Schemaless meets Relational: JSONB

ConFoo 2015
Montreal, Canada

Magnus Hagander
magnus@hagander.net

Magnus Hagander

- PostgreSQL
  - Core Team member
  - Committer
  - PostgreSQL Europe
- Redpill Linpro
  - Infrastructure services
  - Principal database consultant

PostgreSQL

- Relational database (RDBMS)

Relational Database

A relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as invented by E. F. Codd

Relational Database

- Tables
- Columns
- Rows
- (etc etc)

Relational Database

- Strict schema
- Well defined data model

Relational Database

- Migration between defined schemas
- ALTER TABLE etc etc

Schema migrations

- Often considered a problem
  - (one main reason for NoSQL?)
- Not as bad as you think
  - All RDMBSes don't suck at this
  - ... but some definitely do

DDL in PostgreSQL

- Fully transactional
  - ROLLBACK any operations!
- Often no exclusive lock
- Often no rewrite

Rewrite-less DDL

- ALTER TABLE ADD COLUMN
  - with no default
- ALTER TABLE ALTER COLUMN SET TYPE
  - depending on type of course!
- Work done on further reductions!

Still not good enough?

- There's always EAV
  - Entity-Attribute-Value

```
CREATE TABLE eav (
  object int PRIMARY KEY,
  attr text,
  value text
);
```

EAV

- Yeah, that's evil
- Just don't go there!
  - One of the few "anti-patterns" in SQL

So what about NoSQL

- There's a lot to NoSQL
  - ISAM?
  - dBASE?
  - FoxPro?
- Let's talk document store

NoSQL document store

- Schemaless
  - Usually JSON
- "CAP" vs "ACID"
- Sharding
- etc

NoSQL document store

- Schemaless
- "CAP" vs "ACID"
- Sharding
- etc

Schemaless

- Data still has schema
- It's not strict
- There are no built-in schema migrations

Schemaless

- Schema migration happens in application
- Or all versions of schema are live

Reality-check

- Parts of the schema is known
  - And fixed
- Parts of the schema is dynamic

JSONB

- Enter JSONB

Schemaless recap

- Schemaless is not new in PostgreSQL
- Not even JSON...

Schemaless evolution

- 8.2: hstore
- 8.3: hstore+gin
- 9.0: hstore fully functional kvs
- 9.2: json
- 9.4: jsonb

hstore

- Basic key/value store
- No nesting support
- No datatype support
- Generic index support

JSON

- Simple text storage
  - Just validates JSON format
- Accessor functions
- Requires indexes on specific keys

JSONB

- Deconstructed JSON format
  - Not BSON!
  - API format is still pure JSON
- Data type aware
  - string, int, float, bool, null
  - arrays, hashes
- Nested structures

JSONB

- Regular PostgreSQL datatype
- Full ACID
- Up to 1GB
- Transparent compression etc

Simple usage

```
CREATE TABLE jsontable (
 id SERIAL PRIMARY KEY,
 j JSONB NOT NULL
);
```

Loading data

```
INSERT INTO jsontable VALUES (1,'{
  "name": "Magnus",
  "country": "Sweden",
  "skills": {
    "database": ["postgresql", "sql"],
    "web": ["varnish", "django"]
  }
}');
```

Retrieving data

```
postgres=# SELECT j FROM jsontable WHERE id=1;
                                                          j
---------------------------------------------------------------------------------------------------------------
 {"name": "Magnus", "skills": {"web": ["varnish", "django"], "database": ["postgresql", "sql"]}, "country": "Sweden"}
(1 row)
```

- Key-order not preserved
- Formatting not preserved

Retrieving data

- Element retrieval operator

```
postgres=# SELECT j->>'country' FROM jsontable WHERE id=1;
 ?column?
----------
 Sweden
(1 row)
```

- Or use -> to get json object back

```
postgres=# select j->'skills'->'database' FROM jsontable WHERE id=1;
       ?column?
-----------------------
 ["postgresql", "sql"]
(1 row)
```

Retrieving data

- Once retrieved, just a SQL value
- All regular SQL queries/analytics work

```
postgres=# SELECT j->>'country' AS country,count(*)
postgres-# FROM jsontable GROUP BY country;
 country | count
---------+-------
 Sweden  |     1
....
```

Retrieving data

- Path operators

```
postgres=# SELECT j#>'{skills,database}' FROM jsontable WHERE id=1;
        ?column?
-------------------------
 ["postgresql", "sql"]
(1 row)
```

- or array indexing

```
postgres=# SELECT j#>'{skills,database,0}' FROM jsontable WHERE id=1;
   ?column?
--------------
 "postgresql"
(1 row)
```

Searching data

- Searching by extracted fields

```
postgres=# SELECT id FROM jsontable WHERE j->>'country' = 'Sweden';
 id
----
  1
(1 row)
```

Searching data

- Searching by extracted fields

```
postgres=# SELECT id FROM jsontable WHERE j->>'country' = 'Sweden';
 id
----
  1
(1 row)
```

- Requires index on each field
  - Not very schemaless

Searching data

- Path operators

```
postgres=# SELECT id FROM jsontable WHERE j @> '{"country": "Sweden"}';
 id
----
  1
(1 row)
```

- Nested structure support

```
postgres=# SELECT id FROM jsontable WHERE j @> '{"skills": {"database": ["postgresql"]}}';
 id
----
  1
(1 row)
```

Indexing

- Where JSONB really shines
- Single generic index!

```
CREATE INDEX json_idx
 ON jsontable
 USING gin(j);
```

Indexed containment search

```
postgres=# EXPLAIN SELECT id FROM jsontable
postgres-#  WHERE j @> '{"skills": {"database": ["postgresql"]}}';
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on jsontable  (cost=12.00..16.01 rows=1 width=4)
   Recheck Cond: (j @> '{"skills": {"database": ["postgresql"]}}'::jsonb)
   ->  Bitmap Index Scan on json_idx  (cost=0.00..12.00 rows=1 width=0)
         Index Cond: (j @> '{"skills": {"database": ["postgresql"]}}'::jsonb)
(4 rows)
```

Indexed containment search

- Full path search in one operator
- Deep filtering in the index

Other indexed operators

- Key existence

```
postgres=# EXPLAIN SELECT id FROM jsontable
postgres-#  WHERE j ? 'country';
                                QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on jsontable  (cost=8.00..12.01 rows=1 width=4)
   Recheck Cond: (j ? 'country'::text)
   ->  Bitmap Index Scan on json_idx  (cost=0.00..8.00 rows=1 width=0)
         Index Cond: (j ? 'country'::text)
(4 rows)
```

Other indexed operators

- Key existence
  - On sub-objects
  - No longer indexed
  - (can use expression index)

```
postgres=# EXPLAIN SELECT id FROM jsontable
postgres-#  WHERE j->'skills' ? 'database';
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on jsontable  (cost=0.00..1.03 rows=1 width=4)
   Filter: ((j -> 'skills'::text) ? 'database'::text)
(2 rows)
```

Other indexed operators

- All keys exist

```
postgres=# EXPLAIN SELECT id FROM jsontable
postgres-# WHERE j ?& ARRAY['name', 'firstname'];
                                QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on jsontable  (cost=12.00..16.01 rows=1 width=4)
   Recheck Cond: (j ?& '{name,firstname}'::text[])
   ->  Bitmap Index Scan on json_idx  (cost=0.00..12.00 rows=1 width=0)
         Index Cond: (j ?& '{name,firstname}'::text[])
(4 rows)
```

Other indexed operators

- Any key exists

```
postgres=# EXPLAIN SELECT id FROM jsontable
postgres-# WHERE j ?| ARRAY['name', 'firstname'];
                                QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on jsontable  (cost=12.00..16.01 rows=1 width=4)
   Recheck Cond: (j ?| '{name,firstname}'::text[])
   ->  Bitmap Index Scan on json_idx  (cost=0.00..12.00 rows=1 width=0)
         Index Cond: (j ?| '{name,firstname}'::text[])
(4 rows)
```

Index operator classes

- Default operator class jsonb_ops
  - Handles all operators shown
- Path only operator class jsonb_path_ops
  - Handles only @>
  - Smaller index footprint!
  - Hashed paths

Index operator classes

- Consider using jsonb_path_ops instead
  - Don't use both

```
CREATE INDEX json_idx
 ON jsontable
 USING gin(j jsonb_path_ops);
```

JSON conversion

- Many functions to deal with conversion
- To/from tables
- To/from resultsets
- To/from arrays

JSON conversion

```
postgres=# SELECT row_to_json(b.*, 't')
 FROM pg_stat_bgwriter b;
                        row_to_json
-----------------------------------------------------
 {"checkpoints_timed":1302,                         +
  "checkpoints_req":1,                              +
  "checkpoint_write_time":17259,                    +
  "checkpoint_sync_time":44,                        +
  "buffers_checkpoint":185,                         +
....
```

How to use this?

How to use this?

- tl; dr: don't do what I just showed

How to use this

- Identify fixed schema parts
  - E.g. name and country
  - Actually, maybe also skills..
- Use relational for this data
- It will be faster
- Especially when updating!

How to use this

- Add additional jsonb column(s)
  - When needed
  - Some tables
- For purely unstructured data
- Or data that changes structure over time

Drawbacks of jsonb

- Lack of key compression
- Document level locking
- Document level updating

Advantages of jsonb

- Dynamic schema
- Dynamic index
  - Only need one!

Client support

- Text input/output
  - Supported by all drivers
  - Client level compose/decompose
- Driver integration
  - Driver maps directly to native object
  - e.g. psycopg2 >= 2.5.4

Other technologies

JSQuery

- JSONB specific query language
- Like tsquery for fts
- Available as plugin for 9.4

pg_shard

- Automatic sharding for PostgreSQL
- Shards and replicates on simple expressions
  - E.g. update/delete requires sharding key
- Non-transactional across shards

ToroDB

- MongoDB frontend
  - (Speaks MongoDB protocol)
- PostgreSQL backend
  - Fully relational, not jsonb

VODKA

- Even more indexing
- Combined access methods
  - (in one index)

Summary

- PostgreSQL is a platform for data management
  - Both structured and unstructured
- Reality is usually complex
  - Requires both!

Thank you!

Magnus Hagander
magnus@hagander.net
@magnushagander
http://www.hagander.net/talks/