PostgreSQL Checking Autovacuum

Last updated by | Hamza Aqel | Jan 25, 2022 at 2:03 AM PST

Please don't modify or move as this is part of GT , please contact haaqel@microsoft.com if needed

Problem Statement: Why my dead tuples are not cleaned, is Auto Vacuum running at all?

We get this request often from our Customers and will try to list some of the common causes for such incidents

1. There are too many dead tuples in my z, why is the Auto Vacuum not cleaning them?
2. I see a big bloat in my database, isn't vacuum supposed to clean?
3. My query plans are bad, why isn't vacuum updating stats?
4. Are we seeing XID wraparound warning and transactions failing?

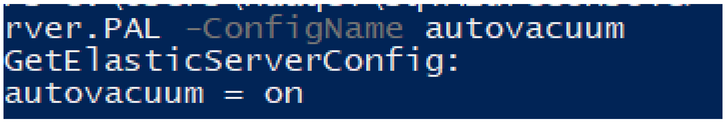
This is not a comprehensive guide to solve all vacuum issues, but a good place to start. Here are the common debugging steps.

1. Is vacuum running? Check if auto vacuum daemon is running at all, default is "ON", you can check at table level or instance level using a CAS command to check if the parameter is ON :
   for table level settings , checkout the column reloptions in pg_class:

   PSQL> select relname,reloptions from pg_class ;

   for instance level:

   Get-ElasticServerConfig -SubscriptionId xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx -ResourceGroup ResGrpName -ServerName Srvname -ServerType PostgreSQL.Server.PAL -ConfigName autovacuum



```
rver.PAL -ConfigName autovacuum
GetElasticServerConfig:
autovacuum = on
```

```
Psql>
SELECT * FROM pg_stat_activity where query like '%vacuum%';

You should see a record, such as, below row

-[ RECORD 1 ]----+-----------------------------------------------------------------
datid            |
datname          |
pid              | 100
usesysid         |
usename          |
application_name |
client_addr      |
client_hostname  |
client_port      |
backend_start    | 2019-06-17 17:13:19.313748+00
xact_start       |
query_start      |
state_change     |
wait_event_type  | Activity
wait_event       | AutoVacuumMain
```

```
state            |
backend_xid      |
backend_xmin     |
query            |
backend_type     | autovacuum launcher
```

- Another approach: Check sandbox logs using kusto table (MonRdmsPgSqlSandbox)
- The above approach is accurate a nd good but this is simple kusto query - so easy and reliable.

  sandob logs: 2021-10-31 06:03:53 UTC-617de9bd.9c-LOG: autovacuum :
  database ""stored_value_coupons"", frozen xid: 148606136, multixact id: 1, last autovacuum time: 2021-10-31 06:03:36.774231+00

2) If you see the vacuum running in step (1), we can check the progress of auto vacuum

```
Psql> select * from pg_stat_progress_vacuum;


-[ RECORD 1 ]-------+--------------
pid                 | 104701
datid               | 16419
datname             | analytics
relid               | 1911284001
phase               | scanning heap
heap_blks_total     | 155738368
heap_blks_scanned   | 75619358
heap_blks_vacuumed  | 74784601
index_vacuum_count  | 6
max_dead_tuples     | 178956970
num_dead_tuples     | 32346941
```

Or an improviser for better format

```
SELECT
    p.pid,
    now() - a.xact_start AS duration,
    coalesce(wait_event_type ||'.'|| wait_event, 'f') AS waiting,
    CASE
        WHEN a.query ~ '^autovacuum.*to prevent wraparound' THEN 'wraparound'
        WHEN a.query ~ '^vacuum' THEN 'user'
        ELSE 'regular'
    END AS mode,
    p.datname AS database,
    p.relid::regclass AS table,
    p.phase,
    pg_size_pretty(p.heap_blks_total * current_setting('block_size')::int) AS
table_size,
    pg_size_pretty(pg_total_relation_size(relid)) AS total_size,
    pg_size_pretty(p.heap_blks_scanned * current_setting('block_size')::int) AS
scanned,
    pg_size_pretty(p.heap_blks_vacuumed * current_setting('block_size')::int) AS
vacuumed,
    round(100.0 * p.heap_blks_scanned / p.heap_blks_total, 1) AS scanned_pct,
    round(100.0 * p.heap_blks_vacuumed / p.heap_blks_total, 1) AS vacuumed_pct,
    p.index_vacuum_count,
    round(100.0 * p.num_dead_tuples / p.max_dead_tuples,1) AS dead_pct
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a using (pid)
ORDER BY now() - a.xact_start DESC;
```

Sample output

```
-[ RECORD 1 ]-------+------------------
pid                 | 104701
duration            | 03:21:51.330818
waiting             | f
mode                | regular
database            | analytics
table               | events
phase               | vacuuming indexes
table_size          | 1188 GB
total_size          | 1682 GB
scanned             | 601 GB
vacuumed            | 571 GB
scanned_pct         | 50.0
vacuumed_pct        | 48.0
index_vacuum_count  | 6
dead_tup_pct        | 100.0
```

Check the phase which it's stuck in, it will offer clues where the auto vacuum is spending most of its time and the possible causes.

3. Check the history of AutoVacuum using

```
select relid , schemaname , relname , seq_scan , seq_tup_read , last_vacuum ,
last_autovacuum , last_analyze, last_autoanalyze from  pg_stat_user_tables
```

4. In some (or most of the customer) scenarios, we have Auto Vacuum running and progressing fine but the table's dead tuples are
Not cleaned up and/or bloat is increasing.

```
Psql> SELECT * FROM pg_stat_all_tables where relname = '<>';
-[ RECORD 1 ]-------+------------------------------
relid               | 992159
schemaname          | public
relname             | contacts__fulltext
seq_scan            | 0
seq_tup_read        | 0
idx_scan            | 649164
idx_tup_fetch       | 442809300
n_tup_ins           | 0
n_tup_upd           | 0
n_tup_del           | 1816244
n_tup_hot_upd       | 0
n_live_tup          | 410668555
n_dead_tup          | 9615572
n_mod_since_analyze | 18
last_vacuum         |
last_autovacuum     |
last_analyze        | 2019-07-01 20:10:49.537791+00
last_autoanalyze    |
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 1
autoanalyze_count   | 0
```

You can see live (currently visible) and dead (will not be seen by anyone) tuples count. One of the reason why vacuum daemon not cleaning up the table could is autovacuum_vacuum_scale_factor, which specifies the fraction of the table size when deciding whether to trigger a VACUUM. In this example, autovacuum_vacuum_scale_factor is set to 0.05, the engine calculates the scale factor as 9615572 ÷ (410668555 + 9615572) = 0.0228 which is smaller than 0.05, and that's why vacuum is not triggered for this table

3. There is also a minimum value to be met for autovacuum_vacuum_threshold when deciding whether to trigger a vacuum, this is in addition to the above calculation, this is to prevent excessive vacuuming i.e. if the threshold prevents vacuum for low values, such as 1 row or 10 rows, but if the threshold is set too high, such as 1 Million, vacuum will **not** trigger until we have a million dead rows.

The condition for auto vacuum to trigger on table is dead_tuples >= table_size * scale_factor + threshold. Based on (3) and (4) please tune
the config parameters for the vacuum to take effect.

5. In some cases, where an immediate mitigation is needed, you can issue vacuum manually.

```
Psql> VACUUM ANALYZE <table>
```

6. Scenarios where we have tuples not cleaned up, and still holding up Transaction Id preventing the XID wrap around. This manifests as severe downtime as no new connections are accepted by the engine.

```
Psql> select  pg_database.datminmxid; <-- Find the database with the smallest value

Psql-to-above-db> select pg_class.relminmxid; <-- To find the culprit table
```

Check if there is still any open transaction(neither committed nor rolled back) that might be preventing the cleaning.

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

As a mitigation, close the open transaction and also manually unfreeze the xid.

```
Psql> VACUUM FREEZE <table>; <-- This should advance the pg_class.relminmxid of the table
```

Other Possible Scenarios where the vacuum is not running in addition to the above:

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;
```