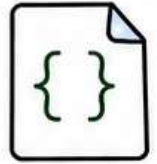


Chapter 25: MySQL JSON

25.1 Introduction to JSON

- * JSON (JavaScript Object Notation) is a **lightweight data-interchange format**.
- * It stores data in **key-value pairs** and is easy for humans to read and write.
- * JSON is language independent and widely used in **web applications**, **APIs**, **NoSQL databases**, etc.
- * MySQL supports **JSON data type** to store, query and manage JSON data efficiently.
- * Using JSON, we can store **semi-structured data** more flexibly in MySQL.



JSON



MySQL

Key Points :

- ✓ JSON is a text format for exchanging data.
- ✓ It uses key-value pairs, arrays and objects.
- ✓ MySQL has built-in support for storing and querying JSON data.
- ✓ JSON is ideal for semi-structured or nested data.

Example JSON Document :

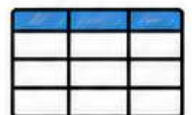
```
{
  "name": "Alice",
  "age": 25,
  "skills": ["SQL", "Python", "Java"],
  "address": {
    "city": "Hyderabad",
    "state": "Telangana"
  }
}
```

25.2 JSON Data Type

- * MySQL provides a **JSON data type** to store JSON documents in a column.
- * It stores data in a **binary format** for better performance.
- * MySQL **validates** the JSON data before storing it.
- * JSON columns can be **indexed** using generated columns or functional indexes.
- * It is supported starting from **MySQL 5.7.8** and improved in later versions.



JSON Data



Table

Syntax:

```
CREATE TABLE table_name (
    column_name JSON
);
```

Example :

 Create a table with JSON column

```
CREATE TABLE customers (
    id INT PRIMARY KEY,
    info JSON
);
```

Example Data :

id	info (JSON)
1	{ "name": "Alice", "age": 25, "city": "Hyderabad" }
2	{ "name": "Bob", "age": 30, "city": "Bangalore" }
3	{ "name": "Charlie", "age": 22, "city": "Chennai" }

Key Points :

- ✓ JSON is stored in binary format for efficiency.
- ✓ MySQL validates JSON data before inserting.
- ✓ Use JSON data type for semi-structured data.
- ✓ Supported from MySQL 5.7.8 and above.
- ✓ Can be indexed using generated columns.



Key Takeaway :

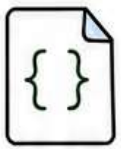


MySQL JSON data type allows you to store and query **semi-structured data** flexibly and efficiently.



25.3 Creating JSON Data

- * We can create JSON data manually using **JSON text format**.
- * JSON data must be enclosed in **single quotes** when inserting into a table.
- * Keys must be in **double quotes** and strings must be in **double quotes**.
- * **Numbers, true, false** and **null** are used without quotes.
- * Arrays are written inside **square brackets []**.
- * Objects are written inside **curly braces { }**.



JSON



MySQL

Syntax :

```
INSERT INTO table_name (json_column)
VALUES ('{ "key1" : value1, "key2" : value2, ... }');
```

Example :

```
-- Insert JSON data into a table
INSERT INTO students (info)
VALUES ('{ "name" : "Alice", "age" : 21,
          "courses" : ["SQL", "Python", "Java"],
          "active" : true }');
```

Output :

id	info (JSON)
1	{ "name" : "Alice", "age" : 21, "courses" : ["SQL", "Python", "Java"], "active" : true }

Key Takeaway :



You can manually create JSON data using **JSON text format** and store it in a JSON column.

Key Points :

- ✓ JSON data is created using standard JSON text format.
- ✓ Always enclose JSON value in **single quotes** in SQL.
- ✓ Keys and string values use **double quotes**.
- ✓ Arrays use **[]** and objects use **{ }**.
- ✓ JSON column stores semi-structured data.

Sample JSON Values :

- Object : { "id" : 1, "name" : "Bob" }
- Array : [10, 20, 30]
- Nested : { "person" : { "name" : "Eve", "age" : 25 } }



25.4 JSON_OBJECT()

- * **JSON_OBJECT()** is a MySQL function used to create a JSON object.
- * It takes **key-value pairs** as arguments and returns a JSON object.
- * Keys must be strings (in **single quotes**).
- * Values can be column values, expressions, numbers, strings, etc.
- * Very useful while selecting data in JSON format.

Syntax :

```
JSON_OBJECT( 'key1', value1, 'key2', value2, ... )
```

Example :

```
SELECT id,
       JSON_OBJECT( 'name', name,
                    'age', age,
                    'course', course,
                    'active', active ) AS student_json
FROM students;
```

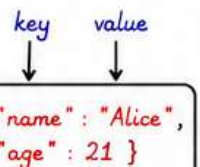
Output :

id	student_json (JSON)
1	{ "name" : "Alice", "age" : 21, "course" : "SQL", "active" : true }
2	{ "name" : "Bob", "age" : 22, "course" : "Python", "active" : false }
3	{ "name" : "Charlie", "age" : 23, "course" : "Java", "active" : true }

Key Takeaway :



Use **JSON_OBJECT()** to easily build JSON objects from column values or expressions in your **SELECT** queries.



JSON_OBJECT()



MySQL

Key Points :

- ✓ **JSON_OBJECT()** creates a JSON object.
- ✓ Provide **key-value pairs** as arguments.
- ✓ Keys must be strings (in **single quotes**).
- ✓ Values can be columns, expressions, literals, or functions.
- ✓ It returns a single JSON object.

Another Example :

```
SELECT JSON_OBJECT( 'id', id,
                    'name', name,
                    'marks', (maths + science) )
AS result_json
FROM students;
```



25.5 JSON_ARRAY()

- * **JSON_ARRAY()** is a MySQL function used to create a JSON array.
- * It takes **values** as arguments and returns a JSON array.
- * Values can be column values, expressions, numbers, strings, etc.
- * Very useful while storing lists or multiple values in JSON format.

Syntax :

```
JSON_ARRAY( value1, value2, value3, ... )
```

Example :

```
SELECT id, JSON_ARRAY(name, age, course) AS student_array
FROM students;
```

Output :

id	student_array (JSON)
1	["Alice", 21, "SQL"]
2	["Bob", 22, "Python"]
3	["Charlie", 23, "Java"]

Key Takeaway :



Use **JSON_ARRAY()** to create JSON **arrays** from a list of values and store multiple items in a single JSON column.

Key Points :

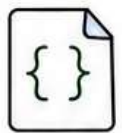
- ✓ **JSON_ARRAY()** creates a JSON array.
- ✓ Values are passed as arguments.
- ✓ Can contain any valid JSON data type.
- ✓ Helpful for storing multiple values as a list.
- ✓ Returns a single JSON array.

Another Example :

```
SELECT JSON_ARRAY(10, 'Hello', true, NULL)
AS example_array;
```

Output :

example_array (JSON)
[10, "Hello", true, null]



JSON



MySQL

{ 1, 2, 3 }

JSON Array



25.6 JSON_EXTRACT()

- * **JSON_EXTRACT()** is used to extract data from a JSON document.
- * It takes a JSON document and a **path expression** as arguments.
- * It returns the matched value(s) from the JSON document.
- * Path expression follows **JSON Path Syntax**.
- * Useful for reading parts of JSON data stored in a column.

Syntax :

```
JSON_EXTRACT( json_doc, path )
```

Example Table :

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad" } }
2	{ "name": "Bob", "age": 22, "skills": ["Java", "C"], "address": { "city": "Bangalore" } }

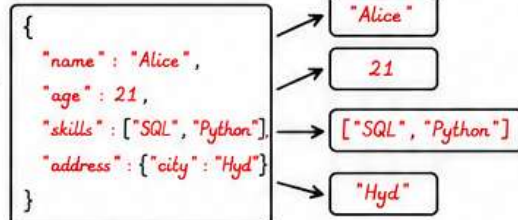
JSON Path Syntax (Common Patterns) :

Pattern	Meaning
\$.key	Get value of key
\$.key1.key2	Get nested value
\$.array[*]	Get all elements of an array
\$.array[0]	Get element at index 0

Key Takeaway :



Use **JSON_EXTRACT()** to read specific values from JSON data using **JSON Path** expressions.



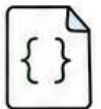
JSON_EXTRACT()

Examples :

Query	Result (JSON)
1 SELECT JSON_EXTRACT(info, '\$.name') FROM students;	"Alice" "Bob"
2 SELECT JSON_EXTRACT(info, '\$.age') FROM students;	21 22
3 SELECT JSON_EXTRACT(info, '\$.skills') FROM students;	["SQL", "Python"] ["Java", "C"]
4 SELECT JSON_EXTRACT(info, '\$.address.city') FROM students;	"Hyderabad" "Bangalore"

Key Points :

- ✓ **JSON_EXTRACT()** extracts data from a JSON document.
- ✓ Requires JSON document and **path expression**.
- ✓ Uses **JSON Path Syntax**.
- ✓ Returns the matched value(s) as JSON.
- ✓ Very useful for reading nested JSON data.



JSON



MySQL



25.7 JSON_VALUE()

- * **JSON_VALUE()** is used to extract a **scalar value** from a JSON document.
- * It returns a value of type **VARCHAR(512)** or the specified data type.
- * It is similar to **JSON_EXTRACT()**, but returns unquoted scalar values.
- * If the path does not match or the value is not a scalar, it returns **NULL**.
- * Useful when we want a single value from JSON data.

Syntax:

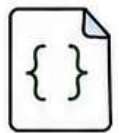
```
JSON_VALUE( json_doc, path [ RETURNING data_type ]
           [ NULL ON EMPTY | ERROR ON EMPTY ] )
```

Example Table:

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad" } }
2	{ "name": "Bob", "age": 22, "skills": ["Java", "C"], "address": { "city": "Bangalore" } }

Examples:

Query	Result
1 SELECT JSON_VALUE(info, '\$.name') FROM students;	"Alice" "Bob"
2 SELECT JSON_VALUE(info, '\$.age') FROM students;	21 22
3 SELECT JSON_VALUE(info, '\$.address.city') FROM students;	Hyderabad Bangalore
4 SELECT JSON_VALUE(info, '\$.skills[0]') FROM students;	SQL Java
5 SELECT JSON_VALUE(info, '\$.skills[1]') FROM students;	Python C



JSON



MySQL

"Alice"
JSON_VALUE()



Key Points:

- ✓ **JSON_VALUE()** returns scalar values (string, number, boolean, null).
- ✓ Returns **VARCHAR(512)** by default or specified data type.
- ✓ If no match found → **NULL** (or as per option).
- ✓ Use **NULL ON EMPTY** or **ERROR ON EMPTY** for better control.
- ✓ Useful for retrieving single values from JSON data.

Example with returning type:

```
SELECT JSON_VALUE(info, '$.age' RETURNING UNSIGNED)
AS age_num FROM students;
```

Key Takeaway:



Use **JSON_VALUE()** when you need a single scalar value from JSON data in a specific data type.

25.8 JSON_SET()

- * **JSON_SET()** is used to **set or update values** in a JSON document.
- * It returns a new JSON document with the changes.
- * It does not modify the original JSON document.
- * If the path does not exist, it adds the new value.
- * We can set multiple path-value pairs in a single call.

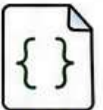
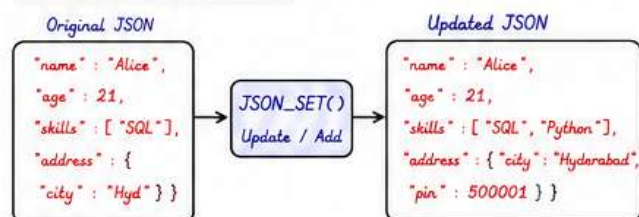
Syntax:

```
JSON_SET( json_doc, path, value [, path, value] ... )
```

Example Table:

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "pin": 500001 } }
2	{ "name": "Bob", "age": 22, "skills": ["Java"], "address": { "city": "Bangalore" } }

How JSON_SET() Works:



JSON



MySQL

Example:

Query	Result (JSON)
1 SELECT JSON_SET(info, '\$.age', 25) AS updated_info FROM students WHERE id = 1;	{ "name": "Alice", "age": 25, ... }
2 SELECT JSON_SET(info, '\$.address.city', 'Mumbai') FROM students WHERE id = 1;	{ ... "city": "Mumbai", ... }
3 SELECT JSON_SET(info, '\$.skills[1]', 'Java') FROM students WHERE id = 1;	{ ... "skills": ["SQL", "Java", ...] }
4 SELECT JSON_SET(info, '\$.address.pin', 500002) FROM students WHERE id = 2;	{ ... "pin": 500002 }
5 SELECT JSON_SET(info, '\$.name', 'Alice K', '\$.skills', JSON_ARRAY('SQL', 'Python', 'ML')) FROM students WHERE id = 1;	"name": "Alice K", "skills": ["SQL", "Python", "ML", ...]

Key Points:

- ✓ **JSON_SET()** updates or adds values in JSON document.
- ✓ It returns a new JSON document.
- ✓ Original JSON data remains unchanged.
- ✓ Can update multiple paths in a single call.
- ✓ If path does not exist, it is created.

Key Takeaway:



Use **JSON_SET()** to update or add values in JSON data and return the modified JSON document.



25.9 JSON_INSERT()

- * **JSON_INSERT()** is used to **insert values** into a JSON document.
- * It inserts only if the path **does not already exist**.
- * If the path exists, the original value remains unchanged.
- * It returns a new JSON document with the inserted values.
- * We can insert multiple path-value pairs in a single call.

Syntax:

`JSON_INSERT( json_doc, path, value [, path, value] ... )`

Example Table:

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad" } }
2	{ "name": "Bob", "age": 22, "skills": ["Java", "address": { "city": "Bangalore" } }

Key Takeaway:



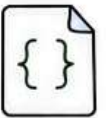
Use **JSON_INSERT()** to add new values only when the specified path does not already exist in the JSON document.

Key Points:

- ✓ **JSON_INSERT()** adds values only if the path **does not exist**.
- ✓ If the path exists, no changes are made.
- ✓ Returns a new JSON document.
- ✓ Original JSON document is not modified.
- ✓ Can insert multiple path-value pairs.

Examples:

Query	Result (JSON)
1 <code>SELECT JSON_INSERT(info, '\$.phone', '9000011111') FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad"}, "phone": "9000011111" }
2 <code>SELECT JSON_INSERT(info, '\$.address.state', 'TS') FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "TS" } }
3 <code>SELECT JSON_INSERT(info, '\$.skills[1]', 'C++') FROM students WHERE id = 2;</code>	{ "name": "Bob", "age": 22, "skills": ["Java", "C++"], "address": { "city": "Bangalore" } }
4 <code>SELECT JSON_INSERT(info, '\$.email', 'alice@example.com', \$.address.pincode, 500081) FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "email": "alice@example.com", "address": { "city": "Hyderabad", "pincode": 500081 } }



JSON



MySQL



Insert



25.10 JSON_REPLACE()

- * **JSON_REPLACE()** is used to **replace values** in a JSON document.
- * It replaces the value only if the path **already exists**.
- * If the path does not exist, no changes are made.
- * It returns a new JSON document with the replaced values.
- * We can replace multiple path-value pairs in a single call.

Syntax:

`JSON_REPLACE( json_doc, path, value [, path, value] ... )`

Example Table:

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "TS", "phone": "9000011111" } }
2	{ "name": "Bob", "age": 22, "skills": ["Java", "C++"], "address": { "city": "Bangalore" } }

Key Takeaway:



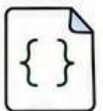
Use **JSON_REPLACE()** to update **existing values** in a JSON document. It modifies only the paths that already exist.

Key Points:

- ✓ **JSON_REPLACE()** replaces the value only if the path **exists**.
- ✓ If the path does not exist, the JSON document remains unchanged.
- ✓ Returns a new JSON document.
- ✓ Original JSON document is not modified.
- ✓ Can replace multiple path-value pairs.

Examples:

Query	Result (JSON)
1 <code>SELECT JSON_REPLACE(info, '\$.phone', '9123456789') FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "TS", "phone": "9123456789" } }
2 <code>SELECT JSON_REPLACE(info, '\$.address.city', 'Secunderabad') FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Secunderabad", "state": "TS", "phone": "9000011111" } }
3 <code>SELECT JSON_REPLACE(info, '\$.skills[1]', 'JavaScript') FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "JavaScript"], "address": { "city": "Hyderabad", "state": "TS", "phone": "9000011111" } }
4 <code>SELECT JSON_REPLACE(info, \$.email, 'alice@college.com', \$.address.state, 'Telangana') FROM students WHERE id = 1;</code>	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "Telangana", "phone": "9000011111", "email": "alice@college.com" } }



JSON



MySQL



Replace

Note: **INSERT** adds new values only if the path **does not exist**.
REPLACE updates values only if the path **already exists**.

→ Both functions return a new JSON document.



25.11 JSON_REMOVE()

- * **JSON_REMOVE()** is used to **remove a value** from a JSON document.
- * It removes the value at the specified **path(s)**.
- * If the path exists, the value is removed.
- * If the path does not exist, the JSON document remains unchanged.
- * It returns a new JSON document with the removed value(s).
- * We can remove multiple path-value pairs in a single call.

Syntax:

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

Example Table:

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "TS" }, "phone": "9000011111" }
2	{ "name": "Bob", "age": 22, "skills": ["Java", "C++"], "address": { "city": "Bangalore", "state": "KA" } }

Key Takeaway:



Use **JSON_REMOVE()** to delete values from a JSON document at the specified path(s) and get a new JSON document.

Key Points:

- ✓ **JSON_REMOVE()** removes the value at the specified path(s).
- ✓ If the path does not exist, no changes are made.
- ✓ Returns a new JSON document.
- ✓ Original JSON document is not modified.
- ✓ Can remove multiple paths in a single call.



JSON



MySQL



Remove



Examples:

Query	Result (JSON)
1 SELECT JSON_REMOVE(info, '\$.phone') FROM students WHERE id = 1;	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "TS" } }
2 SELECT JSON_REMOVE(info, '\$.address.city') FROM students WHERE id = 1;	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "state": "TS" }, "phone": "9000011111" }
3 SELECT JSON_REMOVE(info, '\$.skills[0]') FROM students WHERE id = 2;	{ "name": "Bob", "age": 22, "skills": ["C++"], "address": { "city": "Bangalore", "state": "KA" } }
4 SELECT JSON_REMOVE(info, '\$.address', '\$.phone') FROM students WHERE id = 1;	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"] }
5 SELECT JSON_REMOVE(info, '\$.skills[1]', '\$.age') FROM students WHERE id = 2;	{ "name": "Bob", "skills": ["Java"], "address": { "city": "Bangalore", "state": "KA" } }

25.12 JSON_TABLE()

- * **JSON_TABLE()** is used to convert JSON data into a **relational table**.
- * It maps JSON data to columns and rows.
- * It is very useful for querying JSON data using SQL.
- * Supports nested data, arrays and objects.
- * Available from MySQL 8.0.4 and above.

Syntax:

```
JSON_TABLE( json_doc, path  
COLUMNS (  
    column_name data_type PATH path_expr [ NULL | NOT NULL ],  
    ...  
    nested_path COLUMNS (  
        column_name data_type PATH path_expr [ NULL | NOT NULL ]  
    )  
) AS alias;
```

Example Table:

id	info (JSON)
1	{ "name": "Alice", "age": 21, "skills": ["SQL", "Python"], "address": { "city": "Hyderabad", "state": "TS" } }
2	{ "name": "Bob", "age": 22, "skills": ["Java", "C++"], "address": { "city": "Bangalore", "state": "KA" } }

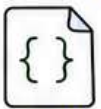
Key Takeaway:



Use **JSON_TABLE()** to transform JSON data into relational rows and columns for easy querying.

Key Points:

- ✓ **JSON_TABLE()** converts JSON data into rows and columns.
- ✓ Requires column definitions with PATH expressions.
- ✓ Supports nested paths and arrays.
- ✓ Can be used in SELECT, JOIN, WHERE, etc.
- ✓ Returns a virtual table (derived table).



JSON



MySQL



Table



Examples:

Query	Result															
1 SELECT s.id, jt.name, jt.age FROM students s, JSON_TABLE(s.info, '\$' COLUMNS (name VARCHAR(50) PATH '\$.name', age INT PATH '\$.age') AS jt;	<table><tr><th>id</th><th>name</th><th>age</th></tr><tr><td>1</td><td>Alice</td><td>21</td></tr><tr><td>2</td><td>Bob</td><td>22</td></tr></table>	id	name	age	1	Alice	21	2	Bob	22						
id	name	age														
1	Alice	21														
2	Bob	22														
2 SELECT s.id, jt.skill FROM students s, JSON_TABLE(s.info, '\$.skills[*]' COLUMNS (skill VARCHAR(50) PATH '\$') AS jt;	<table><tr><th>id</th><th>skill</th></tr><tr><td>1</td><td>SQL</td></tr><tr><td>1</td><td>Python</td></tr><tr><td>2</td><td>Java</td></tr><tr><td>2</td><td>C++</td></tr></table>	id	skill	1	SQL	1	Python	2	Java	2	C++					
id	skill															
1	SQL															
1	Python															
2	Java															
2	C++															
3 SELECT s.id, jt.city, jt.state FROM students s, JSON_TABLE(s.info, '\$.address' COLUMNS (city VARCHAR(50) PATH '\$.city', state VARCHAR(50) PATH '\$.state') AS jt;	<table><tr><th>id</th><th>city</th><th>state</th></tr><tr><td>1</td><td>Hyderabad</td><td>TS</td></tr><tr><td>2</td><td>Bangalore</td><td>KA</td></tr></table>	id	city	state	1	Hyderabad	TS	2	Bangalore	KA						
id	city	state														
1	Hyderabad	TS														
2	Bangalore	KA														
4 -- Nested example: extract skills inside objects SELECT s.id, jt.name, sk.skill FROM students s, JSON_TABLE(s.info, '\$' COLUMNS (name VARCHAR(50) PATH '\$.name', NESTED PATH '\$.skills[*]' COLUMNS (skill VARCHAR(50) PATH '\$')) AS jt JOIN JSON_TABLE(s.info, '\$.skills[*]' COLUMNS (skill VARCHAR(50) PATH '\$')) AS sk ON s.id = jt.id;	<table><tr><th>id</th><th>name</th><th>skill</th></tr><tr><td>1</td><td>Alice</td><td>SQL</td></tr><tr><td>1</td><td>Alice</td><td>Python</td></tr><tr><td>2</td><td>Bob</td><td>Java</td></tr><tr><td>2</td><td>Bob</td><td>C++</td></tr></table>	id	name	skill	1	Alice	SQL	1	Alice	Python	2	Bob	Java	2	Bob	C++
id	name	skill														
1	Alice	SQL														
1	Alice	Python														
2	Bob	Java														
2	Bob	C++														

Note: **JSON_TABLE()** makes it possible to work with JSON data just like normal relational tables.



25.13 JSON_Indexing

- * **JSON indexing** improves the performance of queries on JSON data.
- * MySQL allows indexing on values extracted from JSON documents using **generated columns** (virtual or stored).
- * We create an index on the generated column.
- * This helps in faster search, filtering and sorting on JSON data.

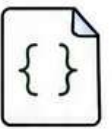
Example Table :

id	info (JSON)
1	{ "name" : "Alice", "age" : 21, "skills" : ["SQL", "Python"], "address" : { "city" : "Hyderabad" } }
2	{ "name" : "Bob", "age" : 22, "skills" : ["Java", "C++"], "address" : { "city" : "Bangalore" } }
3	{ "name" : "Charlie", "age" : 23, "skills" : ["Python", "ML"], "address" : { "city" : "Mumbai" } }

Problem :



If we run a query like
WHERE JSON_EXTRACT(info, '\$.age') > 21,
 MySQL scans the whole table (full table scan).
 To optimize, we need an index.



JSON



MySQL



Step 1 : Add a Generated Column

```
ALTER TABLE students
ADD COLUMN age INT
GENERATED ALWAYS AS (JSON_UNQUOTE(JSON_EXTRACT(info, '$.age'))) STORED ;
```

Step 2 : Create an Index on the Generated Column

```
CREATE INDEX idx_age
ON students(age);
```

Now the table looks like :

id	info (JSON)	age (Generated)
1	{ "name" : "Alice", "age" : 21, ... }	21
2	{ "name" : "Bob", "age" : 22, ... }	22
3	{ "name" : "Charlie", "age" : 23, ... }	23

Step 3 : Use the Index in Queries

```
SELECT id, JSON_EXTRACT(info, '$.name') AS name, age
FROM students
WHERE age > 21;
```



MySQL will use the index **idx_age** for faster filtering.

More Examples of Indexing JSON Values :

1. Index on nested value (city)

```
ALTER TABLE students
ADD COLUMN city VARCHAR(50)
GENERATED ALWAYS AS (JSON_UNQUOTE(JSON_EXTRACT(info, '$.address.city'))) STORED ;

CREATE INDEX idx_city ON students(city);
```

2. Index on array element (first skill)

```
ALTER TABLE students
ADD COLUMN first_skill VARCHAR(50)
GENERATED ALWAYS AS (JSON_UNQUOTE(JSON_EXTRACT(info, '$.skills[0]'))) STORED ;

CREATE INDEX idx_first_skill ON students(first_skill);
```

Important Notes :

- * Only extracted scalar values can be indexed (string, number, date, etc.).
- * JSON documents, arrays or objects cannot be indexed directly.
- * Generated columns can be **VIRTUAL** or **STORED**.
- * **STORED** columns are faster for indexing.
- * MySQL uses **B-TREE** index for these columns.

Summary :

Concept	How to Do	Example
Extract JSON value	Use JSON_EXTRACT()	JSON_EXTRACT(info, '\$.age')
Add generated column	ALTER TABLE ... ADD COLUMN ... GENERATED ALWAYS AS (JSON_EXTRACT(...)) STORED ;	age INT GENERATED ALWAYS AS (JSON_UNQUOTE(JSON_EXTRACT(info, '\$.age'))) STORED
Create index	CREATE INDEX index_name ON table(column);	CREATE INDEX idx_age ON students(age);
Use in query	Query using the generated column	WHERE age > 21;

Key Takeaway :



Use **generated columns** to extract values from JSON and create **indexes** on them for better performance.