Replacing EAV with JSONB in PostgreSQL

coussej.github.io/2016/01/14/Replacing-EAV-with-JSONB-in-PostgreSQL/

> **TL;DR**: JSONB has potential for greatly simplifying schema design without sacrificing query performance.

Introduction

It must be one of the oldest use cases in the world of relational databases: you have an entity, and you need to store certain properties of this entity. But, not all entities have the same set of properties, and properties will be added in the future.

The most naive way to solve this problem would be to create a column in your table for each property, and just fill in the ones that are relevant. Great! Problem solved. Until your table contains millions of records and you need to add a new non-null property.

Enter Entity-Attribute-Value. I've seen this pattern in almost every database I've worked with. One table contains the entities, another table contains the names of the properties (attributes) and a third table links the entities with their attributes and holds the value. This gives you the flexibility for having different sets of properties (attributes) for different entities, and also for adding properties on the fly *without locking your table for 3 days*.

Nonetheless, I wouldn't be writing this post if there were no downsides to this approach. Selecting one or more entities based on 1 attribute value requires 2 joins: one with the attribute table and one with the value table. Need entities bases on 2 attributes? That's 4 joins! Also, the properties usually are all stored as strings, which results in type casting, both for the result as for the WHERE clause. If you write a lot of ad-hoc queries, this is very tedious.

Despite these obvious shortcomings, EAV has been used for a long time to solve this kind of problem. It was a necessary evil, and there just was no better alternative. But then PostgreSQL came along with a new feature…

Starting from PostgreSQL 9.4, a JSONB datatype was added for storing binary JSON data. While storing JSON in this format usualy takes slightly more space and time then plain text JSON, executing operations on it is much faster. Also JSONB supports indexing, making querying it even faster.

This new data type enables us to replace the tedious EAV pattern by adding a single JSONB column to our entity table, greatly simplifying the database design. But many argue that this must come with a performance cost. That's why I created this benchmark.

Test database setup

For this comparison, I created a database on a fresh PostgreSQL 9.5 installation on an 80 $ DigitalOcean Ubuntu 14.04 box. After tuning some settings in *postgresql.conf*, I ran this script using psql.

The following tables were created for representing the data as EAV.

```
CREATE TABLE entity (
  id           SERIAL PRIMARY KEY,
  name         TEXT,
  description  TEXT
);
CREATE TABLE entity_attribute (
  id          SERIAL PRIMARY KEY,
  name        TEXT
);
CREATE TABLE entity_attribute_value (
  id                  SERIAL PRIMARY KEY,
  entity_id           INT    REFERENCES entity(id),
  entity_attribute_id INT    REFERENCES entity_attribute(id),
  value               TEXT
);
```

The table below represents the same data, but with the attributes in a JSONB column which I called properties.

```
CREATE TABLE entity_jsonb (
  id          SERIAL PRIMARY KEY,
  name        TEXT,
  description TEXT,
  properties  JSONB
);
```

A lot simpler, isn't it?

Then, I loaded the exact same data for both patterns for a total of 10 million entities in the form of the one below. This way, we have some different data types among the attribute set.

```
{
  id:          1
  name:        "Entity1"
  description: "Test entity no. 1"
  properties:  {
    color:        "red"
    lenght:       120
    width:        3.1882420
    hassomething: true
    country:      "Belgium"
  }
}
```

So, we now have the same data stored in both formats. Let's start comparing!

Query aesthetics

Earlier it was already clear that the design of the database was greatly simplified by using a JSONB column for the properties instead of using a 3 tabes EAV model. But does this also reflect in the queries?

Updating a single entity property looks like this:

```
-- EAV
UPDATE entity_attribute_value
SET value = 'blue'
WHERE entity_attribute_id = 1
  AND entity_id = 120;

-- JSONB
UPDATE entity_jsonb
SET properties = jsonb_set(properties, '{"color"}', '"blue"')
WHERE id = 120;
```

Admittedly, the latter doesn't look simpler. To update the property in the JSONB object, we have to use the *jsonb_set()* function, and we have to pass our new value as a JSONB object. However, we don't need to know any id's upfront. When you look at the EAV example, you have to know both the entity_id and the entity_attribute_id to perform the update. If you want to update a property in the JSONB column based on the entity name, go ahead, it's all in the same row.

Now, let's select that entity we just updated, based on its new color:

```
-- EAV
SELECT e.name
FROM entity e
  INNER JOIN entity_attribute_value eav ON e.id = eav.entity_id
  INNER JOIN entity_attribute ea ON eav.entity_attribute_id = ea.id
WHERE ea.name = 'color' AND eav.value = 'blue';

-- JSONB
SELECT name
FROM entity_jsonb
WHERE properties ->> 'color' = 'blue';
```

I think we can agree that the second is both shorter (no joins!) and more pleasing to the eye. A clear win for JSONB! Here, we use the JSON ->> operator to get the color as a text value from the JSONB object. There is also a second way to achieve the same result in the JSONB model, using the @> containment operator:

```
-- JSONB
SELECT name
FROM entity_jsonb
WHERE properties @> '{"color": "blue"}';
```

This is a bit more complex: we check if the JSON object in the properties column contains the object on the right of the operator. Less readable, more performant (see later).

The simplification of using JSONB is even stronger when you need to select multiple properties at once. This is where the JSONB approach really shines: we just select the properties as extra columns in our result set, without the need for joins.

```
-- JSONB
SELECT name
  , properties ->> 'color'
  , properties ->> 'country'
FROM entity_jsonb
WHERE id = 120;
```

With EAV, you would need 2 joins per property you want to query.

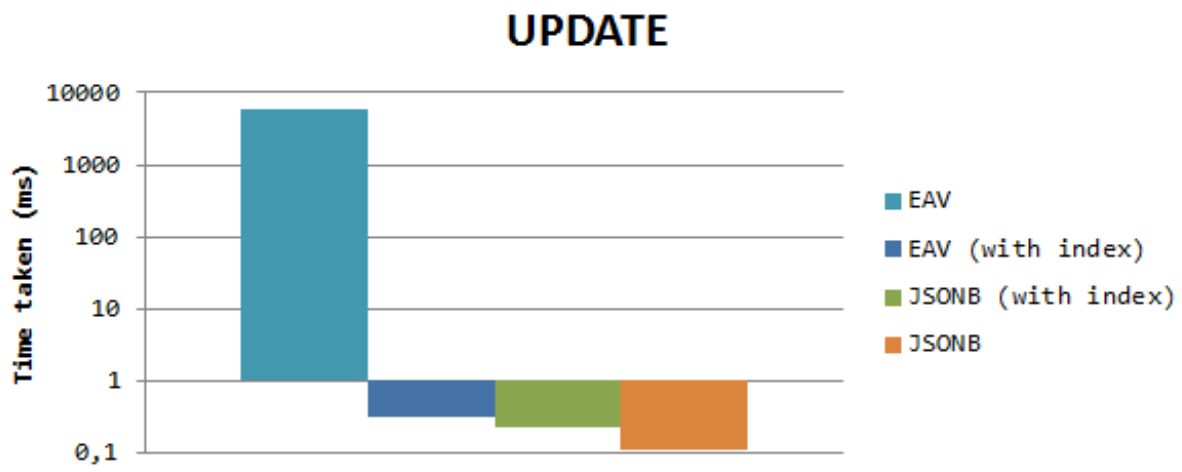
In my opinion, the queries above show a great simplification in database design. If you want more examples on how to query JSONB data, check out this post.

Now, let's talk performance.

Performance

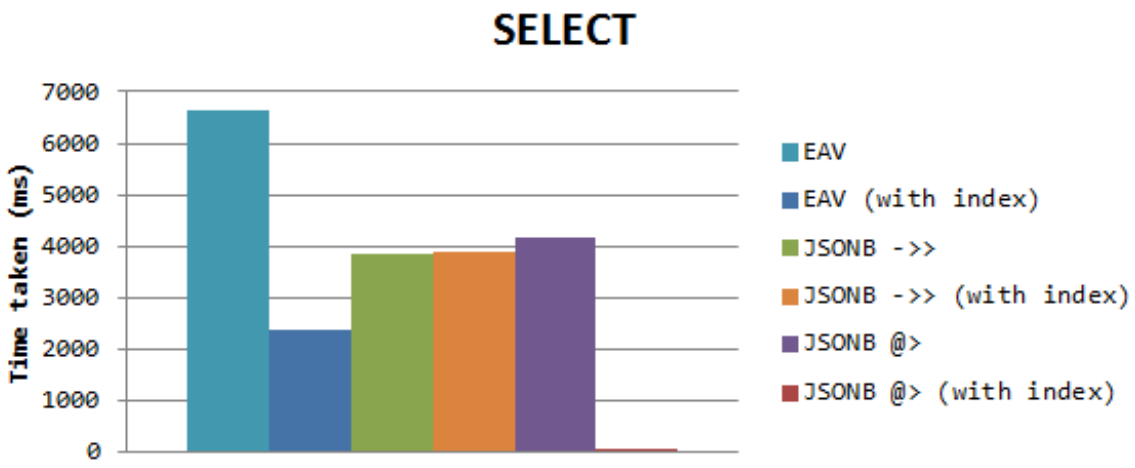
To compare performance, I used EXPLAIN ANALYSE on the queries above to see how long they take. I did each query at least three times, because the first time the query planner needs some more time. At first, I executed the queries without any indexes. This will obviously be in the advantage of JSONB, as the joins required for EAV can't make use of index scans (the foreign key fields aren't indexed). After that, I created an index on the 2 foreign key columns in the EAV value table, and also a GIN index on the JSONB column

For updating the data, this resulted in these execution times (in ms). Note that the scale is logarithmic:



Here, we see that the JSONB is much (>50000x) faster than EAV when not using any indexes, for the reason mentioned above. When we index the foreing key columns the difference is almost eliminated, but JSONB is still 1.3x faster than EAV. Notice that the index on the JSONB column does not have any effect here, as we don't use the properties column in the criteria.

For selecting data based on a property value, we get the following results (normal scale):



Here we can see that JSONB was again faster without indexes for EAV, but when the index is used EAV is the fastest. But then I noticed the times for the JSONB queries were the same, pointing me to the fact that the GIN index is not used. Apparently, when you use a GIN index on the full properties column, it only has effect when using the containment (@>) operator. I added this to the benchmark and it had a huge effect on the timing: only 0.153ms! That's 15000x faster then EAV, and 25000x faster than the **->>** operator.

For me, that's fast enough.

Table size

Let's compare the sizes of both approaches. In psql we can show the size of all tables and indexes using the **\dti+** command:

```
test-# \dti+
                                                 List of relations
                      Name                       | Type  |         Table          |
Size   |
-------------------------------------------------+-------+------------------------+----
-----+
 entity                                          | table |                        | 730
MB  |
 entity_attribute                                | table |                        | 48
kB   |
 entity_attribute_name_idx                       | index | entity_attribute       | 16
kB   |
 entity_attribute_name_key                       | index | entity_attribute       | 16
kB   |
 entity_attribute_pkey                           | index | entity_attribute       | 16
kB   |
 entity_attribute_value                          | table |                        |
2338 MB |
 entity_attribute_value_entity_attribute_id_idx | index | entity_attribute_value |
1071 MB |
 entity_attribute_value_entity_id_idx            | index | entity_attribute_value |
1071 MB |
 entity_attribute_value_pkey                     | index | entity_attribute_value |
1071 MB |
 entity_jsonb                                    | table |                        |
1817 MB |
 entity_jsonb_pkey                               | index | entity_jsonb           |
214 MB  |
 entity_jsonb_properties_idx                     | index | entity_jsonb           |
104 MB  |
 entity_pkey                                     | index | entity                 |
214 MB  |
(13 rows)
```

For the EAV model, the tables add up to 3068MB and the indexes add up to 3427MB, resulting in a total 6.43GB. On the other hand, the JSONB model uses 1817MB for the table, and 318MB for the indexes, totalling 2,08GB. Thats 3x less. This suprised me a bit, because we store the property names in each JSONB object. But when you think about it, in EAV we store 2 integer foreign keys per attribute value, resulting in 8 bytes of extra data. Also, in EAV all property values are stored as text, while JSONB will use numeric and boolean values internally where possible, resulting in less space.

Conclusion

In general, I think storing entity properties in JSONB format can greatly simplify your database design and maintenance. If, like me, you do a lot of ad-hoc querying, having everything stored in the same table as the entity is really useful. The fact that it simplifies interacting with your data is already a plus, but the resulting database is also 3x smaller and from my tests it seems that the performance penalties are very limited. In some cases JSONB even performes faster then EAV, which makes it even better.

However, this benchmark does of course not cover all aspects (like entities with a very large number of properties, a large increase in the number of properties of existing data, …), so if you have any suggestions on how to improve it, please feel free to leave a comment!