FOSDEM 2021
MariaDB devroom

JSON Support in MariaDB

News, non-news, and the bigger picture

Sergei Petrunia
MariaDB developer

JSON Path

Non-news

JSON Path

- A lot of JSON functions accept “JSON Path” expressions
  - locate element(s) in JSON document
- Path language in MariaDB wasn’t documented
  - It’s more than “foo.bar.baz”
- Caution: there are many different JSON Path definitions on the web
  - SQL uses SQL:2016, “SQL/JSON Path language”

SQL/JSON Path language

```
path: [mode] $ [step]*
```

- mode is **lax** (the default) or **strict**
- **$** is the context element (root by default)
- followed by several steps.

Object member selection step

- Select a member

  `.name`

- Select all members

  `.*`

  - produces a *sequence* of values

```
{
  "name": "MariaDB"
  "version": "10.5"
}
```

`$.name` → `"MariaDB"`

`$.*` → `"MariaDB"` `"10.5"`

- In strict mode, the context must be an object, it must have a member with the specified name.
- lax mode “ignores” missing elements, “unwraps” 1-element arrays

Array element selection step

- Select array elements
  - one `[N]`
  - range `[N to M]`
  - last element `[last]`
  - list of ^^ `[N1,N2,...]`
  - all elements `[*]`

```
[
  10, "abc", {"x":0}
]
```

`$[0]` → `10`

`$[last]` → `{"x":0}`

`$[*]` → `10` `"abc"` `{"x":0}`

- Produces a *sequence* of elements
- Strict mode: indexes must be within bounds

Filters

- Filters elements in *sequence*

```
?(predicate)
```

- Predicate can have
  - AND/OR formulas (&&, ||)
  - comparisons
  - arithmetics
  - some functions
  - parameters passed from outside

```
[
  {
    "item":"Jeans",
    "color":"blue"
  },
  {
    "item":"Laptop",
    "color":"black"
  }
]
```

```
$[*]?(@.color=="black").item
```

→ `"Laptop"`

MariaDB and MySQL

- Support only lax mode
  - Not fully compliant: MySQL BUG#102233, MDEV-24573.
- Object member selection is supported
- Array selection: **[N]** is supported
  - MySQL also supports **[last]** and **[N to M]**
- Filters are not supported
  - expressions, arithmetics, functions, passing variables

Recursive search extension

- Task: find “**foo**” anywhere in the JSON document
- SQL/JSON Path doesn’t allow this
- Extension: **wildcard search step**
- Select all (direct and indirect) children:

```
step: **
```

- Example: find “price” anywhere: `$**.price`
- PostgreSQL also supports this, the syntax is `.**`

Usage example: Optimizer trace

- Optimizer Trace is JSON, log of query optimizer's actions
  - Deeply-nested
  - "Recursive", as SQL allows nesting of subqueries/CTEs/etc

```
select
  JSON_DETAILED(JSON_EXTRACT(trace, '$**.rows_estimation'))
from
  information_schema.optimizer_trace;
```

- Filters would be helpful: `$**.rows_estimation?(@table=="tbl1")`

JSON Path summary

- Language for pointing to node(s) in JSON document
  - SQL:2016, “SQL/JSON Path language”
- MariaDB and MySQL implement a subset
  - lax mode only
  - no support for filters (BAD)
  - array index – only [N]
    - MySQL also allows [last] and [M to N]
- MySQL and MariaDB have recursive-search extension (GOOD)

JSON Path in other databases

- PostgreSQL seems have the most compliant implementation
  - Supports filtering, strict/lax modes, etc.
- Other databases support different and [very] restrictive subsets.

JSON_TABLE

News

JSON_TABLE

- JSON_TABLE converts JSON input to a table
  - It is a “table function”
  - to be used in the FROM clause
- Introduced in SQL:2016
- Supported by Oracle DB, MySQL 8
- Under development in MariaDB
  - Trying to get it into 10.6

JSON_TABLE by example

- Start with a JSON document:

```
set @json_doc='
[
  {"name": "Laptop", "price": 1200},
  {"name": "Jeans",  "price": 60}
]';
```

- Use JSON_TABLE to produce tabular output:

```
select *
from
  JSON_TABLE(@json_doc,
             '$[*]' columns(name  varchar(32) path '$.name',
                            price int         path '$.price')
  ) as T
```

JSON_TABLE by example

- Start with a JSON document:

```
set @json_doc='
[
  {"name": "Laptop", "price": 1200},
  {"name": "Jeans",  "price": 60}
]';
```

- Use JSON_TABLE to produce tabular output:

```
select *
from
  JSON_TABLE(@json_doc,
             '$[*]' columns(name  varchar(32) path '$.name',
                            price int         path '$.price')
  ) as T
+--------+-------+
| name   | price |
+--------+-------+
| Laptop |  1200 |
| Jeans  |    60 |
+--------+-------+
```

JSON_TABLE syntax

Source document

```
select *
from
  JSON_TABLE(@json_doc,
             '$[*]' columns(name  varchar(32) path '$.name',
                            price int         path '$.price')
  ) as T
```

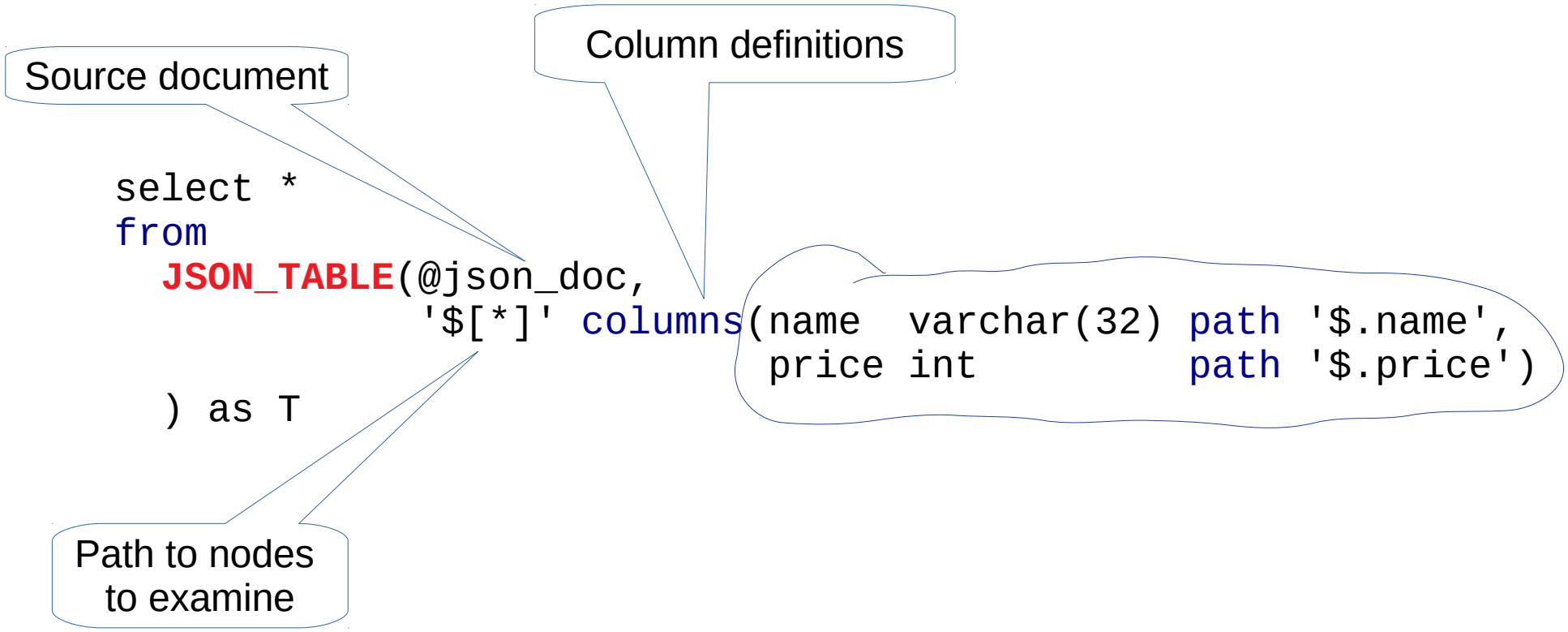
JSON_TABLE syntax

Source document

```
select *
from
  JSON_TABLE(@json_doc,
             '$[*]' columns(name  varchar(32) path '$.name',
                            price int         path '$.price')
  ) as T
```

Path to nodes to examine

JSON_TABLE syntax



JSON_TABLE syntax

Source document

Column definitions

```
select *
from
  JSON_TABLE(@json_doc,
             '$[*]' columns(name  varchar(32) path '$.name',
                            price int         path '$.price')

  ) as T
```

Path to nodes to examine

Column name

Path where to get the value from. The context item is the node being examined

Nested paths

```
set @json_doc='
[
  {"name": "Laptop",  "colors": ["black", "white", "red"] },
  {"name": "T-Shirt", "colors": ["yellow", "blue"] }
]';


select *
from
  JSON_TABLE(@json_doc,
             '$[*]'
             columns(name varchar(32) path '$.name',
                     nested path '$.colors[*]'
                       columns (
                         color varchar(32) path '$'
                       )
                     )
            ) as T
```

Nested paths

```
set @json_doc='
[
  {"name": "Laptop",  "colors": ["black", "white", "red"] },
  {"name": "T-Shirt", "colors": ["yellow", "blue"] }
]';
```

```
select *
from
  JSON_TABLE(@json_doc,
             '$[*]'
             columns(name varchar(32) path '$.name',
                     nested path '$.colors[*]'
                       columns (
                         color varchar(32) path '$'
                       )
                     )
           ) as T
```

```
+---------+--------+
| name    | color  |
+---------+--------+
| Laptop  | black  |
| Laptop  | white  |
| Laptop  | red    |
| T-Shirt | yellow |
| T-Shirt | blue   |
+---------+--------+
```

Multiple nested paths

- NESTED PATH can be nested
- Can have “sibling” NESTED PATHs
  - The standard allows to specify how to unnest (“the PLAN clause”)
  - The default way is “outer join” like:

```
{
  "name":   "T-Shirt",
  "colors": ["yellow", "blue"],
  "sizes":  ["Small", "Medium", "Large"]
}
```

```
+---------+--------+--------+
| name    | color  | size   |
+---------+--------+--------+
| T-Shirt | yellow | NULL   |
| T-Shirt | blue   | NULL   |
| T-Shirt | NULL   | Small  |
| T-Shirt | NULL   | Medium |
| T-Shirt | NULL   | Large  |
+---------+--------+--------+
```

  - MySQL (and soon MariaDB) only support “outer join”-like unnesting
    - “no PLAN clause support”.

Error handling

Handling missing values and/or conversion errors

```
columns(column_name type path '$path' [action on empty]
                                      [action on error])

action: NULL
        default 'string'
        error
```

- `on empty` is used when JSON element is missing
- `on error` is used on datatype conversion error or non-scalar JSON.
- Both MariaDB and MySQL support this

JSON_TABLE and joins

JSON_TABLE and joins

orders

1	[
	  {"name": "Laptop", "price": 1200},
	  {"name": "Jeans", "price": 60}
	]

2	[
	  {"name": "T-Shirt", "price": 10},
	  {"name": "Headphones", "price": 100}
	]

JSON_TABLE and joins

orders

1	[
	  {"name": "Laptop", "price": 1200},
	  {"name": "Jeans", "price": 60}
	]

2	[
	  {"name": "T-Shirt", "price": 10},
	  {"name": "Headphones", "price": 100}
	]

```
+----------+------------+-------+
| order_id | name       | price |
+----------+------------+-------+
|        1 | Laptop     |  1200 |
|        1 | Jeans      |    60 |
|        2 | T-Shirt    |    10 |
|        2 | Headphones |   100 |
+----------+------------+-------+
```

```
select
  orders.order_id,
  order_items.name,
  order_items.price
from
  orders,
  JSON_TABLE(orders.items_json,
             '$[*]' columns(name  varchar(32) path '$.name',
                            price int         path '$.price')
  ) as order_items
```

JSON_TABLE and joins

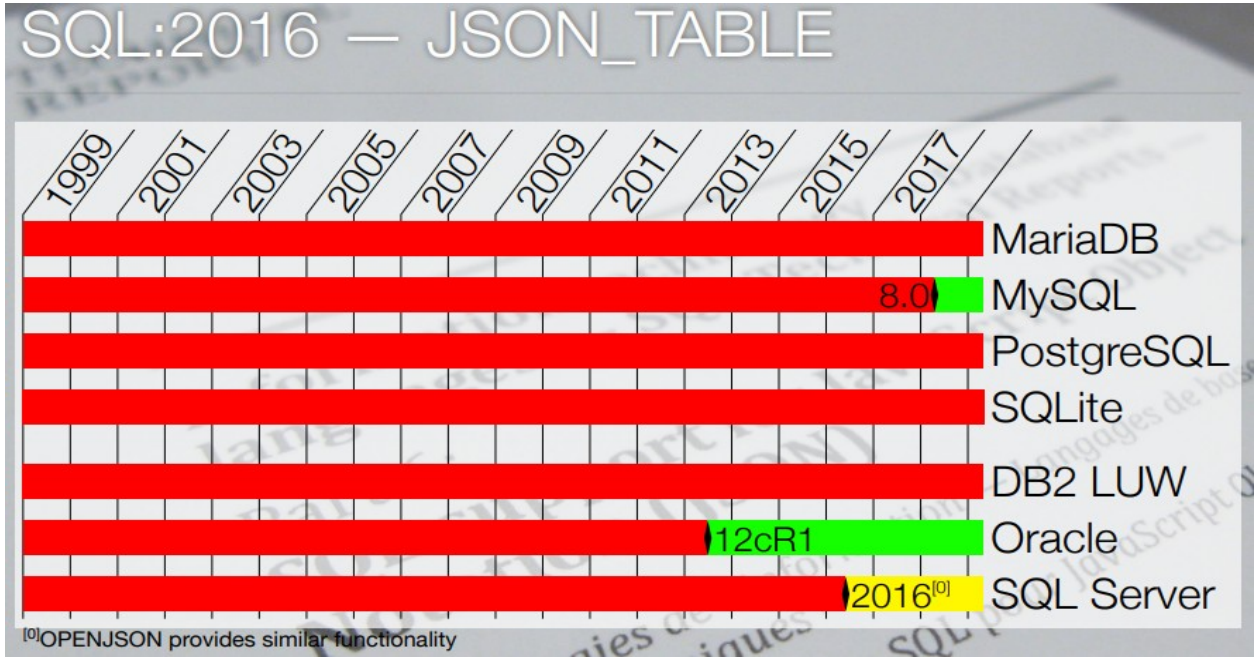
- JSON_TABLE’s argument can refer to other tables
- LATERAL-like semantics: contents of JSON_TABLE(...) depends on the parameter
- Allows to do “normalization” for contents of columns with JSON data

JSON_TABLE Summary

- A *table function* to convert JSON data to relational form
- Introduced in SQL:2016
  - The standard specifies a lot of features
- MySQL 8 implements a subset
  - PLAN clause is not supported
- MariaDB: MDEV-17399, under development
  - Will implement a subset very similar to MySQL

JSON_TABLE in other databases

- Quoting Markus Winand, https://modern-sql.com/slides/ModernSQL-2019-05-30.pdf:



- PostgreSQL have JSON_TABLE under development, too.

The takeaways

- SQL:2016 introduced JSON support
- MySQL 8 has implemented a subset of it
  - The subset is reasonably good
  - There are some extensions
- MariaDB is catching up
  - including the extensions

The low-hanging fruits

- JSON Path: [last], [N to M]
- JSON Path: filtering support
- Improved JSON_DETAILED function
  - It’s a JSON pretty-printer
  - Used in development with JSON
  - It works, but is fairly dumb
- All are great to start contributing
  - Contact us if interested.

Thanks!

Q & A