Postgres and Vacuum

Transactions, MVCC, Vacuum, and why it matters

Charly Batista

Percona

charly.batista@percona.com

Who am I?

I'm Charly Batista :)

- PostgreSQL Tech Leader @ Percona
- Database lover
- Working with IT for over 20 years
- Craft beer and coffee lover
- You can find me at:

  https://github.com/elchinoo
  https://www.linkedin.com/in/charlyfrankl

Agenda

MVCC

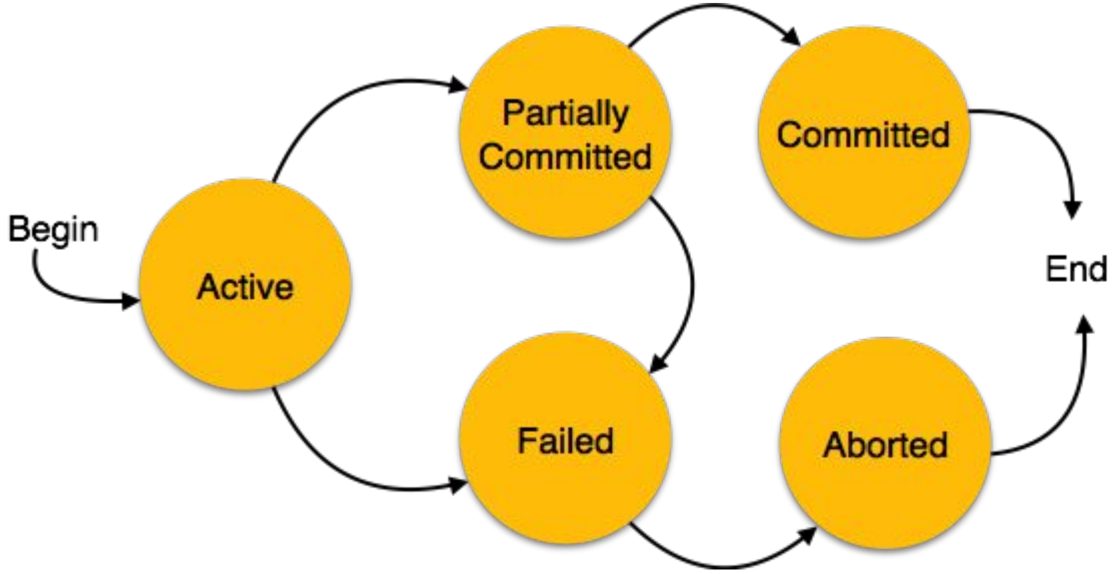
- What are and why do we need transactions?
- MVCC, what exactly is it?
- MVCC on Postgres

Vacuum and Autovacuum

- What is vacuum?
- How about autovacuum?
- Settings and tuning

What are transactions?

- A transaction is a logical, atomic unit of work that contains one or more SQL
- Transaction gives some guarantees:
  - Atomicity
  - Consistency
  - Isolation
  - Durability



How does it look like?

```
pagila=# begin;
BEGIN
pagila=# insert into city values(default, 'Brasilia', 15, now());
INSERT 0 1
pagila=# insert into city values(default, 'Shanghai', 23, now());
INSERT 0 1
pagila=# commit;
COMMIT
pagila=#
```

Atomicity

- An atomic transaction is an indivisible and irreducible series of database operations such that either all occur, or nothing occurs:
  - Indivisible
  - All or nothing
- Example bank transfer
  - John transfers $100 to Alice
    - *Withdraw $100 from John’s bank account*
    - *Deposit $100 into Alice’s bank account*

Atomicity

```
pagila=# begin;
BEGIN
pagila=# insert into country values(default, 'BRAZIL', now());
INSERT 0 1
pagila=# select * from country where country = 'BRAZIL';
 country_id | country |          last_update
------------+---------+------------------------------
        110 | BRAZIL  | 2022-05-10 09:33:23.82151+00
(1 row)

pagila=# insert into city values(default, 'Brasilia', 110, now());
INSERT 0 1
pagila=# select c.city_id, c.city, ct.country from city c join country ct on
ct.country_id = c.country_id where country = 'BRAZIL';
 city_id | city          | country
---------+--------------+-------------
     603 | 'Brasilia'   | 'BRAZIL'
(1 row)
pagila=# commit;
COMMIT
```

Consistency

- A transaction must keep the database in a valid state
- Guarantees
  - Any transactions started in the future necessarily see the effects of other transactions committed in the past
  - Database constraints are not violated, particularly once a transaction commits
  - Operations in transactions are performed accurately, correctly, and with validity, with respect to application semantics

Consistency

```
pagila=# begin;
BEGIN
pagila=# insert into city values(default, 'Brasilia', 110, now());
ERROR:  insert or update on table "city" violates foreign key constraint
"city_country_id_fkey"
DETAIL:  Key (country_id)=(110) is not present in table "country".

pagila=#
```

Isolation

- Determines how transaction integrity is visible to other users
- Makes the transaction think it is the only transaction working inside the database
- There are different levels of isolation
  - Read Uncommitted
  - Read Committed
  - Repeatable Read
  - Serializable
  - Postgres default transaction isolation = 'read committed'

Isolation

- Ideally prevent race conditions
- Also help prevent anomalies:
  - Dirty read
  - Non-repeatable read
  - Phantom read
  - Serialization anomaly

Durability

- Committed transactions can not be rolled back
- Committed transactions must survive

MVCC

- Multiversion concurrency control
- Concurrency control method commonly used by databases
- Optimistic - means no locking
  - Readers and Writers do not block each other
- First perform changes in a protected area then change the database state
- Main idea: Version your database (Multiversion :-D)

MVCC on Postgres

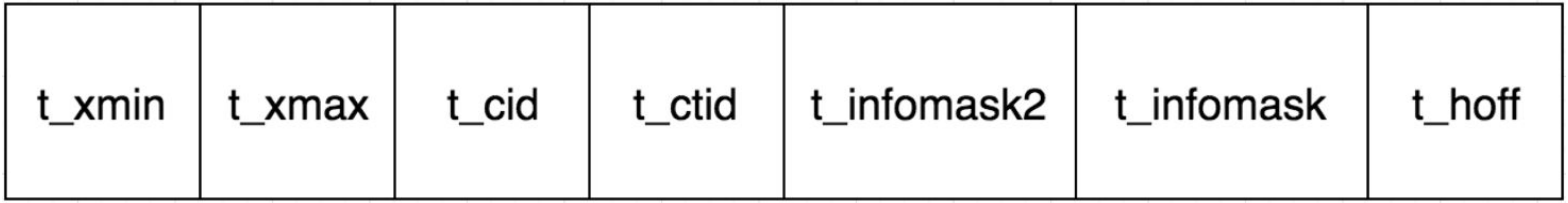
- Read uncommitted not really implemented on Postgres
- No Rollback segments for UNDO
  - UNDO management is within tables
- Rows are never really deleted
- A tuple contains the hidden columns to help managing transactions
  - xmin, xmax, cmin, cmax, …
- Transaction identifiers (xid or transaction IDs) are 32-bit unsigned integer
- ID’s 0, 1 and 2 are reserved.
- Can be inspected, for example “`select xmin, xmax, cmin, cmax, a from tbl;`”

MVCC on Postgres

- T1: Start transaction (txid 200)
- T2: Start transaction (txid 201)
- T3: Execute SELECT commands of txid 200 and 201
- T4: Execute UPDATE command of txid 200
- T5: Execute SELECT commands of txid 200 and 201
- T6: Commit txid 200
- T7: Execute SELECT command of txid 201

Heap Tuples

- Each Heap tuple in a table contains a HeapTupleHeaderData structure.



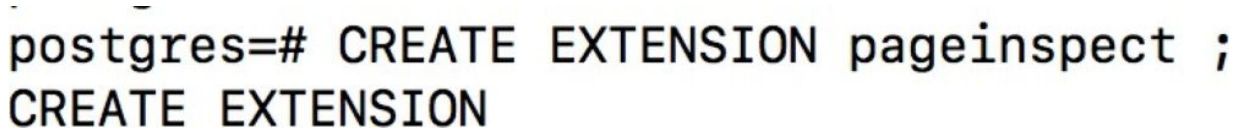
t_xmin	t_xmax	t_cid	t_ctid	t_infomask2	t_infomask	t_hoff

HeapTupleHeaderData Structure

- **t_xmin** : txid of the transaction that inserted this tuple
- **t_xmax** : txid of the transaction that issued an update/delete on this tuple and not committed yet or when the delete/update has been rolled back and 0 when nothing happened.
- **t_cid** : The position of the SQL command within a transaction that has inserted this tuple, starting from 0. If 5th command of transaction inserted this tuple, cid is set to 4.
- **t_ctid** : Contains the block number of the page and offset number of line pointer that points to the tuple.

Extension : pageinspect

- Included with the contrib module
- Show the contents of a page/block
- 2 functions we could use to get tuple level metadata and data
  - get_raw_page : reads the specified 8KB block
  - heap_page_item_attrs : shows metadata and data of each tuple
- Create the Extension pageinspect:



```
postgres=# CREATE EXTENSION pageinspect ;
CREATE EXTENSION
```

MVCC on Postgres

- Multiple versions are amazing but can be problematic:
  - Dead tuples occupies space on table
    - *Bloat issues*
  - Postgres has a 32 bit unsigned integer transaction ID:
    - *We have a limited number of available transactions;*
    - *We need a way to prevent the transaction ID to wraparound(?)*
    - *It means that it can and (if we don't do anything), it will wraparound!*
- **How can we solve those issues?**

VACUUM / AUTOVACUUM

- **VACUUM** — garbage-collect and optionally analyze a database
- Here we'll talk about 4 major variations of vacuum on Postgres:
  - Full
  - Freeze
  - Vacuum
  - Autovacuum
- **Only rows that are not in any currently running transactions can be vacuumed**
  - It means that long running transactions can prevent dead rows to be removed;
  - Long running transactions can prevent vacuum to freeze old transaction IDs;

VACUUM / AUTOVACUUM

- **VACUUM**: reclaims storage occupied by dead tuples
- Here we’ll talk about 4 major variations of vacuum on Postgres:
  - Full: rebuilds the table and returns empty space to the filesystem;
  - Freeze: runs an aggressive “freezing” of tuples to freeze transaction IDs;
  - Analyze: performs a VACUUM and then an ANALYZE for each selected table;
  - Autovacuum: a feature that automates the execution of VACUUM and ANALYZE;
- **Only rows that are not in any currently running transactions can be vacuumed**
  - It means that long running transactions can prevent dead rows to be removed;
  - Long running transactions can prevent vacuum to freeze old transaction IDs;

Autovacuum

- **Always have the parameter autovacuum set to ON**;
- Background Process : Stats Collector tracks the usage and activity information;
- We cannot really control when it runs;
- PostgreSQL identifies the tables needing vacuum or analyze depending on certain parameters, for example threshold and scale factor;
- **Threshold**: autovacuum_vacuum_threshold/autovacuum_analyze_threshold : Minimum number of obsolete records or DML's needed to trigger an autovacuum;
- **Scale factor**: autovacuum_vacuum_scale_factor/autovacuum_analyze_scale_factor: Fraction of the table records that will be added to the formula. For example, a value of 0.2 equals to 20% of the table records;

Autovacuum

- **VACUUM threshold** for a table := autovacuum_vacuum_scale_factor * number of tuples + autovacuum_vacuum_threshold
  - If the actual number of dead tuples in a table exceeds this effective threshold, due to updates and deletes, that table becomes a candidate for autovacuum
- **ANALYZE threshold** for a table := autovacuum_analyze_scale_factor * number of tuples + autovacuum_analyze_threshold
  - Any table with a total number of inserts/deletes/updates exceeding this threshold since last analyze is eligible for an autovacuum analyze.

Some things we must know

- Setting global parameters alone may not be appropriate, all the time.
- Regardless of the table size, if the condition for autovacuum is reached, a table is eligible for autovacuum vacuum or analyze.
  - Consider 2 tables with ten records and a million records.
  - Frequency at which a vacuum or an analyze runs automatically could be greater for the table with just ten records.
  - Use table level autovacuum settings instead.
  - ALTER TABLE foo.bar SET (autovacuum_vacuum_scale_factor = 0, autovacuum_vacuum_threshold = 100);

Some things we must know

- Autovacuum reads block_size pages of a table from disk (default of 8KB), and modifies/writes to the pages containing dead tuples;
- Involves both read and write IO and may be heavy on big tables with huge amount of dead tuples;
- There are other autovacuum parameters like:
  - autovacuum_vacuum_cost_limit
  - autovacuum_vacuum_cost_delay
  - vacuum_cost_page_hit
  - vacuum_cost_page_miss
  - vacuum_cost_page_dirty
  - etc…

Summary is

- If we need to learn one thing from this presentation is:

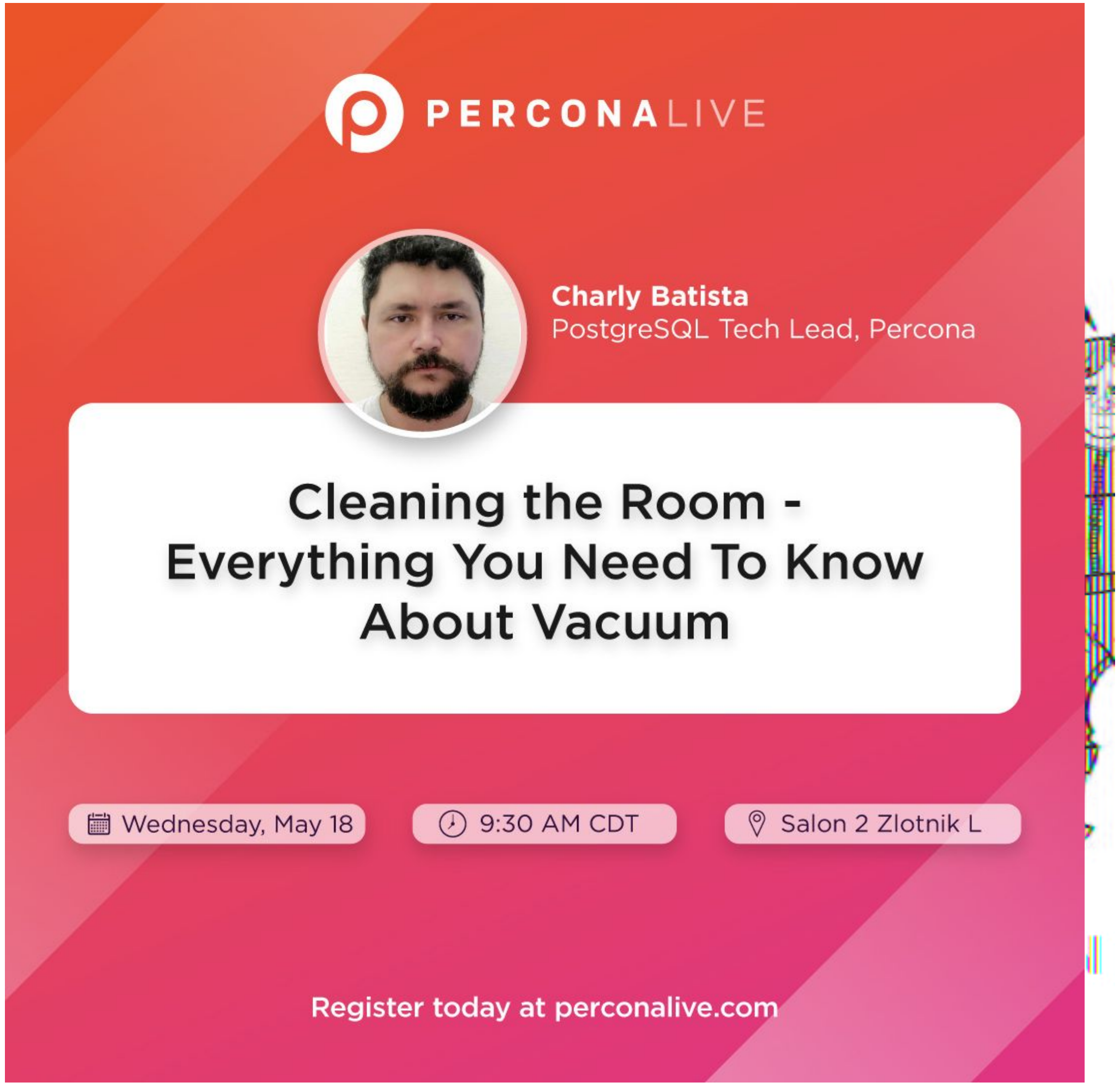
  NEVER DISABLE YOUR AUTOVACUUM
- Transactions are not free, don’t let a transaction open for a long time;
- Transaction IDs are not unlimited, make sure your autovacuum is able to freeze them;
- Vacuum is able to prevent bloating to increase but not able to pack the table;
- Vacuum FULL is the only one able to return disk space back, but it locks the table;
- Sometimes we need to have per-table autovacuum configuration;
- Again, NEVER DISABLE YOUR AUTOVACUUM. If it’s causing problems is because you didn’t understand how it works and you may need to make it more aggressive!!!

Thank you!

Want to learn more about vacuum?

I will be talking about PostgreSQL internals and vacuum at

Percona Live this year!



PERCONA LIVE

Charly Batista
PostgreSQL Tech Lead, Percona

Cleaning the Room - Everything You Need To Know About Vacuum

Wednesday, May 18 | 9:30 AM CDT | Salon 2 Zlotnik L

Register today at perconalive.com

PERCONA LIVE
The biggest open source database conference in the world.
May 16th-18th, 2022 in Austin, Texas

Percona Live 2022 will be May 16-18 at the AT&T Hotel and Conference Center in Austin, Texas, USA.

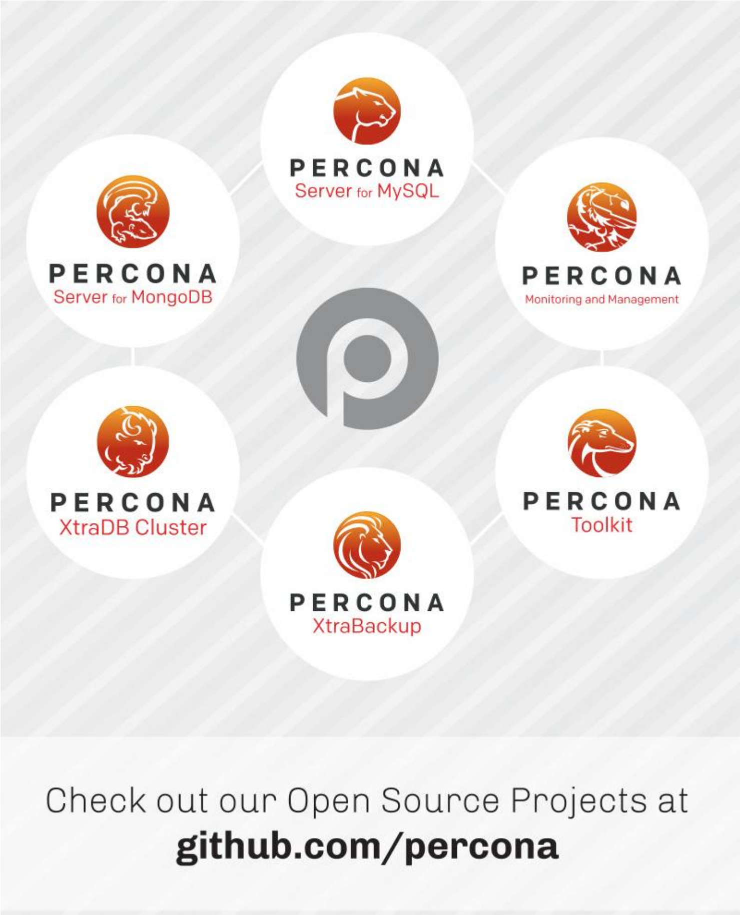
https://www.percona.com/live/conferences

Join in: Percona Community

- **Write for our community blog**
  percona.com/community-blog

- **Join in with our community forums**
  percona.com/forums

- **Contribute to our open source projects**
  github.com/percona



Check out our Open Source Projects at
github.com/percona

We are hiring!

- **We are a remote first company**

- **Some of our current open positions:**
  - C Software Engineer (PostgreSQL)
  - Support Consultant - PostgreSQL
  - PostgreSQL DBA (Remote)
  - Senior Product Manager

You can contact me or check at percona.com/careers for more info. I’m looking forward to hearing from you!