17 March 2021

Power JSON with PostgreSQL

Simon Riggs
Postgres Fellow

EDB™

Part 2: SQL/JSON and Schema Design

Example of Valid JSON

```
{
   "oid": 521,
   "ots": "2021-01-16T15:52:14.70032+00:00",
   "price": 5.62,
   "descr": "Example of some text data",
   "boolfield": true,
   "tags": ["blue", "green", "red"],
   "addr": {
        "city":  "New York",
        "state": "NY"
   },
   "other": null
}
```

Advanced Search with SQL/JSON (jsonpath)

Simple SQL/JSON searches

```
-- Does the field "new" exist within the JSON
SELECT jdoc FROM j WHERE  jdoc @@
     'exists($.new)'; -- not indexable

-- Does the field "new" have a value of "true"
SELECT jdoc FROM j WHERE jdoc @@
     '$.new == true'; -- indexable

-- Does array field "tags" contain "a"
SELECT jdoc FROM j WHERE jdoc @@
     '$.tags[*] == "a"';   -- indexable

-- Find rows where price between X and Y
SELECT jdoc FROM j WHERE jdoc  @?
     '$.price ? (@ > 11.08) ? (@ < 11.12)';
                                   -- not indexable
```

- Comprehensive JSON Path language for searching within JSON documents

- $ is top-level
- .*key* is top level fieldname
- [*] means all in array

- ? adds a filter onto expression
- ==, <, <=, >=, >, != **etc..**
- == is indexable

More Advanced SQL/JSON searches

```
-- Find rows where price between X and Y
SELECT jdoc FROM j WHERE jdoc  @?
    '$.price ? (@ > 11.08) ? (@ < 11.12)';
-- Find rows where price between X and Y
SELECT jdoc FROM j WHERE jdoc  @?
    '$.price ? (@ > 11.08 && @ < 11.12)';
-- Find rows where price between X and Y
SELECT jdoc FROM j WHERE jdoc  @?
    '$ ? (@.price > 11.08 && @.price < 11.12)';

-- Find rows where ots is in Jan 2021
SELECT jdoc FROM j WHERE jdoc @?
    '$ ? (@.ots starts with "2021-01")';
-- Find rows where ots is in Jan 2021
SELECT jdoc FROM j WHERE jdoc @?
    '$.ots ? (@ starts with "2021-01")';
```

- Can add multiple ? filters
- Can use logical connectives
- Many ways of writing same query
- Illustrates use of @ to denote current location

SQL/JSON searching in trees

```
{
  "myapp": {
    "cust": {
      "addr": {
          "country": "UK"
       },
      "tags": ["a", "b", "c"]
    }
  }
}
-- Find UK customers who have a tag of "b"
SELECT jdoc FROM j WHERE jdoc @?
     '$.myapp.cust
          ? (@.addr.country == "UK")
          ? (@.tags[*] == "b")';
```

- Make sure to use **@?**
- Traverse to a common starting point in tree, then
- filter by different arms of the JSON tree, by descending from the current location "@"
- Missing fields/structure do not throw ERRORs
- With equality searches this query is indexable!

Schema design with JSON

Adding an automatic “_id” field

```
CREATE SEQUENCE j_id_seq;
CREATE OR REPLACE FUNCTION _id_auto ()
RETURNS trigger LANGUAGE plpgsql AS $$
 BEGIN
  NEW.jdoc := jsonb(format('{"_id":"%s"}',
             to_char(nextval('j_id_seq'),
                 'FM0000000000000000')))
             || NEW.jdoc;
    RETURN NEW;
 END;
$$;
CREATE TRIGGER j_id_auto
BEFORE INSERT OR UPDATE ON j
FOR EACH ROW EXECUTE FUNCTION _id_auto();
CREATE UNIQUE INDEX ON j ((jdoc->>'_id'));
```

- Create a SEQUENCE
- Format the result of the nextval() function to add an extra field to the JSONB jdoc column

- Automatically added to each new row with a BEFORE trigger

- Add a unique index

CHECK() constraint on JSONB

```
CREATE TABLE j
(jdoc JSONB
 CHECK (jdoc @? '$.myapp.cust
                 ? (exists(@.addr.country))
                 ? (exists(@.tags))'
        )
);

INSERT INTO j VALUES ('{"myapp": {"cust":
{"addr": {"country": "UK"}, "tags": ["a", "b",
1]}}}');
```

- CHECK constraints can be used to implement checks on incoming data to validation JSON schema

- Allows both strictness and flexibility in JSON schema

- Example uses a complex JSONpath query

TOAST and JSON data

```
-- Default settings are appropriate for JSONB




-- Take no action until this size: default 2kB
ALTER TABLE j
    SET (toast_tuple_target = 4096);

ALTER TABLE j
    ALTER COLUMN jdoc SET STORAGE MAIN;
```

- JSONB is a TOASTable datatype, meaning long values for that column may be moved into a side "TOAST" table
- Shorter values will still be held in main table
- For medium length JSON, **may** want to play with toast_tuple_target to get rows to stay in main table

Update Effects

- UPDATE inserts new row versions for each change
  - Does **not** affect TOASTed data unless it is explicitly updated
  - Any change to any part of JSONB data will cause non-HOT updates and, potentially, table bloat

- **Suggest** moving frequently updated fields out of JSONB as columns
  - When those columns change, JSONB data will not be rewritten
  - HOT updates, if the columns are not indexed

Compression

```
-- Default settings are appropriate for JSONB

-- SET STORAGE EXTERNAL
-- external but not compressed
-- is not currently appropriate for JSONB
```

- JSONB may also be compressed when it is moved into a TOASTable datatype

- TOAST Compression only effective with repeated values, so field names are **never** compressed

- Consider various mechanisms for compression

Fieldname Compression

- Example: {"verylongfieldname": "value"}
  - "verylongfieldname" occupies 18 bytes in a JSONB column - not typically compressed by TOAST
- Summary of Overheads
  - Overhead per row is sum(lengths of all fields) i.e. **lots!**
  - Overhead 2 bytes/row in a ZSON column - much better!
    - In practice, % of fieldnames is about 10-50% of JSON, so a typical saving might be a 15-20% space saving, or more if some values are repeated
  - Overhead of 0 bytes/row if we use a separate column for each field
    - i.e. 1**00% space saving on fieldname overhead**
- This is why we encourage the use of separating data into columns

Frequency Analysis of JSONB fields

```
SELECT  jsonb_object_keys(jdoc) as key
       ,count(*)
FROM j
GROUP BY key
ORDER BY count(*) DESC;
  key   | count
-------+--------
 price | 100000
 ots   | 100000
 oid   |  99000
 new   |     25

SELECT count(*) FROM j;
  count
--------
 100000
```

- Analyze frequency distribution of JSON fields to identify fields present in many or all rows so we can move them into columns

JSON Use Cases

How to use JSON

- External JSON
  - Direct storage - store JSON in same format it is sent
  - "Data Mapper" - JSON externally, columns in database,

    Columns externally, JSON in database
- Other Use Cases
  - Tagging - avoid heavily normalized schemas (4th, 5th Normal Form)
  - Denormalized data - single system performance
  - "Single View" - Multi-database cache - an Enterprise Pattern
  - Migration away from JSON-only databases (e.g. Mongo)

EDB™

Data Mapper

- Map from JSON to a View "Output"
  - Start with table with JSON data
  - Create View that shows that data relationally

- Map from a table to JSON "Input"
  - Start with a table with normal columns
  - Create View that shows data as JSON

Data Mapper - Output

```
CREATE TABLE jout_type (
oid      integer,
ots      timestamp,
tags     text[],
descr    text,
other    text,
price    numeric(5,2),
boolfield bool);

CREATE VIEW joutput AS
SELECT map.*
FROM j, LATERAL jsonb_populate_record(
                 NULL::jout_type,
                 jdoc) AS map;
```

- Create a table to use as a TYPE for mapping
- Create View that maps all of the fields in jout_type that match fieldnames in jdoc
- Only works for matching fieldnames

Data Mapper - Output

```
postgres=# select * from joutput;
-[ RECORD 1 ]----------------------------------
oid       | 521
ots       | 2021-01-16 15:52:14.70032
tags      | {blue,green,red}
descr     | Example of some text data
other     | SQLNULL
price     | 5.62
boolfield | t
```

jsonb_populate_record() conversion rules

- To convert a JSON value to the SQL type of an output column, the following rules are applied in sequence:
    - A JSON null value is converted to a SQL null in all cases.
    - If the output column is of type json or jsonb, the JSON value is just reproduced exactly.
    - If the output column is a composite (row) type, and the JSON value is a JSON object, the fields of the object are converted to columns of the output row type by recursive application of these rules.
    - Likewise, if the output column is an array type and the JSON value is a JSON array, the elements of the JSON array are converted to elements of the output array by recursive application of these rules.
    - Otherwise, if the JSON value is a string, the contents of the string are fed to the input conversion function for the column's data type.
    - Otherwise, the ordinary text representation of the JSON value is fed to the input conversion function for the column's data type.
- If the first parameter is NOT NULL then it will be used to provide default values if the above yields NULL

Data Mapper - Output - Matching all fields

```
CREATE VIEW joutput AS
SELECT map.*
         ,jdoc->addr AS addr
FROM j, LATERAL jsonb_populate_record(
                 NULL::jout_type,
                 jdoc) AS map;
```

```
postgres=# select * from joutput;
-[ RECORD 1 ]--------------------------------
oid       | 521
ots       | 2021-01-16 15:52:14.70032
tags      | {blue,green,red}
descr     | Example of some text data
other     | SQLNULL
price     | 5.62
boolfield | t
addr      | {"city": "New York", "state": "NY"}
```

- Pick up unmatched fields by bringing them out directly from the JSON column

Data Mapper

- Allows you to send and receive JSON data into your applications
- Allows you to store any or all JSON fields as columns
  - Take advantage of **implicit compression** of normal columns
    - Much better than just storing and compressing JSON
  - Utilize more UPDATE-friendly designs
  - Clearer indexing strategies

JSON
Additional Topics

JSON Functions

```
SELECT DISTINCT proname FROM pg_proc WHERE proname like 'jsonb%';
```

jsonb_agg
jsonb_agg_finalfn
jsonb_agg_transfn
jsonb_array_element
jsonb_array_element_text
jsonb_array_elements
jsonb_array_elements_text
jsonb_array_length
jsonb_build_array
jsonb_build_object
jsonb_cmp
jsonb_concat
jsonb_contained
jsonb_contains
jsonb_delete
jsonb_delete_path
jsonb_each
jsonb_each_text
jsonb_eq
jsonb_exists
jsonb_exists_all
jsonb_exists_any
jsonb_extract_path
jsonb_extract_path_text
jsonb_ge
jsonb_gt
jsonb_hash
jsonb_hash_extended
jsonb_in
jsonb_insert
jsonb_le
jsonb_lt
jsonb_ne
jsonb_object
jsonb_object_agg
jsonb_object_agg_finalfn
jsonb_object_agg_transfn
jsonb_object_field
jsonb_object_field_text
jsonb_object_keys
jsonb_out
jsonb_path_exists
jsonb_path_exists_opr
jsonb_path_exists_tz
jsonb_path_match
jsonb_path_match_opr
jsonb_path_match_tz
jsonb_path_query
jsonb_path_query_array
jsonb_path_query_array_tz
jsonb_path_query_first
jsonb_path_query_first_tz
jsonb_path_query_tz
jsonb_populate_record
jsonb_populate_recordset
jsonb_pretty
jsonb_recv
jsonb_send
jsonb_set
jsonb_set_lax
jsonb_strip_nulls
jsonb_to_record
jsonb_to_recordset
jsonb_to_tsvector
jsonb_typeof

- 65 different functions for manipulating JSON and JSONB
- 11 are for operators
- 14 for JSON path
- Others utility functions

PLv8

- Procedural Language handler for Javascript
- Create functions and execute them in JS

- Some issues with stability of PL/v8
- No longer available on some platforms

MongoDB Foreign Data Wrapper

- Open source EXTENSION, maintained and supported by EDB

- Query the BSON data directly in MongoDB
- Set up a Foreign Table that maps
  - BSON to JSONB
  - BSON to PostgreSQL column data
  - or a mix of those two
- Send INSERTs, UPDATEs and DELETEs thru updatable views
- Caches connection data to allow fast response

Sample MongoDB data

MongoDB server:

```
-- Create database
use testdb

-- Create and insert data(2 documents) into the collection 'warehouse'

db.warehouse.insert ({"_id" :
ObjectId("58a1ebbaf543ec0b90545859"),"warehouse_id" :
NumberInt(1),"warehouse_name" : "UPS","warehouse_created" :
ISODate("2014-12-12T07:12:10Z")});
db.warehouse.insert ({"_id" :
ObjectId("58a1ebbaf543ec0b9054585a"),"warehouse_id" :
NumberInt(2),"warehouse_name" : "Laptop","warehouse_created" :
ISODate("2015-11-11T08:13:10Z")});
```

- Create sample DB
- Insert some data

Access MongoDB data

```
CREATE EXTENSION mongo_fdw;
CREATE SERVER mongo_server
       FOREIGN DATA WRAPPER mongo_fdw OPTIONS (...);
CREATE USER MAPPING FOR myuser
       SERVER mongo_server OPTIONS (...);
CREATE FOREIGN TABLE warehouse (
 _id                 name,
 warehouse_id        int,
 warehouse_name      text,
 warehouse_created   timestamptz
)  SERVER mongo_server
  OPTIONS (database 'db', collection 'warehouse');
SELECT * FROM warehouse WHERE warehouse_id = 1;
_id               | 53720b1904864dc1f5a571a0
warehouse_id      | 1
warehouse_name    | UPS
warehouse_created | 2014-12-12 12:42:10+05:30
(1 row)
```

- Access MongoDB server
- Foreign Table
  - IMPORT FOREIGN SCHEMA not yet available
- Access data

JSON and The SQL Standard

SQL Standard Compliance

- https://www.postgresql.org/docs/current/features.html

- **ISO/IEC 9075-1 Framework (SQL/Framework)**
- **ISO/IEC 9075-2 Foundation (SQL/Foundation)**
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- **ISO/IEC 9075-9 Management of External Data (SQL/MED)**
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- **ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)**
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- **ISO/IEC 9075-14 XML-related specifications (SQL/XML)**
- ISO/IEC 9075-15 Multi-dimensional arrays (SQL/MDA)
- ISO/IEC 9075-16 SQL Property Graph Queries SQL/PGQ

JSON and SQL/JSON

```
SELECT  /* Current PostgreSQL */
 json_build_object(
     'code', f.code,
     'title', f.title,
     'did', f.did
 ) AS paramount
FROM films AS f WHERE did = 103;
```

```
SELECT      /* SQL/JSON */
 JSON_OBJECT(
     'code' VALUE f.code,
     'title' VALUE f.title,
     'did' VALUE f.did
 ) AS paramount
FROM films AS f WHERE did = 103;
```

- PostgreSQL already supported many JSON features

- SQL Standard has adopted the syntax proposal from Oracle/MySQL, so we must add new implementations
- ***Lots of work!!!***

Conclusions

Vision

- PostgreSQL will **actively follow standards** from SQL, IEEE, OGC, IETF (RFCs), Unicode etc..
    - (and contribute if possible)
    - **More standards compliance features coming in PG15+**
- “Hyperconverged Postgres” combines multiple types of data into one integrated, robust and secure DBMS, with specialized data types and supporting data types
    - Relational data for operations and analytics
    - Document data in JSON/XML/Full Text
    - Time Series
    - Temporal/Historical
    - Graph
    - GIS

EDB Value Add

- **Support** for all Production versions of PostgreSQL
- **RDBA** for JSON applications
- **pgAdmin** and **PEM** to manage your databases
- **Maintaining and Extending PostgreSQL**
- **Expertise**… thanks to my colleagues for blogs and feedback
    - Boriss Mejias
    - Thom Brown
    - Dave Page
    - Marco Nenciarini
    - Andrew Dunstan
    - Mark Linster
    - Priti Sarode

End of Part 2

simon.riggs@enterprisedb.com

EDB™