PostgreSQL JSON & JSONB Cheatsheet

A comprehensive cheat sheet for working with JSON and JSONB data types in PostgreSQL, covering operators, functions, indexing, and performance tips.

JSON/JSONB Operators

Access Operators

`->` (int/text)	Access JSON array element (by index) or object field (by key). **Examples:** `'{"a": [1, 2, 3]}'::json -> 1` # Returns `2` (JSON) `'{"a": {"b": "c"}}'::json -> 'a'` # Returns `{"b": "c"}` (JSON)
`->>` (int/text)	Access JSON array element (by index) or object field (by key) as text. **Examples:** `'{"a": [1, 2, 3]}'::json ->> 1` # Returns `2` (text) `'{"a": {"b": "c"}}'::json ->> 'a'` # Returns `{"b": "c"}` (text)
`#>` (text[])	Access JSON element at the specified path. **Example:** `'{"a": {"b": [1, 2, 3]}}'::json #> '{a,b,1}'` # Returns `2` (JSON)
`#>>` (text[])	Access JSON element at the specified path as text. **Example:** `'{"a": {"b": [1, 2, 3]}}'::json #>> '{a,b,1}'` # Returns `2` (text)

Containment and Existence Operators

`?` (text)	Check if key exists within JSON object or element exists within JSON array.
	
	Examples:
	`'{"a": 1, "b": 2}'::jsonb ? 'a'` # Returns `true`
	`'[1, 2, 3]'::jsonb ? '2'` # Returns `true`

`?|` (text[])	Check if any of the keys in the text array exist within the JSON object or if any of the elements exist within the JSON array. **Example:** `'{"a": 1, "b": 2}'::jsonb ?| array['a', 'c']` # Returns `true`
`?&` (text[])	Check if all of the keys in the text array exist within the JSON object or if all of the elements exist within the JSON array. **Example:** `'{"a": 1, "b": 2}'::jsonb ?& array['a', 'b']` # Returns `true`
`@>` (jsonb)	Check if the left JSON contains the right JSON as a sub-object or sub-array (containment). **Example:** `'{"a": 1, "b": 2}'::jsonb @> '{"a": 1}'::jsonb` # Returns `true`
`<@` (jsonb)	Check if the right JSON contains the left JSON as a sub-object or sub-array (contained in). **Example:** `'{"a": 1}'::jsonb <@ '{"a": 1, "b": 2}'::jsonb` # Returns `true`

`-` (text or int)	Delete a key (text) from a JSON object or an element (int) from a JSON array.
	
	Examples:
	`'{"a": 1, "b": 2}'::jsonb - 'a'` # Returns `{"b": 2}`
	`'[1, 2, 3]'::jsonb - 1` # Returns `[1, 3]`

JSON/JSONB Functions

JSONB Creation and Manipulation

`to_jsonb(anyelement)`	Converts any SQL value to JSONB. **Example:** `to_jsonb('hello'::text)` # Returns `"hello"`
`jsonb_build_object(VARIADIC "any")`	Builds a JSONB object from a variadic list of key/value pairs. **Example:** `jsonb_build_object('key1', 1, 'key2', 'value2')` # Returns `{"key1": 1, "key2": "value2"}`
`jsonb_build_array(VARIADIC "any")`	Builds a JSONB array from a variadic list of values. **Example:** `jsonb_build_array(1, 'two', null)` # Returns `[1, "two", null]`

`jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])`	Replaces a value inside a JSONB object or array. If `create_missing` is true, missing keys will be created.
	
	Example:
	`jsonb_set('{"a": 1}'::jsonb, '{a}', '2'::jsonb)` # Returns `{"a": 2}`
	`jsonb_set('{"a": 1}'::jsonb, '{b}', '2'::jsonb, true)` # Returns `{"a": 1, "b": 2}`

`jsonb_insert(target jsonb, path text[], new_value jsonb, insert_after boolean)`	Inserts a new value into a JSONB array. If `insert_after` is true, the value is inserted after the specified element, otherwise before.
	
	Example:
	`jsonb_insert('[1, 2]'::jsonb, '{1}', '3'::jsonb, true)` # Returns `[1, 2, 3]`
	`jsonb_insert('[1, 2]'::jsonb, '{1}', '3'::jsonb, false)` # Returns `[1, 3, 2]`

`jsonb_strip_nulls(jsonb)`	Removes all null values from the specified JSONB value. **Example:** `jsonb_strip_nulls('{"a": 1, "b": null}'::jsonb)` # Returns `{"a": 1}`

JSONB Navigation and Extraction

`jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])`	Extracts JSONB value at the given path. **Example:** `jsonb_extract_path('{"a": {"b": 2}}'::jsonb, 'a', 'b')` # Returns `2`
`jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])`	Extracts JSONB value at the given path as text. **Example:** `jsonb_extract_path_text('{"a": {"b": 2}}'::jsonb, 'a', 'b')` # Returns `2` (text)
`jsonb_object_keys(jsonb)`	Returns a set of keys in the outermost JSONB object. **Example:** `jsonb_object_keys('{"a": 1, "b": 2}'::jsonb)` # Returns `a`, `b`
`jsonb_array_length(jsonb)`	Returns the number of elements in the JSONB array. **Example:** `jsonb_array_length('[1, 2, 3]'::jsonb)` # Returns `3`
`jsonb_array_elements(jsonb)`	Expands the outermost JSONB array into a set of JSONB elements. **Example:** `SELECT value FROM jsonb_array_elements('[1, {"a": "b"}, 3]'::jsonb)` # Returns rows with `1`, `{"a": "b"}`, `3`
`jsonb_each(jsonb)`	Expands the outermost JSONB object into a set of key-value pairs. **Example:** `SELECT key, value FROM jsonb_each('{"a": 1, "b": "val"}'::jsonb)` # Returns rows with `a`, `1` and `b`, `"val"`

JSONB Indexing

GIN Indexes

GIN (Generalized Inverted Index) indexes are very useful for indexing JSONB columns, especially when you need to search for keys or values within the JSONB data.

Key Points:

- GIN indexes are lossy, meaning they may require rechecking the actual row to confirm a match.
- They are best suited for queries that involve existence (`?`, `?|`, `?&`) and containment (`@>`, `<@`) operators.
- GIN indexes can be created on expressions, allowing you to index specific parts of the JSONB data.

Creating a GIN index for JSONB:

```
CREATE INDEX idx_data_gin ON users USING GIN (data);
```

Creating a GIN index on a specific key:

```
CREATE INDEX idx_data_path_gin ON users USING GIN ((data -> 'key'));
```

Using GIN index for existence checks:

```
SELECT * FROM users WHERE data ? 'name';
```

Using GIN index for containment checks:

```
SELECT * FROM users WHERE data @> '{"city": "New York"}'::jsonb;
```

BRIN Indexes

BRIN (Block Range Index) indexes are suitable when the JSONB data has a natural correlation with the physical order of the table (e.g., time-series data).

Key Points:

- BRIN indexes are much smaller than GIN indexes.
- They are less effective for random access patterns.
- They work best when data is inserted in a way that nearby rows have similar JSONB values.

Creating a BRIN index for JSONB:

```
CREATE INDEX idx_data_brin ON users USING BRIN (data);
```

Using BRIN index (effectiveness depends on data order):

```
SELECT * FROM users WHERE (data ->> 'timestamp')::timestamp BETWEEN '2023-01-01' AND '2023-01-31';
```

Expression Indexes

Expression indexes allow you to index specific parts of the JSONB data, improving query performance for specific use cases.

Key Points:

- You can create indexes on JSONB accessors (`->`, `->>`) or functions.
- This is useful when you frequently query a specific field or transform the JSONB data.

Creating an expression index on a JSONB key:

```
CREATE INDEX idx_data_name ON users ((data ->> 'name'));
```

Creating an expression index on a function applied to JSONB data:

```
CREATE INDEX idx_data_timestamp ON users (((data ->> 'timestamp')::timestamp));
```

Using expression indexes:

```
SELECT * FROM users WHERE data ->> 'name' = 'John';
```

```
SELECT * FROM users WHERE (data ->> 'timestamp')::timestamp > '2023-01-15';
```

JSON Performance Tips

Choosing Between JSON and JSONB

JSON:

- Stores the data as plain text.
- Preserves the exact formatting, including whitespace and key order.
- Parsing required on every access, making reads slower.

JSONB:

- Stores the data in a decomposed binary format.
- Eliminates insignificant whitespace and key order.
- Faster to process due to the parsed format.
- Takes up more space due to indexing and storage overhead.

In most cases, **JSONB is the preferred choice** due to its superior query performance. Use JSON only when you need to preserve the exact original formatting of the JSON data.

Query Optimization

Use Indexes:

- Create appropriate indexes based on your query patterns. Use GIN indexes for existence and containment checks, and expression indexes for specific key lookups.

Avoid Full Table Scans:

- Ensure your queries are using indexes to avoid full table scans, which can be slow for large tables.

Use Specific Operators:

- Use the correct operators (`->`, `->>`, `#>`, `#>>`) to extract data. Use `->>` when you only need the text representation.

Use Prepared Statements:

- If you have dynamic queries or repeated queries, utilize prepared statements to improve performance.

Casting:

- When comparing JSONB values cast them properly if needed. Type casting is crucial for performance when comparing values within JSONB columns. For example, if comparing a timestamp stored as text in JSONB, cast it to a timestamp for efficient querying:

```
SELECT * FROM users WHERE (data ->> 'timestamp')::timestamp > '2023-01-15';
```

Data Modeling Considerations

Avoid Storing Large JSON Documents:

- If possible, break down large JSON documents into smaller, more manageable parts or separate columns.

Use Consistent Data Types:

- Ensure the data types within your JSON documents are consistent to avoid unexpected behavior and improve query performance.

Normalize Data:

- Consider normalizing frequently accessed data into separate columns to avoid parsing JSONB data every time.