INFO 610 Fall 2020

Week 9.5, Oct 15, 2020

JSON

chrisfauerbach.github.io/info610_fall_2020/

Diving into JSON data types

- JSON data type
  - https://www.postgresql.org/docs/current/datatype-json.html

Bear with me, two slides of 'manual'

What is JSON?

JSON data types are for storing JSON (JavaScript Object Notation) data, as specified in RFC 7159

Sure, you can store JSON text in the **text** field but you miss out on features of structured data.

PostgreSQL offers two types for storing JSON data: json and jsonb. To implement efficient query mechanisms for these data types, PostgreSQL also provides the jsonpath data type

The json and jsonb data types accept almost identical sets of values as input. The major practical difference is one of efficiency. The json data type stores an exact copy of the input text, which processing functions must reparse on each execution; while jsonb data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. jsonb also supports indexing, which can be a significant advantage.

json vs jsonb

json: raw text stored but guaranteed JSON. Parsing happens at query time

jsonb: parsed jsonb. can be indexed. much more efficient at query time

In general, most applications should prefer to store JSON data as jsonb, unless there are quite specialized needs, such as legacy assumptions about ordering of object keys.

JSON	primitive type	PostgreSQL type Notes
string	text	\u0000 is disallowed, as are Unicode escapes representing characters not available in the database encoding
number	numeric	NaN and infinity values are disallowed
boolean	boolean	Only lowercase true and false spellings are accepted
null	(none)	SQL NULL is a different concept

Casting to JSON

```
-- 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;
```

JSONB table

```
DROP TABLE IF EXISTS users;

CREATE TABLE users (id serial primary key, first_name varchar,
        last_name varchar, phone_number varchar, meta_data jsonb);

INSERT INTO users (first_name, last_name, phone_number, meta_data)
VALUES
('Chris', 'Fauerbach', '8045551212', '{"favorite_color":"purple", "interests":
["gardening", "beekeeping", "teaching", "technology"]}'),
('Joe', 'Schmoe', '5405551212', '{"favorite_color":"red", "interests": ["trucks",
"dirt", "beer"], "dislikes": ["cleanliness"]}'),
('Debbie', 'Downer', '7035551212', '{"favorite_color":"grey", "interests":
["darkness", "complaining"]}');
```

Interact with the data

```
SELECT * FROM users;

SELECT first_name, last_name, meta_data->>'interests' FROM users;

SELECT first_name, last_name FROM users where meta_data->>'favorite_color' =
'grey';

SELECT first_name, last_name FROM users where (meta_data->>'interests')::jsonb ?
'beer';

SELECT count(*) FROM users WHERE meta_data ? 'dislikes';

SELECT jsonb_array_elements_text(meta_data->'interests') as interests FROM users
WHERE id = 1;
```

Indexes

Suppose we have a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);

-- Imagine now we insert 3 million records.
-- 3 million records.
```

and the application issues many queries of the form:

```
SELECT * FROM test1 WHERE content = 'name';
```

Let's talk about how that works?

Find all the instances of 'the' in your favorite book

Page by page, very very slow

Big O(n)

Indexes are the key to this

What is an index?

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

An index is a copy of selected columns of data from a table, called a database key or simply key, that can be searched very efficiently that also includes a low-level disk block address or direct link to the complete row of data it was copied from. Some databases extend the power of indexing by letting developers create indexes on functions or expressions. For example, an index could be created on upper(last_name), which would only store the upper-case versions of the last_name field in the index. Another option sometimes supported is the use of partial indices, where index entries are created only for those records that satisfy some conditional expression. A further aspect of flexibility is to permit indexing on user-defined functions, as well as expressions formed from an assortment of built-in functions.

Creating an Index

```
CREATE INDEX test1_id_index ON test1 (id);
CREATE INDEX test1_content_index ON test1 (content);
```

Now, you can get rows with certain 'content' really quickly.

Imagine you have an 'index' at the back of your book that has explicit references to every instance of the word 'the'.

(yes, it's a HUGE list of places , but , a computer can dig through it SUPER quickly)

Big O(1)

Create an index on JSONB

This is magical

```
CREATE INDEX idxfinished ON users ((data->>'interests'));
```

To do

No explicit new homework assignments.

Keep working on your project.

** Discuss Tuesdays