11.5 The JSON Data Type

MySQL supports a native `JSON` data type defined by RFC 7159 that enables efficient access to data in JSON (JavaScript Object Notation) documents. The `JSON` data type provides these advantages over storing JSON-format strings in a string column:

- Automatic validation of JSON documents stored in `JSON` columns. Invalid documents produce an error.
- Optimized storage format. JSON documents stored in `JSON` columns are converted to an internal format that permits quick read access to document elements. When the server later must read a JSON value stored in this binary format, the value need not be parsed from a text representation. The binary format is structured to enable the server to look up subobjects or nested values directly by key or array index without reading all values before or after them in the document.

MySQL 8.0 also supports the *JSON Merge Patch* format defined in RFC 7396, using the `JSON_MERGE_PATCH()` function. See the description of this function, as well as Normalization, Merging, and Autowrapping of JSON Values, for examples and further information.

> **Note**
>
> This discussion uses `JSON` in monotype to indicate specifically the JSON data type and “JSON” in regular font to indicate JSON data in general.

The space required to store a `JSON` document is roughly the same as for `LONGBLOB` or `LONGTEXT`; see Section 11.7, “Data Type Storage Requirements”, for more information. It is important to keep in mind that the size of any JSON document stored in a `JSON` column is limited to the value of the `max_allowed_packet` system variable. (When the server is manipulating a JSON value internally in memory, it can be larger than this; the limit applies when the server stores it.) You can obtain the amount of space required to store a JSON document using the `JSON_STORAGE_SIZE()` function; note that for a `JSON` column, the storage size—and thus the value returned by this function—is that used by the column prior to any partial updates that may have been performed on it (see the discussion of the JSON partial update optimization later in this section).

Prior to MySQL 8.0.13, a `JSON` column cannot have a non-`NULL` default value.

Along with the `JSON` data type, a set of SQL functions is available to enable operations on JSON values, such as creation, manipulation, and searching. The following discussion shows examples of these operations. For details about individual functions, see Section 12.17, “JSON Functions”.

A set of spatial functions for operating on GeoJSON values is also available. See Section 12.16.11, “Spatial GeoJSON Functions”.

`JSON` columns, like columns of other binary types, are not indexed directly; instead, you can create an index on a generated column that extracts a scalar value from the `JSON` column. See Indexing a Generated Column to Provide a JSON Column Index, for a detailed example.

The MySQL optimizer also looks for compatible indexes on virtual columns that match JSON expressions.

In MySQL 8.0.17 and later, the `InnoDB` storage engine supports multi-valued indexes on JSON arrays. See Multi-Valued Indexes.

MySQL NDB Cluster 8.0 supports `JSON` columns and MySQL JSON functions, including creation of an index on a column generated from a `JSON` column as a workaround for being unable to index a `JSON` column. A maximum of 3 `JSON` columns per `NDB` table is supported.

Partial Updates of JSON Values

In MySQL 8.0, the optimizer can perform a partial, in-place update of a `JSON` column instead of removing the old document and writing the new document in its entirety to the column. This optimization can be performed for an update that meets the following conditions:

- The column being updated was declared as `JSON`.
- The `UPDATE` statement uses any of the three functions `JSON_SET()`, `JSON_REPLACE()`, or `JSON_REMOVE()` to update the column. A direct assignment of the column value (for example, `UPDATE mytable SET jcol = '{"a": 10, "b": 25}'`) cannot be performed as a partial update.

Updates of multiple `JSON` columns in a single `UPDATE` statement can be optimized in this fashion; MySQL can perform partial updates of only those columns whose values are updated using the three functions just listed.

- The input column and the target column must be the same column; a statement such as `UPDATE mytable SET jcol1 = JSON_SET(jcol2, '$.a', 100)` cannot be performed as a partial update.

  The update can use nested calls to any of the functions listed in the previous item, in any combination, as long as the input and target columns are the same.

- All changes replace existing array or object values with new ones, and do not add any new elements to the parent object or array.

- The value being replaced must be at least as large as the replacement value. In other words, the new value cannot be any larger than the old one.

  A possible exception to this requirement occurs when a previous partial update has left sufficient space for the larger value. You can use the function `JSON_STORAGE_FREE()` see how much space has been freed by any partial updates of a `JSON` column.

Such partial updates can be written to the binary log using a compact format that saves space; this can be enabled by setting the `binlog_row_value_options` system variable to `PARTIAL_JSON`.

It is important to distinguish the partial update of a `JSON` column value stored in a table from writing the partial update of a row to the binary log. It is possible for the complete update of a `JSON` column to be recorded in the binary log as a partial update. This can happen when either (or both) of the last two conditions from the previous list is not met but the other conditions are satisfied.

See also the description of `binlog_row_value_options`.

The next few sections provide basic information regarding the creation and manipulation of JSON values.

Creating JSON Values

A JSON array contains a list of values separated by commas and enclosed within `[` and `]` characters:

```
["abc", 10, null, true, false]
```

A JSON object contains a set of key-value pairs separated by commas and enclosed within `{` and `}` characters:

```
{"k1": "value", "k2": 10}
```

As the examples illustrate, JSON arrays and objects can contain scalar values that are strings or numbers, the JSON null literal, or the JSON boolean true or false literals. Keys in JSON objects must be strings. Temporal (date, time, or datetime) scalar values are also permitted:

```
["12:18:29.000000", "2015-07-29", "2015-07-29 12:18:29.000000"]
```

Nesting is permitted within JSON array elements and JSON object key values:

```
[99, {"id": "HK500", "cost": 75.99}, ["hot", "cold"]]
{"k1": "value", "k2": [10, 20]}
```

You can also obtain JSON values from a number of functions supplied by MySQL for this purpose (see Section 12.17.2, “Functions That Create JSON Values”) as well as by casting values of other types to the `JSON` type using `CAST(`***`value`***` AS JSON)` (see Converting between JSON and non-JSON values). The next several paragraphs describe how MySQL handles JSON values provided as input.

In MySQL, JSON values are written as strings. MySQL parses any string used in a context that requires a JSON value, and produces an error if it is not valid as JSON. These contexts include inserting a value into a column that has the `JSON` data type and passing an argument to a function that expects a JSON value (usually shown as ***`json_doc`*** or ***`json_val`*** in the documentation for MySQL JSON functions), as the following examples demonstrate:

- Attempting to insert a value into a `JSON` column succeeds if the value is a valid JSON value, but fails if it is not:

  ```
  mysql> CREATE TABLE t1 (jdoc JSON);
  Query OK, 0 rows affected (0.20 sec)

  mysql> INSERT INTO t1 VALUES('{"key1": "value1", "key2": "value2"}');
  Query OK, 1 row affected (0.01 sec)

  mysql> INSERT INTO t1 VALUES('[1, 2,');
  ERROR 3140 (22032) at line 2: Invalid JSON text:
  "Invalid value." at position 6 in value (or column) '[1, 2,'.
  ```

  Positions for “at position ***`N`***” in such error messages are 0-based, but should be considered rough indications of where the problem in a value actually occurs.

- The `JSON_TYPE()` function expects a JSON argument and attempts to parse it into a JSON value. It returns the value's JSON type if it is valid and produces an error otherwise:

```
mysql> SELECT JSON_TYPE('["a", "b", 1]');
+----------------------------+
| JSON_TYPE('["a", "b", 1]') |
+----------------------------+
| ARRAY                      |
+----------------------------+

mysql> SELECT JSON_TYPE('"hello"');
+----------------------+
| JSON_TYPE('"hello"') |
+----------------------+
| STRING               |
+----------------------+

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

MySQL handles strings used in JSON context using the `utf8mb4` character set and `utf8mb4_bin` collation. Strings in other character sets are converted to `utf8mb4` as necessary. (For strings in the `ascii` or `utf8mb3` character sets, no conversion is needed because `ascii` and `utf8mb3` are subsets of `utf8mb4`.)

As an alternative to writing JSON values using literal strings, functions exist for composing JSON values from component elements. `JSON_ARRAY()` takes a (possibly empty) list of values and returns a JSON array containing those values:

```
mysql> SELECT JSON_ARRAY('a', 1, NOW());
+----------------------------------------+
| JSON_ARRAY('a', 1, NOW())              |
+----------------------------------------+
| ["a", 1, "2015-07-27 09:43:47.000000"] |
+----------------------------------------+
```

`JSON_OBJECT()` takes a (possibly empty) list of key-value pairs and returns a JSON object containing those pairs:

```
mysql> SELECT JSON_OBJECT('key1', 1, 'key2', 'abc');
+---------------------------------------+
| JSON_OBJECT('key1', 1, 'key2', 'abc') |
+---------------------------------------+
| {"key1": 1, "key2": "abc"}             |
+----------------------------------------+
```

`JSON_MERGE_PRESERVE()` takes two or more JSON documents and returns the combined result:

```
mysql> SELECT JSON_MERGE_PRESERVE('["a", 1]', '{"key": "value"}');
+-----------------------------------------------------+
| JSON_MERGE_PRESERVE('["a", 1]', '{"key": "value"}') |
+-----------------------------------------------------+
| ["a", 1, {"key": "value"}]                          |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

For information about the merging rules, see Normalization, Merging, and Autowrapping of JSON Values.

(MySQL 8.0.3 and later also support `JSON_MERGE_PATCH()`, which has somewhat different behavior. See JSON_MERGE_PATCH() compared with JSON_MERGE_PRESERVE(), for information about the differences between these two functions.)

JSON values can be assigned to user-defined variables:

```
mysql> SET @j = JSON_OBJECT('key', 'value');
mysql> SELECT @j;
+------------------+
| @j               |
+------------------+
| {"key": "value"} |
+------------------+
```

However, user-defined variables cannot be of `JSON` data type, so although `@j` in the preceding example looks like a JSON value and has the same character set and collation as a JSON value, it does *not* have the `JSON` data type. Instead, the result from `JSON_OBJECT()` is converted to a string when assigned to the variable.

Strings produced by converting JSON values have a character set of `utf8mb4` and a collation of `utf8mb4_bin`:

```
mysql> SELECT CHARSET(@j), COLLATION(@j);
+-------------+---------------+
| CHARSET(@j) | COLLATION(@j) |
+-------------+---------------+
| utf8mb4      | utf8mb4_bin    |
+--------------+----------------+
```

Because `utf8mb4_bin` is a binary collation, comparison of JSON values is case-sensitive.

```
mysql> SELECT JSON_ARRAY('x') = JSON_ARRAY('X');
+-----------------------------------+
| JSON_ARRAY('x') = JSON_ARRAY('X') |
+-----------------------------------+
|                                 0 |
+-----------------------------------+
```

Case sensitivity also applies to the JSON `null`, `true`, and `false` literals, which always must be written in lowercase:

```
mysql> SELECT JSON_VALID('null'), JSON_VALID('Null'), JSON_VALID('NULL');
+--------------------+--------------------+--------------------+
| JSON_VALID('null') | JSON_VALID('Null') | JSON_VALID('NULL') |
+--------------------+--------------------+--------------------+
|                  1 |                  0 |                  0 |
+--------------------+--------------------+--------------------+

mysql> SELECT CAST('null' AS JSON);
+----------------------+
| CAST('null' AS JSON) |
+----------------------+
| null                 |
+----------------------+
1 row in set (0.00 sec)

mysql> SELECT CAST('NULL' AS JSON);
ERROR 3141 (22032): Invalid JSON text in argument 1 to function cast_as_json:
"Invalid value." at position 0 in 'NULL'.
```

Case sensitivity of the JSON literals differs from that of the SQL `NULL`, `TRUE`, and `FALSE` literals, which can be written in any lettercase:

```
mysql> SELECT ISNULL(null), ISNULL(Null), ISNULL(NULL);
+--------------+--------------+--------------+
| ISNULL(null) | ISNULL(Null) | ISNULL(NULL) |
+--------------+--------------+--------------+
|            1 |            1 |            1 |
+--------------+--------------+--------------+
```

Sometimes it may be necessary or desirable to insert quote characters (`"` or `'`) into a JSON document. Assume for this example that you want to insert some JSON objects containing strings representing sentences that state some facts about MySQL, each paired with an appropriate keyword, into a table created using the SQL statement shown here:

```
mysql> CREATE TABLE facts (sentence JSON);
```

Among these keyword-sentence pairs is this one:

```
mascot: The MySQL mascot is a dolphin named "Sakila".
```

One way to insert this as a JSON object into the `facts` table is to use the MySQL `JSON_OBJECT()` function. In this case, you must escape each quote character using a backslash, as shown here:

```
mysql> INSERT INTO facts VALUES
     >   (JSON_OBJECT("mascot", "Our mascot is a dolphin named \"Sakila\"."));
```

This does not work in the same way if you insert the value as a JSON object literal, in which case, you must use the double backslash escape sequence, like this:

```
mysql> INSERT INTO facts VALUES
     >   ('{"mascot": "Our mascot is a dolphin named \\"Sakila\\"."}');
```

Using the double backslash keeps MySQL from performing escape sequence processing, and instead causes it to pass the string literal to the storage engine for processing. After inserting the JSON object in either of the ways just shown, you can see that the backslashes are present in the JSON column value by doing a simple `SELECT`, like this:

```
mysql> SELECT sentence FROM facts;
+---------------------------------------------------------+
| sentence                                                |
+---------------------------------------------------------+
| {"mascot": "Our mascot is a dolphin named \"Sakila\"."} |
+---------------------------------------------------------+
```

To look up this particular sentence employing `mascot` as the key, you can use the column-path operator `->`, as shown here:

```
mysql> SELECT col->"$.mascot" FROM qtest;
+---------------------------------------------+
| col->"$.mascot"                             |
+---------------------------------------------+
| "Our mascot is a dolphin named \"Sakila\"." |
+---------------------------------------------+
1 row in set (0.00 sec)
```

This leaves the backslashes intact, along with the surrounding quote marks. To display the desired value using `mascot` as the key, but without including the surrounding quote marks or any escapes, use the inline path operator `->>`, like this:

```
mysql> SELECT sentence->>"$.mascot" FROM facts;
+-----------------------------------------+
| sentence->>"$.mascot"                   |
+-----------------------------------------+
| Our mascot is a dolphin named "Sakila". |
+-----------------------------------------+
```

Note

The previous example does not work as shown if the `NO_BACKSLASH_ESCAPES` server SQL mode is enabled. If this mode is set, a single backslash instead of double backslashes can be used to insert the JSON object literal, and the backslashes are preserved. If you use the `JSON_OBJECT()` function when performing the insert and this mode is set, you must alternate single and double quotes, like this:

```
mysql> INSERT INTO facts VALUES
     > (JSON_OBJECT('mascot', 'Our mascot is a dolphin named "Sa
```

See the description of the `JSON_UNQUOTE()` function for more information about the effects of this mode on escaped characters in JSON values.

Normalization, Merging, and Autowrapping of JSON Values

When a string is parsed and found to be a valid JSON document, it is also normalized. This means that members with keys that duplicate a key found later in the document, reading from left to right, are discarded. The object value produced by the following `JSON_OBJECT()` call includes only the second `key1` element because that key name occurs earlier in the value, as shown here:

```
mysql> SELECT JSON_OBJECT('key1', 1, 'key2', 'abc', 'key1', 'def');
+------------------------------------------------------+
| JSON_OBJECT('key1', 1, 'key2', 'abc', 'key1', 'def') |
+------------------------------------------------------+
| {"key1": "def", "key2": "abc"}                       |
+------------------------------------------------------+
```

Normalization is also performed when values are inserted into JSON columns, as shown here:

```
mysql> CREATE TABLE t1 (c1 JSON);

mysql> INSERT INTO t1 VALUES
     >     ('{"x": 17, "x": "red"}'),
     >     ('{"x": 17, "x": "red", "x": [3, 5, 7]}');

mysql> SELECT c1 FROM t1;
+------------------+
| c1               |
+------------------+
| {"x": "red"}     |
| {"x": [3, 5, 7]} |
+------------------+
```

This “last duplicate key wins” behavior is suggested by RFC 7159 and is implemented by most JavaScript parsers. (Bug #86866, Bug #26369555)

In versions of MySQL prior to 8.0.3, members with keys that duplicated a key found earlier in the document were discarded. The object value produced by the following `JSON_OBJECT()` call does not include the second `key1` element because that key name occurs earlier in the value:

```
mysql> SELECT JSON_OBJECT('key1', 1, 'key2', 'abc', 'key1', 'def');
+------------------------------------------------------+
| JSON_OBJECT('key1', 1, 'key2', 'abc', 'key1', 'def') |
+------------------------------------------------------+
| {"key1": 1, "key2": "abc"}                           |
+------------------------------------------------------+
```

Prior to MySQL 8.0.3, this “first duplicate key wins” normalization was also performed when inserting values into JSON columns.

```
mysql> CREATE TABLE t1 (c1 JSON);

mysql> INSERT INTO t1 VALUES
     >     ('{"x": 17, "x": "red"}'),
     >     ('{"x": 17, "x": "red", "x": [3, 5, 7]}');

mysql> SELECT c1 FROM t1;
+-----------+
| c1        |
+-----------+
| {"x": 17} |
| {"x": 17} |
+-----------+
```

MySQL also discards extra whitespace between keys, values, or elements in the original JSON document, and leaves (or inserts, when necessary) a single space following each comma (`,`) or colon (`:`) when displaying it. This is done to enhance readability.

MySQL functions that produce JSON values (see Section 12.17.2, “Functions That Create JSON Values”) always return normalized values.

To make lookups more efficient, MySQL also sorts the keys of a JSON object. *You should be aware that the result of this ordering is subject to change and not guaranteed to be consistent across releases*.

Merging JSON Values

Two merging algorithms are supported in MySQL 8.0.3 (and later), implemented by the functions `JSON_MERGE_PRESERVE()` and `JSON_MERGE_PATCH()`. These differ in how they handle duplicate keys: `JSON_MERGE_PRESERVE()` retains values for duplicate keys, while `JSON_MERGE_PATCH()` discards all but the last value. The next few paragraphs explain how each of these two functions handles the merging of different combinations of JSON documents (that is, of objects and arrays).

> **Note**
>
> `JSON_MERGE_PRESERVE()` is the same as the `JSON_MERGE()` function found in previous versions of MySQL (renamed in MySQL 8.0.3). `JSON_MERGE()` is still supported as an alias for `JSON_MERGE_PRESERVE()` in MySQL 8.0, but is deprecated and subject to removal in a future release.

Merging arrays. In contexts that combine multiple arrays, the arrays are merged into a single array. `JSON_MERGE_PRESERVE()` does this by concatenating arrays named later to the end of the first array. `JSON_MERGE_PATCH()` considers each argument as an array consisting of a single element (thus having 0 as its index) and then applies “last duplicate key wins” logic to select only the last argument. You can compare the results shown by this query:

```
mysql> SELECT
    ->   JSON_MERGE_PRESERVE('[1, 2]', '["a", "b", "c"]', '[true, false]') AS Pres
    ->   JSON_MERGE_PATCH('[1, 2]', '["a", "b", "c"]', '[true, false]') AS Patch\(
*************************** 1. row ***************************
Preserve: [1, 2, "a", "b", "c", true, false]
   Patch: [true, false]
```

Multiple objects when merged produce a single object. `JSON_MERGE_PRESERVE()` handles multiple objects having the same key by combining all unique values for that key in an array; this array is then used as the value for that key in the result. `JSON_MERGE_PATCH()` discards values for which duplicate keys are found, working from left to right, so that the result contains only the last value for that key. The following query illustrates the difference in the results for the duplicate key a:

```
mysql> SELECT
    ->   JSON_MERGE_PRESERVE('{"a": 1, "b": 2}', '{"c": 3, "a": 4}', '{"c": 5, "d'
    ->   JSON_MERGE_PATCH('{"a": 3, "b": 2}', '{"c": 3, "a": 4}', '{"c": 5, "d": 3
*************************** 1. row ***************************
Preserve: {"a": [1, 4], "b": 2, "c": [3, 5], "d": 3}
   Patch: {"a": 4, "b": 2, "c": 5, "d": 3}
```

Nonarray values used in a context that requires an array value are autowrapped: The value is surrounded by `[` and `]` characters to convert it to an array. In the following statement, each argument is autowrapped as an array (`[1]`, `[2]`). These are then merged to produce a single result array; as in the previous two cases, `JSON_MERGE_PRESERVE()` combines values having the same key while `JSON_MERGE_PATCH()` discards values for all duplicate keys except the last, as shown here:

```
mysql> SELECT
      ->   JSON_MERGE_PRESERVE('1', '2') AS Preserve,
      ->   JSON_MERGE_PATCH('1', '2') AS Patch\G
*************************** 1. row ***************************
Preserve: [1, 2]
   Patch: 2
```

Array and object values are merged by autowrapping the object as an array and merging the arrays by combining values or by “last duplicate key wins” according to the choice of merging function (`JSON_MERGE_PRESERVE()` or `JSON_MERGE_PATCH()`, respectively), as can be seen in this example:

```
mysql> SELECT
      ->   JSON_MERGE_PRESERVE('[10, 20]', '{"a": "x", "b": "y"}') AS Preserve,
      ->   JSON_MERGE_PATCH('[10, 20]', '{"a": "x", "b": "y"}') AS Patch\G
*************************** 1. row ***************************
```

```
Preserve: [10, 20, {"a": "x", "b": "y"}]
   Patch: {"a": "x", "b": "y"}
```

Searching and Modifying JSON Values

A JSON path expression selects a value within a JSON document.

Path expressions are useful with functions that extract parts of or modify a JSON document, to specify where within that document to operate. For example, the following query extracts from a JSON document the value of the member with the `name` key:

```
mysql> SELECT JSON_EXTRACT('{"id": 14, "name": "Aztalan"}', '$.name');
+---------------------------------------------------------+
| JSON_EXTRACT('{"id": 14, "name": "Aztalan"}', '$.name') |
+---------------------------------------------------------+
| "Aztalan"                                               |
+---------------------------------------------------------+
```

Path syntax uses a leading `$` character to represent the JSON document under consideration, optionally followed by selectors that indicate successively more specific parts of the document:

- A period followed by a key name names the member in an object with the given key. The key name must be specified within double quotation marks if the name without quotes is not legal within path expressions (for example, if it contains a space).
- `[`***`N`***`]` appended to a ***`path`*** that selects an array names the value at position ***`N`*** within the array. Array positions are integers beginning with zero. If ***`path`*** does not select an array value, ***`path`***[0] evaluates to the same value as ***`path`***:

```
mysql> SELECT JSON_SET('"x"', '$[0]', 'a');
+------------------------------+
| JSON_SET('"x"', '$[0]', 'a') |
+------------------------------+
| "a"                          |
+------------------------------+
1 row in set (0.00 sec)
```

- `[`***`M`***` to `***`N`***`]` specifies a subset or range of array values starting with the value at position ***`M`***, and ending with the value at position ***`N`***.

  `last` is supported as a synonym for the index of the rightmost array element. Relative addressing of array elements is also supported. If ***`path`*** does not select an array value, ***`path`***[last] evaluates to the same value as ***`path`***, as shown later in this section (see Rightmost array element).

- Paths can contain `*` or `**` wildcards:
  - `.[*]` evaluates to the values of all members in a JSON object.
  - `[*]` evaluates to the values of all elements in a JSON array.
  - ***`prefix`***`**`***`suffix`*** evaluates to all paths that begin with the named prefix and end with the named suffix.
- A path that does not exist in the document (evaluates to nonexistent data) evaluates to `NULL`.

Let `$` refer to this JSON array with three elements:

```
[3, {"a": [5, 6], "b": 10}, [99, 100]]
```

Then:

- `$[0]` evaluates to `3`.
- `$[1]` evaluates to `{"a": [5, 6], "b": 10}`.
- `$[2]` evaluates to `[99, 100]`.
- `$[3]` evaluates to `NULL` (it refers to the fourth array element, which does not exist).

Because `$[1]` and `$[2]` evaluate to nonscalar values, they can be used as the basis for more-specific path expressions that select nested values. Examples:

- `$[1].a` evaluates to `[5, 6]`.
- `$[1].a[1]` evaluates to `6`.
- `$[1].b` evaluates to `10`.
- `$[2][0]` evaluates to `99`.

As mentioned previously, path components that name keys must be quoted if the unquoted key name is not legal in path expressions. Let `$` refer to this value:

```
{"a fish": "shark", "a bird": "sparrow"}
```

The keys both contain a space and must be quoted:

- `$."a fish"` evaluates to `shark`.

- `$."a bird"` evaluates to `sparrow`.

Paths that use wildcards evaluate to an array that can contain multiple values:

```
mysql> SELECT JSON_EXTRACT('{"a": 1, "b": 2, "c": [3, 4, 5]}', '$.*');
+---------------------------------------------------------+
| JSON_EXTRACT('{"a": 1, "b": 2, "c": [3, 4, 5]}', '$.*') |
+---------------------------------------------------------+
| [1, 2, [3, 4, 5]]                                       |
+---------------------------------------------------------+
mysql> SELECT JSON_EXTRACT('{"a": 1, "b": 2, "c": [3, 4, 5]}', '$.c[*]');
+------------------------------------------------------------+
| JSON_EXTRACT('{"a": 1, "b": 2, "c": [3, 4, 5]}', '$.c[*]') |
+------------------------------------------------------------+
| [3, 4, 5]                                                  |
+------------------------------------------------------------+
```

In the following example, the path `$**.b` evaluates to multiple paths (`$.a.b` and `$.c.b`) and produces an array of the matching path values:

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

Ranges from JSON arrays. You can use ranges with the `to` keyword to specify subsets of JSON arrays. For example, `$[1 to 3]` includes the second, third, and fourth elements of an array, as shown here:

```
mysql> SELECT JSON_EXTRACT('[1, 2, 3, 4, 5]', '$[1 to 3]');
+------------------------------------------------+
| JSON_EXTRACT('[1, 2, 3, 4, 5]', '$[1 to 3]') |
+------------------------------------------------+
| [2, 3, 4]                                      |
+------------------------------------------------+
1 row in set (0.00 sec)
```

The syntax is ***`M`*** `to` ***`N`***, where ***`M`*** and ***`N`*** are, respectively, the first and last indexes of a range of elements from a JSON array. ***`N`*** must be greater than ***`M`***; ***`M`*** must be greater than or equal to 0. Array elements are indexed beginning with 0.

You can use ranges in contexts where wildcards are supported.

Rightmost array element. The `last` keyword is supported as a synonym for the index of the last element in an array. Expressions of the form `last - N` can be used for relative addressing, and within range definitions, like this:

```
mysql> SELECT JSON_EXTRACT('[1, 2, 3, 4, 5]', '$[last-3 to last-1]');
+--------------------------------------------------------+
| JSON_EXTRACT('[1, 2, 3, 4, 5]', '$[last-3 to last-1]') |
+--------------------------------------------------------+
| [2, 3, 4]                                              |
+--------------------------------------------------------+
1 row in set (0.01 sec)
```

If the path is evaluated against a value that is not an array, the result of the evaluation is the same as if the value had been wrapped in a single-element array:

```
mysql> SELECT JSON_REPLACE('"Sakila"', '$[last]', 10);
+-----------------------------------------+
| JSON_REPLACE('"Sakila"', '$[last]', 10) |
+-----------------------------------------+
| 10                                      |
+-----------------------------------------+
1 row in set (0.00 sec)
```

You can use ***`column`*****`->`*****`path`*** with a JSON column identifier and JSON path expression as a synonym for `JSON_EXTRACT(`***`column`***`, `***`path`***`)`. See Section 12.17.3, “Functions That Search JSON Values”, for more information. See also Indexing a Generated Column to Provide a JSON Column Index.

Some functions take an existing JSON document, modify it in some way, and return the resulting modified document. Path expressions indicate where in the document to make changes. For example, the `JSON_SET()`, `JSON_INSERT()`, and `JSON_REPLACE()` functions each take a JSON document, plus one or more path-value pairs that describe where to modify the document and the values to use. The functions differ in how they handle existing and nonexisting values within the document.

Consider this document:

```
mysql> SET @j = '["a", {"b": [true, false]}, [10, 20]]';
```

`JSON_SET()` replaces values for paths that exist and adds values for paths that do not exist:.

```
mysql> SELECT JSON_SET(@j, '$[1].b[0]', 1, '$[2][2]', 2);
+--------------------------------------------+
| JSON_SET(@j, '$[1].b[0]', 1, '$[2][2]', 2) |
+--------------------------------------------+
| ["a", {"b": [1, false]}, [10, 20, 2]]      |
+--------------------------------------------+
```

In this case, the path `$[1].b[0]` selects an existing value (`true`), which is replaced with the value following the path argument (`1`). The path `$[2][2]` does not exist, so the corresponding value (`2`) is added to the value selected by `$[2]`.

`JSON_INSERT()` adds new values but does not replace existing values:

```
mysql> SELECT JSON_INSERT(@j, '$[1].b[0]', 1, '$[2][2]', 2);
+-----------------------------------------------+
| JSON_INSERT(@j, '$[1].b[0]', 1, '$[2][2]', 2) |
+-----------------------------------------------+
| ["a", {"b": [true, false]}, [10, 20, 2]]      |
+-----------------------------------------------+
```

`JSON_REPLACE()` replaces existing values and ignores new values:

```
mysql> SELECT JSON_REPLACE(@j, '$[1].b[0]', 1, '$[2][2]', 2);
+------------------------------------------------+
| JSON_REPLACE(@j, '$[1].b[0]', 1, '$[2][2]', 2) |
+------------------------------------------------+
| ["a", {"b": [1, false]}, [10, 20]]             |
+------------------------------------------------+
```

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.

`JSON_REMOVE()` takes a JSON document and one or more paths that specify values to be removed from the document. The return value is the original document minus the values selected by paths that exist within the document:

```
mysql> SELECT JSON_REMOVE(@j, '$[2]', '$[1].b[1]', '$[1].b[1]');
+---------------------------------------------------+
| JSON_REMOVE(@j, '$[2]', '$[1].b[1]', '$[1].b[1]') |
+---------------------------------------------------+
| ["a", {"b": [true]}]                              |
+---------------------------------------------------+
```

The paths have these effects:

- `$[2]` matches `[10, 20]` and removes it.

- The first instance of `$[1].b[1]` matches `false` in the `b` element and removes it.
- The second instance of `$[1].b[1]` matches nothing: That element has already been removed, the path no longer exists, and has no effect.

JSON Path Syntax

Many of the JSON functions supported by MySQL and described elsewhere in this Manual (see Section 12.17, "JSON Functions") 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 *Identifier Names and Identifiers*, in the *ECMAScript Language Specification*). Path expressions, like JSON text, should be encoded using the `ascii`, `utf8mb3`, or `utf8mb4` character set. 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 synonym for the document in JSON path expressions.

> **Note**
>
> Some implementations support column references for scopes of JSON paths; MySQL 8.0 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 functions that take paths as arguments, such as `JSON_CONTAINS_PATH()`, `JSON_SET()`, and `JSON_REPLACE()`. For examples which include the use of the `*` and `**` wildcards, see the description of the `JSON_SEARCH()` function.

MySQL 8.0 also supports range notation for subsets of JSON arrays using the `to` keyword (such as `$[2 to 10]`), as well as the `last` keyword as a synonym for the rightmost element of an array. See Searching and Modifying JSON Values, for more information and examples.

Comparison and Ordering of JSON Values

JSON values can be compared using the =, <, <=, >, >=, <>, `!=`, and <=> operators.

The following comparison operators and functions are not yet supported with JSON values:

- `BETWEEN`
- `IN()`
- `GREATEST()`
- `LEAST()`

A workaround for the comparison operators and functions just listed is to cast JSON values to a native MySQL numeric or string data type so they have a consistent non-JSON scalar type.

Comparison of JSON values takes place at two levels. The first level of comparison is based on the JSON types of the compared values. If the types differ, the comparison result is determined solely by which type has higher precedence. If the two values have the same JSON type, a second level of comparison occurs using type-specific rules.

The following list shows the precedences of JSON types, from highest precedence to the lowest. (The type names are those returned by the `JSON_TYPE()` function.) Types shown together on a line have the same precedence. Any value having a JSON type listed earlier in the list compares greater than any value having a JSON type listed later in the list.

```
BLOB
BIT
OPAQUE
DATETIME
TIME
DATE
BOOLEAN
ARRAY
OBJECT
STRING
INTEGER, DOUBLE
NULL
```

For JSON values of the same precedence, the comparison rules are type specific:

- `BLOB`

  The first ***N*** bytes of the two values are compared, where ***N*** is the number of bytes in the shorter value. If the first ***N*** bytes of the two values are identical, the shorter value is ordered before the longer value.

- `BIT`

  Same rules as for `BLOB`.

- `OPAQUE`

  Same rules as for `BLOB`. `OPAQUE` values are values that are not classified as one of the other types.

- `DATETIME`

  A value that represents an earlier point in time is ordered before a value that represents a later point in time. If two values originally come from the MySQL `DATETIME` and `TIMESTAMP` types, respectively, they are equal if they represent the same point in time.

- `TIME`

  The smaller of two time values is ordered before the larger one.

- `DATE`

  The earlier date is ordered before the more recent date.

- `ARRAY`

  Two JSON arrays are equal if they have the same length and values in corresponding positions in the arrays are equal.

  If the arrays are not equal, their order is determined by the elements in the first position where there is a difference. The array with the smaller value in that position is ordered first. If all values of the shorter array are equal to the corresponding values in the longer array, the shorter array is ordered first.

  Example:

  ```
  [] < ["a"] < ["ab"] < ["ab", "cd", "ef"] < ["ab", "ef"]
  ```

- `BOOLEAN`

  The JSON false literal is less than the JSON true literal.

- `OBJECT`

  Two JSON objects are equal if they have the same set of keys, and each key has the same value in both objects.

  Example:

  ```
  {"a": 1, "b": 2} = {"b": 2, "a": 1}
  ```

  The order of two objects that are not equal is unspecified but deterministic.

- `STRING`

  Strings are ordered lexically on the first ***`N`*** bytes of the `utf8mb4` representation of the two strings being compared, where ***`N`*** is the length of the shorter string. If the first ***`N`*** bytes of the two strings are identical, the shorter string is considered smaller than the longer string.

Example:

```
"a" < "ab" < "b" < "bc"
```

This ordering is equivalent to the ordering of SQL strings with collation `utf8mb4_bin`. Because `utf8mb4_bin` is a binary collation, comparison of JSON values is case-sensitive:

```
"A" < "a"
```

- `INTEGER`, `DOUBLE`

  JSON values can contain exact-value numbers and approximate-value numbers. For a general discussion of these types of numbers, see Section 9.1.2, “Numeric Literals”.

  The rules for comparing native MySQL numeric types are discussed in Section 12.3, “Type Conversion in Expression Evaluation”, but the rules for comparing numbers within JSON values differ somewhat:

  - In a comparison between two columns that use the native MySQL `INT` and `DOUBLE` numeric types, respectively, it is known that all comparisons involve an integer and a double, so the integer is converted to double for all rows. That is, exact-value numbers are converted to approximate-value numbers.
  - On the other hand, if the query compares two JSON columns containing numbers, it cannot be known in advance whether numbers are integer or double. To provide the most consistent behavior across all rows, MySQL converts approximate-value numbers to exact-value numbers. The resulting ordering is consistent and does not lose precision for the exact-value numbers. For example, given the scalars 9223372036854775805, 9223372036854775806, 9223372036854775807 and 9.223372036854776e18, the order is such as this:

    ```
    9223372036854775805 < 9223372036854775806 < 9223372036854775807
    < 9.223372036854776e18 = 9223372036854776000 < 9223372036854776001
    ```

  Were JSON comparisons to use the non-JSON numeric comparison rules, inconsistent ordering could occur. The usual MySQL comparison rules for numbers yield these orderings:

  - Integer comparison:

    ```
    9223372036854775805 < 9223372036854775806 < 9223372036854775807
    ```

(not defined for 9.223372036854776e18)

- Double comparison:

```
9223372036854775805 = 9223372036854775806 = 9223372036854775807 = 9.223372
```

For comparison of any JSON value to SQL `NULL`, the result is `UNKNOWN`.

For comparison of JSON and non-JSON values, the non-JSON value is converted to JSON according to the rules in the following table, then the values compared as described previously.

Converting between JSON and non-JSON values

The following table provides a summary of the rules that MySQL follows when casting between JSON values and values of other types:

Table 11.3 JSON Conversion Rules

other type	CAST(other type AS JSON)	CAST(JSON AS other type)
JSON	No change	No change
utf8 character type (`utf8mb4`, `utf8mb3`, `ascii`)	The string is parsed into a JSON value.	The JSON value is serialized into a `utf8mb4` string.
Other character types	Other character encodings are implicitly converted to `utf8mb4` and treated as described for this character type.	The JSON value is serialized into a `utf8mb4` string, then cast to the other character encoding. The result may not be meaningful.
`NULL`	Results in a `NULL` value of type JSON.	Not applicable.
Geometry types	The geometry value is converted into a JSON document by calling `ST_AsGeoJSON()`.	Illegal operation. Workaround: Pass the result of `CAST(`***`json_val`***` AS CHAR)` to `ST_GeomFromGeoJSON()`.
All other types	Results in a JSON document consisting of a single scalar value.	Succeeds if the JSON document consists of a single scalar value of the target type and that scalar value can be cast to the target type. Otherwise, returns `NULL` and produces a warning.

`ORDER BY` and `GROUP BY` for JSON values works according to these principles:

- Ordering of scalar JSON values uses the same rules as in the preceding discussion.

- For ascending sorts, SQL `NULL` orders before all JSON values, including the JSON null literal; for descending sorts, SQL `NULL` orders after all JSON values, including the JSON null literal.
- Sort keys for JSON values are bound by the value of the `max_sort_length` system variable, so keys that differ only after the first `max_sort_length` bytes compare as equal.
- Sorting of nonscalar values is not currently supported and a warning occurs.

For sorting, it can be beneficial to cast a JSON scalar to some other native MySQL type. For example, if a column named `jdoc` contains JSON objects having a member consisting of an `id` key and a nonnegative value, use this expression to sort by `id` values:

```
ORDER BY CAST(JSON_EXTRACT(jdoc, '$.id') AS UNSIGNED)
```

If there happens to be a generated column defined to use the same expression as in the `ORDER BY`, the MySQL optimizer recognizes that and considers using the index for the query execution plan. See Section 8.3.11, “Optimizer Use of Generated Column Indexes”.

Aggregation of JSON Values

For aggregation of JSON values, SQL `NULL` values are ignored as for other data types. Non-`NULL` values are converted to a numeric type and aggregated, except for `MIN()`, `MAX()`, and `GROUP_CONCAT()`. The conversion to number should produce a meaningful result for JSON values that are numeric scalars, although (depending on the values) truncation and loss of precision may occur. Conversion to number of other JSON values may not produce a meaningful result.