PERCONA

# JSON Data Improvements in MySQL 8.0

Dave Stokes
@Stoker
David.Stokes@percona.com

ORACLE DevLive | Level Up | MySQL Summit

# JSON Data Improvements in MySQL 8.0

**Dave Stokes**

Percona

# JSON Data Improvements in MySQL 8.0

The JSON data type was introduced in MySQL 5.7 and was dramatically improved in version 8.0.

Chief among these changes was the introduction of JSON_TABLE(), which temporarily transforms JSON data into structured data for processing with SQL commands such as window functions.

And you also gain the ability to test JSON data for required fields, range checks, and data type checks to ensure that bad data does not make it into your database instance.

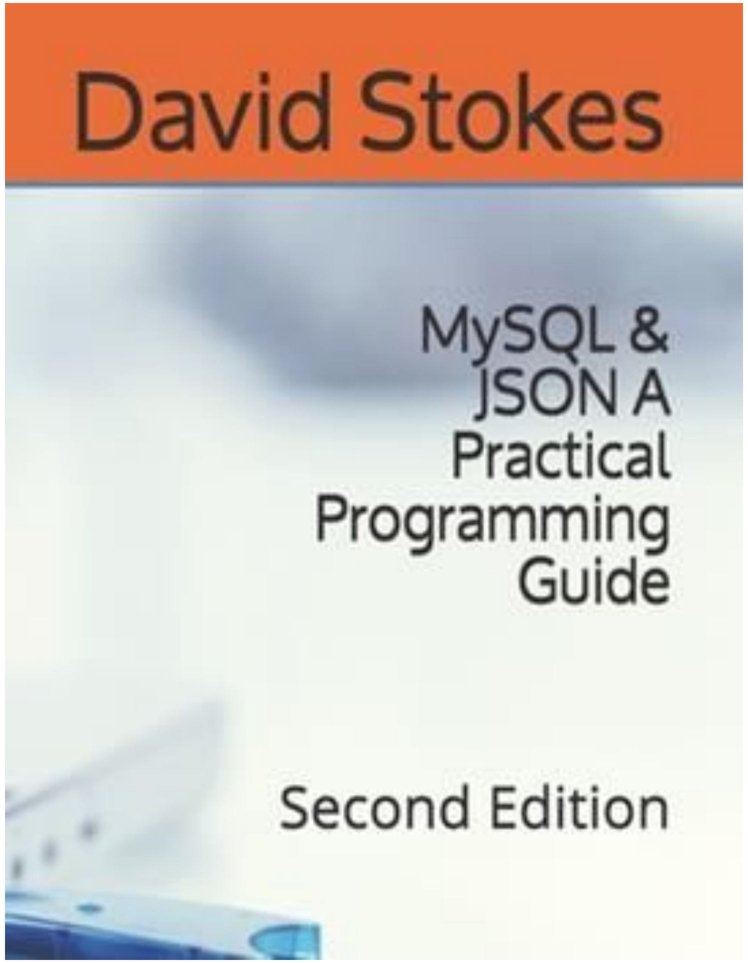
These and other MySQL 8.0 JSON features will be covered in this session.

# About me

Technology Evangelist at Percona

Long time open source advocate

Author



David Stokes

MySQL & JSON A Practical Programming Guide

Second Edition

@Stoker   David.Stokes@Percona.com

# Differences - SQL versus NoSQL

# Traditional Relational Databases

1. **Normalized data - Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity.**

2. **Present the data to the user as relations with logical connection between different tables.**

3. **Provide relational operators to manipulate the data in tabular form.**

4. **Strict Data Types enforce ‘rigor’ on data.**

5. **Data decisions upfront.**

# NoSQL JSON Databases

1. **Freeform & Flexible - data stored in key/value pairs.**
2. **No rigor on data.**
3. **Many different formats in same schema.**
4. **Data decisions on output.**

# Quiz Time!

```
SQL >CREATE TABLE q1 (question1 INT, question2 CHAR(5));
SQL >insert into q1 values (1,'Southern California Linux Expo 20x');
ERROR: 1406: Data too long for column 'question2' at row 1
SQL > insert into q1 values ('1oo','SCaLE');
ERROR: 1265: Data truncated for column 'question1' at row 1
```

**What is in table q1?**

```
SQL > select * from q1;
Empty set (0.0009 sec)
```

NoSQL vendors claimed JSON solved many problems with Structured Query Language (SQL)!

# ~ 10 years ago

Then they announced they were going to support relational features like transactions.

Somewhat succeeded.

Relational Databases Added JSON support

# So, What is JSON?

# JavaScript Object Notation - https://en.wikipedia.org/wiki/JSON

JSON (JavaScript Object Notation, pronounced /ˈdʒeɪsən/; also /ˈdʒeɪˌsɒn/) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

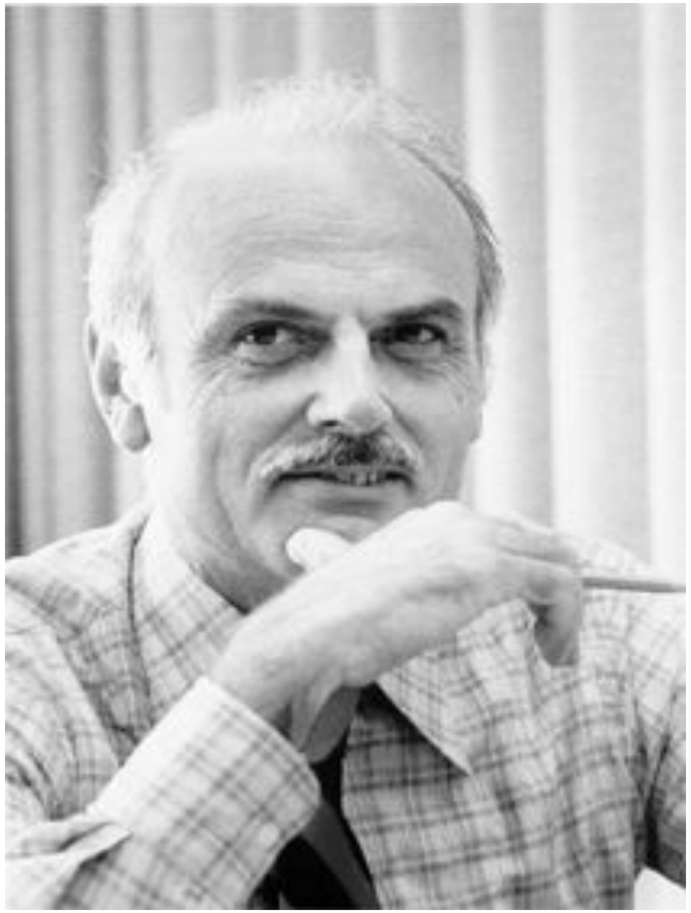
# The difference between how Developers and DBAs view data

```
{
  “id”:   12345,
  “name”: “A. Programmer”,
  “age”:  21,
  “languages”: [“PHP”,”GO”]
}
```

```
CREATE TABLE staff (
        id INTEGER AUTO_INCREMENT,
        name CHAR(100) NOT NULL,
        department INT UNSIGNED NOT NULL,
        languages    CHAR(255)
 );
```

# Relational Model

# Dr. Edgar F. Codd



# Structured Query Language

Only Programming language from the 1970s still heavily used

It introduced the concept of accessing many records with one single command

Data divvied up into logical groupings - customer, product, order, etc.

Originally designed to minimize data duplication
(disk drives were slow and expen$ive in 1970s/80s)

particularly useful in handling structured data, i.e. data incorporating relations among entities and variables

# So why didn’t JSON Document Databases Replace Relational Systems?

# QUIZ 2

```
SQL > create table q2 (foo JSON);
Query OK, 0 rows affected (0.0096 sec)
SQL > insert into q2 values ('{ "A" : 1, "A": "a", "A": [1,2]}');
Query OK, 1 row affected (0.0080 sec)
```

## The answer

```
SQL > select * from q2;
+-----------------+
| foo             |
+-----------------+
| {"A": [1, 2]}   |
+-----------------+
1 row in set (0.0005 sec)
```

# JSON is free form

UTF8MB4!

Do not have to change tables to add new field - DDL operations can be expensive with a RDMS

Documents not rows

Data too easily duplicated, gets outdated

Many-to-many relationships are very hard to manage

Nested Objects

May not meet systemic data usage needs

Consistency-ish.

No rigor applied to data :
email
eMail
e-mail
electronicMail
electonicMail

Easy to abandon old data

Agile style practices are not optimized for database operations

What is the biggest priority - development ease or using data?

# MySQL & JSON

MySQL added a JSON datatype with MySQL 5.7 - 2015

- Data stored in a binary blob
  - Sorted by key
  - ~1gb payload

# Confession:

You could store a JSON document in a database ***BEFORE*** there was a JSON data

- ➔ Document was stored in a TEXT field
- ➔ To search you use REGEX
- ➔ Hard to extract just one or a few components of the string
- ➔ Expensive to read, process and rewrite the entire revised string

# MySQL JSON Example

```
CREATE TABLE ato (id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, data JSON);

INSERT INTO ato (data) VALUES ('{"Name": "Dave", "Answer": 42}');
```

```
 SELECT id, data FROM ato\G
*************************** 1. row ***************************
  id: 1
data: {"Name": "Dave", "Answer": 42}
1 row in set (0.0012 sec)
```

# MySQL

```
SELECT data->>'$.Answer' FROM ato\G
*************************** 1. row ***************************
data->>'$.Answer': 42
1 row in set (0.0008 sec)
```

# -> versus ->>

**SELECT data->'$.Name' FROM ato;**

```
+-----------------+
| data->'$.Name' |
+-----------------+
| "Dave"          |
+-----------------+
1 row in set (0.0010 sec)
```

**SELECT data->>'$.Name' FROM ato;**

```
+------------------+
| data->>'$.Name' |
+------------------+
| Dave             |      → strips the "'s
+------------------+
1 row in set (0.0010 sec)
```

# JSON Functions

# MySQL's JSON Functions - 12.18.1 of Manual

| Name | Description |
|---|---|
| -> | Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT(). |
| ->> | Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()). |
| JSON_ARRAY() | Create JSON array |
| JSON_ARRAY_APPEND() | Append data to JSON document |
| JSON_ARRAY_INSERT() | Insert into JSON array |
| JSON_CONTAINS() | Whether JSON document contains specific object at path |
| JSON_CONTAINS_PATH() | Whether JSON document contains any data at path |
| JSON_DEPTH() | Maximum depth of JSON document |
| JSON_EXTRACT() | Return data from JSON document |
| JSON_INSERT() | Insert data into JSON document |
| JSON_KEYS() | Array of keys from JSON document |
| JSON_LENGTH() | Number of elements in JSON document |
| JSON_MERGE() | Merge JSON documents, preserving duplicate keys. Deprecated synonym for JSON_MERGE_PRESERVE() |
| JSON_MERGE_PATCH() | Merge JSON documents, replacing values of duplicate keys |
| JSON_MERGE_PRESERVE() | Merge JSON documents, preserving duplicate keys |
| JSON_OBJECT() | Create JSON object |
| JSON_OVERLAPS() | Compares two JSON documents, returns TRUE (1) if these have any key-value pairs or array elements in common, otherwise FALSE (0) |
| JSON_PRETTY() | Print a JSON document in human-readable format |
| JSON_QUOTE() | Quote JSON document |
| JSON_REMOVE() | Remove data from JSON document |
| JSON_REPLACE() | Replace values in JSON document |
| JSON_SCHEMA_VALID() | Validate JSON document against JSON schema; returns TRUE/1 if document validates against schema, or FALSE/0 if it does not |
| JSON_SCHEMA_VALIDATION_REPORT() | Validate JSON document against JSON schema; returns report in JSON format on outcome on validation including success or failure and reasons for failure |
| JSON_SEARCH() | Path to value within JSON document |
| JSON_SET() | Insert data into JSON document |
| JSON_STORAGE_FREE() | Freed space within binary representation of JSON column value following partial update |
| JSON_STORAGE_SIZE() | Space used for storage of binary representation of a JSON document |
| JSON_TABLE() | Return data from a JSON expression as a relational table |
| JSON_TYPE() | Type of JSON value |
| JSON_UNQUOTE() | Unquote JSON value |
| JSON_VALID() | Whether JSON value is valid |
| JSON_VALUE() | Extract value from JSON document at location pointed to by path provided; return this value as VARCHAR(512) or specified type 8.0.21 |
| MEMBER OF() | Returns true (1) if first operand matches any element of JSON array passed as second operand, otherwise returns false (0) 8.0.17 |

MySQL supports two aggregate JSON functions JSON_ARRAYAGG() and JSON_OBJECTAGG(). JSONPRETTY() for pretty printing. And You can see how much storage space a given JSON value takes up, and how much space remains for additional storage, using JSON_STORAGE_SIZE() and JSON_STORAGE_FREE()

# Generated Column - Extract Data to be Indexed

```
ALTER TABLE ato ADD COLUMN h CHAR(25) GENERATED ALWAYS as (data->"$.Name");

CREATE INDEX h_index on ato(h);
Query OK, 0 rows affected (0.0324 sec)

Records: 0  Duplicates: 0  Warnings: 0
explain format=tree select data->>"$.Name" FROM ato WHERE h = 'Dave'\G
*************************** 1. row ***************************
EXPLAIN: -> Filter: (ato.h = 'Dave')  (cost=0.35 rows=1)
    -> Index lookup on ato using h_index (h='Dave')  (cost=0.35 rows=1)

1 row in set (0.0011 sec)
```

# Multi-Valued Indexes - Great for Arrays

```
mysql> CREATE TABLE s (id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    -> name CHAR(20) NOT NULL,
    -> j JSON,
    -> INDEX nbrs( (CAST(j->'$.nbr' AS UNSIGNED ARRAY)))
    -> );
```

```
mysql> SELECT * FROM s;
+----+-------+----------------------+
| id | name  | j                    |
+----+-------+----------------------+
|  1 | Moe   | {"nbr": [1, 7, 45]}  |
|  2 | Larry | {"nbr": [2, 7, 55]}  |
|  3 | Curly | {"nbr": [5, 8, 45]}  |
|  4 | Shemp | {"nbr": [3, 6, 51]}  |
+----+-------+----------------------+
```

Previously you were limited to a 1:1 index:row limit!

# Using Multi-value Indexed Field

```
mysql>  SELECT * FROM s WHERE 7 MEMBER OF (j->"$.nbr");
+----+-------+---------------------+
| id | name  | j                   |
+----+-------+---------------------+
|  1 | Moe   | {"nbr": [1, 7, 45]} |
|  2 | Larry | {"nbr": [2, 7, 55]} |
+----+-------+--------------------
```

MEMBER OF(), JSON_CONTAINS() & JSON_OVERLAP()

# JSON Table - Unstructured data temporarily structured

```
mysql> select country_name, IndyYear from countryinfo,
json_table(doc,"$"
            columns (  country_name char(20) path "$.Name",
                       IndyYear int path "$.IndepYear")
            ) as stuff
where IndyYear > 1992;
+----------------+----------+
| country_name   | IndyYear |
+----------------+----------+
| Czech Republic |     1993 |
| Eritrea        |     1993 |
| Palau          |     1994 |
| Slovakia       |     1993 |
+----------------+----------+
4 rows in set, 67 warnings (0.00 sec)
```

**Now the JSON data can be process with SQL**!

# JSON Table - Handle missing data

```
mysql> SELECT name,
          Info->>"$.Population",
          Pop FROM city2,
          JSON_TABLE(Info,"$" COLUMNS
          ( Pop INT PATH "$.Population"
          DEFAULT '999'
          ON ERROR DEFAULT
          '987' ON EMPTY))
         AS x1;
+-------+-----------------------+-------+
| name  | Info->>"$.Population" | Pop   |
+-------+-----------------------+-------+
| alpha | 100                   |   100 |
| beta  | fish                  |   999 |
| delta | 15                    |    15 |
| gamma | NULL                  |   987 |
+-------+-----------------------+-------+
4 rows in set, 1 warning (0.00 sec)
```

# Add Rigor To Your JSON Data

# JSON-Schema.org's work shown in MySQL - Use a template to define properties of a Key & their Values

The document properties are checked against this template and rejected if they do not pass muster!

```
set @s='{"type": "object",
    "properties": {
      "myage": {
      "type" : "number",
      "minimum": 28,
      "maximum": 99
   }
  }
}';
```

And here is our test document where we use a value for 'myage' what is between the minimum and the maximum.

```
set @d='{  "myage": 33}';
```

# Test

**Now we use JSON_SCHEMA_VALID() to test if the test document passes the validation test, with 1 or true as a pass and 0 or false as a fail.**

**select JSON_SCHEMA_VALID(@s,@d);**

```
+--------------------------+
| JSON_SCHEMA_VALID(@s,@d) |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

# REQUIRED Fields & Constraint Check

```
CREATE TABLE `testx` (
 `col` JSON,
 CONSTRAINT `myage_inRange`
 CHECK (JSON_SCHEMA_VALID('{"type": "object",
   "properties": {
    "myage": {
    "type" : "number",
    "minimum": 28,
    "maximum": 99
   }
  },"required": ["myage"]
 }', `col`) = 1)
);
```

```
insert into testx values('{"myage":27}');
ERROR 3819 (HY000):
Check constraint 'myage_inRange' is
violated.
insert into testx values('{"myage":97}');
Query OK, 1 row affected (0.02 sec)
```

## `JSON_SCHEMA_VALIDATION_REPORT(`*`schema,document`*`)`

Validates a JSON *`document`* against a JSON *`schema`*.The schema must be a valid JSON object, and the document must be a valid JSON document. Provided that these conditions are met, the function returns a report, as a JSON document, on the outcome of the validation. If the JSON document is considered valid according to the JSON Schema, the function returns a JSON object with one property `valid` having the value "true".

If the JSON document fails validation, the function returns a JSON object which includes the properties listed here:

- `valid`: Always "false" for a failed schema validation
- `reason`: A human-readable string containing the reason for the failure
- `schema-location`: A JSON pointer URI fragment identifier indicating where in the JSON schema the validation failed (see Note following this list)
- `document-location`: A JSON pointer URI fragment identifier indicating where in the JSON document the validation failed (see Note following this list)
- `schema-failed-keyword`: A string containing the name of the keyword or property in the JSON schema that was violated

# Simple Example 1 -- the exemplar, the new document, and the test

```
set @s='{"type": "object",
   "properties": {
     "myage": {
     "type" : "number",
     "minimum": 28,
     "maximum": 99
   }
 }
}';
```

```
set @d='{  "myage": 33}'
```

```
select JSON_SCHEMA_VALID(@s,@d);
+--------------------------+
| JSON_SCHEMA_VALID(@s,@d) |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

# Simple Example 2 -- the exemplar, the new document, and the test

```
set @s='{"type": "object",
    "properties": {
      "myage": {
      "type" : "number",
      "minimum": 28,
      "maximum": 99
   }
 }
}';
```

```
set @d='{  "myage": “foo”}'
```

```
select JSON_SCHEMA_VALID(@s,@d);
+--------------------------+
| JSON_SCHEMA_VALID(@s,@d) |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.00 sec)
```

# Simple Example 3 -- the exemplar, the new document, and the test

```
set @s='{"type": "object",
    "properties": {
      "myage": {
      "type" : "number",
      "minimum": 28,
      "maximum": 99
    }
  }
}';
```

```
set @d='{  "myage": 16}'
```

```
select JSON_SCHEMA_VALID(@s,@d);
+--------------------------+
| JSON_SCHEMA_VALID(@s,@d) |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.00 sec)
```

# JSON_SCHEMA_VALIDATION_REPORT()

```
select
JSON_PRETTY(JSON_SCHEMA_VALIDATION_REPORT(@s,@d))\G
*************************** 1. row ***************************
JSON_PRETTY(JSON_SCHEMA_VALIDATION_REPORT(@s,@d)): {
  "valid": false,
  "reason": "The JSON document location '#/myage' failed requirement
'minimum' at JSON Schema location '#/properties/myage'",
  "schema-location": "#/properties/myage",
  "document-location": "#/myage",
  "schema-failed-keyword": "minimum"
}
```

# REQUIRE Fields

```
CREATE TABLE `testx` (
 `col` JSON,
 CONSTRAINT `myage_inRange`
 CHECK (JSON_SCHEMA_VALID('{"type": "object",
   "properties": {
    "myage": {
    "type" : "number",
    "minimum": 28,
    "maximum": 99
   }
  },"required": ["myage"]
 }', `col`) = 1)
);
```

```
insert into testx values('{"myage":27}');
ERROR 3819 (HY000):
Check constraint 'myage_inRange' is
violated.
insert into testx values('{"myage":97}');
Query OK, 1 row affected (0.02 sec)
```

# Recommendations

(from the PostgreSQL manual)

Representing data as JSON can be considerably more flexible than the traditional relational data model, which is compelling in environments where requirements are fluid.

It is quite possible for both approaches to co-exist and complement each other within the same application.

However, even for applications where maximal flexibility is desired, it is still recommended that JSON documents have a somewhat fixed structure.

The structure is typically unenforced (though enforcing some business rules declaratively is possible), but having a predictable structure makes it easier to write queries that usefully summarize a set of “documents” (datums) in a table.

JSON data is subject to the same concurrency-control considerations as any other data type when stored in a table.

Although storing large documents is practicable, keep in mind that any update acquires a row-level lock on the whole row.

Consider limiting JSON documents to a manageable size in order to decrease lock contention among updating transactions.

Ideally, JSON documents should each represent an atomic datum that business rules dictate cannot reasonably be further subdivided into smaller datums that could be modified independently.

# Wrap up!

# Use JSON in your relational tables!

**For speed use relational columns.**

**PLAN your schemas by how you want to use the data.**

**Use JSON_TABLE() to temporarily make unstructured data structured for use with SQL.**

**Use generated columns to materialize JSON data into structured columns.**

**Do not use JSON as a 'junk drawer' or an excuse for your lack of planning.**

**DO NOT overly embed data in your JSON document - the more complex the path the higher the probability of an oops! Complication is not your friend down the road.**

**And do not use JSON to break general normalization rules or 'reinvent the wheel'.**

# Percona Live - https://www.percona.com/live/conferences

May 22–24 at the Denver Marriott Tech Center!

Databases run better with PERCONA

Open Source. Lower Costs. Performance & Security. No Vendor Lock-in.

# Thank You!

David.Stokes@Percona.com

@Stoker

Speakerdeck.com/Stoker

ORACLE DevLive | Level Up | MySQL Summit

# Please rate this session.

**Session ID – MS16**