JSON on RDBMS Databases

Martti Laiho

Standalone collection of Appendices for the tutorial "JSON on RDBMS Databases"
The latest version of this pdf is available by link JSON Data Maintenance

Contents

Appendix 1. JSON Data Maintenance

In Search for the JSON Update Patterns

JSON Data Structure, Types and Terminology

JSON, JavaScript Object Notation, based on JavaScript, defines language-independent simple textual data structures, adaptable in most modern programming languages and SQL dialects. The terminology of JSON datatypes used on documentations and articles of various SQL/JSON dialects varies, but we try use the following terminology.

The JSON data types applied in SQL/JSON are following:

Primitive (scalar) value types:

Number: an integer, decimal, or floating-point number in textual format

String: a sequence of zero or more Unicode characters, the sequence enclosed in double quotes (except the escaped characters, see the RFC 8259)

Boolean: the value presented either by literal *true* or *false*

null: empty value presented by the literal *null*

Structured types (nestable):

Array: ordered list of zero or more JSON *elements* (ISO OBP: tokens) of any JSON data type, the list enclosed in square brackets. An element can be unnamed scalar or nestable value, for example
[a, b, c] or [1, a, 1, {b:bb}, {c:cc}], without unique requirement, i.e. same value can appear multiple times as

element in the array. Standalone name/value pairs are not allowed, but can appear enclosed in curly braces, i.e. as object members.

JSON object: consists of comma separated unordered list of zero or more *members (a.k.a. fields*[1] *in SQL/JSON or properties)*, the list enclosed in curly braces, for example an empty object as { }. The *members* are name/value pairs (a.k.a. key/value pairs) in format "name":"value", where the <name> (a.k.a. *key*) is a string enclosed in double quotes, and the <value> is any of JSON data types, an atomic type, an array, or an object. This recursive definition allows hierarchically structured JSON objects similar to XML documents. The names of members SHOULD be defined unique[2] within the object, using WITH UNIQUE KEYS clause (Petkovic 2017). However, the same name can appear for members on different nesting levels without problems.

Some implementations, such as Oracle SQL/JSON, don't require keys to be enclosed in double quotes. Numeric values, literals, as well as array and object values are not quoted.

JSON document: in SQL/JSON a JSON document in SQL statements is a JSON object enclosed in single quotes as SQL string as follows: '{ <list of members> }'

Note: all keys and the JSON literals in SQL/JSON are case-sensitive.

"CRUD" operations to the JSON structures

The ISO SQL/JSON standard is focused on JSON Query Language, leaving the UPDATE/DELETE part to be solved by SQL implementations in RDBMS systems.

In RDBMS (SQL) databases JSON data is stored as a JSON document per column of JSON type (native or implementation dependent SQL type) in SQL tables. An entire JSON document is stored as part of SQL INSERT operation or can be replaced as a whole as part of SQL UPDATE of a JSON column.

The question of uniqueness of JSON object member names is a bit confusing, since in case of duplicate names either the first or the last one will be the acting member depending on the implementation. We will discuss this topic in a later chapter, experimenting on how our selected RDBMS products behave in this respect. Both Oracle and Db2 LUW require the member names in an object to be unique. In the following we restrict to use cases where the object member names are unique.

For the CRUD operations we propose following set of *maintenance **use cases*** reasoned from the technical JSON data structure we discussed above.

[1]The term "field" has been used, for example, in SQL Server, MySQL, PostgreSQL articles/documentation.
[2] According to IETF JSON specification RFC 8259 Dec 2017 "The names within an object SHOULD be unique".. "When the names within an object are not unique, the behavior of software that receives such an object is unpredictable." Considering this the WITHOUT UNIQUE KEYS of SQL/JSON proposal and in some implementations is just a "casting defict" and should be removed. We will debate on this in the last chapter below.

Basic use case operations on part of **JSON objects** include following:

Case 1: Inserting a new member

Case 2.1: Updating value of an existing member found by key

Case 2.2: Updating value of ALL existing members found by given value
since the JSON data model does not require values of members to be unique

Case 3.1: Deleting an existing member found by key

Case 3.2: Deleting ALL existing members found by given value
since the JSON data model does not require values of members to be unique

and use case operations on part of **JSON arrays**

Case 4: Inserting a new element

Case 5.1: Updating value of an existing element found by position

Case 5.2: Updating the value of ALL existing elements found by given value
since the JSON data model does not require values of elements in an array to be unique

Case 6.1: Deleting an existing element found by position

Case 6.2: Deleting ALL existing elements found by given value
since the JSON data model does not require values of elements in an array to be unique

The paper "Implementation of JSON Update Framework in RDBMSs" (Petkovic, Feb 2020) presents almost similar list of UPDATE *primitives*, except that inserts of array elements before or after an existing element are considered as different cases, commenting that these are optional "because there is no ordering of objects [elements], generally".

The "found by given value" operations may not be typical operations on JSON data in practice, but considering the JSON data model, as we have described above, possible operations at least in theory. The designers of the ANSI SQL/JSON proposal seem to have overlooked the possible need to query all name-value pairs or name of the member containing the given value. The "ALL operations" on JSON object or array which doesn't contain duplicate values are just single JSON operations, but in operations on duplicate values may need procedural logic or use of programming languages.

Now, after presenting these JSON maintenance use cases, we continue the search for maintenance patterns, the working solutions for these use cases, based on SQL/JSON implementations in the systems we have studied.

Beside the top level of JSON document, the access patterns which we have discussed above can be applied also on accessing nested objects and arrays by proper *path expressions*.

Instead of the typical "Joe and his friends" examples in JSON literature, for our test cases we use following minimalistic documents, based just on technical JSON data structure and data types, one without duplicate values

```
'{"mem1":123, "mem2":"123", "mem3":true, "mem4":null, "mem5":[ 123, "123", true, null, [1, 2],
{} ], "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} } }'
```

and one with duplicate values of members, and duplicate array elements

```
'{"mem1": 123, "mem2": "123", "mem3": true, "mem4": null, "mem5":[ 123, "123", "string", true,
123, 124, 124, null, [1, 2], {}, 123 ], "mem6": { "m61":1, "m62":"123", "m63": true, "m64":
null, "m65":[2, 3], "m66":{} }, "mem7": 123, "mem8": "123" }'
```

Note: Also array elements could include any nested JSON structures, but according to our tests without key names. See the unnamed array and object in our example.

In the following chapters we will be experimenting with selected RDBMS systems on how to implement the JSON update use cases defined above. Every experiment will be run in a transaction which is rolled back to save the original content for next experiments.

JSON manipulation experiments using SQL/JSON of Db2 for LUW

We have discussed on use of Db2 already in the main paper "JSON_on_RDBMS_Databases" and in following we will be experimenting just on the proposed JSON update patterns above using Db2 12.1.1 on Windows 11.

Setting up the experiment

```
CREATE TABLE T1(
K  INT NOT NULL PRIMARY KEY,
J  BLOB,
CONSTRAINT Chk_UserData CHECK (SYSTOOLS.BSON_Validate(J) = 1)
);

-- inserting/initializing the contents
DELETE FROM T1;
-- basic document without duplicates
INSERT INTO T1 (K, J) VALUES
(1, SYSTOOLS.JSON2BSON('{
       "mem1":123,
       "mem2":"123",
       "mem3":true,
       "mem4":null,
       "mem5": [ 123, "123", true, null, [1, 2], {} ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
      }'));
-- document with duplicate keys and elements
INSERT INTO T1 (K, J) VALUES
(2, SYSTOOLS.JSON2BSON('{
       "mem1": 123,
       "mem2": "123",
       "mem3": true,
       "mem4": null,
       "mem5": [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} },
       "mem7": 123,
       "mem8": "123"
      }'));
COMMIT;
```

-- verifying the contents of document K = 2

```
db2 => SELECT JSON_QUERY(J, 'strict $' RETURNING VARCHAR(1000)) AS json FROM T1 WHERE K=2;

JSON
------------------------------------------------------------------------------------------------
. . .
{ "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123", "string",
true, 123, 124, 124, null, [ 1, 2 ], {  }, 123 ], "mem6" : { "m61" : 1, "m62" : "123", "m63" :
true, "m64" : null, "m65" : [ 2, 3 ], "m66" : {  } }, "mem7" : 123, "mem8" : "123"
}

  1 record(s) selected.

db2 =>
```

Accessing JSON objects

Case 1: Adding a new member on top level

```
UPDATE T1
SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {mem7:"123"}}')
WHERE K = 1;
-- verifying the contents
SELECT K, JSON_QUERY(J, '$.mem7' RETURNING CHAR(60)) AS mem7
FROM T1 WHERE K = 1;
--
ROLLBACK;

db2 => UPDATE T1
db2 (cont.) => SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {mem7:"123"}}')
db2 (cont.) => WHERE K = 1;
  Number of rows affected : 1
DB20000I  The SQL command completed successfully.
db2 => -- verifying the contents
db2 => SELECT K, JSON_QUERY(J, '$.mem7' RETURNING CHAR(60)) AS mem7
db2 (cont.) => FROM T1 WHERE K = 1;

K           MEM7
----------- ------------------------------------------------------------
          1 "123"

  1 record(s) selected.

db2 => --
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 2.1: Updating value of an existing member found by key

```
UPDATE T1
SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {mem1: 124}}')
WHERE K = 1;
-- verifying the contents
SELECT K, JSON_QUERY(J, '$.mem1' RETURNING CHAR(60)) AS mem1
FROM T1 WHERE K = 1;
--
ROLLBACK;

db2 => UPDATE T1
db2 (cont.) => SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {mem1: 124}}')
db2 (cont.) => WHERE K = 1;
  Number of rows affected : 1
DB20000I  The SQL command completed successfully.
db2 => -- verifying the contents
db2 => SELECT K, JSON_QUERY(J, '$.mem1' RETURNING CHAR(60)) AS mem1
db2 (cont.) => FROM T1 WHERE K = 1;

K           MEM1
----------- ------------------------------------------------------------
          1 124

  1 record(s) selected.

db2 => --
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 2.2: Updating value of all existing members found by given value

For this pattern we apply cleaned version of the stored procedure solution presented in Appendix 3 as follows:

```
--#SET TERMINATOR @
CREATE OR REPLACE PROCEDURE Case2_2
          (IN  kp INT,
           IN  given_value VARCHAR(100),
           IN  new_value VARCHAR(100)
          )
SPECIFIC Case2_2
LANGUAGE SQL
BEGIN ATOMIC
    DECLARE json VARCHAR(1000);
    DECLARE loop INT default 0;
    DECLARE id INT;
    DECLARE memName  VARCHAR(20);
    DECLARE oldValue VARCHAR(1000);
    DECLARE sqlcode  INT DEFAULT 0;
    DECLARE keycurs CURSOR FOR
      SELECT t.id, t.key
      FROM UNNEST(JSON_KEYS((SELECT JSON_QUERY(J, '$') FROM T1 WHERE K=kp)))
      WITH ORDINALITY AS t(key, id)
      FOR FETCH ONLY
      WITH UR;
    SELECT BSON_TO_JSON(J) INTO json FROM T1 WHERE K = kp;
    OPEN keycurs;
    WHILE sqlcode = 0 AND loop < 100 DO
       SET loop = loop + 1;
       FETCH keycurs INTO id, memName; -- Extract the value at the current index
       SET memName = REPLACE(memName, '"', '');
       SELECT JSON_QUERY(J, '$.' || memName || ' ') INTO oldValue
       FROM T1 WHERE K = kp;
       IF (oldValue = given_value) THEN
          UPDATE T1
          SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {' || memName || ': ' || new_value || '}} ')
          WHERE K = kp;
       END IF;
    END WHILE;
END;
@
--#SET TERMINATOR ;
```

Tested as follows:

```
db2 => SELECT cast(K as smallint) as k,
db2 (cont.) => JSON_QUERY(J, 'strict $' RETURNING varchar(1000)) as result
db2 (cont.) => FROM T1 WHERE K=2;
K      RESULT
-----------------------------------------------
     2 { "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123",
"string", true, 123, 124, 124, null, [ 1, 2 ], {  }, 123 ], "mem6" : { "m61" : 1, "m62" :
"123", "m63" : true, "m64" : null, "m65" : [ 2, 3 ], "m66" : {  } }, "mem7" : 123, "mem8" :
"123" }

1 record(s) selected.

db2 => CALL Case2_2 (2, '123', '129');

  Return Status = 0

db2 => SELECT cast(K as smallint) as k,
db2 (cont.) => JSON_QUERY(J, 'strict $' RETURNING varchar(1000)) as result
db2 (cont.) => FROM T1 WHERE K=2;
K      RESULT
-----------------------------------------------
     2 { "mem1" : 129, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123",
"string", true, 123, 124, 124, null, [ 1, 2 ], {  }, 123 ], "mem6" : { "m61" : 1, "m62" :
"123", "m63" : true, "m64" : null, "m65" : [ 2, 3 ], "m66" : {  } }, "mem7" : 129, "mem8" :
"123" }

1 record(s) selected.
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 3.1: Deleting an existing member found by key

Db2 SQL/JSON does not include function such as JSON_REMOVE. Only available option for deleting a JSON object member is to mark it deleted by the literal value “null”, reported as “-“, as follows:

```
UPDATE T1
SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {mem1: null}}')
WHERE K = 1;
-- verifying the contents
SELECT K, JSON_QUERY(J, '$.mem1' RETURNING CHAR(60)) AS mem1
FROM T1 WHERE K = 1;
--
ROLLBACK;

db2 => UPDATE T1
db2 (cont.) => SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {mem1: null}}')
db2 (cont.) => WHERE K = 1;
  Number of rows affected : 1
DB20000I  The SQL command completed successfully.
db2 => -- verifying the contents
db2 => SELECT K, JSON_QUERY(J, '$.mem1' RETURNING CHAR(60)) AS mem1
db2 (cont.) => FROM T1 WHERE K = 1;

K           MEM1
----------- ------------------------------------------------------------
          1 -

  1 record(s) selected.

db2 => --
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 3.2: Deleting all existing members found by given value

For this pattern we have modified from Case 2.2 the following solution

```
--#SET TERMINATOR @
CREATE OR REPLACE PROCEDURE Case3_2
          (IN  kp INT,
           IN  given_value VARCHAR(100)
          )
SPECIFIC Case3_2
LANGUAGE SQL
BEGIN ATOMIC
    DECLARE json VARCHAR(1000);
    DECLARE loop INT default 0;
    DECLARE id INT;
    DECLARE memName  VARCHAR(20);
    DECLARE oldValue VARCHAR(1000);
    DECLARE sqlcode  INT DEFAULT 0;
    DECLARE keycurs CURSOR FOR
      SELECT t.id, t.key
      FROM UNNEST(JSON_KEYS((SELECT JSON_QUERY(J, '$') FROM T1 WHERE K=kp)))
      WITH ORDINALITY AS t(key, id)
      FOR FETCH ONLY
      WITH UR;
    SELECT BSON_TO_JSON(J) INTO json FROM T1 WHERE K = kp;
    OPEN keycurs;
    WHILE sqlcode = 0 AND loop < 100 DO
       SET loop = loop + 1;
       FETCH keycurs INTO id, memName; -- Extract the value at the current index
       SET memName = REPLACE(memName, '"', '');
       SELECT JSON_QUERY(J, '$.' || memName || ' ') INTO oldValue
```

```
       FROM T1 WHERE K = kp;
       IF (oldValue = given_value) THEN
          UPDATE T1
           SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {' || memName || ': null }} ')
          WHERE K = kp;
       END IF;
    END WHILE;
END;
@
--#SET TERMINATOR ;
```

Tested as follows

```
db2 => SELECT cast(K as smallint) as k,
db2 (cont.) => JSON_QUERY(J, 'strict $' RETURNING varchar(1000)) as result
db2 (cont.) => FROM T1 WHERE K=2;
K      RESULT
------ ----------------------------------------------
     2 { "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123",
"string", true, 123, 124, 124, null, [ 1, 2 ], {  }, 123 ], "mem6" : { "m61" : 1, "m62" :
"123", "m63" : true, "m64" : null, "m65" : [ 2, 3 ], "m66" : {  } }, "mem7" : 123, "mem8" :
"123" }

  1 record(s) selected.

db2 => CALL Case3_2 (2, '123');

  Return Status = 0

db2 => SELECT cast(K as smallint) as k,
db2 (cont.) => JSON_QUERY(J, 'strict $' RETURNING varchar(1000)) as result
db2 (cont.) => FROM T1 WHERE K=2;

K      RESULT
------ ----------------------------------------------------
     2 { "mem1" : null, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123",
"string", true, 123, 124, 124, null, [ 1, 2 ], {  }, 123 ], "mem6" : { "m61" : 1, "m62" :
"123", "m63" : true, "m64" : null, "m65" : [ 2, 3 ], "m66" : {  } }, "mem7" : null, "mem8" :
"123" }

  1 record(s) selected.

db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Note: Deleted members are not removed in Db2, but marked as deleted by value “null”. To remove those members having value “null”, we would need to first copy the json data to string operations in some external routine, and finally replace the original row column in database by the operated json data.

Accessing JSON arrays

Case 4: Adding a new element into an array

```
-- using by too big index, the new element is appended in the array
UPDATE T1
SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {"mem5.100": 127}}')
WHERE K = 1;
-- verifying the contents
SELECT K, JSON_QUERY(J, '$.mem5' RETURNING CHAR(60)) AS mem5
FROM T1 WHERE K = 1;
--
ROLLBACK;

db2 => UPDATE T1
```

```
db2 (cont.) => SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {"mem5.100": 127}}')
db2 (cont.) => WHERE K = 1;
  Number of rows affected : 1
DB20000I  The SQL command completed successfully.
db2 => -- verifying the contents
db2 => SELECT K, JSON_QUERY(J, '$.mem5' RETURNING CHAR(60)) AS mem5
db2 (cont.) => FROM T1 WHERE K = 1;

K           MEM5
----------- ------------------------------------------------------------
          1 [ 123, "123", true, null, [ 1, 2 ], {  }, 127 ]

  1 record(s) selected.

db2 => --
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 5.1: Updating value of an existing element found by position

Updating existing element in given position in array value of given member is supported in Db2 SQL/JSON as follows for index 0 of mem5 member:

```
UPDATE T1
SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {"mem5.0": 124}}')
WHERE K = 1;
-- verifying the contents
SELECT K, JSON_QUERY(J, '$.mem5' RETURNING CHAR(60)) AS mem5
FROM T1 WHERE K = 1;
--
ROLLBACK;

db2 => UPDATE T1
db2 (cont.) => SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {"mem5.0": 124}}')
db2 (cont.) => WHERE K = 1;
  Number of rows affected : 1
DB20000I  The SQL command completed successfully.
db2 => -- verifying the contents
db2 => SELECT K, JSON_QUERY(J, '$.mem5' RETURNING CHAR(60)) AS mem5
db2 (cont.) => FROM T1 WHERE K = 1;

K           MEM5
----------- ------------------------------------------------------------
          1 [ 124, "123", true, null, [ 1, 2 ], {  } ]

  1 record(s) selected.

db2 => --
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 5.2: Updating value of all existing elements found by value

For this pattern we have built following stored procedure which first checks number of elements and then browses the array of elements one at a time and updates those having the “given value” by the “new value”

```
--#SET TERMINATOR @
CREATE OR REPLACE PROCEDURE Case5_2
          (IN  kp INT,
           in  memName     VARCHAR(20),
           IN  given_value VARCHAR(100),
           IN  new_value   VARCHAR(100)
          )
SPECIFIC Case5_2
LANGUAGE SQL
BEGIN ATOMIC
```

```
   DECLARE ind       INT;
   DECLARE elemCount INT;
   DECLARE qry_expr  VARCHAR(100);
   DECLARE old_value VARCHAR(100);
   DECLARE upd_expr  VARCHAR(100);
   --
   SELECT SYSTOOLS.JSON_LEN(J, 'mem5') INTO elemCount
     FROM T1 WHERE K = kp;
   -- Loop through the JSON members
   SET ind = 0;
   WHILE ind < elemCount DO
     BEGIN
       -- Extract the value at the current index
       SET qry_expr = '$.'|| memName || '[' || ind || ']' ;
       SELECT JSON_VALUE(J, '' || qry_expr || '') INTO old_value
          FROM T1 WHERE K = kp;
       IF (old_value = given_value) THEN BEGIN
          SET upd_expr =
             '{$set: {"' || memName || '.' || ind || '": ' || new_value || ' }}';
          UPDATE T1
             SET J = SYSTOOLS.JSON_UPDATE(J, '' || upd_expr || '')
             WHERE K = kp;
         END;
       END IF;
       -- Increment the index
       SET ind = ind + 1;
     END;
   END WHILE;
END;
@
--#SET TERMINATOR ;
```

Testing the pattern as follows

```
SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;
CALL Case5_2 (1, 'mem5','123', '127');
SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;
ROLLBACK;

db2 => SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;
1
-----------------------------------------------------------------------------------------------
{ "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123", true,
null, [ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null, "m65"
: [ 2, 3 ], "m66" : {  } } }

  1 record(s) selected.

db2 => CALL Case5_2 (1, 'mem5','123', '127');

  Return Status = 0
db2 => SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;
1
-----------------------------------------------------------------------------------------------
{ "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 127, 127, true, null,
[ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null, "m65" : [
2, 3 ], "m66" : {  } } }

  1 record(s) selected.

db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Note: The solution treats integer value 123 and string value “123” as the sme.

Case 6.1: Deleting an existing element found by position

Like deleting a member by only marking it as deleted by value “null”, the same concerns elements in the array value of a given member.

```
-- See Baklarz p 135
-- "It is not actually possible to remove an item from an array, but it is
--  possible to set the specific array value to null."

UPDATE T1
SET J = SYSTOOLS.JSON_UPDATE(J, '{$unset: {"mem5.0": null}}')
WHERE K = 1;
-- verifying the contents
SELECT K, JSON_QUERY(J, '$.mem5' RETURNING CHAR(60)) AS mem5
FROM T1 WHERE K = 1;
--
ROLLBACK;
```

```
db2 => UPDATE T1
db2 (cont.) => SET J = SYSTOOLS.JSON_UPDATE(J, '{$unset: {"mem5.0": null}}')
db2 (cont.) => WHERE K = 1;
  Number of rows affected : 1
DB20000I  The SQL command completed successfully.
db2 => -- verifying the contents
db2 => SELECT K, JSON_QUERY(J, '$.mem5' RETURNING CHAR(60)) AS mem5
db2 (cont.) => FROM T1 WHERE K = 1;

K           MEM5
----------- ------------------------------------------------------------
          1 [ null, "123", true, null, [ 1, 2 ], {  } ]

  1 record(s) selected.

db2 => --
db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Case 6.2: Deleting all existing elements found by given value

This pattern is based on the pattern 5.2 modified to replace the old matching elements by literal “null”.

```
--#SET TERMINATOR @
CREATE OR REPLACE PROCEDURE Case6_2
          (IN  kp INT,
           in  memName     VARCHAR(20),
           IN  given_value VARCHAR(100)
          )
SPECIFIC Case6_2
LANGUAGE SQL
BEGIN ATOMIC
   DECLARE ind       INT;
   DECLARE elemCount INT;
   DECLARE qry_expr  VARCHAR(100);
   DECLARE old_value VARCHAR(100);
   DECLARE upd_expr  VARCHAR(100);
   --
   SELECT SYSTOOLS.JSON_LEN(J, 'mem5') INTO elemCount
     FROM T1 WHERE K = kp;
   -- Loop through the JSON members
   SET ind = 0;
   WHILE ind < elemCount DO
     BEGIN
       -- Extract the value at the current index
       SET qry_expr = '$.'|| memName || '[' || ind || ']' ;
       SELECT JSON_VALUE(J, '' || qry_expr || '') INTO old_value
```

```
          FROM T1 WHERE K = kp;
       IF (old_value = given_value) THEN BEGIN
          SET upd_expr =
             '{$set: {"' || memName || '.' || ind || '": null }}';
          UPDATE T1
             SET J = SYSTOOLS.JSON_UPDATE(J, '' || upd_expr || '')
             WHERE K = kp;
         END;
       END IF;
       -- Increment the index
       SET ind = ind + 1;
     END;
   END WHILE;
END;
@
--#SET TERMINATOR ;
COMMIT;

SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;
CALL Case6_2 (1, 'mem5','123');
SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;
ROLLBACK;

db2 => SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;

1
-----------------------------------------------------------------------------------------------
{ "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123", true,
null, [ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null, "m65"
: [ 2, 3 ], "m66" : {  } } }

  1 record(s) selected.

db2 => CALL Case6_2 (1, 'mem5','123');

  Return Status = 0
db2 => SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;

1
-----------------------------------------------------------------------------------------------
{ "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ null, null, true,
null, [ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null, "m65"
: [ 2, 3 ], "m66" : {  } } }

  1 record(s) selected.

db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

The “deleted” elements are now just replaced by “null” values. However, copying the whole element list of the selected member as string to a variable “elements”, using REPLACE(elements, 'null,', '') function the null elements can be removed. Then updating the selected member with the updated element list, all “deleted” elements will be removed like in following experiment:

```
db2 => CALL Case6_2 (1, 'mem5','123');

  Return Status = 0
db2 => SELECT JSON_QUERY(J, '$' returning varchar(300)) FROM T1 WHERE K = 1;

1
{ "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ null, null, true,
null, [ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null, "m65"
: [ 2, 3 ], "m66" : {  } } }

  1 record(s) selected.

db2 => --#SET TERMINATOR @
```

```
db2 => CREATE OR REPLACE PROCEDURE CleanNullsFor
db2 (cont.) =>    (IN  kp INT,
db2 (cont.) =>     IN  memName       VARCHAR(30),
db2 (cont.) =>     OUT qryPath       VARCHAR(100),
db2 (cont.) =>     OUT upd_expr      VARCHAR(500),
db2 (cont.) =>     OUT elems_before  VARCHAR(400),
db2 (cont.) =>     OUT elements      VARCHAR(400)
db2 (cont.) =>    )
db2 (cont.) => SPECIFIC CleanNullsFor
db2 (cont.) => LANGUAGE SQL
db2 (cont.) => BEGIN
db2 (cont.) =>   -- DECLARE elements VARCHAR(4000);
db2 (cont.) =>   -- DECLARE path VARCHAR(100);
db2 (cont.) =>   SET qryPath = '$.' || memName ;
db2 (cont.) =>   SELECT JSON_QUERY(J, '' || qryPath || ''RETURNING VARCHAR(400))
db2 (cont.) =>      INTO elements
db2 (cont.) =>      FROM T1 WHERE K = kp;
db2 (cont.) =>   set elems_before = elements;
db2 (cont.) =>   SET elements = REPLACE(elements, 'null,', '');
db2 (cont.) =>   SET elements = REPLACE(elements, ' ', '');
db2 (cont.) =>   SET elements = REPLACE(elements, ',', ', ');
db2 (cont.) =>   SET elements = REPLACE(elements, ', null]', ']');
db2 (cont.) =>   -- SET updPath = '$.' || memName || ': ' || elements || '' ;
db2 (cont.) =>   SET upd_expr =
db2 (cont.) =>      '{$set: {"' || memName || '": ' || elements || '}}';
db2 (cont.) =>   UPDATE T1
db2 (cont.) =>   SET J = SYSTOOLS.JSON_UPDATE(J, '' || upd_expr || '')
db2 (cont.) =>     WHERE K = kp;
db2 (cont.) => END;
db2 (cont.) => @
DB20000I  The SQL command completed successfully.
db2 => --#SET TERMINATOR ;
db2 => --
db2 => CALL CleanNullsFor(1, 'mem5',?,?,?,?);

  Value of output parameters
  --------------------------
  Parameter Name  : QRYPATH
  Parameter Value : $.mem5

  Parameter Name  : UPD_EXPR
  Parameter Value : {$set: {"mem5": [true, [1, 2], {}]}}

  Parameter Name  : ELEMS_BEFORE
  Parameter Value : [ null, null, true, null, [ 1, 2 ], {  } ]

  Parameter Name  : ELEMENTS
  Parameter Value : [true, [1, 2], {}]

  Return Status = 0
db2 => SELECT JSON_QUERY(J, '$.mem5' RETURNING VARCHAR(400)) FROM T1 WHERE K= 1;

1
--------------------------------------------------------------------------------
[ true, [ 1, 2 ], {  }
]

  1 record(s) selected.

db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
```

Now that we have verified the intermediate values of the OUT parameters, we can move them into local variables. To change the semantics of “delete” to “remove”, the actions of “`CleanNullsFor`” should be applied at the end of procedure “`Case6_2`”.

Summary

SQL/JSON implementations on different Db2 editions vary. We are interested in the Db2 for LUW (Linux, Unix and Windows) edition. Surprisingly, in this edition the JSON_TABLE function does not include the FOR ORDINALITY clause. Also, it doesn't include means for sorting out the count of top-level members in a JSON document, and no means for accessing the members by position. The missing JSON_KEYS() function could be implemented by some Python subprogram applying the keys() function as following

```
>>> myjson = {"mem1":123,"mem2":"123","mem3":true,"mem4":null,"mem5": [ 123, "123", true, null, [1, 2], {} ],"mem6": { "m61":1,
"m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }}
>>> print(myjson.keys())
dict_keys(['mem1', 'mem2', 'mem3', 'mem4', 'mem5', 'mem6'])
>>>
```

but the interface needs to be sorted out somehow (see Appendix 3)

JSON manipulation experiments using SQL/JSON of Oracle 23ai

The ISO SQL/JSON of Oracle native JSON data type based on its binary storage implementation OSON discussed by Liu et al. (SIGMOD 2016, VLDB 2020) and Gugnani et al. (SIGMOD 2025) featuring direct in-memory updates of JSON document without full document replacements.
In addition to the general numeric data types of JSON, JSON/OSON extends types to packed decimal, IEEE float/double, date, timestamp, interval, and raw types of SQL primitives.

From SQL/JSON 2016 standard JSON/OSON implements cast functions of strings such as .number(), .string(), .date(), .binary() to non-string data types.

The SQL/JSON path expression/filter expression implementations of JSON_TRANSFORM function will greatly simplify our JSON update experiments and obviously provide performance boost.

Setting up the experiment

For our experiments we create the following SQL table for storing our JSON document

```
CREATE TABLE T1(
K  INT NOT NULL PRIMARY KEY,
J  JSON);
```

Inserting JSON document as part of an inserted rows

```
-- our basic document without duplicates
INSERT INTO T1 (K, J) VALUES
(1, '{ mem1: 123,
       mem2: "string",
       mem3: true,
       mem4: null,
       mem5: [ 123, "string", true, null, [] ],
       mem6: { m62: "string", m63: true, m64: null, m65: [], m66: {} }
    }' );
-- document with duplicate keys and elements
INSERT INTO T1 (K, J) VALUES
(2, '{ mem1: 123,
```

```
       mem2: "123",
       mem3: true,
       mem4: null,
       mem5: [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ],
       mem6: { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3],
"m66":{} },
       mem7: 123,
       mem8: "123"
    }' );
COMMIT;
```

In spite of the ISO SQL/JSON specification the JSON update operations in different RDBMS systems and even versions differ. In the following we will focus on use of Oracle 23ai implementation making extensive use of JSON path and its filter expressions[3].

Accessing JSON objects

Case 1: Adding a new member on top level

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, INSERT '$.mem7' = 'new')
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    123,
    "123",
    true,
    null,
    [
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  },
```

[3]Filter expressions are defined in chapter "5.12 Filter expression" of "SQL/JSON: Part 2 -Querying JSON", and also explained on page "17.2 SQL/JSON Path Expression Syntax" of JSON Developer's Guide.

```
  "mem7" : "new"
}

SQL> ROLLBACK;

Rollback complete.
```

Case 2.1: Updating value of an existing member found by key

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, SET '$.mem1' = 124)
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 124,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    123,
    "123",
    true,
    null,
    [
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.
```

Case 2.2: Updating value of all existing members found by given value

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REPLACE '$.*?(@==123)' = 124)
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
```

```
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 124,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" : 124,
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.

SQL>
```

Case 3.1: Deleting an existing member found by key

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REMOVE '$.mem1')
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    123,
    "123",
    true,
    null,
    [
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
```

```
  }
}

SQL> ROLLBACK;

Rollback complete.
```

Case 3.2: Deleting all existing members found by given value

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REMOVE '$.*?(@==123)')
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.

UPDATE T1
SET J = JSON_TRANSFORM (J, REMOVE '$.*?(@==123)')
WHERE K = 2;
SELECT JSON_SERIALIZE (J PRETTY)
FROM T1 WHERE K = 2;
ROLLBACK;

SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REMOVE '$.*?(@==123)')
  3* WHERE K = 2;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 2;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem3" : true,
  "mem4" : null,
  "mem6" :
  {
    "m61" : 1,
    "m62" : "123",
    "m63" : true,
    "m64" : null,
```

```
    "m65" :
    [
      2,
      3
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.
```

Accessing JSON arrays

Case 4: Adding a new element into an array

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, APPEND '$.mem5' = 124)
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
_______________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    123,
    "123",
    true,
    null,
    [
    ],
    124
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.

SQL>
```

Case 5.1: Updating value of an existing element found by position

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, SET '$.mem5[0]' = 124)
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    124,
    "123",
    true,
    null,
    [
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.
```

Case 5.2: Updating value of all existing elements found by value

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REPLACE '$.mem5[*]?(@==123)' = 124)
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    124,
    "123",
    true,
    null,
    [
```

```
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.
```

Case 6.1: Deleting an existing element found by position

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REMOVE '$.mem5[0]')
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
_______________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    "123",
    true,
    null,
    [
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;
```

```
Rollback complete.

SQL>
```

Case 6.2: Deleting all existing elements found by given value

```
SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REMOVE '$.mem5[*]?(@==123)')
  3* WHERE K = 1;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY)
  2* FROM T1 WHERE K = 1;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    "123",
    true,
    null,
    [
    ]
  ],
  "mem6" :
  {
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
    ],
    "m66" :
    {
    }
  }
}

SQL> ROLLBACK;

Rollback complete.

SQL> UPDATE T1
  2  SET J = JSON_TRANSFORM (J, REMOVE '$.mem5[*]?(@==123)')
  3* WHERE K = 2;

1 row updated.

SQL> SELECT JSON_SERIALIZE (J PRETTY) FROM T1 WHERE K = 2;

JSON_SERIALIZE(JPRETTY)
__________________________
{
  "mem1" : 123,
  "mem2" : "123",
  "mem3" : true,
  "mem4" : null,
  "mem5" :
  [
    "string",
    true,
    124,
```

```
    124,
    null,
    [
      1,
      2
    ],
    {
    }
  ],
  "mem6" :
  {
    "m61" : 1,
    "m62" : "123",
    "m63" : true,
    "m64" : null,
    "m65" :
    [
      2,
      3
    ],
    "m66" :
    {
    }
  },
  "mem7" : 123,
  "mem8" : "123"
}

SQL> ROLLBACK;

Rollback complete.
```

JSON manipulation experiments using T-SQL/JSON of SQL Server

See the documentation at
https://learn.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver17

Experimenting the basic operations on part of JSON objects or arrays

Setting up the experiment

In following the tests are run by SQL Server Express edition 2022

```
USE JsonDemo;
DROP TABLE T1;
CREATE TABLE T1(
K  INT NOT NULL PRIMARY KEY,
J  NVARCHAR(MAX),     -- Note: the JSON type is not yet available
CONSTRAINT CHK_J_JSON CHECK (ISJSON(J) = 1)
);
```

For our experiments we insert into table T1 the following contents:

```
-- Inserting our test documents
-- Note: in T-SQL/JSON key names need to be enclosed in double quotes!

BEGIN TRANSACTION;
-- our basic document without duplicates
INSERT INTO T1 (K, J) VALUES
(1, SYSTOOLS.JSON2BSON('{
       "mem1":123,
       "mem2":"123",
```

```
       "mem3":true,
       "mem4":null,
       "mem5": [ 123, "123", true, null, [1, 2], {} ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
      }'));
-- document with duplicate keys and elements
INSERT INTO T1 (K, J) VALUES
(2, SYSTOOLS.JSON2BSON('{
       "mem1": 123,
       "mem2": "123",
       "mem3": true,
       "mem4": null,
       "mem5": [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} },
       "mem7": 123,
       "mem8": "123"
      }'));
COMMIT;
```

T-SQL/JSON doesn't have any pretty print format for JSON, but for example, we can use JSON_VALUE for simple values by limiting value lengths by LEFT() functions

```
SELECT LEFT(K, 4) AS K,
    LEFT(JSON_VALUE(J, '$.mem1'), 4) AS mem1,
    LEFT(JSON_VALUE(J, '$.mem2'), 6) AS mem2,
    LEFT(JSON_VALUE(J, '$.mem3'), 6) AS mem3,
    LEFT(JSON_VALUE(J, '$.mem4'), 6) AS mem4,
    LEFT(JSON_VALUE(J, '$.mem5[0]'), 10) AS mem5_0,
    LEFT(JSON_VALUE(J, '$.mem5[4]'), 10) AS mem5_4,
    LEFT(JSON_VALUE(J, '$.mem5[4][1]'), 10) AS mem5_4_1,
    LEFT(JSON_VALUE(J, '$.mem6.m61'), 10) AS mem6_m61,
    LEFT(JSON_VALUE(J, '$.mem6.m66'), 10) AS mem6_m66
FROM T1;

K    mem1 mem2   mem3   mem4   mem5_0     mem5_4     mem5_4_1   mem6_m61   mem6_m66
---- ---- ------ ------ ------ ---------- ---------- ---------- ---------- ----------
1    123  string true   NULL   123        NULL       2          1          NULL

(1 row affected)
```

OPENJSON function of T-SQL/JSON

https://www.sqlservertutorial.net/sql-server-json-functions/sql-server-openjson/

Many examples SQL Server documentation use the proprietary function OPENJSON of T-SQL/JSON. OPENJSON maps JSON text into a set of rows and columns, returning columns: key, value, and type, for each key/value pair in the JSON.

```
syntax:
       OPENJSON( jsonExpression [ , path ] )  [ <with_clause> ]
       <with_clause> ::= WITH ( { colName type [ column_path ] [ AS JSON ] } [ ,...n ] )
```

Optional WITH clause defines an explicit schema, specifying columns, their types, and the JSON path for each value, allowing load JSON data directly into SQL Server tables.

In following we try to apply the Example 6 - Simple example with JSON content for parsing key/value pairs in our test JSON.

```
ALTER DATABASE JsonDemo SET COMPATIBILITY_LEVEL = 130;
DECLARE @json NVARCHAR(MAX);
SELECT @json = J FROM T1 WHERE K=1;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);

key      value
```

```
-------- ----------------------------------------------------------------------------------------
mem1     123
mem2     string
mem3     true
mem4     NULL
mem5     [ 123, "string", true, null, [1, 2], {} ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }

(6 rows affected)

DECLARE @json NVARCHAR(MAX);
SELECT @json = J FROM T1 WHERE K=2;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);

key      value
-------- ----------------------------------------------------------------------------------------
mem1     123
mem2     123
mem3     true
mem4     NULL
mem5     [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
mem7     123
mem8     123

(8 rows affected)
```

Surprisingly, in the OPENJSON report above, both the integer value 123 and string value “123” are listed as integers.

Manipulation tests of our JSON test document:

Accessing JSON objects

Case 1: Adding a new member on top level

```
EGIN TRANSACTION;
UPDATE T1
SET J = JSON_MODIFY(J, '$.mem7', 'new value')
WHERE K = 1;

(1 row affected)

SELECT LEFT(JSON_VALUE(J, '$.mem7'), 10) AS mem7 FROM T1;
mem7
----------
new value
ROLLBACK;
```

Case 2.1: Updating value of an existing member found by key

```
BEGIN TRANSACTION;
UPDATE T1
SET J = JSON_MODIFY(J, '$.mem4', 'new value')
WHERE K = 1;

(1 row affected)

SELECT LEFT(JSON_VALUE(J, '$.mem4'), 10) AS mem4 FROM T1;
mem4
----------
new value

ROLLBACK;
```

Case 2.2: Updating value of all existing members found by given value

For this we will use following script, first to the basic test document

```
BEGIN TRANSACTION;
DECLARE @json NVARCHAR(MAX);
DECLARE @path NVARCHAR(20);
DECLARE @key NVARCHAR(20);
SELECT @json = J FROM T1 WHERE K=1;
DECLARE cur CURSOR FOR
  SELECT [key]
  FROM OPENJSON(@json)
  WHERE [value] = '123';
OPEN cur;
FETCH NEXT FROM cur INTO @key;
WHILE @@FETCH_STATUS = 0
  BEGIN
    SET @path = CONCAT('$.', @key);
    UPDATE T1 SET J = JSON_MODIFY(J, @path, '124');
    FETCH NEXT FROM cur INTO @key;
  END;
CLOSE cur;
DEALLOCATE cur;
GO
SELECT LEFT(JSON_VALUE(J, '$.mem1'), 10) AS mem1
FROM T1;

(1 row affected)
mem1
----------
124

(1 row affected)
ROLLBACK;
```

.. and applying it next to JSON document of K 2:

```
BEGIN TRANSACTION;
DECLARE @json NVARCHAR(MAX);
DECLARE @path NVARCHAR(20);
DECLARE @key NVARCHAR(20);
SELECT @json = J FROM T1 WHERE K=2;
DECLARE cur CURSOR FOR
  SELECT [key]
  FROM OPENJSON( @json )
  WHERE [value] = '123';
OPEN cur;
FETCH NEXT FROM cur INTO @key;
WHILE @@FETCH_STATUS = 0
  BEGIN
    SET @path = CONCAT('$.', @key);
    UPDATE T1 SET J = JSON_MODIFY(J, @path, '124');
    FETCH NEXT FROM cur INTO @key;
  END;
CLOSE cur;
DEALLOCATE cur;
GO
DECLARE @json NVARCHAR(MAX);
SELECT @json = J FROM T1 WHERE K=2;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);
GO

(2 rows affected)

(2 rows affected)
```

```
(2 rows affected)

(2 rows affected)
key      value
-------- ----------------------------------------------------------------------------------------
mem1     124
mem2     124
mem3     true
mem4     NULL
mem5     [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
mem7     124
mem8     124

(8 rows affected)
ROLLBACK;
```

Surprisingly beside the integer value 123 of members mem1, mem7 and mem8, this affected also to string “123” in mem2.

Case 3.1: Deleting an existing member found by key

```
BEGIN TRANSACTION;
UPDATE T1 SET J = JSON_MODIFY(J, '$.mem1', NULL);
GO
DECLARE @json NVARCHAR(MAX);
SELECT @json = J FROM T1 WHERE K=1;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);
GO

(2 rows affected)
key      value
-------- ----------------------------------------------------------------------------------------
mem2     string
mem3     true
mem4     NULL
mem5     [ 123, "string", true, null, [1, 2], {} ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }

(5 rows affected)
ROLLBACK;
```

Unlike by some other implementations, in T-SQL/JSON setting the value of member to NULL removes the selected member.

Case 3.2: Deleting ALL existing members found by given value

```
BEGIN TRANSACTION;
DECLARE @json NVARCHAR(MAX);
DECLARE @path NVARCHAR(20);
DECLARE @key NVARCHAR(20);
SELECT @json = J FROM T1 WHERE K=2;
DECLARE cur CURSOR FOR
  SELECT [key]
  FROM OPENJSON( @json )
  WHERE [value] = '123';
OPEN cur;
FETCH NEXT FROM cur INTO @key;
WHILE @@FETCH_STATUS = 0
  BEGIN
    SET @path = CONCAT('$.', @key);
```

```
    UPDATE T1 SET J = JSON_MODIFY(J, @path, NULL);
    FETCH NEXT FROM cur INTO @key;
  END;
CLOSE cur;
DEALLOCATE cur;
GO
DECLARE @json NVARCHAR(MAX);
SELECT @json = J FROM T1 WHERE K=2;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);
GO

(2 rows affected)

(2 rows affected)

(2 rows affected)

(2 rows affected)
key      value
-------- -------------------------------------------------------------------------------------------
mem3     true
mem4     NULL
mem5     [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }

(4 rows affected)
ROLLBACK;
```

Like in 3.1 the function call of `JSON_MODIFY(J, @path, NULL)` removed all affected members.

Accessing JSON arrays

Case 4: Adding a new element into an array

```
BEGIN TRANSACTION;
UPDATE T1
SET J = JSON_MODIFY(J, 'append $.mem6.m65', 4)
WHERE K = 1;

(1 row affected)

SELECT J FROM T1 WHERE K = 1;
J
-------------------------------------------------------------------------------------------------
{ "mem1":123, "mem2":"123", "mem3":true, "mem4":null,
   "mem5": [ 123, "123", true, null, [1, 2], {} ],
   "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3,4], "m66":{} }
}

(1 row affected)
ROLLBACK;
```

Case 5.1: Updating value of an existing element found by position

```
BEGIN TRANSACTION;
UPDATE T1
SET J = JSON_MODIFY(J, '$.mem6.m65[1]', 5)
WHERE K = 1;

(1 row affected)

SELECT J FROM T1 WHERE K = 1;
J
-------------------------------------------------------------------------------------------------
{ "mem1":123, "mem2":"123", "mem3":true, "mem4":null,
```

```
   "mem5": [ 123, "123", true, null, [1, 2], {} ],
   "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 5], "m66":{} }
}

(1 row affected)
ROLLBACK;
```

Case 5.2: Updating value of all existing elements found by value

We test this on the array of mem5 in document K=2, first copying the array contents into temporary variable @arr5, then replacing all 123 values by value 125, and finally updating the array value of member mem5 by the modified variable:

```
BEGIN TRANSACTION;
DECLARE @json NVARCHAR(MAX);
DECLARE @arr5 NVARCHAR(100);
SELECT @json = J FROM T1 WHERE K=2;
SELECT @arr5 = [value]
FROM OPENJSON(@json) WHERE [key] = 'mem5';
SET @arr5 = REPLACE(@arr5, '123', '125');
UPDATE T1
SET J = JSON_MODIFY(J, '$.mem5', @arr5);
-- verifying the contents
SELECT @json = J FROM T1 WHERE K=2;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);

key      value
-------- ----------------------------------------------------------------------------------------
mem1     123
mem2     123
mem3     true
mem4     NULL
mem5     [ 125, "125", "string", true, 125, 124, 124, null, [1, 2], {}, 125 ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
mem7     123
mem8     123

(8 rows affected)
ROLLBACK;
```

Case 6.1: Deleting an existing element found by position

Our example below, based on solution by Kari Silpiö, will remove the second element (string “123”) from the array value of member mem5. The general Common Table Expression (CTE) is explained, for example at https://learn.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver16

```
BEGIN TRANSACTION;
DECLARE @json NVARCHAR(MAX);
DECLARE @arr5 NVARCHAR(MAX);
-- Replacing the element [1] by empty string ''
UPDATE T1
SET J = JSON_MODIFY(J, '$.mem5[1]', '')
WHERE K = 2;
-- The '' value is updated as "" which will be as removed as follows
WITH cte (arrayAsString) AS
  (SELECT JSON_QUERY(J, '$.mem5') FROM T1 WHERE K = 2)
SELECT @arr5 = REPLACE(REPLACE(arrayAsString, ',""', ''),', ""', '')
FROM cte;
-- updated @arr5 will now be set back to the document
UPDATE T1
```

```
SET J = JSON_MODIFY(J, '$.mem5', @arr5);
SELECT @json = J FROM T1 WHERE K=2;

(1 row affected)
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);

(2 rows affected)
key      value
-------- -------------------------------------------------------------------------------------
mem1     123
mem2     123
mem3     true
mem4     NULL
mem5     [ 123, "string", true, 123, 124, 124, null, [1, 2], {}, 123 ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
mem7     123
mem8     123

(8 rows affected)
ROLLBACK;
```

Note: this did not touch the original null element on the array.

Case 6.2: Deleting all existing elements found by given value

In following we will experiment on deleting all elements of duplicate value 123 in the array of mem5 in our test document K=2

```
BEGIN TRANSACTION;
DECLARE @json NVARCHAR(MAX);
DECLARE @arr  NVARCHAR(MAX);
-- reading the JSON to temporary variable @json
SELECT @json = J FROM T1 WHERE K=2;
-- reading the array value of mem5 temporary variable @arr
SELECT @arr = [value]
FROM OPENJSON(@json) WHERE [key] = 'mem5';
print 'tracing the work on the arr copy:';
print @arr;
-- Replacing elements having value 123 by empty string ''
-- (thus keeping possible null values)
SET @arr = REPLACE(@arr, '123', '');
print @arr;
-- fixing the carbage strings
SET @arr = REPLACE(@arr, '[ ,', '[');
SET @arr = REPLACE(@arr, '"",', '');
SET @arr = REPLACE(@arr, ', ,', ',');
-- restoring the fixed array as new value of mem5
UPDATE T1
SET J = JSON_MODIFY(J, '$.mem5', @arr)
WHERE K=2;
print 'verifying the contents';
SELECT @json = J FROM T1 WHERE K=2;
SELECT LEFT([key], 8) AS [key], [value]
FROM OPENJSON(@json);

tracing the work on the arr copy:
[ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ]
[ , "", "string", true, , 124, 124, null, [1, 2], {},  ]

(1 row affected)
verifying the contents
key      value
-------- -------------------------------------------------------------------------------------
mem1     123
mem2     123
mem3     true
```

```
mem4     NULL
mem5     [  "string", true, 124, 124, null, [1, 2], {},  ]
mem6     { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
mem7     123
mem8     123

(8 rows affected)
ROLLBACK;
```

Summary

The missing filter expression implementation makes current T-SQL/JSON quite different for developers, compared with the SQL/JSON implementations of Oracle and PostgreSQL. However, the dedicated OPENJSON function makes life more easy.

JSON manipulation experiments using pSQL/JSON of PostgreSQL

PostgreSQL community proceeding with the development of the OpenSource edition of the PostgreSQL system has been innovative extending the relational system with its native JSON implementation in 2012, before ANSI woke up to need for the SQL/JSON standard. So, the PostgreSQL already had "exotic" JSON manipulation operators of its own, when the ANSI workgroup came in 2014 with their proposal on SQL/JSON query language. The PostgreSQL community was awake and adapted the SQL/JSON query functions in the SQL language of their own, including the path expression and filter expressions, with flavours of their own.

For the storage datatype PostgreSQL has both plain textual JSON and binary JSONB with type-sensitive functions of their own and type cast operators "::type" back and forth.

Special JSON extract operators "->" to text value and "->>" to object type add options in its pSQL/JSON dialect (see NEON's article).

The result is a "culture shock" for us, developers who have seen PostgreSQL just as an SQL dialect.

Setting up the experiment

In following the tests are run by version 17 of PostgreSQL in Debian 12 VM in which we have created testdb database.
Using psql client we turn autocommit mode off, create table T1 and insert there our test document as follows

```
dbtech@debian11:~$ psql testdb
psql (13.18 (Debian 13.18-0+deb11u1))
Type "help" for help.

testdb=# select version();
PostgreSQL 17.5 ...

-- Note: this is case sensitive!
\set AUTOCOMMIT OFF

CREATE TABLE T1(
K  INT NOT NULL PRIMARY KEY,
J  JSONB);
```

Note: JSONB is the binary storage solution of PostgreSQL. Benefits of JSONB over the textual JSON solution are listed, for example, in the "PostgreSQL JSON" tutorial by NEON.
For our experiments we insert into table T1 the following contents:

```
-- Row with a simple JSON document without duplicate values
INSERT INTO T1 (K, J) VALUES
(1, '{ "mem1":123, "mem2":"123", "mem3":true, "mem4":null,
   "mem5": [ 123, "123", true, null, [1, 2], {} ],
   "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
}');

-- Row with JSON document with duplicate keys and elements
INSERT INTO T1 (K, J) VALUES
(2, '{ "mem1": 123,
       "mem2": "123",
       "mem3": true,
       "mem4": null,
       "mem5": [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ],
       "mem6": { "m61":1, "m62":"string", "m63": true, "m64": null, "m65":[2, 3], "m66":{} },
       "mem7": 123,
       "mem8": "123"}');
COMMIT;
```

Note: PostgreSQL requires JSON key names to be enclosed in double quotes, but numeric values, literals, as well as array and object values are not quoted.

Reporting whole document contents by pSQL “SELECT * FROM ..” generates an archaic UNIX pager report view for dump terminals from which view we can get rid only by pressing the key of letter “q” – a shocking case if you don’t remember the solution. Use of the pager view in pSQL sessions can be turned off by command

```
testdb=> \pset pager off
Pager usage is off.
```

A more compact formatting can be obtained by the “composite form” provided by the PostgreSQL JSON function `jsonb_each()` as follows:

```
testdb=# (SELECT (jsonb_each(J)).* FROM T1 WHERE K=1);
...
 key  |                                    value
------+-------------------------------------------------------------------------------
 mem1 | 123
 mem2 | "123"
 mem3 | true
 mem4 | null
 mem5 | [123, "123", true, null, [1, 2], {}]
 mem6 | {"m61": 1, "m62": "string", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}
(6 rows)
(EOD)
```

As default, also this report of the “composite form” would use the pager view.

For shorter reporting, contents of a single member can be generated by the extract operator “->”, for example “mem1” as follows

```
testdb=> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
 k | mem1
---+------
 1 | 123
(1 row)
```

PostgreSQL JSON functions include also “pretty print” solution, as we demonstrate below on member “mem5”:

```
testdb=> SELECT jsonb_pretty(J->'mem5') AS mem5
```

```
FROM T1 WHERE K=1;
     mem5
---------------
 [             +
     123,      +
     "123",+
     true,     +
     null,     +
     [         +
         1,    +
         2     +
     ],        +
     {         +
     }         +
 ]
(1 row)

testdb=*>
```

Accessing JSON objects on top level in the path expression

Case 1: Adding a new member on top level

For this pattern pSQL/JSON has ready functionality

```
UPDATE T1
SET J = jsonb_set(J, '{mem7}', '"new"')
WHERE K = 1;

testdb=> UPDATE T1
SET J = jsonb_set(J, '{mem7}', '"new"')
WHERE K = 1;
UPDATE 1
testdb=> SELECT K, J -> 'mem7' AS mem7 FROM T1 WHERE K = 1;
 k | mem7
---+-------
 1 | "new"
(1 row)
```

Case 2.1: Updating value of an existing member found by key

For this pattern pSQL/JSON has ready functionality

```
UPDATE T1
SET J = jsonb_set(J, '{mem1}', '124')
WHERE K = 1;
--
testdb=*> UPDATE T1
SET J = jsonb_set(J, '{mem1}', '124')
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
 k | mem1
---+------
 1 | 124
(1 row)

testdb=*> rollback;
ROLLBACK
testdb=> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
 k | mem1
---+------
 1 | 123
(1 row)
```

Case 2.2: Updating value of all existing members found by given value

The JSON structure does not require member values to be unique, so it is possible that multiple members have accidently the same value.
But let's start with the simple case assuming that there are no duplicate values.

Searching the key value of the member having the given value, for example '123' can be done as follows

```
SELECT key FROM T1, json_each(J) WHERE value = '123';
```

and passing the key to following update

```
UPDATE T1
SET J = jsonb_set(J, '{key}', '124')
WHERE K = 1;
```

.. might look something like following

```
testdb=*> UPDATE T1 SET J = jsonb_set(J,
  '{(SELECT key::jsonb FROM T1, json_each(T1.J::jsonb)
    WHERE value::jsonb = 123)}', '124', true)
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
 k | mem1
---+------
 1 | 123
(1 row)

testdb=*> rollback;
ROLLBACK
```

So, the update has failed, even if PostgreSQL seems to accept the statement. The embedded SELECT statement in the quoted expression is not a legal jsonb path expression and just gets ignored.
The problem can be solved by forcing the SELECT statement string to be evaluated as an object by concat function as follows

```
testdb=> SELECT concat('{',(SELECT key
                            FROM T1, jsonb_each(T1.J)
                            WHERE K=1 AND value = '123'),'}');
 concat
--------
 {mem1}
(1 row)
```

and applying this in array for the path expression of the jsonb_set by the concat function

```
testdb=> UPDATE T1
SET J = jsonb_set(J,
                  concat('{',(SELECT key
                              FROM T1, jsonb_each(T1.J)
                              WHERE K=1 AND value = '123'),'}')::text[],
                  '124'::jsonb,
                  true)
WHERE K=1;
UPDATE 1
testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=1) as dummy
WHERE value IN ('123','124');
 key  | value
```

```
------+-------
 mem1 | 124
(1 row)

testdb=*> rollback;
ROLLBACK
```

So, this works in case of a unique value.
Next, we test the case of duplicate values using the slightly modified version of our test document of value 2 for the key K where the JSON column has multiple members having the same value 123. Note that in JSON the integer value 123 is different than the string value "123". Following query refreshes now the contents for us

```
(SELECT (jsonb_each(J)).* FROM T1 WHERE K=2);

 key  |                                      value
------+-------------------------------------------------------------------------------
 mem1 | 123
 mem2 | "string"
 mem3 | true
 mem4 | null
 mem5 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
 mem6 | {"m61": 1, "m62": "string", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}
 mem7 | 123
 mem8 | "123"
(8 rows)
```

Applying now the same UPDATE statement raises ERROR

```
testdb=*> UPDATE T1
SET J = jsonb_set(J,
                  concat('{',(SELECT key
                              FROM T1, jsonb_each(T1.J)
                              WHERE K=2 AND value = '123'),'}')::text[],
                  '124'::jsonb,
                  true)
WHERE K=2;
ERROR:  more than one row returned by a subquery used as an expression

testdb=!> rollback;
ROLLBACK
```

The problem is that the embedded SELECT now returns two tuples, not one, so the concat function fails since it is no longer concatenating a single string. There is no way to have `jsonb_set` update more than one value at a time.

The best alternative is to update one key at a time, using a loop in a PL/pgSQL function in which we pass data via local variables between SQL statements, as shown below.

```
testdb=> CREATE OR REPLACE FUNCTION
   update_values_in_T1J(doc_no integer,
                        oldvalue jsonb,
                        newvalue jsonb,
                        countlimit integer default 2147483647
                       ) RETURNS void AS $$
    DECLARE akey text;
    BEGIN
    FOR akey IN
       SELECT * FROM
         (SELECT key from
             (SELECT (jsonb_each(J)).* FROM T1 WHERE K = doc_no) as dummy1
          WHERE value = oldvalue
             LIMIT countlimit) AS dummy1
       ORDER BY key ASC
       LOOP
          UPDATE T1
```

```
          SET J = jsonb_set((SELECT J FROM T1 WHERE K=doc_no),
                            concat('{',akey,'}')::text[],
                            newvalue)
          WHERE K=doc_no;
       END LOOP;
   RETURN;
   END;
   $$ LANGUAGE plpgsql;
CREATE FUNCTION
testdb=*> COMMIT;
COMMIT
```

By the optional parameter countlimit we pass value to LIMIT on keys selected by the SELECT scanning members of the given oldvalue, to be included in the IN list of member keys to be passed then to the LOOP of UPDATE statements.

Applying this to the case of no duplicates, in the row of K=1:

```
testdb=> SELECT update_values_in_T1J (1, '123', '456');
 update_values_in_t1j
----------------------

(1 row)

testdb=*> (SELECT (jsonb_each(J)).* FROM T1 WHERE K=1);

 key  |                                      value
------+---------------------------------------------------------------------------------
 mem1 | 456
 mem2 | "string"
 mem3 | true
 mem4 | null
 mem5 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
 mem6 | {"m61": 1, "m62": "string", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}
(6 rows)testdb=*> (SELECT (jsonb_each(J)).* FROM T1 WHERE K=2);
```

Applying this to the case of multiple duplicates, in the row of K=2:

```
testdb=*> SELECT update_values_in_T1J (2, '123', '456');
 update_values_in_t1j
----------------------

(1 row)

testdb=*> (SELECT (jsonb_each(J)).* FROM T1 WHERE K=2);
 key  |                                      value
------+---------------------------------------------------------------------------------
 mem1 | 456
 mem2 | "string"
 mem3 | true
 mem4 | null
 mem5 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
 mem6 | {"m61": 1, "m62": "string", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}
 mem7 | 456
 mem8 | "123"
(8 rows)
```

Note finally that this function works for any values, not just integers.

```
testdb=*> SELECT update_values_in_T1J (2, '123','[1,2,3]');
 update_values_in_t1j
----------------------

(1 row)

testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
```

```
      FROM T1
      WHERE K=2) as dummy
WHERE value IN ('123','124','[1,2,3]');
 key  |   value
------+-----------
 mem1 | [1, 2, 3]
 mem7 | 124
(2 rows)

testdb=*> SELECT update_values_in_T1J (2, '[1,2,3]','123');
 update_values_in_t1j
----------------------

(1 row)

testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=2) as dummy
WHERE value IN ('123','124','[1,2,3]');
 key  | value
------+-------
 mem1 | 123
 mem7 | 124
(2 rows)

testdb=*> rollback;
ROLLBACK
testdb=>
```

So, our function above provides a proper pattern for updating value of existing members having the given value.

Case 3.1: Deleting an existing member found by key

For this pattern pSQL/JSON has ready functionality

```
testdb=> UPDATE T1 SET J = J - 'mem1' WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
 k | mem1
---+------
 1 |
(1 row)

testdb=*> ROLLBACK;
ROLLBACK
testdb=> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
 k | mem1
---+------
 1 | 123
(1 row)

testdb=*> ROLLBACK;
ROLLBACK
```

Case 3.2: Deleting all existing members found by given value

Example deleting member having value 123 on JSON without duplicates, K = 1:

```
testdb=> UPDATE T1 SET J = J -
  (SELECT key FROM T1, jsonb_each(J::jsonb)
    WHERE value = '123')
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem1' AS mem1 FROM T1 WHERE K = 1;
```

```
 k | mem1
---+------
 1 |
(1 row)
```

Trying to apply this to members of duplicate value, K= 2:

```
Testdb*> UPDATE T1 SET J = J -
  (SELECT key FROM T1, jsonb_each(J::jsonb)
    WHERE value = '123')
WHERE K = 2;
ERROR:  more than one row returned by a subquery used as an expression
testdb=!> ROLLBACK;
ROLLBACK
testdb=>
```

To solve the issue, we modify a new version of the PL/pgSQL function “update_values_in_T1J” as follows

```
CREATE OR REPLACE FUNCTION
   remove_members_in_T1J (doc_no integer,
                          givenvalue jsonb,
                          countlimit integer default 2147483647
                         ) RETURNS void AS $$
    DECLARE akey text;
    BEGIN
    FOR akey IN
       SELECT * FROM
         (SELECT key from
             (SELECT (jsonb_each(J)).* FROM T1 WHERE K = doc_no) as dummy1
          WHERE value = givenvalue
             LIMIT countlimit) AS dummy1
       ORDER BY key ASC
       LOOP
          UPDATE T1
          SET J = J - akey
          WHERE K = doc_no;
   END LOOP;
   RETURN;
   END;
   $$ LANGUAGE plpgsql;
COMMIT;
```

Experimenting with the JSON document without duplicates, K= 1

```
testdb=> SELECT remove_members_in_T1J (1, '123');
 remove_members_in_t1j
-----------------------

(1 row)

testdb=*> SELECT (jsonb_each(J)).* FROM T1 WHERE K = 1;

 key  |                                        value
------+-------------------------------------------------------------------------------------
 mem2 | "123"
 mem3 | true
 mem4 | null
 mem5 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
 mem6 | {"m61": 1, "m62": "string", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}
(5 rows)
```

Experimenting with the JSON document of duplicates, K= 2

```
testdb=*> SELECT remove_members_in_T1J (2, '123');
 remove_members_in_t1j
-----------------------
```

```
(1 row)

testdb=*> SELECT (jsonb_each(J)).* FROM T1 WHERE K = 2;

 key  |                                        value
------+-----------------------------------------------------------------------------------
 mem2 | "123"
 mem3 | true
 mem4 | null
 mem5 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
 mem6 | {"m61": 1, "m62": "string", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}
 mem8 | "123"
(6 rows)
testdb=*> ROLLBACK;
ROLLBACK
testdb=>
```

So, this works on both cases, and is a working pattern for the case of "Deleting all existing members found by given value".

Accessing JSON arrays

Case 4: Adding a new element

For this pattern pSQL/JSON has ready functionality

```
testdb=> UPDATE T1
SET J = jsonb_set(J, '{mem5}', J->'mem5' || '124')
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |                      mem5
---+-------------------------------------------------
 1 | [123, "123", true, null, [1, 2], {}, 124]
(1 row)

testdb=*> ROLLBACK;
ROLLBACK
```

Case 5.1: Updating value of an existing element found by position

Using the simple path expression `{mem5,0}` we access the first element in the array of member mem5

```
testdb=> UPDATE T1
SET J = jsonb_set(J, '{mem5,0}', '124')
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |                   mem5
---+-----------------------------------------
 1 | [124, "123", true, null, [1, 2], {}]
(1 row)

testdb=*> ROLLBACK;
ROLLBACK
testdb=>
```

Case 5.2: Updating value of all existing elements found by value

The JSON structure does not require element values in an array to be unique, so it is possible to have multiple duplicate values in the same array. However, let's start with the simple case assuming that there are no duplicate values.

Trying to apply similar solution as Case 2.2 "Updating value of an existing members found by given value" first applying the following "Updating value found by position" for the first element “123” of member mem5 to the new value “124”:

```
testdb=> UPDATE T1
SET J = jsonb_set(J,
                  concat('{mem5,',
                         (SELECT ordinality::integer -1 AS index
                          FROM T1, jsonb_array_elements(J->'mem5')
                          WITH ORDINALITY
                          WHERE value = '123'), ' }')::text[],
        '124')
WHERE K = 1;
ERROR:  malformed array literal: "{mem5, }"
DETAIL:  Unexpected "}" character.
testdb=!> rollback;
ROLLBACK
```

Unfortunately, this has the same drawback as the one for Case 2.2; namely, it does not work when there is more than one value to be updated.
It furthermore fails when there are no values to update.

```
testdb=> UPDATE T1
SET J = jsonb_set(J,
                  concat('{mem5,',
                         (SELECT ordinality::integer -1 AS index
                          FROM T1, jsonb_array_elements(J->'mem5')
                          WITH ORDINALITY
                          WHERE value = '125'), ' }')::text[],
        '124')
WHERE K = 1;
ERROR:  malformed array literal: "{mem5, }"
DETAIL:  Unexpected "}" character.
testdb=!> rollback;
ROLLBACK
```

Two solutions which avoid these limitations are presented. The first converts the JSONB array to an ordinary PostgreSQL array, performs the deletion operation there, and then converts back to a JSONB array.

```
testdb=> UPDATE T1
   SET J = jsonb_set(J,
                     '{mem5}',
                     (SELECT array_to_json(
                        array_replace(
                          array(SELECT jsonb_array_elements((J->'mem5'))),
                             '123'::jsonb,
                             '124'::jsonb))
                      FROM T1 WHERE K=1)::jsonb
                    )
WHERE K=1;
UPDATE 1
testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=1) as dummy
WHERE key='mem5';
 key  |                      value
------+-------------------------------------------------
 mem5 | [124, "123", true, null, [1, 2], {}]
(1 row)
```

So, this works, as well as applied to case of multiple duplicates in K=2

```
testdb=*> UPDATE T1
SET J = jsonb_set(J,
                     '{mem5}',
```

```
                    (SELECT array_to_json(
                       array_replace(
                         array(SELECT jsonb_array_elements((J->'mem5'))),
                            '123'::jsonb,
                            '124'::jsonb))
                     FROM T1 WHERE K=2)::jsonb
                   )
WHERE K=2;
UPDATE 1
testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=2) as dummy
WHERE key='mem5';
 key  |                               value
------+-------------------------------------------------------------------
 mem5 | [124, "123", "string", true, 124, 124, 124, null, [1, 2], {}, 124]
(1 row)

testdb=*> ROLLBACK;
ROLLBACK
```

The second solution employs a PL/pgSQL function, experimented with the duplicate elements in document of K=2 as follows:

```
testdb=> CREATE OR REPLACE FUNCTION
     replace_in_array_of_T1J (doc_no integer,
                              jkey text,
                              jexpr jsonb,
                              newvalue jsonb,
                              countlimit integer default 2147483647)
                                   RETURNS void AS $$
    DECLARE eposition integer;
    BEGIN
    FOR eposition IN
       SELECT ordinality::integer
           FROM T1, jsonb_array_elements(J->jkey) WITH ORDINALITY
           WHERE K=doc_no AND value = jexpr
                 LIMIT countlimit
       LOOP
           UPDATE T1
           SET J = jsonb_set(J, concat('{',jkey,',',eposition-1,'}')::text[],
                             newvalue)
           WHERE K = doc_no;
           RAISE NOTICE 'Entry in position % of % replaced.', eposition, jkey;
       END LOOP;
   RETURN;
   END;
   $$ LANGUAGE plpgsql;
testdb=*> COMMIT;
COMMIT

testdb=> SELECT replace_in_array_of_T1J (1, 'mem5','123','124');
NOTICE:  Entry in position 1 of mem5 replaced.
 replace_in_array_of_t1j
-------------------------

(1 row)

testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=1) as dummy
WHERE key='mem5';
 key  |                value
------+---------------------------------------
 mem5 | [124, "123", true, null, [1, 2], {}]
(1 row)

testdb=*>
```

```
SELECT replace_in_array_of_T1J (2, 'mem5','123','124');

SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=2) as dummy
WHERE key='mem5';

SELECT replace_in_array_of_T1J (2,'mem5','123','124',1);

SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
testdb=*> SELECT replace_in_array_of_T1J (2, 'mem5','123','124');
NOTICE:  Entry in position 1 of mem5 replaced.
NOTICE:  Entry in position 5 of mem5 replaced.
NOTICE:  Entry in position 11 of mem5 replaced.
 replace_in_array_of_t1j
-------------------------

(1 row)

testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=2) as dummy
WHERE key='mem5';
 key  |                               value
------+-------------------------------------------------------------------------
 mem5 | [124, "123", "string", true, 124, 124, 124, null, [1, 2], {}, 124]
(1 row)

testdb=*>

testdb=!> rollback;
ROLLBACK
```

The optional argument for `countlimit` limits the number of values which are replaced:

```
testdb=> SELECT replace_in_array_of_T1J (2,'mem5','123','124',1);
NOTICE:  Entry in position 1 of mem5 replaced.
 replace_in_array_of_t1j
-------------------------

(1 row)

testdb=*> SELECT *
FROM (SELECT (jsonb_each(J)).*
      FROM T1
      WHERE K=1) as dummy
WHERE key='mem5';
 key  |                      value
------+-----------------------------------------------------
 mem5 | [124, "string", true, 123, null, [1, 2], {}, 123]
(1 row)

testdb=*>
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
 k |                             mem5
---+-------------------------------------------------------------
 1 | [123, "123", true, 123, 124, 124, null, [1, 2], {}, 123]
(1 row)

testdb=*> select remove_from_mem5 (2, '123');
NOTICE:  Entry in position 1 of mem5 removed.
 remove_from_mem5
------------------
```

```
(1 row)

testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
 k |                          mem5
---+-------------------------------------------------------
 1 | ["123", true, 123, 124, 124, null, [1, 2], {}, 123]
(1 row)

testdb=*>
testdb=!> ROLLBACK;
ROLLBACK
```

Case 6.1: Deleting an existing element found by position

For this pattern pSQL/JSON has ready functionality with which experiment by deleting the first element (note: JSON arrays are 0-indexed) from the array value of member mem5:

```
testdb=> UPDATE T1
SET J = jsonb_set(J, '{mem5}',
    (J->'mem5')::jsonb - 0 )
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |                mem5
---+-------------------------------------
 1 | ["123", true, null, [1, 2], {}]
(1 row)

testdb=*> rollback;
ROLLBACK
testdb=>
```

Case 6.2: Deleting all existing elements found by given value

Let's start experimenting with the case of no duplicates, K=1
This is a tricky issue, for example if we try to delete the integer element 123

```
testdb=> UPDATE T1
SET J = jsonb_set(J, '{mem5}',
    (J->'mem5')::jsonb - '123')
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |                  mem5
---+----------------------------------------
 1 | [123, "123", true, null, [1, 2], {}]
(1 row)

testdb=*> rollback;
ROLLBACK
testdb=>
```

Even if we did not get error message, this delete failed since value '123' in this context means a string value while literal 123 is assumed to present an index value.
Next, we try filter solution to find the array index of the element to be deleted:

```
testdb=*> SELECT ordinality -1 AS index
FROM T1, jsonb_array_elements(J->'mem5') WITH ORDINALITY
WHERE value = '123';
 index
-------
     0
(1 row)
```

and embedding this to our UPDATE

```
testdb=*> UPDATE T1
SET J = jsonb_set(J, '{mem5}',
    (J->'mem5')::jsonb  -
    (SELECT ordinality -1 AS index
       FROM T1, jsonb_array_elements(J->'mem5') WITH ORDINALITY
       WHERE value = '123'
    )::integer )
WHERE K = 1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |                mem5
---+-------------------------------------
 1 | ["123", true, null, [1, 2], {}]
(1 row)

testdb=*> ROLLBACK;
ROLLBACK
testdb=>
```

Voila, it worked! However, following solution is more general translating a JSONB array to a PostgreSQL array and back

```
testdb=> UPDATE T1
   SET J = jsonb_set(J,
                     '{mem5}',
                     (SELECT array_to_json(
                        array_remove(
                          array(SELECT jsonb_array_elements((J->'mem5'))),
                            '123'::jsonb))
                      FROM T1 WHERE K=1)::jsonb
                    )
WHERE K=1;
UPDATE 1
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |               mem5
---+---------------------------------
 1 | ["123", true, null, [1, 2], {}]
(1 row)

testdb=*> rollback;
ROLLBACK
testdb=>
```

A proper pattern needs to work for documents with duplicate elements.
Here is one which is similar to the PL/pgsql solution for Case 5.2.

```
testdb=> CREATE OR REPLACE FUNCTION
    remove_from_array_of_T1J (doc_no integer,
                              jkey text,
                              jexpr jsonb,
                              countlimit integer default 2147483647)
                              RETURNS void AS $$
    DECLARE eposition integer;
    BEGIN
    FOR eposition IN
       SELECT ordinality::integer
           FROM T1, jsonb_array_elements(J->jkey) WITH ORDINALITY
           WHERE K=doc_no AND value = jexpr
       ORDER BY ordinality DESC
         LIMIT countlimit
       LOOP
           UPDATE T1
           SET J = jsonb_set(J,
```

```
                            concat('{',jkey,'}')::text[],
                           (J->jkey)::jsonb - (eposition-1))
          WHERE K = doc_no;
          RAISE NOTICE 'Entry in position % of % removed.', eposition, jkey;
       END LOOP;
   RETURN;
   END;
   $$ LANGUAGE plpgsql;
CREATE FUNCTION
testdb=*> COMMIT;
COMMIT
testdb=>

testdb=> SELECT remove_from_array_of_T1J (1, 'mem5','123');
NOTICE:  Entry in position 1 of mem5 removed.
 remove_from_array_of_t1j
--------------------------

(1 row)

testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 1;
 k |              mem5
---+---------------------------------
 1 | ["123", true, null, [1, 2], {}]
(1 row)

testdb=*> SELECT remove_from_array_of_T1J (2, 'mem5','123');
NOTICE:  Entry in position 11 of mem5 removed.
NOTICE:  Entry in position 5 of mem5 removed.
NOTICE:  Entry in position 1 of mem5 removed.
 remove_from_array_of_t1j
--------------------------

(1 row)

testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
 k |                         mem5
---+-------------------------------------------------------
 2 | ["123", "string", true, 124, 124, null, [1, 2], {}]
(1 row)

testdb=*>
```

As for the function `replace_in_array_of_T1J` for Case 5.2, this function has an optional argument which limits the number of replacements. Let's first see the original contents and then apply the function

```
testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
 k |                                mem5
---+-------------------------------------------------------------------
 2 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
(1 row)

testdb=> SELECT remove_from_array_of_T1J (2,'mem5','123',1);
NOTICE:  Entry in position 11 of mem5 removed.
 remove_from_array_of_t1j
--------------------------

(1 row)

testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
 k |                              mem5
---+-----------------------------------------------------------------
 2 | [123, "123", "string", true, 123, 124, 124, null, [1, 2], {}]
(1 row)

testdb=*> rollback;
```

```
ROLLBACK
```

Note: By the ORDER BY .. DESC the function removed the last matching element and changing the order to ASC the function would remove the first matching element.

```
testdb=*> SELECT remove_from_array_of_T1J (2, 'mem5','123', 1);
NOTICE:  Entry in position 1 of mem5 removed.
 remove_from_array_of_t1j
--------------------------

(1 row)

testdb=*> SELECT K, J -> 'mem5' AS mem5 FROM T1 WHERE K = 2;
 k |                              mem5
---+-----------------------------------------------------------------
 2 | ["123", "string", true, 123, 124, 124, null, [1, 2], {}, 123]
(1 row)

testdb=*> rollback;
ROLLBACK
```

JSON manipulation experiments using SQL/JSON of MySQL/MariaDB

A brief history

Before entering the experiments with MySQL/MariaDB JSON, we'd like to look back in history and the impact of MySQL on our DBTechLab tutorials for DBTechNet workshops. We do owe a lot to the works of Open Source legends Linus Torvalds, Monty Widenius and Heikki Tuuri, who started their careers on Helsinki area in the nineties. The Linux movement started by Linus, and Oracle's VirtualBox platform have enabled the cross university "laboratory" platform for our workshops and shared materials. In 99 Monty appeared in our workshop at Haaga-Helia on SQL-99 by Ocelot[4] praising his MySQL, a real "No SQL" database of hobbyists at that time, on performance due to missing transaction processing facilities, while he was in fact interested in hiring the Ocelot people. Soon MySQL was switched into a SQL engine and its database engine was replaced by InnoDB engine of Heikki Tuuri to use transactions. After Oracle acquired MySQL and InnoDB, Monty's new team has continued development of the pure Open Source version of MySQL code as MariaDB. It is great that both of these versions now continue on top of the development of database technologies, including SQL/JSON.

Comparing JSON implementations of MySQL and MariaDB

A difference between MySQL and MariaDB is that while MySQL implements native JSON data type defined in RFC 7159 (Petkovic, 2020). The JSON data type of MariaDB is just an alias name for LONGTEXT COLLATE utf8mb4_bin. However, the JSON alias includes automatically JSON_VALID function as its CHECK constraint.

Following Table 1.1 of JSON functions has been built based on documentation on web sites of both products. Marking by "Y" means "listed", "y" found but not listed, while blank means that the function has not found in the documentation, so the list is not accurate and the function may be implemented already but not yet documented, or may be implemented in future. This indicates the fast development on the technology and versions.

Table 1.1 List of JSON functions in MySQL and/or MariaDB

[4] Peter Gulutzan & Trudy Pelzer, the authors of the book "SQL-99 Complete, Really"

JSON functions of	MySQL	MariaDB
JSON_ARRAY	Y	Y
JSON_ARRAY_AGG	Y	JSON_ARRAYAGG
JSON_ARRAY_APPEND	Y	Y
JSON_ARRAY_INSERT	Y	Y
JSON_ARRAY_INTERSECT		Y
JSON_COMPACT		Y
JSON_CONTAINS	Y	Y
JSON_CONTAINS_PATH	Y	Y
JSON_DEPTH		Y
JSON_DETAILED		Y
JSON_EQUALS		Y
JSON_EXISTS		Y
JSON_EXTRACT	Y	Y
JSON_INSERT	Y	Y
JSON_KEYS	Y	Y
JSON_KEY_VALUE		Y
JSON_LENGTH		Y
JSON_LOOSE		Y
JSON_MERGE		Y
JSON_MERGE_PATCH		Y
JSON_MERGE_PRESERVE		Y
JSON_NORMALIZE		Y
JSON_OBJECT	Y	Y
JSON_OBJECT_AGG	Y	JSON_OBJECTAGG
JSON_OBJECT_FILTER_KEYS		Y
JSON_OBJECT_TO_ARRAY		Y
JSON_OVERLAPS	Y	Y
JSON_PRETTY		Y
JSON_QUERY		Y
JSON_QUOTE		Y
JSON_REMOVE	y	y
JSON_REPLACE	Y	y
JSON_SEARCH	Y	Y
JSON_SET		Y
JSON_TABLE	Y	y
JSON_TYPE	Y	Y
JSON_UNQUOTE	Y	Y
JSON_VALID	Y	Y
JSON_VALUE	Y	Y
value MEMBER OF	Y	

The SQL/JSON experiments below are run using MariaDB 11.8.2 server on Windows 11 desktop.

Setting up the experiment

```
MariaDB [(none)]> use testdb
Database changed
MariaDB [testdb]>

USE Testdb;

DROP TABLE T1;

CREATE TABLE T1(
K  INT NOT NULL PRIMARY KEY,
J  JSON);

-- Inserting our test documents
-- Note: in T-SQL/JSON key names need to be enclosed in double quotes!
START TRANSACTION;
DELETE FROM T1;
-- our basic document without duplicates
INSERT INTO T1 (K, J) VALUES
(1,  '{
       "mem1":123,
       "mem2":"123",
       "mem3":true,
       "mem4":null,
       "mem5": [ 123, "123", true, null, [1, 2], {} ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
      }');
-- document with duplicate keys and elements
INSERT INTO T1 (K, J) VALUES
(2,  '{
       "mem1": 123,
       "mem2": "123",
       "mem3": true,
       "mem4": null,
       "mem5": [ 123, "123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} },
       "mem7": 123,
       "mem8": "123"
      }');
COMMIT;
-- Duplicate members?
INSERT INTO T1 (K, J) VALUES
(3, '{ "mem1":123, "mem1":124 }');

SELECT LEFT(K, 4) AS K, LEFT(JSON_VALUE(J, '$.mem1'), 4) AS mem1
FROM T1 WHERE K=3;

MariaDB [testdb]> -- Duplicate members?
MariaDB [testdb]> INSERT INTO T1 (K, J) VALUES
    -> (3, '{ "mem1":123, "mem1":124 }');
Query OK, 1 row affected (0.001 sec)

MariaDB [testdb]>
MariaDB [testdb]> SELECT LEFT(K, 4) AS K, LEFT(JSON_VALUE(J, '$.mem1'), 4) AS mem1
    -> FROM T1 WHERE K=3;
+------+------+
| K    | mem1 |
+------+------+
| 3    | 123  |
+------+------+
1 row in set (0.000 sec)
```

Some querying models

Beside the SQL/JSON reporting functions, simple SELECT

```
MariaDB [testdb]> SELECT J FROM T1 WHERE K=1;
+-----------------------------------------------------------------------------------------------
```

```
| J
+-----------------------------------------------------------------------------------------------
| {
       "mem1":123,
       "mem2":"123",
       "mem3":true,
       "mem4":null,
       "mem5": [ 123, "123", true, null, [1, 2], {} ],
       "mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }
      } |
+-----------------------------------------------------------------------------------------------
1 row in set (0.001 sec)
```

or pretty-print of JSON documents

```
MariaDB [testdb]> SELECT JSON_PRETTY(J) FROM T1 WHERE K=1;
+------------------------------------------------------------
JSON_PRETTY(J)
+------------------------------------------------------------
| {
    "mem1": 123,
    "mem2": "123",
    "mem3": true,
    "mem4": "new value",
    "mem5":
    [
        123,
        "123",
        true,
        null,
        [
            1,
            2
        ],
        {
        }
    ],
    "mem6":
    {
        "m61": 1,
        "m62": "123",
        "m63": true,
        "m64": null,
        "m65":
        [
            2,
            3
        ],
        "m66":
        {
        }
    }
} |
+------------------------------------------------------------
1 row in set (0.001 sec)
```

and following functions are available

```
JSON_KEYES()

MariaDB [testdb]> SELECT JSON_KEYS(J) FROM T1 WHERE K= 1;
+-------------------------------------------------+
| JSON_KEYS(J)                                    |
+-------------------------------------------------+
| ["mem1", "mem2", "mem3", "mem4", "mem5", "mem6"] |
+-------------------------------------------------+
1 row in set (0.001 sec)
```

```
JSON_EXTRACT()

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem5') FROM T1 WHERE K=1;
+--------------------------------------+
| JSON_EXTRACT(J, '$.mem5')            |
+--------------------------------------+
| [123, "123", true, null, [1, 2], {}] |
+--------------------------------------+
1 row in set (0.000 sec)
```

Accessing JSON objects

Case 1: Adding a new member on top level

```
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_SET(J, '$.mem7', 'new value')
    -> WHERE K = 1;
Query OK, 1 row affected (0.001 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> --
MariaDB [testdb]> SELECT LEFT(JSON_VALUE(J, '$.mem7'), 10) AS mem7
    -> FROM T1 WHERE K = 1;
+-----------+
| mem7      |
+-----------+
| new value |
+-----------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.001 sec)
```

Case 2.1: Updating value of an existing member found by key

```
a)
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_SET(J, '$.mem4', 'new value')
    -> WHERE K = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem4') AS mem4 FROM T1 WHERE K = 1;
+-----------+
| mem4      |
+-----------+
| new value |
+-----------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.014 sec)
```

b) as synonym of JSON_SET() MySQL and MariaDB have JSON_REPLACE()

```
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_REPLACE(J, '$.mem4', 'new value')
```

```
    -> WHERE K = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem4') AS mem4 FROM T1 WHERE K = 1;
+-------------+
| mem4        |
+-------------+
| "new value" |
+-------------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.001 sec)
```

Case 2.2: Updating value of all existing members found by given value

This “ALL operation” has proven to be difficult. As temporary solution we built the logic of the pattern into following MariaDB SQL procedure tailored just for record K=1 in our table T1. Also writing a more general-purpose procedure proved to be challenging and the generalization has now been left out scope of our paper.

```
drop procedure UpdMembersOfGivenValue;
DELIMITER //
CREATE PROCEDURE UpdMembersOfGivenValue(
                 IN json_data   JSON,
                 IN given_value VARCHAR(255),
                 IN new_value   VARCHAR(255) )
BEGIN
    DECLARE key_name VARCHAR(255);
    DECLARE value VARCHAR(255);
    DECLARE path VARCHAR(255);
    DECLARE idx INT DEFAULT 0;
    DECLARE total_keys INT;
    SET @keys = JSON_KEYS(json_data);
    SET total_keys = JSON_LENGTH(@keys);
    WHILE idx < total_keys DO
        SET key_name = JSON_UNQUOTE(JSON_EXTRACT(@keys, CONCAT('$[', idx, ']')));
        SET path = CONCAT('$.', key_name );
        SET value = JSON_VALUE(json_data, path);
        CASE value
           WHEN given_value THEN
              UPDATE T1
                SET J = JSON_SET(J, path, new_value)
                WHERE K = 1;
           ELSE BEGIN END;
        END CASE;
        SET idx = idx + 1;
    END WHILE;
END;
//
DELIMITER ;
```

Test run as follows

```
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> CALL UpdMembersOfGivenValue(1, (SELECT J FROM T1 WHERE K=1), 123, 127);

+--------+-------------+-------+
| path   | given_value | value |
+--------+-------------+-------+
| $.mem1 | 123         | 123   |
+--------+-------------+-------+
1 row in set (0.002 sec)
```

```
+--------+-------------+-------+
| path   | given_value | value |
+--------+-------------+-------+
| $.mem2 | 123         | 123   |
+--------+-------------+-------+
1 row in set (0.004 sec)

Query OK, 2 rows affected (0.006 sec)

MariaDB [testdb]> SELECT * FROM T1 WHERE K=1;
+---+-----------------------------------------------------------------------------------------------
| K | J
+---+-----------------------------------------------------------------------------------------------
| 1 | {"mem1": "127", "mem2": "127", "mem3": true, "mem4": null, "mem5": [123, "123", true,
null, [1, 2], {}], "mem6": {"m61": 1, "m62": "123", "m63": true, "m64": null, "m65": [2, 3],
"m66": {}}} |
+---+-----------------------------------------------------------------------------------------------
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.000 sec)
```

Case 3.1: Deleting an existing member found by key

```
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1 SET J = JSON_REMOVE(J, '$.mem1')
    -> WHERE K=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_KEYS(J) FROM T1 WHERE K= 1;
+-----------------------------------------+
| JSON_KEYS(J)                            |
+-----------------------------------------+
| ["mem2", "mem3", "mem4", "mem5", "mem6"] |
+-----------------------------------------+
1 row in set (0.000 sec)

MariaDB [testdb]> SELECT LEFT(JSON_VALUE(J, '$.mem1'), 10) AS mem1
    -> FROM T1 WHERE K=1;
+------+
| mem1 |
+------+
| NULL |
+------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.013 sec)
```

Case 3.2: Deleting all existing members found by given value

Solved by modification from Case 2.2 solution

```
drop procedure DelMembersOfGivenValue;
DELIMITER //
CREATE PROCEDURE DelMembersOfGivenValue(
                  IN json_data   JSON,
                  IN given_value VARCHAR(255),
                  IN new_value   VARCHAR(255) )
BEGIN
    DECLARE key_name VARCHAR(255);
```

```
    DECLARE value VARCHAR(255);
    DECLARE path VARCHAR(255);
    DECLARE idx INT DEFAULT 0;
    DECLARE total_keys INT;
    SET @keys = JSON_KEYS(json_data);
    SET total_keys = JSON_LENGTH(@keys);
    WHILE idx < total_keys DO
        SET key_name = JSON_UNQUOTE(JSON_EXTRACT(@keys, CONCAT('$[', idx, ']')));
        SET path = CONCAT('$.', key_name );
        SET value = JSON_VALUE(json_data, path);
        CASE value
           WHEN given_value THEN
              SELECT path, given_value, value;
              UPDATE T1
                SET J = JSON_REMOVE(J, path)
                WHERE K = 1;
           ELSE BEGIN END;
        END CASE;
        SET idx = idx + 1;
    END WHILE;
END;
//
DELIMITER ;

START TRANSACTION;
CALL DelMembersOfGivenValue((SELECT J FROM T1 WHERE K=1), 123, 127);
SELECT * FROM T1 WHERE K=1;
ROLLBACK;

MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> CALL DelMembersOfGivenValue((SELECT J FROM T1 WHERE K=1), 123, 127);
+--------+-------------+-------+
| path   | given_value | value |
+--------+-------------+-------+
| $.mem1 | 123         | 123   |
+--------+-------------+-------+
1 row in set (0.001 sec)

+--------+-------------+-------+
| path   | given_value | value |
+--------+-------------+-------+
| $.mem2 | 123         | 123   |
+--------+-------------+-------+
1 row in set (0.002 sec)

Query OK, 2 rows affected (0.004 sec)

MariaDB [testdb]> SELECT * FROM T1 WHERE K=1;
+---+-----------------------------------------------------------------------------------------------
| K | J     |
+---+-----------------------------------------------------------------------------------------------
| 1 | {"mem3": true, "mem4": null, "mem5": [123, "123", true, null, [1, 2], {}], "mem6":
{"m61": 1, "m62": "123", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}} |
+---+-----------------------------------------------------------------------------------------------
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.001 sec)
```

Accessing JSON arrays

Case 4: Adding a new element into an array

Petkovic (2020) reports of following MySQL functions for inserting new element into a JSON array

a) to insert new value in given position
`JSON_ARRAY_INSERT(jdoc, '$.member[position]', "value")`

b) to append the new value at the end of the array
`JSON_ARRAY_APPEND(jdoc, '$.member', "newvalue")`

In following we apply these to our MariaDB basic document

```
a)
START TRANSACTION;
UPDATE T1
SET J = JSON_ARRAY_INSERT(J, '$.mem5[6]', 6)
WHERE K = 1;
SELECT JSON_EXTRACT(J, '$.mem5') AS mem5 FROM T1 WHERE K=1;
ROLLBACK;

MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_ARRAY_INSERT(J, '$.mem5[6]', 6)
    -> WHERE K = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem5') AS mem5 FROM T1 WHERE K=1;
+----------------------------------------+
| mem5                                   |
+----------------------------------------+
| [123, "123", true, null, [1, 2], {}, 6] |
+----------------------------------------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.013 sec)

b)
START TRANSACTION;
UPDATE T1
SET J = JSON_ARRAY_APPEND(J, '$.mem5', 5)
WHERE K = 1;
SELECT JSON_EXTRACT(J, '$.mem5') AS mem5 FROM T1 WHERE K=1;
ROLLBACK;

MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_ARRAY_APPEND(J, '$.mem5', 5)
    -> WHERE K = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem5') AS mem5 FROM T1 WHERE K=1;
+----------------------------------------+
| mem5                                   |
+----------------------------------------+
| [123, "123", true, null, [1, 2], {}, 5] |
+----------------------------------------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.001 sec)
```

Case 5.1: Updating value of an existing element found by position

```
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)
```

```
MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_SET(J, '$.mem5[1]', "125")
    -> WHERE K = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem5') AS mem5 FROM T1 WHERE K=1;
+-------------------------------------+
| mem5                                |
+-------------------------------------+
| [123, "125", true, null, [1, 2], {}] |
+-------------------------------------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.000 sec)
```

Case 5.2: Updating value of all existing elements found by value

We have not found direct functions for this pattern, but MariaDB's stored procedure language provides means for implementing the needed logic steps as follows. The code contains temporary indented "select" statements for tracing the steps.

```
drop procedure UpdElementsOfGivenValue;
DELIMITER //
CREATE PROCEDURE UpdElementsOfGivenValue(
                  IN json_data   JSON,
                  IN member_key  VARCHAR(255),
                  IN given_value VARCHAR(255),
                  IN new_value   VARCHAR(255) )
BEGIN
    DECLARE value VARCHAR(255);
    DECLARE path VARCHAR(255);
    DECLARE array VARCHAR(255);
    DECLARE idx INT DEFAULT 0;
    DECLARE total_elems INT;
    SET path = CONCAT('$.', member_key, '[*]' );
      select path;
    SET array = JSON_EXTRACT(json_data, path);
      select array;
    SET total_elems = JSON_LENGTH(array);  -- ?
      select total_elems;
    -- Loop through each key
    WHILE idx < total_elems DO
        SET path = CONCAT('$.', member_key, '[', idx, ']') ;
        SET value = JSON_EXTRACT(json_data, path);
        CASE value
           WHEN given_value THEN
                select path, given_value, value;
              UPDATE T1
                SET J = JSON_SET(J, path, new_value)
                WHERE K = 1;
           ELSE BEGIN END;
        END CASE;
        SET idx = idx + 1;
    END WHILE;
END;
//
DELIMITER ;

START TRANSACTION;
CALL UpdElementsOfGivenValue((SELECT J FROM T1 WHERE K=1), 'mem5', 123, 127);
SELECT * FROM T1 WHERE K=1;
ROLLBACK;
```

Case 6.1: Deleting an existing element found by position

JSON_REMOVE() works also for array elements

```
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T1
    -> SET J = JSON_REMOVE(J, '$.mem5[1]')
    -> WHERE K = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT JSON_EXTRACT(J, '$.mem5') AS mem5 FROM T1 WHERE K=1;
+-------------------------------+
| mem5                          |
+-------------------------------+
| [123, true, null, [1, 2], {}] |
+-------------------------------+
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.014 sec)
```

Case 6.2: Deleting all existing elements found by given value

Modifying new version from the stored procedure of pattern 5.2 the implementation of this pattern was a quite simple task

```
drop procedure DelElementsOfGivenValue;
DELIMITER //
CREATE PROCEDURE DelElementsOfGivenValue(
                  IN json_data   JSON,
                  IN member_key  VARCHAR(255),
                  IN given_value VARCHAR(255),
                  IN new_value   VARCHAR(255) )
BEGIN
    DECLARE value VARCHAR(255);
    DECLARE path VARCHAR(255);
    DECLARE array VARCHAR(255);
    DECLARE idx INT DEFAULT 0;
    DECLARE total_elems INT;
    SET path = CONCAT('$.', member_key, '[*]' );
    SET array = JSON_EXTRACT(json_data, path);
    SET total_elems = JSON_LENGTH(array);  -- ?
    WHILE idx < total_elems DO
        SET path = CONCAT('$.', member_key, '[', idx, ']') ;
        SET value = JSON_EXTRACT(json_data, path);
        CASE value
           WHEN given_value THEN
              UPDATE T1
                SET J = JSON_REMOVE(J, path)
                WHERE K = 1;
           ELSE BEGIN END;
        END CASE;
        SET idx = idx + 1;
    END WHILE;
END;
//
DELIMITER ;

START TRANSACTION;
CALL DelElementsOfGivenValue((SELECT J FROM T1 WHERE K=1), 'mem5', 123, 127);
SELECT * FROM T1 WHERE K=1;
ROLLBACK;
```

Test run

```
MariaDB [testdb]> DELIMITER ;
```

```
MariaDB [testdb]>
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> CALL DelElementsOfGivenValue((SELECT J FROM T1 WHERE K=1), 'mem5', 123,
127);
Query OK, 1 row affected (0.001 sec)

MariaDB [testdb]> SELECT * FROM T1 WHERE K=1;
+---+-------------------------------------------------------------------------------------------
| K | J
+---+-------------------------------------------------------------------------------------------
| 1 | {"mem1": 123, "mem2": "123", "mem3": true, "mem4": null, "mem5": ["123", true, null, [1,
2], {}], "mem6": {"m61": 1, "m62": "123", "m63": true, "m64": null, "m65": [2, 3], "m66": {}}}
|
+---+-------------------------------------------------------------------------------------------
1 row in set (0.000 sec)

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.000 sec)
```

Summary

A difference between MySQL and MariaDB is that while MySQL implements true JSON data type, The JSON data type of MariaDB is just an alias name for LONGTEXT COLLATE utf8mb4_bin. However, the JSON alias includes automatically JSON_VALID function as its CHECK constraint.

On duplicate keys of object members MariaDB documentation says that only the first key-value will be effective, as we tested above.

Filter expressions of SQL/JSON proposal are not implemented in MySQL/MariaDB.

Compared with the JSON UPDATE implementations in other DBMS products, the remove operation of members or elements of Oracle and MySQL/MariaDB are the best implementations.

The list of JSON functions of MySQL and MariaDB provides topics for many study reports!

On UNIQUE KEYS requirement of JSON members

The JSON Data Structure model which we presented in the beginning of this tutorial assumes that member names (i.e. keys) are unique inside every JSON object. According to IETF JSON specification RFC 8259 Dec 2017 “The names within an object SHOULD be unique” and continues “.. When the names within an object are not unique, the behavior of software that receives such an object is un-predictable.” However, the RFC does not deny possibility of non-unique keys of object members, and we need to remember that JSON structures in general are applied for data interfacing between systems.

The possibility of non-unique keys has however consequences apart from software that is not able to process the JSON content properly:
(1) 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 JSON object is 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 } }.
(2) The access and processing software has to deal with two possibilities, either the access addresses one single element or it receives an unordered collection of elements. In the latter case the collection may only be processed as a whole in order to yield deterministic results. In the extreme case the

consequences are that the JSON object can only be handled as a string which would make the JSON extension of SQL obsolete.

Applying JSON technology as storage structure in databases has special constraint requirements for this uniqueness case. In the context of database structures the statement needs to be changed into form "*The key names within an object NEED to be unique*". SQL/JSON extends the relational model into new hybrid model, which needs to be implemented preserving the strict constraint rules, transaction and security service, and preventing "unpredictable behavior" on data.

Considering the reasoning above, it is strange that the SQL professionals in ANSI SQL WG3 have ended up in SQL/JSON proposal Part 2 to present the options of WITH or WITHOUT UNIQUE KEYS[5] clauses to the SQL extension, and even worse: "Since enforcing a constraint is costly, the default is not to check, that is, T.C IS JSON is equivalent to T.C IS JSON WITHOUT UNIQUE KEYS". In this quote, the T.C stands for column C in table T, examples used in the SQL/JSON proposal Part 2 paper.

One might argue, that the "WITH UNIQUE KEYS" clause would prevent some external JSON files with duplicate members from loading to the database, but this is not a reason to break the consistency of data in the database. We need to understand that the context for JSON in databases is more demanding than for JSON files in general. The duplicate members in external JSON files need to be taken care by the loading interface.

For debating on the issue, we have conducted following series of tests trying to create duplicate members on top level of a JSON document and testing how JSON queries behave on accessing these. The tests below show that SQL/JSON implementations at least in current versions of Db2 and Oracle are built according the UNIQUE KEYS model of JSON members, but other tested RDBMS products seem to behave differently and will not support the "WITH UNIQUE KEYS" clause, but for PostgreSQL a workaround has been found:

Db2 for LUW:

Db2 for LUW version 12,1,1 does not recognize the CHECK constraint "IS JSON WITH UNIQUE KEYS" for BLOB-typed JSON column.

```
-- Duplicate object members test:
INSERT INTO T1 (K, J) VALUES
(3,    JSON_TO_BSON('{    "duplica":"First",    "duplica":"Second",    "duplica":"Third","du-
plica":"Last"}'));

db2 => INSERT INTO T1 (K, J) VALUES
db2   (cont.)   =>   (3,   JSON_TO_BSON('{   "duplica":"First",   "duplica":"Second",   "du-
plica":"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 =>
```

Fine, but by accident we have found the article "JSON - Uniqueness controls for key names**"** at https://www.ibm.com/support/pages/json-uniqueness-controls-key-names (on Db2 for i system) saying that the WITHOUT UNIQUE KEYS or WITH UNIQUE KEYS clause has been added to the JSON publishing functions. Even worse – "JSON behavior is changed to default to allowing duplicate key names within JSON documents". This is worrying - do the (young ?) implementers of 'Db2 for i' believe that the SQL/SQL specification needs to be implemented without critics, even on risk of breaking the integrity of Db2.

[5] See the "IS JSON" map of implementations in RDBMS products by Markus Winand

Oracle 23ai:

Oracle recommends using the CHECK constraint “IS JSON WITH UNIQUE KEYS” for text-based JSON columns to avoid inconsistent contents. However, this constraint is automatically included for column based on Oracle’s native JSON data type, which we are using.

```
-- Duplicate object members test:
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second", "duplica":"Third","duplica":"Last"}');

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.
*Action:   Provide JSON data with unique key names in each JSON object.
```

SQL Server XE

SQL Server XE does not recognize the constraint IS JSON WITH UNIQUE KEYS. Let’s test what happens when we enter 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)

SELECT LEFT(K, 4) AS K, JSON_VALUE(J, '$.duplica') 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)

-- How about others if the “First” gets removed
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)

ROLLBACK;
```

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

PostgreSQL

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

```
BEGIN;
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second", "duplica":"Third","duplica":"Last"}');
(SELECT J FROM T1 WHERE K=3);
SELECT J->'duplica' AS Duplica FROM T1 WHERE K=3;
(SELECT (JSONB_EACH(J)).* FROM T1 WHERE K=3);
-- How about if the First gets removed
UPDATE T1 SET J = J - 'duplica' WHERE K=3;
SELECT J->'duplica' AS Duplica FROM T1 WHERE K=3;
-- Who are hiding behind?
(SELECT (JSONB_EACH(J)).* FROM T1 WHERE K=3);
(SELECT J FROM T1 WHERE K=3);
ROLLBACK;
```

Now, applying the script step by step we get following results

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

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

This shows that in PostgreSQL the last duplicate is the only one stored, while others are removed.
So, the Last is also the “First”

```
testdb=> -- How about if the First gets removed
testdb=> UPDATE T1 SET J = J - 'duplica' WHERE K=3;
UPDATE 1
testdb=> SELECT J->'duplica' AS Duplica FROM T1 WHERE K=3;
 duplica
---------

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

testdb=> (SELECT J FROM T1 WHERE K=3);
```

```
 j
----
 {}
(1 row)

testdb=> ROLLBACK;
ROLLBACK
```

Even after we remove the effective duplicate, the other duplicates remain unavailable in the same transaction! Reason to this is that on JSONB typed column the last duplica wins while others are removed automatically without warnings. Is this the service we want? Note that we will *silently loose the information in value parts of those automatically deleted members due to accidently having same key names*!

Current version of PostgreSQL support WITH UNIQUE KEYS constraint clause only for the publishing function JSON_OBJECT().

For JSON column in CREATE TABLE command the UNIQUE KEYS constraint can be created as PL/pgSQL function to be used in CHECK constraint of the JSON column (solution found by Bing, but source unknown):

```
CREATE OR REPLACE FUNCTION unique_keys(js json)
RETURNS boolean LANGUAGE plpgsql IMMUTABLE AS $$
DECLARE
    keys text[];
BEGIN
    -- Extract keys
    SELECT array_agg(key) INTO keys
    FROM json_each(js);
    -- Compare array length with distinct length
    RETURN array_length(keys, 1) = (
        SELECT count(DISTINCT k) FROM unnest(keys) AS k
    );
END;
$$;

CREATE TABLE T (
K  SERIAL PRIMARY KEY,
J  JSONB NOT NULL,
CONSTRAINT with_unique_keys CHECK (unique_keys(J))
);
```

The big difference between these UNIQUE KEYS protection is that while the just trusting on the silent JSONB automatic deletion of first duplicas, the alternative of CONSTRAINT prevention from duplicate keys will generate error message and *abort the whole PostgreSQL transactio*n, but this the right solution!.

MariaDB

MariaDB does not recognize the constraint IS JSON WITH UNIQUE KEYS. Let's test what happens when we enter 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 [testdb]> SELECT LEFT(K, 4) AS K, JSON_VALUE(J, '$.duplica') AS Duplica
    -> FROM T1 WHERE K=3;
+------+---------+
| K    | Duplica |
+------+---------+
```

```
| 3    | First   |
+------+---------+
1 row in set (0.000 sec)

MariaDB [testdb]> -- But, let's see them all
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)

MariaDB [testdb]> -- How about others if the "First" gets 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

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

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

MariaDB [testdb]> ROLLBACK;
Query OK, 0 rows affected (0.001 sec)
```

The current MariaDB version does not support WITH UNIQUE KEYS clause.

Summary – A Critique of SQL/JSON

In the Abstract of ANSI SQL WG3 in SQL/JSON Part 1 proposal the WG3 says "It is important that SQL respond to the requirement to support JSON data by providing facilities for storage, retrieval, querying, and manipulation of JSON data in the context of SQL". In 1.1.8.3 "Updating JSON data" the WG3 says on updating mechanisms "..is beyond the scope of the present proposal and may be addressed in some future proposal".

Postponing the updating part of SQL/JSON standard, criticised also by Petkovic (2020), has led to "Babel effect" messing among the JSON implementations, which we have seen in our experiments on various RDBMS products on solutions for the technical update patterns of the JSON data model. Some solutions are now quite complicated, and we welcome ideas for better solutions.

While even some parts of the SQL/JSON standard have not yet been implemented in all mainstream RDBMS products, some maintenance functions seem to become popular. Also, even late introduction of language standardisation is not a catastrophe, as proven by PostgreSQL adaption of SQL/JSON functions alongside their earlier JSON query implementation. So, we have reason to expect SQL/JSON 2.0 (?) standard, to be extended by JSON data manipulation language part.

Acknowledgements

We thank prof. Stephen J. Hegner on his PostgreSQL PL/pgSQL examples, observations and solutions on JSON structure manipulation. In fact, much of this tutorial is credit to him.
We thank also Mark Gillis at Triton Consulting UK on Db2 advice, Timo Leppänen at Oracle Finland on Oracle 23ai information, and Tim Hall at ORACLE-BASE on Oracle-Relational Duality examples.

References

Baklarz G., Bird P., "Db2 Version 11 JSON Highlights", 2019, Db2V11-JSON-ebook.pdf at https://ibm.ent.box.com/s/g6gxnq9l3se03vgjkcrp27f68c7bxpur

ECMA, “The JSON Data Format”, Standard ECMA-404, 2013, https://ecma-international.org/wp-content/uploads/ECMA-404_1st_edition_october_2013.pdf

Gillis M., "Messing with JSON data in Db2", 2023, https://www.triton.co.uk/messing-with-json-data-in-db2/

Gugnani S., et al. “JSON Relational Duality: A Revolutionary Combination of Document, Object, and Relational Models”, SIGMOD 2025, https://dl.acm.org/doi/pdf/10.1145/3722212.3724441

Hall T., “JSON_TRANSFORM in Oracle Database 21c”, at https://oracle-base.com/articles/21c/json_transform-21c

IETF, “The JavaScript Object Notation (JSON) Data Interchange Format”, Dec 2017, RFC 8259 at https://www.rfc-editor.org/rfc/rfc8259

ISO/IEC, “Part 6: Support for JSON”, iTeh STANDARD PREVIEW at https://cdn.standards.iteh.ai/samples/78937/ec0892ead0034dfca333cd75bb5f348b/ISO-IEC-19075-6-2021.pdf

ISO, “Part 8: Specification of JavaScript Object Notation Encoding Rulers” at ISO Online Browsing Platform (OBP) at https://www.iso.org/obp/ui#iso:std:iso-iec:8825:-8:ed-2:v1:en:term:3.7.7

JSON.org, “Introducing JSON”, https://www.json.org/json-en.html

JSON.org, “The application/json Media Type for JavaScript Object Notation (JSON)”, RFC 4627

Liu Z. H., et al., “Closing the functional and Performance Gap between SQL and NoSQL”, SIGMOD 2016, https://dl.acm.org/doi/pdf/10.1145/2882903.2903731

Liu Z. H., et al., “Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database”, Proceedings of VLDB, Vol 13 No 12, 2020, https://dl.acm.org/doi/10.14778/3415478.3415534

Melton J, et al., "ANSI SQL/JSON: part 1", Mar 4 2014, at https://www.wiscorp.com/pub/DM32.2-2014-00024R1_JSON-SQL-Proposal-1.pdf

NEON, “PostgreSQL JSON”, PostgreSQL Tutorial at https://neon.com/postgresql/postgresql-tutorial/postgresql-json

NEON, “PostgreSQL JSON Extract”, https://neon.com/postgresql/postgresql-json-functions/postgresql-json-extract

NEON, “PostgreSQL JSON Path”, https://neon.com/postgresql/postgresql-json-functions/postgresql-json-path

Oracle Help Center, “11 Oracle SQL Function JSON_TRANSFORM”, JSON Developer’s Guide, at https://docs.oracle.com/en/database/oracle/oracle-database/21/adjsn/oracle-sql-function-json_transform.html

Oracle Help Center, "17.2 SQL/JSON Path Expression Syntax", JSON Developer's Guide, at https://docs.oracle.com/en/database/oracle/oracle-database/23/adjsn/sql-json-path-expression-syntax.html#GUID-AEBAD813-99AB-418A-93AB-F96BC1658618

Petkovic D., "Implementation of JSON Update Framework in RDBMSs", Feb 2020, at https://www.researchgate.net/publication/339331359_Implementation_of_JSON_Update_Framework_in_RDBMSs,

Petkovic D., "SQL/JSON Standard: Properties and Deficiencies", Oct 24 2017, at https://www.researchgate.net/publication/320594498_SQLJSON_Standard_Properties_and_Deficiencies

PostgreSQL.org, "9.16. JSON Functions and Operators", 2025, at https://www.postgresql.org/docs/current/functions-json.html

PostgreSQL.org, "Chapter 41. PL/pgSQL - SQL Procedural Language", 2025, at https://www.postgresql.org/docs/17/plpgsql.html

Zemke F, et al., "SQL/JSON: part 2 - Querying JSON", Mar 4 2014, at https://www.wiscorp.com/pub/DM32.2-2014-00025r1-sql-json-part-2.pdf

Wikipedia, "JSON" at https://en.wikipedia.org/wiki/JSON

Wikipedia, "PostgreSQL" at https://en.wikipedia.org/wiki/PostgreSQL

Winand M. , "A Lot Has Changed Since SQL-92" see the map of IS JSON implementations in RDBMS products, at https://modern-sql.com/caniuse/is-json

DataCamp, "PostgreSQL Querying & Filtering JSON Fields" at https://www.datacamp.com/doc/postgresql/querying-&-filtering-json-fields

* * *

We have covered related issues also in some of our earlier DBTechNet papers:

Laiho M., "Getting Started with DBTechLab VM", 2024
- RDBMSs, open source programming languages and tools

Laiho M., Laux F., et al., "SQL Transactions" – Theory and hands-on exercises, 2012,
at https://dbtechnet.org/documents/?dir=67
- basics of SQL transaction, language versions

Laiho M., Laux F., et al. "SQL Stored Routines", 2016
- Db2 stored procedures, etc

Laiho M., Kurki M., et al. "Introduction to Transaction Programming", 2019
. data access APIs and transaction protocols

Laiho M., "MongoDB Transactions", 2025
- Native JSON database

Laiho M., Laux F., et al. "XML, SQL/XML and the Big Three", 2011
- XPath expressions

Index

Appendix 2 Counting number of object members on top level?

For the algorithm of updating values or removing ALL members *found by value*, browsing [0..max] of the JSON member list in Cases 2.2, 3.2, etc. in the Appendix 1, we need to sort out the count of members and elements. In fact, Db2 SQL/JSON is the only one for which we have not yet found general solution for these ALL cases. The solutions of the other RDBMS systems present interesting variations for this simple topic.

Db2 for LUW

This can be solved based on the JSON_KEYS function which we have implemented as external stored procedure written in C language, see Appendix 3.

Oracle 23ai

```
SQL> SELECT MAX(idx) AS count_members
  2  FROM (SELECT jt.idx
  3        FROM T1,
  4        JSON_TABLE(J, '$.*'
  5          COLUMNS (
  6           idx  FOR ORDINALITY
  7         )
  8        ) jt
  9* WHERE T1.K=1);

   COUNT_MEMBERS
_________________
               6
```

SQL Server

```
DECLARE @json VARCHAR(1000)
SELECT @json=J FROM T1 WHERE K= 1;
SELECT COUNT(*) AS MemberCount
FROM OPENJSON(@json);
MemberCount
-----------
6

(1 row affected)
```

PostgreSQL

```
testdb=> SELECT jsonb_object_keys(J)
FROM T1 WHERE K=1;
 jsonb_object_keys
-------------------
 mem1
 mem2
 mem3
 mem4
 mem5
 mem6
(6 rows)

testdb=> SELECT COUNT(*) FROM
(SELECT jsonb_object_keys(J)
FROM T1 WHERE K=1);
 count
-------
```

```
     6
(1 row)
```

MariaDB:

```
MariaDB [testdb]> SELECT K, JSON_LENGTH(J) AS count_members
    -> FROM T1 WHERE K=1;
+---+---------------+
| K | count_members |
+---+---------------+
| 1 |             6 |
+---+---------------+
1 row in set (0.009 sec)
```

Appendix 3 Implementing JSON_KEYS function to Db2 for LUW?

JSON_KEYS function is not included in SQL/JSON standard, but It has been implemented in MySQL/MariaDB, PostgreSQL, and some DBMS products outside our JSON experiments. Sean Stuber reports on his PL/SQL function for listing the keys in Oracle SQL/JSON. In SQL Server a solution can be based on OPENJSON function.

The JSON_KEYS function lists the key names on top of a JSON object. The listing provides basis for accessing all top-list members of the JSON document providing means for accessing the corresponding value contents. As such it is a key function for implementing the JSON update patterns for Cases 2.2, 3.2 for Db2 LUW in the Appendix 1.

The missing JSON_KEYS() function of Db2 LUW could be implemented by PYTHON subprogram applying the following

```
>>> myjson = {"mem1":123,"mem2":"123","mem3":true,"mem4":null,"mem5": [ 123, "123", true, null, [1, 2], {} ],"mem6":
{ "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2, 3], "m66":{} }}
>>> print(myjson.keys())
dict_keys(['mem1', 'mem2', 'mem3', 'mem4', 'mem5', 'mem6'])
>>>
```

but the Python interface needs to be sorted out somehow. The Python support for Db2 LUW has been available for some years, and unofficial instructions are available in GitHub at https://github.com/ibmdb/python-ibmdb.

> While experimenting with the JSON on Python we found out that Python requires JSON literals “true” and “false” to start by upper case letters “True” and “False”, and it doesn’t accept the literal “null”, but for JSON_KEYS() function we don’t need to care about these.

Implementing JSON_KEYS as external routine written in C language

Finally, we decided to implement the JSON_KEYS function to Db2 for LUW using external routine written in C installed in Windows DLL file to be accessed by wrapper interface to build the JSON update pattern Case 2.2 for Db2 LUW.

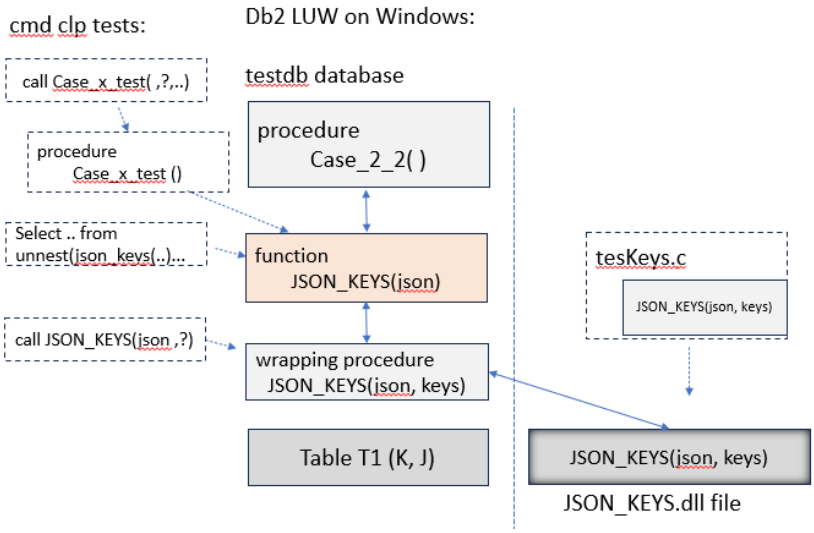

Figure x. Building of the JSON_KEYS function for Db2 LUW

For the C-compiler to Windows 11 platform we installed the GNU gcc port MinGW-w64. Then compiling our JSON_KEYS.c code into DLL file by

```
gcc -shared -Os -s -o JSON_KEYS.dll JSON_KEYS.c
```

The interface of the generic JSON_KEYS function for limited sizes in C is following

```
void JSON_KEYS (char json[4001], char keys[401])
```

The algorithm of collecting the key names of all top-level members from a JSON document, skipping the various value types and nested parts is quite complicated and required many test rounds, so before the final test rounds a hosting test program was coded and tested step by step separately from the DLL file.

Disclaimer:

The compiled DLL file is available ZIPped at JSON_KEYS.zip without any guarantees, but passed in all our tests.

We leave programming of the corresponding code as exercise for interested readers.

The JSON_KEYS function in DLL can be accessed as external stored procedure of Db2, by creating the interface as follows

```
CREATE OR REPLACE PROCEDURE JSON_KEYS (
     IN  json     VARCHAR(1000),
     OUT keys     VARCHAR(400))
   SPECIFIC JSON_KEYS_F
   EXTERNAL NAME 'C:\Users\Martti\JSON_KEYS.dll'
   FENCED
   LANGUAGE C
   PARAMETER STYLE GENERAL
   NO SQL
   DETERMINISTIC EXTERNAL ACTION;
```

Note: The EXTERNAL NAME clause needs to be updated to reference the directory where the DLL file is stored.

The implementation of the DLL for use of an external procedure turned out to be a tricky process. When an external Db2 procedure is created, the DLL file becomes reserved, and it is impossible to update or even delete. To replace the DLL file with a new version, we applied following steps in Db2 session and Command Prompt window of Windows (in standalone environment without any concurrent use):

Db2:

```
DROP PROCEDURE JSON_keys;
```

```
COMMIT;
FORCE APPLICATION ALL;
QUIT;
```

Command Prompt:

```
DEL JSON_KEYS.dll
```

The error message “Access is denied.” was solved by restarting the whole Windows server. This of course, is not possible in production environment. The problem is avoided by compiling new version of the DLL file using name versioning for the DLL file as follows and then deleting the old version of DLL file

Creating the interface (wrapper procedure)

```
db2 => CREATE OR REPLACE PROCEDURE JSON_KEYS (
db2 (cont.) =>      IN  json     VARCHAR(1000),
db2 (cont.) =>      OUT keys     VARCHAR(400))
db2 (cont.) =>    SPECIFIC JSON_KEYS_0_1
db2 (cont.) =>    EXTERNAL NAME 'C:\Users\Martti\JSON_KEYS_0_1.dll'
db2 (cont.) =>    FENCED
db2 (cont.) =>    LANGUAGE C
db2 (cont.) =>    PARAMETER STYLE GENERAL
db2 (cont.) =>    NO SQL
db2 (cont.) =>    DETERMINISTIC EXTERNAL ACTION;
DB20000I  The SQL command completed successfully.
db2 => COMMIT;
DB20000I  The SQL command completed successfully.
```

Testing the procedure in Db2 CLP session:

```
db2 => CALL JSON_KEYS('[{"mem1": 123,"mem2": "123"}]', ?);

  Value of output parameters
  --------------------------
  Parameter Name  : KEYS
  Parameter Value : ["mem1","mem2"]

  Return Status = 0

Testing with the JSON of test case K=1

db2 => CALL JSON_KEYS('[{"mem1":123,"mem2":"123","mem3":true,"mem4":null,"mem5": [ 123, "123",
true, null, [1, 2], {} ],"mem6": { "m61":1, "m62":"123", "m63": true, "m64": null, "m65":[2,
3], "m66":{ } }}]', ?);

  Value of output parameters
  --------------------------
  Parameter Name  : KEYS
  Parameter Value : ["mem1","mem2","mem3","mem4","mem5","mem6"]

  Return Status = 0

Testing with the JSON of test case K=2

db2 => CALL JSON_KEYS('[{"mem1": 123,"mem2": "123","mem3": true,"mem4": null,"mem5": [ 123,
"123", "string", true, 123, 124, 124, null, [1, 2], {}, 123 ],"mem6": { "m61":1, "m62":"123",
"m63": true, "m64": null, "m65":[2, 3], "m66":{} },"mem7": 123,"mem8": "123"}]}]', ?);

  Value of output parameters
  --------------------------
  Parameter Name  : KEYS
  Parameter Value : ["mem1","mem2","mem3","mem4","mem5","mem6","mem7","mem8"]

  Return Status = 0
```

Creating the JSON_KEYS() function

Next, we created the SQL interface function for accessing the procedure

```
CREATE TYPE stringArray AS VARCHAR(20) ARRAY[100];
COMMIT;

--#SET TERMINATOR @
CREATE OR REPLACE FUNCTION JSON_KEYS
    (IN  kp    INT,
     IN  json  VARCHAR(1000)
    )
RETURNS stringArray
DETERMINISTIC
LANGUAGE SQL
SPECIFIC JSON_KEYS_F
BEGIN
   DECLARE keys stringArray;
   DECLARE ind INTEGER;
   DECLARE keyList VARCHAR(400);
   DECLARE key VARCHAR(20);
   SET keys = ARRAY[];
   SET ind = 1;
   -- SET keyList = '"mem1","mem2","mem3","mem4","mem5","mem6"';
   CALL JSON_KEYS(json, keyList); -- already tested to work properly
   SET keyList = REPLACE(keyList, '[', '');   -- removing the square brackets
   SET keyList = REPLACE(keyList, ']', '');   -- to enable the use "ARRAY{?]
   WHILE (LENGTH(keyList) > 0 ) DO
     SET key = SUBSTR(keyList, 1,
               COALESCE(NULLIF(LOCATE(',', keyList) - 1, -1),
               LENGTH(keyList)));
     IF (LOCATE(',', keyList) > 0) THEN
         SET keyList = SUBSTR(keyList, LOCATE(',', keyList) + 1);
     ELSE
         SET keyList = '';
     END IF;
     SET keys[ind] = key;
     SET ind = ind+1;
   END WHILE;
   RETURN keys;
END;
@
--#SET TERMINATOR ;

-- testing the JSON_KEYS() function
SELECT t.id, t.key
FROM UNNEST(JSON_KEYS(1, '[{"mem1":123,"mem2":231}]'))
WITH ORDINALITY AS t(key, id);

db2 => SELECT t.id, t.key
db2 (cont.) => FROM UNNEST(JSON_KEYS(1, '[{"mem1":123,"mem2":231}]'))
db2 (cont.) => WITH ORDINALITY AS t(key, id);

ID          KEY
----------- --------------------
          1 "mem1"
          2 "mem2"

  2 record(s) selected.

db2 => SELECT t.id, t.key
db2 (cont.) => FROM UNNEST(JSON_KEYS(1, (SELECT JSON_QUERY(J, '$') FROM T1 WHERE K=1)))
db2 (cont.) => WITH ORDINALITY AS t(key, id);

ID          KEY
----------- --------------------
          1 "mem1"
          2 "mem2"
```

```
          3 "mem3"
          4 "mem4"
          5 "mem5"
          6 "mem6"

  6 record(s) selected.

db2 =>
```

Creating test version for the Case 2.2

Now that we have implemented the JSON_KEYS() function for Db2 for LUW on Windows, it is time to apply it for our JSON maintenance cases. We will use cursor programming to browse thru the list of keys found by the JSON_KEYS() function,

```
db2 => DECLARE keycurs CURSOR FOR
db2 (cont.) => SELECT t.id, t.key
db2 (cont.) => FROM UNNEST(JSON_KEYS((SELECT JSON_QUERY(J, '$') FROM T1 WHERE K=1)))
db2 (cont.) => WITH ORDINALITY AS t(key, id)
db2 (cont.) => FOR FETCH ONLY
db2 (cont.) => WITH UR;
DB20000I  The SQL command completed successfully.
db2 => OPEN keycurs;
DB20000I  The SQL command completed successfully.
db2 => FETCH keycurs;

ID          KEY
----------- --------------------
          1 "mem1"

  1 record(s) selected.
. . .
db2 => CLOSE keycurs;
DB20000I  The SQL command completed successfully.
```

and by the current key we read the value of the corresponding member. If the value matches with the given value for processing of the Case, then the member will be updated or deleted depending on the Case.

```
--#SET TERMINATOR @
CREATE OR REPLACE PROCEDURE Case2_2
          (IN  kp INT,
           IN  given_value VARCHAR(100),
           IN  new_value VARCHAR(100),
           -- OUT parameters are meant only for testing purposes:
           OUT id          INT,
           OUT mem_Name    VARCHAR(20),
           OUT old_value   VARCHAR(100),
           OUT json_out    VARCHAR(1000)
          )
SPECIFIC Case2_2
LANGUAGE SQL
BEGIN ATOMIC
    DECLARE json VARCHAR(1000);
    DECLARE memCount INT;
    DECLARE memName  VARCHAR(20);
    DECLARE oldValue VARCHAR(1000);
    DECLARE sqlcode  INT DEFAULT 0;
    DECLARE keycurs CURSOR FOR
      SELECT t.id, t.key
      FROM UNNEST(JSON_KEYS((SELECT JSON_QUERY(J, '$') FROM T1 WHERE K=kp)))
      WITH ORDINALITY AS t(key, id)
      FOR FETCH ONLY
      WITH UR;
    SELECT BSON_TO_JSON(J) INTO json FROM T1 WHERE K = kp;
```

```
     set json_out = json; -- only for tracing
    -- Loop through the JSON members
    OPEN keycurs;
    WHILE sqlcode = 0 DO
       FETCH keycurs INTO id, memName; -- Extract the value at the current index
         set mem_Name = memName; -- only for tracing
       SET memName = REPLACE(memName, '"', '');
       SELECT JSON_QUERY(J, '$.' || memName || ' ') INTO oldValue
       FROM T1 WHERE K = kp;
         set old_Value = oldValue; -- only for tracing
       -- Check if the value matches '123'
       IF (oldValue = given_value) THEN
          UPDATE T1
          SET J = SYSTOOLS.JSON_UPDATE(J, '{$set: {' || memName || ': ' || new_value || '}} ')
          WHERE K = kp;
       END IF;
       --  return; -- stop for tracing
    END WHILE;
END;
@
--#SET TERMINATOR ;
```

We can now apply test version of the Case 2.2 by reading the output parameters by questions marks as follows:

```
CALL Case2_2 (1, '123', '127', ?, ?, ?, ?);

db2 => CALL Case2_2 (1, '123', '127', ?, ?, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : ID
  Parameter Value : 6

  Parameter Name  : MEM_NAME
  Parameter Value : mem6

  Parameter Name  : OLD_VALUE
  Parameter Value : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null, "m65" : [ 2, 3 ],
"m66" : {  } }

  Parameter Name  : JSON_OUT
  Parameter Value : { "mem1" : 123, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [
123, "123", true, null, [ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true,
"m64" : null, "m65" : [ 2, 3 ], "m66" : {  } } }

  Return Status = 0
db2 =>
```

Note: the JSON_OUT parameter shows the original json, whereas we need to check the updated document by JSON_QUERY as follows:

```
db2 => SELECT cast(K as smallint) as k,
db2 (cont.) => JSON_QUERY(J, 'strict $' RETURNING VARCHAR(1000)) as result
db2 (cont.) => FROM T1 WHERE K=1;

K      RE-
SULT
------ -------------------------------------------------------------------------------
     1 { "mem1" : 127, "mem2" : "123", "mem3" : true, "mem4" : null, "mem5" : [ 123, "123",
true, null, [ 1, 2 ], {  } ], "mem6" : { "m61" : 1, "m62" : "123", "m63" : true, "m64" : null,
"m65" : [ 2, 3 ], "m66" : {  } }
}

  1 record(s) selected.

db2 => rollback;
```

```
DB20000I  The SQL command completed successfully.
```

How about keys of nested objects?

```
db2 => SELECT t.id, t.key
db2 (cont.) => FROM UNNEST(JSON_KEYS((SELECT JSON_QUERY(J, '$.mem6') FROM T1 WHERE K=1)))
db2 (cont.) => WITH ORDINALITY AS t(key, id);

ID          KEY
----------- --------------------
          1 "m61"
          2 "m62"
          3 "m63"
          4 "m64"
          5 "m65"
          6 "m66"

  6 record(s) selected.

db2 =>
```

As our JSON_KEYS() now works, we can proceed by removing the output variables and implement the Case 2.2 and Case 3.2.

References

GitHub, “Python support for IBM Db2 for LUW and IBM Db2 for z/OS”, https://github.com/ibmdb/python-ibmdb

Silpiö K., “C-KIELI”, 1992, ATK-instituutti
- great source of C language for Finnish readers

Stuber S. D., “How to create a PL/SQL function to iterate JSON Keys for SQL”, 2024, https://seanstuber.com/2024/03/02/how-to-create-a-pl-sql-function-to-iterate-json-keys-for-sql/

Wikipedia, “Mingw-w64”, at https://en.wikipedia.org/wiki/Mingw-w64