On UNIQUE KEYS in SQL/JSON Implementations - Experimenting with JSON support in DBMS products

Martti Laiho
DBTechNet
www.dbtechnet.org
formerly at Haaga Helia
email: martti.laiho@gmail.com

Fritz Laux (retired)
Fakultät Informatik
Reutlingen University
D-72762 Reutlingen, Germany
email: fritz.laux@reutlingen-university.de

Abstract—**The popularity of the JavaScript Object Notation (JSON) data format has led to its early support by major database vendors. Relational database systems added a JSON data type and implemented functions for manipulation and querying JSON data. The ISO/IEC Standardization lagged behind this development and only recently specified the update and maintenance of JSON data. Now we have a Babel of confusion with different syntax for most major relational systems. This complicates and hinders reliable JSON data exchange, SQL portability and knowledge transfer between developers. In the context of relational databases with strong consistency support, JSON requires reliable and deterministic data retrieval behavior. This paper we make the reality check and investigate how popular relational database products (Db2 for Linux/Unix/Windows (LUW), Oracle 23ai, SQL Server, PostgreSQL, and MariaDB) handle JSON data manipulation and how they conform to the ISO/IEC 19075-6:2021 and ISO/IEC 9075-2:2023 standards.**

Keywords-ISO SQL/JSON; UNIQUE KEYS; data retrieval; data update;

I. Introduction

JSON [1] has rapidly become a standard for data representation and exchange due to its simplicity. Initially designed for data interchange in NoSQL systems like MongoDB, its integration into Relational Database Systems (RDBMS) has grown in popularity. The adoption of JSON allows RDBMSs to handle semi-structured data, expanding their versatility to accommodate more dynamic data models. A JSON document is hierarchically structured consisting of primitive (scalar) value types (number, boolean, string), JSON literals (true, false, null) and structured types (array, JSON object) which may be nested to form a data hierarchy.

The following example is a valid JSON document:

```
{
  "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url":"http://www.example.com/img4819",
            "size":"125x100"
         },
        "Animated" : false,
        "IDs": [116, 943, 234, 38793]
    }
}
```

It consists of one JSON object "Image" (case sensitive!) and 6 members. Each object is enclosed in curly brackets and consists of a "name" and a "value" separated by a colon. In the IETF RFC 8259 [1] specification the "name" acts as a key for the object. This is why the ISO/IEC standard calls it "key" and we will adopt this convention because our paper deals with databases. The value of the key "Thumbnail" is again a JSON object. The value of "IDs" is an array with 4 numbers.

According to the JSON specification RFC 8259 "The names (keys) within an object SHOULD be unique" and continues ".. When the names (keys) within an object are not unique, the behaviour of software that receives such an object is unpredictable." The RFC does not deny the possibility of non-unique keys of object keys, and we need to remember that JSON structures in general are applied for data exchange between systems. However, the ISO/IEC 19075 SQL/JSON specification [8] for the RDBMS context needs to be stricter than the generic JSON specification of RFC 8259, as we will see later.

However, the possibility of non-unique keys has consequences apart from software being unable to process the JSON content properly: (1) Because the member elements of a JSON object can be found only by key names, it is not possible to identify a single member if two or more members have the same key on the same structural level, for example: `{ "key1":"val", "key1":"val" }`. Because the members of a JSON object are unordered, we cannot refer to the "first" or "second" member. However, it is possible to have the same key name on different levels of the JSON object like `{ "key1": { "key1":"val1" } }`. In this case, the value of `"key1"` on the second level is accessed as `"key1"."key1"`. (2) The processing software has to deal with two possibilities, either the access addresses one single member or it receives a collection of elements where the ordering has no meaning. In the latter case, the collection must be processed as a whole in order to yield correct results.

SQL/JSON extends the relational model into a new hybrid model, whose implementation should preserve RDBMS functionality including transaction, strict consistency, and the prevention of "unpredictable behaviour" when processing data.

A. Contribution

We present systematic tests for Db2 for LUW, Oracle 23ai, SQL Server, PostgreSQL, and MariaDB and look how they behave when (1) attempting to insert duplicate keys, and (2) retrieving and updating objects with duplicate keys. To the best of our knowledge, no scientific publications address how SQL/JSON databases behave when there are duplicate JSON keys in a document. The paper provides advice how to prevent unintended behaviour or even inconsistent JSON data processing for the tested database systems.

B. Structure of the Paper

The paper is structured as follows: Section II discusses literature on the JSON update framework for RDBMS. In Section III we present and discuss the behaviour of five databases with reference to retrieval and manipulation of duplicate JSON objects. Section IV discusses the results and Section V concludes with a critique of the situation and recommends a work-around for developers to avoid these problems.

II. Related Work

Most papers on handling SQL/JSON are promotional work about implementations of JSON data type (e.g., JSON Binary (JSONB) for PostgreSQL [2], Optimized JSON (OSON) for Oracle [3]) and model control features (e.g., key uniqueness control for Db2 [4]) or JSON functions and operators.

Petkovic [5] presents a comprehensive description based on the proposed ISO SQL/JSON standard. At that time (2017) only a "light weight" functionality was defined, lacking native JSON data type and update functions for JSON data. In 2020 the big relational database vendors had already implemented their own JSON data types (like BSON, OSON, JSONB) before the ISO standard was ready in 2021. The databases also provided a set of manipulation functions for their JSON data types. The internal representation of the JSON data type is not relevant for our experiments as it affects only the performance, storage space needed, and some JSON function behavior. The JSON standard and its external representation is always text based. Its functionality was investigated by Petkovic [6]. He tested INSERT, DELETE, and REPLACE operations. The INSERT distinguished the cases of inserting an object (key:value pair) and inserting an array element before and after a certain element. The DELETE distinguished two cases: delete an object and delete the n^{th} element of an array. The REPLACE made the same distinction as the DELETE. His observation was that SQL Server, PostgreSQL, and MySQL fully supported the above modification operations but with a very different syntax and different functions. Petkovic's paper does not consider the case of duplicate JSON members.

III. JSON data manipulation tests

In this section, we look at Db2 for LUW[1], Oracle 23ai[2], SQL Server[3], PostgreSQL[4], and MariaDB[5]. Some of the tested products allow duplicate JSON object members and others require unique keys. We place special focus on both situations when testing the SELECT, UPDATE, JSON_VALUE, and JSON_MODIFY functionality.

[1]Version 12.1.1 Community Edition
[2]Version 23ai Free
[3]2017 Express Edition
[4]Version 17
[5]Version 11.8.2

A. The Standard for SQL/JSON

JSON was first proposed as a data exchange format in 2014, and RFC8259 recommended that the JSON object members SHOULD have unique keys as mentioned in the Introduction. ISO/IEC 21778:2017 [7] and the standard of 2021, ISO/IEC19075-6:2021 [8], are even more relaxed and do not mention unique keys: "The JSON syntax does not impose any restrictions on the strings used as names, does not require that name strings be unique, and does not assign any significance to the ordering of name/value pairs" [7]. This is unfortunate and allows the developers to implement JSON retrieval functions in an incompatible way. The latest ISO standard of 2023 [9][10] introduces the options WITH UNIQUE KEYS and WITHOUT UNIQUE KEYS but does not define a default. On the implementation side, as we will see, database developers mostly decide for the retrieval of JSON data to ignore duplicate keys and return either the first or last matching object without any notice. This makes data exchange problematic because duplicates may be lost. Duplicates can be avoided if the insert statement uses the clause "WITH UNIQUE KEYS".

The following tests are based on the study "JSON Data Maintenance" [11] conducted by the main author of this paper. An extensive study considering the general support of JSON data in relational databases is found in the document "JSON on RDBMS Databases" [12] by the same author.

B. Test set-up

We created the following table for our test set-up, using default values for each RDBMS product tested:

```
CREATE TABLE T1 (
K  INTEGER NOT NULL PRIMARY KEY,
J  <json | clob | blob | jsonb> );
-- depending on the RDBMS
```

We then attempted to insert JSON objects with duplicate key names (here: `"duplica"`). The exact syntax for the JSON column depends on the SQL dialect of the database:

```
-- generic insert statement
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
      "duplica":"Third","duplica":"Last"}');
-- use syntax for specific product
```

C. Db2 for LUW

In the following INSERT statement the function JSON_TO_BSON converts the external, text based JSON document into an IBM specific binary format, called BSON.

```
db2 => INSERT INTO T1 (K, J) VALUES
db2 (cont.) => (3, JSON_TO_BSON('{ "duplica":
 "First", "duplica":"Second", "duplica":
 "Third", "duplica":"Last"}'));
DB21034E  The command was processed as an SQL
 statement because it was not a valid Command
 Line Processor command.  During SQL
 processing it returned:
SQL16406N  JSON data has non-unique keys.
db2 =>
```

Db2 apparently uses WITH UNIQUE KEYS by default. However, the DB2 version for the *IBM I* operating system has now added the clauses WITHOUT UNIQUE KEYS and WITH UNIQUE KEYS to the JSON publishing functions and changed its default to WITHOUT UNIQUE KEYS. This is worrying because the SQL/JSON specification should not be implemented without criticism when there is a risk of breaking the content integrity of Db2. Having WITH UNIQUE KEYS as default would definitely be the better choice.

D. Oracle 23ai

Oracle recommends using the CHECK constraint "IS JSON WITH UNIQUE KEYS" for text-based JSON columns to avoid inconsistent contents. This constraint is automatically activated for columns based on Oracle's native JSON data type, called OSON, which we are using. So, the effect is the same as using the WITH UNIQUE KEYS clause.

```
SQL> -- Duplicate object members test:
SQL> INSERT INTO T1 (K, J) VALUES
  2* (3, '{ "duplica":"First", "duplica":"Second",
   "duplica":"Third","duplica":"Last"}');
Error starting at line : 1 in command -
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
 "duplica":"Third","duplica":"Last"}')
Error at Command Line : 1 Column : 13
Error report -
SQL Error: ORA-40473: duplicate key names 'duplica'
in JSON object
JZN-00007: Object member key 'duplica' is not unique
Help: https://docs.oracle.com/error-help/db/ora-
40473/40473. 00000 -  "duplicate key names '%s' in
JSON object"
*Cause:    The provided JavaScript Object Notation
(JSON) data had duplicate key names in one object.
```

Oracle reports a verbose error message when trying to insert duplicate JSON members.

E. SQL Server

The SQL Server XE does not recognize the constraint IS JSON WITH UNIQUE KEYS. Let's test what happens when we enter a JSON document having duplicate member keys:

```
-- Duplicate object members test:
BEGIN TRANSACTION;
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
 "duplica":"Third","duplica":"Last"}');
(1 row affected)
 -- The following LEFT function is only used
 -- to limit the column width
SELECT LEFT(K, 4) AS K, LEFT(JSON_VALUE(J,
'$.duplica'),  10) AS Duplica
FROM T1 WHERE K=3;
K    Duplica
---- ----------
3    First
(1 row affected)

-- But let's see them all
SELECT J FROM T1 WHERE K=3;
J
----------------------------------------
{ "duplica":"First", "duplica":"Second",
  "duplica":"Third","duplica":"Last"}
(1 row affected)

-- Let's remove the 'first' member. There is
-- no real REMOVE, the member is set to NULL
UPDATE T1 SET J = JSON_MODIFY(J, '$.duplica', NULL)
WHERE K=3;
(1 row affected)

SELECT LEFT(K, 4) AS K, JSON_VALUE(J, '$.duplica')
AS Duplica FROM T1 WHERE K=3;
K    Duplica
---- ----------
3    Second
(1 row affected)

-- OK, let's see them all
SELECT J FROM T1 WHERE K=3;
J
-----------------------------------------
{ "duplica":"Second", "duplica":"Third",
  "duplica":"Last"}
(1 row affected)
```

Duplicate keys cannot be prevented in SQL Server, as it would appear that the "WITH UNIQUE KEYS" clause has not been implemented yet.

F. PostgreSQL

Using the following script, we experiment how duplicate object members behave in a PostgreSQL transaction.

```
testdb=> BEGIN;
BEGIN
testdb=> INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
  "duplica":"Third","duplica":"Last"}');
INSERT 0 1

-- Next, select the duplica members
testdb=> SELECT J->'duplica' AS Duplica FROM T1
  WHERE K=3;
 duplica
---------
 "Last"
(1 row)
```

After the insert we might expect that PostgreSQL has accepted the duplicates. But, the immediately following query displayed only the last duplica. Listing all members in the following query does not bring up other duplicate members.

```
-- select each member
testdb=> (SELECT (JSONB_EACH(J)).* FROM T1
  WHERE K=3);
   key   | value
---------+--------
 duplica | "Last"
(1 row)

-- This pure SQL query confirms the case
testdb=*> (SELECT J FROM T1 WHERE K=3);
          j
---------------------
 {"duplica": "Last"}
(1 row)
```

Selecting the whole JSON document reveals that only the last duplicate has been stored. All other duplicates have been

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

dropped. As consequence there is nothing left if the object "duplica":"Last" gets removed:

```
testdb=> UPDATE T1 SET J = J - 'duplica' WHERE K=3;
UPDATE 1
testdb=>(SELECT(JSONB_EACH(J)).* FROM T1 WHERE K=3);
 key | value
-----+-------
(0 rows)
```

The reason for this is that already during the INSERT of the JSONB typed column the last "duplica" wins while others are removed automatically without warnings. The PostgreSQL manual states that in case of duplicate object members only the last one is stored. But, it does not mention that this is done silently and no exception is raised. Is this the service we want? Note that we will silently lose the information in the value parts of those automatically deleted members due to accidentally having the same key names! The current version of PostgreSQL supports the WITH UNIQUE KEYS constraint clause only for the publishing function JSON_OBJECT().

G. MariaDB

MariaDB does not recognize the constraint IS JSON WITH UNIQUE KEYS. Let's test what happens when we enter a JSON document having duplicate member keys:

```
MariaDB [testdb]> -- Duplicate object members test:
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> INSERT INTO T1 (K, J) VALUES
   -> (3, '{ "duplica":"First", "duplica":"Second",
      "duplica":"Third","duplica":"Last"}');
Query OK, 1 row affected (0.000 sec)
```

MariaDB accepts duplicates without complaining because the default for inserts is “WITHOUT UNIQUE KEYS”. This can be verified with the following query:

```
MariaDB [testdb]> SELECT J FROM T1 WHERE K=3;
+----------------------------------------------+
| J
+----------------------------------------------+
| { "duplica":"First", "duplica":"Second",
 "duplica":"Third","duplica":"Last"}          |
+----------------------------------------------+
1 row in set (0.000 sec)

-- using the JSON_VALUE function returns only
-- the first member, i.e. a single member
MariaDB [testdb]> SELECT LEFT(K, 4) AS K, LEFT(
 JSON_VALUE(J, '$.duplica'), 10) AS Duplica
    -> FROM T1 WHERE K=3;
+------+---------+
| K    | Duplica |
+------+---------+
| 3    | First   |
+------+---------+
1 row in set (0.000 sec)

MariaDB [testdb]> -- How about others if the
                     "duplica" get removed
MariaDB [testdb]> UPDATE T1 SET J = JSON_
         REMOVE(J, '$.duplica') WHERE K=3;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Only the the “first” duplicate member was removed as the following queries reveal.

```
MariaDB [testdb]> SELECT LEFT(K, 4) AS K, LEFT(
  JSON_VALUE(J, '$.duplica'), 10) AS Duplica
    -> FROM T1 WHERE K=3;
+------+---------+
| K    | Duplica |
+------+---------+
| 3    | Second  |
+------+---------+
1 row in set (0.000 sec)

MariaDB [testdb]> SELECT J FROM T1 WHERE K=3;
+----------------------------------------------+
| J                                            |
+----------------------------------------------+
| {"duplica": "Second", "duplica": "Third",
  "duplica": "Last"}                           |
+----------------------------------------------+
1 row in set (0.000 sec)
```

These tests show that the current MariaDB version stores duplicates and the JSON_VALUE and JSON_REMOVE only retrieve resp. remove the first member of duplicate objects.

IV. Discussion

We see quite different behaviour for the five database products. The default configuration of Db2 and Oracle is WITH UNIQUE KEYS making it impossible to insert duplicate keys. This makes it simple to select a particular key and get a unique result. SQL Server, PostgreSQL, and MariaDB support only WITHOUT UNIQUE KEYS and therefore allow duplicates with quite different behaviour. PostgreSQL silently ignores all duplicates except the last one during the insert. It effectively forces the JSON members to have unique KEYS and loses possible duplicates. This is intolerable because users are not aware of dropping all members except the last one. There is a way to fix this flaw in the insert statement. The missing functionality WITH UNIQUE KEYS can be created as a PL/pgSQL function used as CHECK constraint for the JSON column. The CHECK constraint can be activated when the table is created. Example: `CREATE TABLE T (K SERIAL PRIMARY KEY, J JSONB NOT NULL, CONSTRAINT with_unique_keys CHECK (unique_keys(J));`

SQL Server and MariaDB store all members including the duplicates. The problem with duplicates arises when you retrieve duplicate values. Only the first duplicate is retrieved by the JSON_VALUE function ignoring all other duplicates because this function can only return a single member. In fact, the function returns the wrong result. The problem with the implementation of this function has already been mentioned in RFC8259 in 2016 but until now nothing was changed. As consequence the JSON object must be handled outside the database using string manipulation and updated as a whole again. This can be achieved with the JSON_QUERY(expression [, path]) function which retrieves all objects matching the path. The result is a string that can be manipulated and subsequently updated in the database with the JSON_MODIFY function.

TABLE I. BEHAVIOUR OF THE TESTED DATABASES

DBMS	(1) WITH UNIQUE KEYS	(2) WITHOUT UNIQUE KEYS	default	insert duplica	retrieves duplica	comment
Db2 LUW 12.1.1 Community Edition	yes	yes	(1)	no	n/a	
Oracle Version 23ai Free	yes	yes	(1)	no	n/a	
SQL Server 2017 Express Edition	no	yes	(2)	yes	first	other duplicas hidden by JSON_VALUE
PostgreSQL Version 17	n/a	n/a	(2)	no	last	other duplicas silently dropped
MariaDB Version 11.8.2	n/a	n/a	(2)	yes	first	other duplicas hidden by JSON_VALUE

V. CONCLUSION

We have presented test scenarios with special emphasis on unique JSON object keys for five popular DBMS supporting the SQL/JSON extension.

The tests show that these five database products behave quite differently making the data exchange between different products error-prone. The results are summarized in Table I. Great care is required to insist that the database uses the option WITH UNIQUE KEYS if possible.

Our recommendations is to always use the WITH UNIQUE KEYS options when exchanging JSON documents between systems and particularly when storing JSON documents in a database. If this option is not supported as in SQL Server and MariaDB unique keys can be forced with check constraints to protect against duplicate keys in JSON documents.

If the JSON data inevitably contain duplicate keys then the JSON_VALUE function is not useful as it returns only a scalar value even when there are duplicate JSON object keys. We recommend to use the JSON_QUERY function instead to retrieve the duplicates and manipulate then the result outside the database. For example, the duplicate object member keys (e.g. {`"duplica":"first", "duplica":"second", "duplica":"third"`} could be transformed into an JSON object array {`"duplica": ["first", "second", "third"]`}.

PostgreSQL is a special case as it silently drops all duplicates except the last one. This lulls the user into a false sense of security. We hope that the next version of PostgreSQL will address the above issue and provide a standard conform solution.

In a future version of the ISO/IEC SQL/JSON standard we hope to see fuctionality that can correctly handle the situation of duplicate JSON object member keys.

REFERENCES

[1] T. Bray (ed.), "The JavaScript Object Notation (JSON) Data Interchange Format", [Online]. URL: https://dl.acm.org/doi/pdf/10.17487/RFC8259, Dec. 2017, (Accessed Jan 24, 2026)

[2] G. Turutin and M. Puzevich, "PostgreSQL JSONB-based vs. Typed-column Indexing: A Benchmark for Read Queries", IEEE Dataport, October 29, 2025, doi:10.21227/fxws-3a11

[3] Z. H. Liu et al., "Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database". Proc. VLDB Endow. 13(12): 3059-3071 (2020)

[4] N.N., IBM, "Uniqueness controls for key names", Apr. 2023, [Online]. URL: https://www.ibm.com/support/pages/json-uniqueness-controls-key-names, (Accessed Jan 24, 2026)

[5] D. Petkovic, "SQL/JSON Standard: Properties and Deficiencies", Datenbank-Spektrum, Volume 17, pp 277-287, Oct 24 2017, [Online]. URL: https://link.springer.com/content/pdf/10.1007/s13222-017-0267-4.pdf, (Accessed Jan 24, 2026)

[6] D. Petkovic, "Implementation of JSON Update Framework in RDBMSs", International Journal of Computer Applications Foundation of Computer Science (FCS), NY, USA, Volume 177 - Number 37, 2020, DOI: 10.5120/ijca2020919881

[7] N. N., ISO/IEC 21778:2017 "Information technology - The JSON data interchange syntax", [Online]. URL: https://www.iso.org/standard/71616.html , (Accessed Jan 24, 2026)

[8] N. N., ISO/IEC 19075-6:2021 "Information technology — Guidance for the use of database language SQL Part 6: Support for JSON", Edition 1, [Online]. URL: https://www.iso.org/standard/78937.html, (Accessed Jan 24, 2026)

[9] N.N., ISO/IEC 9075-2:2023 "Information technology — Database languages SQLPart 2: Foundation (SQL/Foundation)", [Online]. URL: https://www.iso.org/standard/76584.html, Edition 6, 2023, URL: https://www.iso.org/standard/76584.html, (Accessed Jan 24, 2026)

[10] P. Eisentraut, "SQL:2023 is finished: Here is what's new", [Online]. URL: https://peter.eisentraut.org/blog/2023/04/04/sql-2023-is-finished-here-is-whats-new, Apr. 4, 2023, (Accessed Jan 24, 2026)

[11] M. Laiho, "JSON Data Maintenance", Appendix 1, Nov. 2025, [Online]. URL: https://dbtechnet.org/wp-content/uploads/JSON-Data-Maintenance.pdf, (Accessed Jan 27, 2026)

[12] M. Laiho, "JSON on RDBMS Databases", Sept. 2025, [Online]. URL: https://drive.google.com/file/d/1eoU9KIQrMwI0YkdPdPm6jIgi-0hRffNI/view, (Accessed Jan 24, 2026)