PostgreSQL, The Big, The Fast and The (NOSQL on) Acid

Federico Campoli

09 Jan 2016

Table of contents

Table of contents

The Big

Image by Caitlin - https://www.flickr.com/photos/lizard_queen

PostgreSQL, an history of excellence

- Created at Berkeley in 1982 by database's legend Prof. Stonebraker
- In the 1994 Andrew Yu and Jolly Chen added the SQL interpreter
- In the 1996 becomes an Open Source project.
- The project's name changes in PostgreSQL

PostgreSQL, an history of excellence

- Created at Berkeley in 1982 by database's legend Prof. Stonebraker
- In the 1994 Andrew Yu and Jolly Chen added the SQL interpreter
- In the 1996 becomes an Open Source project.
- The project's name changes in PostgreSQL
- Fully ACID compliant
- High performance in read/write with the MVCC
- Tablespaces

PostgreSQL, an history of excellence

- Created at Berkeley in 1982 by database's legend Prof. Stonebraker
- In the 1994 Andrew Yu and Jolly Chen added the SQL interpreter
- In the 1996 becomes an Open Source project.
- The project's name changes in PostgreSQL
- Fully ACID compliant
- High performance in read/write with the MVCC
- Tablespaces
- Runs on almost any unix flavour
- From the version 8.0 is native on *cough* MS Windows *cough*
- HA with hot standby and streaming replication
- Heterogeneous federation

PostgreSQL, an history of excellence

- Created at Berkeley in 1982 by database's legend Prof. Stonebraker
- In the 1994 Andrew Yu and Jolly Chen added the SQL interpreter
- In the 1996 becomes an Open Source project.
- The project's name changes in PostgreSQL
- Fully ACID compliant
- High performance in read/write with the MVCC
- Tablespaces
- Runs on almost any unix flavour
- From the version 8.0 is native on *cough* MS Windows *cough*
- HA with hot standby and streaming replication
- Heterogeneous federation
- Procedural languages (pl/pgsql, pl/python, pl/perl...)
- Support for NOSQL features like HSTORE and JSON

Development

- Old ugly C language
- New development cycle starts usually in June
- New version released usually by the end of the year
- At least 4 LTS versions
- Can be extended using shared libraries
- Extensions (from the version9.1)
- BSD like license

Limits

- Database size. No limits.
- Table size, 32 TB
- Row size 1.6 TB
- Rows in table. No limits.
- Fields in table 250 - 1600 depending on data type.
- Tables in a database. No limits.

Data types

Alongside the general purpose data types PostgreSQL have some exotic types.

- Range (integers, date)
- Geometric (points, lines etc.)
- Network addresses
- XML
- JSON
- HSTORE (extension)

Surprise surprise!

PostgreSQL 9.5, UPSERT, Row Level Security, and Big Data

Release date: 2016-01-07

- UPSERT aka INSERT, ON CONFLICT UPDATE
- Row Level Security, allows security "policies" which filter which rows particular users are allowed to update or view.

BIG DATA!

- BRIN - Block Range Indices
- CUBE, ROLLUP and GROUPING SETS
- IMPORT FOREIGN SCHEMA
- TABLESAMPLE

Table of contents

The Fast

Image by Hein Waschefort - http://commons.wikimedia.org/wiki/User:Hein_waschefort

Page layout

A PostgreSQL's data file is an array of fixed length blocks called pages. The default size is 8kb.

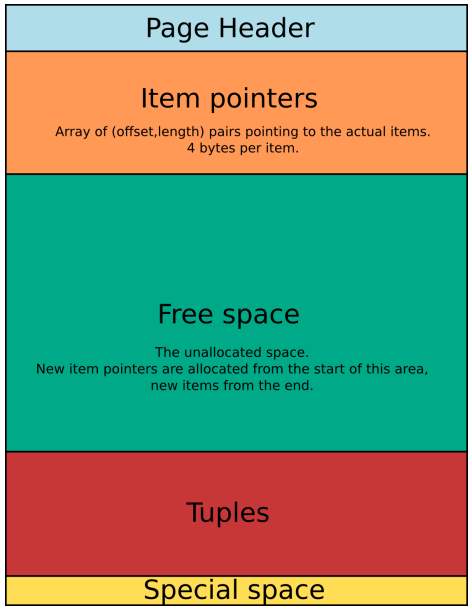
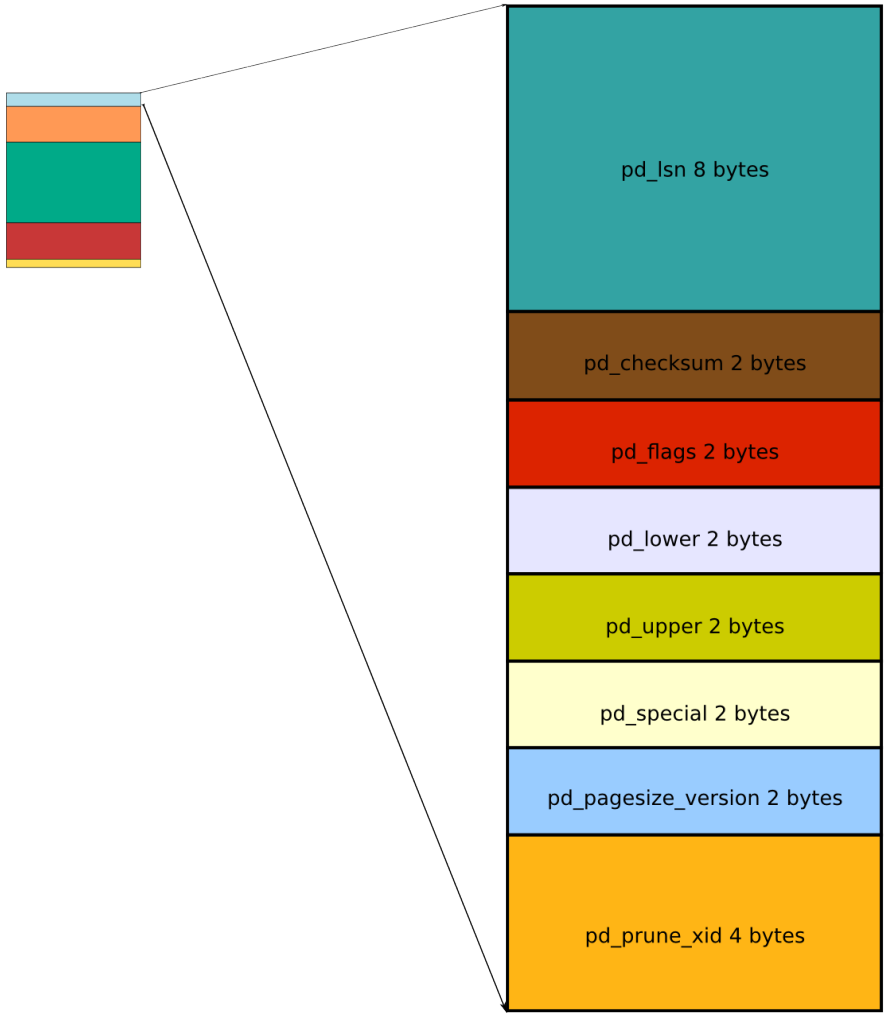


Figure : Index page

Page layout

Each page have an header used to enforce the durability, and the optional page's checksum. There are some pointers used to track the free space inside the page.



Tuple layout

Just after the header there is a list of pointers to the physical tuples stored in the page's end. Each tuple is and array of raw data, called datum. The nature of this datum is unknown to the postgres process. The datum becomes the data type when PostgreSQL loads the page in memory. This requires a system catalogue look up.

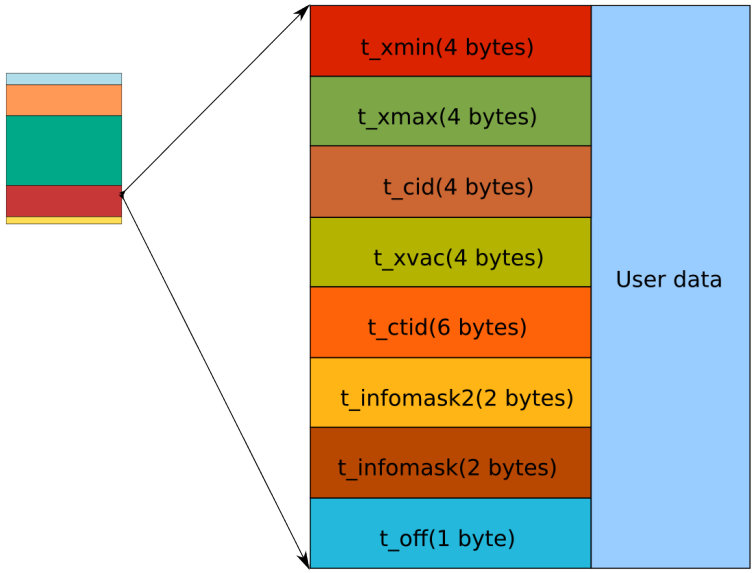


Figure : Tuple structure

The tuple's header is used in the MVCC.

The magic of the MVCC

- Any operation in PostgreSQL happens through transactions.
- By default when a single statement is successfully completed the database commits automatically the transaction.
- It's possible to wrap multiple statements in a single transaction using the keywords [BEGIN;]..... [COMMIT; ROLLBACK]
- The minimal possible level the transaction isolation is READ COMMITTED.
- PostgreSQL from 9.2 supports the snapshot export to other sessions.

There's no such thing like an update

Where's the catch?

There's no such thing like an update

Where's the catch?
PostgreSQL actually NEVER performs an update.

The UPDATE behaviour is to add a new row version and to keep the old one for read consistency.

Dead tuples and VACUUM

The tuples left in place for read consistency are called dead.

A dead tuple is left in place for any transaction that should see it. This adds overhead to any I/O operation.

- VACUUM clears the dead tuples
- VACUUM is designed to have the minimal impact on the database normal activity
- VACUUM removes only dead tuples no longer visible to the open transactions
- VACUUM prevents the xid wraparound failure

Table of contents

The (NOSQL on) Acid

JSON

JSON - JavaScript Object Notation

- The version 9.2 adds JSON as native data type
- The version 9.3 adds the support functions for JSON
- JSON is stored as text
- JSON is parsed and validated on the fly
- The 9.4 adds JSONB (binary) data type

JSON - Examples
From record to JSON

```
postgres=# SELECT row_to_json(ROW(1,'foo'));
     row_to_json
---------------------
 {"f1":1,"f2":"foo"}
(1 row)
```

Expanding JSON into key to value elements

```
postgres=# SELECT * from json_each('{"a":"foo", "b":"bar"}');

 key | value
-----+-------
 a   | "foo"
 b   | "bar"
(2 rows)
```

HSTORE

HSTORE is a custom data type used to store key to value items

- Is an extension
- Data stored as text
- A shared library does the magic transforming the datum in HSTORE
- Is similar to JSON without nested elements

HSTORE - Examples
From record to HSTORE

```
postgres=# SELECT hstore(ROW(1,2));
        hstore
----------------------
 "f1"=>"1", "f2"=>"2"
(1 row)
```

HSTORE expansion to key to value elements

```
postgres=# SELECT * FROM each('a=>1,b=>2');
 key | value
-----+-------
 a   | 1
 b   | 2
(2 rows)
```

JSON and HSTORE

There is a subtile difference between HSTORE and JSON. HSTORE is not a native data type.

The JSON is a native data type and the conversion happens inside the postgres process.

The HSTORE requires the access to the shared library.

Because the conversion from the raw datum happens for each tuple loaded in the shared buffer can affect the performance's overall.

JSONB

Because JSON is parsed and validated on the fly and and this can be a bottleneck.

The new JSONB introduced with PostgreSQL 9.4 is parsed, validated and transformed at insert/update's time. The access is then faster than the plain JSON but the storage cost can be higher.
The functions available for JSON are also available in the JSONB flavour.

Some numbers

Let's create three tables with text,json and jsonb type fields.
Each record contains the same json element generated on
http://beta.json-generator.com/4kwCt-fwg


```
[ {
"_id": "56891aba27402de7f551bc91",
"index": 0,
"guid": "b9345045-1222-4f71-9540-6ed7c8d2ccae",
"isActive": false,
.............
3,
{
"id": 1,
"name": "Bridgett Shaw"
}
],
"greeting": "Hello, Johnston! You have 8 unread messages.",
"favoriteFruit": "apple"
}
]
```

Some numbers

```
DROP TABLE IF EXISTS t_json ;
DROP TABLE IF EXISTS t_jsonb ;
DROP TABLE IF EXISTS t_text ;

CREATE TABLE t_json as
SELECT
'<JSON ELEMENT>'::json as js_value
FROM
generate_series(1,100000);
Query returned successfully: 100000 rows affected, 14504 ms execution time.


CREATE TABLE t_text as
SELECT
'<JSON ELEMENT>'::text as t_value
FROM
generate_series(1,100000);
Query returned successfully: 100000 rows affected, 14330 ms execution time.


CREATE TABLE t_jsonb as
SELECT
'<JSON ELEMENT>'::jsonb as jsb_value
FROM
generate_series(1,100000);
Query returned successfully: 100000 rows affected, 14060 ms execution time.
```

Table size

```
SELECT
        pg_size_pretty(pg_total_relation_size(oid)),
        relname
FROM
        pg_class
WHERE
        relname LIKE 't\_%'
;

pg_size_pretty | relname
----------------+---------
 270 MB         | t_json
 322 MB         | t_jsonb
 270 MB         | t_text
(3 rows)
```

Sequential scans

TEXT

```
EXPLAIN  (BUFFERS, ANALYZE) SELECT * FROM t_text;

Seq Scan on t_text  (cost=0.00..1637.00 rows=100000 width=18) (actual time
    =0.016..17.624 rows=100000 loops=1)
   Buffers: shared hit=637
 Planning time: 0.040 ms
 Execution time: 28.967 ms
(4 rows)
```

Sequential scans

JSON

```
EXPLAIN  (BUFFERS, ANALYZE) SELECT * FROM t_json;

Seq Scan on t_json  (cost=0.00..1637.09 rows=100009 width=32) (actual time
     =0.018..15.443 rows=100000 loops=1)
   Buffers: shared hit=637
 Planning time: 0.045 ms
 Execution time: 25.268 ms
(4 rows)
```

Sequential scans

JSONB

```
EXPLAIN  (BUFFERS, ANALYZE) SELECT * FROM t_jsonb;

Seq Scan on t_jsonb  (cost=0.00..1637.00 rows=100000 width=18) (actual time
     =0.015..18.943 rows=100000 loops=1)
   Buffers: shared hit=637
 Planning time: 0.043 ms
 Execution time: 31.072 ms
(4 rows)
```

Sequential scan with json access

TEXT

```
EXPLAIN  (BUFFERS, ANALYZE) SELECT t_value::json->'index' FROM t_text;

Seq Scan on t_text  (cost=0.00..2387.00 rows=100000 width=18) (actual time
     =0.159..7748.381 rows=100000 loops=1)
   Buffers: shared hit=401729
 Planning time: 0.028 ms
 Execution time: 7760.263 ms
(4 rows)
```

Sequential scan with json access

JSON

```
EXPLAIN  (BUFFERS, ANALYZE) SELECT js_value->'index' FROM t_json;

Seq Scan on t_json  (cost=0.00..1887.11 rows=100009 width=32) (actual time
     =0.254..5787.267 rows=100000 loops=1)
   Buffers: shared hit=401730
 Planning time: 0.044 ms
 Execution time: 5798.153 ms
(4 rows)
```

Sequential scan with json access

JSONB

```
EXPLAIN  (BUFFERS, ANALYZE) SELECT jsb_value->'index' FROM t_jsonb;

Seq Scan on t_jsonb  (cost=0.00..1887.00 rows=100000 width=18) (actual time
     =0.138..1678.222 rows=100000 loops=1)
   Buffers: shared hit=421729
 Planning time: 0.048 ms
 Execution time: 1688.752 ms
(4 rows)
```

Table of contents

Wrap up

- Schema less data are useful. They are flexible and powerful.
- Never forget the *update* strategy in PostgreSQL
- The lack of horizontal scalability in PostgreSQL can be a serious problem.
- An interesting project for a distributed cluster is PostgreSQL XL - http://www.postgres-xl.org/

- CitusDB is a powerful DWH oriented DBMS with horizontal scale capabilities - should become open source in the next release
- Never forget PostgreSQL is a RDBMS
- Get a DBA on board

Questions

Questions?

Contacts

- Twitter: 4thdoctor_scarf
- Personal blog: http://www.pgdba.co.uk
- PostgreSQL Book: http://www.slideshare.net/FedericoCampoli/postgresql-dba-01
- Brighton PostgreSQL Meetup: http://www.meetup.com/Brighton-PostgreSQL-Meetup/

License and copyright

This presentation is licensed under the terms of the Creative Commons Attribution NonCommercial ShareAlike 4.0

http://creativecommons.org/licenses/by-nc-sa/4.0/

- The elephant photo is copyright by Caitlin - https://www.flickr.com/photos/lizard_queen
- The cheetah photo is copyright by Hein Waschefort - http://commons.wikimedia.org/wiki/User:Hein_waschefort
- The elephant logos are copyright of the PostgreSQL Global Development Group - http://www.postgresql.org/

PostgreSQL, The Big, The Fast and The (NOSQL on) Acid

Federico Campoli

09 Jan 2016