JSON in MySQL: The Ultimate Guide

A guide to working with JSON data in MySQL

Ben Brumm

www.databasestar.com

JSON in MySQL: The Ultimate Guide

MySQL has quite a few features for storing and working with JSON data.

In this guide, you'll learn:

- What JSON is and why you might want to use it
- Creating a table with a JSON field
- How to add, read, update, and delete JSON data
- Tips for performance, validating and working with JSON

Let's get into the guide.

If you want to download a PDF version of this guide, enter your email below and you'll receive it shortly.

What is JSON and Why Should I Use It?

JSON stands for JavaScript Object Notation, and it's a way to format and store data.

Data can be represented in a JSON format so it can be read and understood by other applications or parts of an application.

It's similar to HTML or XML - it represents your data in a certain format that is readable by people but designed to be readable by applications.

Why Use JSON In Your Database?

So why would you use JSON data in your database?

If you need a structure that's flexible.

A normalised database structure, one with tables and columns and relationships, works well for most cases. Recent improvements in development practices also mean that altering a table is not as major as it was in the past, so adjusting your database once it's in production is possible.

However, if your requirements mean that your data structure needs to be flexible, then a JSON field may be good for your database.

One example may be where a user can add custom attributes. If it was done using a normalised database, this may involve altering tables, or creating an Entity Attribute Value design, or some other method.

If a JSON field was used for this, it would be much easier to add and maintain these custom attributes.

The JSON data can also be stored in your database and processed by an ORM (Object Relational Mapper) or your application code, so your database may not need to do any extra work.

What Does JSON Data Look Like?

Here's a simple example of JSON data:

```
{
  "id": "1",
  "username": "jsmith",
  "location": "United States"
}
```

It uses a combination of different brackets, colons, and quotes to represent your data.

Let's take a look at some more examples.

Name/Value Pair

JSON data is written as name/value pairs. A name/value pair is two values enclosed in quotes.

This is an example of a name/value pair:

```
"username": "jsmith"
```

The name is "username" and the value is "jsmith". They are separated by a colon ":".

This means for the attribute of username, the value is jsmith. Names in JSON need to be enclosed in double quotes.

Objects

JSON data can be enclosed in curly brackets which indicate it's an object.

```
{"username": "jsmith"}
```

This is the same data as the earlier example, but it's now an object. This means it can be treated as a single unit by other areas of the application.

How does this help? It's good for when there are multiple attributes:

```
{
  "username": "jsmith",
  "location": "United States"
}
```

Additional attributes, or name/value pairs, can be added by using a comma to separate them.

You'll also notice in this example the curly brackets are on their own lines and the data is indentented. This is optional: it's just done to make it more readable.

Arrays

JSON also supports arrays, which is a collection of records within an object. Arrays in JSON are included in square brackets and have a name:

```
{
  "username": "jsmith",
  "location": "United States",
  "posts": [
    {
      "id":"1",
      "title":"Welcome"
    },
    {
      "id":"4",
      "title":"What started it all"
     }
  ]
}
```

In this example, this object has an attribute called "posts". The value of posts is an array, which we can see by the opening square bracket "[".

Inside the square bracket, we have a set of curly brackets, indicating an object, and inside those we have an id of 1 and a title of Welcome. We have another set of curly brackets indicating another object.

These two objects are posts and they are contained in an array.

And that covers the basics of what JSON is.

If JSON is new to you, don't worry, it gets easier as you work with it more.

If you're experienced with JSON, you'll find the rest of the guide more useful as we go into the details of working with JSON in MySQL.

How to Create and Populate JSON Field in MySQL

So you've learned a bit about JSON data and why you might want to use it.

How do we create a field in MySQL?

Creating a JSON Field

We create a new field with a data type of JSON.

Here's an example.

```
CREATE TABLE product (
id INT,
product_name VARCHAR(200),
attributes JSON
);
```

We have created a table called product. It has an id and a product name. There's also an attributes column, which has the data type of JSON.

Adding a JSON column is as easy as that.

You might be thinking, if JSON data is just text data, why do we need to do anything special? Can't we just create a large CLOB or VARCHAR field and add the JSON data to it?

We could, but the main benefit of a JSON data type is the format validation. With a simple CLOB field, we can store a text value. However, it means that we can store JSON that's invalid: missing attribute names or curly brackets.

With a JSON field, the data is automatically validated for us. We won't be able to store invalid data in the table.

Also, we get to use the various MySQL JSON functions on the JSON data to make working with it easier.

Adding Data to a JSON Field

Now we've got our JSON field, how do we add data to it?

There are a few ways. We can enter the data into a JSON-formatted string, or use a built-in MySQL function. Let's see both of them.

We can add our first product like this:

```
INSERT INTO product (id, product_name, attributes)
VALUES (1, 'Chair', '{"color":"brown", "material":"wood",
"height":"60cm"}');
```

We can run this statement and the record is inserted. If we select the data from the table, this is what we see:

```
SELECT
id,
product_name,
attributes
FROM product;
```

id	product_name	attributes
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}

The JSON data is shown exactly as we entered it.

Using the method above, we needed to enter in the data in exactly the right format.

However, we can use the JSON_OBJECT function to make this easier. The JSON_OBJECT function accepts a list of name/value pairs which are converted to a JSON object.

We can insert a record into the table using this function (which has been formatted to make it easier to read):

```
INSERT INTO product (id, product_name, attributes)
VALUES (2, 'Table', JSON_OBJECT(
  "color", "brown",
  "material", "wood",
  "height", "110cm"
));
```

This makes it easier to specify attributes as you don't need to remember the curly brackets or the colons.

We can select from the table to see the results.

```
SELECT
id,
product_name,
attributes
FROM product;
```

id	product_name	attributes
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}

The JSON data is shown with the data we entered.

Inserting Arrays

If you want to insert JSON data that contains arrays, you can either enter it using text in a JSON format, or use a function called JSON_ARRAY.

Here's how to insert an array by just specifying it in a JSON format.

```
INSERT INTO product (id, product_name, attributes)
VALUES (3, 'Desk', '{"color":"black", "material":"metal",
"drawers":[{"side":"left", "height":"30cm"}, {"side":"left",
"height":"40cm"}]}');
```

This will insert a new product that has an array of drawers. As you can probably see by this statement, reading it (and writing it) is a bit tricky.

You can insert simpler arrays using this method too.

```
INSERT INTO product (id, product_name, attributes)
VALUES (4, 'Side Table', '{"color":"brown", "material":["metal",
"wood"]}');
```

The INSERT statements will work, and the data will look like this:

```
SELECT
id,
product_name,
attributes
FROM product;
```

id	product_name	attributes
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	{"color": "brown", "material": ["metal", "wood"]}

There is an easier way to insert array data in JSON in MySQL.

Inserting Arrays with JSON_ARRAY

The JSON_ARRAY function in MySQL lets you easily specify array data when inserting JSON data in MySQL.

Let's insert another record into our table. We can use the JSON_ARRAY function along with the JSON_OBJECT function.

We'll insert a simple array, and one with objects.

Here's a simple array:

```
INSERT INTO product (id, product_name, attributes)
VALUES (5, 'Dining Table', JSON_OBJECT(
  "color", "brown",
  "material", JSON_ARRAY(
    "wood", "metal"
  )
));
```

We specify the JSON_ARRAY function, and inside the function, we specify the different attributes to be added to the array.

Here's an insert statement with objects in the array.

```
INSERT INTO product (id, product_name, attributes)
VALUES (6, 'Large Desk', JSON_OBJECT(
  "color", "white",
  "material", "metal",
  "drawers", JSON_ARRAY(
'{"side":"left", "height":"50cm"}',
'{"side":"right", "height":"50cm"}'
  )
));
```

Using the JSON_ARRAY function helps us to validate the data we're inserting, and reduces the chances of getting an error due to a misplaced comma, quote, or bracket.

Here's what our table looks like now.

```
SELECT
id,
product_name,
attributes
FROM product;
```

id	product_name	attributes
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}

3	Desk	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

The new records with the array data are shown, and the arrays are enclosed in square brackets.

We can see the last row, with an id of 6, has backslashes in its value. This is due to the fact an array object is used for an attribute. We'll see how to display this better later.

How to Read and Filter JSON Data in MySQL

Once you've got some JSON data in a table, the next step is to read it.

How do we do that?

We can run a simple SELECT statement to see the data in the table.

```
SELECT
id,
product_name,
attributes
FROM product;
```

id	product_name	attributes
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

This shows us the data in the JSON column, and it looks just like a text value.

The good thing with this is that any application can easily read this field and work with it how they want (display it, filter it, and so on).

What if we wanted to do more in our database?

Selecting Individual Attributes

The JSON data is stored in something that looks like a text field. However, it's quite easy to get attributes and values out of this text field and display them.

We can extract a value from the JSON field and display it in a separate column. We do this using a combination of "path expressions" and the JSON_EXTRACT function.

We need to use the JSON_EXTRACT function to search for a particular attribute in the JSON value. And we need to use a "path expression" to specify the attribute.

How do we do this?

First, let's write the path expression, and then put that into our JSON_EXTRACT function.

The path expression lets us specify the attribute we want to search for. It starts with a $ symbol, and we specify a dot then the name of the attribute we're looking for.

For example, to specify the "color" attribute, out path expression would look like this:

```
'$.color'
```

To specify the "material" attribute, we can use this path expression:

```
'$.material'
```

If we had a height attribute enclosed in a dimensions attribute, our path expression would look like this:

```
'$.dimensions.height'
```

We use the dot to specify the next level in the hierarchy of attributes.

How do we use this to filter our data? We combine this path expression with the JSON_EXTRACT function.

The JSON_EXTRACT function takes two parameters:

```
JSON_EXTRACT (column, path_expression)
```

We can use this in an example.

Displaying a Field using JSON_EXTRACT

Let's say we want to display the color attribute in a separate column in our results.

First, we write the path expression for the color attribute:

```
'$.color'
```

Then, we add this to our JSON_EXTRACT function:

```
JSON_EXTRACT(attributes, '$.color')
```

Finally, we add this to our SELECT clause to show it as a separate column.

```
SELECT
id,
product_name,
JSON_EXTRACT(attributes, '$.color') AS color,
attributes
FROM product;
```

This query will show all records, and show the color attribute as a separate column.

id	product_name	color	attributes
1	Chair	"brown"	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	"brown"	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	"black"	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	"brown"	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	"brown"	{"color": "brown", "material": ["wood", "metal"]}

6	Large Desk	"white"	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

We can see the separate column here.

Path Expression Examples

We can see another field using JSON_EXTRACT by specifying the attribute name:

```
SELECT
id,
product_name,
JSON_EXTRACT(attributes, '$.height') AS height,
attributes
FROM product;
```

Here we are extracting the attribute called height. This is available in some records but not others.

id	product_name	height	attributes
1	Chair	"60cm"	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	"110cm"	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	null	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	null	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	null	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	null	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

A null value is shown for records that don't have this attribute.

What about attributes that are arrays, such as "material" in this example?

```
SELECT
id,
product_name,
JSON_EXTRACT(attributes, '$.material') AS material,
attributes
```

```
FROM product;
```

id	product_name	material	attributes
1	Chair	"wood"	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	"wood"	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	"metal"	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	["metal", "wood"]	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	["wood", "metal"]	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	"metal"	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

What if we want to see an attribute that's inside another attribute? For example, the first of the "drawer" attributes?

Because "drawer" is an array, we can't use the dot notation to get the attribute like this:

```
JSON_EXTRACT(attributes, '$.drawers.side') AS side
```

This will return a null value as there is no attribute called side: it's part of an array.

However, we can use a number to reference the position in the array.

You can return the first object using [0]:

```
JSON_EXTRACT(attributes, '$.drawers[0]')
```

The second object can be found using [1], the third object using [2], and so on.

So, our query to extract the first item in the array is:

```
SELECT
id,
product_name,
JSON_EXTRACT(attributes, '$.drawers[0]') AS drawer,
attributes
FROM product;
```

The results are:

id	product_name	drawer	attributes
1	Chair	null	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	null	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	{"side": "left", "height": "30cm"}	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	null)	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	null	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	"{\side\":\"left\", \"height\":\"50cm\ "}"	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

So, as you can see, there are a range of ways you can use the JSON_EXTRACT function with a path expression to get the attribute you want.

Filtering on JSON Data in MySQL

Let's say we wanted to see our Chair product, which has a brown color, wood material, and a height of 60cm. But we want to filter on the JSON attributes for this example.

Let's try this query.

```
SELECT
id,
product_name,
attributes
FROM product
WHERE attributes = '{"color":"brown", "material":"wood",
"height":"60cm"}';
```

We can run this query. However, we don't get any results.

Unfortunately we can't just filter on a JSON column like that.

What if we try using the LIKE keyword with a partial match?

```
SELECT
id,
```

```
product_name,
attributes
FROM product
WHERE attributes LIKE '%"color":"brown"%';
```

No results found.

That doesn't give us the result we want either. Also, using wildcard searches can be quite slow if there is a lot of data in the table.

However, there are several features in MySQL that make it possible to filter on JSON data.

Using JSON_EXTRACT to Filter Data

Let's say we want to find all products where the color is brown. The color is a part of the attributes JSON column in our table.

Our path expression would look like this:

```
'$.color'
```

We can write the JSON_EXTRACT function like this:

```
JSON_EXTRACT(attributes, '$.color')
```

We then add this to our SELECT statement

```
SELECT
id,
product_name,
attributes
FROM product
WHERE JSON_EXTRACT(attributes, '$.color') = 'brown';
```

We've added this JSON_EXTRACT function to our WHERE clause, and added a condition where the color is equal to "brown".

The results of this query are shown below.

id	product_name	attributes

1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}
4	Side Table	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	{"color": "brown", "material": ["wood", "metal"]}

We can see that the results only show records where the color attribute is brown.

A Shortcut for JSON_EXTRACT

Our earlier example used the JSON_EXTRACT function to filter records based on JSON attributes.

```
SELECT
id,
product_name,
attributes
FROM product
WHERE JSON_EXTRACT(attributes, '$.color') = 'brown';
```

There is a shortcut in MySQL for the JSON_EXTRACT function: the -> symbols.

This means you can use -> to write a JSON_EXTRACT function. Using this symbol, the same query can be represented like this:

```
SELECT
id,
product_name,
attributes
FROM product
WHERE attributes -> '$.color' = 'brown';
```

The results are:

id	product_name	attributes
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}
4	Side Table	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	{"color": "brown", "material": ["wood", "metal"]}

This shows the same result as using the actual function name.

This can also be used anywhere the JSON_EXTRACT function can be used, such as in the SELECT clause.

```
SELECT
id,
product_name,
attributes -> '$.color' AS color,
attributes
FROM product;
```

id	product_name	color	attributes
1	Chair	brown	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	brown	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	black	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	brown	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	brown	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	white	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

Searching for Data using JSON_CONTAINS

The JSON_CONTAINS function will look for a match and return 1 if it is found or 0 if it is not found.

The syntax is:

```
JSON_CONTAINS(target, candidate [, path])
```

It takes three parameters:

- target: the JSON document to search within.
- candidate: the JSON document to search for
- path: an optional path value to search for within the target

Let's see some examples.

This example

```
SELECT
id,
product_name,
JSON_CONTAINS(attributes, '{"color":"brown"}') AS contain_json,
attributes
FROM product;
```

id	product_name	contain_json	attributes
1	Chair	1	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	1	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	0	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	1	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	1	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	0	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

We can also use the JSON_OBJECT to construct either of the parameters, as JSON_OBJECT returns a JSON data type.

```
SELECT
id,
product_name,
JSON_CONTAINS(
  attributes,
  JSON_OBJECT("material", "wood")
) AS contain_json,
attributes
FROM product;
```

id	product_name	contain_json	attributes
1	Chair	1	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	1	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	0	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	1	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	1	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	0	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

You can search within a specific path by using the path parameter.

Here's a query to find whether or not the attributes contain a name value pair of height and 40cm within the drawers attribute:

```
SELECT
id,
product_name,
JSON_CONTAINS(attributes, '{"height":"40cm"}', '$.drawers') AS
contain_json,
attributes
FROM product;
```

id	product_name	contain_json	attributes
1	Chair	null	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	null	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	1	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	null	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	null	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	0	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

It shows 1 for found, 0 for not found, and null where the drawers attribute does not exist.

Finding the Path using JSON_SEARCH

The JSON_SEARCH function lets you find the path to a specified string.

The syntax is:

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

The parameters are:

- json_doc: this is the JSON field or document to search in.

- one_or_all: specify either "one" to terminate the search after the first match or "all" to return all matching results.
- search_str: the string to search for within json_doc.
- escape_char: used if you want to specify a literal % or _ within your search string (as those are wildcard characters).
- path: search for the string in the specified path

Let's see some examples.

Here's an example of searching for the string "brown".

```
SELECT
id,
product_name,
JSON_SEARCH(attributes, 'one', 'brown') AS search_result,
attributes
FROM product;
```

The results are:

id	product_name	search_result	attributes
1	Chair	"$.color"	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	"$.color"	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	null	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	"$.color"	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	"$.color"	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	null	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

We can see that some rows have returned a value of "$.color". This is the path that contains the word "brown".

What if we look for all paths containing the letter "o"

```
SELECT
id,
product_name,
JSON_SEARCH(attributes, 'all', '%o%') AS search_result,
attributes
FROM product;
```

The results are:

id	product_name	search_result	attributes
1	Chair	["$.color", "$.material"]	{"color": "brown", "height": "60cm", "material": "wood"}
2	Table	["$.color", "$.material"]	{"color": "brown", "height": "110cm", "material": "wood"}
3	Desk	null	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	["$.color", "$.material[1]"]	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	["$.color", "$.material[0]"]	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	null	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

We can see that some rows have multiple results, which are contained within an array (the square brackets).

The JSON_SEARCH function is pretty powerful if you need to find the path based on a string.

Finding the Type of Value using JSON_TYPE

There's a function called JSON_TYPE that lets you find the type of a provided JSON value. Some examples of types are:

- Object
- Array
- Integer

This can be helpful if you need to perform logic or more processing on a JSON value.

Let's see some examples.

Here's an example of using JSON_TYPE on a full JSON value that we've seen in our table:

```
SELECT
id,
product_name,
JSON_TYPE(attributes) AS jtype,
```

```
attributes
FROM product
WHERE id = 1;
```

The results are:

id	product_name	jtype	attributes
1	Chair	OBJECT	{"color": "black", "depth": "60cm", "width": "100cm", "height": "60cm", "material": "wood"}

The result of the JSON_TYPE function for this data is OBJECT, indicating that the JSON string is a JSON object.

We can use this on arrays as well.

```
SELECT
'["a", "b", "c"]' AS json_data,
JSON_TYPE('["a", "b", "c"]') AS jtype;
```

Here are the results:

json_data	jtype
["a", "b", "c"]	ARRAY

We can see that the JSON_TYPE has shown it is an array.

This JSON_TYPE function can be used on different values to determine what type it is.

How to Update JSON Data in MySQL

Reading JSON is one thing. What if you need to update JSON data?

You could extract the string, do some substring and replacement work on the string, and add it into the field, but that's error-prone and a lot of work.

There are a few functions in MySQL that allow you to update a field quite easily.

Here's a summary of the methods.

Requirement	**Function**
Add a new key and value	JSON_INSERT
Update a value for an existing key	JSON_REPLACE
Add a new key and value or update an existing key's value	JSON_SET

Let's see some examples of them.

Insert a New Item

One way to update JSON data is to add a new item to an existing JSON value. This can be done with the JSON_INSERT function.

The JSON_INSERT function syntax looks like this:

```
JSON_INSERT(json_doc, path, val [, path, val…] )
```

The parameters are:

- json_doc: the JSON field to be updated
- path: the path to add a new value for
- val: the value to add for the path

This function will return the updated JSON value. So, because we're updating an existing row in the table, we use the UPDATE statement and this function.

Here's the first item in our table:

```
SELECT
id,
product_name,
attributes
FROM product
WHERE id = 1;
```

id	**product_name**	**attributes**
1	Chair	{"color": "brown", "height": "60cm", "material": "wood"}

Let's say we want to add a new attribute name and value, in addition to the color, height, and material that already exist. This new attribute would be "width", and the value is 100cm.

```
UPDATE product
SET attributes = JSON_INSERT(attributes, '$.width', '100cm')
WHERE id = 1;
```

The path is "$.width" as it's a width attribute at the top level (not within another attribute).

We can run the same SELECT statement as above to see the updated value.

id	product_name	attributes
1	Chair	'{"color": "brown", "width": "100cm", "height": "60cm", "material": "wood"}'

We can see that "width" has been added to the list of attributes.

Insert Where Key Already Exists

What happens if we use the JSON_INSERT function but the key already exists? Will the function:

- add a second value, turning the attribute value into an array?
- overwrite the existing value with the new value
- add a second attribute name and value pair
- get ignored?

Let's see.

This query will update the same product using an attribute key that already exists (color). We're also adding a new attribute called "depth", so we can see that the statement has run successfully (by seeing that depth has been added).

Here's the statement:

```
UPDATE product
SET attributes = JSON_INSERT(attributes, '$.color', 'white',
'$.depth', '60cm')
WHERE id = 1;
```

This statement runs successfully.

Here's the SELECT statement and output:

```
SELECT
id,
product_name,
attributes
FROM product
WHERE id = 1;
```

id	product_name	attributes
1	Chair	{"color": "brown", "depth": "60cm", "width": "100cm", "height": "60cm", "material": "wood"}

The depth attribute is added. However, the color attribute is unchanged. This is because the JSON_INSERT function ignores any path in the function where the path already exists. So, the color value stayed as brown and did not update to white.

Updating an Existing Value with JSON_REPLACE

If you want to replace a value of an attribute inside a JSON field with another value, you can use the JSON_REPLACE function.

The syntax of MySQL JSON_REPLACE looks like this:

```
JSON_REPLACE (json_doc, path, val [, path, val…] )
```

The parameters are:

- json_doc: the JSON field to be updated
- path: the path to update the value for
- val: the value to update

The path must refer to an existing attribute key. If it does not exist, then nothing will be done (the attribute will not be added).

Let's see some examples.

Here's our first product ID:

id	product_name	attributes
1	Chair	{"color": "brown", "depth": "60cm", "width": "100cm", "height": "60cm", "material": "wood"}

We can update the value of color to black by using this JSON_REPLACE function. Because we are updating an existing value, we use the UPDATE statement.

```
UPDATE product
SET attributes = JSON_REPLACE(attributes, '$.color', 'black')
WHERE id = 1;
```

We can select from this table and see the record is updated:

id	product_name	attributes
1	Chair	{"color": "black", "depth": "60cm", "width": "100cm", "height": "60cm", "material": "wood"}

Inserting and Updating with JSON_SET

The JSON_INSERT function inserts a new attribute and JSON_REPLACE updates an existing attribute.

The JSON_SET function combines both of these: updates the value if it exists or inserts it if it does not exist.

The syntax is:

```
JSON_SET (json_doc, path, val [, path, val…] )
```

The parameters are:

- json_doc: the JSON field to be updated
- path: the path to update the value for or to insert a new attribute for
- val: the value to update or insert for the attribute

Notice that there are square brackets in the syntax, which indicates optional values. You can add many extra path and value parameters to add or update multiple values.

Let's see an example using product ID 2.

```
SELECT
```

```
id,
product_name,
attributes
FROM product
WHERE id = 2;
```

id	product_name	attributes
2	Table	{"color": "brown", "height": "110cm", "material": "wood"}

Let's change the color value to black and the depth to 100cm. We can use the JSON_SET function to do this.

```
UPDATE product
SET attributes = JSON_SET(attributes, '$.color', 'black', '$.depth',
'100cm')
WHERE id = 2;
```

Here's the updated data:

id	product_name	attributes
2	Table	{"color": "black", "depth": "100cm", "height": "110cm", "material": "wood"}

We can see that the existing color value was changed to black, and the new attribute of depth was added.

How to Delete from a JSON Field in MySQL

There are two DELETE operations you can do when working with JSON fields:

- delete an attribute from a JSON field
- delete a row from your table

Deleting a Row using JSON_EXTRACT

Deleting a row from your table is done in the same way as regular SQL. You can write an SQL statement to delete the row that matches your ID, or using JSON_EXTRACT.

For example, to delete all rows where the color attribute is brown:

```
DELETE FROM product
WHERE JSON_EXTRACT(attributes, '$.color') = 'brown';
```

This will remove the matching records from the table.

Removing an Attribute from a JSON Field

The other way to delete JSON data is to remove an attribute from a JSON field.

This is different to updating, as you're removing the attribute entirely rather than just updating its value to something else.

We can do this with an UPDATE statement and the JSON_REMOVE function.

The JSON_REMOVE function will remove data from a JSON field. The syntax is:

```
JSON_REMOVE (json_doc, path [, path …] )
```

The parameters are:

- json_doc: the JSON field to remove data from.
- path: the path to the attribute to remove

There's no need to specify a value like the other JSON functions.

Let's see an example by removing an attribute from product ID 2.

```
SELECT
id,
product_name,
attributes
FROM product
WHERE id = 2;
```

id	product_name	attributes
2	Table	{"color": "black", "depth": "100cm", "height": "110cm", "material": "wood"}

We can run an UPDATE statement with JSON_REMOVE to remove the "height" attribute.

```
UPDATE product
SET attributes = JSON_REMOVE(attributes, '$.height')
WHERE id = 2;
```

We can then select the data from the table again to see that it has been removed:

id	product_name	attributes
2	Table	{"color": "black", "depth": "100cm", "material": "wood"}

The attribute of height is no longer in the JSON field.

Validating JSON Data

We've seen in the examples so far in this guide that the JSON data type in MySQL automatically validates data for you. It ensures you can only insert valid JSON data into the field.

MySQL also includes a function that lets you check if a string is a valid JSON field. This can be helpful if you're accepting a JSON string from another system.

The function is JSON_VALID. You provide it with a value, and it returns 1 if it's a valid JSON string and 0 if it is not.

The syntax is:

```
JSON_VALID (value)
```

Here are some examples.

Example of Valid JSON String

Let's say we have this JSON string:

```
'{"color": "black", "depth": "100cm", "material": "wood"}'
```

We can test if this is valid by looking at it for quotes and other symbols in the right places. Or we can just pass it to the JSON_VALID function.

```
SELECT
```

```
JSON_VALID('{"color": "black", "depth": "100cm", "material":
"wood"}') AS valid_test;
```

valid_test
1

The result is 1 so it's a valid JSON value.

Example of an Invalid JSON String

What if the value is not valid?

We can see this sample JSON string:

```
'{"color": "black", "depth" "100cm", "material": "wood"}'
```

At first glance it may seem valid. Let's use the JSON_VALID function to check.

```
SELECT
JSON_VALID('{"color": "black", "depth" "100cm", "material": "wood"}')
AS valid_test;
```

valid_test
0

The result is 0 so it's not a valid JSON value. The result does not say where the issue is in the provided string, but it tells you it's invalid so you can look closer at it.

Improve the Performance of JSON Queries

The JSON support and features in MySQL are pretty good, and each version includes more features.

So, given that you can add JSON columns to tables, extract fields, and get all the flexibility of JSON fields with validation, wouldn't it be better to just store all of your data in JSON fields rather than normalised tables?

Well, sometimes that might be a good idea. But then you may be better off using a NoSQL database rather than MySQL.

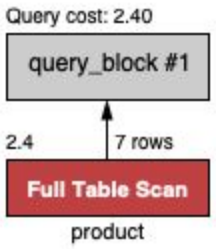
Another reason why using primarily JSON fields to store your data is not a good idea is that it can struggle with performance.

Select Performance

For example, let's say we want to select all products where the color is brown. We can use the JSON_EXTRACT function in the WHERE clause that we saw earlier in this guide:

```
SELECT
id,
product_name,
attributes
FROM product
WHERE JSON_EXTRACT(attributes, '$.color') = 'brown';
```

Let's see the execution plan for this:



The execution plan shows a Full Table Scan step, which is a slow type of access. This might be OK for our table, which only has a few records, but once you start working with larger tables it can be quite slow.

So, using the JSON_EXTRACT function in the WHERE clause will mean a full table scan is used.

What can we do?

One Solution to Selecting Data

Fortunately, MySQL allows you to define a virtual column on the table, and create an index on that virtual column. This should make our query faster.

A virtual column is a column that is a calculation based on another column in the table.

Let's see how we can do that.

First, we create a new column that contains the color attribute:

```
ALTER TABLE product
ADD COLUMN color VARCHAR(100)
GENERATED ALWAYS AS (JSON_EXTRACT(attributes, '$.color'));
```

We can select from the product table to see it.

```
SELECT
id,
product_name,
color,
attributes
FROM product;
```

id	product_name	color	attributes
1	Chair	"black"	{"color": "black", "depth": "60cm", "width": "100cm", "height": "60cm", "material": "wood"}
2	Table	"black"	{"color": "black", "depth": "100cm", "material": "wood"}
3	Desk	"black"	{"color": "black", "drawers": [{"side": "left", "height": "30cm"}, {"side": "left", "height": "40cm"}], "material": "metal"}
4	Side Table	"brown"	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	"brown"	{"color": "brown", "material": ["wood", "metal"]}
6	Large Desk	"white"	{"color": "white", "drawers": ["{\"side\":\"left\", \"height\":\"50cm\"}", "{\"side\":\"right\", \"height\":\"50cm\"}"], "material": "metal"}

Now, we can create an index on this new column.

```
CREATE INDEX idx_prod_color ON product(color);
```

Now, let's select from the table again, filtering on the virtual column instead of the JSON field.

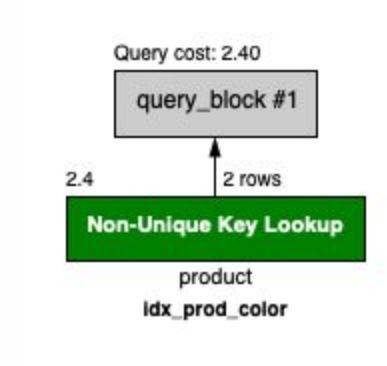
```
SELECT
id,
product_name,
color,
attributes
FROM product
WHERE color = '"brown"';
```

Notice that the filter in the WHERE clause includes double quotes, as this is the value that was extracted from the JSON field.

The results are:

id	product_name	color	attributes
4	Side Table	brown	{"color": "brown", "material": ["metal", "wood"]}
5	Dining Table	brown	{"color": "brown", "material": ["wood", "metal"]}

We can check the execution plan to see how it was run.



We can see that the step is now called "Non-Unique Key Lookup" and the index was used. This is more efficient than the Full Table Scan.

Having said all of that, if you're creating virtual columns to be able to access data in MySQL JSON fields more efficiently just to make your application and database work, then perhaps the JSON field is not right for your database. But only you would know that - each case is different.

Tips for Working with JSON in MySQL

In this guide we've looked at what JSON is, seen how to create JSON fields in MySQL, and seen a range of ways we can work with them.

So, what's the best way to work with JSON fields in MySQL?

Here are some tips I can offer for using JSON in MySQL. They may not apply to your application or database but they are things to consider.

Just because you can, doesn't mean you should
JSON is flexible and quite powerful, but just because you can store data in a JSON field, doesn't mean you should. Consider using the advantages of MySQL relational database and using JSON where appropriate.

Treat the JSON field like a black box
The JSON field can be used to store valid JSON data sent or received by your application. While there are functions for reading from and working with the JSON field, it might be better to just store the JSON data in the field, retrieve it from your application, and process it there.

This is the concept of a black box. The application puts data in and reads data from it, and the database doesn't care about what's inside the field.

It may or may not work for your situation, but consider taking this approach.

Search by the Primary Key and other fields
We've seen that it can be slow to search by attributes inside the JSON field. Consider filtering by the primary key and other fields in the table, rather than attributes inside the JSON field. This will help with performance.

Conclusion

I hope you found this guide useful. Have you used JSON fields in MySQL? What has your experience been like? Do you have any questions? Feel free to use the comments section on the post.