
ORACLE®

JSON Support in MySQL

Evgeny Potemkin
Snr Principal Engineer

Manyi Lu
Snr Engineering Manager

MySQL Optimizer Team
April, 2015

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Program Agenda

1. Introduction
2. JSON datatype
3. Functions to handle JSON data
4. Indexing of JSON data
5. Examples
6. Performance

Why JSON support in MySQL?

- Convenient object serialization format
- Need to effectively process JSON data
- Provide native support for JavaScript applications
- Seemless integration of relational and schema-less data
- Leverage existing database infrastructure for new applications

Storage Options

- Text
  - Fast to insert
  - Human-readable
  - Requires validation
  - Requires parsing
  - Hard to update
- Binary
  - Validate only once
  - Fast access
  - In-place updates
  - Slower to insert
  - Unreadable as-is

New JSON datatype

- Optimized for read intensive workload
- Parse and validation on insert only
- Dictionary
  - Sorted objects' keys
  - Fast access to array cells by index
- In-place updates (future enhancement in SE), space reservation
- Smart size: 64K & 4G
- UTF8

New JSON datatype: Supported Types

- ALL native JSON types
  - Numbers, strings, bool
  - Objects, arrays
- Extended
  - Date, time, datetime, timestamp
  - Other

Examples: CREATE and INSERT

```
CREATE TABLE t1 (data JSON);

INSERT INTO t1(data) VALUES
 (‘{ "series": 1}’), (‘{ "series": 7}’),  (‘{ "series": 3}’),
 (‘{ "series": 4}’), (‘{ "series": 10}’), (‘{ "series": 2}’),
 (‘{ "series": 6}’), (‘{ "series": 5}’),  (‘{ "series": 8}’),
 (‘{ "series": 11}’);
```

Examples: more on INSERT

```
INSERT INTO t1(data) VALUES
  (‘{ “a”: “valid”, “json”: [“text”] }’),
  (JSN_QUOTE('some, might be formatted, { text } with "quotes"'));
```

Examples: SELECT

```
> SELECT * FROM t1 LIMIT 3;
+---------------+
| data          |
+---------------+
| {"series": 1} |
| {"series": 7} |
| {"series": 3} |
+---------------+
```

JSON Comparator: Design Principles

- Polymorphic behavior
- Seamless and consistent comparison
  - JSON vs JSON, JSON vs SQL
  - Different data types always non-equal
  - No automatical type conversion
- Robustness
- Extensive use of caching

JSON Comparator: example

```
SELECT * FROM t1 WHERE
  jsn_extract(data,"$.series") >= 7 AND
  jsn_extract(data,"$.series") <=10;
```

```
+-----------------+
| data            |
+-----------------+
| {"series": 7}   |
| {"series": 10}  |
| {"series": 8}   |
+-----------------+
```

New Functions to Handle JSON Data: Path

[[[database.]table.]column]$<path spec>

- Path expr
  - [[[database.] table.] field]
  - $
  - .identifier
  - [array]
  - .* and [*]
  - **
- Example
  - db.phonebook.data (future extension)
  - document's root
  - $.user.address.street
  - $.user.addresses[2].street
  - $.user.addresses[*].street
  - $.user**.phone

New functions to handle JSON data: Funcs

- Info
  - JSN_VALID()
  - JSN_TYPE()
  - JSN_KEYS()
  - JSN_LENGTH()
  - JSN_DEPTH()
  - JSN_CONTAINS_PATH()
- Modify
  - JSN_REMOVE()
  - JSN_APPEND()
  - JSN_SET()
  - JSN_INSERT()
  - JSN_REPLACE()

New functions to handle JSON data: Funcs

- Create
  - JSN_MERGE()
  - JSN_ARRAY()
  - JSN_OBJECT()
- Get data
  - JSN_EXTRACT()
  - JSN_SEARCH()
- Helper
  - JSN_QUOTE()
  - JSN_UNQUOTE()

Examples: CREATE + SELECT

```
CREATE TABLE t2 AS
  SELECT
    JSN_OBJECT("b_series",
      JSN_ARRAY(
        JSN_EXTRACT(data, "$.series")))
    AS data
    FROM t1;
```

```
> SELECT * FROM t2 LIMIT 3;
+--------------------+
| data               |
+--------------------+
| {"b_series": [1]} |
| {"b_series": [7]} |
| {"b_series": [3]} |
+--------------------+
```

Examples: UPDATE + join

```
UPDATE t1, t2
  SET t1.data=
    JSN_INSERT(t1.data,"$.inverted",
              11 – JSN_EXTRACT(t2.data,"$.b_series[0]"))
WHERE
  JSN_EXTRACT(t1.data, "$.series") =
    JSN_EXTRACT(t2.data,"$.b_series[0]");
```

Examples: result of UPDATE

```
> SELECT * FROM t1 LIMIT 3;
+----------------------------------+
| data                             |
+----------------------------------+
| {"series": 1, "inverted": 10} |
| {"series": 7, "inverted": 4}   |
| {"series": 3, "inverted": 8}   |
+----------------------------------+
```

Examples: a subquery

```
SELECT * FROM t1
WHERE JSN_EXTRACT(data,"$.series") IN
 (
  SELECT JSN_EXTRACT(data,"$.inverted")
  FROM t1
  WHERE JSN_EXTRACT(data,"$.inverted")<4
 );
```

```
+-----------------------------------+
| data                              |
+-----------------------------------+
| {"series": 1, "inverted": 10} |
| {"series": 3, "inverted": 8}  |
+-----------------------------------+
```

Indexing JSON data

- Use Functional Indexes, Luke ☺
- STORED and VIRTUAL types are supported

```
CREATE TABLE t1
    (data JSON, id INT AS (JSN_EXTRACT(data,”$.id”)) STORED,
    PRIMARY KEY(id));
ALTER TABLE t1
    ADD COLUMN id INT AS (JSN_EXTRACT(data, "$.series")),
    ADD INDEX id_idx (id);
```

Indexing JSON data: STORED vs VIRTUAL

- STORED
  - Primary & secondary
  - BTREE, FTS, GIS
  - Mixed with fields
  - Req. table rebuild
  - Not online
- VIRTUAL
  - Secondary only
  - BTREE only
  - Mix with virtual column only
  - No table rebuild
  - Instant ALTER (Coming soon!)
  - Faster INSERT

Indexing JSON data: an example

SELECT data FROM t1 WHERE

- JSN_EXTRACT(data,"$.series") BETWEEN 3 AND 5;
- id BETWEEN 3 AND 5;

```
+--------------------------------+----+
| data                           | id |
+--------------------------------+----+
| {"series": 3, "inverted": 8}   |  3 |
| {"series": 4, "inverted": 7}   |  4 |
| {"series": 5, "inverted": 6}   |  5 |
+--------------------------------+----+
```

Indexing JSON data: an example

```
> EXPLAIN SELECT data FROM t1 WHERE JSN_EXTRACT(data,"$.series")
BETWEEN 3 AND 5;
+----+-------------+-------+-------------+-------+...+-------------------------+
| id | select_type | table | partitions  | type  |   | Extra                   |
+----+-------------+-------+-------------+-------+...+-------------------------+
|  1 | SIMPLE      | t1    | NULL        | range |   | Using index condition |
+----+-------------+-------+-------------+-------+...+-------------------------+

 select `test`.`t1`.`data` AS `data` from `test`.`t1`
 where (`test`.`t1`.`id` between 3 and 5)
```

Comparison to Facebook's solution

- Path expr
  - Glob ops
  - Deterministic name resolution
  - JSON manipulation functions
- More generic indexes
  - Primary keys
  - Virtual secondary keys
  - Any MySQL data type
- More versatile comparator
- Different format
  - In-place updates
  - Faster lookups

Roadmap

- Online alter for virtual columns
- Advanced JSON functions
- In-place update of JSON/BLOB
- Full text and GIS index on virtual columns
- Improved performance through condition pushdown

Questions?

Thank You!

ORACLE

MySQL