Vacuum

Last updated by | Lisa Liu | Nov 6, 2020 at 10:35 AM PST

Why VACUUM?

Whenever rows in a PostgreSQL table are updated or deleted, dead rows are left behind. VACUUM gets rid of them so that the space can be reused. If a table doesn't get vacuumed, it will get bloated, which wastes disk space and slows down sequential table scans (and – to a smaller extent – index scans).

VACUUM also takes care of freezing table rows so to avoid problems when the transaction ID counter wraps around, but that's a different story (we will talk later)

Normally you don't have to take care of all that, because the autovacuum daemon built into PostgreSQL does that for you.

The problem:

If your tables get bloated, the first thing you check is whether autovacuum has processed them or not:

```
SELECT schemaname, relname, n_live_tup, n_dead_tup, last_autovacuum
FROM pg_stat_all_tables
ORDER BY n_dead_tup
   / (n_live_tup
      * current_setting('autovacuum_vacuum_scale_factor')::float8
        + current_setting('autovacuum_vacuum_threshold')::float8)
    DESC
LIMIT 10;
```

If your bloated table does not show up here, n_dead_tup is zero and last_autovacuum is NULL, you might have a problem with the statistics collector.

If the bloated table is right there on top, but last_autovacuum is NULL, you might need to configure autovacuum to be more aggressive so that it gets done with the table.

But sometimes the result will look like this:

```
 schemaname |    relname    | n_live_tup | n_dead_tup |    last_autovacuum
------------+---------------+------------+------------+-------------------
 laurenz    | vacme         |      50000 |      50000 | 2018-02-22 13:20:16
 pg_catalog | pg_attribute  |         42 |        165 |
 pg_catalog | pg_amop       |        871 |        162 |
 pg_catalog | pg_class      |          9 |         31 |
 pg_catalog | pg_type       |         17 |         27 |
 pg_catalog | pg_index      |          5 |         15 |
 pg_catalog | pg_depend     |       9162 |        471 |
 pg_catalog | pg_trigger    |          0 |         12 |
 pg_catalog | pg_proc       |        183 |         16 |
 pg_catalog | pg_shdepend   |          7 |          6 |
(10 rows)
```

Why won't VACUUM remove the dead rows?

VACUUM can only remove those row versions (also known as "tuples") that are not needed any more. A tuple is not needed if the transaction ID of the deleting transaction (as stored in the xmax system column) is older than the oldest transaction still active in the PostgreSQL database (or the whole cluster for shared tables).

This value (22300 in the VACUUM output above) is called the "xmin horizon".

There are three things that can hold back this xmin horizon in a PostgreSQL cluster:

Long-running transactions.

Abandoned replication slots.

Orphaned prepared transactions.

Long-running transactions.

You can find those and their xmin value with the following query:

```
SELECT  pid, datname, usename, state, backend_xmin
FROM  pg_stat_activity
WHERE  backend_xmin IS  NOT  NULL
ORDER  BY  age(backend_xmin) DESC;
```

You can use the pg_terminate_backend() function to terminate the database session that is blocking your VACUUM.

Abandoned replication slots.

A replication slot is a data structure that keeps the PostgreSQL server from discarding information that is still needed by a standby server to catch up with the primary.

If replication is delayed or the standby server is down, the replication slot will prevent VACUUM from deleting old rows.

You can find all replication slots and their xmin value with this query:

```
SELECT  slot_name, slot_type, database, xmin
FROM  pg_replication_slots
ORDER  BY  age(xmin) DESC;
```

Orphaned prepared transactions.

During two-phase commit, a distributed transaction is first prepared with the PREPARE statement and then committed with the COMMIT PREPARED statement.

Once a transaction has been prepared, it is kept “hanging around” until it is committed or aborted. It even has to survive a server restart! Normally, transactions don’t remain in the prepared state for long, but sometimes things go wrong and a prepared transaction has to be removed manually by an administrator.

You can find all prepared transactions and their xmin value with the following query:

```
SELECT  gid, prepared, owner, database, transaction  AS  xmin
FROM  pg_prepared_xacts
ORDER  BY  age(transaction) DESC;
```

Use the ROLLBACK PREPARED SQL statement to remove prepared transactions.

https://www.postgresql.org/docs/11/sql-rollback-prepared.html

How good have you found this content?