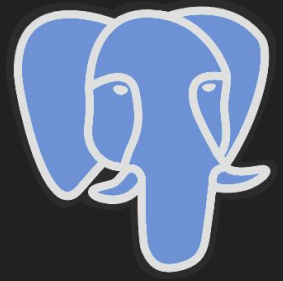
PostgreSQL



PostgreSQL

Veronika Aksamítová, Dominik Toušek, Dávid Veliký, Miroslava Voglová

PostgreSQL

- object-relational database management system (ORDBMS)
- based on POSTGRES
- developed by PostgreSQL Global Development Group

- Supported data types:
  hierarchical document data, key-value and relational data
- Master - slave database architecture (master/ read/write, standby/slave servers)
- MVCC (multiversion concurrency control) - three levels of transaction isolation:
  Read Committed, Repeatable Read and Serializable

PostgreSQL history

- based on POSTGRES (1986 - University of California (professor Michael Stonebraker))
- 1994 - added SQL interpreter and new name Postgres95
- 1996 - renamed as PostgreSQL

PostgreSQL release history

1999 - MVCC

2002- PL/Python

2005 - savepoints, two-phase commit, table partitioning, shared row locking

2010 - binary streaming replication, hot standby

2011 - k-nearest neighbors (k-NN) indexing

2012 - cascading streaming replication, native JSON support, space-partitioned GiST indexes

2013 - dedicated JSON operators

2014 - JSONB data type, GiN index improvements

2016 - JSONB-modifying operators/functions, BRIN (Block Range Indexes) (speed up queries on very large tables)

PostgreSQL users

- Yahoo! (web user behavioral analysis)
- Geni.com (genealogy database)
- Skype
- MusicBrainz (online music encyclopedia)
- The International Space Station (collecting telemetry data in orbit)
- Instagram

NoSQL like capabilities

PostgreSQL - useful data types

- **Hstore**
  - **Implemented as hash map**
  - **Simple key-value store inside relational database**
- **JSON**
  - **Used for JSON document storage and retrieval as whole document**
- **JSONB**
  - **Used for manipulation with JSON document contents**

JSON vs JSONB format

- JSON validity check on insert and update is performed on both formats

JSON

- Data are stored as plain text
- Only basic indexes are available (hash and btree) for the whole column only

JSONB

- Data are stored in binary format
- Slower insert due to conversion process - text -> binary
- Faster read operations - no need to reparse for every query
- Supports indexing of JSON document contents

JSON and JSONB operators

JSON

- Array element/object accessor ->
- Array element/object as text accessor ->>
- Path accessor #>
- Path accessor with conversion to text #>>

JSONB

- Containment operators @>, <@
- Existence operators ?, ?|, ?&

JSON and JSONB CONSTRAINTS

- Enforcing JSON field presence -> IS NOT NULL
- Constraints on JSON fields -> enforcing some range of numbers, ...

```
CREATE TABLE employee (
   data JSON,
   CONSTRAINT validate_id CHECK ((data->>'id')::integer >= 1 AND (data->>'id') IS NOT NULL ),
   CONSTRAINT validate_name
                  CHECK (length(data->>'name') > 0 AND (data->>'name') IS NOT NULL )
);
```

Indexes for JSONB

- BTree indexes
    - Quick but single purpose
- GIN - Generalized Inverted Index
    - Quick and (should be) flexible
    - 2 built-in JSONB specific operators implementations

GIN indexes - simple index vs expression index

Simple index

- Could be used only when operators are applied directly to indexed column
- **Creation:** CREATE INDEX test1_col1_idx ON test1 (col1);
- **Usage:** SELECT * FROM test1 WHERE col1 = 'value';

Expression index

- Could be used when operators are applied to expression results
- **Creation:** CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
- **Usage:** SELECT * FROM test1 WHERE lower(col1) = 'value';

GIN indexes - operators implementations

jsonb_ops (default)

- **Supported operators:** @>, ?, ?& and ?|
- Stores separately all parts of path and value
- Could be very large

jsonb_path_ops

- **Supported operators:** @>
- Stores the whole path to value, including the value, as one index entry
- Smaller and quicker than default implementation

Indexes - example data

- **Table schema:** CREATE TABLE demo_users (
  id INTEGER NOT NULL DEFAULT NEXTVAL('demo_users_id_seq'),
  json_data jsonb,
  CONSTRAINT demo_users_pk PRIMARY KEY (id)
  );

- **Jsonb column format:** {
  “person” : {
  “first_name”:”<name>”,
  “last_name”:”<name>”,
  “gender”:”<male/female>”
  }
  }

(1.000.000 randomly generated rows in table)

GIN indexes - jsonb_ops example

- **Index creation:** CREATE INDEX demo_users_gin_index ON demo_users
  USING gin ((json_data -> 'person' -> 'first_name'));

```
SELECT COUNT(*) FROM demo_users
  WHERE json_data -> 'person' -> 'first_name' @> '"8b1cd13e7d0be574ccec657072ee9212"';
```

- **Index creation: ~14.2s**
- **Access time without index: ~500ms**
- **Access time with index: ~32ms**

GIN indexes - jsonb_path_ops example

- **Index creation:** CREATE INDEX demo_users_gin_path_index ON demo_users
  USING gin ((json_data -> 'person' -> 'first_name') jsonb_path_ops);

```
SELECT COUNT(*) FROM demo_users
  WHERE json_data -> 'person' -> 'first_name' @> '"8b1cd13e7d0be574ccec657072ee9212"';
```

- **Index creation: ~6s**
- **Access time without index: ~500ms**
- **Access time with index: ~18ms**

BTree index - example

- **Index creation:** CREATE INDEX demo_users_btree_index
  on demo_users ((json_data #>> '{person,first_name}'));

SELECT COUNT(*) FROM demo_users
WHERE json_data #>> '{person,first_name}' = '"8b1cd13e7d0be574ccec657072ee9212"';

- **Index creation: ~16.6s**
- **Access time without index: ~500ms**
- **Access time with index: ~14ms**

Scalability

High availability and replication

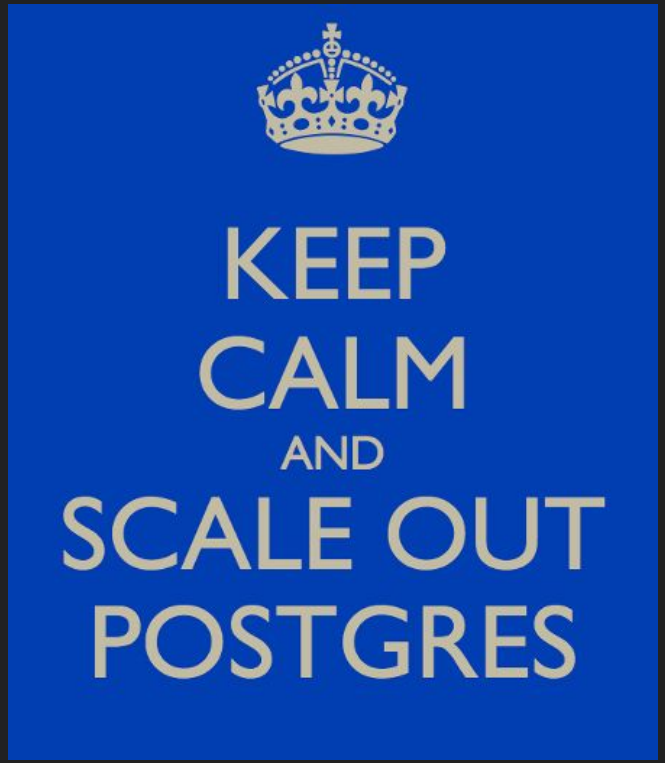
- before 9.0 was achieved by external packages like Slony-I
  - trigger-based, causes overhead on master
  - single master only, master is a single point of failure
  - no good failover system for electing a new master or having a failed master rejoin the cluster
  - slave can execute read-only queries
  - suffers from $O(N^2)$ communications where N = number of nodes
  - table-level granularity allows complex data partitioning configuration
- from 9.0 streaming replication implemented in Postgres core
  - WAL (write-ahead log)
  - slaves can execute queries
  - no overhead on master
- asynchronous replication causes lag (bigger in trigger-based) that can cause inconsistent view of data and possible data lost on fail-over

High availability and replication

when replication implemented in Postgre core is not usable:

- master and slave have different postgres versions
- master and slave on different hardware platform
- master and slave are not identical (multiple databases on master, but only some on slave)

Possibilities of horizontal scalability



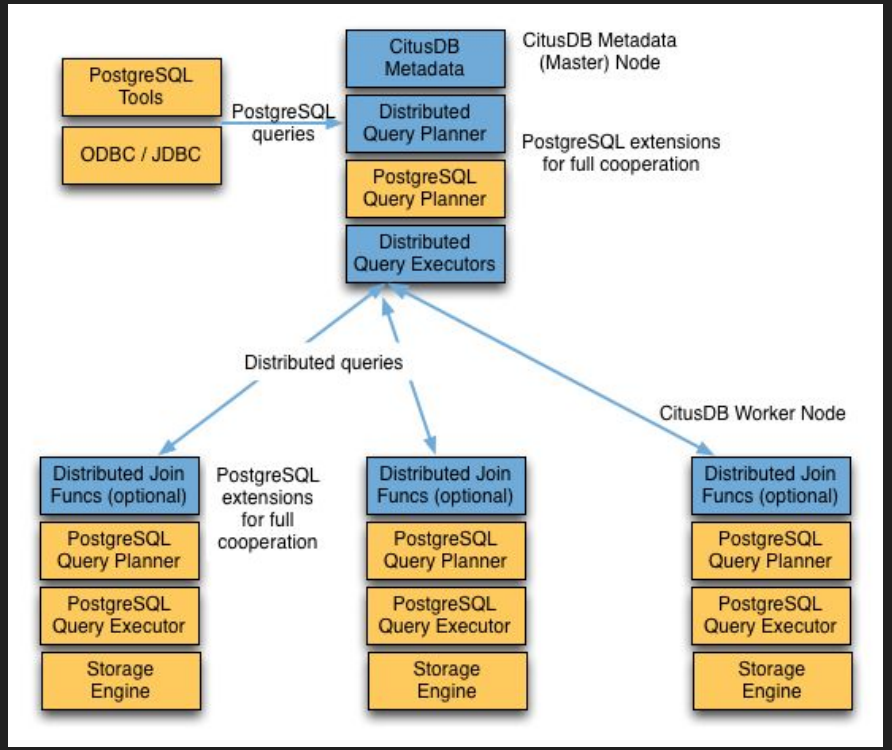
- scaling out through extensions
  - available from Postgres version 9.4
  - no data migration needed
  - no change for already running application needed
  - all core Postgres functionality kept
- other databases capable of horizontal scaling with (modified) Postgres core
  - usually better performance
  - some functionality may not be available
  - examples: GreenPlum database, Postgres-XL

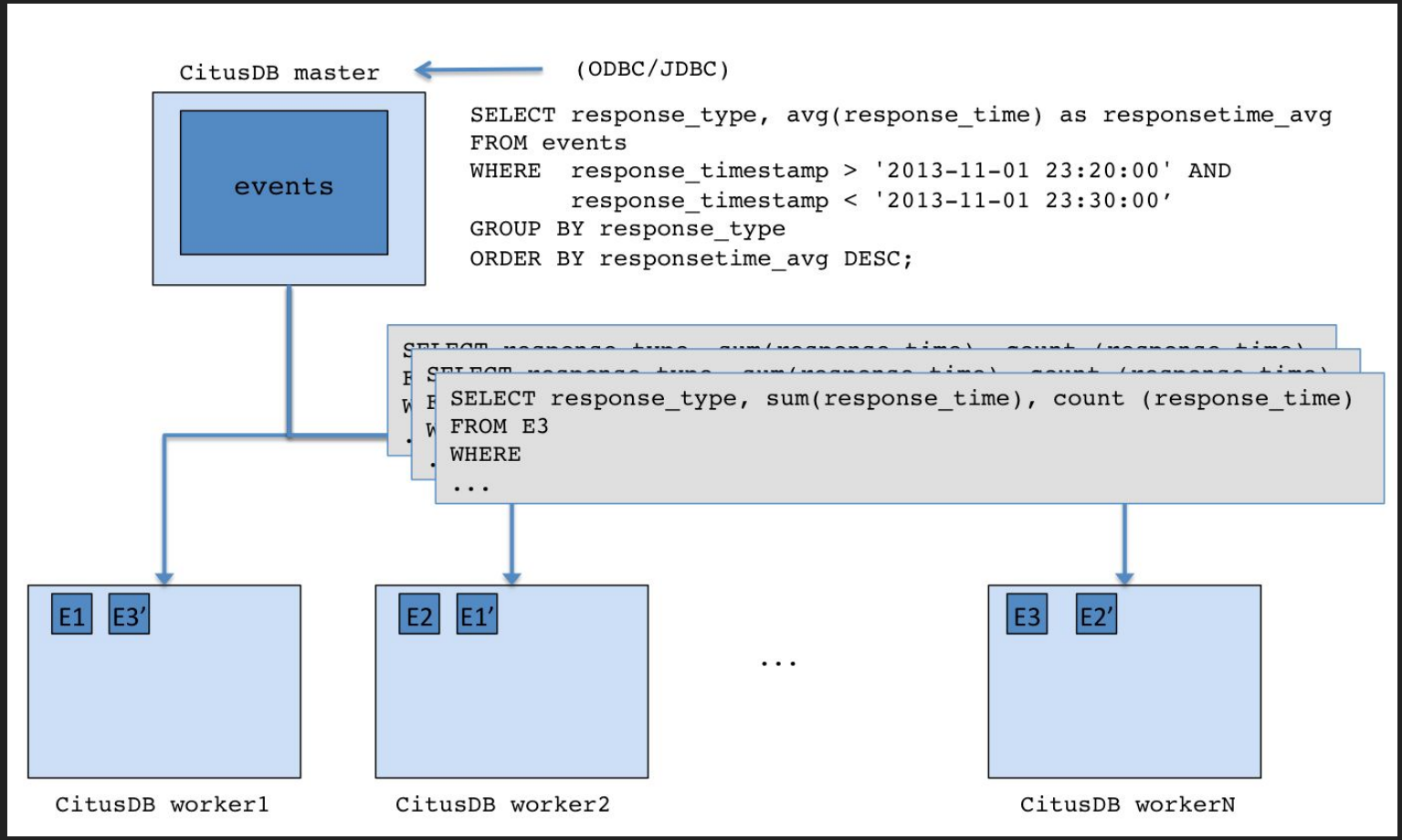
Citus

- turns Postgres into a distributed database
- shared nothing architecture
- enables transparently shard data across multiple PostgreSQL instances
- small shards (default 1GB) for easier rebalancing workload on nodes
- one master node holds metadata about shards in the cluster and parallelizes incoming queries
- each shard is replicated on multiple cluster nodes so the loss of a single node does not impact data availability

Citus - processing query

- the Citus master partitions query into smaller query fragments
- each query fragment can be run independently on a worker shard
- the query fragments are assigned to workers
- master oversees execution of fragments, merges their results, and returns the final result to the user
- the master also applies optimizations that minimize the amount of data transferred across the network.




CitusDB master
(ODBC/JDBC)
events
SELECT response_type, avg(response_time) as responsetime_avg
FROM events
WHERE response_timestamp > '2013-11-01 23:20:00' AND
response_timestamp < '2013-11-01 23:30:00'
GROUP BY response_type
ORDER BY responsetime_avg DESC;
SELECT response_type, sum(response_time), count (response_time)
FROM E3
WHERE
...
E1
E3'
E2
E1'
...
E3
E2'
CitusDB worker1
CitusDB worker2
CitusDB workerN

Citus - node failures

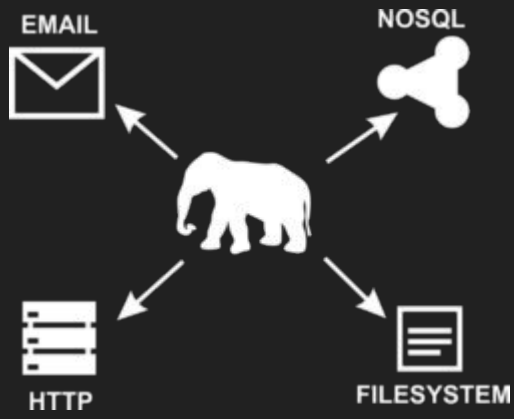
- when a worker node fails, the master node automatically switch to other workers which have a copy of the shard
- when a worker fail mid-query, query is completed by re-routing the query fragment, to other worker
- if a worker is permanently down, users can easily rebalance the shards onto other workers to maintain the same level of availability.
- for automatic recovery from failure of master, it is important to have hot standby node created through streaming replication of PostgreSQL
- if there is no such node, master can be reconstructed only manually

Interaction with NoSQL databases

FDW

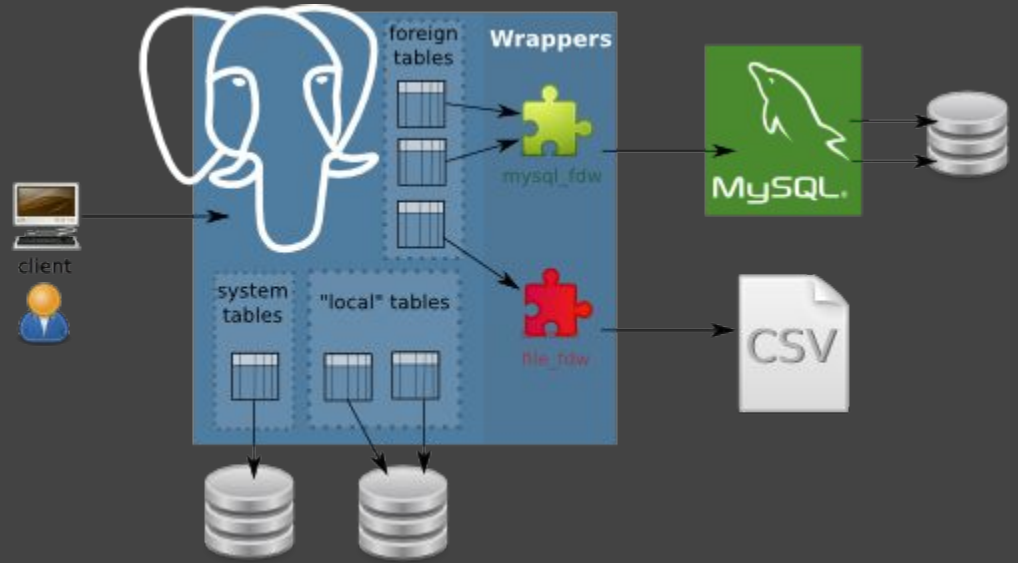
- 2003 SQL/MED
- 2011 PostgreSQL 9.1 read-only support
- 2013 PostgreSQL 9.3 write support
- now variety of FDW:
    - mongo_fdw
    - hadoop_fdw
    - etc. (currently available wrappers)

  ...PGDG support
- Multicorn, Holycorn



img link

FDW - schema



img link

FDW - How To ...

- Load extension first time after install

```
CREATE EXTENSION mongo_fdw;
```

- Create server object

```
CREATE SERVER mongo_server
FOREIGN DATA WRAPPER mongo_fdw
OPTIONS (address '127.0.0.1', port '27017');
```

- Create user mapping

```
CREATE USER MAPPING FOR postgres
SERVER mongo_server
OPTIONS (username 'mongo_user', password 'mongo_pass');
```

- Create foreign table

```
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 table

```
SELECT * FROM warehouse WHERE warehouse_id = 1;
```

Questions?

Sources

http://bsnyderblog.blogspot.sk/2011/09/installing-postgresql-90-on-mac-os-x.html (logo PostreSQL)

https://en.wikipedia.org/wiki/PostgreSQL

http://peter.eisentraut.org/blog/2015/03/03/the-history-of-replication-in-postgresql/

http://www.postgresql.org

https://www.citusdata.com

http://blog.2ndquadrant.com/jsonb-type-performance-postgresql-9-4/