# [TSG-PGSS-PERF]Performance_degradation_caused_by_bloat

Last updated by | Shawn Xiao | Aug 10, 2021 at 4:58 PM PDT

## Issue Description and Background

Sometimes, the customer may discover that the PostgreSQL has been running slower and slower but with no resource limit hit or workload changed. It could be a general slowness that impacts the whole server or could affect only certain queries in certain tables.
This TSG can help you investigate and define the possible cause of such behavior: **data bloat**. To investigate and understand the issue, I will explain the detail in below topics:

- How to verify and mitigate the issue
- How to understand and explain the behavior:
  - what is data bloat and how does it happen?
  - what does VACUUM do and what are the differences between Auto VACUUM and VACUUM FULL?

## How to verify and mitigate the issue?

1. below query can used in a PostgreSQL server to check the table bloat. Please run this query in a off-business hour because it is an IO intensive query that could impact server performance.

```
-- new table bloat query
-- still needs work; is often off by +/- 20%
WITH constants AS (
    -- define some constants for sizes of things
    -- for reference down the query and easy maintenance
    SELECT current_setting('block_size')::numeric AS bs, 23 AS hdr, 8 AS ma
),
no_stats AS (
    -- screen out table who have attributes
    -- which dont have stats, such as JSON
    SELECT table_schema, table_name,
        n_live_tup::numeric as est_rows,
        pg_table_size(relid)::numeric as table_size
    FROM information_schema.columns
        JOIN pg_stat_user_tables as psut
        ON table_schema = psut.schemaname
        AND table_name = psut.relname
        LEFT OUTER JOIN pg_stats
        ON table_schema = pg_stats.schemaname
            AND table_name = pg_stats.tablename
            AND column_name = attname
    WHERE attname IS NULL
        AND table_schema NOT IN ('pg_catalog', 'information_schema')
    GROUP BY table_schema, table_name, relid, n_live_tup
),
null_headers AS (
    -- calculate null header sizes
    -- omitting tables which dont have complete stats
    -- and attributes which aren't visible
    SELECT
        hdr+1+(sum(case when null_frac <> 0 THEN 1 else 0 END)/8) as nullhdr,
        SUM((1-null_frac)*avg_width) as datawidth,
        MAX(null_frac) as maxfracsum,
        schemaname,
        tablename,
        hdr, ma, bs
    FROM pg_stats CROSS JOIN constants
        LEFT OUTER JOIN no_stats
            ON schemaname = no_stats.table_schema
            AND tablename = no_stats.table_name
    WHERE schemaname NOT IN ('pg_catalog', 'information_schema')
        AND no_stats.table_name IS NULL
        AND EXISTS ( SELECT 1
            FROM information_schema.columns
                WHERE schemaname = columns.table_schema
                    AND tablename = columns.table_name )
    GROUP BY schemaname, tablename, hdr, ma, bs
),
data_headers AS (
    -- estimate header and row size
    SELECT
        ma, bs, hdr, schemaname, tablename,
        (datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%ma END)))::numeric AS datahdr,
        (maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma ELSE nullhdr%ma END))) AS nullhdr2
    FROM null_headers
),
table_estimates AS (
    -- make estimates of how large the table should be
    -- based on row and page size
    SELECT schemaname, tablename, bs,
        reltuples::numeric as est_rows, relpages * bs as table_bytes,
    CEIL((reltuples*
            (datahdr + nullhdr2 + 4 + ma -
                (CASE WHEN datahdr%ma=0
                    THEN ma ELSE datahdr%ma END)
                )/(bs-20))) * bs AS expected_bytes,
        reltoastrelid
    FROM data_headers
```

```
        JOIN pg_class ON tablename = relname
        JOIN pg_namespace ON relnamespace = pg_namespace.oid
            AND schemaname = nspname
    WHERE pg_class.relkind = 'r'
),
estimates_with_toast AS (
    -- add in estimated TOAST table sizes
    -- estimate based on 4 toast tuples per page because we dont have
    -- anything better.  also append the no_data tables
    SELECT schemaname, tablename,
        TRUE as can_estimate,
        est_rows,
        table_bytes + ( coalesce(toast.relpages, 0) * bs ) as table_bytes,
        expected_bytes + ( ceil( coalesce(toast.reltuples, 0) / 4 ) * bs ) as expected_bytes
    FROM table_estimates LEFT OUTER JOIN pg_class as toast
        ON table_estimates.reltoastrelid = toast.oid
            AND toast.relkind = 't'
),
table_estimates_plus AS (
-- add some extra metadata to the table data
-- and calculations to be reused
-- including whether we cant estimate it
-- or whether we think it might be compressed
    SELECT current_database() as databasename,
            schemaname, tablename, can_estimate,
            est_rows,
            CASE WHEN table_bytes > 0
                THEN table_bytes::NUMERIC
                ELSE NULL::NUMERIC END
                AS table_bytes,
            CASE WHEN expected_bytes > 0
                THEN expected_bytes::NUMERIC
                ELSE NULL::NUMERIC END
                    AS expected_bytes,
            CASE WHEN expected_bytes > 0 AND table_bytes > 0
                AND expected_bytes <= table_bytes
                THEN (table_bytes - expected_bytes)::NUMERIC
                ELSE 0::NUMERIC END AS bloat_bytes
    FROM estimates_with_toast
    UNION ALL
    SELECT current_database() as databasename,
        table_schema, table_name, FALSE,
        est_rows, table_size,
        NULL::NUMERIC, NULL::NUMERIC
    FROM no_stats
),
bloat_data AS (
    -- do final math calculations and formatting
    select current_database() as databasename,
        schemaname, tablename, can_estimate,
        table_bytes, round(table_bytes/(1024^2)::NUMERIC,3) as table_mb,
        expected_bytes, round(expected_bytes/(1024^2)::NUMERIC,3) as expected_mb,
        round(bloat_bytes*100/table_bytes) as pct_bloat,
        round(bloat_bytes/(1024::NUMERIC^2),2) as mb_bloat,
        table_bytes, expected_bytes, est_rows
    FROM table_estimates_plus
)
-- filter output for bloated tables
SELECT databasename, schemaname, tablename,
    can_estimate,
    est_rows,
    pct_bloat, mb_bloat,
    table_mb
FROM bloat_data
-- this where clause defines which tables actually appear
-- in the bloat chart
-- example below filters for tables which are either 50%
-- bloated and more than 20mb in size, or more than 25%
-- bloated and more than 4GB in size
WHERE ( pct_bloat >= 50 AND mb_bloat >= 10 )
```

```
    OR ( pct_bloat >= 25 AND mb_bloat >= 1000 )
ORDER BY pct_bloat DESC;
```

2. In the returned output if any, taking an example like below, the columns *pct_bloat* and *mb_bloat* indicate the two listed table suffer from a high data bloat. And depends on the physical size of the two tables and frequency of two tables' usage, performance could be greatly impacted

```
databasename   | schemaname |        tablename         | can_estimate | est_rows | pct_bloat | mb_bloat |
---------------+------------+--------------------------+--------------+----------+-----------+----------+
factory        | public     | wkstation                | t            |    23400 |        81 |    14.24 |
fctry          | public     | evt                      | t            |   552139 |        72 |  1515.80 |
(2 rows)
```

3. Run below query to check the table auto vacuum information to ensure the vacuum has been running periodically as expected.

```
with tmp as (
select
    s.schemaname,
    s.relname,
    s.n_live_tup * current_setting('autovacuum_vacuum_scale_factor')::decimal + current_setting('autovacu
    from pg_stat_user_tables s
) select
u.relname,
n_dead_tup,
tmp.threshold as vacuum_threshold, n_dead_tup>tmp.threshold as autovacuum_needed,
ROUND( 100* (n_dead_tup/tmp.threshold)) as pctg_til_autovacuum,
last_autovacuum,
autovacuum_count,
last_autoanalyze,
autoanalyze_count from pg_catalog.pg_stat_user_tables u
join tmp using (schemaname, relname)
order by pctg_til_autovacuum desc;
```

The returned output should look similar as the following where list out the last time auto vacuum was performed. Please note that it is very possible that, though auto vacuum was scheduled successfully, large size of bloat data stil exist. I will explain in the next section. You can refer to TSG at https://supportability.visualstudio.com/AzureDBPostgreSQL/_wiki/wikis/AzureDBPostgreSQL/289205/Autovacuum to check if Auto Vacuum has been running as expected.

| relname | n_dead_tup | vacuum_threshold | autovacuum_needed | pctg_til_autovacuum | last_autovacuum | autovacuum_count | last_autoanalyze | autoanalyze_count |
|---|---|---|---|---|---|---|---|---|
| chart | 45 | 57.35 | f | 78 | 2020-12-07 03:31:02.30383+00 | 1 | 2021-01-07 06:48:35.823114+00 | 3 |
| wokstation | 881 | 1221.80 | f | 72 | 2021-01-26 02:17:46.312831+00 | 130 | 2021-01-27 00:43:21.38182+00 | 907 |
| dvc | 37 | 53.05 | f | 70 | 2021-01-27 06:57:12.35903+00 | 8740 | 2021-01-27 06:56:57.265467+00 | 26255 |
| usr | 37 | 53.35 | f | 69 | | 0 | 2021-01-09 04:04:37.475845+00 | 1 |
| client | 29 | 50.60 | f | 57 | | 0 | 2021-01-12 13:41:02.086978+00 | 1 |
| summary | 39 | 72.50 | f | 54 | 2021-01-27 09:31:01.384565+00 | 170 | 2021-01-27 09:40:50.366348+00 | 4593 |
| metrics | 44 | 92.40 | f | 48 | 2021-01-26 15:31:01.44555+00 | 522 | 2021-01-27 01:00:41.505308+00 | 4548 |
| cmpnt | 26 | 59.75 | f | 44 | | 0 | 2021-01-12 03:14:06.813098+00 | 2 |
| total | 34 | 81.05 | f | 42 | | 0 | 2021-01-27 08:14:52.49304+00 | 935 |
| code | 22 | 55.90 | f | 39 | 2021-01-12 13:40:28.318663+00 | 1 | 2021-01-15 04:54:38.286498+00 | 4 |
| usage | 18 | 50.00 | f | 36 | 2021-01-27 04:00:31.685973+00 | 2879 | 2021-01-27 04:00:31.717202+00 | 8713 |
| chart | 14 | 57.35 | f | 24 | | 0 | 2021-01-08 02:28:08.972908+00 | 2 |
| audit | 36 | 194.25 | f | 19 | | 0 | 2021-01-22 08:45:41.430233+00 | 9 |
| skill | 7 | 52.10 | f | 13 | | 0 | 2021-01-13 02:50:12.958765+00 | 1 |
| fork1 | 1 | 67.25 | f | 1 | | 0 | 2020-12-25 02:57:17.844544+00 | 5 |
| event | 0 | 4088.10 | f | 0 | | 0 | 2021-01-25 12:02:37.971442+00 | 10 |
| reliability | 0 | 50.00 | f | 0 | 2021-01-27 05:00:33.950382+00 | 2862 | 2021-01-27 05:00:33.997253+00 | 8860 |
| alarm | 0 | 84.50 | f | 0 | | 0 | 2020-12-27 02:02:59.380121+00 | 10 |
| evnt | 3 | 27668.15 | f | 0 | 2021-01-26 02:18:12.861803+00 | 9 | 2021-01-26 08:17:23.931656+00 | 31 |
| availability | 0 | 50.00 | f | 0 | 2021-01-27 05:00:34.200382+00 | 3144 | 2021-01-27 05:00:34.231647+00 | 8794 |
| activity | 1 | 3924.15 | f | 0 | | 0 | 2021-01-25 07:07:07.106707+00 | 11 |
| quality | 0 | 60.65 | f | 0 | | 0 | 2021-01-22 08:05:05.615443+00 | 3 |

(44 rows)

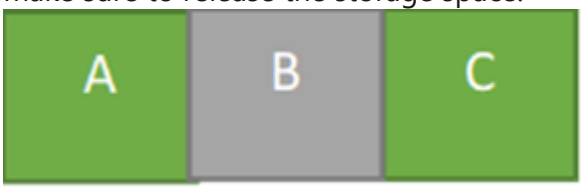
4. Once confirmed data bloat, customer can simply execute *VACUUM FULL* to clean the bloat. **Please avoid calling this query in a busy hour because it will lock the vacuuming table and so causing server**

**unresponsive**. The performance should be improved upon completion.

## Explanation

- What is bloat data?
  In PostgreSQL, when updating or deleting a row, it does not change or drop the record for the row but creates a new row and marks old row as unused. Thus, the marked unused record will still exist and consumes disk space. And the record like this is named as dead tuples and will need be cleaned to ensure the consumed space can be reclaimed. For a busy server with frequent transactions, there could generate a lot of dead tuples and consume storage spaces, which is called data bloat.

- How to deal with dead tuples?
  VACUUM is introduced in PostgreSQL and one of the main tasks VACUUM does is to clean those dead tuples. VACUUM will mark those unused records as reusable so that the new created record can still be stored in the space used by dead tuples.
  For example, below illustrated are three records as A, B, and C. When B is deleted, the record of B will be dead tuples and will still there and taken some storage space. And VCUUM will step in to clean the B to make sure to release the storage space.




- Why VACUUM FULL is still needed even enabled Auto Vacuum? What are the differences between Auto VACUUM and VACUUM FULL?
  VACUUM:
  - It will mark the 'space' used by dead tuples as reusable.
  - The performance impact of VACUUM is minimum so it can be scheduled as auto vacuum that runs periodically.
  - The efficiency is low because it does not physically clean the dead tuples. Taken the example of the above records of A, B, and C, if the size of B (dead tuples) is 5kb, after auto-vacuum, the new created record less than 5kb can reuse the space consumed by B but the new created record large than 5kb will still generate a new record.

  VACUUM FULL:
  - It will physically remove the dead tuples and reclaim the storage space.
  - The performance impact is huge as it will lock the vacuumed tables.
  - The efficiency is high because it physically cleans the dead tuples in disk.
- Why dead tuples and data bloat will impact performance?
  When a query is executing, PostgreSQL optimizer will generate query plan based on statistics. And, since statistics are generated based on data pages and index pages scanned, record spaces consumed by dead tuples and bloat data will impact the result of statistics. Therefore, the optimizer may generate a bad query plan that make query executing slow.

---

If you have any doubts when following this page, please reach out to *xixia* for clarification and wiki/TSG improvement.