Google Cloud

Optimizing, monitoring, and troubleshooting `VACUUM` operations in PostgreSQL

Amarnadh Sai Eluri
Software Engineer
June 20, 2020

Introduction

This document describes the fundamentals of the `VACUUM` operation in PostgreSQL databases. It describes the mechanisms to monitor and tune the database engine that maintains the health of database instances.

PostgreSQL uses a snapshot-based concurrency protocol that creates multiple versions of data rows while modifying the data. These data row versions are used to read a visible version of the data using a computed snapshot without acquiring read-lock on the data row. PostgreSQL maintains transaction IDs (inserted and deleted transaction IDs) for every row of data and uses the transaction IDs along with the computed snapshot to determine the visibility of the row. As the data keeps growing due to old versions of data, the time taken to scan the data (table scan or index scan) increases. To optimize the response time of the scan operation and to use space efficiently, you need to reclaim the versions and the metadata (for example, transaction ID) that is used to maintain the versions. Reclaiming the transaction ID, freezing transaction ID is required to avoid reaching the maximum limit of the transaction ID space. Freezing the transaction ID involves marking flags in the row header or changing the transaction ID to a minimum value. This operation considers all data rows (active data rows and old versions of the data row) and modifies the row header or transaction ID based on the threshold transaction ID, `frozenxid`.

The `VACUUM` operation reclaims the deleted versions (garbage collection) and transaction IDs (freeze transaction ID). The `VACUUM` operation operates on data in different modes with different levels of data availability. Freezing transaction IDs is crucial to the health of the database system because the system blocks writers whenever the used transaction ID space enters reserved space. For an example, see What We Learned from the Recent Mandrill Outage. The `autovacuum` jobs that you configure constantly try to reclaim the transaction ID, but they can fail. This failure is either due to insufficient configuration or because the creation rate for transaction IDs is so high that the `autovacuum` job cannot catch up with workload. The purpose of this document is to show how to use the `VACUUM` operations along with the mechanisms to tune and monitor different aspects of `VACUUM` operations.

Overview of MVCC

PostgreSQL implements snapshot-based concurrency control mechanisms by maintaining multiple copies of the data rows. This mechanism is also called multiversion concurrency control (MVCC). There is no uniform approach to maintaining the older version of the data rows and each database management system (Oracle, SQL Server, and MySQL) implements the mechanism differently to suit their requirements.

The PostgreSQL storage manager does not differentiate between multiple versions of the data

rows. PostgreSQL stores all the data rows of the tables and the metadata that is used for qualifying the data rows. This metadata includes the transaction IDs of the transactions that created the data row and (if the row is updated or deleted) deleted the data row. Additional header information is maintained to resolve the visibility of a row for a computed snapshot (transactional or statement level snapshot). This mechanism forces `UPDATE` operations on data rows to create new rows with modified columns by marking the existing data row as deleted. For example, say there is a transaction with the ID of 100. If you update row `X`, the system marks the row `X` deleted transaction ID with 100 and creates a new row `X"` with the inserted transaction ID of 100. PostgreSQL maintains a forward pointer from the older version to the newer version. In this example, the pointer is from `X` to `X"`.

Database engines that use optimistic concurrency control mechanisms create a snapshot at the beginning of the transaction or statement. This snapshot includes information about the state of transactions in the database engine, for example, the smallest transaction ID. If any transaction ID is smaller than the smallest transaction ID, then the given transaction ID is either committed or rolled back. A PostgreSQL snapshot contains the following information:

- The smallest transaction ID. Any transaction ID that is lower than this number is either committed or rolled back.
- The largest transaction ID. Any transaction ID that is higher than this number is treated like an uncommitted transaction.
- The list of open transactions. For example, the list of transaction IDs that are active at the beginning of the statement or the transaction.

PostgreSQL uses the snapshot information, commit log information, and transaction IDs stored in the data row to determine whether a given row is visible to the statement or the transaction. To simplify the row visibility process, PostgreSQL caches the commit status of the transactions, which is obtained from the commit log (CLOG) in the row header. The successive processes that try to resolve the visibility of the row use this cache.

This transaction and snapshot system differentiates PostgreSQL from other MVCC implementations due to the following reasons:

1. Transaction `COMMIT` and `ROLLBACK` are O(1) operations because they are setting bits in the commit log (2-bit transaction status). This operation avoids the replay of log entries to undo the rolled back transaction.
2. Reading older versions of a row does not involve the additional cost of constructing the row by using the undo log segment. All the rows (both older and newer versions of the row) are accessed by using regular heap access methods. This representation may lead to more I/O while reading the data whenever there is a bloat in the page, such as when there are many updated or deleted rows.
3. Truncating dead tuples or versions is not part of any regular data manipulation language (DML) or read operation. One exception is an `UPDATE` operation that fails to find space

on the page. The operation tries to remove dead rows (like the `VACUUM` operation that removes dead rows) to accommodate the new version of the row on the same page before allocating a new page.

Overview of the `VACUUM` operation

The `VACUUM` operation reclaims the deleted or dead (deleted row which is not visible in the database instance) versions and marks the inserted rows as visible rows (freeze inserted transaction ID). It operates on table data in different modes with different levels of data availability. The following sections highlight the critical operation of VACUUM operations along with the usage of the command.

Syntax

The `VACUUM` operation is invoked by using the following VACUUM command:

```
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns
[, ...] ]
```

`option` can be one of the following values:

```
    FULL [ boolean ]
    FREEZE [ boolean ]
    VERBOSE [ boolean ]
    ANALYZE [ boolean ]
    DISABLE_PAGE_SKIPPING [ boolean ]
    SKIP_LOCKED [ boolean ]
    INDEX_CLEANUP [ boolean ]
    TRUNCATE [ boolean ]
```

`table_and_columns` is in the following format:

```
    table_name [ ( column_name [, ...] ) ]
```

The `VACUUM` operation has different modes with different levels of locking. The locking semantics are explained in a different section. If the `VACUUM` operation is performed in `FULL` mode, most of the options are not applicable because it creates new pages for tables and indexes. The following list provides an overview of the options for the `VACUUM` command:

- `FULL`:
    - This option does the following:
        - Copies live tuples to a new table to create new pages.
        - Recreates new indexes on newly created data pages.
        - Switches the storage, and swaps out the `pg_class.relfilenode` catalog to point to the new storage.
    - This operation is similar to what the `CLUSTER` command does, except it does not use any index. When it recreates new data and index layers, it requires twice the size of the table and indexes. If there are replicas, this operation writes newly created pages in WAL. The `VACUUM FULL` option can use twice the amount of space (old copy, new copy, and page images in WAL records).
- `FREEZE`: This extension freezes transaction IDs and removes dead rows from indexes.
- `ANALYZE`: Updates the statistics of a table or column used by the query optimizer. This command is independent, but can be run as part of the `VACUUM` operation for convenience.
- `DISABLE_PAGE_SKIPPING`: The `VACUUM` operation skips processing for visible pages (determined by using a visibility map associated with the table) because they might not contain dead rows or unfrozen data. This option overrides this behavior and considers all the pages.
- `INDEX_CLEANUP`: This flag is introduced in PostgreSQL 12, indexes are not cleaned as part of the `VACUUM` operation. This command skips the cleanup of index entries when the flag is set to FALSE.
- `SKIP_LOCKED`: The `VACUUM` operation processes each page by acquiring exclusive locks on the page. If the page is busy, the `VACUUM` operation has to wait for the lock. This option avoids considering busy pages.

The following example shows the `VACUUM` options:

```
vacuum (INDEX_CLEANUP OFF) new_orders;
vacuum (ANALYZE, SKIP_LOCKED OFF, INDEX_CLEANUP OFF) orders;
vacuum FULL history;
vacuum ANALYZE stock;
```

Reclaim space

The `VACUUM` operation reclaims space that is associated with dead rows in the following phases:

1. **Prune Heap Only Tuple (HOT) chains**: The DML `UPDATE` operations operate on non-key columns, which result in the chaining of data rows. The `VACUUM` operation removes the dead rows from the chain and points to the first valid version in the chain. If all the rows in the chain are dead (for example, when all rows are deleted and no process can see these rows in the system) then the entire chain is discarded.
2. **Clear indexes**: Remove the index entries associated with dead or deleted rows to clean up index entries. This phase can reduce the height of the index by removing the entries associated with dead rows.
3. **Remove dead rows**: Identify data rows that are dead and reclaim the space associated with these data rows.
4. **Defragment data pages**: Rearrange the rows on blocks to maintain or keep contiguous free space.

Freeze transaction ID

As previously mentioned, PostgreSQL uses transaction IDs in the data row to find the visibility of the rows. Each transaction ID is a 32-bit value and PostgreSQL uses only 2 billion transaction IDs for the entire database instance. A 32-bit transaction ID covers 4 billion transaction IDs, but to support wraparound of the transaction ID, PostgreSQL uses 2 billion as the max space. For more information, see PostgreSQL transaction ID.

Freezing transaction IDs involves updating the tuple header with respective flags (the inserted transaction ID is frozen). Before PostgreSQL version 9.2, transaction ID (inserted transaction ID) in the tuple is replaced as the minimum transaction ID (called the frozen transaction ID).

Database instances with high write-only workloads can consume the entire 2 billion transaction ID space. Any further operations (such as transaction ID 2 billion + 1) can lead to wrong results due to the signed comparison of transaction IDs. To avoid this problem, PostgreSQL blocks writes after reaching an internally defined threshold value. The last 1 million transaction IDs are reserved. When the system reaches the threshold value, new transaction IDs are allocated. This process blocks all write operations, including user-instantiated `VACUUM` operations. This system can be restored by executing the VACUUM operation in single-user mode. The `VACUUM` operation in single-user mode uses aggressive mode and fixes the system by reclaiming the transaction IDs whose changes are visible to all the processes in the system. The inserted rows are visible and deleted rows are invisible. The space reclaimed from these transaction IDs is used for more transactions. This process is called freezing transaction IDs. Freezing transaction IDs is crucial to the health of the database system because the system blocks writers whenever

the used transaction ID space enters reserved space. For an example of the system blocking writers, see What We Learned from the Recent Mandrill Outage.

The `autovacuum` process

To avoid blocking write transactions and to avoid bloating of the database space, pay attention to freeze transaction IDs and to reclaim dead rows/tuples. These operations can be handled by executing `VACUUM` commands on the candidate tables. For more information about identifying candidate tables, see the monitoring and tuning section. These commands read all the pages of the table and perform the required cleanup on the qualified pages. The amount of time spent on a table depends on the size of the table and the number of dirty pages in the buffer cache (`shared_buffers`). Running these operations frequently results in a significant consumption of system resources, without processing any pages, potentially due to having no change in the state of the table. However, running these operations less frequently increases the duration of the command execution and might reduce throughput of the system.

The `autovacuum` process is an internal process that performs the `VACUUM` operation on the tables based on the statistics maintained (per table or database) in the database system. This process considers tables that need attention based on the `autovacuum` settings and performs the required `VACUUM` operations. The `autovacuum` process cleans up multiple tables concurrently by spawning `autovacuum` worker processes. PostgreSQL spawns, by default, three `autovacuum` workers. You can modify this value based on the workload. If many tables need to be processed, try increasing the worker process.

The PostgreSQL configuration parameter `autovacuum_max_workers` controls the maximum number of worker processes associated with the `autovacuum` process. This configuration option is a *static option*, meaning that the database instance needs to be restarted to get the configured number of worker processes.

The following is an example of the `autovacuum_max_worker` parameter:

```
ALTER SYSTEM SET autovacuum_max_workers=8;
```

The `autovacuum` process executes all the operations supported by the general `VACUUM` command, such as reclaiming dead rows, freezing the transaction ID, and refreshing table statistics. `autovacuum` processes tables in one of the two following modes:

- Normal mode: A non-blocking operation that never blocks user operations on the table. `autovacuum` terminates the execution of normal operation whenever it blocks a user-initiated operation.
- Aggressive mode or uninterruptible anti-wraparound mode: A blocking operation that runs to completion by blocking data definition language (DDL) operations.

Monitoring and tuning `VACUUM` operations

Most customers use default or suboptimal `VACUUM` or `autovacuum` configuration options that are not reflective of their workloads. These default configuration values were defined years ago based on the speed of hardware then. We recommend that you update the `VACUUM` configuration options based on your current available hardware resources and the nature of the workload to help you avoid performance or downtime issues.

There are many configuration options or database flags available for `VACUUM` and autovacuum operations. The following sections describe some of the options that are crucial for maintaining the health of the database instance.

Memory

The `VACUUM` or `autovacuum` operations process a table in batches; it collects a set of tuples or blocks and processes them before moving on to the next set of tuples. The following operations are performed for each batch of tuples:

1. Prune HOT chains, which coalesce the chain by removing dead rows from chained tuples.
2. Remove dead rows—the data rows associated with rolled back transactions or committed deleted rows with no visible snapshot on these rows.
3. Freeze rows to reuse the transaction IDs to enable the reusing of the transaction ID.
4. Remove index entries associated with dead or deleted rows by scanning the full index.
5. Rearrange the rows on blocks to maintain contiguous free space.

The amount of memory used by the `VACUUM` or `autovacuum` operation is defined by the `maintenance_work_mem` and `autovacuum_work_mem` configuration parameters. The default value for these options is 64 MB. For example, consider a TPCC table `orders` that contain 10 billion rows with two indexes and 10% of the data is updated (leaving 1 billion dead rows). The `VACUUM` operation creates batches of tuples by storing 8 bytes (assuming that page and row number occupies 8 bytes for calculation purposes) of information; 64 MB can accommodate eight million entries. To apply the `VACUUM` operation, the `orders` table needs at least nine iterations. This iteration results in nine full index scans (index is on 10 billion rows), which consumes a significant amount of memory and CPU. In addition, it also increases the time to complete the `VACUUM` operation.

To avoid these problems, you are required to configure memory-related options based on the following:

- Instance size
- Number of indexes on the table (especially large indexes that don't fit in memory)
- Nature of the workload.

For the `autovacuum` process, these options are used in each `autovacuum` worker process. Each process uses configured memory.

Note: These configuration options are not dynamic configuration parameters, so the configuration needs to be reloaded.

The following is an example of the `maintenance_work_mem` option:

```
tpcc# SHOW maintenance_work_mem;
maintenance_work_mem
----------------------
 16GB
(1 row)

tpcc# ALTER SYSTEM SET maintenance_work_mem = "24GB";

tpcc# SELECT pg_reload_conf();
pg_reload_conf
----------------
 t
 (1 row)

tpcc# SHOW maintenance_work_mem;
maintenance_work_mem
----------------------
 24GB
 (1 row)
```

Note: The `ALTER SYSTEM` command might not be available for Cloud SQL. Use Cloud SQL database flags to modify the configuration option.

Freeze transaction IDS

Modern storage devices are cheaper than older storage devices. The increased storage space can cause performance problems, but does not cause outages for the database system. Reclaiming the transaction ID or freezing the transaction ID is the most crucial operation of the `VACUUM` operation. PostgreSQL blocks all write transactions whenever the transaction ID reaches the reserved space. PostgreSQL writes warning messages while using the last 10 million transaction IDs. When the message occurs, you must schedule the `VACUUM` operation or else the database system might become unusable due to a transaction ID wraparound problem.

If the `autovacuum` or `VACUUM` configuration options are not adapted to the workload, then the `VACUUM` or `autovacuum` operations might not be able to freeze IDs. The rate of write transactions is too high. The following configuration settings increase the frequency of `VACUUM` operations in that they repeat `autovacuum` operations on the table so that they can catch up to the write transaction rate. Frequent `VACUUM` operations can lead to more WAL records because the `VACUUM` operation writes WAL records for all the changes in data or index pages, which contend with runtime activity. Modify the following configuration settings based on the expected WAL data:

- `vacuum_freeze_table_age`: Specifies the threshold limit for triggering the aggressive `VACUUM` operation. It considers all the pages of a table (not simply pages containing dead tuples). A higher value for this option can increase the duration of the `VACUUM` operation and can lead to transaction wraparound problems. To process the table frequently, configure this instance-level value, based on the number of connections and number of CPUs available in the system.
- `vacuum_freeze_min_age`: Specifies the threshold transaction ID to consider for freezing the transaction ID during regular `VACUUM` operations. If the `pg_class.relfrozentxid` table is smaller than this value, the `VACUUM` operation is not performed on this table. If there is enough CPU power and the workload generates too many writes compared to the reads, then setting this value to zero freezes tuples as quickly as possible.
- `autovacuum_freeze_max_age`: Specifies the transaction ID threshold value for the `pg_class.relfrozenxid` table which triggers an aggressive `VACUUM` operation. This option is the same as the `vacuum_freeze_table_age` option, but the `autovacuum` process uses this configuration option. This option can be set at the table level and provides the flexibility to treat hot tables (frequent write operations) differently from warm or cold tables.

The following is an example of the `VACUUM` and `autovacuum` configuration settings:

```
ALTER SYSTEM SET vacuum_freeze_table_age=10000000;
ALTER SYSTEM SET vacuum_freeze_min_age=0;
```

As previously mentioned, some of these options can be at table level.

```
ALTER TABLE orders SET (autovacuum_freeze_max_age=10000000);
```

Note: The `ALTER SYSTEM` command might not be available for Cloud SQL. Use Cloud SQL database flags to modify the configuration option.

Monitor transaction IDs

Optimizing the database options that are suitable for the hardware resources of the system with changing workload is a challenging and continuous process. To find the optimal values for the database configurations, you need to constantly monitor the system and adjust the options based on the monitored results.

The transaction ID space is shared by all the databases in the system (cluster or running database server). You need to find the appropriate tables in any database that holds reclaimed transaction IDs. The transaction ID space is reused only after ensuring that no table in the database system contains footprints of the transaction ID.

PostgreSQL maintains a frozen transaction ID for every table. All rows in the table that contain a transaction ID smaller or equal to the frozen ID are visible or invisible to all processes in the system. Visible rows are inserted and committed data rows, while invisible rows are dead tuples that were removed. This transaction ID is updated at the end of a successful `VACUUM` operation. Frozen transaction IDs, along with current transaction ID information, determine the age of the table. These table-level frozen transaction IDs can compute the database or database instance-level frozen transaction ID. These values are helpful in determining the correct database and tables that can move forward in the transaction ID space.

The reclaiming of transaction IDs depends on the age of the `pg_class.relfrozentxid` of the tables. If there is any concern about the tables selected by the `autovacuum` process, use the following queries to find the tables that can move the transaction ID values.

The following query reports the database system level usage of transaction ID space:

```
WITH max_age AS (
    SELECT 2000000000 as max_old_txid
        , setting AS autovacuum_freeze_max_age
        FROM pg_catalog.pg_settings
        WHERE name = 'autovacuum_freeze_max_age' )
, per_database_stats AS (
    SELECT datname
        , m.max_old_txid::int
        , m.autovacuum_freeze_max_age::int
        , age(d.datfrozenxid) AS oldest_current_txid
    FROM pg_catalog.pg_database d
    JOIN max_age m ON (true)
    WHERE d.datallowconn )
SELECT max(oldest_current_txid) AS oldest_current_txid
    , max(ROUND(100*(oldest_current_txid/max_old_txid::float))) AS
consumed_txid_pct
    , max(ROUND(100*(oldest_current_txid/autovacuum_freeze_max_age::float)))
AS consumed_autovac_max_age
FROM per_database_stats;
```

For example, consider the output of the `consumed_autovac_max_age` query on a high memory instance hosting the TPCC database:

```
oldest_current_txid | consumed_txid_pct | consumed_autovac_max_age
--------------------+-------------------+-------------------------
          208329838 |                10 |                      104
(1 row)
```

These results are taken after running the benchmark for 30 minutes with 1024 TPCC clients and contain the following information:

- `oldest_current_txid` is the oldest transaction ID in all databases.
- `consumed_txid_pct` represents the percentage of transaction ID space used out of the two billion transaction ID spaces. A larger value means that the database system might get into transaction wraparound problems. We recommend keeping the value below 85% to keep the database operational and avoid outages in the database system.
- `consumed_autovac_max_age` represents the percentage of consumed transaction ID space that triggers the aggressive `VACUUM` operations. This percentage is based on the `autovacuum_freeze_max_age`. If the value is more than 100, it triggers an aggressive `VACUUM` operation based on the age of the oldest frozen transaction ID.

Query that identifies the database with the oldest transaction ID

If the previous query reports that `consumed_txid_pct` is too high, the next step is to identify the database that contains the oldest transaction ID. The following query reports information similar to the preceding query but breaks it down at the database level:

```
SELECT datname, age(datfrozenxid) AS frozen_xid_age
     , ROUND(100*(age(datfrozenxid)/2000000000.0::float))
consumed_txid_pct
     , current_setting('autovacuum_freeze_max_age')::int
      - age(datfrozenxid) AS remaining_aggressive_vacuum
FROM pg_database;
```

For example, consider the output of the preceding database-level query on a sample TPCC database:

```
   datname    | frozen_xid_age | consumed_txid_pct | remaining_aggressive_vacuum
--------------+----------------+-------------------+-----------------------------
 cloudsqladmin |       50200078 |                 3 |                   149799922
 template0     |         200078 |                 0 |                   199799922
 postgres      |       50202602 |                 3 |                   149797398
 template1     |         200078 |                 0 |                   199799922
 tpcc          |      209401713 |                 9 |                    -9401713
(5 rows)
```

These results contain the following information:

- `datname` contains the name of the database.
- `frozen_xid_age` represents the age of the database-level frozen transaction ID. A higher value (for example, greater than `autovacuum_freeze_max_age`) means that the database needs attention.
- `consumed_txid_pct` represents the percentage of the transaction ID against the maximum transaction ID limit (2 billion transaction IDs) for the database.
- `remaining_aggressive_vacuum` represents the available transaction ID space before it reaches the aggressive `VACUUM` mode—how close the database is to the `autovacuum_freeze_max_age` value. A negative value means that there are some tables in the database that trigger an aggressive `VACUUM` operation due to the age of `pg_class.relfrozentxid`.

After you identify the target database, you identify the tables within the database that are candidates for the `VACUUM` operation. The following query reports the top 50 tables that need a `VACUUM` operation performed on them. To maintain the health of the database system, we recommend scheduling a `VACUUM` operation on the reported table without impacting I/O, CPU, or WAL. You can perform a `VACUUM` operation on a batch of tables instead of scheduling a `VACUUM` operation on all the tables at once.

Query that identifies tables that need the `VACUUM` operation

The following example identifies the top 50 tables that need the `VACUUM` operation performed on them:

```
SELECT c.oid::regclass
    , age(c.relfrozenxid)
    , pg_size_pretty(pg_total_relation_size(c.oid))
FROM pg_class c
JOIN pg_namespace n on c.relnamespace = n.oid
WHERE relkind IN ('r', 't', 'm')
AND n.nspname NOT IN ('pg_toast')
ORDER BY 2 DESC LIMIT 50;
```

The output is similar to the following:

```
           oid            |    age    | pg_size_pretty
--------------------------+-----------+----------------
 history                  | 210000101 | 12 GB
 stock                    | 210000101 | 37 GB
 customer                 | 210000101 | 20 GB
 order_line               | 210000101 | 176 GB
 pg_statistic             | 128239305 | 736 kB
 district                 | 123162457 | 51 MB
 new_order                | 112077299 | 2938 MB
 warehouse                | 109644862 | 20 MB
 pg_type                  |  90215950 | 192 kB
...
```

Even after providing a better monitoring infrastructure, the current workload can potentially block the `VACUUM` operation such that the `VACUUM` operation is unable to reclaim space or freeze transaction IDs. The `VACUUM` operation can be blocked by the following:

- Long running transactions or snapshots. Identify these transactions and terminate them to unblock the `VACUUM` operation.
- Abandoned replication slots. Drop the abandoned slots.
- Orphaned prepare transaction. Roll back orphaned prepared transactions.
- Long running snapshots or transactions at a standby replica created with `hot_standby_feedback`. Identify the long running session or transactions on replica and terminate the blocking sessions.

The following query retrieves the oldest value for each of the `VACUUM` operation blockers:

```
SELECT
(SELECT max(age(backend_xmin)) FROM pg_stat_activity) as
oldest_running_xact,
(SELECT max(age(transaction)) FROM pg_prepared_xacts) as
oldest_prepared_xact,
(SELECT max(age(xmin)) FROM pg_replication_slots)     as
oldest_replication_slot
(SELECT    max(age(backend_xmin))    FROM    pg_stat_replication)    as
oldest_replica_xact
;
```

Ideally, the `autovacuum` process picks up and processes the appropriate tables after resolving the blocking operations. But it can be useful to identify the tables or databases that have the oldest transaction ID and run the appropriate `VACUUM` commands on the tables.

Reclaim storage space

Even though increasing the space associated with tables and databases is not leading to outages, it can impact the performance of queries. The efficiency of indexes depends on the clusteredness of the data. If there are a significant number of dead tuples, then the index records that point to dead rows consume space and impact the cost of index scans. We recommend that you read as few pages as possible to process queries. Otherwise, the query response time is impacted by I/O latency and can require more I/O than expected. At the same time, having more data pages or blocks can lead to performing the wrong query plans, which might not meet response time requirements.

As explained in the overview section, dead rows are the result of deleted or updated rows. The following are some of the database options that control `VACUUM` operations based on the number of rows and the number of dead tuples in the table:

- `autovacuum_vacuum_threshold`: This value represents the minimum number of dead tuples required before considering a table for a `VACUUM` operation. The default value is 50 tuples. If any table has fewer than 50 dead tuples, the `autovacuum` process does not perform a `VACUUM` operation on the table.
- `autovacuum_vacuum_scale_factor`: This value represents the fraction of table size added to `autovacuum_vacuum_threshold` to qualify for the `VACUUM` operation. The default value for the option is 0.2 or 20% of the table size.

The preceding two options are used by the `autovacuum` process to qualify a table for a `VACUUM` operation. The following formula is used by the `autovacuum` process to qualify the table for a `VACUUM` operation:

```
#of dead tuples in a table (pg_stat_all_tables.n_dead_tup) >
(autovacuum_vacuum_threshold + pg_class.reltuples *
autovacuum_vacuum_scale_factor)
```

Track operations on tables

Because workload changes are based on constantly changing requirements, you need to keep refining the configuration options. To refine the configuration options, you need to track the number of dead tuples produced in the system.

The following query reports the number of logically inserted, updated, and deleted rows for each table in a connected database. These values do not represent reclaimable space because there might be sessions for which the deleted or updated rows are still visible because of MVCC.

Query that identifies logically insert, updated, and deleted rows per table

The following is an example of a query for logically inserted, updated, and deleted rows per database table:

```
SELECT relname, n_tup_ins, n_tup_upd, n_tup_del
FROM pg_stat_all_tables
ORDER BY n_dead_tup DESC LIMIT 20;
```

The output is similar to the following:

```
relname                  | n_tup_ins | n_tup_upd | n_tup_del
-------------------------+-----------+-----------+-----------
 order_line              | 556742761 | 403246159 |         0
 new_order               |  44921508 |         0 |  40196757
 stock                   |  51200000 | 402743889 |         0
 customer                |  15360000 |  78188896 |         0
 orders                  |  55678256 |  40325550 |         0
 district                |      5120 |  80649354 |         0
 warehouse               |       512 |  40329213 |         0
 pg_statistic            |        92 |      4942 |         0
 pg_toast_2619           |      3931 |         0 |      3856

...
```

Query that identifies dead tuples

To reclaim the space, you need to find the exact number of dead tuples. The following is an example query to find dead tuples:

```
SELECT relname, n_live_tup, n_dead_tup, n_tup_hot_upd
FROM pg_stat_all_tables
ORDER BY n_dead_tup DESC LIMIT 20;
```

The output is similar to the following:

```
relname                  | n_live_tup | n_dead_tup | n_tup_hot_upd
-------------------------+------------+------------+---------------
 order_line              |  559111721 |   92094372 |     404036839
 new_order               |    4725336 |    4200727 |             0
 stock                   |   51199928 |    3903734 |     403591655
 customer                |   15362085 |     568888 |      78353381
 orders                  |   55928160 |     399769 |      40411453
 district                |       5120 |       1633 |      80749601
 warehouse               |        512 |        193 |      40341263
 pg_statistic            |        514 |         99 |          4415
 pg_toast_2619           |         88 |         59 |             0
 pg_amproc               |        447 |          0 |             0

...
```

The output contains the following information:

- `N_live_tup`: The number of live tuples.
- `N_dead_tup`: The number of dead tuples. If there are more dead tuples than live tuples, reclaiming space should be evaluated.
- `N_tup_hot_upd`: The number of HOT updated tuples. This value represents the non-index key updates—no index column is modified by this update. Tuples are chained and old tuples eventually are deleted by the `VACUUM` operation.

Throttle the `autovacuum` process

The `VACUUM` operation is a maintenance operation and it should not impact the runtime performance of the application. This operation consists of reading pages, verifying whether the page contains any tuples that need to be reclaimed or frozen, processing the qualified rows, and marking the page as dirty. This operation also requires writing WAL records associated with the operations. This entire process consumes system resources (like CPU, memory, and I/O) and changes the state of the database system buffer cache. If the `VACUUM` operation is triggered (automatically or manually) without knowledge of the current workload, then it impacts the application throughput.

You can delay manual `VACUUM` commands based on the workload, but controlling the `autovacuum` process needs special attention. Throttling is a feature that controls the operations of the `autovacuum` operation. Throttling is a cost-based mechanism and takes the following configuration options into consideration:

- `vacuum_cost_page_hit`: Estimated cost of the `VACUUM` operation if a page is found in the PostgreSQL shared_buffers cache.
- `vacuum_cost_page_miss`: Estimated cost of the `VACUUM` operation if a page is not found in the shared buffer cache, if the page needs to be read from disk or a file system buffer cache.
- `vacuum_cost_page_dirty`: Estimated cost of the `VACUUM` operation if it modifies a clean page and removes either dead tuples or frozen transaction IDs on the page.
- `vacuum_cost_limit:` The cost of the `VACUUM` operation when it processes in batches. When the `VACUUM` operation reaches the limit, the `VACUUM` process goes into a sleep state.
- `vacuum_cost_delay:` The amount of time that a `VACUUM` process sleeps after reaching `vacuum_cost_limit`.

Query that identifies the default values of the `VACUUM` configuration options

The following query shows the default values of the `VACUUM` configuration options:

```
SELECT name, setting
FROM pg_settings WHERE name LIKE 'vacuum_cost%';
```

The output is the following:

```
 name                    | setting
-------------------------+---------
 vacuum_cost_delay       | 0
 vacuum_cost_limit       | 200
 vacuum_cost_page_dirty  | 20
 vacuum_cost_page_hit    | 1
 vacuum_cost_page_miss   | 10
(5 rows)
```

The default cost limit of the `VACUUM` operation is 200 and it never sleeps because the value of `vacuum_cost_delay` is 0).

The `autovacuum` process has its variants of the same configuration options as `autovacuum_vacuum_cost_limit` (default 20 milliseconds, -1 uses `vacuum_cost_limit`). The `autovacuum` process can spawn multiple worker processes to handle different tables. In that case, the `autovacuum_vacuum_cost_limit` is distributed over all the `autovacuum`

worker processes so that each worker process gets a `vacuum_cost_limit` or `autovacuum_max_workers`) value.

These default values do not reflect the state of the current hardware resources. For example, the `VACUUM` operation performs 100 iterations per second with the default `vacuum_cost_limit` of 200 per iteration meaning 20000 per second. This iteration covers the following:

- 20000 * 8 KB block size = 160 Mbps reads from the buffer cache. If the `shared_buffers` are in GB, all the active pages can be in the cache (`vacuum_cost_page_hit` = 1)
- 16 Mbps reads from the disk or file system cache (`vacuum_cost_page_miss` = 10)
- 8 Mbps writes, assuming all pages are clean and processed by the `VACUUM` operation (`vacuum_cost_page_dirty` = 20)

Most of the database systems use gigabytes of `shared_buffers` with advanced I/O devices. We recommend modifying the configuration options based on the hardware and system configuration to avoid multiple smaller `VACUUM` operation iterations into one large operation, which reduces the burden on the WAL subsystem.

Most current database instances can generate more than 160 MB of dirty pages and it is possible that a `VACUUM` operation needs to read the pages from disk to process them. To keep table data always clean, the `VACUUM` operation needs to catch up with the rate of generation of new data.

Throttle at table level

The `VACUUM` operation can't differentiate between hot tables, warm tables, or cold tables. A hot table is a table that has more write operations that need to be handled differently than other tables. For example, it might need more time for each iteration of the `VACUUM` operation compared to other tables. The following `autovacuum` configuration options can be applied to the system level and the table level:

- `autovacuum_vacuum_cost_limit`: Similar to `vacuum_cost_limit`, but used by the `autovacuum` process. The default is -1—the `vacuum_cost_limit` value.
- `autovacuum_vacuum_cost_delay`: Similar to `vacuum_cost_delay`, but used by the `autovacuum` process. The default is -1—the `vacuum_cost_delay` value.

To modify these configuration options, you can use the `ALTER TABLE` command.

The following example modifies the values for the `orders` table:

```
ALTER TABLE orders SET (autovacuum_vacuum_cost_delay=10);
ALTER TABLE orders SET (autovacuum_vacuum_cost_limit=10000);
```

Configuring automated Cloud Monitoring alerts

PostgreSQL generates a warning log message like "`database X must be vacuumed within N transactions`", whenever the transaction ID reaches the warning limit. The default is the last 10 million transaction IDs of 2 billion transaction IDs. To identify potential wraparound problems, you can create a log-based metric to count log messages, and you can set up automated Cloud Monitoring alerts to identify impending problems based on error messages.

Create a log-based metric to track database warning messages

The following step creates a logs-based counter metric in the Cloud Console to track the database warning log messages regarding transaction wraparound:

1. In the Google Cloud Console, go to the **Logs-based metrics** page.

   Go to Logs-based metrics

2. Click **Create Metric**.
3. In the **Filter by label or text search** field, click the drop-down arrow and select **Convert to advanced filter**.
4. Enter the following query: `resource.type="cloudsql_database" textPayload=~"database.*must be vacuumed within.*transactions"` Cloud Monitoring uses regular expressions to match the database messages that contain warnings. For more information, see using regular expressions in advanced queries.
5. Click **Submit Filter**.
6. In the **Metric Editor** panel, set the following fields:
   - **Name**: Choose a name that is unique among the logs-based metrics in your project. For information about naming restrictions, see Troubleshooting.
   - **Description**: Enter a description for the metric.
   - **Labels**: (Optional) Add labels by clicking **Add Item** for each label. For details about defining labels, see Logs-based metric labels.
   - **Type**: Counter.
7. Click **Create Metric**.

The new metric appears in the Logs Viewer's list of metrics. Data is available in less than a minute. For more information, see creating counter metrics.

Create an alert policy using the log-based metric

1. In the Cloud Console, go to the **Logs-based metrics** page.

   Go to Logs-based metrics

2. In the **User-defined metrics** section, find the metric that you created. Click **More**, and select **Create alert from metric**.
3. In the **Target** dialog, do the following:
   - Set the **Aggregator** value to **none**.
   - Click **Show Advanced Options** and set the **Aligner** value to **mean**.
   - For **Condition name**, enter `logging/user/cloudsql-pg-vacuum-txn-id`.
4. In the **Configuration** dialog, do the following:
   - Set **Condition triggers if** to **Any time series violates**.
   - Set **Condition** to **is above**.
   - Set **Threshold** to **0**.
   - Set **For** to **Most Recent**.
5. Click **Save**.

For more information on alerting policies, see creating an alerting policy on a counter metric.

Locking semantics

The `VACUUM` operation uses different levels of locking semantics based on the option used by you or determined by the configuration option. The aggressive `VACUUM` operation is performed based on the `autovacuum_freeze_max_age` value. The following lists the locking behavior used by the `VACUUM` operation:

- The `VACUUM FULL` operation creates an entirely new data and index layer using an `AccessExclusive` lock on the table. No other process is able to access the table while the `VACUUM FULL` operation is being processed.
- Another variation of the `VACUUM Full` operation is the `CLUSTER` command. Like the `VACUUM FULL` operation, the `CLUSTER` option also creates new data and index pages with the difference that the data is ordered based on the given index key.
- Other variations of the `VACUUM` operation acquire the `SharedUpdateExclusive` lock on the table. This lock restricts DDL and other `VACUUM` operations on the table, but allows all the regular operations. To operate on a consistent copy of the data, the `VACUUM` operation acquires the `cleanup_lock` for each page. This `cleanup_lock` is an exclusive lock on the page, so no operations are allowed on the page until it is completed on that page.
- If the `VACUUM` operation is cleaning the table using the `SharedUpdateExclusive` lock, then DDL operations are not allowed on the table. This can lead to blocking data

manipulation language (DML) operations whenever there is a pending DDL request. The backend that is requesting the DDL lock waits on the `VACUUM` operation, and any DML after this DDL request has to wait behind the DDL lock request. This process results in a total halt of the system on a table which is actively being processed by the `VACUUM` operation.

- Locking semantics for both `VACUUM` and `autovacuum` operations are the same except that the `autovacuum` process unblocks any pending operations on the table by canceling the operation that blocks the user operation. If the `autovacuum` process is performing an aggressive freeze then it will run to completion.

`VACUUM` operation on system catalogs

In most cases, the operations on system catalog tables—inserting new entries into the catalog or removing entries—are rare. Applications that frequently create and drop temporary tables can cause bloated system catalog tables. Creating temporary tables results in new tuples in catalog tables and dropping temporary tables marks the tuples as deleted. If the catalogs are bloated, it impacts the performance of all applications because of the latency involved in finding the required catalog data. If a `VACUUM` operation blocks a catalog page or table then it can lead to outages of the system because no query statement is able to complete without catalog data). This results in a requirement that catalogs need to be cleaned up as quickly as possible.

In general, table-level configuration options are optimized to handle the `VACUUM` operation on the table differently from other tables. Unfortunately this is not possible for system catalogs. The `VACUUM` operations on system catalogs use the global configuration options. If the global options are not optimized, and the workload uses many temporary tables, then catalogs will be bloated. Because catalog tables are smaller in size compared to regular user tables, we recommend that you schedule an explicit cron job that performs a non-blocking `VACUUM` operation on system catalog based on the application requirements (can be once per hour).

Reducing outage time

Optimizing and monitoring database instances help in reducing or avoiding `VACUUM` operations-related outages. If there is no action taken based on the monitoring data or from PostgreSQL messages, then you might experience an outage. For example, all write transactions are blocked until the transaction ID space is reclaimed. The following are some recommendations that might reduce the time it takes to complete the `VACUUM` operation:

- Increase the system memory and `autovacuum_maintenance_work_mem` to batch more tuples in each iteration and to complete the work as quickly as possible.
- The `VACUUM` operation generates many WAL records. If there are no replicas configured for this instance, you can reduce the WAL records and the operation completes as quickly as possible. To reduce the WAL generated by the `VACUUM` operation, set `full_page_writes` to `OFF`. This option writes only the tuple header information and

skips writing data-related WAL records. If any operations related to WAL are changed, as a precautionary measure, run the `CHECKPOINT` command before and after the `VACUUM` operation.

- If the table reaches the 2 billion transaction ID limit because none of the tuples are frozen, then try to reduce the amount of work done in single user mode. One possible option is to set `vacuum_freeze_min_age=1,000,000,000`. This new value reduces the number of tuples frozen up to 2X.
- PostgreSQL version 12.0 and later versions support cleanup and `VACUUM` operations without cleaning the index entries. This is crucial because cleaning the index requires a complete index scan, and if there are multiple indexes then the total time depends on index size, as illustrated in the following equation:

```
#of full index scans = # of batches defined by maintenance_work_mem * # of indexes
```

  Larger indexes consume a significant amount of time for the index scan so we recommend setting `INDEX_CLEANUP OFF` to quickly clean up and freeze the table data.
- PostgreSQL versions before 12.0 need to optimize the number of indexes. If there are non-critical indexes, then it can be helpful to drop the non-critical index to speed up the `VACUUM` operation.

Reducing outage of instances with replicas

If there are database instances with replicas, the `VACUUM` operation for the replica happens through the WAL records generated by the primary instance. If the primary instance has optimized the configuration options that maintain the health of tables and database, then the replica is in a healthy state. The replica is configured with enough resources to consider the WAL records generated on the primary instance by the `VACUUM` operation. Those records are applied to the replica to remove dead rows or freeze tuples. However, if the primary instance runs into the transaction ID wraparound problem, then the replica does not get the WAL records associated with the emergency. The `VACUUM` operation is performed in single user mode, but only after the system is started in multi-user mode. Depending on the amount of WAL generated in single-user mode, the replay at replica might take a significant amount of time. To reduce the time it takes to catch up, we recommend that you recreate a new replica instead of waiting for the current replica to catch up with the WAL generated at the primary instance. If the `VACUUM` operation cleaned up a table with the `full_page_writes` configuration option turned off, then replicas need to fetch the pre-images from the disk before applying the changes associated with WAL records. This configuration increases the reply time exponentially.

Enhancements in PostgreSQL-13

The `autovacuum` process uses the number of deleted rows to trigger the `VACUUM` operation on a table. This process can delay the `VACUUM` operation of tables that have more INSERT than `DELETE` or `UPDATE` operations. Ignoring `INSERT`-only tables can lead to the rapid consumption of transaction ID space and lead to potential transaction wraparound problems.

PostgreSQL-13 enhances statistics considered by the `autovacuum` process to perform the `VACUUM` operation on insert-driven tables as quickly as possible. PG13 tracks the number of inserted rows from the previous successful `VACUUM` operation and uses this value to trigger the `VACUUM FREEZE` operation on the respective tables. To cover the insert driven tables, PG13 introduced the following:

- `n_ins_since_vacuum in pg_stat_all_tables`: This counter tracks the number inserted rows from the last `VACUUM` operation.
- `autovacuum_vacuum_insert_threshold`: Specifies the number of tuples that need to be inserted to trigger the `VACUUM` operation. The default value is 1000. If the rate of insertion in the table is too high compared to the default value, then it's better to adjust the value to avoid too frequent `VACUUM` operations on the table.
- `autovacuum_vacuum_insert_scale_factor`: This option is similar to `autovacuum_vacuum_scale_factor`, but deals with inserted rows. This value represents the fraction of the table size to add to `autovacuum_vacuum_insert_threshold` to trigger the `VACUUM` operation. The default is 0.2 (20% of table size).

The `autovacuum` process uses the following formula to freeze a table based on the insert-related options:

```
#of tuples tracked by n_ins_since_vacuum >
autovacuum_vacuum_insert_threshold +  #of rows *
autovacuum_vacuum_insert_scale_factor
```

References

- Documentation: 12: VACUUM
- Documentation: 12: 24.1. Routine Vacuuming
- Documentation: 12: 19.10. Automatic Vacuuming
- Autovacuum Tuning Basics
- PG Phriday: 10 Things Postgres Could Improve - Part 1
- Managing Transaction ID Exhaustion (Wraparound) in PostgreSQL
- Transaction ID

Appendix

The following sections include additional information.

Lock compatibility matrix

The following table shows the lock compatibility matrix for PostgreSQL table level locks. Cells containing **X** mean that the request is not compatible with the current lock mode and the requester might have to wait to get the lock. Empty cells mean that the requested lock mode is compatible with the current lock mode and the requester can immediately get the required lock.

Requested mode	**Current lock mode**							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Warning messages

PostgreSQL prints the following warning message whenever the transaction ID reaches the warning limit (default last 10 million transaction IDs of 2 billion transaction ID):

```
/* complain even if that DB has disappeared */
if (oldest_datname)
    ereport(WARNING,
            (errmsg("database \"%s\" must be vacuumed within %u
transactions",
                    oldest_datname,
                    xidWrapLimit - xid),
             errhint("To avoid a database shutdown, execute a
database-wide vacuum in that database.\n"
                     "You might also need to commit or roll back old
prepared transactions, or drop stale replication slots.")));
else
    ereport(WARNING,
            (errmsg("database with OID %u must be vacuumed within %u
transactions",
                    oldest_datoid,
                    xidWrapLimit - xid),
             errhint("To avoid a database shutdown, execute a
database-wide vacuum in that database.\n"
                     "You might also need to commit or roll back old
prepared transactions, or drop stale replication slots.")));
```

Database-level detailed consumed `txid` percentage

The following query outputs database-level details about `consumed_txid_pct`:

```
SELECT datname, age(datfrozenxid) AS frozen_xid_age
     , ROUND(100*(age(datfrozenxid)/2000000000::float))
consumed_txid_pct
     , 2*1024^3 - 1 - age(datfrozenxid) AS remaining_txid
     , current_setting('autovacuum_freeze_max_age')::int
      - age(datfrozenxid) AS remaining_aggressive_vacuum
FROM pg_database;
```