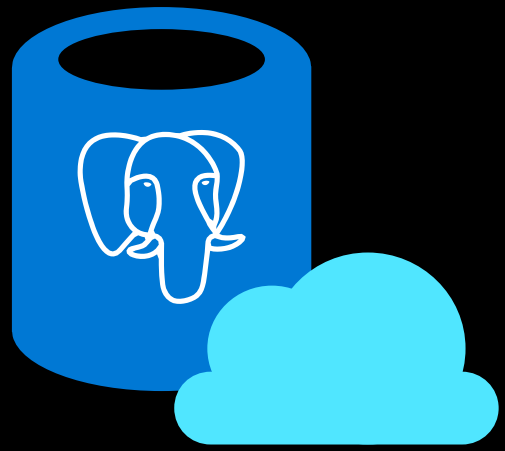
Microsoft

JSONB Tricks: Operators, Indexes, and When to (Not) Use It

Colton Shepard

SF Bay Area Postgres Meetup: Virtual

May 2020



Who am I?

Colton Shepard

Former Citus Data employee, now at Microsoft thanks to the acquisition

Postgres and Citus implementation and migration specialist

Work with on-premise, Azure, edge, and other cloud environments

What is all this, anyways?

JavaScript Object Notation Data Interchange Format

RFC 7158

Human readable lightweight data format

One of 2 Postgres JSON data types.

Decomposed for storage, not stored as string

- Fast and indexable
- High disk usage

```
{
    ”sundae": [{
        "icecream": "strawberry",
        "scoops": "1"
    }, {
        "icecream": ”vanilla",
        "topping": "sprinkles",
        "scoops": "2"
    }]
}
```

Good use cases

Unpredictable data structures

Lots of attributes that are rarely used

Replacing Entity–attribute–value model tables

Arbitrary fields with dynamic values

Customer Story!

The Microsoft VeniceDB project, for Windows telemetry data

Incoming metrics are JSONB formatted

After initial processing/aggregation pass, some JSONB is kept

Easy to add new metrics without major database changes

Process 5TB of data per day (2800 core Citus cluster)

Less Good Use Cases: Statistics Gathering Issues

Postgres gathers statistics on column contents and distribution

When you run a query, it uses these statistics to plan how it runs

Unfortunately, it doesn't get statistics on JSONB values

So, Postgres guesses how to run JSONB queries. Sometimes, it guesses wrong

We often see this when filtering on rarely used keys

This patch has helped, especially for GROUP BY, but we haven't used it widely.

https://www.postgresql.org/message-id/flat/20200113230008.g67iyk4cs3xbnjju%40development

Other Less Good Use Cases

Fields that (almost) always exist

Indexing Concerns

Disk space limitations

Unnecessary future-proofing (e.g. a data type that may change)

Customer Story: Heap Analytics

Use case is event collection where users can attach arbitrary tags to events

This is mostly a good use case- JSONB is great for arbitrary tags!

Unfortunately, JSONB keys are stored in full, so disk usage is magnified

Saved 30% disk usage by turning the top 45 keys into columns

At 1+ PB on a Citus distributed Postgres instance, that's a lot of savings!

Source: https://heap.io/blog/engineering/when-to-avoid-jsonb-in-a-postgresql-schema

Customer Story: Anonymous, Inc.

This is not a current customer, and they will remain anonymous

Tag-based search engine where users could fuzzy match fields and values

Timestamp and other metadata had columns, JSONB for all tags

Queries as written used LIKE operator against keys and values

Could not be indexed, and were basically doing full table scans for all queries

We recommended moving to column based storage, but it didn’t work out

Operators

Operators are stock functions for manipulating and accessing data types

Here are some common examples that you probably recognize

JSONB has its own special ones, as follows.

IS NULL ||

LIKE lower()

> AND

&& *

+ =

-> and ->>

Get JSONB object field by key:

```
citus=> select customerinfo -> 'loyaltyinfo'->'loyaltyprogram' from icecream_orders limit 2;
     ?column?
------------------
 "legacy"
 "printed coupon"
(2 rows)
```

Get same value as text:

```
citus=> select customerinfo -> 'loyaltyinfo'->>'loyaltyprogram' from icecream_orders limit 2;
    ?column?
----------------
 legacy
 printed coupon
(2 rows)
```

The ? operator

Does this string exist as a top-level key within the JSON value?

```
citus=> select customerinfo -> 'name',product from icecream_orders where product ? 'regularcone';
        ?column?         |                                                    product
-------------------------+-------------------------------------------------------------------------------------------
 "John Smith"            | {"regularcone": [{"scoops": "1", "icecream": "vanilla"}]}
 "Unknown Small Child #3" | {"regularcone": [{"scoops": "1", "icecream": "vanilla"}, {"scoops": "1", "icecream": "rockyroad"}]}
(2 rows)
```

Mostly used for filters on key existence

The ? operator

What about this array of strings?

```
citus=> select customerinfo -> 'name',product from icecream_orders where product ?| array['regularcone','wafflecone'];;
        ?column?         |                                          product
-------------------------+-----------------------------------------------------------------------------------------------
 "John Smith"            | {"regularcone": [{"scoops": "1", "icecream": "vanilla"}]}
 "Unknown Small Child #3" | {"regularcone": [{"scoops": "1", "icecream": "vanilla"}, {"scoops": "1", "icecream": "rockyroad"}]}
 "Unknown"               | {"wafflecone": [{"scoops": "7", "icecream": "chocolate"}]}
(3 rows)
```

?& for every element of the array

?| for any element of the array

The ? operator

You can dig deeper if you like:

```
citus=> select customerinfo from icecream_orders where customerinfo -> 'loyaltyinfo' ? 'loyaltynumber';
                                                customerinfo
-------------------------------------------------------------------------------------------------------
 {"name": "John Smith", "loyaltyinfo": {"loyaltynumber": "23523", "loyaltyprogram": "legacy"}}
 {"name": "Unknown", "loyaltyinfo": {"loyaltynumber": "12", "loyaltyprogram": "competitor-Frozen Rock"}}
(2 rows)
```

The @ operator

Does the right JSONB element exist at the top level of the left element?

```
citus=> select customerinfo from icecream_orders where customerinfo @> '{"name":"John Smith"}'::jsonb;
                                                customerinfo
-------------------------------------------------------------------------------------------
 {"name": "John Smith", "loyaltyinfo": {"loyaltynumber": "23523", "loyaltyprogram": "legacy"}}
(1 row)
```

Same logic, other direction:

```
citus=> select customerinfo from icecream_orders where '{"name":"John Smith"}'::jsonb <@ customerinfo ;
                                                customerinfo
-------------------------------------------------------------------------------------------
 {"name": "John Smith", "loyaltyinfo": {"loyaltynumber": "23523", "loyaltyprogram": "legacy"}}
(1 row)
```

#> and #>>

Let’s get the values at this path as a JSON object:

```
citus=> select customerinfo,customerinfo #> '{loyaltyinfo,loyaltyprogram}' from icecream_orders limit 2;
                                    customerinfo                                     |     ?column?
-------------------------------------------------------------------------------------+------------------
 {"name": "John Smith", "loyaltyinfo": {"loyaltynumber": "23523", "loyaltyprogram": "legacy"}} | "legacy"
 {"name": "Unknown Small Child #3", "loyaltyinfo": {"loyaltyprogram": "printed coupon"}}       | "printed coupon"
(2 rows)
```

Let’s get the same values as text instead:

```
citus=> select customerinfo,customerinfo #>> '{loyaltyinfo,loyaltyprogram}' from icecream_orders limit 2;
                                    customerinfo                                     |    ?column?
-------------------------------------------------------------------------------------+----------------
 {"name": "John Smith", "loyaltyinfo": {"loyaltynumber": "23523", "loyaltyprogram": "legacy"}} | legacy
 {"name": "Unknown Small Child #3", "loyaltyinfo": {"loyaltyprogram": "printed coupon"}}       | printed coupon
(2 rows)
```

Operator-friendly indexes

You may have noticed that not all indexes get good results here

Most have very limited operator coverage

Others don’t even work

Lots of indexing advice is for older PG versions

B-Tree

A b-tree index on a jsonb column only handles equality operators

So, it’s nice but it’s not our most flexible option:

```
citus=> CREATE INDEX idx_customerinfo_btree ON icecream_orders USING BTREE(product);
CREATE INDEX
citus=> explain select * from icecream_orders where product = '{"giftcard": "50"}'::jsonb;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx_customerinfo_btree on icecream_orders  (cost=0.13..4.15 rows=1 width=72)
   Index Cond: (product = '{"giftcard": "50"}'::jsonb)
(2 rows)
```

B-Tree Function Index

We can use function indexes to get a bit more functionality out of B-TREE:

```
CREATE INDEX idx_btree_part ON icecream_orders USING BTREE((customerinfo ->> 'name'));
```

```
citus=> explain select * from icecream_orders where customerinfo ->> 'name' = 'John Smith';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_btree_part on icecream_orders  (cost=0.13..4.15 rows=1 width=190)
   Index Cond: ((customerinfo ->> 'name'::text) = 'John Smith'::text)
(2 rows)
```

We’re limited on LIKE operators, though

Hash

Same as b-tree: Equality only, but you can index a function or operator result

LIKE doesn't work, but you can compare JSON directly:

```
citus=> CREATE INDEX idx_product_hash ON icecream_orders USING HASH (product);
CREATE INDEX
citus=> explain select * from icecream_orders where product = '{"giftcard": "50"}'::jsonb;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_product_hash on icecream_orders  (cost=0.00..4.02 rows=1 width=72)
   Index Cond: (product = '{"giftcard": "50"}'::jsonb)
(2 rows)
```

Hash function indexes

Hash gives us some useful results when we move to a function index.

It still won't use the index for LIKE operations, even with seqscan disabled:

```
citus=> CREATE INDEX idx_customerinfo_hash_part ON icecream_orders USING HASH((customerinfo ->> 'name'));
CREATE INDEX
citus=> explain select customerinfo  from icecream_orders where customerinfo ->> 'name' = 'John Smith';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx_customerinfo_hash_part on icecream_orders  (cost=0.00..4.02 rows=1 width=32)
   Index Cond: ((customerinfo ->> 'name'::text) = 'John Smith'::text)
(2 rows)

citus=> explain select customerinfo  from icecream_orders where customerinfo ->> 'name' LIKE 'John%';
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Seq Scan on icecream_orders  (cost=10000000000.00..10000000001.08 rows=1 width=32)
   Filter: ((customerinfo ->> 'name'::text) ~~ 'John%'::text)
(2 rows)
```

GiST, SP-GiST, and BRIN

Apparently not…

```
citus=> CREATE INDEX idx_product_gist ON icecream_orders USING gist (product);
ERROR:  data type jsonb has no default operator class for access method "gist"
HINT:  You must specify an operator class for the index or define a default operator class for the data type.
```

Function indexes work fine:

```
citus=> CREATE INDEX idx_customerinfo_gist_part ON icecream_orders USING gist((customerinfo ->> 'name'));
CREATE INDEX
citus=> explain select * from icecream_orders where customerinfo ->> 'name' = 'John Smith';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx_customerinfo_gist_part on icecream_orders  (cost=0.13..4.15 rows=1 width=72)
   Index Cond: ((customerinfo ->> 'name'::text) = 'John Smith'::text)
(2 rows)
```

GIN

This probably looks familiar if you've read up on this topic, but what does it mean?

66.2. Built-in Operator Classes

The core PostgreSQL distribution includes the GIN operator classes shown in Table 66.1. (Some of the optional modules described in Appendix F provide additional GIN operator classes.)

Table 66.1. Built-in GIN Operator Classes

Name	Indexed Data Type	Indexable Operators
array_ops	anyarray	&& <@ = @>
jsonb_ops	jsonb	? ?& ?| @>
jsonb_path_ops	jsonb	@>
tsvector_ops	tsvector	@@ @@@

Of the two operator classes for type jsonb, jsonb_ops is the default. jsonb_path_ops supports fewer operators but offers better performance for those operators. See Section 8.14.4 for details.

GIN with jsonb_path_ops

```
citus=> CREATE INDEX idx_product_gin_jsonb_path_ops ON icecream_orders USING gin (product jsonb_path_ops);
CREATE INDEX
citus=> explain select product from icecream_orders where product @> '{"regularcone":[]}';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on icecream_orders  (cost=22.00..24.01 rows=1 width=32)
   Recheck Cond: (product @> '{"regularcone": []}'::jsonb)
   ->  Bitmap Index Scan on idx_product_gin_jsonb_path_ops  (cost=0.00..22.00 rows=1 width=0)
         Index Cond: (product @> '{"regularcone": []}'::jsonb)
(4 rows)

citus=> explain select product from icecream_orders where product @> '{"regularcone": [{"icecream":"vanilla"}]}';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on icecream_orders  (cost=4.00..6.01 rows=1 width=32)
   Recheck Cond: (product @> '{"regularcone": [{"icecream": "vanilla"}]}'::jsonb)
   ->  Bitmap Index Scan on idx_product_gin_jsonb_path_ops  (cost=0.00..4.00 rows=1 width=0)
         Index Cond: (product @> '{"regularcone": [{"icecream": "vanilla"}]}'::jsonb)
(4 rows)
```

We can get a whole lot out of the @ operator, but we are limited to just using it.

GIN with jsonb_ops (default)

```
citus=> CREATE INDEX idx_product_gin_jsonb_ops ON icecream_orders USING gin (product jsonb_ops);
CREATE INDEX
citus=> explain select * from icecream_orders where product ? 'regularcone';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on icecream_orders  (cost=4.00..6.01 rows=1 width=72)
   Recheck Cond: (product ? 'regularcone'::text)
   ->  Bitmap Index Scan on idx_product_gin_jsonb_ops  (cost=0.00..4.00 rows=1 width=0)
         Index Cond: (product ? 'regularcone'::text)
(4 rows)

citus=> explain select * from icecream_orders where product ?| array['regularcone','pinecone'];
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on icecream_orders  (cost=6.00..8.01 rows=1 width=72)
   Recheck Cond: (product ?| '{regularcone,pinecone}'::text[])
   ->  Bitmap Index Scan on idx_product_gin_jsonb_ops  (cost=0.00..6.00 rows=1 width=0)
         Index Cond: (product ?| '{regularcone,pinecone}'::text[])
(4 rows)
```

Special mention: GIN function index with gin_trgm_ops

We've seen good results for indexing specific keys for LIKE matching

Unfortunately, you have to do this for each key

```
citus=> CREATE INDEX idx_customerinfo_gin_trgm_ops_part ON icecream_orders USING gin((customerinfo ->> 'name') gin_trgm_ops);
CREATE INDEX
citus=> explain select * from icecream_orders where customerinfo ->> 'name' LIKE 'John %';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on icecream_orders  (cost=12.00..14.02 rows=1 width=72)
   Recheck Cond: ((customerinfo ->> 'name'::text) ~~ 'John %'::text)
   ->  Bitmap Index Scan on idx_customerinfo_gin_trgm_ops_part  (cost=0.00..12.00 rows=1 width=0)
         Index Cond: ((customerinfo ->> 'name'::text) ~~ 'John %'::text)
(4 rows)
```

This is super situational, but it's got us a 10X performance boost once or twice

Best used when there's few keys and you want to use LIKE on the values

Postgres 12

New datatype: jsonpath

Provides a binary representation of the parsed SQL/JSON path expression

1. ***.key*** returns object member with specified ***key***
2. **.*** returns all object members at current level
3. **.**** returns all object members at current level and below
4. **.**{*level*}** or **.**{*start_level* to *end_level*}** returns all at specified level(s)
5. **[*subscript*, ...]** returns the value at specified array location
6. **[*]** returns all array elements.

https://www.postgresql.org/docs/12/functions-json.html#FUNCTIONS-SQLJSON-OP-TABLE

Postgres 12: The jsonb_path_query operator

jsonb_path_query lets us get what's at a specific path.

Here, we see a jsonpath object being used exactly like **->**

```
pg12=# select customerinfo -> 'name', jsonb_path_query(customerinfo,'$.name')
from icecream_orders limit 2;
         ?column?          |     jsonb_path_query
---------------------------+---------------------------
 "John Smith"              | "John Smith"
 "Unknown Small Child #3" | "Unknown Small Child #3"
(2 rows)
```

This is probably the simplest use case

Postgres 12: jsonb_path_query with .* and .**

The column data:

```
{"regularcone": [{"scoops": "1", "icecream": "vanilla"}]}
```

```
pg12=# select jsonb_path_query(product,'$.*' ) from icecream_orders where id = 1;
            jsonb_path_query
-----------------------------------------
 [{"scoops": "1", "icecream": "vanilla"}]
(1 row)
```

```
pg12=# select jsonb_path_query(product,'$.**' ) from icecream_orders where id = 1;
                      jsonb_path_query
-----------------------------------------------------------
 {"regularcone": [{"scoops": "1", "icecream": "vanilla"}]}
 [{"scoops": "1", "icecream": "vanilla"}]
 {"scoops": "1", "icecream": "vanilla"}
 "1"
 "vanilla"
(5 rows)
```

Postgres 12: jsonb_path_query with .**{level}

We can see everything that's at this depth, regardless of intermediate keys:

```
pg12=# select id, jsonb_path_query(product,'$.**{2}') from icecream_orders;
 id |             jsonb_path_query
----+------------------------------------------
  1 | {"scoops": "1", "icecream": "vanilla"}
  2 | {"scoops": "1", "icecream": "vanilla"}
  2 | {"scoops": "1", "icecream": "rockyroad"}
  3 | {"scoops": "1", "icecream": "strawberry"}
  3 | {"scoops": "2", "topping": "sprinkles"}
  4 | {"scoops": "7", "icecream": "chocolate"}
(6 rows)
```

You can also enter a range of levels, but that is really hard to see on a slide :(

Postgres 12: jsonb_path_query with arrays

Let’s look at the first scoop in the array of scoops on each cone:

```
pg12=# select id, jsonb_path_query(product,'$.*[0]') from icecream_orders;
 id |              jsonb_path_query
----+-------------------------------------
  1 | {"scoops": "1", "icecream": "vanilla"}
  2 | {"scoops": "1", "icecream": "vanilla"}
```

Now, let’s do the same for just waffle cones:

```
pg12=# select id, jsonb_path_query(product,'$.waffleconе[0]') from icecream_orders;
 id |              jsonb_path_query
----+-------------------------------------
  4 | {"scoops": "7", "icecream": "chocolate"}
(1 row)
```

Postgres 12: the jsonb_path_exists operator

Gives bool indicating whether the path exists.

```
pg12=# select product,jsonb_path_exists(product,'$.wafflecone') from icecream_orders;
                                             product                                             | jsonb_path_exists
-------------------------------------------------------------------------------------------------+-------------------
 {"regularcone": [{"scoops": "1", "icecream": "vanilla"}]}                                       | f
 {"regularcone": [{"scoops": "1", "icecream": "vanilla"}, {"scoops": "1", "icecream": "rockyroad"}]} | f
 {"sundae": [{"scoops": "1", "icecream": "strawberry"}, {"scoops": "2", "topping": "sprinkles"}]}   | f
 {"wafflecone": [{"scoops": "7", "icecream": "chocolate"}]}                                      | t
 {"giftcard": "50"}                                                                              | f
(5 rows)
```

Useful for filters on key existence like this:

```
pg12=# select product from icecream_orders where jsonb_path_exists(product,'$.wafflecone');
                          product
------------------------------------------------------------
 {"wafflecone": [{"scoops": "7", "icecream": "chocolate"}]}
(1 row)
```

Postgres 12: the jsonb_path_match operator

This allows for any logic that’ll output Boolean results

```
pg12=# select product from icecream_orders where jsonb_path_match(product,'$.*.scoops == "2"');
                                                 product
-----------------------------------------------------------------------------------------
 {"sundae": [{"scoops": "1", "icecream": "strawberry"}, {"scoops": "2", "topping": "sprinkles"}]}
(1 row)
```

Postgres 12: the jsonb_path_query_array operator

Much like jsonb_path_query, but you get an array instead:

```
pg12=# select jsonb_path_query_array(product,'$.regularcone[*]') from icecream_orders where product ? 'regularcone';
                              jsonb_path_query_array
-------------------------------------------------------------------------------
 [{"scoops": "1", "icecream": "vanilla"}]
 [{"scoops": "1", "icecream": "vanilla"}, {"scoops": "1", "icecream": "rockyroad"}]
(2 rows)
```

For comparison:

```
pg12=# select jsonb_path_query(product,'$.regularcone[*]') from icecream_orders where product ? 'regularcone';
             jsonb_path_query
---------------------------------------
 {"scoops": "1", "icecream": "vanilla"}
 {"scoops": "1", "icecream": "vanilla"}
 {"scoops": "1", "icecream": "rockyroad"}
(3 rows)
```

Postgres 12: the jsonb_path_query_first operator

Get first matching result:

```
pg12=# select jsonb_path_query_first(product,'$.regularcone[*]') from icecream_orders where product ? 'regularcone';
         jsonb_path_query_first
---------------------------------------
 {"scoops": "1", "icecream": "vanilla"}
 {"scoops": "1", "icecream": "vanilla"}
(2 rows)
```

Conclusion

JSONB is awesome for unstructured and semi-structured data

If you have arbitrary queryable fields, you should try it

Be careful of disk usage and planner issues

It’s tricky to index, but you have some good options

Postgres 12 has some extra cool options

General resources to learn more

VeniceDB architecture talk:
https://www.youtube.com/watch?v=AeMaBwd90SI

Heap Analytics blog on When to Avoid jsonb:
https://heap.io/blog/engineering/when-to-avoid-jsonb-in-a-postgresql-schema

Bitnine blog on JSONB indexing:
https://bitnine.net/blog-postgresql/postgresql-internals-jsonb-type-and-its-indexes/

Postgres 12 jsonpath information:
https://www.postgresql.org/docs/12/functions-json.html#FUNCTIONS-SQLJSON-PATH

Michael Paquier's announcement about jsonpath:
https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

Citus Data blog post on JSONB in distributed Postgres:
https://www.citusdata.com/blog/2016/07/25/sharding-json-in-postgres-and-performance/

Questions?

Contact @Azure Database for MySQL, PostgreSQL & MariaDB
Twitter: @AzureDBPostgres @citusdata