A geometry can be a valid geometry byte string (WKB value plus SRID) but geometrically invalid. For example, this polygon is geometrically invalid: `POLYGON((0 0, 0 0, 0 0, 0 0, 0 0))`

`ST_Validate()` can be used to filter out invalid geometry data, although at a cost. For applications that require more precise results not tainted by invalid data, this penalty may be worthwhile.

If the geometry argument is valid, it is returned as is, except that if an input `Polygon` or `MultiPolygon` has clockwise rings, those rings are reversed before checking for validity. If the geometry is valid, the value with the reversed rings is returned.

The only valid empty geometry is represented in the form of an empty geometry collection value. `ST_Validate()` returns it directly without further checks in this case.

`ST_Validate()` works only for the cartesian coordinate system and requires a geometry argument with an SRID of 0. An `ER_WRONG_ARGUMENTS` error occurs otherwise.

```
mysql> SET @ls1 = ST_GeomFromText('LINESTRING(0 0)');
mysql> SET @ls2 = ST_GeomFromText('LINESTRING(0 0, 1 1)');
mysql> SELECT ST_AsText(ST_Validate(@ls1));
+------------------------------+
| ST_AsText(ST_Validate(@ls1)) |
+------------------------------+
| NULL                         |
+------------------------------+
mysql> SELECT ST_AsText(ST_Validate(@ls2));
+------------------------------+
| ST_AsText(ST_Validate(@ls2)) |
+------------------------------+
| LINESTRING(0 0,1 1)          |
+------------------------------+
```

This function was added in MySQL 5.7.6.

13.16 JSON Functions

The functions described in this section perform operations on JSON values. For discussion of the `JSON` data type and additional examples showing how to use these functions, see Section 12.6, “The JSON Data Type”.

For functions that take a JSON argument, an error occurs if the argument is not a valid JSON value.

Unless otherwise indicated, the JSON functions were added in MySQL 5.7.8.

A set of spatial functions for operating on GeoJSON values is also available. See Section 13.15.11, “Spatial GeoJSON Functions”.

13.16.1 JSON Function Reference

Table 13.20 JSON Functions

Name	Description
`JSON_APPEND()`	Append data to JSON document
`JSON_ARRAY()`	Create JSON array
`JSON_ARRAY_APPEND()`	Append data to JSON document

Name	Description
`JSON_ARRAY_INSERT()`	Insert into JSON array
`->`	Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT().
`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
`->>`	Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()).
`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
`JSON_OBJECT()`	Create JSON object
`JSON_QUOTE()`	Quote JSON document
`JSON_REMOVE()`	Remove data from JSON document
`JSON_REPLACE()`	Replace values in JSON document
`JSON_SEARCH()`	Path to value within JSON document
`JSON_SET()`	Insert data into JSON document
`JSON_TYPE()`	Type of JSON value
`JSON_UNQUOTE()`	Unquote JSON value
`JSON_VALID()`	Whether JSON value is valid

13.16.2 Functions That Create JSON Values

The functions in this section compose JSON values from component elements.

- `JSON_ARRAY([val[, val] ...])`

  Evaluates a (possibly empty) list of values and returns a JSON array containing those values.

```
mysql> SELECT JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME());
+---------------------------------------------+
| JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME()) |
+---------------------------------------------+
| [1, "abc", null, true, "11:30:24.000000"]   |
+---------------------------------------------+
```

- `JSON_OBJECT([key, val[, key, val] ...])`

  Evaluates a (possibly empty) list of key/value pairs and returns a JSON object containing those pairs. An error occurs if any key name is `NULL` or the number of arguments is odd.

```
mysql> SELECT JSON_OBJECT('id', 87, 'name', 'carrot');
+-----------------------------------------+
| JSON_OBJECT('id', 87, 'name', 'carrot') |
+-----------------------------------------+
| {"id": 87, "name": "carrot"}            |
+-----------------------------------------+
```

- `JSON_QUOTE(json_val)`

  Quotes a string as a JSON value by wrapping it with double quote characters and escaping interior quote and other characters, then returning the result as a `utf8mb4` string. Returns `NULL` if the argument is `NULL`.

  This function is typically used to produce a valid JSON string literal for inclusion within a JSON document.

  Certain special characters are escaped with backslashes per the escape sequences shown in Table 13.21, “JSON_UNQUOTE() Special Character Escape Sequences”.

```
mysql> SELECT JSON_QUOTE('null'), JSON_QUOTE('"null"');
+--------------------+----------------------+
| JSON_QUOTE('null') | JSON_QUOTE('"null"') |
+--------------------+----------------------+
| "null"             | "\"null\""           |
+--------------------+----------------------+
mysql> SELECT JSON_QUOTE('[1, 2, 3]');
+-------------------------+
| JSON_QUOTE('[1, 2, 3]') |
+-------------------------+
| "[1, 2, 3]"             |
+-------------------------+
```

You can also obtain JSON values by casting values of other types to the `JSON` type using `CAST(value AS JSON)`; see Converting between JSON and non-JSON values, for more information.

13.16.3 Functions That Search JSON Values

The functions in this section perform search operations on JSON values to extract data from them, report whether data exists at a location within them, or report the path to data within them.

- `JSON_CONTAINS(json_doc, val[, path])`

  Returns 0 or 1 to indicate whether a specific value is contained in a target JSON document, or, if a *path* argument is given, at a specific path within the target document. Returns `NULL` if any argument is `NULL` or the path argument does not identify a section of the target document. An error occurs if either document argument is not a valid JSON document or the *path* argument is not a valid path expression or contains a `*` or `**` wildcard.

  To check only whether any data exists at the path, use `JSON_CONTAINS_PATH()` instead.

  The following rules define containment:

  - A candidate scalar is contained in a target scalar if and only if they are comparable and are equal. Two scalar values are comparable if they have the same `JSON_TYPE()` types, with the exception that values of types `INTEGER` and `DECIMAL` are also comparable to each other.

  - A candidate array is contained in a target array if and only if every element in the candidate is contained in some element of the target.

  - A candidate nonarray is contained in a target array if and only if the candidate is contained in some element of the target.

- A candidate object is contained in a target object if and only if for each key in the candidate there is a key with the same name in the target and the value associated with the candidate key is contained in the value associated with the target key.

Otherwise, the candidate value is not contained in the target document.

```
mysql> SET @j = '{"a": 1, "b": 2, "c": {"d": 4}}';
mysql> SET @j2 = '1';
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.a');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.a') |
+-------------------------------+
|                             1 |
+-------------------------------+
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.b');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.b') |
+-------------------------------+
|                             0 |
+-------------------------------+

mysql> SET @j2 = '{"d": 4}';
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.a');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.a') |
+-------------------------------+
|                             0 |
+-------------------------------+
mysql> SELECT JSON_CONTAINS(@j, @j2, '$.c');
+-------------------------------+
| JSON_CONTAINS(@j, @j2, '$.c') |
+-------------------------------+
|                             1 |
+-------------------------------+
```

- `JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ...)`

  Returns 0 or 1 to indicate whether a JSON document contains data at a given path or paths. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document, any *`path`* argument is not a valid path expression, or *`one_or_all`* is not `'one'` or `'all'`.

  To check for a specific value at a path, use `JSON_CONTAINS()` instead.

  The return value is 0 if no specified path exists within the document. Otherwise, the return value depends on the *`one_or_all`* argument:

  - `'one'`: 1 if at least one path exists within the document, 0 otherwise.

  - `'all'`: 1 if all paths exist within the document, 0 otherwise.

```
mysql> SET @j = '{"a": 1, "b": 2, "c": {"d": 4}}';
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.a', '$.e');
+---------------------------------------------+
| JSON_CONTAINS_PATH(@j, 'one', '$.a', '$.e') |
+---------------------------------------------+
|                                           1 |
+---------------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@j, 'all', '$.a', '$.e');
+---------------------------------------------+
| JSON_CONTAINS_PATH(@j, 'all', '$.a', '$.e') |
+---------------------------------------------+
|                                             0 |
+-----------------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.c.d');
+----------------------------------------+
| JSON_CONTAINS_PATH(@j, 'one', '$.c.d') |
+----------------------------------------+
|                                      1 |
+----------------------------------------+
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.a.d');
+----------------------------------------+
| JSON_CONTAINS_PATH(@j, 'one', '$.a.d') |
+----------------------------------------+
|                                      0 |
+----------------------------------------+
```

- `JSON_EXTRACT(json_doc, path[, path] ...)`

  Returns data from a JSON document, selected from the parts of the document matched by the *`path`* arguments. Returns `NULL` if any argument is `NULL` or no paths locate a value in the document. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression.

  The return value consists of all values matched by the *`path`* arguments. If it is possible that those arguments could return multiple values, the matched values are autowrapped as an array, in the order corresponding to the paths that produced them. Otherwise, the return value is the single matched value.

```
mysql> SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]');
+--------------------------------------------+
| JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]') |
+--------------------------------------------+
| 20                                         |
+--------------------------------------------+
mysql> SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]', '$[0]');
+----------------------------------------------------+
| JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]', '$[0]') |
+----------------------------------------------------+
| [20, 10]                                           |
+----------------------------------------------------+
mysql> SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[2][*]');
+-----------------------------------------------+
| JSON_EXTRACT('[10, 20, [30, 40]]', '$[2][*]') |
+-----------------------------------------------+
| [30, 40]                                      |
+-----------------------------------------------+
```

  MySQL 5.7.9 and later supports the `->` operator as shorthand for this function as used with 2 arguments where the left hand side is a `JSON` column identifier (not an expression) and the right hand side is the JSON path to be matched within the column.

- *`column->path`*

  In MySQL 5.7.9 and later, the `->` operator serves as an alias for the `JSON_EXTRACT()` function when used with two arguments, a column identifier on the left and a JSON path on the right that is evaluated against the JSON document (the column value). You can use such expressions in place of column identifiers wherever they occur in SQL statements.

  The two `SELECT` statements shown here produce the same output:

```
mysql> SELECT c, JSON_EXTRACT(c, "$.id"), g
     > FROM jemp
     > WHERE JSON_EXTRACT(c, "$.id") > 1
     > ORDER BY JSON_EXTRACT(c, "$.name");
+-------------------------------+-----------+------+
| c                             | c->"$.id" | g    |
+-------------------------------+-----------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 |
| {"id": "4", "name": "Betty"}  | "4"       |    4 |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 |
+-------------------------------+-----------+------+
3 rows in set (0.00 sec)

mysql> SELECT c, c->"$.id", g
     > FROM jemp
     > WHERE c->"$.id" > 1
     > ORDER BY c->"$.name";
+-------------------------------+-----------+------+
| c                             | c->"$.id" | g    |
+-------------------------------+-----------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 |
| {"id": "4", "name": "Betty"}  | "4"       |    4 |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 |
+-------------------------------+-----------+------+
3 rows in set (0.00 sec)
```

This functionality is not limited to `SELECT`, as shown here:

```
mysql> ALTER TABLE jemp ADD COLUMN n INT;
Query OK, 0 rows affected (0.68 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> UPDATE jemp SET n=1 WHERE c->"$.id" = "4";
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT c, c->"$.id", g, n
     > FROM jemp
     > WHERE JSON_EXTRACT(c, "$.id") > 1
     > ORDER BY c->"$.name";
+-------------------------------+-----------+------+------+
| c                             | c->"$.id" | g    | n    |
+-------------------------------+-----------+------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 | NULL |
| {"id": "4", "name": "Betty"}  | "4"       |    4 |    1 |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 | NULL |
+-------------------------------+-----------+------+------+
3 rows in set (0.00 sec)

mysql> DELETE FROM jemp WHERE c->"$.id" = "4";
Query OK, 1 row affected (0.04 sec)

mysql> SELECT c, c->"$.id", g, n
     > FROM jemp
     > WHERE JSON_EXTRACT(c, "$.id") > 1
     > ORDER BY c->"$.name";
+-------------------------------+-----------+------+------+
| c                             | c->"$.id" | g    | n    |
+-------------------------------+-----------+------+------+
| {"id": "3", "name": "Barney"} | "3"       |    3 | NULL |
| {"id": "2", "name": "Wilma"}  | "2"       |    2 | NULL |
+-------------------------------+-----------+------+------+
2 rows in set (0.00 sec)
```

(See Section 14.1.18.6, “Secondary Indexes and Generated Virtual Columns”, for the statements used to create and populate the table just shown.)

This also works with JSON array values, as shown here:

```
mysql> CREATE TABLE tj10 (a JSON, b INT);
Query OK, 0 rows affected (0.26 sec)

mysql> INSERT INTO tj10
     > VALUES ("[3,10,5,17,44]", 33), ("[3,10,5,17,[22,44,66]]", 0);
Query OK, 1 row affected (0.04 sec)

mysql> SELECT a->"$[4]" FROM tj10;
+--------------+
| a->"$[4]"    |
+--------------+
| 44           |
| [22, 44, 66] |
+--------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM tj10 WHERE a->"$[0]" = 3;
+------------------------------+------+
| a                            | b    |
+------------------------------+------+
| [3, 10, 5, 17, 44]           |   33 |
| [3, 10, 5, 17, [22, 44, 66]] |    0 |
+------------------------------+------+
2 rows in set (0.00 sec)
```

Nested arrays are supported. An expression using `->` evaluates as `NULL` if no matching key is found in the target JSON document, as shown here:

```
mysql> SELECT * FROM tj10 WHERE a->"$[4][1]" IS NOT NULL;
+------------------------------+------+
| a                            | b    |
+------------------------------+------+
| [3, 10, 5, 17, [22, 44, 66]] |    0 |
+------------------------------+------+

mysql> SELECT a->"$[4][1]" FROM tj10;
+--------------+
| a->"$[4][1]" |
+--------------+
| NULL         |
| 44           |
+--------------+
2 rows in set (0.00 sec)
```

This is the same behavior as seen in such cases when using `JSON_EXTRACT()`:

```
mysql> SELECT JSON_EXTRACT(a, "$[4][1]") FROM tj10;
+----------------------------+
| JSON_EXTRACT(a, "$[4][1]") |
+----------------------------+
| NULL                       |
| 44                         |
+----------------------------+
2 rows in set (0.00 sec)
```

- *`column->>path`*

  This is an improved, unquoting extraction operator available in MySQL 5.7.13 and later. Whereas the `->` operator simply extracts a value, the `->>` operator in addition unquotes the extracted result. In other words, given a `JSON` column value *`column`* and a path expression *`path`*, the following three expressions return the same value:

- `JSON_UNQUOTE( JSON_EXTRACT(column, path) )`
- `JSON_UNQUOTE(column -> path)`
- `column->>path`

The `->>` operator can be used wherever `JSON_UNQUOTE(JSON_EXTRACT())` would be allowed. This includes (but is not limited to) `SELECT` lists, `WHERE` and `HAVING` clauses, and `ORDER BY` and `GROUP BY` clauses.

The next few statements demonstrate some `->>` operator equivalences with other expressions in the `mysql` client:

```
mysql> SELECT * FROM jemp WHERE g > 2;
+-------------------------------+------+
| c                             | g    |
+-------------------------------+------+
| {"id": "3", "name": "Barney"} |    3 |
| {"id": "4", "name": "Betty"}  |    4 |
+-------------------------------+------+
2 rows in set (0.01 sec)

mysql> SELECT c->'$.name' AS name
    ->     FROM jemp WHERE g > 2;
+----------+
| name     |
+----------+
| "Barney" |
| "Betty"  |
+----------+
2 rows in set (0.00 sec)

mysql> SELECT JSON_UNQUOTE(c->'$.name') AS name
    ->     FROM jemp WHERE g > 2;
+--------+
| name   |
+--------+
| Barney |
| Betty  |
+--------+
2 rows in set (0.00 sec)

mysql> SELECT c->>'$.name' AS name
    ->     FROM jemp WHERE g > 2;
+--------+
| name   |
+--------+
| Barney |
| Betty  |
+--------+
2 rows in set (0.00 sec)
```

See Section 14.1.18.6, “Secondary Indexes and Generated Virtual Columns”, for the SQL statements used to create and populate the `jemp` table in the set of examples just shown.

This operator can also be used with JSON arrays, as shown here:

```
mysql> CREATE TABLE tj10 (a JSON, b INT);
Query OK, 0 rows affected (0.26 sec)

mysql> INSERT INTO tj10 VALUES
```

```
    ->     ('[3,10,5,"x",44]', 33),
    ->     ('[3,10,5,17,[22,"y",66]]', 0);
Query OK, 2 rows affected (0.04 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT a->"$[3]", a->"$[4][1]" FROM tj10;
+-----------+--------------+
| a->"$[3]" | a->"$[4][1]" |
+-----------+--------------+
| "x"       | NULL         |
| 17        | "y"          |
+-----------+--------------+
2 rows in set (0.00 sec)

mysql> SELECT a->>"$[3]", a->>"$[4][1]" FROM tj10;
+------------+---------------+
| a->>"$[3]" | a->>"$[4][1]" |
+------------+---------------+
| x          | NULL          |
| 17         | y             |
+------------+---------------+
2 rows in set (0.00 sec)
```

As with `->`, the `->>` operator is always expanded in the output of `EXPLAIN`, as the following example demonstrates:

```
mysql> EXPLAIN SELECT c->>'$.name' AS name
    ->     FROM jemp WHERE g > 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: jemp
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 5
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select
json_unquote(json_extract(`jtest`.`jemp`.`c`,'$.name')) AS `name` from
`jtest`.`jemp` where (`jtest`.`jemp`.`g` > 2)
1 row in set (0.00 sec)
```

This is similar to how MySQL expands the `->` operator in the same circumstances.

The `->>` operator was added in MySQL 5.7.13.

- `JSON_KEYS(json_doc[, path])`

  Returns the keys from the top-level value of a JSON object as a JSON array, or, if a *`path`* argument is given, the top-level keys from the selected path. Returns `NULL` if any argument is `NULL`, the *`json_doc`* argument is not an object, or *`path`*, if given, does not locate an object. An error occurs if the *`json_doc`* argument is not a valid JSON document or the *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

The result array is empty if the selected object is empty. If the top-level value has nested subobjects, the return value does not include keys from those subobjects.

```
mysql> SELECT JSON_KEYS('{"a": 1, "b": {"c": 30}}');
+---------------------------------------+
| JSON_KEYS('{"a": 1, "b": {"c": 30}}') |
+---------------------------------------+
| ["a", "b"]                            |
+---------------------------------------+
mysql> SELECT JSON_KEYS('{"a": 1, "b": {"c": 30}}', '$.b');
+----------------------------------------------+
| JSON_KEYS('{"a": 1, "b": {"c": 30}}', '$.b') |
+----------------------------------------------+
| ["c"]                                        |
+----------------------------------------------+
```

- `JSON_SEARCH(json_doc, one_or_all, search_str[, escape_char[, path] ...])`

  Returns the path to the given string within a JSON document. Returns `NULL` if any of the *`json_doc`*, *`search_str`*, or *`path`* arguments are `NULL`; no *`path`* exists within the document; or *`search_str`* is not found. An error occurs if the *`json_doc`* argument is not a valid JSON document, any *`path`* argument is not a valid path expression, *`one_or_all`* is not `'one'` or `'all'`, or *`escape_char`* is not a constant expression.

  The *`one_or_all`* argument affects the search as follows:

  - `'one'`: The search terminates after the first match and returns one path string. It is undefined which match is considered first.

  - `'all'`: The search returns all matching path strings such that no duplicate paths are included. If there are multiple strings, they are autowrapped as an array. The order of the array elements is undefined.

  Within the *`search_str`* search string argument, the `%` and `_` characters work as for the `LIKE` operator: `%` matches any number of characters (including zero characters), and `_` matches exactly one character.

  To specify a literal `%` or `_` character in the search string, precede it by the escape character. The default is `\` if the *`escape_char`* argument is missing or `NULL`. Otherwise, *`escape_char`* must be a constant that is empty or one character.

  For more information about matching and escape character behavior, see the description of `LIKE` in Section 13.5.1, “String Comparison Functions”. For escape character handling, a difference from the `LIKE` behavior is that the escape character for `JSON_SEARCH()` must evaluate to a constant at compile time, not just at execution time. For example, if `JSON_SEARCH()` is used in a prepared statement and the *`escape_char`* argument is supplied using a `?` parameter, the parameter value might be constant at execution time, but is not at compile time.

```
mysql> SET @j = '["abc", [{"k": "10"}, "def"], {"x":"abc"}, {"y":"bcd"}]';

mysql> SELECT JSON_SEARCH(@j, 'one', 'abc');
+-------------------------------+
| JSON_SEARCH(@j, 'one', 'abc') |
+-------------------------------+
| "$[0]"                        |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', 'abc');
+-------------------------------+
| JSON_SEARCH(@j, 'all', 'abc') |
+--------------------------------+
| ["$[0]", "$[2].x"]             |
+--------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', 'ghi');
+-------------------------------+
| JSON_SEARCH(@j, 'all', 'ghi') |
+-------------------------------+
| NULL                          |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10');
+------------------------------+
| JSON_SEARCH(@j, 'all', '10') |
+------------------------------+
| "$[1][0].k"                  |
+------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$');
+-----------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$') |
+-----------------------------------------+
| "$[1][0].k"                             |
+-----------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[*]');
+--------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[*]') |
+--------------------------------------------+
| "$[1][0].k"                                |
+--------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$**.k');
+---------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$**.k') |
+---------------------------------------------+
| "$[1][0].k"                                 |
+---------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[*][0].k');
+-------------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[*][0].k') |
+-------------------------------------------------+
| "$[1][0].k"                                     |
+-------------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[1]');
+--------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[1]') |
+--------------------------------------------+
| "$[1][0].k"                                |
+--------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '10', NULL, '$[1][0]');
+-----------------------------------------------+
| JSON_SEARCH(@j, 'all', '10', NULL, '$[1][0]') |
+-----------------------------------------------+
| "$[1][0].k"                                   |
+-----------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', 'abc', NULL, '$[2]');
+---------------------------------------------+
| JSON_SEARCH(@j, 'all', 'abc', NULL, '$[2]') |
+---------------------------------------------+
| "$[2].x"                                    |
+---------------------------------------------+
mysql> SELECT JSON_SEARCH(@j, 'all', '%a%');
+-------------------------------+
| JSON_SEARCH(@j, 'all', '%a%') |
+-------------------------------+
| ["$[0]", "$[2].x"]            |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%');
+-------------------------------+
| JSON_SEARCH(@j, 'all', '%b%') |
+-------------------------------+
| ["$[0]", "$[2].x", "$[3].y"]  |
+-------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', NULL, '$[0]');
+---------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', NULL, '$[0]') |
+---------------------------------------------+
| "$[0]"                                      |
+---------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', NULL, '$[2]');
+---------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', NULL, '$[2]') |
+---------------------------------------------+
| "$[2].x"                                    |
+---------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', NULL, '$[1]');
+---------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', NULL, '$[1]') |
+---------------------------------------------+
| NULL                                        |
+---------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', '', '$[1]');
+-------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', '', '$[1]') |
+-------------------------------------------+
| NULL                                      |
+-------------------------------------------+

mysql> SELECT JSON_SEARCH(@j, 'all', '%b%', '', '$[3]');
+-------------------------------------------+
| JSON_SEARCH(@j, 'all', '%b%', '', '$[3]') |
+-------------------------------------------+
| "$[3].y"                                  |
+-------------------------------------------+
```

For more information about the JSON path syntax supported by MySQL, including rules governing the wildcard operators `*` and `**`, see Section 13.16.6, “JSON Path Syntax”.

13.16.4 Functions That Modify JSON Values

The functions in this section modify JSON values and return the result.

- `JSON_APPEND(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

  Appends values to the end of the indicated arrays within a JSON document and returns the result. This function was renamed to `JSON_ARRAY_APPEND()` in MySQL 5.7.9.

- `JSON_ARRAY_APPEND(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

Appends values to the end of the indicated arrays within a JSON document and returns the result. Returns NULL if any argument is NULL. An error occurs if the *json_doc* argument is not a valid JSON document or any *path* argument is not a valid path expression or contains a * or ** wildcard.

The path/value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

If a path selects a scalar or object value, that value is autowrapped within an array and the new value is added to that array. Pairs for which the path does not identify any value in the JSON document are ignored.

```
mysql> SET @j = '["a", ["b", "c"], "d"]';
mysql> SELECT JSON_ARRAY_APPEND(@j, '$[1]', 1);
+----------------------------------+
| JSON_ARRAY_APPEND(@j, '$[1]', 1) |
+----------------------------------+
| ["a", ["b", "c", 1], "d"]        |
+----------------------------------+
mysql> SELECT JSON_ARRAY_APPEND(@j, '$[0]', 2);
+----------------------------------+
| JSON_ARRAY_APPEND(@j, '$[0]', 2) |
+----------------------------------+
| [["a", 2], ["b", "c"], "d"]      |
+----------------------------------+
mysql> SELECT JSON_ARRAY_APPEND(@j, '$[1][0]', 3);
+-------------------------------------+
| JSON_ARRAY_APPEND(@j, '$[1][0]', 3) |
+-------------------------------------+
| ["a", [["b", 3], "c"], "d"]         |
+-------------------------------------+

mysql> SET @j = '{"a": 1, "b": [2, 3], "c": 4}';
mysql> SELECT JSON_ARRAY_APPEND(@j, '$.b', 'x');
+------------------------------------+
| JSON_ARRAY_APPEND(@j, '$.b', 'x')  |
+------------------------------------+
| {"a": 1, "b": [2, 3, "x"], "c": 4} |
+------------------------------------+
mysql> SELECT JSON_ARRAY_APPEND(@j, '$.c', 'y');
+--------------------------------------+
| JSON_ARRAY_APPEND(@j, '$.c', 'y')    |
+--------------------------------------+
| {"a": 1, "b": [2, 3], "c": [4, "y"]} |
+--------------------------------------+

mysql> SET @j = '{"a": 1}';
mysql> SELECT JSON_ARRAY_APPEND(@j, '$', 'z');
+---------------------------------+
| JSON_ARRAY_APPEND(@j, '$', 'z') |
+---------------------------------+
| [{"a": 1}, "z"]                 |
+---------------------------------+
```

- JSON_ARRAY_INSERT(*json_doc*, *path*, *val*[, *path*, *val*] ...)

  Updates a JSON document, inserting into an array within the document and returning the modified document. Returns NULL if any argument is NULL. An error occurs if the *json_doc* argument is not a valid JSON document or any *path* argument is not a valid path expression or contains a * or ** wildcard or does not end with an array element identifier.

The path/value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

Pairs for which the path does not identify any array in the JSON document are ignored. If a path identifies an array element, the corresponding value is inserted at that element position, shifting any following values to the right. If a path identifies an array position past the end of an array, the value is inserted at the end of the array.

```
mysql> SET @j = '["a", {"b": [1, 2]}, [3, 4]]';
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[1]', 'x');
+------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[1]', 'x') |
+------------------------------------+
| ["a", "x", {"b": [1, 2]}, [3, 4]]  |
+------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[100]', 'x');
+--------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[100]', 'x') |
+--------------------------------------+
| ["a", {"b": [1, 2]}, [3, 4], "x"]    |
+--------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[1].b[0]', 'x');
+-----------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[1].b[0]', 'x') |
+-----------------------------------------+
| ["a", {"b": ["x", 1, 2]}, [3, 4]]       |
+-----------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[2][1]', 'y');
+---------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[2][1]', 'y') |
+---------------------------------------+
| ["a", {"b": [1, 2]}, [3, "y", 4]]     |
+---------------------------------------+
mysql> SELECT JSON_ARRAY_INSERT(@j, '$[0]', 'x', '$[2][1]', 'y');
+----------------------------------------------------+
| JSON_ARRAY_INSERT(@j, '$[0]', 'x', '$[2][1]', 'y') |
+----------------------------------------------------+
| ["x", "a", {"b": [1, 2]}, [3, 4]]                  |
+----------------------------------------------------+
```

Earlier modifications affect the positions of the following elements in the array, so subsequent paths in the same `JSON_ARRAY_INSERT()` call should take this into account. In the final example, the second path inserts nothing because the path no longer matches anything after the first insert.

- `JSON_INSERT(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

  Inserts data into a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The path/value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  A path/value pair for an existing path in the document is ignored and does not overwrite the existing document value. A path/value pair for a nonexisting path in the document adds the value to the document if the path identifies one of these types of values:

  - A member not present in an existing object. The member is added to the object and associated with the new value.

- A position past the end of an existing array. The array is extended with the new value. If the existing value is not an array, it is autowrapped as an array, then extended with the new value.

Otherwise, a path/value pair for a nonexisting path in the document is ignored and has no effect.

For a comparison of `JSON_INSERT()`, `JSON_REPLACE()`, and `JSON_SET()`, see the discussion of `JSON_SET()`.

```
mysql> SET @j = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]');
+----------------------------------------------------+
| JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]') |
+----------------------------------------------------+
| {"a": 1, "b": [2, 3], "c": "[true, false]"}        |
+----------------------------------------------------+
```

- `JSON_MERGE(json_doc, json_doc[, json_doc] ...)`

  Merges two or more JSON documents and returns the merged result. Returns `NULL` if any argument is `NULL`. An error occurs if any argument is not a valid JSON document.

  Merging takes place according to the following rules. For additional information, see Normalization, Merging, and Autowrapping of JSON Values.

  - Adjacent arrays are merged to a single array.
  - Adjacent objects are merged to a single object.
  - A scalar value is autowrapped as an array and merged as an array.
  - An adjacent array and object are merged by autowrapping the object as an array and merging the two arrays.

```
mysql> SELECT JSON_MERGE('[1, 2]', '[true, false]');
+---------------------------------------+
| JSON_MERGE('[1, 2]', '[true, false]') |
+---------------------------------------+
| [1, 2, true, false]                   |
+---------------------------------------+
mysql> SELECT JSON_MERGE('{"name": "x"}', '{"id": 47}');
+-------------------------------------------+
| JSON_MERGE('{"name": "x"}', '{"id": 47}') |
+-------------------------------------------+
| {"id": 47, "name": "x"}                   |
+-------------------------------------------+
mysql> SELECT JSON_MERGE('1', 'true');
+-------------------------+
| JSON_MERGE('1', 'true') |
+-------------------------+
| [1, true]               |
+-------------------------+
mysql> SELECT JSON_MERGE('[1, 2]', '{"id": 47}');
+------------------------------------+
| JSON_MERGE('[1, 2]', '{"id": 47}') |
+------------------------------------+
| [1, 2, {"id": 47}]                 |
+------------------------------------+
```

- `JSON_REMOVE(json_doc, path[, path] ...)`

Removes data from a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or is `$` or contains a `*` or `**` wildcard.

The *`path`* arguments are evaluated left to right. The document produced by evaluating one path becomes the new value against which the next path is evaluated.

It is not an error if the element to be removed does not exist in the document; in that case, the path does not affect the document.

```
mysql> SET @j = '["a", ["b", "c"], "d"]';
mysql> SELECT JSON_REMOVE(@j, '$[1]');
+-------------------------+
| JSON_REMOVE(@j, '$[1]') |
+-------------------------+
| ["a", "d"]              |
+-------------------------+
```

- `JSON_REPLACE(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

  Replaces existing values in a JSON document and returns the result. Returns `NULL` if any argument is `NULL`. An error occurs if the *`json_doc`* argument is not a valid JSON document or any *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The path/value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  A path/value pair for an existing path in the document overwrites the existing document value with the new value. A path/value pair for a nonexisting path in the document is ignored and has no effect.

  For a comparison of `JSON_INSERT()`, `JSON_REPLACE()`, and `JSON_SET()`, see the discussion of `JSON_SET()`.

```
mysql> SET @j = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]');
+-----------------------------------------------------+
| JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]') |
+-----------------------------------------------------+
| {"a": 10, "b": [2, 3]}                              |
+-----------------------------------------------------+
```

- `JSON_SET(`*`json_doc`*`, `*`path`*`, `*`val`*`[, `*`path`*`, `*`val`*`] ...)`

  Inserts or updates data in a JSON document and returns the result. Returns `NULL` if any argument is `NULL` or *`path`*, if given, does not locate an object. An error occurs if the *`json_doc`* argument is not a valid JSON document or the *`path`* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The path/value pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

  A path/value pair for an existing path in the document overwrites the existing document value with the new value. A path/value pair for a nonexisting path in the document adds the value to the document if the path identifies one of these types of values:

  - A member not present in an existing object. The member is added to the object and associated with the new value.

- A position past the end of an existing array. The array is extended with the new value. If the existing value is not an array, it is autowrapped as an array, then extended with the new value.

Otherwise, a path/value pair for a nonexisting path in the document is ignored and has no effect.

The `JSON_SET()`, `JSON_INSERT()`, and `JSON_REPLACE()` functions are related:

- `JSON_SET()` replaces existing values and adds nonexisting values.
- `JSON_INSERT()` inserts values without replacing existing values.
- `JSON_REPLACE()` replaces *only* existing values.

The following examples illustrate these differences, using one path that does exist in the document (`$.a`) and another that does not exist (`$.c`):

```
mysql> SET @j = '{ "a": 1, "b": [2, 3]}';
mysql> SELECT JSON_SET(@j, '$.a', 10, '$.c', '[true, false]');
+--------------------------------------------------+
| JSON_SET(@j, '$.a', 10, '$.c', '[true, false]') |
+--------------------------------------------------+
| {"a": 10, "b": [2, 3], "c": "[true, false]"}    |
+--------------------------------------------------+
mysql> SELECT JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]');
+------------------------------------------------------+
| JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]') |
+------------------------------------------------------+
| {"a": 1, "b": [2, 3], "c": "[true, false]"}         |
+------------------------------------------------------+
mysql> SELECT JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]');
+-------------------------------------------------------+
| JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]') |
+-------------------------------------------------------+
| {"a": 10, "b": [2, 3]}                               |
+-------------------------------------------------------+
```

- `JSON_UNQUOTE(`*`val`*`)`

Unquotes JSON value and returns the result as a `utf8mb4` string. Returns `NULL` if the argument is `NULL`. An error occurs if the value starts and ends with double quotes but is not a valid JSON string literal.

Within a string, certain sequences have special meaning unless the `NO_BACKSLASH_ESCAPES` SQL mode is enabled. Each of these sequences begins with a backslash (`\`), known as the *escape character*. MySQL recognizes the escape sequences shown in Table 13.21, "JSON_UNQUOTE() Special Character Escape Sequences". For all other escape sequences, backslash is ignored. That is, the escaped character is interpreted as if it was not escaped. For example, `\x` is just `x`. These sequences are case sensitive. For example, `\b` is interpreted as a backspace, but `\B` is interpreted as `B`.

Table 13.21 JSON_UNQUOTE() Special Character Escape Sequences

Escape Sequence	Character Represented by Sequence
`\"`	A double quote (`"`) character
`\b`	A backspace character
`\f`	A formfeed character

Escape Sequence	Character Represented by Sequence
`\n`	A newline (linefeed) character
`\r`	A carriage return character
`\t`	A tab character
`\\`	A backslash (`\`) character
`\uXXXX`	UTF-8 bytes for Unicode value *`XXXX`*

Two simple examples of the use of this function are shown here:

```
mysql> SET @j = '"abc"';
mysql> SELECT @j, JSON_UNQUOTE(@j);
+-------+------------------+
| @j    | JSON_UNQUOTE(@j) |
+-------+------------------+
| "abc" | abc              |
+-------+------------------+
mysql> SET @j = '[1, 2, 3]';
mysql> SELECT @j, JSON_UNQUOTE(@j);
+-----------+------------------+
| @j        | JSON_UNQUOTE(@j) |
+-----------+------------------+
| [1, 2, 3] | [1, 2, 3]        |
+-----------+------------------+
```

The following set of examples shows how `JSON_UNQUOTE` handles escapes with `NO_BACKSLASH_ESCAPES` disabled and enabled:

```
mysql> SELECT @@sql_mode;
+------------+
| @@sql_mode |
+------------+
|            |
+------------+

mysql> SELECT JSON_UNQUOTE('"\\t\\u0032"');
+------------------------------+
| JSON_UNQUOTE('"\\t\\u0032"') |
+------------------------------+
|       2                           |
+------------------------------+

mysql> SET @@sql_mode = 'NO_BACKSLASH_ESCAPES';
mysql> SELECT JSON_UNQUOTE('"\\t\\u0032"');
+------------------------------+
| JSON_UNQUOTE('"\\t\\u0032"') |
+------------------------------+
| \t\u0032                     |
+------------------------------+

mysql> SELECT JSON_UNQUOTE('"\t\u0032"');
+----------------------------+
| JSON_UNQUOTE('"\t\u0032"') |
+----------------------------+
|       2                         |
+----------------------------+
```

13.16.5 Functions That Return JSON Value Attributes

The functions in this section return attributes of JSON values.

- `JSON_DEPTH(json_doc)`

  Returns the maximum depth of a JSON document. Returns `NULL` if the argument is `NULL`. An error occurs if the argument is not a valid JSON document.

  An empty array, empty object, or scalar value has depth 1. A nonempty array containing only elements of depth 1 or nonempty object containing only member values of depth 1 has depth 2. Otherwise, a JSON document has depth greater than 2.

```
mysql> SELECT JSON_DEPTH('{}'), JSON_DEPTH('[]'), JSON_DEPTH('true');
+------------------+------------------+--------------------+
| JSON_DEPTH('{}') | JSON_DEPTH('[]') | JSON_DEPTH('true') |
+------------------+------------------+--------------------+
|                1 |                1 |                  1 |
+------------------+------------------+--------------------+
mysql> SELECT JSON_DEPTH('[10, 20]'), JSON_DEPTH('[[], {}]');
+------------------------+------------------------+
| JSON_DEPTH('[10, 20]') | JSON_DEPTH('[[], {}]') |
+------------------------+------------------------+
|                      2 |                      2 |
+------------------------+------------------------+
mysql> SELECT JSON_DEPTH('[10, {"a": 20}]');
+-------------------------------+
| JSON_DEPTH('[10, {"a": 20}]') |
+-------------------------------+
|                             3 |
+-------------------------------+
```

- `JSON_LENGTH(json_doc[, path])`

  Returns the length of JSON document, or, if a *path* argument is given, the length of the value within the document identified by the path. Returns `NULL` if any argument is `NULL` or the *path* argument does not identify a value in the document. An error occurs if the *json_doc* argument is not a valid JSON document or the *path* argument is not a valid path expression or contains a `*` or `**` wildcard.

  The length of a document is determined as follows:

  - The length of a scalar is 1.
  - The length of an array is the number of array elements.
  - The length of an object is the number of object members.
  - The length does not count the length of nested arrays or objects.

```
mysql> SELECT JSON_LENGTH('[1, 2, {"a": 3}]');
+---------------------------------+
| JSON_LENGTH('[1, 2, {"a": 3}]') |
+---------------------------------+
|                               3 |
+---------------------------------+
mysql> SELECT JSON_LENGTH('{"a": 1, "b": {"c": 30}}');
+-----------------------------------------+
| JSON_LENGTH('{"a": 1, "b": {"c": 30}}') |
+-----------------------------------------+
|                                       2 |
+-----------------------------------------+
mysql> SELECT JSON_LENGTH('{"a": 1, "b": {"c": 30}}', '$.b');
+------------------------------------------------+
| JSON_LENGTH('{"a": 1, "b": {"c": 30}}', '$.b') |
+------------------------------------------------+
|                                              1 |
+------------------------------------------------+
```

- `JSON_TYPE(json_val)`

  Returns a `utf8mb4` string indicating the type of a JSON value:

```
mysql> SET @j = '{"a": [10, true]}';
mysql> SELECT JSON_TYPE(@j);
+---------------+
| JSON_TYPE(@j) |
+---------------+
| OBJECT        |
+---------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@j, '$.a'));
+------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@j, '$.a')) |
+------------------------------------+
| ARRAY                              |
+------------------------------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@j, '$.a[0]'));
+---------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@j, '$.a[0]')) |
+---------------------------------------+
| INTEGER                               |
+---------------------------------------+
mysql> SELECT JSON_TYPE(JSON_EXTRACT(@j, '$.a[1]'));
+---------------------------------------+
| JSON_TYPE(JSON_EXTRACT(@j, '$.a[1]')) |
+---------------------------------------+
| BOOLEAN                               |
+---------------------------------------+
```

  `JSON_TYPE()` returns `NULL` if the argument is `NULL`:

```
mysql> SELECT JSON_TYPE(NULL);
+-----------------+
| JSON_TYPE(NULL) |
+-----------------+
| NULL            |
+-----------------+
```

  An error occurs if the argument is not a valid JSON value:

```
mysql> SELECT JSON_TYPE(1);
ERROR 3146 (22032): Invalid data type for JSON data in argument 1
to function json_type; a JSON string or JSON type is required.
```

  For a non-`NULL`, non-error result, the following list describes the possible `JSON_TYPE()` return values:

  - Purely JSON types:
    - `OBJECT`: JSON objects
    - `ARRAY`: JSON arrays

    - `BOOLEAN`: The JSON true and false literals
    - `NULL`: The JSON null literal
  - Numeric types:
    - `INTEGER`: MySQL `TINYINT`, `SMALLINT`, `MEDIUMINT` and `INT` and `BIGINT` scalars
    - `DOUBLE`: MySQL `DOUBLE` `FLOAT` scalars
    - `DECIMAL`: MySQL `DECIMAL` and `NUMERIC` scalars
  - Temporal types:
    - `DATETIME`: MySQL `DATETIME` and `TIMESTAMP` scalars
    - `DATE`: MySQL `DATE` scalars
    - `TIME`: MySQL `TIME` scalars
  - String types:
    - `STRING`: MySQL `utf8` character type scalars: `CHAR`, `VARCHAR`, `TEXT`, `ENUM`, and `SET`
  - Binary types:
    - `BLOB`: MySQL binary type scalars: `BINARY`, `VARBINARY`, `BLOB`
    - `BIT`: MySQL `BIT` scalars
  - All other types:
    - `OPAQUE` (raw bits)

- `JSON_VALID(`*`val`*`)`

  Returns 0 or 1 to indicate whether a value is a valid JSON document. Returns `NULL` if the argument is `NULL`.

```
mysql> SELECT JSON_VALID('{"a": 1}');
+------------------------+
| JSON_VALID('{"a": 1}') |
+------------------------+
|                      1 |
+------------------------+
mysql> SELECT JSON_VALID('hello'), JSON_VALID('"hello"');
+---------------------+-----------------------+
| JSON_VALID('hello') | JSON_VALID('"hello"') |
+---------------------+-----------------------+
|                   0 |                     1 |
+---------------------+-----------------------+
```

13.16.6 JSON Path Syntax

Many of the functions described in previous sections require a path expression in order to identify a specific element in a JSON document. A path consists of the path's scope followed by one or more path legs. For paths used in MySQL JSON functions, the scope is always the document being searched or otherwise operated on, represented by a leading `$` character. Path legs are separated by period characters

(`.`). Cells in arrays are represented by `[N]`, where *N* is a non-negative integer. Names of keys must be double-quoted strings or valid ECMAScript identifiers (see `http://www.ecma-international.org/ecma-262/5.1/#sec-7.6`). Path expressions, like JSON text, should be encoded using the `ascii`, `utf8`, or `utf8mb4` character sets. Other character encodings are implicitly coerced to `utf8mb4`. The complete syntax is shown here:

```
pathExpression:
    scope[(pathLeg)*]

pathLeg:
    member | arrayLocation | doubleAsterisk

member:
    period ( keyName | asterisk )

arrayLocation:
    leftBracket ( nonNegativeInteger | asterisk ) rightBracket

keyName:
    ESIdentifier | doubleQuotedString

doubleAsterisk:
    '**'

period:
    '.'

asterisk:
    '*'

leftBracket:
    '['

rightBracket:
    ']'
```

As noted previously, in MySQL, the scope of the path is always the document being operated on, represented as `$`. You can use `'$'` as a synonynm for the document in JSON path expressions.

Note

Some implementations support column references for scopes of JSON paths; currently, MySQL does not support these.

The wildcard `*` and `**` tokens are used as follows:

- `.*` represents the values of all members in the object.
- `[*]` represents the values of all cells in the array.
- `[prefix]**suffix` represents all paths beginning with *prefix* and ending with *suffix*. *prefix* is optional, while *suffix* is required; in other words, a path may not end in `**`.

  In addition, a path may not contain the sequence `***`.

For path syntax examples, see the descriptions of the various JSON fuinctions that take paths as arguments, such as `JSON_CONTAINS_PATH()` and `JSON_REPLACE()`. For examples which include the use of the `*` and `**` wildcards, see the description of the `JSON_SEARCH()` function.

13.17 Functions Used with Global Transaction IDs