web applications, publishing these data as JSON documents is an established method. The group of four standardized SQL/JSON functions:

- JSON_OBJECT
- JSON_OBJECTAGG
- JSON_ARRAY
- JSON_ARRAYAGG

can be used for this purpose. (These four functions are called constructor functions, because they can be used to construct JSON objects or JSON arrays.)

5.1 JSON_OBJECT

The construction of new JSON objects is necessary, mainly to use them within the corresponding application. The SQL/JSON standard proposes a built-in function called JSON_OBJECT that constructs JSON objects from explicit name/value pairs; the names in those name/value pairs must be tables' columns, while the values may be specified as SQL literals or as any other SQL expressions, including subqueries.

Example 3

```
-- create table and insert two rows
CREATE TABLE department (dept_no INT,
            dept_name CHAR(20), location CHAR(20));
INSERT INTO department VALUES (1, 'Marketing', 'Seattle');
INSERT INTO department VALUES (2, 'Production', 'Boston');
-- json_object()
SELECT JSON_OBJECT
   ( 'deptno' : d.dept_no FORMAT JSON, 'deptname' : d.dept_name
     FORMAT JSON ) AS departments
     FROM deparment AS d;
Output: {“deptno”: 1,  “deptname”: "Marketing" }
        {“deptno”: 2,  “deptname”: "Production" }
```

The query in Example 3 returns one row for each department selected in the SELECT statement. That row contains a single column, which contains a serialization of a JSON object having the department number and name.

5.2 JSON_OBJECTAGG

JSON documents are usually schema-less. For this reason, it is not always possible to construct a JSON object by explicitly specifying the names of the contained name/value pairs. In such a case, it may be desirable to construct a JSON object as an aggregation of information in a relational table. The SQL/JSON standard specifies the JSON_OBJECTAGG functions to perform this functionality. The output of Example 4 is a single row of one column, which contains a serialization of the corresponding JSON object.

Example 4

```
SELECT JSON_OBJECTAGG (dept_name, dept_no )
       from department;
Output:  {"Marketing":1, "Production":2}
```

Additionally, the JSON_OBJECTAGG function can be used to group parts of a relational table according to the values in one or more columns.

Example 5

```
SELECT d.dept_no, JSON_OBJECTAGG (dept_name, dept_no)
    FROM department  AS d
   GROUP BY d.dept_no;
```

5.3 JSON_ARRAY and JSON_ARRAYAGG

Analogously to the JSON_OBJECT function, SQL/JSON specifies the JSON_ARRAY function. This function constructs a JSON array, each element of which is taken from the rows selected in the query. JSON_ARRAY comes in two flavours: One produces the result from an explicit list of values. The other produces its results from a query expression invoked within the function.

Example 6

```
SELECT JSON_ARRAY (dept_no, dept_name)
    FROM department;
Output:
[1,"Marketing"]
[2,"Production"]
```

This query returns one row for each department recorded in the table; that row contains a single column that contains a serialization of a JSON array having the department number and name.

The last publishing function, JSON_ARRAYAGG, constructs a JSON array as an aggregate, similar to JSON_OBJECTAGG. This function supports an optional ORDER BY clause that allows the result of the query to be ordered, before the selected data is extracted to be placed in the corresponding JSON array.

6 Querying JSON

Before we start to discuss SQL/JSON query functions, we will explain how the standard handles errors concerning path expressions. (While all SQL/JSON query functions use path expressions, error handling can be specified in function's path expressions).

Note that the SQL/JSON standard groups errors in two groups: structural and non-structural. Structural errors concern the structure of the document; for example, accessing an array element or an object member that does not exist. Non-structural ones are conventional errors, such as division by zero. In this paper we will discuss only structural errors.

Handling structural errors in a path expression depends whether the *lax* or *strict* mode is specified. In the *lax* mode, structural errors are converted to an empty SQL/JSON sequence, by default, and are handled by the following forms of the ON EMPTY clause:

- NULL ON EMPTY—returns NULL (the default behavior)
- ERROR ON EMPTY—raise an error
- DEFAULT "string" on EMPTY—displays the given string
- EMPTY ARRAY ON EMPTY—displays empty array
- EMPTY OBJECT ON EMPTY—displays empty object

If a path expression is specified in the *strict* mode, the structural errors become "hard" errors, meaning that reported errors are sent back to the invoking routine. To control "hard" errors, the standard specifies the ON ERROR clause, which has the following forms:

- ERROR ON ERROR—Raise the error
- NULL ON ERROR—Return null instead of raising the error (the default behavior)
- TRUE ON ERROR—Return true instead of raising the error. This form of the clause is available only for the JSON_EXISTS function (see Example 7).
- FALSE ON ERROR—Return false instead of raising the error. This form of the clause is available only for the JSON_EXISTS function.
- EMPTY ON ERROR—Return an empty array ([]) instead of raising the error. This form of the clause is available only for the JSON_QUERY function.
- DEFAULT "string" ON ERROR—Return the specified string instead of raising the error (see Example 10).
- EMPTY ARRAY ON ERROR—displays empty array
- EMPTY OBJECT ON ERROR—displays empty object

The standard specifies the following functions (conditions), which are used to query JSON documents:

```
{"who": "Fred" ,"where": "Microsoft" ,
    "friends":[{"name":"Lili","rank":5}, {"name":"Hank","rank":7}]}
{"who": "Tom", "where": "IBM", "friends": [ { "name":
   "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] }
{"who":"Jack", "friends": [ { "name": "Connie" } ] }
{"who":"Joe","friends":[{"name":"Doris"},{"rank":1}]}
{"who":"Mabel","where":"PostgresSQL","friends":[{"name":"Jack","rank": 6}]}
{"who": "Louise", "where": "Hanna" }
```

Fig. 2 Slightly modified document from Fig. 1 (stored in the *json_table* table)

- JSON_EXISTS
- JSON_VALUE
- JSON_QUERY

We will use the JSON document shown in Fig. 2 as a running document in the rest of the paper.

6.1 JSON_EXISTS

The JSON_EXISTS condition takes a path expression and checks if such path selects one or multiple values in the JSON documents. Therefore, with this condition the user can find the rows in which the JSON documents satisfy a given predicate. The condition returns *true* if the JSON value exists and *false* if the JSON value does not exist. (This also means that *false* will be returned, if the value is not a well-formed JSON document.)

```
Example 7
SELECT person_and_friends
  FROM json_table
  WHERE JSON_EXISTS (person_and_friends, 'strict $.friends' );
SELECT person_and_friends
  FROM json_table
  WHERE JSON_EXISTS (person_and_friends, 'strict $.friends'
  TRUE ON ERROR);
```

The first statement in Example 7 queries the table and returns JSON data that consists of all objects which contain the property called “friends”. In this case, the ON ERROR clause is not specified. Therefore, the JSON_EXISTS condition returns *false* for values that are not well-formed JSON documents. (While the *json_table* table contains only well-formed documents, this condition returns *true* for all documents stored in the table, and displays all of them.)

To explain the second statement in Example 7, let us suppose that the *json_table* table contains one more row, for instance the string: “This is not a well-formed JSON document”. Because the TRUE ON ERROR clause is specified in the second query, the JSON_EXISTS condition returns *true* also for values that are not well-formed JSON data. Therefore, the string above will be displayed in the result, too.

6.2 JSON_VALUE

The JSON_VALUE function extracts a scalar value from a JSON document. The function has two arguments: *expression* and *path*, where *expression* is the name of a column that contains JSON data and *path* is a SQL/JSON path expression that specifies the property to extract.

```
Example 8
SELECT JSON_VALUE(person_and_friends,'lax $.where')  AS company
   FROM json_table
    WHERE JSON_VALUE(person_and_friends, 'lax $.who') = 'Fred';
Output: Microsoft
```

The following two examples show error handling for the *strict* and *lax* modes.

```
Example 9
SELECT JSON_VALUE(person_and_friends,'strict $.where'
            RETURNING VARCHAR(25)
            NULL ON ERROR)
             AS company  FROM json_table;

COMPANY
---------------------------------
Microsoft
IBM


PostgresSQL
Hanna
```

Example 9 shows the use of the *strict* mode. In this case, the explicit error handling is specified using the ON ERROR clause (NULL ON ERROR), meaning that in error case SQL NULL is displayed.

In the case of the *lax* mode, one of the forms of the ON EMPTY clause is used for error handling.

```
Example 10
SELECT JSON_VALUE(person_and_friends, 'lax $.where'
                        RETURNING VARCHAR(25)
                         DEFAULT 'empty' ON EMPTY)
            AS company  FROM json_table;

COMPANY
---------------------------------
Microsoft
IBM
empty
empty
PostgresSQL
Hanna
```

The RETURNING clause in Example 10 specifies the data type of the value returned by the function. Users can apply either the RETURNING clause or the PASSING clause. The RETURNING clause returns the result to the calling program, while the PASSING clause is used to pass additional parameters to the SQL/JSON path expression.

6.3 JSON_QUERY

The JSON_QUERY function returns extracts of an object or an array from a JSON document. The syntax of this function is analogous to the syntax of the JSON_VALUE function.

```
Example 11
SELECT  JSON_VALUE (T.person_and_friends, 'lax $.who') AS Who,
     JSON_QUERY (T.person_and_friends, 'lax $.friends' ) AS Friends
  FROM json_table T
  WHERE JSON_EXISTS (t.person_and_friends, 'lax $.friends');
Output:
Who     Friends                                                    .
Fred    [{"name":"Lili","rank":5},{"name":"Hank","rank":7}]
Tom     [{"name":"Sharon","rank":2},{"name":"Monty","rank":3}]
etc.
```

Example 11 shows the use of JSON_VALUE, JSON_QUERY and JSON_EXISTS. The first function displays

a scalar value, the second one an array (with several objects), while the JSON_EXISTS condition is used to choose the rows, which fulfil the given condition.

The SQL/JSON standard introduces another clause, WITH ARRAY WRAPPER, to handle the case, where the result of the JSON_QUERY function is a scalar rather than an array or object. This clause wraps the results using an array wrapper.

Example 12
```
SELECT JSON_QUERY(person_and_friends, 'lax $.friends.name'
                  WITH ARRAY WRAPPER)
                AS company  FROM json_table;
Output:
    ["Lili","Hank"]
    ["Sharon","Monty"]
    ["Connie"]
    ["Doris"]
    ["Jack"]
    []
```

Note that if the array wrapper is applied to the empty sequence, it produces an empty array. The alternative to WITH ARRAY WRAPPER is WITHOUT ARRAY WRAPPER, which is the default value. Additionally, WITH ARRAY WRAPPER has two different forms: WITH UNCONDITIONAL ARRAY WRAPPER and WITH CONDITIONAL ARRAY WRAPPER. The difference is that CONDITIONAL supplies the array wrapper if the result of path expression is anything other than a singleton SQL/JSON array or object. (The default is UNCONDITIONAL.)

7 SQL/JSON: Implementations

In the meantime, several relational DBMSs have implemented parts of the SQL/JSON standard. In this section we will give a concise description, which parts of the standard are implemented in the following systems:

- IBM DB2
- Oracle
- SQL Server

The detailed description of these implementations can be found in [9].

7.1 IBM DB2

IBM has integrated JSON only in its system called DB2 for i [2]. Version 7.1 of this system supports storage of JSON data, while the successive version contains implementation of the JSON_TABLE function. The functions IS JSON and JSON_EXISTS were implemented in the newest version of this database system (Version 7.3).

The JSON_TABLE function supports *lax* and *strict* modes in the same way as they are described in the standard. JSON_EXISTS and JSON_TABLE support the following standardized error handling options: FALSE ON ERROR, TRUE ON ERROR, UNKNOWN ON ERROR and ERROR ON ERROR.

7.2 Oracle

7.2.1 General Features

Oracle has implemented almost the whole specification of the SQL/JSON standard, in the way as the corresponding features are specified in the proposal [7]. (Several slight differences from the standard will be described below.)

Oracle 12c Release 1 was the first version, where the SQL/JSON integration has been implemented. Release 2 additionally contains the implementation of the publishing functions. Oracle supports the *lax* and *strict* mode. The default mode for JSON in Oracle Database is *lax*. All clauses for handling errors in the *strict* mode are supported (ON ERROR clause). Oracle does not support the standardized ON EMPTY clause of the *lax* mode.

Oracle supports the FORMAT JSON option, as well as the RETURNING clause. For JSON_VALUE, the data types VARCHAR2 or NUMBER in a RETURNING clause can be specified. For JSON_QUERY, only VARCHAR2 can be used. If the RETURNING clause is not specified, the default (VARCHAR2(4000)) is applied.

The RETURNING clause also accepts two non-standardized keywords, PRETTY and ASCII. If both are used, the PRETTY keyword must come before ASCII. The effect of keyword PRETTY is to pretty-print the returned data by inserting newline characters and indenting the output. The default behavior is not to pretty-print. The effect of the ASCII option is to automatically escape all non-ASCII Unicode characters. ASCII is allowed only for Oracle SQL functions JSON_VALUE and JSON_QUERY. PRETTY is allowed only for JSON_QUERY.

The IS JSON predicate is supported in the same way, as it is specified in SQL/JSON. Also, the unique key constraint is supported with the WITH UNIQUE KEY clause.

7.2.2 Storing and Projecting JSON Documents

The SQL standard data types VARCHAR, BLOB and CLOB are used to store JSON documents.

Oracle supports the standardized JSON_TABLE function. Simple reports (with no nesting) as well as nested reports (using the NESTED PATH option) can be done with Oracle. The FOR ORDINALITY clause is implemented, too. Oracle does not support the PLAN clause.

7.2.3 Publishing Relational Data

All standardized projecting functions: JSON_OBJECT, JSON_ARRAY, JSON_OBJECTAGG and JSON_ARRAY AGG are supported. Additionally, Oracle has implemented two SQL NULL handling clauses: NULL ON NULL and ABSENT on NULL. The former converts the SQL NULL value to JSON null. For the latter, there is no corresponding output for SQL NULL on input.

7.2.4 Querying JSON Documents

All three functions, JSON_EXISTS, JSON_VALUE and JSON_QUERY, are supported in the same way as they are specified in the SQL/JSON standard. Three wrapper clauses (WITH UCONDITIONAL ARRAY WRAPPER, WITHOUT ARRAY WRAPPER, WITH CONDITIONAL ARRAY WRAPPER) are supported, too. (They can be used only with JSON_QUERY and JSON_TABLE functions.) The clauses EMPTY ARRAY ON ERROR and EMPTY OBJECT ON ERROR are not supported by Oracle.

Oracle supports an alternative way to query JSON documents using dot notation that resembles an attribute dot notation for an abstract data type in object-relational DBMSs. (The examples can be found in [9].)

7.3 SQL Server

7.3.1 General Features

SQL Server 2016 is the first version of this database system, which supports JSON. Microsoft has implemented only a part of standardized functions and predicates [8, 12]. None of general features described in Sect. 2 are implemented in SQL Server. The system supports the non-standardized function ISJSON, which have the same semantic meaning as IS JSON.

7.3.2 Storing and Projecting JSON Documents

SQL Server stores JSON documents in a row form. Therefore, JSON objects can be stored as values of the NVARCHAR data type.

Projecting JSON documents in relational form is supported by SQL Server, but in a non-standardized way. SQL Server supports the OPENJSON function to project JSON documents as relational data. This function is a table-valued function that analyzes a given text to find an array of JSON objects. All objects found in the array are searched and, for each of them, the system generates a row in the output result.

7.3.3 Publishing Relational Data

SQL Server supports the non-standardized FOR JSON clause at the end of the SELECT statement to publish relational data as JSON documents. In that case, every row will be formatted as one object, with values generated as value objects and column names used as key names. This clause has two options, AUTO and PATH. With the former, the format of the JSON output is automatically determined based on the order of columns in the SELECT list and their source tables.

In the PATH mode, column names or column aliases are treated as expressions that indicate how the values are being mapped to JSON. (An expression consists of a sequence of nodes, separated by /. For each slash, the system creates another level of hierarchy in the resulting document.) In contrast to AUTO mode, using the PATH mode allows users to maintain full control over the format of the JSON output.

7.3.4 Querying JSON

SQL Server supports two standardized functions that are used to query JSON documents: JSON_VALUE and JSON_QUERY. The syntax and semantics of both functions is identical to the syntax and semantics described in the SQL/JSON standard.

SQL Server does not support the standardized function JSON_EXISTS. The functionality of this function can be implemented in some cases using JSON_VALUE and JSON_QUERY functions in the WHERE clause of the SELECT statement.

8 Summary

Before discussing the properties and deficiencies of SQL/JSON, we will give a short explanation in relation to JSON and XML. JSON as well as XML are data format languages, but there are several differences between them. There are three general properties (simplicity, extensibility and interoperability), which can be used to evaluate any data format language.

Concerning simplicity, JSON is much simpler than XML, because it has a smaller grammar. In case of extensibility, JSON is not extensible, while XML is extensible, meaning that in XML one can define new tags or attributes to represent data in it. In relation to data formats, interoperability means that a format can transfer data between many different systems. Concerning this characteristic, JSON has the same interoperability potential as XML.

As we already stated, the first proposal of SQL/JSON standard is "light-weighted". One of the reasons for this decision is to allow vendors of RDBMSs to implement the proposed features in their systems as soon as possible.

On the other hand, there are several negative impacts in relation to this decision. The main problem is that several important features are not specified. From our point of view, the following three are the most important, and should be specified in one of the future versions of the SQL/JSON standard:

- JSON documents should be first-class objects in SQL (native storage)
- The syntax of the SQL UPDATE statement should be extended to allow updating of parts of a JON document
- Direct access of external JSON data should be supported

8.1 Native Storage of JSON Documents

The current SQL/JSON standard proposes the storage of JSON documents using the VARCHAR, CLOB or BLOB standard data types. "Native storage" means to store JSON documents as a unit and to create a model that is closely aligned with JSON. This model should include arbitrary levels of nesting and complexity and should be automatically mapped by the system into the underlying storage mechanism.

The benefit of native JSON storage in a database is that the JSON documents are stored in a parsed format and therefore accessible without using parsing. (Excessive parsing on a web server or application server easily leads to performance problems or out-of-memory situations on the servers.) Therefore, the main benefits of native storage for JSON are achieving greater capability for users and better performance.

The proposal already specifies several constructor functions (see Sect. 5). Additionally, several operators should be specified to produce JSON values. These operators should be semantically equivalent to the list of operators proposed for the SQL/XML standard.

A possible (minimal) list of these operators is given below:

- JSON_ CONCAT— takes two JSON documents, combines them, and returns a JSON document
- JSON_KEYCOUNT—returns the number of keys in a JSON document
- JSON_KEYS—returns the list of key names
- JSON_CONTAINS—tests whether a JSON document contains another document
- JSON_EXISTENCE—tests whether a string (JSON value) appears as a key or array element

8.2 Updating Parts of JSON Documents

The current proposal allows users to update only an entire JSON document. The update of the whole document has significant performance disadvantages. For this reason, modification of parts of a JSON document, where these parts are specified in the SET clause of the SQL UPDATE statement is necessary.

The existing standard introduced already the syntax for SQL/JSON path expressions. This syntax can be used to specify the parts of the document, which has to be modified. In that case, the syntax of the SQL UPDATE statement could have the following form:

```
UPDATE table_name SET json_column =
 JSON_UPDATE(json_column , path_expr)
 WHERE condition;
```

where *path_expr* is a SQL/JSON path expression.

8.3 Access of External JSON Documents

Access to JSON documents in the present proposal requires that the JSON data must be stored in relational tables. In other words, none of the specified SQL/JSON functions have parameters that reference external JSON data. The only way to access external data is to insert them into character or binary string and load that string in a row of a relational table.

One way to solve this problem is to use the standardized specification for management of external data (SQL/MED). Another is, to define the syntax of an external (read-only, index-free) table, where JSON data is stored. After that, the data could by inserted in the JSON column using the SQL INSERT statement.

References

1. Avro (2017) Avro. http://avro.apache.org/. Accessed 25 Aug 2017
2. Bestgen, R. – Using DB2 for i with XML and JSON, schd.ws/hosted_files/commons17/6d/26AD
3. BSON. http://bsonspec.org/. Accessed 25 Aug 2017
4. Chasseur C, Li Y, Patel J (2013) Enabling JSON document stores in relational systems. Sixteenth International Workshop on the Web and Databases, New York, 23.6.2013.
5. Liu CH, Hammerschmidt B, McMahon D (2014) JSON data management supporting schema-less development in RDBMS. Proceedings of the 2014 International Conference on Management of Data, Snowbird, 22.06.2014-27.06.2014, pp 1247–1258
6. Liu CH, Hammerschmidt B, McMahon D, Li Y, Chang HJ (2016) Closing the functional and performance gap between SQL and noSQL. Proceedings of the 2016 International Conference on Management of Data, San Fransisco, 26.06.2016-01.07.2016.
7. Oracle Help Center (2017) JSON in oracle database. http://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246. Accessed 25 Aug 2017

8. Petković D (2016) SQL server 2016, A Beginner's Guide. McGraw-Hill, New York
9. Petković D (2017) JSON integration in relational database systems. Int J Comput Appl 168(5):14–19
10. PostgreSQL (2017) PostgreSQL 9.4, JSON types. www.postgresql.org/docs/9.4/static/datatype-json.html. Accessed 25 Aug 2017
11. PostgreSQL (2017) PostgreSQL 9.4, JSON functions. www.postgresql.org/docs/9.4/static/functions-json.html. Accessed 25 Aug 2017
12. Microsoft (2017) SQL server: JSON data. https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server. Accessed 25 Aug 2017
13. Tahara D (2014) Sinew: a new SQL system for multi-structured data. Proceedings of the 2014 International Conference on Management of Data, Snowbird, 22.06.2014-27.06.2014.
14. Zemke F, Hammerschmidt B, Kulkarni K, Liu Z, McMahon D, Melton J, Michels J, Özcan F, Pirahesh H (2014) ANSI SQL/JSON: part 1. https://www.wiscorp.com/pub/DM32.2-2014-00024R1_JSON-SQL-Proposal-1.pdf. Accessed 25 Aug 2017
15. Zemke F, Hammerschmidt B, Kulkarni K, Liu Z, McMahon D, Melton J, Michels J, Özcan F, Pirahesh H (2014) ANSI SQL/JSON: part 2 : querying JSON. www.wiscorp.com/pub/DM32.2-2014-00025r1-sql-json-part-2.pdf. Accessed 27 Aug 2017