
JSON Home Improvement
Christophe Pettus
PostgreSQL Experts, Inc.
SCALE 14x, January 2016

Greetings!

- Christophe Pettus
- CEO, PostgreSQL Experts, Inc.
- thebuild.com — personal blog.
- pgexperts.com — company website.
- Twitter @Xof
- christophe.pettus@pgexperts.com

PGX POSTGRESQL EXPERTS, INC.

JSON, what is?

- JavaScript Object Notation.
- A text format for serializing nested data structures.
- Based on JavaScript's declaration syntax.
- Intended to be passed directly into JavaScript's eval() function (don't do this!)

JSON Primitive Types.

- Strings, always Unicode.
  - De facto, always UTF-8 in flight.
- Numbers, integer and float.
- Boolean: `true` and `false`.
- `null`

JSON Structured Types.

- Arrays, using [ ].
- Hash / dictionaries / whatever you want to call them (the JSON spec calls them Objects), using { }
  - { ‘string’ : value }
  - Keys have to be strings; values can be anything.

PGX POSTGRESQL EXPERTS, INC.

More complex types.

- Everything else is built out of those.
- There’s no type declaration mechanism.
    - “Object” is unfortunate terminology.
- There’s no “schema” or similar validation method.
- Everything is delegated to the application.

The good…

- It's super-simple to generate and parse.
  - The operational part of the spec is five pages, with pretty pictures.
- It's the de facto standard for data interchange in web APIs.
  - POST format is still used, but apps that do that are wrong.

The bad…

- No higher-level standards.
  - How is a datetime represented? I dunno, you figure it out.
- Remember SQL injection attacks? Now we have JSON injection attacks.
  - Don't use eval(). Just. Don't.

And PostgreSQL has JSON!

- It’s a core type.
  - Not a contrib/ or extension module.
- Introduced in 9.2.
- Enhanced in 9.3.
- And really enhanced in 9.4.

We liked JSON so much…

- … we created two types.
  - json
  - jsonb
- json is a pure text representation.
- jsonb is a parsed binary representation.
- Each can be casted to the other, of course.

PGX POSTGRESQL EXPERTS, INC.

json type.

- Stores the actual json text.
- Whitespace included.
- What you get out is what you put in.
- Checked for correctness, but not otherwise processed.

Why use json?

- You are storing the json and never processing it.
- You need to support two JSON "features":
  - Order-preserved fields in objects.
  - Duplicate keys in objects.
- For some reason, you need the *exact* JSON text back out.

PGX POSTGRESQL EXPERTS, INC.

Oh, and…

- jsonb wasn’t introduced until 9.4.
- So, if you are on 9.2-9.3, json is what you’ve got.
- Otherwise, you want to use jsonb.

jsonb

- Parsed and encoded on the way in.
- Stored in a compact, parsed format.
- Considerably more operator and function support.
- Has indexing support.

They're just types.

- Fully transactional, can have multiple `json`/`jsonb` fields in a single table, etc.
- Uses the TOAST mechanism.
  - Can be up to 1GB.
- Can be a NULLable field if you like.

Basic Operators (both json and jsonb)

- -> gets a JSON array element or object field, as JSON.
- ->> gets the array element or object field cast to TEXT.
- #> gets the array element or object field at a path.
- #>> … cast to TEXT.

jsonb only!

- @> — Does the left-hand value contain the right-hand value?
- <@ — Does the right-hand value contain the left hand value?

Containment

- Containment work at the top level of the json object only, and on full JSON structures.
- It does not apply to individual keys.
- It does not apply to nested elements.

@>

```
postgres=# select '{"a": 1, "b": 2}'::jsonb @> '{"a": 1}'::jsonb;
 ?column?
----------
 t
(1 row)


postgres=# select '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;
 ?column?
----------
 t
(1 row)

postgres=# select '{"a": {"b": 7, "c": 8}}'::jsonb @>
                  '{"a": {"c": 8}}'::jsonb;
 ?column?
----------
 t
(1 row)
```

PGX POSTGRESQL EXPERTS, INC.

but.

```
postgres=# select '{"a": {"b": 7}}'::jsonb @> '{"b": 7}'::jsonb;
 ?column?
----------
 f
(1 row)

postgres=# select '{"a": 1, "b": 2}'::jsonb @> '"a"'::jsonb;
 ?column?
----------
 f
(1 row)
```

?, ?|, ?&

- True if:
  - ? — The key on the right-hand side appears in the left-hand side.
  - ?| ?& — Any of the array of keys on the right-hand side appear on the left-hand side.
  - PostgreSQL array type, not JSON array.

PGX POSTGRESQL EXPERTS, INC.

?, ?|, ?&

```
postgres=# select '{"a": 7, "b": 4}'::jsonb ? 'a';
 ?column?
----------
 t
(1 row)

postgres=# select '{"a": 7, "b": 4}'::jsonb ?& ARRAY['a', 'b'];
 ?column?
----------
 t
(1 row)

postgres=# select '{"a": 7, "b": 4}'::jsonb ?| ARRAY['a', 'q'];
 ?column?
----------
 t
(1 row)
```

but.

```
postgres=# select '{"a": {"b": 7, "c": 8}}'::jsonb ? 'b';
 ?column?
----------
 f
(1 row)

postgres=# select '[1, 2, 3, 4]'::jsonb ?| ARRAY[1, 100];
ERROR:  operator does not exist: jsonb ?| integer[]
LINE 1: select '[1, 2, 3, 4]'::jsonb ?| ARRAY[1, 100];
                                     ^
HINT:  No operator matches the given name and argument type(s). You might
need to add explicit type casts.

postgres=# select '[1, 2, 3, 4]'::jsonb ?| '[1, 2]'::jsonb;
ERROR:  operator does not exist: jsonb ?| jsonb
LINE 1: select '[1, 2, 3, 4]'::jsonb ?| '[1, 2]'::jsonb;
                                     ^
HINT:  No operator matches the given name and argument type(s). You might
need to add explicit type casts.
```

PGX POSTGRESQL EXPERTS, INC.

JSON functions

- Lots and lots and lots.
- Create JSON from records, arrays, etc.
- Expand JSON into records, arrays, rowsets, etc.
- Many have both json and jsonb versions.

Example: row_to_json

- Accepts an arbitrary row.
- Returns a json (not jsonb) object.
- For non-string/int/NULL types, uses the output function to create a string.
- Properly handles composite/array types.

Behold!

```
xof=# select row_to_json(rel.*) from rel where array_length(tags, 1) > 2  order
by id limit 3;
                                                          row_to_json
----------------------------------------------------------------------------------
---------------------------------------------
 {"id":636572,"first_name":"OLENE","last_name":"OGRAM","tags":
["female","square","violet"]}
 {"id":636744,"first_name":"SHAYNE","last_name":"GALPIN","tags":
["female","square","silver","aquamarine","green","octogon"]}
 {"id":636769,"first_name":"YASMIN","last_name":"AKEN","tags":
["female","red","green"]}
(3 rows)
```

But seriously…

- … can be used in a trigger to append to an audit table regardless of the schema.
- Extremely useful for shared triggers.

Example: jsonb_each_text

- Takes a jsonb object, and returns a rowset of key/value pairs.
- Returns each as text object.
- Can be used to write the world's most expensive EAV query!

Behold!

```
xof=# WITH s AS (
xof(# SELECT row_to_json(rel.*)::jsonb AS j FROM rel ORDER BY id LIMIT 3
xof(# ) SELECT (s.j->>'id')::bigint AS entity, key as attribute, value FROM s,
LATERAL jsonb_each_text(s.j) WHERE key <> 'id';
 entity | attribute  |   value
--------+------------+------------
 636526 | tags       | ["female"]
 636526 | last_name  | EILTS
 636526 | first_name | REGENA
 636527 | tags       | ["male"]
 636527 | last_name  | POTO
 636527 | first_name | ANTONIO
 636528 | tags       | ["female"]
 636528 | last_name  | LUFSEY
 636528 | first_name | ROXY
(9 rows)
```

PGX POSTGRESQL EXPERTS, INC.

But that would be wrong.

But seriously…

- … it can be used to expand jsonb into relational data for JOINs and the like.
- Often more efficient than using the extraction operators.

Indexing.

Indexing json

- The textual json type has no inherent indexing (that you'd ever use).
- Can do an expression index on extracted values…
- … but that requires knowing exactly which fields / elements you are going to query on.

jsonb indexing.

- jsonb has GIN indexing.
- Default type supports queries with the @>, ?, ?& and ?| operators.
- The query must be against the top-level object for the index to be useful.
- Can query nested objects, but only in paths rooted at the top level.

PGX POSTGRESQL EXPERTS, INC.

jsonb_path_ops

- Optional GIN index type for jsonb.
- Only supports @>.
- Hashes paths for each item, rather than just storing the key itself.
- Faster for @> operations with nesting.

jdoc @> '{"tags": ["qui"]}'

- Both index types support this.
- jsonb_ops (the default) will seach for everything that has “tags”, has “qui”, AND them, and then do a recheck for the path structure.
- jsonb_path_ops will go directly to entries for that path.

PGX POSTGRESQL EXPERTS, INC.

Which to use?

- If you just need @>, jsonb_path_ops will probably be faster.
- If you need the other supported operators, you need jsonb_ops.

PGX POSTGRESQL EXPERTS, INC.

New in PostgreSQL 9.5!

- jsonb_pretty() — Pretty-prints the jsonb structure.
- jsonb || jsonb — Merges two top-level objects (keys from the right-hand side win).
- jsonb - (minus) — Remove a key or array element.

PGX POSTGRESQL EXPERTS, INC.

jsonb_set

- Used to be jsonb_replace.
- Replaces items in the JSON structure based on a path.
- By default, will create missing items as required (optionally, can throw an error instead).

```
postgres=# SELECT jsonb_set('{"a":
"x"}'::jsonb, '{a}', '"y"');
 jsonb_set
------------
 {"a": "y"}
(1 row)

postgres=# SELECT jsonb_set('[{"a": [1, 2,
3]}]'::jsonb, '{0,a,-1}', '"x"'::jsonb);
      jsonb_set
----------------------
 [{"a": [1, 2, "x"]}]
(1 row)
```


So, what can we
do with this?

1: Auditing!

- The problem: Want to keep a record of every change to a set of tables.
- But every table has its own schema.
- Create one audit table per table being tracked?
- Lots of tables, error-prone, have to change schemas two places...

Use JSON!

- Can create a single audit table that handles changes for all child tables.
- Can create a single trigger function that can be attached to any table that needs auditing.

PGX POSTGRESQL EXPERTS, INC.

Audit Table

```
CREATE TABLE audit (
   ts TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
   schema_name VARCHAR NOT NULL,
   table_name VARCHAR NOT NULL,
   operation VARCHAR NOT NULL,
   row_contents JSONB
);
```

Trigger.

```
CREATE OR REPLACE FUNCTION audit() RETURNS TRIGGER AS
$audit$
DECLARE
   record_to_log    JSONB;
BEGIN
    IF TG_OP = 'DELETE' THEN
        record_to_log := row_to_json(OLD.*)::JSONB;
    ELSE
        record_to_log := row_to_json(NEW.*)::JSONB;
    END IF;

    INSERT INTO audit(schema_name, table_name, operation, row_contents)
        VALUES(TG_TABLE_SCHEMA, TG_TABLE_NAME, TG_OP, record_to_log);

    RETURN NULL;

END;
$audit$
    LANGUAGE plpgsql;
```

PGX POSTGRESQL EXPERTS, INC.

Behold!

```
xof=# INSERT INTO x(i, f) VALUES(12, 7.5);
INSERT 0 1
xof=# INSERT INTO y(q) VALUES(ARRAY[1,2,3,4]);
INSERT 0 1
xof=# TABLE audit;
              ts               | schema_name | table_name | operation | 
row_contents
-------------------------------+-------------+------------+-----------
+--------------------------------
 2016-01-20 15:06:11.408046-08 | public      | x          | INSERT    | 
{"f": 7.5, "i": "12", "pk": 4}
 2016-01-20 15:06:22.929203-08 | public      | y          | INSERT    | 
{"q": [1, 2, 3, 4], "pk": 5}
(2 rows)
```

PGX POSTGRESQL EXPERTS, INC.

And you can dedup.

```
CREATE OR REPLACE FUNCTION json_diff(l JSONB, r JSONB) RETURNS JSONB AS
$json_diff$
    SELECT jsonb_object_agg(a.key, a.value) FROM
        ( SELECT key, value FROM jsonb_each(l) ) a LEFT OUTER JOIN
        ( SELECT key, value FROM jsonb_each(r) ) b ON a.key = b.key
    WHERE a.value != b.value OR b.key IS NULL;
$json_diff$
    LANGUAGE sql;
```

Behold!

```
xof=# select json_diff( '{"a": 1, "b": 2}'::jsonb, '{"a": 1, "b":
1}'::jsonb );
 json_diff
-----------
 {"b": 2}
(1 row)

xof=# select json_diff( '{"a": 1, "b": 2}'::jsonb, '{"a": 2, "b":
1}'::jsonb );
    json_diff
------------------
 {"a": 1, "b": 2}
(1 row)

xof=# select json_diff( '{"a": 1}'::jsonb, '{"a": 1}'::jsonb ) is null;
 ?column?
----------
 t
(1 row)
```

PGX POSTGRESQL EXPERTS, INC.

The Good.

- A single trigger function and schema that contains everything.
- Schema changes to the audited tables don't require any further changes.
- The JSONB object can be indexed for faster retrieval.

The Bad.

- Bigger and slower than relational data.
- Joins can be pretty slow.
- Not great for historical tracking that is in common use in the application.
- The single table can get huge: Need a partition / archiving strategy.

2: Post-Deployment Schema Changes

- The problem: A packaged application.
- Each customer runs their own instance (an appliance, for example).
- The application allows users to customize the schema.
  - Additional fields, such as "size" for clothing.

PGX POSTGRESQL EXPERTS, INC.

EAV tables!

- Option 1: Use an Entity-Attribute-Value table.
- The table can get quite large.
- Not very pleasant to join on.

ALTER TABLE

- We could modify the schema on the fly.
- Application needs to understand the additional fields.
- Can make migrations for new versions of application complicated.
- Each installation now becomes slightly unique.

PGX POSTGRESQL EXPERTS, INC.

... or JSON!

- Use a JSON field to hold customizations.
- Can be indexed in reasonable ways.
- Retrieved as part of the record retrieval; no join required.
- Potential space savings from compression for larger objects.

3: Securing Data.

- The problem: You have sensitive data (PCI, HIPAA, passwords) in the database.
- You want to encrypt it.
- So, what do to?

Encrypt Everything!

- Full disk encryption! Problem solved!
  - Uh, no. FDE is useless.
- Encrypt in app or PostgreSQL.
  - Breaks indexing.
  - Most fields don't actually need to be encrypted at rest.

Encrypt Some, Not Others.

- Encrypt only those fields that need encryption.
- Provide hashed or similar external keys for quick lookups.
- But what if there are several separate fields that need encryption at rest?
  - You could separately encrypt them, or...

PGX
POSTGRESQL EXPERTS, INC.

Use JSON!

- Stuff all the sensitive info into a JSON object.
- Encrypt that.
- Use JSON primitives in PostgreSQL if you are encrypting at the database level, or...
- Just return the blob to the app and decrypt it there (yes, this is cheating).

4: Structured Object Storage.

- Sometimes, you just want to store an object.
- Highly variable “schema”.
- Lots and lots and lots of optional fields.
- Hierarchical data that would be painful to decompose.

Use JSON!

- That's what it's there for.
- Faster than XML.
- More powerful than hstore.
- And highly searchable and indexable.
  - In fact, we beat MongoDB in most real-life applications.

5: API Logging.

- Most? Many? Almost all new? APIs are JSON-based.
- It's usually very valuable to log each raw API request for debugging and forensic purposes.
- So…

Use JSON!

- You might want to use JSON and not JSONB in this case.
  - Smaller, faster to insert.
  - But not indexable.
- Consider a separate PostgreSQL instance to avoid bogging down a transactional system.

PGX POSTGRESQL EXPERTS, INC.

In conclusion…

- The JSON functionality of PostgreSQL is an excellent complement to the relational features.
- Take full advantage of it! It's a stable, highly performant part of PostgreSQL.
- Use relational data for most things, but a little bit of JSON can really help.

Thank you!

Questions?

- thebuild.com — personal blog.
- pgexperts.com — company website.
- Twitter @Xof
- christophe.pettus@pgexperts.com

PGX POSTGRESQL EXPERTS, INC.