An Analysis of the JSON Functionalities Evolution Across Different Versions of Oracle Relational DBMS

Srdja Lj. Bjeladinovic*, Marko S. Asanovic**, Natasa M. Gospic***
* University of Belgrade, Faculty of organizational science, Belgrade, Serbia
** Adriatic University Bar, Faculty for traffic, communication and logistics, Budva, Montenegro
*** Adriatic University Bar, Faculty for traffic, communication and logistics, Budva, Montenegro
srdja.bjeladinovic@fon.bg.ac.rs
asanovicmarko@live.com
n.gospic@gmail.com

***Abstract* — Modern web applications, as a necessity, require fast, efficient, and accurate data exchange, which all participants in communication can uniformly interpret. Consistency in interpreting exchanged data can be achieved using text, self-describing and platform-independent file formats such as XML and, recently, the even more popular JSON. JSON represents a common model for NoSQL databases. Also, it has been supported for years in relational databases from leading manufacturers, such as Oracle. This paper systematises, analyses, and compares the functionalities for working with JSON across different versions of Oracle DBMS, starting with version 12c R1, which introduced support for native JSON, concluding with the recently introduced version. For selected use cases, experimental tests were performed, which demonstrated the impact on performance achieved by the supported JSON functionalities from version 12c R1 to 19c of Oracle relational database management systems (DBMSs).**

***Keywords*: JSON, Oracle, relational DBMS.**

I. INTRODUCTION

JSON (JavaScript Object Notation) is currently a very popular format of self-describing text files used in modern web-based applications. This format is straightforward for using by humans and machines [1]. The popularity of JSON has influenced the emergence of database management systems (DBMSs) based on it, which is primarily applicable for NoSQL databases. NoSQL DBMSs are adjusted for working with large and complex data types [2]. A clear example is MongoDB, a representative of document-oriented NoSQL databases, suitable for storing JSON data [3], which in December 2020 was ranked as the 5th most common DBMS in use, with a growth of over 36% compared to the same period last year [4]. Increased use of JSON and NoSQL databases have contributed to the fact that, after including support for XML and object types, the relational databases' manufacturers also introduced support for working with JSON.

A. Motivation

A review of versions of Oracle DBMS, the leading manufacturer of relational databases [4], found that back in 2013, the version 12c Release 1 (12c R1) [5] introduced support for working with native JSON. Native JSON supports data storing and querying without obligatory schema definition [6]. After that, each new version [7][8] introduced an extension of the initial functionalities. Functionally "the richest" version that supports native JSON was introduced in 2019 [9]. Oracle announced the introduction of support for binary JSON in the version 21c. The observed versions innovations have produced the motivation to form research reviewing the evolution of supported JSON functionalities across different versions of Oracle relational DBMS, starting with native JSON in the version 12c R1 to announced binary JSON in 21c. In addition to the theoretical comparison of available functionalities by versions, their similarities and differences in working with JSON leave room for practical testing and analysis of the impact on statement execution time as one of the performance indicators.

B. Research questions

The motivation for this paper profiled the following research questions:

1) Which functionalities for working with JSON data does Oracle relational DBMS support, and what innovations did Oracle introduce after the version 12c R1?

2) What is the impact on the execution time of JSON manipulation statements achieved by the latest version of Oracle relational DBMS?

C. Methodology

The methodology used in this paper contains various techniques, methods, and tools. In the beginning, the relevant literature was collected, and the most significant works were described and analysed, which resulted in the emergence of this paper motivation. Deduction led to the formulation of research questions. The available functionalities for working with JSON data according to

versions of Oracle relational DBMS were systematised, after which analysis and comparison took place. Finally, test use-cases were created and performed over specific DBMSs. Using the SQL Developer tool, the impact of selected use cases on command execution time was measured as a performance indicator.

II. Related work

The authors [10] analyse the specifics of working with JSON within Oracle 12c R1, emphasising JSON DataGuide, and Oracle JSON (OSON). They conclude that adding JSON DataGuide and OSON binary formats to the DBMS provides the ability to integrate SQL and NoSQL databases, allowing users to manage relational data and data with flexible structure in a single DBMS.

The authors [11] investigate some of the approaches in modelling JSON data and corresponding JSON schemas. Tree schema models represent the base of the analysis. The paper researches the advantages and disadvantages of these schemas based on uncertain data, XML data, and graph data. The authors conclude that it is necessary to compromise between functionality, complexity, efficiency, and the possibility of extension when modelling JSON data.

The paper [12] provides an overview of the introduction of JSON support in different DBMSs. In 2012, PostgreSQL introduced support for native JSON in the version 9.2, while they added support for binary JSON in the version 9.6. MySQL introduced native JSON support in the version 5.7. Microsoft SQL Server installed support in 2016 in the version 13.00. In order to support JSON structure in their DBMS they used the NVARCHAR type. In 2013, Oracle published the version 12c R1 [5], which introduced support for working with native JSON [13]. Oracle announced support for working with binary JSON for 2021. At this moment, PostgreSQL provides significant level of support for working with JSON.

III. Analysis of supported functionalities in working with JSON

This chapter provides a retrospective of the most important functionalities for working with JSON through versions of Oracle DBMS, starting with the version 12c R1, which for the first time introduced support for working with JSON, ending with the latest version 21c.

A. Oracle 12c R1

Since the version 12c R1, Oracle has supported native JSON. It is not one of the predefined data types, instead conditioned VARCHAR or CLOB columns are used for storing JSON documents (rarely NVARCHAR, NCLOB, and BLOB type). Dedicated CHECK constraint (IS JSON) validates the format of data stored in these columns [5]. Traditional INSERT statement can achieve data storing for columns with the structure of a JSON document. Updating the column with JSON data is done with the UPDATE statement. The entire JSON document must be provided, i.e. it is impossible to change an individual property of the JSON document. There are several ways for data displaying. The first way is to create a query over the JSON document's nested elements, starting with the root element, using the "dot" notation. A limiting factor in using a "dot" notation for data access is the inability to express the difference between a non-existent element and an existing element without value [5]. For both types of elements, the result will be "null". Using the JSON_EXISTS function within the WHERE clause solves this problem. It returns "null" as a result only for the elements without value while ignoring non-existent properties. Function JSON_VALUE returns a scalar, i.e. does not return collections or nested parts, while JSON_QUERY returns a JSON document's snippet as a result [5]. JSON_TABLE includes the functionalities of previous functions. The syntax of the JSON_TABLE function supports the customisation of the resulting columns (their order, structure, filtering criteria, etc.) in a manner characteristic for relational database views.

B. Oracle 12c R2

A significant novelty brought by 12c R2 was the possibility of procedural use of the functions presented in 12c R1, i.e. their usage within PL/SQL blocks. In addition to the above, SQL/JSON functions have been introduced, aiming to generate JSON based on SQL. One of them is the JSON_OBJECT function, which specifies a list of key-value pairs, and displays the result in the JSON object format, regardless of the types of source data. The JSON_OBJECTAGG creates the resulting JSON object containing aggregated key-value pairs (obtained based on several resulting query rows). The JSON_ARRAY is used to create JSON objects, each containing a list of key-value pairs. JSON_ARRAYAGG is a function that aggregates records that satisfy a condition into a single resulting JSON array based on the passed expression [7].

C. Oracle 18c

In earlier versions of Oracle DBMS, the return value type of JSON functions (such as SQL/JSON functions) was VARCHAR, and the default value was VARCHAR(4000) [8]. The JSON_OBJECTAGG and JSON_ARRAYAGG functions could optionally return a CLOB as well. The version 18c allowed that the remaining SQL/JSON functions and the SQL_QUERY function could have a return value type of CLOB or BLOB [8].

TABLE I.
EXECUTED STATEMENTS IN ORACLE 12C R1 AND 19C

	INSERT	UPDATE	SELECT
12c R1	insert into product (id, product_data)…	update product set product_data = '{"Product_ID":"100", "Full_name":"NotebookX123", … }'	SELECT p.product_data.Product.*Full_name*, … FROM product p;
19c	insert into product (id, product_data)…	update product p set p.product_data = json_mergepatch (p.product_data, '{"*Full_name*": "*NotebookX123*"});	SELECT JSON_OBJECT(p.product_data.*Full_name*, …) FROM product p;

D. Oracle 19c

The version 19c introduced noticeable syntax improvements like the syntax of the SQL_OBJECT function. This version supported wildcard "*" to reference all columns in one step, equal to its "traditional" use within the SQL SELECT statement. A more flexible application of this function is also possible through the parametric listing of the column names, which appear as the key names of the resulting JSON object. The JSON_SERIALIZE function enabled conversion from any supported data type to text, primarily used for conversion from a BLOB column or to convert the results of an SQL/JSON function (which, using the RETURNING clause, returns the result in binary format). However, the most important innovation introduced by the version 19c is the elimination of the JSON data update limitations, which had been present since the version 12c R1. Namely, until version 19c, it was impossible to update part of the JSON document. In previous versions, if it was necessary to change the value for a key or add a new key-value pair, the solution was to update the entire JSON document, which resulted in ineffectiveness and increased possibility of errors (like the omission of key-value couples and mistyping). The JSON_MERGEPATCH function, introduced in 19c, has eliminated these shortcomings. It has also enabled updating the values within the existing key-value pair and adding a new couple, with the default retention of other key-value couples [9].

E. Oracle 21c

Oracle's JSON data type, known as OSON, is an optimised JSON binary format. OSON has made it possible to specify the JSON type within the CREATE TABLE statement after defining the column name, allowing the table column to use JSON type in the same manner as other predefined types, such as number, varchar, date, etc. When executing an insert statement over a table containing a JSON column, a valid JSON can be passed directly, as a value in single quotes, or as a JSON constructor parameter. This constructor supports parameters of type VARCHAR, CLOB, and BLOB. Dedicated SQL/JSON display functions in the version 21c also support a JSON type parameter. In addition to the mentioned functions, JSON_SERIALIZE has also been enhanced with support for the JSON data type. Forwarding a JSON column to this function makes it easier to display the resulting JSON document. For displaying binary data is not possible to use "dot" notation alone. Function JSON_SERIALIZE can receive JSON fragment, accessed using "dot" notation. INSERT statement, with a JSON constructor for JSON column, is used to insert data. Introduced function JSON_TRANSFORM improves function JSON_MERGEPATCH from 19c and uses the SET operator, which is intuitive for updates. Whether the name of a key, passed to this function, is present in the document or not, an update or a new key-value insert performs. In addition to the SET operator, other operators can be used within the JSON_TRANSFORM function for explicit insert, replace, append, rename and remove [14].

The limiting factor in the process of selecting Oracle DBMS versions for testing was that, at the time of writing, version 21c was available for download exclusively as a client for Linux. Enterprise edition installations provide full support of all DBMS functionalities. Because of that, they are suitable for testing impact on performance (more precisely, statement execution time) across different versions of Oracle DBMS. As explained, 21c was not considered, and the selection included the version that introduced native JSON support (12c R1) and the latest available Enterprise edition version (19c).

IV. EXPERIMENTAL RESULTS

For conducting practical testing of supported JSON functionalities, an environment set up on a PC with an i7 2.9 GHz CPU, 16 GB of RAM, and SSD HDD and Windows OS was used. For testing, a model was created using UML class diagram (Figure 1). Testing included INSERT, UPDATE, and SELECT statements, over 10, 100, 1000, and 10000 rows. All executed statements included JSON data. INSERT statement added the entire record (which is obligatory). UPDATE and SELECT were performed over a fragment of the JSON document to test the real-life scenario (update or display certain document elements, not the entire one). Column "product_data" stores JSON document structure and contains the IS JSON validation. Table I shows executed statements.

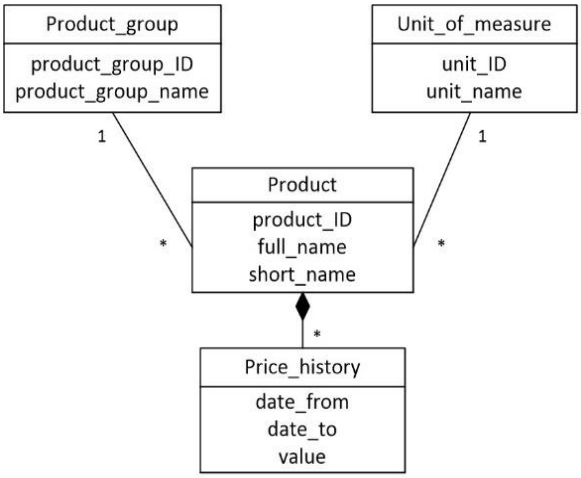

Figure 1 – Created UML class diagram

Each statement for each data set was executed 12 times. Table II shows calculations for the average of 10 executions after eliminating the minimum and maximum execution values to reduce the impact of the outliners. Execution times are shown in seconds.

A review of the execution times shows no significant differences between 12c and 19c versions for the insert statement (Table II). For entering 10 and 100 records, 12c achieved a slight advantage over 19c (0.022 and 0.029 versus 0.029 and 0.033, respectively). In contrast, when increasing the number of records to 1000 and 10000, 19c achieved a slight advantage (0.05 and 0.456 against 0.061 and 0.58, respectively). The explanation can be found in the same syntax for insert among versions 12c and 19c. The above is not the case with the UPDATE and SELECT statements.

TABLE II.
AVERAGE EXECUTION TIMES (IN SEC.) FOR TESTED COMMANDS OVER ORACLE 12C R1 AND 19C

Statement	INSERT			
Record	10	100	1000	10000
12c R1	0.022	0.029	0.061	0.580
19c	0.029	0.033	0.050	0.456
Statement	UPDATE			
Record	10	100	1000	10000
12c R1	0.057	0.064	0.126	0.788
19c	0.040	0.042	0.045	0.151
Statement	SELECT			
Record	10	100	1000	10000
12c R1	0.038	0.059	0.114	0.512
19c	0.025	0.033	0.083	0.286

Namely, to update the data in the version 12c, the UPDATE statement is executed, forwarding the entire JSON document after the SET operator, even if only one element needs to be updated. This is mandatory, in order to keep the other elements unchanged. In the version 19c, this is not the case. Only a fragment of the updated JSON document can be provided, while the other elements retain their values by default. The version 19c accomplishes this using JSON_MERGEPATCH function.

The effect of the absence of partial update support in the version 12c is visible through the achieved times of 0.057, 0.064, 0.126, and 0.788 for 10, 100, 1000, and 10000 records, respectively. Oracle 19c achieved a noticeably shorter execution time: 0.04, 0.042, 0.045, and 0.151 (Table II).

Although the average execution time did not exceed 1 second, the execution time of 19c is noticeably five times shorter on the tested examples with 10,000 records. Finally, the data representation was compared using the "dot" notation. Because of the lack support of JSON_OBJECT function, which is optimized for display, 12c version achieved slower execution times (0.038, 0.059, 0.114, and 0.512 versus 0.025, 0.033, 0.083, and 0.286 in the version 19c).

V. CONCLUDING REMARKS AND DIRECTIONS OF FURTHER RESEARCH

The paper provides an overview of the functionality of working with JSON on different versions of Oracle relational DBMSs. An analysis of the supported functionalities was performed, starting with the first version that introduced support for JSON (12c R1) and concluding with the latest version, which is still unavailable for complete installation (21c). The number and flexibility of functions that handle data, with the structure of JSON documents, have changed through versions. In the version 12c R1, the native JSON was introduced. Oracle 19c is the most stable and feature-rich version for working with native JSON. Representation of this is also visible through the test execution results, in which (especially with 1000 and 10000 tested records) Oracle 19c achieved a noticeably shorter average execution time of data display and data update statements, compared to 12c. These results primarily depict improved syntax for already existing functions (such as JSON_OBJECT) and newly introduced functions (such as JSON_MERGEPATCH). Although the documentation for Oracle 21c has been available since recently, 21c was not selected among candidates, because its installation was not yet available (not as an Enterprise edition like other tested DBMSs, but exclusively as a client for Linux). Therefore, the test compared the first and "the most mature" version of the DBMS that supports native JSON. The comparison with 21c and its binary JSON will represent the direction of future research when the availability of the installation allows it.

REFERENCES

[1] JsonOrg, 2021. Introduction to JSON. Available on: https://www.json.org/json-en.html. Retrieved: January 2021.

[2] Bjeladinović, S., Babarogić, S., Marjanović, Z., 2012. Comparison of relational and NoSQL systems, Proceedings of XIII International Symposium SymOrg 2012, ISBN: 978-86-7680-255-5, pp. 974-980, FON, Belgrade, Serbia.

[3] Bjeladinovic, S., 2018. A fresh approach for hybrid SQL/NoSQL database design based on data structuredness. Enterpr. Inf. Syst. 12 (8–9), 1202–1220. http://dx.doi.org/10.1080/17517575.2018.1446102.

[4] Solid IT, 2020. DB-engines ranking. Available on: https://db-engines.com/en/ ranking. Retrieved: December 2020.

[5] Oracle Base, JSON Support in oracle database 12c Release 1 (12.1.0.2). Available on: https://oracle-base.com/articles/12c/json-support-in-oracle-database-12cr1. Retrieved: January 2021.

[6] JSON in Oracle Database. Available on: https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246. Retrieved: January 2021.

[7] Oracle Base, JSON Support Enhancements in Oracle Database 12c Release 2 (12.2). Available on: https://oracle-base.com/articles/12c/JSON-support-in-oracle-database-12cr2. Retrieved: January 2021.

[8] Oracle Base, JSON_EQUAL Condition in Oracle Database 18c. Available on: https://oracle-base.com/articles/18c/json_equal-condition-18c. Retrieved: January 2021.

[9] Oracle Base, JSON_MERGEPATCH in Oracle Database 19c. Available on: https://oracle-base.com/articles/19c/json_mergepatch-19c. Retrieved January 2021.

[10] [Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. 2014. JSON data management: supporting schema-less development in RDBMS. International Conference on Management of Data (SIGMOD '14). Association for Computing Machinery, New York, NY, USA, 1247–1258.

[11] Survey on JSON Data Modelling, Teng Lv, Ping Yan and Weimin He, Journal of Physics: Conference Series, Volume 1069, 3rd Annual International Conference on Information System and Artificial Intelligence (ISAI2018) 22–24 June 2018, Suzho

[12] Piech, M., & Marcjan, R. (2018). A new approach to storing dynamic data in relational databases using JSON. Computer Science, 19(1), 3. doi:https://doi.org/10.7494/csci.2018.19.1.2505

[13] Mehmet Salih Deveci, Oracle Database Version History & Oracle Release Versions, September 2, 2019. Available on: https://ittutorial.org/oracle-version-history-oracle-database-release-versions/

[14] Oracle Base, JSON Data Type in Oracle Database 21c. Available on: https://oracle-base.com/articles/21c/json-data-type-21c. Retrieved: January 2021.