Detailed Understanding of MVCC and Autovacuum Internals in PostgreSQL

MVCC and Autovacuum Internals

Avinash Vallarapu (Avi)

PostgreSQL Support Tech Lead

PGCONF India, 2019

15th Jan, 2019

PERCONA

Topics being discussed today …

- UNDO Management
- Transaction ID’s and PostgreSQL hidden columns
- MVCC and how different is it from other RDBMS’s
- Why Autovacuum ?
- Autovacuum settings
- Tuning Autovacuum

UNDO Management - Oracle and PostgreSQL

- Oracle and MySQL have separate storage for UNDO
  - May be limited space
  - ORA-01555 - Snapshot too Old
  - ORA-30036: unable to extend segment by 8 in undo tablespace
  - Requires no special care to cleanup bloat.
- PostgreSQL
  - Maintains UNDO within a table through versions - old and new row versions.
  - Transaction ID’s are used to identify a version a query can use.
  - A background process to delete old row versions explicitly.
  - No additional writes to a separate UNDO storage in the event of writes.
  - Row locks stored on tuple itself and no separate lock table.

MVCC

- MVCC : Multi-Version Concurrency Control
- Data Consistency
- Prevents viewing Inconsistent data
- Readers and Writers do not block each other
- No Rollback segments for UNDO
- UNDO management is within tables.
- A tuple contains the minimum and maximum transaction ids that are permitted to see it.
- Just like SELECT statements executing WHERE xmin <= txid_current() AND (xmax = 0 OR txid_current() < xmax)

Transaction ID's in PostgreSQL

- Each transaction is allocated a transaction ID (txid).
- txid is a 32-bit unsigned integer
- 4 Billion (4,294,967,296) ID's
  - 2 Billion in the past are visible and
  - 2 Billion in the future are not visible.
- ID's - 0, 1 and 2 are reserved.

  0 - INVALID txid
  1 - Used in initialization of Cluster
  2 - Frozen txid

- txid is circular.

Hidden Columns in PostgreSQL Tables

```
[avi@percona:]$psql -d percona -c "SELECT attname, format_type (atttypid, atttypmod) \
FROM pg_attribute WHERE attrelid::regclass::text='foo.bar' \
ORDER BY attnum"
 attname  |      format_type
----------+-----------------------
 tableoid | oid
 cmax     | cid
 xmax     | xid
 cmin     | cid
 xmin     | xid
 ctid     | tid
 id       | integer
 name     | character varying(20)
(8 rows)
```

Hidden Columns - xmin and xmax

- xmin : Transaction ID that inserted the tuple
- 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.

```
[avi@percona:]$psql -d percona -c "CREATE TABLE foo.bar (id int, name varchar (20) )"
CREATE TABLE
[avi@percona:]$psql -d percona -c "select txid_current()"
 txid_current
--------------
         1603
(1 row)

[avi@percona:]$psql -d percona -c "INSERT INTO foo.bar VALUES (generate_series(1,10), 'avi')"
INSERT 0 10
[avi@percona:]$psql -d percona -c "select xmin, xmax, id, name from foo.bar"
 xmin | xmax | id | name
------+------+----+------
 1604 |    0 |  1 | avi
 1604 |    0 |  2 | avi
 1604 |    0 |  3 | avi
 1604 |    0 |  4 | avi
 1604 |    0 |  5 | avi
 1604 |    0 |  6 | avi
 1604 |    0 |  7 | avi
 1604 |    0 |  8 | avi
 1604 |    0 |  9 | avi
 1604 |    0 | 10 | avi
(10 rows)
```

Extension : pg_freespacemap

- PostgreSQL uses **FSM** to choose the page where a tuple can be Inserted.
- **FSM** stores free space information of each page
- Using the extension **pg_freespacemap**, we can see the freespace available inside each page of a table.

```
percona=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION
[avi@percona:]$ psql -d percona
psql (10.6)
Type "help" for help.

percona=# \x
Expanded display is on.
percona=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
FROM pg_freespace('foo.bar');
-[ RECORD 1 ]---+------
blkno           | 0
avail           | 7776
freespace ratio | 94.00
```

Delete a Record and see what happens ….

Session 1

```
[avi@percona:]$psql -d percona
psql (10.6)
Type "help" for help.

percona=# BEGIN;
BEGIN
percona=# DELETE FROM foo.bar WHERE id = 9;
DELETE 1
percona=#
```

Session 2

```
percona=# BEGIN ;
BEGIN
percona=# select xmin, xmax, id, name from foo.bar;
 xmin | xmax | id | name
------+------+----+------
 1604 |    0 |  1 | avi
 1604 |    0 |  2 | avi
 1604 |    0 |  3 | avi
 1604 |    0 |  4 | avi
 1604 |    0 |  5 | avi
 1604 |    0 |  6 | avi
 1604 |    0 |  7 | avi
 1604 |    0 |  8 | avi
 1604 | 1605 |  9 | avi
 1604 |    0 | 10 | avi
(10 rows)
```

Now COMMIT the DELETE and see …

Session 1

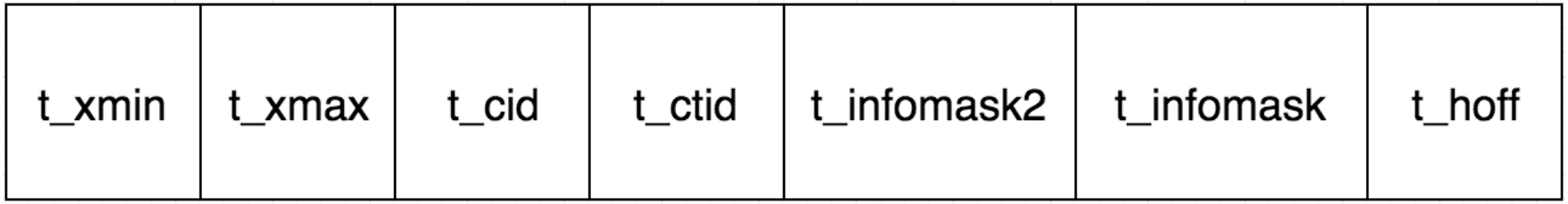
```
[avi@percona:]$psql -d percona
psql (10.6)
Type "help" for help.

percona=# BEGIN;
BEGIN
percona=# DELETE FROM foo.bar WHERE id = 9;
DELETE 1
percona=# COMMIT;
COMMIT
percona=#
```

Session 2

```
percona=# select xmin, xmax, id, name from foo.bar;
 xmin | xmax | id | name
------+------+----+------
 1604 |    0 |  1 | avi
 1604 |    0 |  2 | avi
 1604 |    0 |  3 | avi
 1604 |    0 |  4 | avi
 1604 |    0 |  5 | avi
 1604 |    0 |  6 | avi
 1604 |    0 |  7 | avi
 1604 |    0 |  8 | avi
 1604 |    0 | 10 | avi
(9 rows)
```

Heap Tuples

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



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
```

```
[avi@percona:]$ psql -d percona -c "\dt+ foo.bar"
                    List of relations
 Schema | Name | Type  |  Owner   |    Size    | Description
--------+------+-------+----------+------------+-------------
 foo    | bar  | table | postgres | 8192 bytes |
(1 row)

[avi@percona:]$psql -d percona -c "SELECT t_xmin, t_xmax, t_field3 as t_cid, t_ctid \
> FROM heap_page_items(get_raw_page('foo.bar', 0))"
 t_xmin | t_xmax | t_cid | t_ctid
--------+--------+-------+--------
   1604 |      0 |     0 | (0,1)
   1604 |      0 |     0 | (0,2)
   1604 |      0 |     0 | (0,3)
   1604 |      0 |     0 | (0,4)
   1604 |      0 |     0 | (0,5)
   1604 |      0 |     0 | (0,6)
   1604 |      0 |     0 | (0,7)
   1604 |      0 |     0 | (0,8)
   1604 |   1605 |     0 | (0,9)
   1604 |      0 |     0 | (0,10)
(10 rows)
```

```
percona=# SELECT lp,
       t_ctid AS ctid,
       t_xmin AS xmin,
       t_xmax AS xmax,
       (t_infomask & 128)::boolean AS xmax_is_lock,
       (t_infomask & 1024)::boolean AS xmax_committed,
       (t_infomask & 2048)::boolean AS xmax_rolled_back,
       (t_infomask & 4096)::boolean AS xmax_multixact,
       t_attrs[1] AS p_id,
       t_attrs[2] AS p_val
FROM heap_page_item_attrs(
        get_raw_page('foo.bar', 0),
        'foo.bar'
     ) where lp = 9;
-[ RECORD 1 ]----+-----------
lp               | 9
ctid             | (0,9)
xmin             | 1604
xmax             | 1605
xmax_is_lock     | f
xmax_committed   | t
xmax_rolled_back | f
xmax_multixact   | f
p_id             | \x09000000
p_val            | \x09617669
```

Delete a Record and Rollback ….

```
percona=# BEGIN;
BEGIN
percona=# DELETE FROM foo.bar WHERE id = 6;
DELETE 1
percona=# ROLLBACK;
ROLLBACK
```

Perform a select that sets the hint bits, after reading the commit log.
It is an IO in fact :(

```
percona=# select * from foo.bar where id = 6;
 id | name
----+------
  6 | avi
(1 row)
```

```
percona=# SELECT lp,
       t_ctid AS ctid,
       t_xmin AS xmin,
       t_xmax AS xmax,
       (t_infomask & 128)::boolean AS xmax_is_lock,
       (t_infomask & 1024)::boolean AS xmax_committed,
       (t_infomask & 2048)::boolean AS xmax_rolled_back,
       (t_infomask & 4096)::boolean AS xmax_multixact,
       t_attrs[1] AS p_id,
       t_attrs[2] AS p_val
FROM heap_page_item_attrs(
        get_raw_page('foo.bar', 0),
        'foo.bar'
     ) where lp = 6;
-[ RECORD 1 ]----+-----------
lp               | 6
ctid             | (0,6)
xmin             | 1604
xmax             | 1606
xmax_is_lock     | f
xmax_committed   | f
xmax_rolled_back | t
xmax_multixact   | f
p_id             | \x06000000
p_val            | \x09617669
```

- Just like SELECT statements executing
  WHERE xmin <= txid_current() AND (xmax = 0 OR txid_current() < xmax)

- The above statement must be understandable by now ….

Space occupied by the DELETED tuple ?

VACUUM / AUTOVACUUM

- **Live Tuples** : Tuples that are Inserted or up-to-date or can be read or modified.
- **Dead Tuples** : Tuples that are changed (Updated/Deleted) and unavailable to be used for any future transactions.
- Continuous transactions may lead to a number of dead rows. A lot of space can be rather re-used by future transactions.
- **VACUUM** in PostgreSQL would cleanup the dead tuples and mark it to free space map.
- transaction ID (**xmax**) of the deleting transaction must be older than the oldest transaction still active in PostgreSQL Server for vacuum to delete that tuple.
- **Autovacuum** in PostgreSQL automatically runs VACUUM on tables as a background process.
- Autovacuum is also responsible to run **ANALYZE** that updates the statistics of a Table.

Background Processes in PostgreSQL

```
[avi@percona:]$ps -eaf | grep postgres
postgres 13532     1  0 Feb12 ?        00:00:00 /usr/pgsql-10/bin/postgres -D /var/lib/pgsql/10/data
postgres 13533 13532  0 Feb12 ?        00:00:00 postgres: logger process
postgres 13535 13532  0 Feb12 ?        00:00:00 postgres: checkpointer process
postgres 13536 13532  0 Feb12 ?        00:00:00 postgres: writer process
postgres 13537 13532  0 Feb12 ?        00:00:00 postgres: wal writer process
postgres 13538 13532  0 Feb12 ?        00:00:00 postgres: autovacuum launcher process
postgres 13539 13532  0 Feb12 ?        00:00:00 postgres: archiver process
postgres 13540 13532  0 Feb12 ?        00:00:01 postgres: stats collector process
postgres 13541 13532  0 Feb12 ?        00:00:00 postgres: bgworker: logical replication launcher
```

Let us run a VACUUM and see now …

```
[avi@percona:]$psql -d percona -c "VACUUM foo.bar"
VACUUM

[avi@percona:]$psql -d percona -c "SELECT t_xmin, t_xmax, t_field3 as t_cid, t_ctid \
FROM heap_page_items(get_raw_page('foo.bar', 0))"
 t_xmin | t_xmax | t_cid | t_ctid
--------+--------+-------+--------
   1604 |      0 |     0 | (0,1)
   1604 |      0 |     0 | (0,2)
   1604 |      0 |     0 | (0,3)
   1604 |      0 |     0 | (0,4)
   1604 |      0 |     0 | (0,5)
   1604 |   1606 |     0 | (0,6)
   1604 |      0 |     0 | (0,7)
   1604 |      0 |     0 | (0,8)
        |        |       |
   1604 |      0 |     0 | (0,10)
(10 rows)
```

Does it show some extra free space in the page now ???

Use pg_freespacemap again …

```
percona=# \x
Expanded display is on.
percona=#
percona=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
FROM pg_freespace('foo.bar');
-[ RECORD 1 ]---+------
blkno           | 0
avail           | 7808
freespace ratio | 95.00
```

When does Autovacuum run ??

- To start **autovacuum**, you must have the parameter **autovacuum** set to ON.
- Background Process : **Stats Collector** tracks the usage and activity information.
- PostgreSQL identifies the tables needing vacuum or analyze depending on certain parameters.
- Parameters needed to enable autovacuum in PostgreSQL are :
  autovacuum = on # (ON by default)
  track_counts = on # (ON by default)
- An automatic vacuum or analyze runs on a table depending on a certain mathematic equations.

- **Autovacuum VACUUM**
  - 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 vacuum.
- **Autovacuum ANALYZE**
  - 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.

- **autovacuum_vacuum_scale_factor** or **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** or **autovacuum_analyze_threshold** : Minimum number of obsolete records or dml's needed to trigger an autovacuum.

- Let's consider a table: foo.bar with 1000 records and the following autovacuum parameters.

  autovacuum_vacuum_scale_factor = 0.2
  autovacuum_vacuum_threshold = 50
  autovacuum_analyze_scale_factor = 0.1
  autovacuum_analyze_threshold = 50

- Table : foo.bar becomes a candidate for autovacuum VACUUM when,
  Total number of Obsolete records = (0.2 * 1000) + 50 = 250

- Table : foo.bar becomes a candidate for autovacuum ANALYZE when,
  Total number of Inserts/Deletes/Updates = (0.1 * 1000) + 50 = 150

Tuning Autovacuum in PostgreSQL

- 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);
```

- There cannot be more then **autovacuum_max_workers** number of auto vacuum processes running at a time. Default is 3.
- Each autovacuum runs with a gap of **autovacuum_naptime**, default is 1 min.

Can i increase autovacuum_max_workers ?
Is VACUUM IO Intensive ?????

- Autovacuum reads 8KB (default **block_size**) pages of a table from disk 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.

- Autovacuum IO Parameters :

  autovacuum_vacuum_cost_limit : total cost limit autovacuum could reach (combined by all autovacuum jobs).

  autovacuum_vacuum_cost_delay : autovacuum will sleep for these many milliseconds when a cleanup reaching *autovacuum_vacuum_cost_limit* cost is done.

  vacuum_cost_page_hit : Cost of reading a page that is already in shared buffers and doesn't need a disk read.

  vacuum_cost_page_miss : Cost of fetching a page that is not in shared buffers.

  vacuum_cost_page_dirty : Cost of writing to each page when dead tuples are found in it.

- Default Values for the Autovacuum IO parameters
  ————————————————————————————————

  autovacuum_vacuum_cost_limit = -1 (Defaults to vacuum_cost_limit) = 200
  autovacuum_vacuum_cost_delay = 20ms
  vacuum_cost_page_hit = 1
  vacuum_cost_page_miss = 10
  vacuum_cost_page_dirty = 20

- Let’s imagine what can happen in 1 second. (1 second = 1000 milliseconds)

- In a best case scenario where read latency is 0 milliseconds, autovacuum can wake up and go for sleep 50 times (1000 milliseconds / 20 ms) because the delay between wake-ups needs to be 20 milliseconds.

  1 second = 1000 milliseconds = 50 * **autovacuum_vacuum_cost_delay**

- **Read IO limitations with default parameters**
  - If all the pages with dead tuples are found in shared buffers, in every wake up 200 pages can be read. Cost associated per reading a page in shared_buffers is 1.

    So, in 1 second, (50 * 200/vacuum_cost_page_hit * 8 KB) = **78.13 MB** can be read by autovacuum.

  - If the pages are not in shared buffers and need to fetched from disk, an autovacuum can read : 50 * ((200 / vacuum_cost_page_miss) * 8) KB = **7.81 MB** per second.
- **Write IO limitations with default parameters**
  - To delete dead tuples from a page/block, the cost of a write operation is : vacuum_cost_page_dirty, set to 20 by default.
  - At the most, an autovacuum can write/dirty : 50 * ((200 / vacuum_cost_page_dirty) * 8) KB = **3.9 MB** per second.

Transaction ID Wraparound

- Transaction with txid := n, inserted a record.
  t_xmin := n
- After some time, we are now at a txid := (2.1 billion + n)
  Tuple is visible to a SELECT now.
- Now let us say that the txid is := (2.1 billion + n + 1). The same SELECT fails as the txid := n is now considered to be the past.
- This is usually referred to as : Transaction ID Wraparpound in PostgreSQL.
- Vacuum in PostgreSQL re-writes the t_xmin to the frozen txid when the t_xmin is older than (current txid - vacuum_freeze_min_age)
- Until 9.3, xmin used to be updated with an invalid and visible txid : 3, upon FREEZE.
- Starting from 9.4, the XMIN_FROZEN bit is set to the t_infomask field of tuples and avoids re-writing the tuples.

Best Strategy

- Do not just add more autovacuum workers. See if you are fine for more IO caused by autovacuum and tune all the IO settings.
- Busy OLTP systems require your thorough supervision for automation of manual vacuum.
- Perform routine manual vacuum in low peak or non-business hours to ensure a less bloated database at all times.
- A database with finely tuned autovacuum settings and routine maintenance tasks is always healthy.

Champions of Unbiased
Open Source Database Solutions

Questions ??