Abstract - JSON is a popular and proven standard for specifying self-describing text files with a flexible structure. To maintain its position in the market, Oracle introduced support for JSON data in the 12c R1 version of its DBMS. This version has introduced functions for storing and managing JSON data in native form but also showed some limitations. Each new version introduced new or updated JSON functions. The 21c can store JSON data in binary form, provides more straightforward syntax and even supports JSON as a predefined data type. The paper aims to compare the performance when the underlying storage of JSON is native or binary. A data model and seven use cases were designed to demonstrate earlier and new functionalities. Additionally, experiments showed the impact of JSON data stored in native (19c and 21c) and binary form (21c) on the average execution time and costs of SQL statements.

Keywords: JSON, Oracle, relational DBMS.

1. INTRODUCTION

JavaScript Object Notation (often referred to by an acronym - JSON) is a widely known file format predominantly used in data exchange between software applications written in different programming languages [1]. JSON files can transfer not only content but also the structure of the exchanged content, which contributes to overall interoperability. JSON stores data as key-value pairs; the key is the name of the attribute used for referencing, and the value is the particular value of that attribute. By following such principle, JSON simplifies writing and understanding documents by machines and programmers. JSON is platform-independent and highly compatible with notations used in modern NoSQL database management systems [2]. A well-known representative for storing JSON is a document-oriented MongoDB [3]. In March 2023, this database was ranked as the 5th most popular DBMS [4]. JSON popularity led to relational DBMS having to adjust data storing not only in XML format but also in JSON format to stay relevant in the market. Oracle DBMS, being one of the biggest manufacturers of relational databases, has implemented several solutions for manipulation with JSON files through different versions.

1.1 Motivation

Oracle introduced mechanisms for working with native JSON in version 12c Release 1. In later versions, Oracle added and improved methods for working with native JSON formats, such as supporting not only native JSON but also binary JSON. This paper aims to give a systematic survey of changes that happened from when this kind of format was first introduced (12c) to the latest version (21c) and assess functional, syntax, and performance changes that occurred. In addition to the literature overview, practical experiments were performed to determine significant differences between versions 19c and 21c by running various queries and statements that include manipulation of fields that hold JSON values. The overall goal was to compare and analyze each version's enhancements, focusing mainly on switching from native to binary JSON in the latest two versions.

1.2 Research Questions

The paper aims to treat the following research questions:

1) Which JSON functionalities has Oracle relational DBMS introduced, and how have those functionalities upgraded and changed through time and versions?

2) What is the essence of changes introduced with the binary form for storing JSON data in version 21c?

3) How do the new methods for working with JSON data stored in binary form compare to native ones regarding average execution time, statement cost, and syntax simplicity?

1.3 Methodology

In order to answer the research questions, this paper uses various methods. Firstly, a literature overview was conducted to acquire relevant knowledge on similar topics. Based on the analyzed works, the motivation for this paper emerged. Deduction helped to formulate research questions.
The official Oracle documentation represents the source of methods for working with JSON format through different versions. The analysis and comparison of the documentation led to testing those methods and comparing their performance. The chosen method for testing was the comparison of specific use cases on the same data in different versions. The tool used for this phase was the SQL Developer tool for running statements for the defined use cases and measuring execution time which is used as a performance indicator.

This paper has seven sections. Related work and relevant papers on this topic were analyzed and discussed in the second section. The overview, description, and comparative analysis of JSON functionalities are in the third section, as well as the description and explanation of every novelty within each new version of Oracle. The fourth section consists of the data model and test use case specifications. The report from the test use cases is in the fifth section, alongside with graphical depiction of the results. The sixth section presents a conclusion and ideas for further research, followed by a list of references.

2. RELATED WORK

The JSON model represents the basis of document-oriented NoSQL databases [5]. Unlike the ACID characteristics of relational database transactions, NoSQL databases provide BASE transaction properties in addition to schema-less support, flexible structure, and suitability for data analytics. However, the NoSQL document store databases lack some benefits relational databases offer. Relational databases provide declarative language that enables complex statement-based queries and advanced processing. The databases can also manage transactions, which is crucial for maintaining data consistency and integrity. The authors [6] provide a detailed comparison between Oracle Database, an SQL database management system, and MongoDB, a NoSQL document-oriented database management system, covering various aspects such as theoretical differences, features, restrictions, integrity, distribution, system requirements, architecture, query performance, and insertion times.

On the other hand, the paper [7] evaluates document-based and relational databases using MongoDB and MySQL, considering data storage, management, and performance for CRUD operations, highlighting the pros and cons of each model. The flexibility of the NoSQL model, and the need for support of complex operations such as multiple joins, group by, etc., supported in relational databases, reflected the emergence of the symbiosis of the JSON model and relational databases. However, to defend their leading market position while mixing the proven consistency with the flexibility of semi-structured data, relational databases introduced the support for JSON data. The paper [8] proposes storing and managing JSON data in relational database systems for transaction management and concurrency control advantages, comparing it to NoSQL document stores, and exploring the adoption of schema-oblivious XML-to-Relational mapping techniques for JSON-to-Relational mapping. Additionally, some authors [9] highlight integrating JSON into relational databases to leverage the benefits of standard SQL features, ACID transaction model, and efficient data management, enabling a combination of fast JSON development, relational consistency, and SQL efficiency for analysis and complex queries. However, Liu et al. [10] were among the first to address the advantages of introducing the JSON model into relational databases and supporting the mapping of JSON objects into predefined types of relational databases. In [11], the author explores the implementation of JSON data update operations in relational database systems, specifically MySQL, PostgreSQL, and SQL Server, comparing various approaches and evaluating their compatibility with a framework of primitive update operations while assessing their coverage.

Oracle 12c R1 introduced the support for managing native JSON in relational databases [12]. This version is the first version in the series with a "c" suffix that replaces the previous suffix "g" that denoted grid technology. Oracle 12c R1 indicates a significant technical improvement with the introduction of database cloud service. Liu et al. [10] emphasize that the usage of native JSON in Oracle 12c R1 can lead to more efficient and effective integration of relational data and data with varying structures into a single database. Although native and binary JSON share many similarities in structure and syntax, BSON (binary JSON) is essentially a binary-encoded serialization of JSON. It is designed to provide additional features such as improved performance and support for more data types (dates, double and binary data) while maintaining compatibility with JSON. Binary JSON provides a simpler syntax and supports JSON as a predefined data type on which a column or variable can be directly created [13].

Liu et al. [10] realized that there was a need for binary JSON support. They decided to implement it and make it more accessible to everyone and easier to use, even before it became an integral part of Oracle DBMS. Their research focused on two specific features – JSON DataGuide (metadata summary of the structural and type information contained in a set of JSON documents) and OSON (Oracle JSON) binary format. The authors’ follow-up work [14]
introduced advancements, such as the tested binary format for the JSON data type. However, due to the absence of Oracle 21c on the market at the time, no tests were conducted on that particular version. Nevertheless, their research has significantly impacted the exploration in this area.

The XML is considered the ancestor of JSON due to its role in enabling the exchange of structured and precisely formatted data. JSON, on the other hand, was created as a lightweight alternative but still uses many of XML's principles that enable the unification of the data interpretation. The authors' analysis [15] indicates that in terms of exploring various methods for modelling JSON data, it is vital to find a balance between functionality, complexity, efficiency, and the ability to expand.

The evidence of JSON's importance for relational databases can be observed in the examples of PostgreSQL [16], MySQL [17], Microsoft SQL Server [18], and Oracle, which all have introduced some level of JSON support in their systems. The source [12] gives an overview of the evolution of JSON in Oracle.

The authors [19] point out that PostgreSQL offers a particularly robust level of support. The paper outlines the specific versions in which each DBMS introduced native JSON support and binary JSON support, where applicable. It also provides some additional details about the approach taken by each system. Overall, the paper aims to provide a comprehensive look at the advantages and disadvantages of using JSON in databases, such as flexibility, ease of use, and potential performance issues in storing complex data structures. Moreover, Oracle introduced support for more compact and potentially faster binary JSON in the 21c release.

As this type of database is widely used for storing and managing structured data, authors [20] compare the performance of different relational databases. Regarding performance analysis, authors in their work [21] point out the performance analysis of MySQL and SQL Server RDBMS. Each DBMS constantly tries to deliver and improve its performance. Every new edition introduces new functionalities, but there is always a trade-off between the versions, performance, and functionalities. Through their works [22] and [23], some authors have dealt with JSON in relational databases.

The examined papers analyzed the JSON functionalities in different versions of Oracle DBMS. The new 21c version introduced practical support for binary JSON, which allows an expansion of the previous research through experiments made on the same data set. The experimentation allows a better understanding of the benefits and limitations of using binary over native JSON.

Moreover, this creates a basis for perceiving and analyzing the differences between binary and native JSON functionalities in Oracle DBMS. In their paper [24], the authors introduce different functionalities used for storing and reading JSON documents throughout various Oracle versions and analysis and experimental results. The previous papers analyze the difference between native and binary JSON from a theoretical point of view. Besides the theoretical analyses of functionalities presented in the source [24], the recent introduction of the Oracle 21c version establishes the possibility of practically testing new syntax, new functionalities, and their impact on the selected performance indicator, i.e. execution time and cost.

3. ANALYSIS OF SUPPORTED JSON FUNCTIONALITIES

This chapter includes an overview and analysis of supported JSON functionalities in different Oracle versions (12c R1, 12c R2, 18c, 19c, 21c).

3.1 Oracle 12c R1

Oracle provided support for JSON for the first time in version 12c R1. In this version, columns that store JSON are not of a predefined JSON data type but a VARCHAR or CLOB type used to store JSON documents (other types, such as BLOB, NCLOB, or NVARCHAR, are rarely used). Because none data stored as VARCHAR would adhere to the JSON rule by itself, a particular CHECK constraint (IS JSON) needs to be defined on previously mentioned fields to ensure that the data follow the rules for JSON formatting. Therefore, the mentioned constraint acts as a JSON validator. IS JSON uses a lax syntax which refers to less strict JavaScript syntax for object fields that supports non-case-sensitive boolean and null values. Strict JavaScript JSON validation must be explicitly stated, as it is not considered a default option. More detailed information on the difference between those two syntaxes can be found in the source [25]. There are no differences in inserting JSON data compared to other data types. Data insertion uses a simple INSERT statement that expects a valid JSON format document. JSON data validation, before INSERT, should be done through external tools. As far as updating data is concerned, it is done by replacing the whole document with the updated one, i.e., it is impossible to modify just one part of the JSON document. The consequence is that a new record holds updated values but also needs to hold unmodified values copied from the previous version of the document. Oracle 12c R1 offers multiple options for the display of JSON data. One option would be to form a query for nested JSON elements, which would start from the first (root) element and move through the document by using the dot notation. This option has an intuitive syntax,
but its case sensitivity can be a problem, e.g., using a code formatting tool in SQL Developer can change letters’ cases, which leads to different query results.

One of the drawbacks of dot notation is the lack of differentiation between an element that does not exist (and, therefore, does not hold any value) and an element that exists but does not hold any value [26] because the result in both cases will be null. Using the JSON_EXISTS method in the WHERE clause can solve this problem, which returns null only for existing elements without any value and completely ignores undefined elements and does not show them in the result set. JSON_VALUE and JSON_QUERY, the same as the previous method, use JSON Path. The return value of the JSON_VALUE method is a single scalar value and not a collection of elements or nested elements, whereas JSON_QUERY passes a part of the document as a result [26]. JSON_TABLE is a method that combines the functionalities of the previous three methods. Even though its syntax is more complex, JSON_TABLE method usage increases the query-building efficiency because it eliminates the need for writing multiple queries [26]. Its syntax supports customization of the order and structure of the resulting columns, as well as filtering it on certain conditions in the same manner as views are made in relational databases.

As the support for JSON format is introduced in this version, there are few methods for data manipulation, and there is still a vast difference between predefined types and JSON data types. JSON type is implemented by using other types with additional constraints.

### 3.2 Oracle 12c R2

A vital upgrade of version 12c R2 is the ability to use the methods from version R1 in PL/SQL. Apart from that, JSON/SQL methods can generate JSON from SQL. One such method is JSON_OBJECT which specifies a list of key-value pairs that allows the user to display a result set as a JSON object even though their source data type might differ. Even if the data is stored as a table record, made up of only predefined types (VARCHAR, NUMBER, etc.), the result would still have a structure of a JSON object with a list of key-value pairs passed as arguments in the JSON_OBJECT method. The JSON_OBJECTAGG function for aggregate creation of the object creates a JSON object from multiple query results. The structure of individual objects within the array may differ, i.e., for each object, a record of its key-value pairs is formed. Another method, JSON_ARRAYAGG, is a method used to combine and aggregate records that fulfill specific criteria into a single JSON array. This method receives the expression determining which records meet the requirement [26]. In this version, support for working with PL/SQL has been improved by offering new types of objects for generating, managing, and parsing JSON objects in memory, such as JSON_ELEMENT_T, JSON_ARRAY_T, JSON_KEY_LIST, JSON_OBJECT_T, JSON_SCALAR_T [27].

One of the main upgrades in this version is additional methods for working with JSON format that made creating and interacting with JSON objects and arrays easier.

### 3.3 Oracle 18c

The main improvement in this version of Oracle, concerning working with JSON, is an expansion of return data types. In the previous versions, the default return data type of SQL to JSON functions was a VARCHAR data type with a default length set to 4000. In some cases, methods JSON_OBJECTAGG and JSON_ARRAYAGG could return data in CLOB data type. Version 18c enabled CLOB or BLOB as return values for other SQL/JSON methods (such as JSON_OBJECT and JSON_ARRAY) alongside the SQL_QUERY method [27]. Another useful improvement in this version is TREAT (... AS JSON) method, which treats a column that does not have IS JSON check constraint, as it does. The syntax of JSON_TABLE is simplified. The new JSON_EQUAL method returns true if the content of two JSON documents is the same regardless of formatting and order of elements inside, which was not the case in the previous version, where the comparison was made as a string comparison [28].

This version brought more methods for easier JSON data manipulation, as well as a more efficient method for comparison that is sensitive to different placement of fields inside of the document.

### 3.4 Oracle 19c

This version established different syntax changes making the existing functionalities easier and more intuitive. The best example is the SQL_OBJECT method that added a wildcard character "*". This character is used to reference all columns instead of listing them individually and is analogous to selecting all data at once in SQL. Considering that argument cannot contain the key-value pairs, the result will consist of column names for keys. At the same time, table rows that satisfy the defined criteria will populate the values. Another option that made the use of the previous method more flexible is through listing column names as parameters which represent keys in the resulting JSON document. The column names passed as parameters are case-sensitive. Still, the column names in tables are not, which results in matching the corresponding column names and keys from the parameter list, while the key in the document is still case-sensitive. This technique can also be combined with using the ":" sign; the key names should be written left from the sign, and the
column name that holds the corresponding value should be written after the sign. Version 19c introduced this shortcut as an alternative to the traditional writing of key-value pairs from previous versions. For displaying data outside the JSON data columns, in previous versions, Oracle used the JSON_TABLE method in combination with left outer joins. This version simplified data display by introducing a NESTED clause which eliminates the need for using left joins and the JSON_TABLE method because the column names that would be listed as parameters in the JSON_TABLE method are passed after the NESTED clause. This clause also supports direct access to fields nested inside the JSON document, i.e., a column inside of another column. JSON_SERIALIZE is a feature that allows data to be transformed from any data type into text format, enabling easy manipulation and storage of data in JSON. It can convert data from BLOB or transform the result from some SQL/JSON methods that usually return results in binary format.

The most significant upgrade of this version was overcoming the unnecessary update of the whole document. That problem occurred while modifying some (but not all) elements in the document. Prior to this version, it was impossible to update only the elements that have changed; the only way to do it was to replace the whole old document with a completely new one that would contain the modified values and the same, unmodified values which lead to inflexibility and higher probability of errors (that would occur during the frequent copying of unmodified values). This version introduced the improvement through the JSON_MERGE_PATCH method. This method can change or add specific key-value pairs without changing the rest of the document, which holds the same values by default [29]. This method made it easier and more intuitive for users to change only the parts of the document that they want to without wasting resources to copy the old values. JSON_MERGE_PATCH can also show the data in the SELECT clause without changing the database.

The main upgrade that was introduced in this version was the ability to directly access and change only the updated parts of the JSON document compared to the previous versions, where that was not an option which led to more efficient updates.

3.5 Oracle 21c

JSON has undergone several improvements over the years, introducing new functionalities that have addressed some of its significant limitations. JSON_MERGE_PATCH resolved the problem of partial updates in version 19c but has still not achieved the simplicity of using predefined data types.

Meanwhile, Oracle first mentioned the JSON data type that represents an optimized format of binary JSON in version 20c but achieved wider functional availability in version 21c. OSON has made it easier to work with JSON data type when specifying a column within the CREATE TABLE command. Declaring a JSON column with the same syntax as columns of any other predefined type is possible. When inserting data into a table that contains a JSON column, there are several ways to pass the JSON data directly, within the single quotes, or as a JSON constructor parameter.

Since the data is stored in binary format, the support of JSON data type is given through dedicated SQL/JSON functions. Execution of JSON_VALUE, JSON_QUERY, and JSON_TABLE obtain the same result as in previous versions but now based on the JSON column and JSON_PATH to the specific element. As the JSON_SERIALIZE function also supports passing JSON column for the more straightforward rendering of the resulting JSON document, it is not allowed to use "dot" notation to represent binary notation, but only to access a fragment of a JSON column. JSON_SCALAR function, used to generate an instance of JSON based on a scalar value, can contain the JSON_SERIALIZE function as an argument and then display the binary record in text format.

The INSERT statement and JSON constructor, which accepts values of different types (VARCHAR, CLOB, and BLOB), can achieve data entry and value passing for the JSON column. From version 19c and the JSON_MERGE_PATCH function, it was unnecessary to update the entire document when inserting or updating the key-value pair. Version 21c introduced the JSON_TRANSFORM function that uses the SET operator followed by key-value pair or other operators for explicitly entering a new element – INSERT, adding an element at the end of the collection – APPEND, explicitly updating the value – REPLACE, for deleting an element – REMOVE, etc. [30].

Oracle 21c version supports working with JSON fields very similarly to other predefined data types and simplifies partial updates of JSON documents introduced in version 19c.

4. DATA MODEL AND USE CASES SPECIFICATION

Figure 1 displays the created data model for conducting practical testing and comparison of the native and binary JSON. The given data model describes a part of the banking system related to clients and their cards, client type and tax group. One client belongs to only one tax group based on which the taxes are calculated and later paid. Client type refers to the type of bank client that entitles the client to the specific services and operations they can perform. Each client can have zero to many cards,
which can only exist and be identified if they belong to a particular client.

JSON functionalities tests involved using Oracle DBMS Enterprise Edition versions 19c and 21c. Oracle 19c is a version with the most mature and complete functionalities for working with native JSON, and it is a representative of a DBMS with native JSON support. Oracle 21c is a representative of Oracle DBMS with binary JSON, as it is essentially the first and most current version that it supports. In order to demonstrate the complex and nested structure typical for working with JSON data, the entire model is implemented in one table column. Table 1 shows seven use cases (UC_1, UC_2, UC_3, UC_4, UC_5, UC_6, UC_7) and their specifications created for testing purposes. The third and fourth columns list how to implement the described statements in versions 19c and 21c.

The idea behind selecting these use cases was to cover all CRUD (Create, Read, Update, Delete) operations typical for working with data. In SQL terminology, these operations are covered with DML (Data Manipulation Language). Besides DML statements, one DDL (Data Definition Language) statement is selected. The first use case represents the CREATE TABLE statement because creating a table with JSON data type precedes the usage of the DML operations on JSON data. UC_2 and UC_3 cover select JSON data. Two types of UPDATE are presented with UC_4 and UC_5. UC_6 deals with INSERT, while UC_7 covers DELETE.

Use cases UC_2, UC_3, UC_4, UC_5 and UC_6 are implemented in two ways in 21c. The first is marked with (N) and typically contains new or extended functionality of version 21c compared to 19c. In use cases marked with (O), underlying data is in native form. The second way (labelled with O) represents the implementation of a specific use case using functionality also available in 19c. In the third column (for the implementation in 19c), therefore, all functionalities are marked as (O).

UC_2 describes selecting the entire JSON object. JSON_OBJECT function selects objects stored in native form (Oracle 19c (O) and 21c (O)), while the JSON_SERIALIZE function selects the entire object stored in binary form. UC_3 further elaborates on selecting data and illustrates selecting part of JSON object with WHERE clause. UC_4 and UC_5 illustrate Object UPDATE and Array UPDATE, respectively. JSON_MERGE_PATCH function is used for selecting JSON data stored in native form for both cases, as opposed to JSON_TRANSFORM used when JSON data is stored in binary form. UC_6 describes inserting data into a table which contains JSON data. JSON constructor is used for inserting when JSON data is in binary form while forwarding a JSON document was used for inserting when JSON data is in native form. UC_7 illustrates deleting JSON data where DELETE with JSON_OBJECT function is used when JSON data is stored in both native and binary form.

Version 19c implements the model as a column of type CLOB with a check constraint used by IS JSON (O), while in version 21c, the corresponding column is just a predefined type of JSON (N). The Create table statement is shown in Table 1 and represents the first tested use case (UC_1).

UC_2 represents a use case for selecting and displaying JSON data. Version 19c uses the JSON_OBJECT function in combination with dot notation. Although the JSON_SERIALIZE function (for conversion from supported types to a text record) was introduced in the 19c version, practical tests achieved almost the same execution time as the JSON_OBJECT function. The explanation lies in the fact that only since version 21c is the JSON data type fully supported. For this reason, 21c tests use the JSON_SERIALIZE function (N) and previously available JSON_OBJECT (O). In addition, the fetching of the last row is included in the average execution time of the SELECT statement. Fetching inclusion eliminates the unrepresentative display time of only the first 50 rows, which is the default setting within the used tool.

The third use case (UC_3) uses the same functionalities for 19c and 21c but combined with the WHERE clause.

The fourth (UC_4) and the fifth use case (UC_5) represent UPDATE statements, with the difference that in UC_4, a JSON object is updated. UC_5 updates the array within the JSON document. Versions 19c and 21c use JSON_MERGE_PATCH as the old functionality (O), while 21c implements JSON_TRANSOFRM (N) as the new functionality.
<table>
<thead>
<tr>
<th>Use case</th>
<th>Statement</th>
<th>Implementation in 19c</th>
<th>Implementation in 21c</th>
</tr>
</thead>
</table>
| UC_1 | CREATE table with JSON data | (O) Creating a table with a CLOB column and CHECK constraint 
CREATE TABLE client_json_native (client_id number, client_data CLOB constraint client_json_check check (client_data IS JSON)); | (N) Creating a table with JSON data type 
CREATE TABLE client_json_binary (client_id number, client_data JSON); |
| UC_2 | SELECT JSON object | (O) Using the JSON_OBJECT function with dot notation 
SELECT JSON_OBJECT( 
client_data.Client.Client_ID, 
client_data.Client.Client_type.Client_type_ID, 
client_data.Client.Tax_group. Tax_group_ID, 
client_data.Client.Card[0].CCV, ... 
) FROM client_json_native c; | (N) Using the JSON_SERIALIZE function 
SELECT client_id, json_serialize(client_data) as json_data from client_json_binary; |
| UC_3 | SELECT JSON object with WHERE | (O) Using the JSON_OBJECT function with dot notation for selecting and filtering 
SELECT JSON_OBJECT( 
client_data.Client.Client_ID, 
client_data.Client.Client_type.Client_type_ID, 
client_data.Client.Tax_group. Tax_group_ID, 
client_data.Client.Card[0].CCV, ... 
) FROM client_json_native c 
WHERE (SELECT JSON_OBJECT( 
client_data.Client.Client_name) FROM client_json_native ...) = '{"Client_address":...}'; | (N) Using the JSON_SERIALIZE function and JSON_OBJECT for filtering 
SELECT client_id, json_serialize(client_data) as json_data from client_json_binary 
WHERE (SELECT JSON_OBJECT( 
client_data.Client.Client_name) FROM client_json_binary c) = 
'{"Client_address":...}'; |
| UC_4 | Object UPDATE | (O) Using the JSON_MERGEPATCH function 
UPDATE client_json_native c 
SET c.client_data = JSON_MERGEPATCH( 
client_data, 
{"Client":{"Client_name":"..."}, 
"id":...} 
); | (N) Using the JSON_TRANSFORM function 
UPDATE client_json_binary SET client_data = JSON_TRANSFORM( 
client_data, set 
"$.Client.Client_name" = '...'; 
); |
| UC_5 | Array UPDATE | (O) Using the JSON_MERGEPATCH function 
UPDATE client_json_native c 
SET c.client_data = JSON_MERGEPATCH( 
client_data, 
{"Client":{"Card": 
[
"Card_valid_from":...", "CCV":...
]}, 
"id":...}) WHERE ...; | (N) Using the JSON_TRANSFORM function 
UPDATE client_json_binary SET client_data = JSON_TRANSFORM(client_data, set 
"$.Client.Card[0].Card_valid_from" = '"...", set 
"$.Client.Card[0].CCV" = '"..." WHERE ...; |
| UC_6 | INSERT | (O) Forwarding a JSON document 
INSERT into client_json_native (client_id, client_data) 
...; 
"Client":{ 
"ID":..., "Name":...
}, 
"id":...}; | (N) Using the JSON constructor 
INSERT into client_json_binary (client_id, client_data) 
...; 
JSON({ 
"Client":{ 
"ID":..., "Name":...
}, 
"id":...}); |
| UC_7 | DELETE objects with WHERE | (O) DELETE with JSON_OBJECT 
DELETE from client_json_native 
WHERE (SELECT JSON_OBJECT( 
client_data.Client.Client_name) FROM client_json_native c 
WHERE ...) = '{"Client_name":...}'; | (N) DELETE with JSON_OBJECT 
DELETE from client_json_binary 
WHERE (SELECT JSON_OBJECT( 
client_data.Client.Client_name) FROM client_json_binary c 
WHERE ...) = '{"Client_name":...}'; |
UC_6 includes the entry of new records, and 19c and 21c (O) use direct forwarding of the JSON document, while 21c (N) implements new functionality with the JSON constructor.

The seventh use case deletes records, and it does not contain any version specifics, so it is marked (O) for 19c and (N) for 21c.

5. EXPERIMENTAL RESULTS

A virtual private cloud server (VPS) with Windows OS and four vCPU cores, 8 GB RAM and 200 GB SSD HDD was set up to practically test JSON functionalities and performance in 19c and 21c of Oracle Enterprise Edition DBMS. Testing in such an environment made it possible to conduct testing for each use case under the same conditions, which gives the data credibility and validity. UML class diagram (Figure 1) presents a model used to perform designed use case tests.

Analyses included seven use cases (CREATE TABLE, SELECT, SELECT with WHERE, UPDATE object, UPDATE array, INSERT and DELETE statements) on seven different data sets of 10,000, 25,000, 50,000, 75,000, 100,000, 250,000 and 500,000 rows.

Both DBMSs executed each statement 12 times on each data set. The final results include average execution times in seconds for each data set, calculated from 10 executions after removing the minimum and maximum values to avoid the influence of outliers. The following survey includes analyses of execution times, syntax complexity and resource cost for the mentioned seven use cases. Flushing buffer cache and shared pool memory was conducted after each test execution to ensure a consistent environment.

5.1 UC_1 (CREATE table with JSON data)

UC_1 in versions 19c and 21c depicts syntax differences. The average time was not measured for two reasons. The first is that the command to create a new table, which contains JSON data, is executed negligibly rarely compared to the other analyzed statements. Hence, the execution time is not essential. In addition, the performance of UC_1 is not affected by the amount of data, unlike the other analyzed use cases and CREATE AS SELECT statement. However, for this statement, it is essential to analyze the simplicity of the syntax in each of the versions. Oracle 21c uses dedicated and predefined JSON data type in binary format. Therefore, columns and variables can be of JSON data type. It also introduced JSON constructor and supported existing functions for working with JSON.

Contrary to this, Oracle 19c stores JSON data in VARCHAR or CLOB type columns (optionally NVARCHAR, NCLOB and BLOB). Additionally, it requires IS JSON constraint or even IS JSON (strict) constraint to validate data as JSON. Changes in 21c noticeably simplify the syntax for creating a table with JSON column, and that represents an essential benefit in comparison to 19c. Table 1 shows the difference in syntax for both Oracle versions.

5.2 UC_2 (SELECT JSON data)

UC_2 shows the difference between SELECT with the fetch of the last record in 19c and 21c, using JSON_OBJECT with dot notation (O), and using 21c with JSON_SERIALIZE function (N).

Figure 2 shows the average execution time of the UC_2. The 19c (O) has achieved $1.281\, (10,000\, \text{records}),\, 2.067\, (25,000\, \text{records}),\, 3.384\, (50,000\, \text{records}),\, 4.53\, (75,000\, \text{records}),\, 6.198\, (100,000\, \text{records}),\, 14.602\, (250,000\, \text{records}),\, 34.836\, (500,000\, \text{records})\, \text{seconds\, on\, average}\). The 21c (O) took $0.793\, (10,000\, \text{records}),\, 1.496\, (25,000\, \text{records}),\, 2.973\, (50,000\, \text{records}),\, 4.16\, (75,000\, \text{records}),\, 4.94\, (100,000\, \text{records}),\, 13.343\, (250,000\, \text{records}),\, 29.135\, (500,000\, \text{records})\, \text{seconds\, on\, average}\). Noticeably the fastest was 21c (N) with $0.718\, (10,000\, \text{records}),\, 1.132\, (25,000\, \text{records}),\, 2.461\, (50,000\, \text{records}),\, 3.617\, (75,000\, \text{records}),\, 3.97\, (100,000\, \text{records}),\, 11.137\, (250,000\, \text{records}),\, 21.674\, (500,000\, \text{records})\, \text{seconds\, on\, average}\).

Results show that Oracle 21c generally has faster execution times over Oracle 19c for both observed options. It is noticeable that Oracle 21c (N) has the shortest average execution times, where the trend line shows that the difference increases with the number of records. Average execution times correspond to statement costs of 1,728; 9,136; and 6,029 for 19c (O), 21c (O) and 21c (N), respectively.

By analyzing UC_2, it can be concluded that 21c is more suitable for selecting JSON data in binary form.

5.3 UC_3 (SELECT JSON with WHERE)

UC_3 extends the statement from UC_2 with the WHERE clause while keeping other functions, JSON_OBJECT with dot notation, and JSON_SERIALIZE for working JSON data stored in native and binary form.
Figure 3 illustrates the average execution time of the UC_3. The 19c (O) has achieved 0.714 (10,000 records), 1.591 (25,000 records), 2.711 (50,000 records), 3.564 (75,000 records), 5.71 (100,000 records), 12.898 (250,000 records), 27.817 (500,000 records) seconds on average. The 21c (O) took 0.577 (10,000 records), 1.661 (25,000 records), 2.625 (50,000 records), 3.424 (75,000 records), 4.654 (100,000 records), 11.154 (250,000 records), 23.898 (500,000 records) seconds on average. Same as in UC_2, the fastest was 21c (N) with 0.391 (10,000 records), 0.808 (25,000 records), 1.826 (50,000 records), 2.838 (75,000 records), 3.744 (100,000 records), 9.484 (250,000 records), 20.898 (500,000 records) seconds on average.

Observed results are consistent with results from UC_2, where Oracle 21c achieved faster times for working with JSON data in both binary and native form. It is also consistent that working with data in binary achieved shorter average execution times over data in native form. Even though UC_3 performed faster in terms of average execution time, resource cost was much higher, and they correspond to statement costs of 9.875.845; 2.737.340; and 100.000 for 19c (O), 21c (O) and 21c (N), respectively.

Analyses of UC_3 led to a similar conclusion as analyses of UC_2 that JSON data in binary form is more suitable for use cases focused on selecting data.

5.4 UC_4 (Object UPDATE)

The shortest average execution time for UC_4 shows an apparent distinction between using 21c (N) on one side and 19c (O) and 21c (O) on the other. For object updates of 10,000, 25,000, 50,000, 75,000, 100,000, 250,000 and 500,000 records, the 19c (O) took 1.593, 3.091, 4.023, 6.824, 10.448, 19.903, 56.063 seconds in average; the 21c (O) 1.106, 2.963, 5.214, 8.872, 12.956, 27.930, 54.625 seconds respectively, while 21c (N) took 1.609, 3.953, 5.537, 10.183, 15.478, 38.996, 77.568 as can be seen on Figure 4.

Test results for object updates show that 21c (O) achieves faster time with a larger volume of records than Oracle 19c, which achieved better results with 250,000 and fewer records. UC_4 results for resource cost also show that Oracle 21c requires more resources to execute statements where costs for both 21c (O) and 21c(N) were 6,029 opposed to 1,698 for 19c (O). Analyses of UC_4 can lead to the conclusion that old functionalities (O) (available in both 19c and 21c) for working with JSON data are more suitable for updating objects.

5.5 UC_5 (Array UPDATE)

Figure 5 shows the average execution time of the UC_5. The 19c (O) has achieved 1.579 (10,000 records), 2.986 (25,000 records), 4.242 (50,000 records), 6.197 (75,000 records), 9.179 (100,000 records), 19.667 (250,000 records), 37.140 (500,000 records) seconds on average. The 21c (O) took 1.148 (10,000 records), 2.637 (25,000 records), 5.444 (50,000 records), 7.745 (75,000 records), 12.217 (100,000 records), 28.997 (250,000 records), 55.408 (500,000 records) seconds on average. Noticeably the slowest was 21c (N) with 1.169 (10,000 records), 2.731 (25,000 records), 5.605 (50,000 records), 7.998 (75,000 records), 14.354 (100,000 records), 41.473 (250,000 records), 76.221 (500,000 records) seconds on average.
Average execution times show that Oracle 19c is consistently the fastest at executing UC_5 on 50,000 and more records where a significant gap can be seen between 19c (O) and 21c (O) and also between 21c (O) and 21c (N). However, this comes at increased costs (19c) compared to object UPDATE, where 21c (O) and 21c (N) show the same statement cost as UC_4 of 6.029.

Array UPDATE for 19c (O) comes with double costs compared to UC_4 with 3.396. Resource costs for Oracle 19c for working with JSON data are still below the values gathered for Oracle 21c. However, compared to Object UPDATE, Array UPDATE is far more demanding regarding needed resources of 19c, which is not the case for Oracle 21c.

5.6 UC_6 (INSERT)
The average execution time of UC_6 shows that 21 (O) performs significantly faster than 19c (O) as well as 21 (N). On records of 10,000, 25,000, 50,000, 75,000, 100,000, 250,000 and 500,000 21c (N) as slowest achieved 0.512, 1.358, 2.645, 4.133, 5.451, 13.206, 26.071 respectively. Compared to these, 19c (O) performed similarly with execution times of 0.416, 0.829, 1.521, 2.425, 3.976, 12.746, and 28.767. By far, the shortest execution times were attained with 21 (O) with times of 0.327, 0.631, 1.244, 2.126, 2.517, 6.107, and 13.711. Figure 6 shows a graphic representation of execution times.

These results come with the same resource costs for all three observed variations, 19c (O), 21c (O) and 21c (N), with a resource cost of 2. Results show no discrepancy between versions or functionalities for working with JSON data regarding resources. Data show that Oracle 21c using old functionalities is more suited for executing INSERT statements when working with JSON data.

5.7 UC_7 (DELETE)
UC_7 performs faster on Oracle 21c where times achieved on tests for records of 10,000, 25,000, 50,000, 75,000, 100,000, 250,000 and 500,000 of 0.286, 0.656, 1.278, 1.806, 2.294, 5.768, 10.979 compare to 0.725, 0.924, 1.450, 2.636, 4.216, 7.829, 14.553 for 19c as can be seen on Figure 7. It can be concluded that Oracle 21c, with its JSON in binary form, is more adapted for executing Delete statements over JSON data.

Figure 6 - UC_6 Results

6. CONCLUDING REMARKS AND DIRECTIONS FOR FURTHER RESEARCH
The article gives a summary of functionalities between working with the native and binary form for storing JSON data across two versions of Oracle DBMSs 19c and 21c, each chosen as representative based on the latest versions of both supported formats. The analyses addressed syntax complexity, resource cost, and average execution time across functionalities for working with native and binary JSON. Binary JSON in 21c has a more extensive spectrum of functionalities mainly because new have been added and old preserved.

Designed use cases show that the syntax for creating a table that stores JSON data in a native form is more complex than the syntax for JSON data stored in a binary form. This is because the need to apply IS JSON constraint has been eliminated when working with JSON data in binary form. Syntax simplification is also evident with UC_2 and UC_3, where function JSON_SERIALIZE for working with JSON data stored in binary form simplified syntax for selecting entire or part of JSON object. Similarly, the UPDATE statement, executed in UC_4 and UC_5, is simplified with the new JSON_TRANSFORM function. Additionally, JSON_TRANSFORM is an atomic function which gives additional benefits. For deleting JSON data, the syntax is the same for both
observed cases when JSON data is stored in binary or native form. It can be concluded that working with JSON data stored in binary form has a more refined syntax.

It can be concluded that Oracle 19c, across all JSON tests, requires less or the same resource to perform the desired statement, which leads to the conclusion that Oracle 21c is more resource dependent for managing JSON data. Executing UC_2 in Oracle 21c, the distinction between data stored in binary and native form in resource cost can be seen where performing statement over data stored in binary form requires fewer resources than in its native form. Overall, this increased resource cost of Oracle 21c compared to Oracle 19c when working with JSON data can be countered by the fact that many computers working with DBMSs have on-par or better performance than the machine used for testing, with the ability to provide and cover this increased resource demand. Average execution times for tests show that Oracle 21c, when JSON data is stored in binary form, is more suited for situations where use case scenarios mainly consist of selecting JSON data. Results also show that storing JSON data in native form is more suitable when emphasizing the UPDATE statement. Oracle 19c achieved shorter times with Object and Array UPDATE. Only at 500,000 records, Oracle 21c achieved better results in Object update of data stored in native than binary form. When dealing with data insertion, UC_6 results show that data stored in native form in Oracle 21c reached significantly shorter times across all numbers of records. The distinction can also be seen in UC_7, where the DELETE statement is performed faster in Oracle 21c (Figure 7). According to the tested use cases, Oracle 21c has an advantage in the systems needing frequent data insertion and selection. In comparison, 19c has an advantage in frequent data updates when dealing with arrays. In general, Oracle 21c gives the option to use both binary and native forms to store JSON data which provides more versatility in choosing how to create desired tables for storing JSON data.

Use cases presented and achieved results focused on JSON functionality, and other types of statements were not tested (e.g. DML statements over non-JSON data).

The direction of further research represents a comparison of other improvements of JSON data type and optimization of new functionalities introduced with binary JSON at observed versions. Besides that, an important direction for further research represents a comparison between "traditional" SQL statements between Oracle versions 19c and 21c, which will complement the presented work and give a broader answer to the question: Are the achieved results in this test exclusively JSON-related? Comparison of working with JSON data type between different DBMS providers also presents an opportunity for further research.

7. References


