Putting data into Postgres as JSON

Olga Shevchuk

Ways of storing JSON data in Postgres

- Two different data types for storing JSON data: JSON and JSONB
- JSON data type stores an exact copy of the JSON input text → parsing is necessary to retrieve a particular field.
- JSONB is stored data in a decomposed binary format; that is, not as an ASCII/UTF-8 string, but as binary code → no parsing is needed.

Here are some valid JSONB expressions.

```
-- Simple scalar/primitive value
-- Primitive values can be numbers, quoted strings, true, false, or null
SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

JSON vs JSONB

```
oshevchuk=> select '{"user_id":1,    "paying":true,"paying":false}'::json, '{"user_id":1, "paying":true, "paying":false}'::jsonb;
                      json                      |              jsonb
------------------------------------------------+---------------------------------
 {"user_id":1,    "paying":true,"paying":false} | {"paying": false, "user_id": 1}
(1 row)
```

The whitespace, the order of the keys and the duplicate keys are preserved in the JSON column but not in the JSONB column

Using JSONB

Creating a table with a JSONB column (equivalent for JSON column):

```
create table sales (
    id serial not null primary key,
    info jsonb not null
);
```

```
                                    Table "public.sales"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-----------------------------------
 id     | integer |           | not null | nextval('sales_id_seq'::regclass)
 info   | jsonb   |           | not null |
```

Inserting a JSON document:

```
insert into sales values (1, '{"name": "Alice", "paying": true, "tags": ["admin"]}');
```

```
oshevchuk=> select * from sales;
 id |                          info
----+-------------------------------------------------------
  1 | {"name": "Alice", "tags": ["admin"], "paying": true}
(1 row)
```

Updating a JSON document

Updating by inserting a whole document:

```
update sales set info = '{"name": "Bob", "paying": false, "tags": []}';
```

```
oshevchuk=> select * from sales;
 id |                      info
----+-------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
(1 row)
```

If more than 1 row, to update data at a specific row, use where

```
update sales set info = '{"name": "Emily", "paying": true, "tags": ["user"], "salary": 3000}' where id = 2;
```

Original

```
 id |                                                      info
----+---------------------------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
  2 | {"name": "Amy", "tags": ["user"], "paying": true, "salary": 3000}
(2 rows)
```

Changed

```
 id |                                                      info
----+---------------------------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
  2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
(2 rows)
```

Updating a JSON document

Updating by adding a key (only JSONB)

```
update sales set info = info || '{"country": "Canada"}';
```

```
 id |                                     info
----+----------------------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false, "country": "Canada"}
  2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000, "country": "Canada"}
(2 rows)
```

Updating by removing a key

```
update sales set info = info - 'country';
```

```
 id |                              info
----+----------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
  2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
(2 rows)
```

Querying the JSON document

Two operators to query JSON documents: -> and ->>. Work on both JSON as well as JSONB columns.

- ->, #> returns a column of JSON objects
- ->>, #>> returns a column of values of type text

Get by key:

```
select info->'name' as name from sales;
```

```
  name
---------
 "Bob"
 "Emily"
(2 rows)
```

Can chain -> operator

```
id |                                                    info
---+------------------------------------------------------------------------------
 1 | {"name": "Bob", "tags": [], "paying": false}
 2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
 3 | {"name": "Alice", "agent": {"bot": true}}
```

```
select info->'agent'->'bot' as boolean from sales where id = 3;
```

```
 boolean
---------
 true
(1 row)
```

Operator	Right Operand Type	Description	Usage Example	Example Results
->	int	Get JSON array element (indexed from zero, negative integers count from the end)	'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2	{"c":"baz"}
->	text	Get JSON object field by key	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	Get JSON array element as text	'[1,2,3]'::json->>2	3
->>	text	Get JSON object field as text	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	Get JSON object at specified path	'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	Get JSON object at specified path as text	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'	3

https://www.postgresql.org/docs/9.5/functions-json.html

Filter results

One can filter a result set using the where clause through JSON keys

```
select * from sales where info->'paying' = 'false';
```

```
 id |                                 info
----+-------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
```

JSONB Containment with @>

The containment operator **@>** tests whether one document (a set or an array) contains another.

```
select '{"name": "Alice", "agent": {"bot": true} }'::jsonb @> '{"agent": {"bot": false}}';
-- returns false

select '{"name": "Alice", "agent": {"bot": true} }'::jsonb @> '{"agent": {"bot": true}}';
-- return true
```

```
id |                                  info
---+-------------------------------------------------------------------------
 1 | {"name": "Bob", "tags": [], "paying": false}
 2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
 3 | {"name": "Alice", "tags": ["user", "admin"], "agent": {"bot": true}}
```

```
select info->'name' from sales where info->'tags' @> '["user"]';
```

```
"Emily"
"Alice"
```

JSONB Existence with ?

We can use the JSONB existence operator ? to check if an object key or array element is present:

```
select '{"name": "Alice", "agent": {"bot": true} }'::jsonb -> 'agent' ? 'bot';   //true
```

Checking for array element presence:

```
 id |                                  info
----+-------------------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
  2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
  3 | {"name": "Alice", "tags": ["user", "admin"], "agent": {"bot": true}}
```

```
select info->'name' from sales where info->'tags' ? 'user';
```

```
"Emily"
"Alice"
```

JSON and JSONB functions

jsonb_each(jsonb) (or **json_each(json)**): expands the top-level JSON object into a set of key-value pairs.

```
select * from jsonb_each( '{"name": "Alice", "agent": {"bot": true} }'::jsonb );
```

```
  key   |      value
--------+---------------
 name   | "Alice"
 agent  | {"bot": true}
(2 rows)
```

JSON and JSONB functions

json_object_keys(json) or **jsonb_object_keys(jsonb)** returns the set of keys of the top-level JSON object.

```
select jsonb_object_keys( '{"name": "Alice", "agent": {"bot": true} }'::jsonb );
```

```
 jsonb_object_keys
-------------------
 name
 agent
(2 rows)
```

JSON and JSONB functions

json_extract_path(json, text[]) or **jsonb_extract_path(jsonb, text[])** returns a JSON object pointed to by a “path” of type text[] (equivalent to #> operator)

```
select jsonb_extract_path( '{"name": "Alice", "agent": {"bot": true} }'::jsonb, 'agent', 'bot');
```

```
 jsonb_extract_path
--------------------
 true
```

JSON and JSONB functions

jsonb_pretty(jsonb) returns the indented JSON text.

```
select jsonb_pretty( '{"name": "Alice", "agent": {"bot": true} }'::jsonb );
```

```
     jsonb_pretty
----------------------
 {
     "name": "Alice",
     "agent": {
         "bot": true
     }
 }
(1 row)
```

JSON and JSONB functions

jsonb_array_elements(jsonb) or **jsonb_array_elements_text(jsonb)** expands a json array to a set of json/text values. Enables to use aggregate functions on the output.

```
 id |                                        info
----+-------------------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
  2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
  3 | {"name": "Alice", "tags": ["user", "admin"], "agent": {"bot": true}}
```

```
select jsonb_array_elements_text(info->'tags') as tags from sales where id = 3;
```

```
 tags
-------
 user
 admin
(2 rows)
```

```
select jsonb_array_elements(info->'tags') as tags from sales where id = 3;
```

```
   tags
---------
 "user"
 "admin"
```

Creating indices(JSONB)

```
 id |                                  info
----+-------------------------------------------------------------------------
  1 | {"name": "Bob", "tags": [], "paying": false}
  2 | {"name": "Emily", "tags": ["user"], "paying": true, "salary": 3000}
  3 | {"name": "Alice", "tags": ["user", "admin"], "agent": {"bot": true}}
```

```
CREATE INDEX people_paying ON sales ((info->'paying'));
```

- This index will automatically speed up all the aggregate functions that we run on people who are paying (where info->’paying’ = true) because of the people_paying index.
- For efficiency, it is recommended to index anything that is subject to be used on a where clause when filtering results.

Indexing with JSONB

WithOut Index

```
jsonb=# select * from customer where name->>'id' =
'cust000000001';

-[ RECORD 1 ]--+--------------------------------------------------
-----------------------------------------------

id             | cust000000001

name           | {"id": "cust000000001", "title": "Dr.",
"initial": "Q", "last name": "Kordon", "first name": "Bob"}

customer_since |


Time: 430.838 ms
```

With Index

```
jsonb=# CREATE INDEX customer_jsonbid_idx ON customer ((name-
>>'id'));


jsonb=# select * from customer where name->>'id' =
'cust000000001';

-[ RECORD 1 ]--+--------------------------------------------------
-----------------------------------------------

id             | cust000000001

name           | {"id": "cust000000001", "title": "Dr.",
"initial": "Q", "last name": "Kordon", "first name": "Bob"}

customer_since |


Time: 0.640 ms
```

JSONB benefits and drawbacks

JSONB **benefits:**

- more efficiency,
- significantly faster to process due to binary representation
- supports indexing
- simpler schema designs

JSONB **drawbacks**:

- slightly slower input (due to added conversion overhead),
- it may take more disk space than plain JSON due to a larger table footprint
- certain queries (especially aggregate ones) may be slower due to the lack of statistics.

References

- https://www.blendo.co/blog/storing-json-in-postgresql/
- https://www.postgresql.org/docs/9.4/datatype-json.html#:~:text=jsonb%20also%20supports%20indexing%2C%20which,of%20keys%20within%20JSON%20objects.
- https://postgresconf.org/system/events/document/000/001/001/PGCONF-JSONB.pdf
- When To Avoid JSONB In A PostgreSQL Schema - Heap
- Querying JSON (JSONB) data types in PostgreSQL · Advanced SQL · SILOTA
- Faster Operations with the JSONB Data Type in PostgreSQL - Compose Articles
- https://www.infoworld.com/article/3651356/jsonb-in-postgresql-today-and-tomorrow.html#:~:text=The%20key%20capabilities%20of%20JSONB,take%20those%20one%20by%20one.