Unleash the full potential of PostgreSQL and JSON

Charly Batista

PERCONA

I am Charly Batista

I’m the Postgres Tech Lead at Percona. I’m passionate about database and algorithmic efficiency. I’ve been working with Database and Development for more than 20 years always trying to get the best performance of the databases, data structures and algorithms... Other than that, I’m a Brazilian who lives in China for more than 6 years but currently located in South America.

You can find me at https://www.linkedin.com/in/charlybatista
or more contacts info by the end of this presentation


Are you passionate about Open Source?!
We're hiring!
Join us!
#RemoteWork
percona.com/careers

Agenda

- Who am I?
- What is this talk about?
- Json or jsonb?
- Operators and Indexes
- To JSON or Not? Database Design
- Recap
- Questions

What is this talk about?

We’re also going to talk about Postgres, not only json

- Most of the tests shown here were done by my colleague Matt Yonkovit
  - https://www.percona.com/blog/storing-and-using-json-within-postgresql-part-one/
- Should I use json or jsonb?
- How can I access my data?
  - What are the operators for json in Postgres?
  - Do they work with all predicates?
  - Can I compare json objects?
- How about indexes?
  - What are the index available for json in Postgres?
  - Can I use b-tree indexes on json documents?
- How my database design impacts my performance?

What is this talk not about?

- Now, let’s understand what this talk isn’t about:
  - We are not going to talk about internals and its implementation
  - Not going to talk about jsonpath Type today (it deserves its own talk)
  - No history and motivations to use json
    - No fancy graphs today :(
    - No comparisons with other databases
  - No live demo :(
    - It always fails

Json or jsonb?

Json or jsonb?

- JSON type is stored as an exact text copy of what was input
  - Processing functions must reparse on each execution
  - preserve semantically-insignificant white space between tokens
  - if a JSON object within the value contains the same key more than once, all the key/value pairs are kept
    - The processing functions consider the last value as the operative one

Json or jsonb?

- JSONB is stored decomposed
  - No reparsing is needed
  - Does not preserve white space,
  - Does not preserve the order of object keys,
  - Does not keep duplicate object keys.
    - If duplicate keys are specified in the input, only the last value is kept
  - Can use indexes more efficiently
- JSONB is probably going to be your best option in most use cases

Operators and Indexes

Operators and Indexes

- There are operators that work with json and jsonb types

Operator	Right Operand Type	Return type	Description
->	int	json or jsonb	Get JSON array element (indexed from zero, negative integers count from the end)
->	text	json or jsonb	Get JSON object field by key
->>	int	text	Get JSON array element as text
->>	text	text	Get JSON object field as text
#>	text[]	json or jsonb	Get JSON object at the specified path
#>>	text[]	text	Get JSON object at the specified path as text

Operators and Indexes

- There are operators that work with only with jsonb types
- Many of these operators can be indexed by jsonb operator classes
- We need to be careful with the expected operand types
- There are creation functions, for example to_json and to_jsonb
- JSON processing functions
  - jsonb_pretty is one of my favorite :D
  - There are many others like json_each
  - A comprehensive list can be found in the documentation

Operators and Indexes

Operator	Right Operand Type	Description
@>	jsonb	Does the left JSON value contain the right JSON path/value entries at the top level?
<@	jsonb	Are the left JSON path/value entries contained at the top level within the right JSON value?
?	text	Does the *string* exist as a top-level key within the JSON value?
?|	text[]	Do any of these array *strings* exist as top-level keys?
?&	text[]	Do all of these array *strings* exist as top-level keys?
||	jsonb	Concatenate two jsonb values into a new jsonb value
-	text	Delete key/value pair or *string* element from left operand. Key/value pairs are matched based on their key value.
-	text[]	Delete multiple key/value pairs or *string* elements from left operand. Key/value pairs are matched based on their key value.
-	integer	Delete the array element with specified index (Negative integers count from the end). Throws an error if top level container is not an array.
#-	text[]	Delete the field or element with specified path (for JSON arrays, negative integers count from the end)
@?	jsonpath	Does JSON path return any item for the specified JSON value?
@@	jsonpath	Returns the result of JSON path predicate check for the specified JSON value. Only the first item of the result is taken into account. If the result is not Boolean, then null is returned.

Operators and Indexes

- Jsonb can make use of GIN index to improve query performance
- Say we have below data structure

```
create table movies_json (
    ai_myid serial primary key,
    imdb_id varchar(255),
    json_column json not null
);

create unique index movies_json_imdb_idx on movies_json(imdb_id);
CREATE INDEX gin_index ON movies_json USING gin (jsonb_column);


create table movies_jsonb (
    ai_myid serial primary key,
    imdb_id varchar(255),
    jsonb_column jsonb not null
);

create unique index movies_jsonb_imdb_idx on movies_jsonb(imdb_id);
CREATE INDEX movies_jsonb_gin_index ON movies_jsonb USING gin (json_column);
```

Operators and Indexes

- Querying the JSON column we get:

```
movie_json_test=# explain (verbose true, analyze true) select * from movies_json where json_column->>'ti
tle' = 'Avengers: Endgame (2019)';
                                                                  QUERY PLA
N
------------------------------------------------------------------------------------------------------
----------------------------------
 Gather  (cost=1000.00..55775.85 rows=1880 width=1059) (actual time=694.047..2516.879 rows=1 loops=1)
   Output: ai_myid, imdb_id, json_column
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on public.movies_json  (cost=0.00..54587.85 rows=783 width=1059) (actual time=1
905.511..2512.010 rows=0 loops=3)
         Output: ai_myid, imdb_id, json_column
         Filter: ((movies_json.json_column ->> 'title'::text) = 'Avengers: Endgame (2019)'::text)
         Rows Removed by Filter: 125119
         Worker 0:  actual time=2511.276..2511.277 rows=0 loops=1
         Worker 1:  actual time=2511.322..2511.322 rows=0 loops=1
 Planning Time: 0.166 ms
 Execution Time: 2516.897 ms
(12 rows)
```

Operators and Indexes

- While the JSONB column gives us:

```
movie_json_test=# explain (verbose true, analyze true) select * from movies_jsonb where jsonb_column->
>'title' = 'Avengers: Endgame (2019)';
                                                                                                  QUERY PLA
N
-------------------------------------------------------------------------------------------------------------
-------------------------------
 Gather  (cost=1000.00..54116.60 rows=1873 width=1025) (actual time=723.324..726.914 rows=1 loops=1)
   Output: ai_myid, imdb_id, jsonb_column
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on public.movies_jsonb  (cost=0.00..52929.30 rows=780 width=1025) (actual time=
548.982..721.730 rows=0 loops=3)
         Output: ai_myid, imdb_id, jsonb_column
         Filter: ((movies_jsonb.jsonb_column ->> 'title'::text) = 'Avengers: Endgame (2019)'::text)
         Rows Removed by Filter: 125119
         Worker 0:  actual time=720.995..720.995 rows=0 loops=1
         Worker 1:  actual time=202.751..720.994 rows=1 loops=1
 Planning Time: 0.038 ms
 Execution Time: 726.933 ms
(12 rows)
```

Operators and Indexes

- This is a simple test with a small json document
- The more complex the document gets the larger is the difference between them
- The test was pretty consistent over 100 runs:

JSON

Average Time: 2.5492

Min Time: 2.5297428970225155

Max Time: 2.56536191503983

JSONB

Average Time: 0.747

Min Time: 0.7297536049736664

Max Time: 0.7827945239841938

Operators and Indexes

- It’s not all flowers, writes (INSERT/UPDATES) will lose some performance
  - After a few tests we got:
    - Inserting 10K records into a JSON column (via insert into select from): 72.851 ms
    - Inserting 10K records into a JSONB column (via insert into select from): 754.045 ms
  - If you are doing a heavy insert workload you need to plan for it!
- Larger jsonb documents will go to TOAST, losing performance
- Type conversions can ruin the day
  - Making assumptions on the data within your JSON/JSONB columns can cause index invalidation, errors or odd results

Operators and Indexes

- Query the data and “answer” some questions might be difficult
  - Say we have below:

    tjson (

    id serial not null constraint list_tjson_pkey primary key,

    meta jsonb not null

    );
  - **INDEX idx_tjson_jsonb_ops ON tjson USING GIN;**
  - **1 | {"2012-05-29": {"state": "notavailable"}, "2016-06-24": {"state": "available"}}**
  - **2 | {"2001-08-15": {"state": "undefined"}, "2006-10-07": {"state": "notavailable"}, "2013-12-12": {"state": "available"}}**
  - **3 | {"2005-01-02": {"state": "available"}}**

Operators and Indexes

- We want to answer the below questions:
  - Entries where they have a particular state for a given date
    - **select * from tjson where meta->'2021-01-01'->'state' = 'available'**
  - Same as above, but instead the state is one of several
    - **select * from tjson where meta->'2021-01-01'->'state' in ('available', 'some_other_state');**
  - Does the tjson include this date?
    - **select * from tjson where meta->'2021-01-03' is not NULL;**

We want to answer the below questions

- Entries where they have a particular state for a given date
    - select * from tjson where meta->'2021-01-01'->'state' = 'available'

We want to answer the below questions

- Entries where they have a particular state for a given date
  - select * from tjson where meta->'2021-01-01'->'state' = 'available'

  A: select * from tjson where meta @> '{"2021-01-01": {"state": "available"}}';

- What if we use the below notation?

  select * from tjson where meta->'2021-01-01'->'state' @> '"available"'::jsonb;

We want to answer the below questions

- Same as above, but instead the state is one of several
  - select * from tjson where meta->'2021-01-01'->'state' in ('available', 'some_other_state');

We want to answer the below questions

- Same as above, but instead the state is one of several
  - select * from tjson where meta->'2021-01-01'->'state' in ('available', 'some_other_state');

    A: select * from tjson where meta @> ANY (ARRAY['{"2021-06-27":{"state": "available"}}', '{"2021-06-27":{"state": "notavailable"}}']::jsonb[]);

We want to answer the below questions

- Does the tjson include this date?
    - select * from tjson where meta->'2021-01-03' is not NULL;

We want to answer the below questions

- Does the tjson include this date?
  - select * from tjson where meta->'2021-01-03' is not NULL;

  A: select * from tjson where meta @> '{"2021-06-27":{}}';

To JSON or Not? Database Design

To JSON or Not? Database Design

- Those are just some design advises that might help performance and usage
- While jsonb with GIN indexes work for a lot of use cases it’s not a full replacement for everything
- You might want to experiment with generated columns for example
- Maybe store some queriable data in another column outside the data and use expression indexes
- In some cases you just want to go with the old school Normalized Tables, with jsonb stored for easy use!

To JSON or Not? Database Design

- We did some tests illustrate a few of the trade-offs around performance when picking a specific design(1):

Simple Query for Movie Title (random 10 titles)	800ms	0.3ms	0.2ms	0.2ms	0.2ms
Select top 10 movies	841ms	831ms	0.9ms	0.9ms	0.3ms
Select all movies for an actor	1009.0ms	228ms	5588.0ms(2)	5588.0ms(2)	0.8ms
Count movies for a given actor	5071ms	5075ms	5544ms	NA	1ms

To JSON or Not? Database Design

- We used the movie-related tables
- Each test was repeated 100 times, and the average results were listed. Min/Max is available as well
- The nested json for our “cast info” was not easily indexed, there are some things we could do to mitigate this, but it is a bit more complicated

Recap

Recap

- Using JSONB is probably going to be your best option in most use cases if need JSON;
- Be very careful of type conversions and making assumptions on the data within your JSON/JSONB columns. You may get errors or odd results!
- Use the available indexes, generated columns, and expression indexes to gain substantial performance benefits;
- The more straightforward the JSON, the easier it will be to pull out and interact with the data you need;
- Nested JSON data can be pulled out in a few different ways using functions like jsonb_to_rescordset;

Recap

- JSON data whose format changes (elements added or removed) may make using some functions difficult
- JSON within a well-built, designed application and database can be wonderful and offer many great benefits
- JSON just dumped into databases usually won’t scale well with Postgres
- Database design still matters!

Percona stands for evolution

Percona stands for ease-of-use

Percona stands for freedom

Percona & PostgreSQL - Better Together

PERCONA


Are you passionate about Open Source?!
We're hiring!
Join us!
#RemoteWork
percona.com/careers

Questions?

Hope we still have time for some questions, if not…

You can find me on LinkedIn at https://www.linkedin.com/in/charlybatista

Or drop me an email at charly.batista@percona.com

Thank You!