PostgreSQL: The NoSQL Way

Hans-Jürgen Schönig

November 4, 2015

Requirement: Unstructured data

Structured vs. unstructured

- Relational systems are a traditional way to store data.
- The main principle:
  - "Think first".
  - Create a structure
  - Load the data

Unstructured data

- Content can change at runtime
- Fields might be added and changed on the fly
- This can be done in the relational model as well:
  - However, it is harder
  - DDLs in PostgreSQL need as little locking as possible

Changing structures

- Changing relations between objects is hard
  - For example: Changing 1:1 to 1:n or to m:n
- Changing columns and types is relatively easy

The NoSQL approach

- Key / Value store
- Simple distributed keys

A typical problem: An example

- Is it possible to write software without structure?
- Example: A simple point ...
  - A normal point: (x, y)
- GPS:
  - 48 07'29.0"N 16 33'27.0"E
  - 48.124722, 16.557500
- North / South is the y-coordinate and not the x-coordinate

Structure

- Somebody has to know about the structure of the data
- It is either the app or the database or both

PostgreSQL functionality

New data types

- PostgreSQL provides special data types for a NoSQL workload:
  - hstore
  - json
  - jsonb
- JSON is RFC 7159 compliant

hstore: Key / Value Store

- hstore is a traditional key / value store
- hstore is a proprietary format
- It has been around for some year
- The storage format is highly efficient

json documents

- The json-type validates input data
- Internally data is stored as text
- Special operators are available for json
- Use case: Ideal for data, which is stored but not read frequently.

jsonb: Binary json

- PostgreSQL validates data and stores the content in a binary format.
- Efficiency is key to success.
- Reads do not have to parse data all over again.
- Use case: jsonb is usually the method of choice

What does it look like?


```
SELECT '5'::json;

-- Array elements (elements need not be of same type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"],
    "tags": {"a": 1, "b": null}}'::json;
```

Order inside the document (1)


```
SELECT '{"bar": "baz", "balance": 7.77,
    "active":false}'::json;
                       json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)
```

Order inside the document (2)


```
SELECT '{"bar": "baz", "balance": 7.77,
    "active":false}'::jsonb;
                        jsonb
-----------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

Simple operations (1)

- Checking for existence

```
test=# SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
 ?column?
----------
 t
(1 row)
```

Simple operations (2)

- Does one document contain the other?

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

Simple operations (3)

- Extract elements

```
test=# SELECT '[{"a":"foo"},{"b":"bar"},
    {"c":"baz"}]'::json->2;
  ?column?
-------------
 {"c":"baz"}
(1 row)
```

Simple operations (4)

- Fetching data using keys

```
test=# SELECT '{"a": {"b":"foo"}}'::json->'a';
  ?column?
--------------
 {"b":"foo"}
(1 row)
```

Simple operations (5)

- Accessing a path

```
test=# SELECT '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}';
   ?column?
---------------
 {"c": "foo"}
(1 row)
```

Creating and disassembling documents

Relational -> Json

- Relational data can be transformed to json easily

A simple example


```
test=# SELECT row_to_json(x)
    FROM (VALUES (1, 2), (3, 4)) AS x;
        row_to_json
------------------------------
 {"column1":1,"column2":2}
 {"column1":3,"column2":4}
(2 rows)
```

Aggregating json documents


```
test=# SELECT json_agg(row_to_json(x))
    FROM (VALUES (1, 2), (3, 4)) AS x;
                        json_agg
-----------------------------------------------------
 [{"column1":1,"column2":2}, {"column1":3,"column2":4}]
(1 row)
```

Transforming json documents


```
test=# SELECT *
    FROM json_each('{"a":"foo", "b":"bar"}');
 key | value
-----+-------
 a   | "foo"
 b   | "bar"
(2 rows)
```

json -> record


```
test=# SELECT *
    FROM    json_to_record('{"a":1,"b":[1,2,3],"c":"bar"}')
        AS x(a int, b text, d text);
 a |    b    | d
---+---------+---
 1 | [1,2,3] |
(1 row)
```

Indexing

Indexing json

- It is easily possible to index single fields.
- However, using GIN to index the entire document is easier
- A GIN index can index every field in the document

GIN: Made simple . . .


```
CREATE INDEX idx_name ON t_table USING gin (json_field);
```

Performance:

- Just two words: IT ROCKS !

GIN Internals

- Internally GIN uses varbit encoding
- A GIN index is small
- All fields are indexed

Conclusion

PostgreSQL

- PostgreSQL offers a variety of features to store unstructured data.
- Extensions can be written easily.
- Unstructured data can be stored in a relational powerhouse.
- More functionality is yet to come.

Finally . . .

Any questions?