Schinckel.net

Querying JSON in Postgres

Posted:
2014-05-25 @ 06:55:15
Tags:
postgres json
sql
Comments:
58 Comments.

Yesterday, I discovered how you can enable jsonb in postgres/psycopg2.

Today, I experimented around with how to query the data in json columns. There is documentation, but it wasn't initially clear to me how the different operations worked.

```
CREATE TABLE json_test (
  id serial primary key,
  data jsonb
);

INSERT INTO json_test (data) VALUES
  ('{}'),
  ('{"a": 1}'),
  ('{"a": 2, "b": ["c", "d"]}'),
  ('{"a": 1, "b": {"c": "d", "e": true}}'),
  ('{"b": 2}');
```

So far, so good. Let's see what's in there, to check:

```
SELECT * FROM json_test;
```

```
 id |                 data
----+--------------------------------------
  1 | {}
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  5 | {"b": 2}
(5 rows)
```

Super. Let's have a go at filtering those results. There are several operators that we can use (and we'll soon see why we chose jsonb).

Equality

Only available for jsonb, we can test that two JSON objects are identical:

```
SELECT * FROM json_test WHERE data = '{"a":1}';
```

```
 id | data
----+------
  1 | {"a": 1}
(1 row)
```

Containment

Again, jsonb only, we can see if one JSON object

```
SELECT * FROM json_test WHERE data @> '{"a":1}';
```

Give me all objects that contain the key "a", with the value 1 associated with that key:

```
 id |                 data
----+--------------------------------------
  2 | {"a": 1}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
(2 rows)
```

Containment goes both ways:

```
SELECT * FROM json_test WHERE data <@ '{"a":1}';
```

In this case, we can see that the query object is a superset of the empty object, as well as matching exactly to object 2.

```
 id |   data
----+----------
  1 | {}
  2 | {"a": 1}
(2 rows)
```

Key/element existence

The last batch of jsonb only operators: we can test for the existence of a key (or an element of type string in an array, but we'll get to those later).

```
SELECT * FROM json_test WHERE data ? 'a';
```

Give me all objects that have the key a.

```
 id |                 data
----+--------------------------------------
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
(3 rows)
```

We can also test for objects that have *any* of a list of keys:

```
SELECT * FROM json_test WHERE data ?| array['a', 'b'];
```

```
 id |                 data
----+--------------------------------------
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  5 | {"b": 2}
(4 rows)
```

```
SELECT * FROM json_test WHERE data ?& array['a', 'b'];
```

```
 id |                 data
----+--------------------------------------
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
(2 rows)
```

Key-path traversal

We can also filter records that have a matching key-path. In simple cases, using the containment operators might be simpler, but in more complex situations, we would need to use these. These operations can also be used to extract a value, although at this stage I'm only interested in using them as part of a `WHERE` clause.

```
SELECT * FROM json_test WHERE data ->> 'a' > '1';
```

Give me all the records where the value of the element associated with key `a` is greater than `1`. Notice the need to use a text value, rather than a number. I'm still investigating how this will play out.

```
 id |           data
----+---------------------------
  3 | {"a": 2, "b": ["c", "d"]}
(1 row)
```

We can also do comparisons between primitives, objects and arrays:

```
SELECT * FROM json_test WHERE data -> 'b' > '1';
```

```
 id |                 data
----+--------------------------------------
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  5 | {"b": 2}
(3 rows)
```

So, it seems that arrays and objects sort greater than numbers.

We can also look deeper down the path:

```
SELECT * FROM json_test WHERE data #> '{b,c}' = '"d"';
```

Give me objects where element `b` has a child object that has element `c` equal to the string `"d"`. Neat.

```
 id |                 data
----+--------------------------------------
  4 | {"a": 1, "b": {"c": "d", "e": true}}
```

```
SELECT * FROM json_test WHERE data #>> '{b,c}' = 'd';
```

```
 id |                 data
----+-------------------------------------
  4 | {"a": 1, "b": {"c": "d", "e": true}}
(1 row)
```

Don’t cross the streams…

So, all good so far. We can query stuff, and this same stuff can be used to index jsonb columns, too.

However, the more astute reader may have noticed that I’ve been dealing with json data that has an object as it’s root. This needn’t be the case: arrays are also valid json, indeed so are any of the allowable atoms:

```
SELECT
  'null'::json,
  'true'::json,
  'false'::json,
  '2'::json,
  '1.0001'::json,
  '"abc"'::json,
  '1E7'::jsonb;
```

Note the last one is a jsonb, which converts to canonical form:

```
 json | json | json  | json |  json   | json  |  jsonb
------+------+-------+------+---------+-------+----------
 null | true | false | 2    | 1.00001 | "abc" | 10000000
(1 row)
```

Note also that a json `null` is different to an SQL `NULL`.

So, what happens when we start storing objects of mixed “type” in a json column?

I’m glad you asked.

```
INSERT INTO json_test (data)
VALUES ('[]'), ('[1,2,"a"]'), ('null'), ('1E7'), ('"abc"');

SELECT * FROM json_test;
```

```
 id |                 data
----+-------------------------------------
  1 | {}
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  5 | {"b": 2}
  6 | []
  7 | [1, 2, "a"]
```

So far, so good. We can store those objects. And query?

Equality testing works fine:

```
SELECT * FROM json_test WHERE data = '{"a":1}';
SELECT * FROM json_test WHERE data = 'null';
```

Containment, too works as expected.

```
SELECT * FROM json_test WHERE data @> '{"a":1}';
SELECT * FROM json_test WHERE data <@ '{"a":1}';
```

Key and element existence perform reliably: perhaps surprisingly, the one query will match elements in an array, as well as keys in an object.

```
SELECT * FROM json_test WHERE data ? 'a';
```

```
 id |                 data
----+--------------------------------------
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  7 | [1, 2, "a"]
(4 rows)
```

```
SELECT * FROM json_test WHERE data ?| array['a', 'b'];
```

```
 id |                 data
----+--------------------------------------
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  5 | {"b": 2}
  7 | [1, 2, "a"]
(5 rows)
```

```
SELECT * FROM json_test WHERE data ?& array['a', 'b'];
```

```
 id |                 data
----+--------------------------------------
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
(2 rows)
```

But, as soon as we start doing key or element ‘get’, we hit a problem:

```
SELECT * FROM json_test WHERE data ->> 'a' > '1';

ERROR: cannot call jsonb_object_field_text
       (jsonb ->> text operator) on an array
```

We can still use the key-path traversal, though, unless we

```
SELECT * FROM json_test WHERE data #> '{b,c}' = '"d"' AND id < 8;
```

```
 id |                 data
----+--------------------------------------
  4 | {"a": 1, "b": {"c": "d", "e": true}}
(1 row)
```

Note the syntax for a key path: it only allows for strings (which json keys must be), or integers (which array indices are).

This seems like a pretty severe limitation. I'm not sure how things like MongoDB handle this, but in hindsight, if you are storing both array-based and object-based json data in the one column, you are probably going to be in a world of hurt anyway.

…or, maybe, do cross the streams

All is not lost, however: it's possible to get just the object-based rows:

```
SELECT * FROM json_test WHERE data @> '{}';
```

```
 id |                 data
----+--------------------------------------
  1 | {}
  2 | {"a": 1}
  3 | {"a": 2, "b": ["c", "d"]}
  4 | {"a": 1, "b": {"c": "d", "e": true}}
  5 | {"b": 2}
(5 rows)
```

You could then combine this with a previously-forbidden query:

```
SELECT * FROM json_test WHERE data @> '{}' AND data ->> 'a' > '1';
```

```
 id |           data
----+---------------------------
  3 | {"a": 2, "b": ["c", "d"]}
(1 row)
```

Indeed, postgres is so awesome you don't even need to ensure the `data @> '{}` bit comes first!

But what about limiting to just array-typed data? Turns out we can use the same trick:

```
SELECT * FROM json_test WHERE data @> '[]';
```

```
 id |    data
----+------------
  6 | []
```

```
SELECT * FROM json_test WHERE data @> '[]' AND data ->> 1 = '2';
```

```
 id |    data
----+-------------
  7 | [1, 2, "a"]
(1 row)
```

Worth noting is that the @> operator is only available on jsonb columns, so you won't be able to query mixed-form data in a regular json column.

Wow! What's next?

This foray into querying jsonb data in postgres was an aside to a project I'm working on to bring json(b) querying to django. With django 1.7's new custom lookup features, it will be possible to write things like:

```
# Exact
MyModel.objects.filter(data={'a': 1})
MyModel.objects.exclude(data={})
# Key/element existence
MyModel.objects.filter(data__has='a')
MyModel.objects.filter(data__has_any=['a', 'b'])
MyModel.objects.filter(data__has_all=['a', 'b'])
# Sub/superset of key/value pair testing
MyModel.objects.filter(data__contains={'a': 1})
MyModel.objects.filter(data__in={'a': 1, 'b': 2})
# Get element/field (compare with json)
MyModel.objects.filter(data__get=(2, {'a': 1}))
# Get element/field (compare with scalar, including gt/lt comparisons)
MyModel.objects.filter(data__get=(2, 'a'))
MyModel.objects.filter(data__get__gt=('a', 1))
# key path traversal, compare with json or scalar.
MyModel.objects.filter(data__get=('{a,2}', {'foo': 'bar'}))
MyModel.objects.filter(data__get=('{a,2}', 2))
MyModel.objects.filter(data__get__lte=('{a,2}', 2))
```

I'm still not sure about the lookup names, especially the last set. The name "get" seems a little generic, and maybe we could use different lookup names for the input type, although only integer and string values are permitted.

← older newer →

Schinckel.net

Projects Tags About Contact

♡ Recommend 10 Share Sort by Oldest

Join the discussion…

Nitin Surya • 3 years ago

While using JSONB, I came across a problem:

Lets say I have a column:
x

{a: 10}
{a: 20}
{a: 30}

In this case, how can i query for all the elements containing a = 10 or a = 20. I was not able to find any solution for 'or', e.g.
-- select * from table where a in (10,20);
or so.

How can I query on such condition?

1 ^ | ⌄ • Reply • Share ›

Matthew Schinckel Author > Nitin Surya • 3 years ago

One of the issues is that you need to be querying JSONB with JSONB. That is, the values you want to test against must be JSON objects themselves.

There are a couple of ways you can query this particular case:

```
SELECT * FROM jsonb_test WHERE data -> 'a' IN ('10','20');
```

That queries directly on the key, and relies on the fact that it will coerce '10' into a JSON object. This query will not match '{"a":"10"}', for instance.

You could also use the key path query:

```
SELECT * FROM jsonb_test WHERE data #> '{a}' IN ('10','20');
```

I suspect the former would perform better.

1 ^ | ⌄ • Reply • Share ›

Hari K T • 3 years ago

Nice!.

Thanks for the article.

^ | ⌄ • Reply • Share ›

Mohit Gupta • 3 years ago

While using Json type in postgresql