Performance of JSON Updates using Different Storage Forms

Dušan Petković
Technical University of Applied Sciences
Rosenheim, 83024, Germany

ABSTRACT

This paper discusses performance of modification operations on JSON documents, stored in different relational storage forms. The first form is a “raw” document form, meaning that an exact copy of the JSON data is stored into relational table. The second one is the native form of MYSQL, which belongs to a group of binary formats for storing JSON. We discuss first the update primitives, which build a foundation for modification operations on JSON data. The existing forms of SQL UPDATE of the database system are used to implement these operations using two pairs of tables, which significantly differ in relation to their volume. We compare the performance of JSON updates on documents stored using the TEXT data type when data are stored in “raw” document form and using the JSON data type, when data are stored in the native (binary) form. Our study divides the update primitives in two groups: one for the modification operations on objects (name-value pairs) and the other one for the modification operations on arrays. For both groups of operations, we measure the performance when the data are stored in the proprietary binary format and in “raw” document one. Our measures show that for all UPDATE operations, except for the initial loading process, the modification of JSON data stored in the binary form is significantly faster than the modification of the same data stored in the “raw” document form. Additionally, improvements in execution time of update statements are higher in the case of the large JSON documents. In other words, the bigger the JSON document, the more significant the performance gains.

General Terms

Relational database systems

Keywords

RDBMSs, JSON, UPDATE, performance

Artifact Availability:
The data, and/or other artifacts have been made available at https://doi.org/10.6084/m9.figshare.23805153.

1. INTRODUCTION

JSON is a simple data format used for data interchange. The structure of JSON content follows the syntax structure for JavaScript. This data format is built on two structures: a collection of name/value pairs and an ordered list of values (see Figure 1).

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

Figure 1: An example of JSON document

Generally, a JSON string contains either one or more name-value pairs called objects or an array. An array is surrounded by a pair of square brackets and contains an ordered list of values, which are separated by commas from each other. An object is presented inside a pair of curly brackets and contains a comma-separated list of unordered name-value pairs. A name-value pair consists of a field name, followed by the colon, followed by the corresponding value. Each object can contain other objects and arrays. (The same is true for arrays.) JSON supports also elementary data types: string, number and Boolean.

Therefore, the JSON document in Figure 1 shows a JSON string called **info**, which describes a single person, Fred, his affiliation, specified as a name-value pair, and his friends, specified as an array, which contains, in this example, two objects: **name** and **rank**.

The current SQL/JSON standard [1] proposes the storage of JSON data in relational tables using the existing standard data types. In other words, this approach advocates to store JSON data into character string columns that are defined within SQL tables. That permits JSON documents to be used in SQL queries in the same way as the data stored in other columns of the same tables. By choosing to use columns declared using the standard data types, the standardization committee avoids the overhead needed to create a new SQL data type without losing any significant advantages of standard data types.

The specified approach is “light-weight” one, meaning that an exact copy of JSON documents is stored into relational tables. (As the SQL standardization committee members quote, the primary reason for taking this approach is to improve the chances of its quick adoption into the SQL standard, as well as the rapid implementation by vendors of relational database systems.)

The alternative way [2] would be to store JSON documents in a form that is closely aligned with JSON semantics. This model should include arbitrary levels of nesting and complexity and should be automatically mapped by the system into the underlying storage form.

2. RELATIONAL STORAGE OF JSON

There are two general techniques to store JSON documents in relational form:

- Storage as a “raw” document
- Native storage

Storing data in its “raw” form means that an exact copy of the data is stored into a relational table. In other words, relational database systems use character string data types to store the data. This storage form allows insertion of data in an easy way. The retrieval of such data is very efficient if the entire document is retrieved. To retrieve parts of the documents, special types of indices are helpful.

Native storage means that the corresponding data are stored in their parsed form. In other words, the document is stored in an internal (usually binary) 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 is difficult, because the created content may not be an exact copy of the document.

Note that an additional technique called shredding is also used. In this case, a document is decomposed into separate columns of one or more relational tables. For the decomposition process, the schema of that document is used. This technique cannot be applied to JSON data in general, because the corresponding schema usually does not exist. Therefore, only schema-oblivious approaches for mapping into relational tables can be generally applied to JSON.

2.1 Native Storage vs "Raw" Storage

The native storage of JSON data has several advantages in relation to storage in the "raw" form. These are:

- Richer type system
- Efficient updates

In case of native storage form, its binary format allows an extension of the existing SQL type system, which contains standard data types. On the other hand, the type system for JSON comprises only three simple data types: string, number, and Boolean.

In case of "raw" storage form, update operations are usually performed so, that the entire document is rewritten, even when only a small part of the document is modified. On the other hand, the use of native storage allows so called partial updates, meaning that only a portion of a document, which is referenced by the modification operation is updated.

3. MYSQL BINARY STORAGE FORMAT

In this paper we use Version 8 of the MySQL database system, while it allows us to measure performance of UPDATE operations on JSON data stored in the "raw" document form as well as in the native form. MySQL supports "raw" form, when JSON data are stored in the TEXT data type, and native one, when the data is stored in the JSON type.
Note that another system, PostgreSQL, supports "raw" and native form in the same way as MySQL, but the corresponding UPDATE operations are not supported for the "raw" storage form of this system.

3.1 Structure of the MySQL Binary Format

The storage of documents declared using the JSON data type in My SQL Version 8 is achieved using the proprietary binary format. The goal of the binary format is to allow storing, querying and modifying JSON data efficiently. First, complex and simple JSON data types are handled in the different ways [3]. In case of objects, the content is stored in three sections: table of pointers, keys and values.

Table of pointers contains all the keys and values, in the order in which they are stored. Each pointer comprises information about where the data associated with the key or value is located. Having a table of pointers to the keys and values at the beginning of the binary representation makes it possible to look up a member in the middle of the document directly without scanning past all the members that precede it. In key-value pairs, the keys are ordered by their length, and keys with the same length are sorted in ascending order. The process of sorting using the length allows the process of binary search, so that the content of the keys must not be referenced. The values are sorted in the same order as their corresponding keys.

If the document is an array, it has two sections: the dictionary and the values. The dictionary contains the set of names of array elements. Arrays are encoded by a single size prefixed offset array containing the offsets of the elements in the array. If the document is a scalar, it has a single section which contains the scalar value.

3.2 Partial Update

Since Version 8, MySQL binary storage format supports partial updates, meaning that modifications of JSON documents can be made without replacing the entire document, as it is necessary in case when JSON data is stored in "raw" form. Therefore, partial updates are applied only for the columns declared using the JSON data type. Also, this feature works only with the following three SQL/JSON functions: JSON_SET(), JSON_REPLACE(), and JSON_REMOVE().

Another feature concerning partial update concerns transaction log. When partial update is activated, it causes the update operation to write only the modified part(s) of the JSON document to the after image in the transaction log, instead of storing the whole document in the log. With this optimization, transaction log size is proportional to the size of the modified part rather than the full document size.

4. JSON UPDATE LANGUAGE

In this section we will discuss which primitives should be implemented for update of JSON documents as well as the JSON update operations implemented in MySQL.

4.1 JSON Update Primitives

We assume the presence of a SQL/JSON path expression to be evaluated and to return tuples of references to the selected objects. This will be the target of the sequence of operations discussed below. Note that the term "path" in this and the following subsections specifies the SQL/JSON path expression as it is defined in the SQL/JSON standard [1].

An update operation is a sequence of primitive operations of the following types [4]:
• **Insert (content**): inserts a new content.
The following case exists:
- Insert an object
• **InsertBefore (path, conten**t): inserts new content in front of the referenced element.
The following cases exist:
- Inserts an object before another one
Note that the operation "inserting an object before another one" is optional, because there is no ordering of objects, generally.
- Insert an array member before another array member (new array member must be of the same data type as the existing members of the array)
• **InsertAfter (path, content)**
Inserts new content directly after of the referenced element.
The following cases exist:
- Insert an array member after another array member (new array member must be of the same data type as the existing members of the array)
- Insert an object after another one
Note that the operation "inserting an object after another one" is optional, because there is no ordering of objects, generally.
• **Delete (content)**
Deletes the content under the given path expression.
The following cases exist:
- Delete an object with the given name
- Delete the n-th element of the array with the given name
• **Replace (content)**
Replace the content under the given path expression.
The following cases exist:
- Replace the value of the object with the given name
- Replace the n-th element of the array with the given name
• **Rename (property, name)** Renames the specified object or array.

4.2 MySQL: JSON Update Operations

MySQL Version 8 supports three SQL/JSON functions to update JSON documents:

- JSON_SET()
- JSON_REPLACE()
- JSON_REMOVE().

These functions are defined inside the SQL UPDATE statement. Note that MySQL supports several other SQL/JSON update functions, but these functions do not support the partial update of JSON data [5] and therefore cannot be used to measure performance of JSON updates in MySQL.

The JSON_SET() function is used to modify JSON values as well as to insert the new ones. In other words, this function checks first whether the path declared in the statement exist or not. If it exists, the function replaces values for the path. If not, it adds the specified value for that path.

Example 1 shows how the JSON_SET() function can be used to update an existing value, while Example 2 shows the inserting functionality of the function. (All examples in this section are related to the JSON document given in Figure 1.The document is stored into the table called **my_json** using the CREATE TABLE statement.)

Example 1

```
-- The value of the object called where is changed to "Google"
UPDATE my_json
  SET c1 = JSON_SET(c1, '$.info.where', 'Google')
  where JSON_EXTRACT (c1, '$.info.where') = 'Microsoft';
```

Example 2

```
-- The new object (name-value pair) will be inserted
UPDATE my_json
  SET c1 = JSON_SET (c1, '$.info.age', 47 )
  where JSON_extract ( c1, "$.info.who") = "Fred";
```

The JSON_REPLACE() function modifies existing values. The UPDATE statement in Example 3 modifies the name of the person called "Fred".

Example 3

```
UPDATE my_json
  SET c1 = JSON_REPLACE (c1, '$.info.who', "Freddie" )
  where JSON_extract ( c1, "$.info.where") = "Google";
```

The JSON_REMOVE() function removes a value specified in the path expression of the UPDATE statement and returns the original JSON document without the values selected by the path expression in case that they exist within that document. In Example 4, the first element of the array called friends will be the deleted.

Example 4

```
UPDATE my_json SET c1 =
 JSON_REMOVE (c1, '$.info.friends[0]' )
    WHERE JSON_EXTRACT (c1, "$.info.who") = "Fred";
```

5. PERFORMANCE ISSUES

Before we start to discuss performance of different update primitives, we will give some introduce preliminary issues.

5.1 Preliminaries

We compare the performance of JSON updates on documents stored using "raw" document form and implemented with the TEXT data type with the performance of JSON updates on documents stored using native (binary) form and implemented with the JSON data type. Also, we show that all JSON document updates, except one, benefit when using the native storage form.

All experiments ran on a Windows 10 computer with Intel Core i5-6200 and CPU with 2.4 GHz speed, as well as with 32 GB RAM. Additionally, we use MySQL Version V8.0, which supports "raw" form of JSON data as well as the native storage form.

To measure performance of medium-size and very large JSON document, we used two different data sets: the first one, called ZIP data, contains the US zip codes [6]. The zip code data set contains ca. 30,000 US zip codes. The structure of the JSON sample data includes the name of the city ("city"), its population ("pop") and location ("loc"), as well as the acronym of the state to which the city belongs ("state"). Each JSON document is 85 bytes long, in average.

The second data set, which is used to measure performance, contains eight documents. Each of them is approximately 1 MB large and the structure of them is similar with the JSON document shown in Figure 1, plus an additional column called **data**. The **data** column is a string of 1 MB length (see cases I3 and I4 in Figure 2a.)

According to the discussion above, we have the following four tables, which we use for our tests:

- table_text_30K
- table_json_30K
- info_large_text
- info_large_json.

The first two tables are created using the SQL statements shown in cases CT1 and CT2 of Figure 2a, respectively. The other two tables are created using CT3 and CT4 of the same figure, respectively. Analogously, cases I1 and I2 show how a JSON document from the ZIP data set is inserted into the first two tables, respectively. Also, I3 and I4 shows how a row of the large data set is inserted into the second two tables, respectively.

5.2 Performance Measures

We measure performance of the following update primitives:

- Initial load
- Replace a value of an object
- Delete an object
- Insert an object
- Modify an array's element
- Delete an array's element
- Insert an array's element

Note that the y-axis of all figures below shows the execution time of the corresponding query, shown in seconds.

```
-- CT1 Create table with ZIP data (JSON data type)
CREATE TABLE table_JSON_30K (c1 JSON);
-- CT2 Create table with ZIP data (TEXT data type)
CREATE TABLE table_TEXT_30K (c2 TEXT);
-- CT3 Create large table (JSON data type)
CREATE TABLE info_large_json
(id INT PRIMARY KEY AUTO_INCREMENT,
        json_col JSON);
-- CT4 Create large table (TEXT data type)
CREATE TABLE info_large_text
(id INT PRIMARY KEY AUTO_INCREMENT,
        text_col LONGTEXT);
-- I1
-- Insert a row into ZIP table (JSON data type)
INSERT INTO table_json_30k VALUES ('{ "city" : "AGAWAM", "loc"
:[-72.622739,42.070206 ], "pop" 15338, "state":"MA","_id":"01001" }')
-- I2
-- Insert a row into ZIP table (TEXT data type)
INSERT INTO table_text_30k VALUES ('{ "city" : "AGAWAM", "loc" :
[ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA", "_id" :
"01001" }')
-- I3
-- Insert a row into large table (JSON data type)
INSERT INTO info_large_json(json_col) VALUES
(JSON_OBJECT("who","Fred","where","Microsoft",
"friends",JSON_ARRAY( JSON_OBJECT("name","Lili","rank",1),
 JSON_OBJECT("name","Hank","rank",8)),
 "data",  REPEAT("x", 100 * 100 * 100)));
-- I4
-- Insert a row into large table (TEXT data type)
INSERT      INTO      info_large_text(text_col)      VALUES
(JSON_OBJECT("who", "Fred", "where", "Microsoft",
 "friends", JSON_ARRAY( JSON_OBJECT("name","Lili","rank",1),
JSON_OBJECT("name","Hank","rank",8)),
"data",  REPEAT("x", 100 * 100 * 100)));

--UO1
-- Replace a value of an object (zip table, JSON type)
UPDATE table_json_30K
 SET  c1  =  JSON_SET  (c1,  '$.city',  "NEW_BARRE"  )
   where JSON_EXTRACT ( c1, "$.city") = "BARRE";
```

```
--UO2
-- Replace an object's value (zip-table, TEXT  type)
UPDATE table_text_30K
SET c2 = JSON_SET (c2, '$.city',  "NEW_BARRE" )
where JSON_EXTRACT ( c2, "$.city") = "BARRE";
--UO3
-- Replace an object's value (large able,TEXT type)
  UPDATE info_large_text
SET text_col = JSON_SET (text_col, '$.where', "Facebook" )
WHERE JSON_EXTRACT ( text_col, "$.where") = "Microsoft";
 --UO4
-- Replace an object's value (large table,JSON  type)
  UPDATE info_large_json
    SET json_col = JSON_SET (json_col, '$.where', "Facebook" )
   WHERE JSON_EXTRACT ( json_col, "$.where") = "Microsoft";
 --DO1
-- Delete an object (ZIP table, TEXT data type)
UPDATE table_text_30k
    SET c2 = JSON_REMOVE (c2, '$.state')
   WHERE JSON_EXTRACT (c2, "$.state") = "MA";
--DO2
-- Delete an object (ZIP table, JSON data type)
UPDATE table_json_30k
    SET c1 = JSON_REMOVE (c1, '$.state')
   WHERE JSON_EXTRACT ( c1, "$.state") = "MA";
--DO3
-- Delete an object (large table, JSON data type)
UPDATE info_large_json
    SET json_col = JSON_REMOVE
     (json_col, '$.where')
  WHERE JSON_EXTRACT ( json_col, "$.where") = "Facebook";
--DO4
-- Delete an object (large table, TEXT data type)
UPDATE info_large_text
   SET text_col = JSON_REMOVE (text_col, '$.where')
   WHERE JSON_EXTRACT
     ( text_col, "$.where") = "Facebook";
```

Figure 2a: Examples of SQL/JSON Update Statements

```
--IA1
-- Insert an array's element (ZIP table, JSON data type)
UPDATE table_json_30k
    SET c1 = JSON_SET (c1, '$.loc[1]', -70.00000);
--IA2
-- Insert an array's element (ZIP table, TEXT data type)
    UPDATE table_text_30k
    SET c2 = JSON_SET (c2, '$.loc[1]', -70.00000);
--IA3
-- Insert an array's element (large table, JSON type)
UPDATE info_large_json SET json_col =
 JSON_SET (json_col,'$.friends[1]',
 JSON_OBJECT("name","Peter", "rank",6));
--IA4
-- Insert an array's element (large table, TEXT type)
UPDATE info_large_text
  SET text_col = JSON_SET (text_col,'$.friends[1]',
JSON_OBJECT("name","Peter", "rank",6));
-- UA1
Update an array's element (zip table ,JSON type)
UPDATE table_json_30k
    SET c1 = JSON_SET (c1, '$.loc[0]', 0.00)
where JSON_EXTRACT(c1,"$.city")= "SPRINGFIELD";
-- UA2
-- Update an array's element  (zip table, TEXT type)
UPDATE table_text_30k
    SET c2 = JSON_SET (c2, '$.loc[0]', 0.00)
  Where JSON_EXTRACT ( c2, "$.city") =
 "SPRINGFIELD";
-- UA3
-- Update an array's element(large table,JSON type)
UPDATE info_large_json
    SET json_col =
JSON_SET (json_col, '$.friends[1].rank', 7);
-- UA4
-- Update an array's element  (large table, TEXT type)
UPDATE info_large_text SET
 text_col=JSON_SET(text_col,'$.friends[1].rank', 7);
```

```
-- DA1
-- Delete an element of an array(zip table,JSON type)
UPDATE table_json_30k
    SET c1 = JSON_REMOVE (c1, '$.loc[1]');
-- DA2
-- Delete an element of an array (zip table,TEXT type)
UPDATE table_text_30k
    SET c2 = JSON_REMOVE (c2, '$.loc[1]');
-- DA3
-- Delete an array's element(large table, JSON type)
UPDATE info_large_json
    SET json_col = JSON_REMOVE (json_col, '$.friends[1]');
-- DA4
-- Delete an array's element (large table,TEXT type)
 UPDATE info_large_text
SET text_col=
JSON_REMOVE(text_col, '$.friends[1]');
-- IO1
-- Insert an Object (zip table, TEXT type)
UPDATE table_text_30k
   SET c2 = JSON_SET (c2, '$.info.age', null)
 where JSON_EXTRACT ( c1, "$.city") = "CUSHMAN";
-- IO2
-- Insert an Object (zip table, JSON type)
UPDATE table_json_30k
   SET c1 = JSON_SET (c1, '$.info.age', null)
Where JSON_EXTRACT(c1,"$.city") = "CUSHMAN";
-- IO3
-- Insert an Object (large table, JSON type)
UPDATE info_large_json
   SET json_col = JSON_set (json_col, '$.data112',
        REPEAT("x", 100 * 100 * 100 ))
where JSON_EXTRACT(json_col, "$.who") = "Fred";
  -- IO4
-- Insert an Object (large table, TEXT type)
  UPDATE info_large_text
   SET text_col = JSON_SET (text_col, '$.data112',
 REPEAT("x", 100 * 100 * 100 ))
where JSON_EXTRACT( text_col, "$.who") = "Fred";
```

Figure 2b: Examples of SQL/JSON Updates Cont.)

5.2.1 Initial Load

During the initial load, the four tables mentioned above are loaded with JSON documents. The **table_text_30K** and **table_json_30K** tables are loaded with the JSON data concerning ZIP codes. The **info_large_text** and **info_large_json** tables contain the same content: the number of inserted rows is eight and each row comprises a JSON document of ca 1 MB length.

Figure 3 shows the execution time needed for initial load of tables **table_json_30K**, **table_text_30**, **info_large_json** and **info_large_text**, respectively. The chart shows that loading the ZIP data into the column of the TEXT type is a little bit faster than the loading of the same data in the column of the JSON data type. Also, loading the data into very large JSON documents concerning the column of the TEXT data type is approximately 7% faster than the loading of the same data into the column of the JSON data type. The difference in loading time is due to the different format of both data types. In case of the TEXT data type, the data is loaded "raw", meaning that, during the loading process, there are no transformations at all, while the JSON data type involves certain transformations into the corresponding binary storage form.

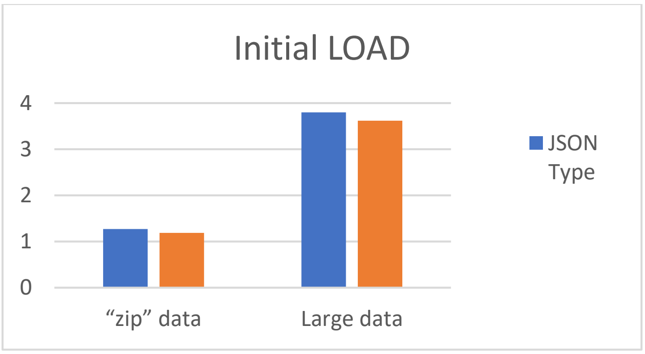

Figure 3: Initial Load

5.2.2 *Replace an Object's Value*

The second modification on objects is updating an object's value. For this measurement we use cases UO1 and UO2 for the ZIP data as well as UO3 and UO4 for the large JSON documents, respectively (see Figure 2a).

Figure 4 shows an improvement of 3.9x in case of the replacement of a value for the ZIP data and an improvement of 9x, in case of the same operation for large JSON documents. The reason for these improvements is that in case of the JSON data type, partial updates are executed, while in case of the TEXT data type, after the update operation is performed, the entire JSON document will be replaced. In other words, in case of the TEXT data type the system generates a temporary new document to fully replace the previous stored document.

The significant performance improvements in case of large JSON documents are due to the additional optimization of the transaction log entries for the JSON data type in relation to modifications. In case of the TEXT data type, the database system generates a log entry proportional to the size of the JSON document. For the JSON data type, the improvement is that transaction log size is usually proportional to the actual delta update size rather than to the full document size. Therefore, the larger the document, the better the response time.

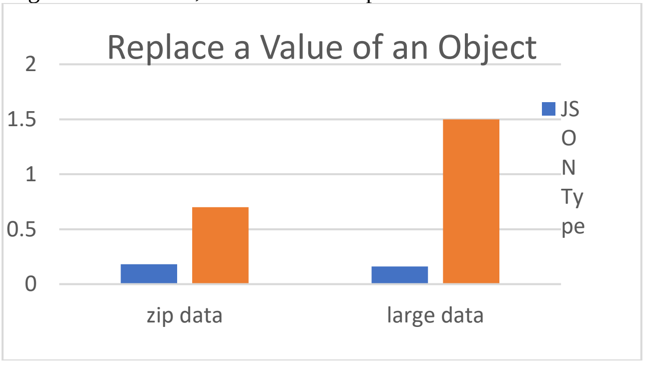

Figure 4: Replace an Object's Value

5.2.3 *Delete an Object*

The next update primitive on objects is deleting one or more objects. For this measurement we use cases DO1 and DO2 for the ZIP data as well as DO3 and DO4 for the large JSON documents, respectively (see Figure 2a).

Figure 5 shows an improvement of 2x, when an object for the ZIP data is deleted, and an improvement of 11x, in case of the same operation for large JSON documents. The reason for these improvements is the same as already described in the previous subsection.

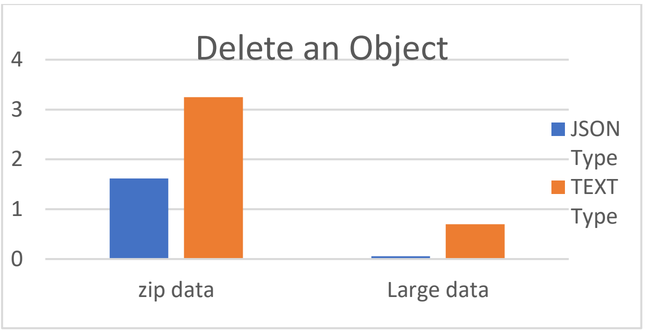

Figure 5: Delete an Object

5.2.4 *Insert an Object*

The last modification on objects is inserting one or more objects. For this measurement we use cases IO1 and IO2 for the ZIP data as well as IO3 and IO4 for the large JSON documents, respectively (see Figure 2b).

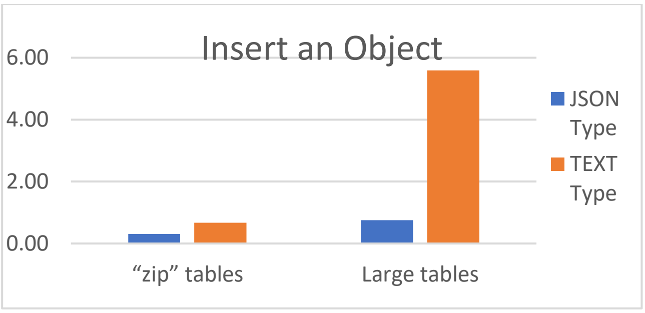

Figure 6: Insert an Object

Figure 6 shows an improvement of 2x in case, when an object for the ZIP data is inserted, and an improvement of 7.5x, in case of the same operation for large JSON documents. The reason for these improvements is the same as already described in the previous subsection(s).

5.2.5 *Modify an Array's Element*

In this subsection we show the performance measurements in relation to replacement of an array's element. For these measurements we use cases UA1 and UA2 for the ZIP data, as well as UA3 and UA4 for the large JSON documents, respectively (see Figure 2b).

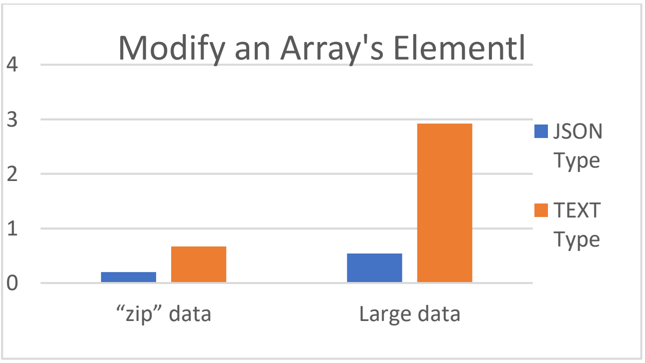

Figure 7: Modify an Array's Element

Figure 7 shows an improvement of approximately 2.5x in case when an element of an array for the ZIP data is modified, and an improvement of 4.4x, in case of the same operation for large JSON documents. The improvements in case of the ZIP data are similar with the improvements in case of modification of an object, described in the previous subsections.

On the other hand, the improvements in relation to the large JSON documents are smaller compared with the improvements in relation to objects' modifications, because they modify more rows that require additional operations than in case of modification on JSON objects.

5.2.6 Delete an Array's Element
Another update operation on arrays is deleting an element of an array. For these measurements we use cases DA1 and DA2 for the ZIP data, as well as DA3 and DA4 for the large JSON documents, respectively (see Figure 2b).

Figure 8 shows an improvement of approximately 1.6x in case, when an element of an array for the ZIP data is deleted, and an improvement of 4.5x, in case of the same operation for large JSON documents. The improvements in case of the ZIP data are similar with the improvements in case of modification of an array's element, described in the previous subsection.

On the other hand, performance improvements concerning large documents are smaller for the native storage form in relation to the "raw" storage form due to the same reason as in case of modifying an array's element.

5.2.7 Insert an Array's Element
The last update primitive on arrays, which is discussed in this paper is insertion of an array's element. For these measurements we use cases IA1 and IA2 for the ZIP data, as well as IA3 and IA4 for the large JSON documents, respectively (see Figure 2b).

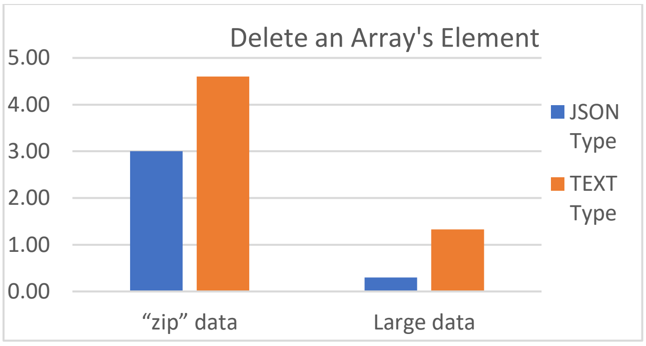

Figure 8: Delete an Array's Element

Figure 9 shows an improvement of approximately 1.3x in case, when a new element of an array for the ZIP data is inserted, and an improvement of 4.1x, in case of the same operation for large JSON documents. The improvements for this operation in case of native (binary) storage form are similar as the corresponding improvements described in both subsections above.

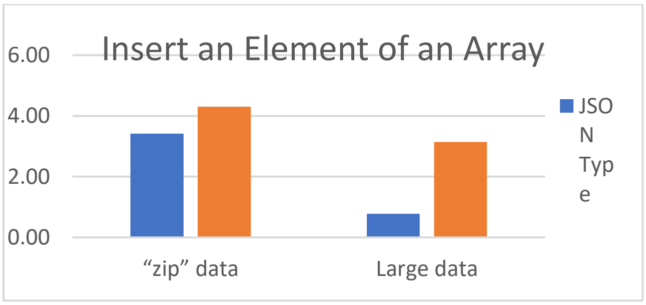

Figure 9: Insert an Array's Element

6. RELATED WORK
Besides My SQL, several other RDBMSs support the JSON data type. PostgreSQL[7] supports a proprietary native storage format to implement the JSON type. Oracle [8] uses its own in-memory binary storage to store JSON data, which is declared using the JSON data type [9]. Teradata supports the JSON type with different forms of storage formats [10]. To these formats belong, among others, BSON [11] and UBJSON [12]. IBM Db2[13] uses also BSON for the native storage of JSON documents. Several NoSQL database systems, such as MongoDB [14] and Couchbase DB [15] use BSON as a binary storage format to store JSON data.

The design of binary storage of JSON data is similar by Oracle, PostgreSQL and Sinew[16]. They all support the navigation based on field key names, which are indexed in the binary format to speed up query. Therefore, all of these binary formats achieve faster query performance than "raw" document form. Concerning partial update operations on JSON data, only MySQL V8.0 [17] and Oracle V20.c [18] support this feature. That way, all modification operations on JSON data do not replace the entire JSON document(s), but only the part referenced by the corresponding UPDATE statement.

7. CONCLUSIONS
This paper discusses the performance of modifications on JSON documents, stored in "raw" form as well as in native form. To measure performance, we use MySQL Version 8.0. Our measures show that for all UPDATE operations, except for the initial loading process, the modification of JSON data stored in the binary form is significantly faster than the modification of the same data stored in the "raw" document form. Additionally, the following rule holds: the bigger the JSON document, the more significant the performance gains.

8. REFERENCES
[1] SQL/JSON 2016 Standard: ISO/IEC TR 19075-6:2017, Information technology Part 6: SQL support for JSON, http://standards.iso.org/ittf/PubliclyAvailableStandards.

[2] D. Petković, SQL/JSON Standard: Properties and Deficiencies, Datenbank Spektrum, Vol.17, No.3, 2017, DOI: 10.1007/s13222-017-0267-4.

[3] JSON datatype and binary storage format, https://dev.mysql.com/worklog/task/?id=8132

[4] D.Petković – Implementation of JSON Update Frameworks in RDBMSs, International Journal of Computer Applications 177(37):35-39, DOI: 10.5120/ijca2020919881.

[5] Partial Update in MySQL,https://dev.mysql.com/doc/refman/8.0/en/json.html#json-partial-updates

[6] US ZIP Code Data Catalog, https://catalog.data_gov/dataset1.

[7] PostgreSQL: The JSON data type, https://www.postgresql.org/ docs/9.4/datatype-json.html

[8] Z.H. Liu, et al. JSON data management: supporting schemaless development in RDBMS. SIGMOD Conference 2014, 1247-1258 2014, DOI: 10.1145/2588555.2595628

[9] Z.H. Liu, et al., Native JSON Datatype Support, Proc. Of the VLDB Endowment, Vol.13, No. 12, 2020, https://doi.org/10.14778/3415478.3415534

[10] Teradata JSON Datatype: https://docs.teradata.com/ reader/ C8cVEJ54PO4~YXWXeXGvsA/4IAzgRsj_8aRj5pCQoEqzA

[11] BSON, http://bsonspec.org/

[12] UBJSON: http://ubjson.org/

[13] DB2 JSON support, https://www.ibm.com/support/ knowledgecenter/en/SSEPEK11.0.0/json/src/tpc/db2z_json functions.html

[14] MongoDB storage formats, https://www.mongodb.com/json-and-bson

[15] Couchbase JSON Support, https://developer.couchbase.com/ docum/server/3.x/developer/dev-guide-3.0/using-json-docs.html

[16] D. Tahara, et al: Sinew: a SQL system for multi-structured data. *SIGMOD Conference* 2014: 815-826, DOI: 10.1145/2588555.2612183

[17] MySQL, Partial Updates of JSON Values, https://dev.mysql. com/ doc/refman/8.0/en/json.html#json-partial-updates.

[18] Oracle, Piece-wise update of JSON, https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/overview-of-inserting-updating-loading-JSON-data.html#GUID-94E37619-C242-44F0-B1C3-9A63859AD0C5.