Unleash the Power of Storing JSON in Postgres

Last updated: 2018-02-27

by Leigh Halliday | 22 Comments

Development

Reading Time: 10 minutes

An article by Sarah Mei titled "Why you should never use MongoDB" discusses the issues you'll run into if you try to use a NoSQL database when a relational database would be far superior. An example of this is when data that was thought to be in a silo needs to cross boundaries (what relational DBs are great at). Another example is when you store a user's name in various places for easy access, but when the user updates their name you're forced to find all of those places to make sure their information is up to date.

My experience making websites has been in line with this sentiment: Unless your data objects live in complete silos from one another (and you're sure they will be that way for the foreseeable future), you'll probably be better off using a relational database like Postgres.

Up until fairly recently, you had to make that difficult decision up-front when modelling your data: **document or relational database**? Yes, you could use two separate databases, using each tool for what they're best at. However, you'd be increasing the complexity of your app and also of your development and server environments.

JSON support in Postgres

Postgres has had JSON support for a while, but to be honest it wasn't that great due to a lack of indexing and key extractor methods. With the release of version 9.2, Postgres added native JSON support. You could finally use Postgres as a "NoSQL" database. In version 9.3, Postgres improved on that by adding additional constructor and extractor methods. 9.4 added the ability to store JSON as "Binary JSON" (or JSONB), which strips out insignificant whitespace (not a big deal), adds a tiny bit of overhead when inserting data, but provides a huge benefit when querying it: **indexes**.

With the release of version 9.4, JSON support tried to make the question "Do I use a document or relational database?" unnecessary. Why not have both?

> "Document database? Relational? Why not have both?" via @codeship
>
> CLICK TO TWEET

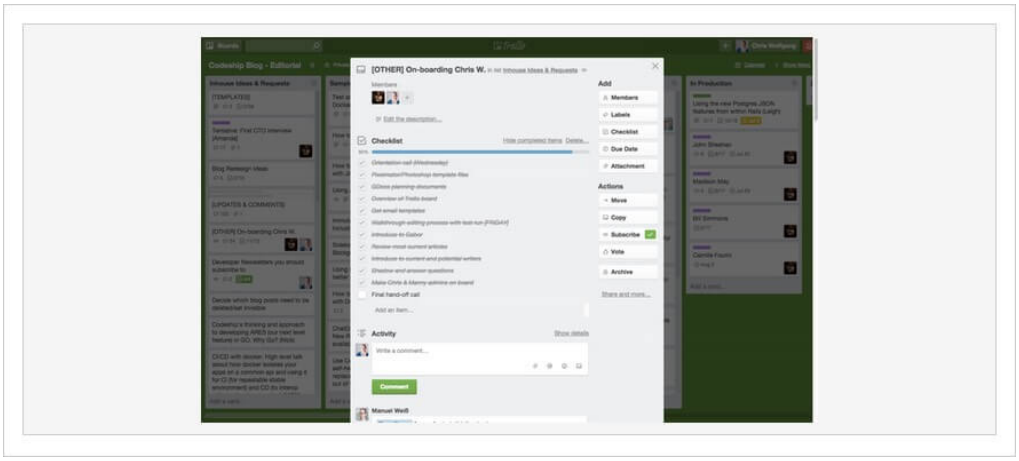
I'm not going to argue that Postgres handles JSON as well as MongoDB. MongoDB was, afterall, specifically made as a JSON document store and has some pretty great features like the aggregation pipeline. But the truth is that Postgres now handles JSON pretty well.

Why would I even want JSON data in my DB?

I still believe that most data is modelled very well using a relational database. The reason for this is because website data tends to be relational. A user makes purchases and leaves reviews, a movie has actors which act in various movies, etc. However, there are use cases where it makes

a lot of sense to incorporate a JSON document into your model. For example, it's perfect when you need to:

Avoid complicated joins on data that is siloed or isolated. Think of something like Trello, where they can keep all information about a single card (comments, tasks, etc…) together with the card itself. Having the data denormalized makes it possible to fetch a card and it's data with a single query.



Maintain data that comes from an external service in the same structure and format (as JSON) that it arrived to you as. What ends up in the database is exactly what the API provided. Look at the charge response object from Stripe as an example; it's nested, has arrays, and so on. Instead of trying to normalize this data across five or more tables, you can store it as it is (and still query against it).

Avoid transforming data before returning it via

your JSON API. Look at this nasty JSON response from the FDA API of adverse drug events. It's deeply nested and has multiple arrays — to build this data real-time on every request would be incredibly taxing on the system.

How to use JSONB in Postgres

Now that we have gone over some of the benefits and use-cases for storing JSON data in Postgres, let's take a look at how it's actually done.

Defining columns

JSONB columns are just like any other data type now. Here's an example of creating a cards table that stores its data in a JSONB column called "data."

```
CREATE TABLE cards (
  id integer NOT NULL,
  board_id integer NOT NULL,
  data jsonb
);
```

Inserting JSON data

To insert JSON data into the database we pass the whole JSON value as a string.

```
INSERT INTO cards VALUES (1, 1, '{"name": "Paint house
INSERT INTO cards VALUES (2, 1, '{"name": "Wash dishes
INSERT INTO cards VALUES (3, 1, '{"name": "Cook lunch'
INSERT INTO cards VALUES (4, 1, '{"name": "Vacuum", "t
INSERT INTO cards VALUES (5, 1, '{"name": "Hang painti
```

Querying data

Data you can't access is fairly useless. Next we'll look at how to query what we've previously inserted into the DB.

```
SELECT data->>'name' AS name FROM cards
```

```
name
----------------
Paint house
Wash dishes
Cook lunch
Vacuum
Hang paintings
(5 rows)
```

Filtering results

It's very common to filter your query based on a column, and with a JSONB column we can actually step into the JSON document and filter our results based on the different properties it has. In the example below our "data" column has a property called "finished", and we only want results where finished is true.

```
SELECT * FROM cards WHERE data->>'finished' = 'true';
```

```
id | board_id |                                      da
----+----------+-----------------------------------------
 1 |        1 | {"name": "Paint house", "tags": ["Impr
(1 row)
```

Checking for column existence

Here we'll find a count of the records where the data

Here we'll find a count of the records where the data column contains a property named "ingredients."

```
SELECT count(*) FROM cards WHERE data ? 'ingredients';
```

```
count
-------
    1
(1 row)
```

Expanding data

If you've worked with relational databases for a while, you'll be quite familiar with aggregate methods: count, avg, sum, min, max, etc. Now that we're dealing with JSON data, a single record in our database might contain an array. So, instead of shrinking the results into an aggregate, we can now expand our results.

```
SELECT
  jsonb_array_elements_text(data->'tags') as tag
FROM cards
WHERE id = 1;
```

```
tag
--------------
Improvements
Office
(2 rows)
```

There are three things I'd like to point out about the example above:

> Two rows were returned even though we queried a single row from our database. This is equal to the number of tags that this row contained.

I used the **jsonb** form of the method instead of the **json** form. Use the one that matches how you defined the column.

I accessed the **tags** field using **->** instead of **->>** like before. **->** will return the attribute as a JSON object, whereas **->>** will return the property as integer or text (the parsed form of the attribute).

The real benefit of JSONB: Indexes

We want our application to be fast. Without indexes, the database is forced to go from record to record (a *table scan*), checking to see if a condition is true. It's no different with JSON data. In fact, it's most likely worse since Postgres has to step in to each JSON document as well.

I've increased our test data from five records to 10,000. This way we can begin to see some performance implications when dealing with JSON data in Postgres, as well as how to solve them.

```
SELECT count(*) FROM cards WHERE data->>'finished' = '
```

```
count
-------
4937
(1 row)
```

```
Aggregate (cost=335.12..335.13 rows=1 width=0) (actual
    Filter: ((data ->> 'finished'::text) = 'true'::tex
    Rows Removed by Filter: 5062
Planning time: 0.071 ms
Execution time: 4.465 ms
```

Now, that wasn't that slow of a query at 5ms, but let's see if we can improve it.

```
CREATE INDEX idxfinished ON cards ((data->>'finished')
```

If we run the same query which now has an index, we end up cutting the time in half.

```
count
-------
4937
(1 row)
```

```
Aggregate (cost=118.97..118.98 rows=1 width=0) (actual
    Recheck Cond: ((data ->> 'finished'::text) = 'true
    Heap Blocks: exact=185
    -> Bitmap Index Scan on idxfinished (cost=0.00..4.
        Index Cond: ((data ->> 'finished'::text) = 'tr
Planning time: 0.084 ms
Execution time: 2.199 ms
```

Our query is now taking advantage of the **idxfinished** index we created, and the query time has been approximately cut in half.

More complicated indexes

One of the cool things about the JSON support in Postgres is that you can query to see if an array contains a certain value.

```
SELECT count(*) FROM cards
WHERE
  data->'tags' ? 'Clean'
  AND data->'tags' ? 'Kitchen';
```

```
count
-------
1537
(1 row)
```

```
Aggregate (cost=385.00..385.01 rows=1 width=0) (actual
    Filter: (((data -> 'tags'::text) ? 'Clean'::text)
    Rows Removed by Filter: 8463
Planning time: 0.063 ms
Execution time: 6.710 ms
(6 rows)

Time: 7.314 ms
```

As of Postgres 9.4, along with the JSONB data type came GIN (Generalized Inverted Index) indexes. With GIN indexes, we can quickly query data using the JSON operators @>, ?, ?&, and ?|. For details about the operators, you can visit the Postgres documentation.

```
CREATE INDEX idxgintags ON cards USING gin ((data->'ta
```

```
count
-------
1537
(1 row)
```

```
Aggregate (cost=20.03..20.04 rows=1 width=0) (actual t
    Recheck Cond: (((data -> 'tags'::text) ? 'Clean'::
    Heap Blocks: exact=185
    -> Bitmap Index Scan on idxgintags (cost=0.00..16.
        Index Cond: (((data -> 'tags'::text) ? 'Clean'
Planning time: 0.088 ms
Execution time: 2.706 ms
(8 rows)

Time: 3.248 ms
```

Again we see the speed doubling. This would be even more pronounced if our dataset were larger than 10,000 records. The **explain analyze** output also shows us how it is using the **idxgintags**index.

Lastly, we can add a GIN index on the entire data field, which will allow us a bit more flexibility in terms of how we can query the data.

```
CREATE INDEX idxgindata ON cards USING gin (data);

SELECT count(*) FROM cards
WHERE
  data @> '{"tags": ["Clean", "Kitchen"]}';
```

```
count
-------
1537
(1 row)
```

```
Time: 2.837 ms
```

How can I do this in Rails?

Let's explore how to create tables with JSONB columns in Rails, as well as how to query those JSONB columns and update the data. For more information, you can refer to the Rails documentation on this subject.

Defining JSONB columns

First things first we need to create a migration which will create a table that has a column specified as JSONB.

```
create_table :cards do |t|
  t.integer :board_id, null: false
  t.jsonb :data
end
```

Querying JSON data from within Rails

Let's define a scope to help us find "finished" cards. It should be noted that even though the **finished** column is a JSON true value, when querying for it we will need to use the String true. If we look at the **finished** column in Rails we will see a TrueClass value, and it is also a JSON true value when viewing the data in psql, but despite that it will need to be queried using a String.

```
card.data["finished"].class
# TrueClass
```

Here is the code to add a **:finished** scope to our Card class. We won't be able to use the regular where clause

syntax that we're used to, but will have to rely on a more Postgres specific syntax. It should be noted that the finished column needs to be wrapped in a String too, which is how you refer to JSON columns in Postgres.

```
class Card < ActiveRecord::Base

  scope :finished, -> {
    where("cards.data->>'finished' = :true", true: "tr
  }

end
```

```
irb(main):001:0> Card.finished.count
   (2.8ms)  SELECT COUNT(*) FROM "cards" WHERE (cards.
=> 4937
```

Manipulating JSON data in Rails

Any column defined as JSON or JSONB will be represented as a Hash in Ruby.

```
irb(main):001:0> card = Card.first
  Card Load (0.9ms)  SELECT  "cards".* FROM "cards"  C
=> #<Card id: 1, board_id: 1, data: {"name"=>"Organize
irb(main):002:0> card.data
=> {"name"=>"Organize Kitchen", "tags"=>["Fast", "Orga
irb(main):003:0> card.data.class
=> Hash
```

Updating JSON data in Rails

Updating our JSON data is quite easy. It's simply a matter of changing the Hash value and then calling save on our model. To update the "finished" field to true, we would run this:

```
irb(main):004:0> card.data["finished"] = true
=> true
irb(main):005:0> card.save
   (0.2ms)  BEGIN
  SQL (0.9ms)  UPDATE "cards" SET "data" = $1 WHERE "c
   (6.6ms)  COMMIT
=> true
```

You'll notice that both Rails and Postgres can't update just the single "finished" value in the JSON data. It actually replaces the whole old value with the whole new value.

Conclusion

We've seen that Postgres now contains some very powerful JSON constructs. Mixing the power of relational databases (a simple inner join is a beautiful thing, is it not?) with the flexibility of the JSONB data type offers many benefits without the complexity of having two separate databases.

You are also able to avoid making compromises that are sometimes present in document databases (if you ever have to update a reference to the user's name in five different places, you'll know what I'm talking about.) Give it a try! Who says you can't teach an old dog some new tricks?