JSON Integration in Relational Database Systems

Dušan Petković
University of Applied Sciences
Hochschulstr. 1, Rosenheim, 83024, Germany

ABSTRACT
Recently, a new era of application development is emerging, which us based upon big data technology and the ease of access to compute resources, such as mobile devices. All these issues can be better supported using JSON (and JavaScript) technology. Almost all relational database systems have integrated JSON, partly according to the specification given in the ANSI SQL standard and partly according to other specifications. In this article we discuss several JSON features and investigate how different relational database systems (RDBMSs) have integrated them. Of all database systems discussed in this paper Oracle has implemented the most concepts specified in the ANSI SQL/JSON standard. In contrast to Oracle, PostgreSQL have not implemented any standardized features. Also, we discuss conformance of all these implementations in relation to the ANSI SQL/JSON standard and give suggestions, which important features should be implemented in the future releases of the RDBMSs.

Keywords
JSON, SQL/JSON, relational database systems, JSON integration

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. The following example:

```
{"info": {"who": "Fred" ,"where": "BBC",
"friends":[{"name": "Lili", "rank":5},{"name": "Hank", "rank": 7}]}}
```

shows a JSON string called **info** that describes a single person, Fred, his affiliation, BBC, and his friends, Lili and Hank. Generally, a JSON string contains either an array of values or an object, which is an array 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).

1.1 Why Support JSON in RDBMSs?
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 objects, 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 administers all data. The same is true for security and transaction management, because the system takes over the management of security and transaction processing, meaning that this functionality does not need to be implemented in users' programs. Finally, integrating JSON in a RDBMS increases productivity, because the system takes over a lot of tasks that otherwise must be implemented programmers in their programs.

1.1.1 Integration Features Discussed
For all RDBMSs mentioned in this article, the following features will be discussed:

- Storing JSON documents in RDBMSs
- Presentation of JSON documents
- Indexing techniques for JSON documents
- Querying JSON documents

1.1.1.1. Storing JSON in RDBMSs
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 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 required.

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.

1.1.1.2. Presentation of JSON
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.

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 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.

1.1.1.3. Indexing Techniques for JSON

Generally, indexes are used to increase performance of queries. For this reason, indexing JSON documents to better perform queries on them is one of the most important features, which a RDBMS must support, when JSON documents are integrated in the system. There are three general techniques, which can be used to index JSON documents: database-oriented approach, information retrieval (IR)-oriented approach and hybrid approach. Database-oriented approach means that traditional techniques for RDBMSs, such as B+-tree and bitmap indexes are used for indexing. IR approach uses techniques from information retrieval, which are usually based upon inverted lists, to search for values in JSON. The hybrid approach is a mix of the both indexing techniques.

1.1.1.4. Querying JSON Documents

In contrast to XML community, which introduced special language (XQuery) to query XML data, there is no corresponding query language for JSON. For this reason, the idea behind querying methods in relation to JSON is to extend SQL with appropriate operators and functions that embed a simple path expression language to navigate and query JSON object instances.

1.2 Related Work

Besides RDBMSs, vendors of NoSQL data stores, like MongoDB [6] and CouchDB [2], 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 [14].

The topic of several articles is JSON integration in RDBMSs. Liu et al. [4] present three architectural principles for schema-less development within RDBMSs. The same author investigates in [5], how JSON data model can be added to a RDBMS. The same idea, but discussed from the different aspect is debated in [1].

1.3 Roadmap

The rest of the paper is organized as follows. Section 2 describes the implementation of features in Oracle, while Section 3 explains how these features are integrated in PostgreSQL. The next section discusses integration of JSON in MS SQL Server. The structure of these three sections is identical. Section 5 summarizes the results. In this section we discuss conformance of all these implementations in relation to the ANSI SQL/JSON standard and give suggestions, which important features should be implemented in the future releases of the database systems.

2. ORACLE

JSON data in Oracle Database can be used in a similar way as XML data. This means that these data, in contrast to relational data can be stored, indexed and queried without any need for a schema, which specifies that data. The description of JSON implementation can be found in [7].

2.1 Oracle: Storing JSON Documents

JSON documents are stored in Oracle using SQL data types VARCHAR2, CLOB and BLOB. The latter is recommended by Oracle, because that way there is no need for any character-set conversion.

Example 1

```
-- Create table and inserts
CREATE TABLE ora_json_table
(id RAW(16) NOT NULL,
person_and_friends CLOB,
PRIMARY KEY (id),
CHECK (person_and_friends IS JSON));
INSERT INTO ora_json_table (id, person_and_friends)  VALUES
  (SYS_GUID() ,  N'{"info":{"who": "Fred" ,"where": "Microsoft" ,
 "friends":[{"name":"Lili","rank":5}, {"name":"Hank","rank":7}]}}');
INSERT INTO ora_json_table (id, person_and_friends)  VALUES
( SYS_GUID() ,  N'{"info":{"who": "Tom", "where": "IBM", "friends": [ {
"name":   "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] }}');
 INSERT INTO ora_json_table (id, person_and_friends)  VALUES
(SYS_GUID() ,N'{"info":{"who":"Jack", "friends":
[ { "name": "Connie" } ] }}');
  INSERT INTO ora_json_table (id, person_and_friends)  VALUES
(SYS_GUID()
,N'{"info":{"who":"Joe","friends":[{"name":"Doris"},{"rank":1}]}}');
  INSERT INTO ora_json_table (id, person_and_friends)  VALUES
(SYS_GUID() ,N' {"info": {"who":"Mabel",
"where":"PostgresSQL","friends":[{"name":"Jack","rank": 6}]}}');
INSERT INTO ora_json_table (id, person_and_friends)  VALUES
(SYS_GUID() ,
    N' {"info":{"who": "Louise", "where": "Hanna" }}');
```

IS JSON in the CREATE TABLE statement of Example 1 is an Oracle/JSON function, which checks whether the inserted document is a valid JSON instance. (The term "valid" describes data instances that satisfy all JSON syntactic requirements.) Note that this function is specified in the ANSI SQL/JSON standard, and Oracle is the only system discussed in this article, which supports this standardized function.

2.2 Oracle: Presentation of JSON

Oracle Database supports projecting of JSON documents as relational data, as well as publishing relational data as JSON documents. As will be seen below, publishing relational data cannot be done in Oracle Database in a simple way, as it is the case with other RDBMSs.

2.2.1 Projecting JSON documents

The standardized **json_table**() function can be used to present JSON documents in relational form. As all other Oracle JSON functions, this one uses path expressions to display selected data in relational form.

Example 2

```
 SELECT jt.* FROM   ora_json_table,
     JSON_TABLE(person_and_friends, '$.info[*]'
       COLUMNS (
         "Friend"               PATH '$.who',
  "Affiliation" VARCHAR2(20) PATH '$.where')) "JT";
```

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 specifies which part of the JSON document should be projected to the given column name.)

2.2.2 Publishing Relational Data as JSON

Oracle supports several interfaces, which can be used for publishing. We will briefly discuss two of them:

-APEX_JSON
- PL/JSON

APEX_JSON is a PL/SQL API that provides utilities for parsing and generating JSON documents. PL/JSON is an open source library for working with JSON in Oracle Database. The core of the library is based on two object types, JSON and JSON_LIST, which correspond to the types of structures that exist in JSON and which are used to transform relational data in the JSON data types.

2.3 Oracle: Indexing Techniques for JSON

There are several different forms of indexes, which can be used to enhance performance of queries in relation to JSON documents. These are:

- Composite B-Tree indexes
- Bitmap indexes
- Function-based indexes
- Full-Text Search indexes

The composite B-tree and bitmap index are well-known indexes used by Oracle to index columns of relational tables. For this reason, only function-based indexes and full-text search indexes will be discussed.

The following example shows how function-based index can be created, while Example 7 shows the use of a full-text search index.

Example 3 (function-based index)

```
CREATE INDEX json_docs_email_idx
  ON ora_json_table (JSON_VALUE(person_and_friends,
'$.info.who'
  RETURNING VARCHAR2 ERROR ON ERROR));
```

The index in example above will be used on all queries which search for affiliation of a person. (The **json_value**() function will be explained in the next section.) The extended discussion concerning the use of each index form for different queries can be found in Oracle documentation [8].

2.4 Oracle: Querying JSON Documents

Generally, Oracle Database supports three functions in relation to JSON:

- json_table()
- json_value()
- json_query()

and three conditions:

- is_json()
- json_textcontains()
- json_exists()

The **json_table**() function is used to present JSON documents in relational form (see Example 2), while the **is_json**() condition tests whether the particular JSON document is valid (see Example 1). All other functions and a condition will be discussed below.

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 text and ***path*** is a path expression that specifies the property to extract.

Example 4

```
SELECT id,JSON_VALUE(person_and_friends,'$.info.where')
          AS company  FROM ora_json_table
    WHERE JSON_VALUE(person_and_friends, '$.info.who') =
                        'Fred';
-----------------------------------------------------------
4A2E6BE991044B53BAFCC41B52C2 Microsoft
```

The **json_query**() function returns extracts of an object or an array from a JSON document. The syntax is analogous to the syntax of the **json_value**() function.

Example 5

```
SELECT person_and_friends
  FROM ora_json_table
  WHERE JSON_QUERY
        (person_and_friends, '$.info.who') = 'Fred';
-----------------------------------------------------------
{"info":{"who": "Fred" ,"where": "Microsoft" ,
 "friends":[{"name":"Lili","rank"]}}
```

Note that Oracle functions **json_value**() and **json_query**() accept the standardized RETURNING clause, which specifies the data type of the value returned by the function (see Example 3).

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.

Example 6

```
SELECT o.person_and_friends.info.friends
    FROM ora_json_table o
  WHERE o.person_and_friends.info.who = 'Fred' ;
```

Oracle additionally supports a search for values using the full text search indexes, together with the **json_textcontains**() function. Therefore, the full-text index is created first and, after that, the **json_textcontains**() function is used to search for values. (The following example searches for value "Jack". This value appears in the key called "who" as well as in the key called "friends".)

Example 7

```
-- First, create full-text index
CREATE INDEX json_index
  ON ora_json_table(person_and_friends)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS ('SECTION GROUP
CTXSYS.JSON_SECTION_GROUP SYNC (ON COMMIT)');
-- Second, use the json_contains() function
SELECT person_and_friends Jack_is_here
FROM ora_json_table
WHERE JSON_TEXTCONTAINS( person_and_friends, '$','Jack');
```

```
{"info":{"who":"Jack", "friends": [ { "name": "Connie" }   ] }}
{"info": {"who":"Mabel",
"where":"PostgresSQL","friends":[{"name":"Jack"}]
```

The **json_exits**() functions takes a path expression and checks if such path selects one or multiple values in the JSON documents.

Example 8

```
SELECT count(*)
FROM ora_json_table
WHERE JSON_EXISTS (preson_and_friends, '$.info.friends')
AND  '$.info.who' = 'Fred';
```

3. POSTGRESQL

PostgreSQL started to integrate JSON with V9.2. However, the main enhancements (functions and operators) came with V9.3. The description of JSON data types can be found in [10], while functions are described in [11]. Use of PostgreSQL functions and operators for querying JSON documents is given in [12].

3.1 PostgreSQL: Storing JSON

PostgreSQL supports two data types for storing JSON documents: JSON and JSONB. There are two significant differences between these data types. First, the JSON data type stores an exact copy of the input text, which processing functions must reparse on each execution, while JSONB data are stored in a decomposed binary form. Therefore, data stored using the latter data type will be loaded slower, because of conversion overhead, but it will be processed significantly faster, since no reparsing is needed. Second, the JSONB data type supports more operators than the JSON type.

Example 9

```
-- Creating a table and loading data
CREATE TABLE postgres_json_table
(id INT, person_and_friends JSONB);
-- Insertion of rows
INSERT INTO postgres_json_table  VALUES
  (1, '{"info":{"who": "Fred" ,"where": "Microsoft" ,
 "friends":[{"name":"Lili","rank":5}, {"name":"Hank","rank":7}]}}');
INSERT INTO postgres_json_table   VALUES
  (2, '{"info":{"who": "Tom", "where": "Oracle", "friends": [ { "name":
 "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] }}');
INSERT INTO postgres_json_table  VALUES
  (3, '{"info":{"who":"Jack", "friends": [ { "name": "Connie" } ] }}');
INSERT INTO postgres_json_table   VALUES
 (4, '{"info":{"who":"Joe","friends":[{"name":"Doris"},{"rank":1}]}}');
INSERT INTO postgres_json_table  VALUES
  (5, ' {"info": {"who":"Mabel",
 "where":"PostgresSQL","friends":[{"name":"Jack","rank": 6}]}}');
INSERT INTO postgres_json_table   VALUES
      (6, ' {"info":{"who": "Louise", "where": "IBM" }}');
```

3.2 PostgreSQL: Presentation of JSON

3.2.1 Projecting JSON Documents

PostgreSQL supports several operators that are used, among other things, to project JSON documents in relational form. Table 1 shows the most important operators and their description. The first column specifies the operator, the second the right operand type and the last one describes the corresponding operator. (As will be seen below, the same operators can be used for querying data.)

Table 1: The most important Postgres operators

->	Int	Get JSON array element
->	Text	Get JSON object field by key
->>	Int	Get JSON array element as text
->>	Text	Get JSON object field as text

From Table 1 is clear that the -> operator returns the original JSON object, while ->> returns text. Therefore, chaining the both operators, and using the ->> operator as the last one on the right side, allows users to project any part of a JSON document in relational form.

Example 10

```
SELECT id, person_and_friends->'info'->>'who' AS x
    FROM postgres_json_table;
----------------------
1   Fred
2   Tom
etc.
```

3.2.2 Publishing Relational Data as JSON

PostgreSQL offers the **row_to_json**() function to publish relational data as JSON documents.

Example 11

```
-- create table and insert two rows
CREATE TABLE department (dept_no INT,
            dept_name CHAR(20), location CHAR(20));
-- Insert two rows
INSERT INTO department VALUES (1, 'Marketing', 'Seattle');
INSERT INTO department VALUES (2, 'Production', 'Boston');
-- Use row_to_json() to publish relational data as JSON document
SELECT row_to_json(row(dept_no, dept_name))
        FROM  department;
-------------------
{"f1":1, "f2":Marketing          "}
{"f1":2, "f2":Production         "}
```

Note that disadvantage of the **row_to_json**() function is that the output can be displayed in a fixed form, and users cannot influence that output in any way.

3.3 Indexing Techniques for JSON

PostgreSQL supports the GIN index for indexing JSON documents. GIN indexes belong to a class of function-based indexes, which can be used to efficiently search for keys or key/value pairs occurring within documents of the JSONB data type.

Example 12

```
CREATE INDEX idxgin ON postgres_json_table USING   GIN
(person_and_friends);
```

If the GIN index in the example above is created, queries containing the path/value-exists operator @> will make the use of the created index. (The @> operator is explained in Example 14.)

3.4 Querying JSON

PostgreSQL supports two JSON operators, -> and ->>, shown in Table 1. They can be used in the condition of the WHERE clause to query documents.

Example 13

```
SELECT id, person_and_friends->'info'->>'where'
  FROM postgres_json_table
WHERE person_and_friends->'info'->>'who' = 'Mabel';
-----------------------------------
5    PostgreSQL
```

The same operators can be used to access elements of an array in a JSON document. In that case the last argument of the path expression will be a number, defining the index of the corresponding element. (The first element of an array has the index 0.)

Example 14
```
SELECT id, person_and_friends->'info'->'friends'->0 FROM
postgres_json_table WHERE id = 3;
-------------------------
 3 | { "name": "Connie" }
```

PostgreSQL supports test for containment, using the <@ operator.

Example 15
```
SELECT id FROM postgres_json_table
   WHERE person_and_friends @>' {"name": "Jack"}';
```

4. MS SQL SERVER

SQL Server 2016 is the first version of Microsoft's database systems, which supports integration of JSON documents [9]. For this reason, only part of JSON features, offered by Oracle and PostgreSQL, are supported in this version. SQL Server documentation can be found in [13].

4.1 SQL Server: Storage of JSON

SQL Server stores JSON documents similar as Oracle Database, using the NVARCHAR data type.

Example 16
```
-- create table
CREATE TABLE MS_json_table
(id INT PRIMARY KEY IDENTITY, person_and_friends
NVARCHAR(2000));
-- Insert rows
INSERT INTO json_table (person_and_friends)  VALUES
  (N'{"info":{"who": "Fred" ,"where": "Microsoft" ,
 "friends":[{"name":"Lili","rank":5}, {"name":"Hank","rank":7}]}}');
INSERT INTO json_table (person_and_friends)  VALUES
  (N'{"info":{"who": "Tom", "where": "Oracle", "friends": [ { "name":
 "Sharon", "rank": 2}, {"name": "Monty", "rank": 3} ] }}');
INSERT INTO json_table (person_and_friends)  VALUES
  (N'{"info":{"who":"Jack", "friends": [ { "name": "Connie" } ] }}');
INSERT INTO json_table (person_and_friends)  VALUES
(N'{"info":{"who":"Joe","friends":[{"name":"Doris"},{"rank":1}]}}')
INSERT INTO json_table (person_and_friends)  VALUES
  (N' {"info": {"who":"Mabel",
"where":"PostgresSQL","friends":[{"name":"Jack","rank": 6}]}}');
INSERT INTO json_table (person_and_friends)  VALUES
      (N' {"info":{"who": "Louise", "where": "IBM" }}');
```

4.2 SQL Server: Presentation of JSON

MS SQL Server supports publishing JSON as relational data as well as presentation of relational data as JSON documents. For the latter, there are several options, which allow users to maintain full control over the format of the output.

4.2.1 Publishing JSON Documents as Relational Data

SQL Server supports the OPENJSON function to publish JSON documents as relational data. This non-standardized 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.

There are two forms of the OPENJSON function:

- With a predefined result schema
- Without the schema

If a schema exists, it defines mapping rules that specify what properties will be mapped to the returned columns. Without such a schema, the result is set of key-value pairs.

Example 17
```
DECLARE @json NVARCHAR(MAX) =
N' {"info":{ "who": "Fred" ,"where": "Microsoft" ,
 "friends":[{"name":"Lili","rank":5}, {"name": "Hank", "rank": 7}]}'
SELECT [key], value FROM OPENJSON(@json, N' $.info.friends');
------  -------------------------- --------
0           {"name": "Lili", "rank": 5}
1           "name": "Hank", "rank": 7}
```

4.2.2 Publishing Relational Data as JSON Documents

SQL Server supports the FOR JSON clause at the end of the SELECT statement to present 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. This mode returns the result set of a query as a simple, nested JSON tree. Each table in the FROM clause from which at least one column appears in the SELECT list is represented as an object. The columns in the SELECT list are mapped to the appropriate attributes of that object.

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.

Example 18
```
-- department table from Example 10 is used here
SELECT  dept_no AS [Department.Number], dept_name AS
[Department.Name]
   FROM department FOR JSON PATH, ROOT ('Departments');
---------------------------------- ----------------------------------
{"Departments":
[{"Department":{"Number":"d1
","Name":"Research"}},{"Department":{"Number":"d2
","Name":"Accounting   "}},{"Department":{"Number":"d3
","Name":"Marketing  "}}]}
```

4.3 Indexing Techniques for JSON

Because JSON documents are stored using the NVARCHAR data type, SQL Server offers traditional B-tree indices to index JSON documents. Special indexing methods, as in case of Oracle Database, are not supported.

4.4 SQL Server: Querying JSON

SQL Server supports two functions that are used to query JSON documents:

- json_value()
- json_query()

The syntax and semantics of both functions is identical to the syntax and semantics of the Oracle functions with the same name. For this reason, Examples 4 and 5 can be used in the same way for SQL Server as for Oracle Database.

SQL Server supports another function called **isjson**(), to test whether a string contains valid JSON document. This function is analogous to the IS JSON function supported by Oracle. The function returns 1 if the string contains valid JSON; otherwise, it returns 0.

Example 19

```
SELECT id, person_and_friends
FROM json_table  WHERE isjson(person_and_friends) > 0;
```

SQL Server does not support the standardized function **json_exists**(), which is implemented in Oracle (see Example 8). The functionality of this JSON function can be implemented (in some cases) using **json_value**() or **json_query**() function in the WHERE clause of the SELECT statement.

5. SUMMARY

This section will discuss two different issues:

- Conformance of the JSON integration in the RDBMSs in relation to the ANSI SQL/JSON standard
- Missing features

5.1 Conformance to the Standard

Of all database systems discussed in this paper, Oracle has implemented the most features specified in the ANSI SQL/JSON standard. The storage of JSON documents as well as the IS JSON condition are implemented according to the specifications in the standard. The implementation of the functions **json_query**() and **json_value**() functions also conforms to the standard. The standardized RETURNING clause for **json_value**() and **json_query**() functions is implemented by Oracle, too, as shown in Example 3.

In contrast to Oracle, PostgreSQL has not implemented any features specified in standard. (Even the JSON data type is completely different to the proposal in the standard.) The reason for this is that PostgreSQL implemented JSON features before the ANSI SQL/JSON standard has been accepted. It seems so that PostgreSQL integration of JSON has been completed. Therefore, it is unclear whether any of features specified in the standard will be implemented in a future version of PostgreSQL.

SQL Server implemented just a few standardized features in relation to JSON integration. The data type for JSON documents in SQL Server is equivalent to the specification in the standard. Also, **json_query**() and **json_value**() functions conforms to the standard. All other existing features described in Chapter 4 are non-standardized. SQL Server 2016 is the first version of the Microsoft's database system. For this reason, we can expect that several other standardized JSON features will be part of one of future versions of SQL Server.

5.2 Missing Features and Future Work

The most important feature that is not specified in the standard and therefore not implemented in the RDBMSs discussed in this paper is the native support of JSON data type, the same way as the XML data type is implemented in Oracle and SQL Server. It seems so that native JSON support will not be implemented in the near future, because even the ANSI SQL/JSON standard [16] proposed only the "lightweight" approach to specify support for JSON in the context of the SQL language. Another missing feature is the use of the SQL UPDATE statement to modify parts of a JSON document. The present proposal in the standard does not specify any mechanism for modifying parts of JSON data. This means that the use of an UPDATE statement completely *replace* the value of a column storing JSON data. All RDBMSs discussed above do the same; they allow an UPDATE statement on a column storing JSON data, but this update always writes a new version of the whole document. The way, how UPDATE could be implemented can be found in two papers: the first one [3], gives a theoretical foundations, while [15] describes the implementation of XML in Oracle Database. In the future work we intend to discuss different ways, how modification of parts of a JSON document should be done and to compare their performance.

6. REFERENCES

[1] Chasseur, C. et al. – Enabling JSON Document Stores in Relational Systems, WebDB, 2013.

[2] CouchDB, www.couchdb.org

[3] Li, C., Ling, T.W. & Hu, M. - Efficient updates in dynamic XML data: from binary string to quaternary string, VLDB Journal (2008) 17: 573.

[4] Liu, C.H. et al. – JSON Data Management Supporting Schema-Less Development in RDBMS, SIGMOD/PODS'14, p. 1247-58, June 2014.

[5] Liu, C.H. et al. – Closing the functional and Performance Gap between SQL and NoSQL, SIGMOD'16.

[6] MongoDB, www.mongodb.org

[7] JSON in Oracle Database, http://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246

[8] Indexing JSON Data in Oracle Database, https://oracle-base.com/articles/12c/indexing-json-data-in-oracle-database-12cr1

[9] Petkovic, D. SQL Server 2016, A Beginner's Guide, McGraw Hill Educational, 2016.

[10] PostgreSQL 9.4, JSON Types, www.postgresql.org/docs/9.4/static/datatype-json.html

[11] PostgreSQL 9.4, JSON Functions , www.postgresql.org/docs/9.4/static/functions-json.html

[12] Schinckel, M. Querying JSON in Postgres, schinckel.net/2014/05/25/querying-json-in-postgres

[13] SQL Server: JSON Data, https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server

[14] Tahara, D, et al. – Sinew: A new SQL System for Multi-Structured data SIGMOD Conf. 2014.

[15] UPDATEXML, https://docs.oracle.com/cd/B19306_01/server.102/b14200/functions205.htm

[16] Zemke, F et al.- ANSI SQL/JSON: Part 1, www.wiscorp.com/pub/DM32.2-2014-00025r1-sql-json-part-1.pdf, 2014.