Vacuum

Primary Owner aws-se-wiki-owners (LDAP)

Last modified 3 months ago by murkas.

Overview

Creator(s)	Duration to complete	Last updated
beravi@, murkas@, acchall@	60 minutes	03/28/2024

Purpose

Do you know what Vacuum and Autovacuum is? Have you worked on any Vacuum and Autovacuum related cases? Do you know how to troubleshoot some of the most common scenarios?

In this wiki, we'll walk you through an introduction to several concepts, dive into vacuum and how it operates, classify autovacuum parameters, take a look at logging, CW metrics, and DA, and take a look at how to troubleshoot common scenarios.

Prerequisites
Before you start reading through this wiki, consider the recommended prerequisites to best prepare you to learn.

1. Implement an Early Warning System for Transaction ID Wraparound in Amazon RDS for PostgreSQL
2. Understanding autovacuum in Amazon RDS for PostgreSQL environments

Important concepts

Before we get into Vacuum and Autovacuum, there are a few important concepts you need to be aware of that are listed below.

MVCC and Transaction IDs

PostgreSQL uses Multi-Version Concurrency Control (MVCC) to maintain multiple versions of a row when performing data modifications. During UPDATE and DELETE operations on a table, the database keeps the old versions of the rows for other active transactions that may need a consistent view of the data.

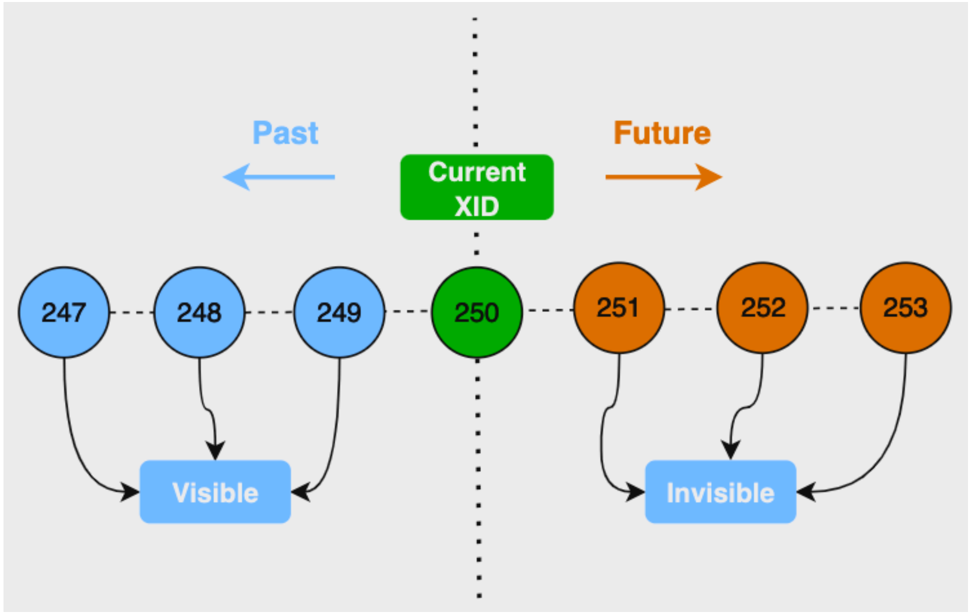
Every transaction (whether created explicitly through 'BEGIN' or START TRANSACTION statements or implicit single-statements transactions) is assigned with a unique identifier called the "Transaction ID", referred to as the XID when it first writes to the database. Transaction ID is a 32-bit unsigned integer that can generate approximately 4 billion transactions.

PostgreSQL stores two metadata columns named xmin and xmax, for every tuple to record the transaction ID information, which are used to track the status of each row. PostgreSQL's multiversion concurrency control (MVCC) mechanism relies on being able to compare the transaction IDs (XIDs) stored in a tuple's xmin and xmax columns with the current active transaction ID (XID). By performing these comparisons, PostgreSQL can determine whether a row version is visible to the current transaction or not.

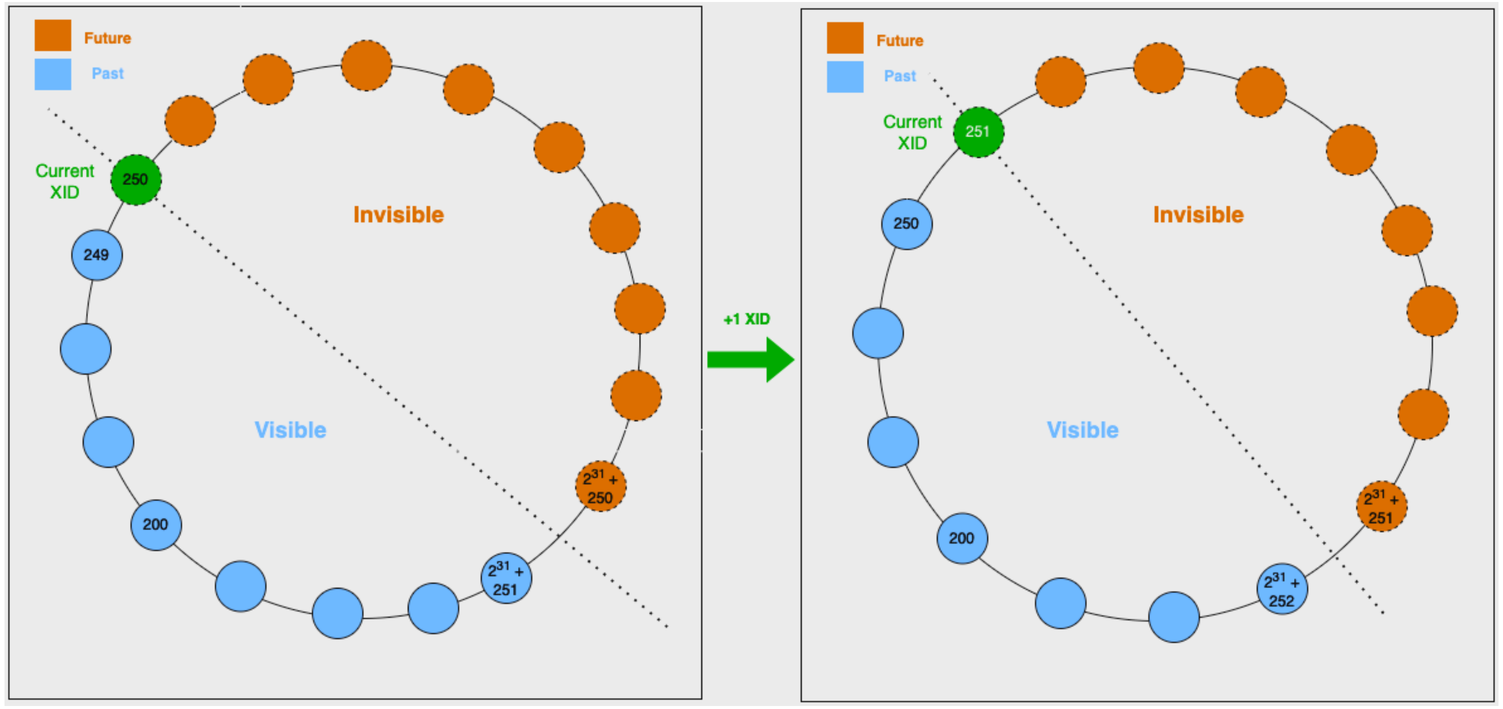
Note: MVCC is achieved using the tuple’s t_xmin and t_xmax values, the commit log (clog), and the current transaction snapshot. For the purpose of this wiki, we are discussing Transaction IDs only.

Let’s take an example **transaction ID 250**.

Any tuples with a transaction ID greater than 250 are considered to be in the future, and because of that, are invisible to the transaction with XID 250. Similarly, any tuples with a XID less than 250 are in the past, and are visible to the transaction with XID 250.



In practical database systems, 4 billion transaction IDs are not enough to accommodate most of the workloads, and for this reason, PostgreSQL treats the transaction ID space as circular with no endpoint. PostgreSQL uses modulo-2^{32} arithmetic, which means that for every transaction ID, there are 2 billion transaction IDs that are older and 2 billion transaction IDs that are newer.



Note: When the PostgreSQL cluster runs for more than 4 billion transactions, the transaction ID counter wraps around.

Transaction ID wraparound

Now that we understand transaction ID and its purpose in PostgreSQL, let's take a look at an example to understand transaction ID wraparound problem.

Scenario:

→ Time T1 and Current XID 250:

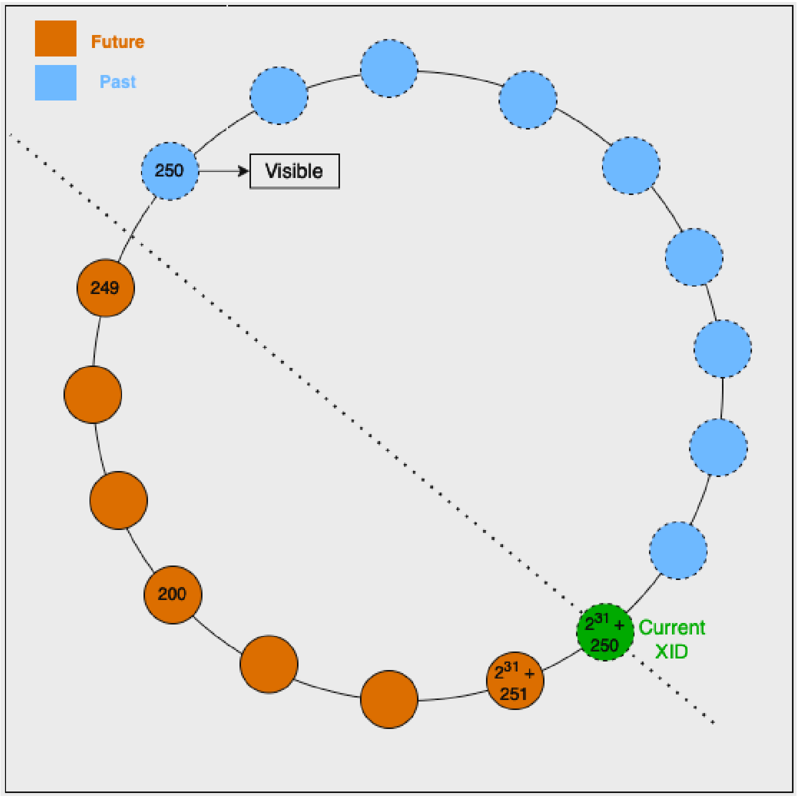
Let's assume that a record was inserted by a transaction with XID 250. This record will have a xmin value of 250 and xmax value of 0.

< After a long time, the inserted record is never modified. This means that the record's xmin and xmax are still the same >

→ Time T20 and Current XID 2.1 billion + 250:

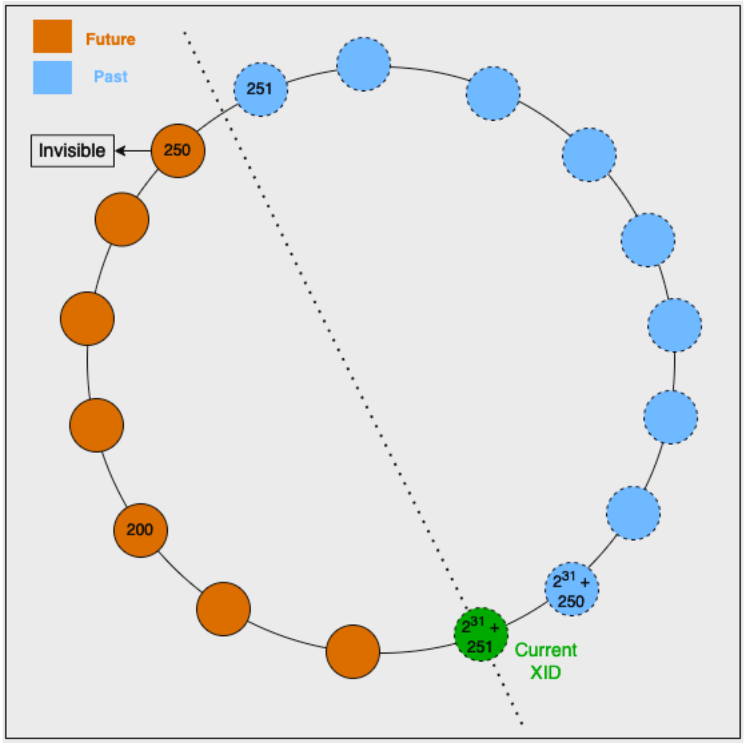
Why 2.1 billion?
Remember that PostgreSQL uses modulo-2^{32} arithmetic. This means that for every transaction ID, there are 2 billion transaction IDs that are older and 2 billion transaction IDs that are newer.

When the transaction with XID of 2.1 billion + 250, the record with xmin of 250 is visible because XID 250 is in the past, as seen below.



→ Time T21 and current XID 2.1 billion + 251:

When the transaction with XID 2.1 billion + 251 executes a select command, the inserted record with xmin of 250 is not visible as the tuple is considered to be in the future from the viewpoint of the current XID.



As discussed earlier, the transaction ID space is circular. When the PostgreSQL database runs for a long time (more than 4 billion transactions), the XID counter wraps around zero and starts over. Due to this behavior, the XIDs that were in the past (such as XID 250 above) appear to be in the future and result in a scenario mimicking catastrophic data loss.

This problem is known as **Transaction ID wraparound**.

Dead tuples

A dead tuple in PostgreSQL is a row in a table that has been marked for deletion but is not yet physically removed from the table.

A delete operation logically deletes the tuple, and update operation logically deletes the existing tuple and inserts a new tuple. These

perfor-

What is Vacuum?

VACUUM is a critical process in PostgreSQL that helps remove bloat on the table and helps prevent transaction ID wraparound. We'll see how vacuum addresses these problems in the following sections.

Phases of Vacuum

PostgreSQL has a handy view 'pg_stat_progress_vacuum' that provides more information on the progress of ongoing vacuum operations whether triggered manually via the VACUUM command or automatically via autovacuum.

For an ongoing vacuum operation, *pg_stat_progress_vacuum* can report one of the following phases:

initializing	VACUUM is preparing to begin scanning the heap. This phase is expected to be very brief.
scanning heap	VACUUM is currently scanning the heap. It will prune and defragment each page if required, and possibly perform freezing activity. The heap_blks_scanned column can be used to monitor the progress of the scan.
vacuuming indexes	VACUUM is currently vacuuming the indexes. If a table has any indexes, this will happen at least once per vacuum, after the heap has been completely scanned. It may happen multiple times per vacuum if maintenance_work_mem (or, in the case of autovacuum, autovacuum_work_mem if set) is insufficient to store the number of dead tuples found.
vacuuming heap	VACUUM is currently vacuuming the heap. Vacuuming the heap is distinct from scanning the heap, and occurs after each instance of vacuuming indexes. If heap_blks_scanned is less than heap_blks_total, the system will return to scanning the heap after this phase is completed; otherwise, it will begin cleaning up indexes after this phase is completed.
cleaning up indexes	VACUUM is currently cleaning up indexes. This occurs after the heap has been completely scanned and all vacuuming of the indexes and the heap has been completed.
truncating heap	VACUUM is currently truncating the heap so as to return empty pages at the end of the relation to the operating system. This occurs after cleaning up indexes.

performing final cleanup	VACUUM is performing final cleanup. During this phase, VACUUM will vacuum the free space map, update statistics in pg_class, and report statistics to the cumulative statistics system. When this phase is completed, VACUUM will end.

These phases can be broadly classified into the following steps:

a) Acquire a ShareUpdateExclusiveLock on the table and scan pages for dead tuples

Vacuum scans the table pages to a make a list of dead tuples that are stored in the memory structure maintenance_work_mem. Vacuum then removes any index tuples that point to these dead tuples, removes the dead tuples, freezes any old tuples if needed in the pages that it visits, and reallocates the live tuples in the page to defragment the table.

Freezing tuples will be discussed in detail in the subsequent sections.

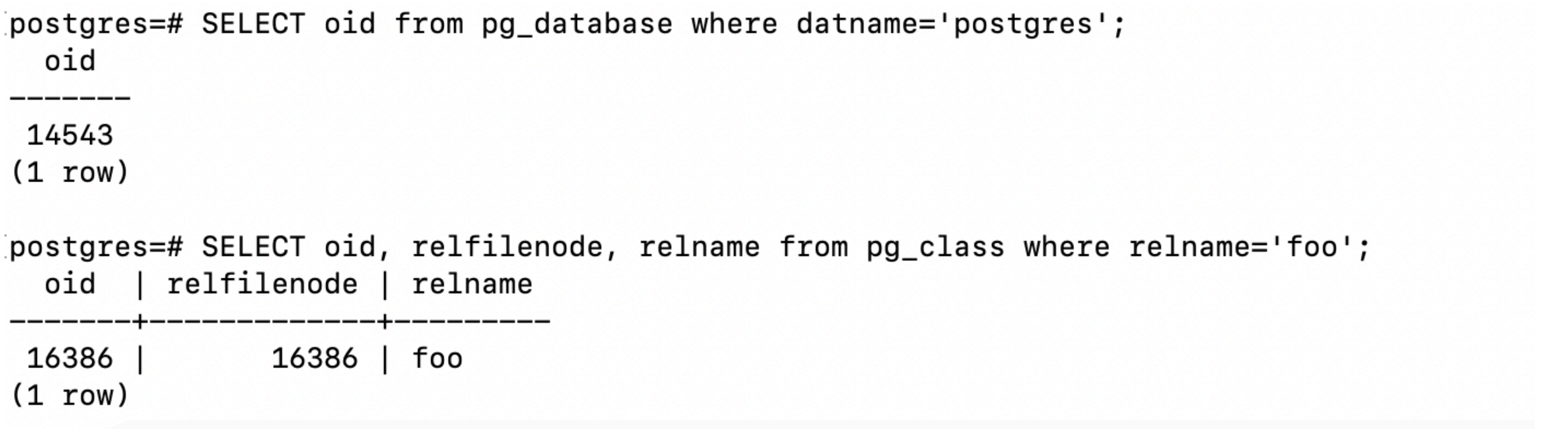
b) Update the Free Space Map and Visibility Map of the table

Vacuum also maintains internal files associated with the table called the Visibility Map and the Free Space Map.

Visibility Map maintains two bits per page. The first bit signifies the tuple visibility information for each page in the table, in other words, whether a given page in a table has dead tuples or not. This information allows subsequent vacuum to skip scanning pages that do not have dead tuples, which speeds up processing. The second bit determines if all tuples on the page have been frozen or not, this speeds up anti-wraparound vacuums that can skip pages that have all frozen tuples. Visibility map information is reset whenever a page is modified. The file is stored as the relfilenode with a “_vm" suffix for each table.

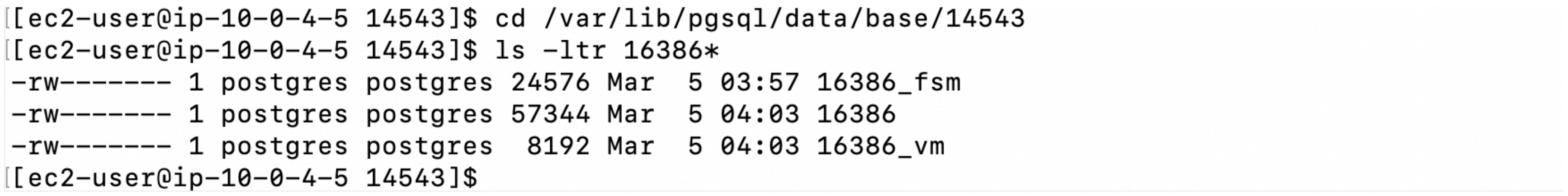
All tables and indexes also have a **Free Space Map** that stores information about available free space for a given page. PostgreSQL uses this information when inserting data into a table/index. The file is stored as relfilenode with a ”_fsm" suffix for each table.

For example, for the given database table “foo“ with a relfilenode value of 16386 in database ‘postgres’, the Visibility Map and Free Space Map are ‘16386_vm’ and ‘16386_fsm’ respectively.



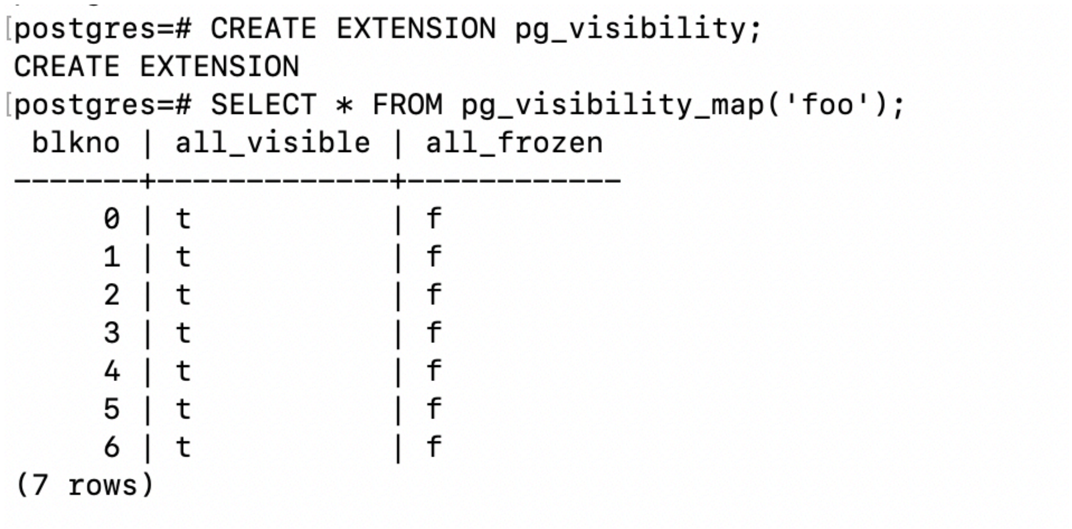
```
postgres=# SELECT oid from pg_database where datname='postgres';
  oid
-------
 14543
(1 row)

postgres=# SELECT oid, relfilenode, relname from pg_class where relname='foo';
  oid  | relfilenode | relname
-------+-------------+---------
 16386 |       16386 | foo
(1 row)
```



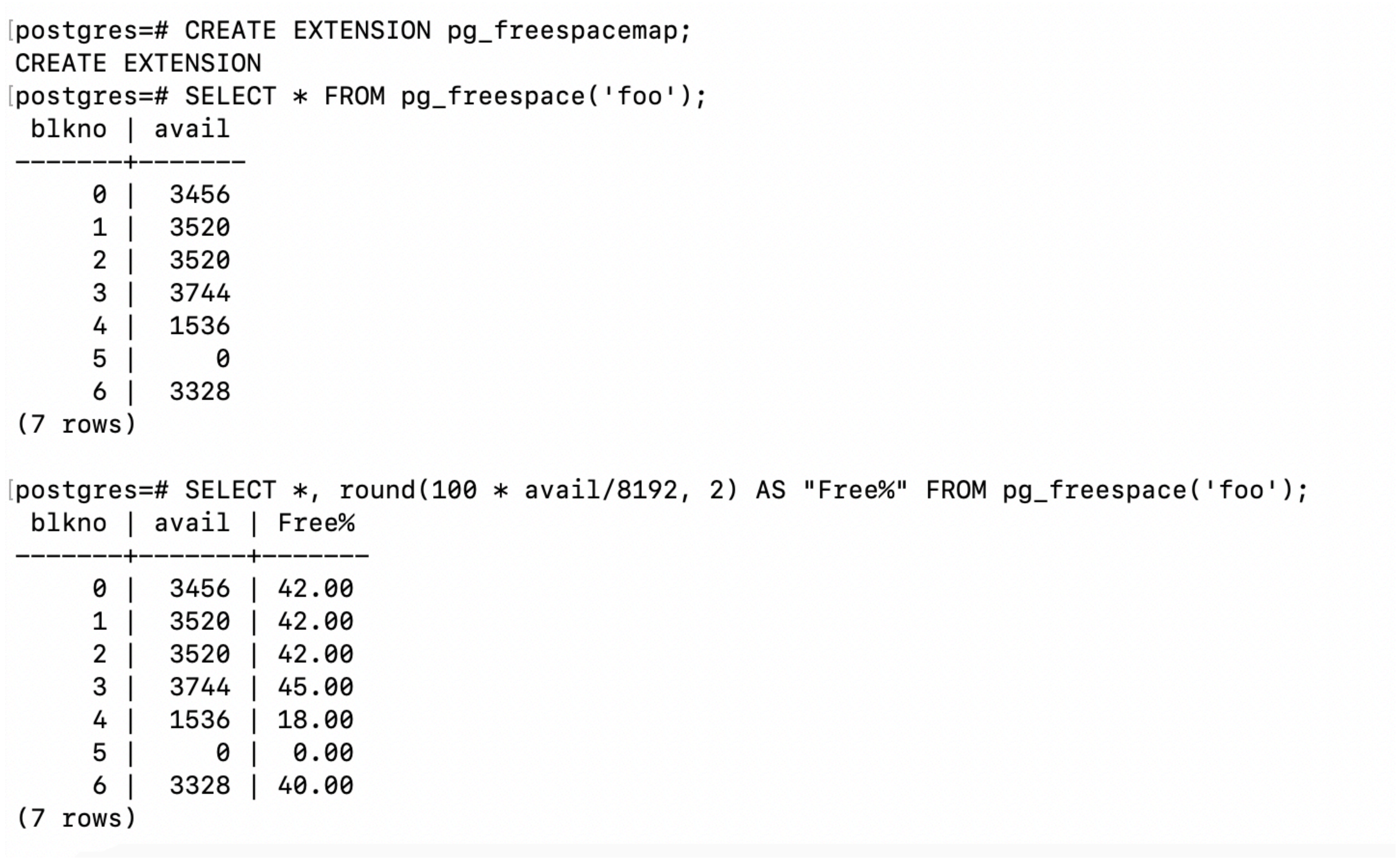
```
[[ec2-user@ip-10-0-4-5 14543]$ cd /var/lib/pgsql/data/base/14543
[[ec2-user@ip-10-0-4-5 14543]$ ls -ltr 16386*
-rw------- 1 postgres postgres 24576 Mar  5 03:57 16386_fsm
-rw------- 1 postgres postgres 57344 Mar  5 04:03 16386
-rw------- 1 postgres postgres  8192 Mar  5 04:03 16386_vm
[[ec2-user@ip-10-0-4-5 14543]$
```

Using the PostgreSQL extension “pg_visibility”, the following is the output for the visibility map of table ‘foo’ that shows pages that have all visible tuples using the ‘all_visible' field and all frozen tuples using the ’all_frozen' field.



```
[postgres=# CREATE EXTENSION pg_visibility;
CREATE EXTENSION
[postgres=# SELECT * FROM pg_visibility_map('foo');
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
     2 | t           | f
     3 | t           | f
     4 | t           | f
     5 | t           | f
     6 | t           | f
(7 rows)
```

PostgreSQL also provides a handy extension pg_freespacemap to determine available free space in bytes in each table. Below is an example output using pg_freespacemap extension to determine the available space per page and percentage for each 8KB page, using the extension's function pg_freespace().



```
[postgres=# CREATE EXTENSION pg_freespacemap;
 CREATE EXTENSION
[postgres=# SELECT * FROM pg_freespace('foo');
  blkno | avail
 -------+-------
      0 |  3456
      1 |  3520
      2 |  3520
      3 |  3744
      4 |  1536
      5 |     0
      6 |  3328
 (7 rows)

[postgres=# SELECT *, round(100 * avail/8192, 2) AS "Free%" FROM pg_freespace('foo');
  blkno | avail | Free%
 -------+-------+-------
      0 |  3456 | 42.00
      1 |  3520 | 42.00
      2 |  3520 | 42.00
      3 |  3744 | 45.00
      4 |  1536 | 18.00
      5 |     0 |  0.00
      6 |  3328 | 40.00
 (7 rows)
```

c) Updates statistics and system catalogs

Vacuum performs post-processing and clean-up, such as truncating the last page of a table if it has no tuples, updating statistics, and system catalogs for each table.

How vacuum solves wraparound

To deal with the transaction ID wraparound problem discussed above, PostgreSQL implemented a process called **Freeze**. Periodic vacuuming solves this problem, as the VACUUM operation is responsible for marking rows as Frozen. The freeze process scans the table for any old rows. In other words, any rows that were inserted before any active transactions, and assigns a flag bit in the metadata indicating that these rows are frozen.

Frozen row versions do not follow the normal XID comparison rules and are always considered older than every normal XID. As a result, they appear to be “in the past” regardless of transaction ID wraparound, and prevent the old XIDs from being mistaken as fresh new XIDs in the future.

Note: In PostgreSQL versions before 9.4, freezing was implemented by actually replacing a rows insertion XID with FrozenTransactionId, which was visible in the rows xmin system column. Newer versions set a flag bit, preserving the rows original xmin for possible forensic use.

If the number of unvacuumed transactions (transactions not subjected to freeze process) reaches (2^31 - 10,000,000), the log starts warning that vacuuming is needed. If the number of unvacuumed transactions reaches (2^31 - 1,000,000), PostgreSQL sets the database to read-only mode to avoid wraparound data loss. The database requires an offline, single-user, standalone vacuum before it can allow writes.

```
2^31 is representing 2 billion.
```

What is Database Age

The age() function in PostgreSQL is used to determine the age or number of transactions that have occurred since a specified XID (transaction ID). In PostgreSQL, the term "Database age" refers to the difference between the current XID and the oldest unfrozen XID recorded in the database. On the other hand, "Table age" refers to the difference between the current XID and the oldest unfrozen XID recorded specifically for a particular table within the database.

Database age is an important metric that we use for monitoring and preventing transaction wraparound issues. Database age can be monitored leveraging the pg_database catalog table. By tracking the database age, you can ascertain if it is nearing the wraparound limit (2 billion). Similarly, we can leverage the pg_class catalog table for monitoring the “Table age”.

The relfrozenxid column of a table's pg_class row is used to calculate the age of a table. The relfrozenxid contains the freeze cutoff XID that was used by the last VACUUM for that table. All normal XIDs older than this cutoff XID (represented as relfrozenxid) are guaranteed to have been frozen.

The datfrozenxid column of a database's pg_database row is used to calculate the age of the database. The datfrozenxid is a lower bound on the unfrozen XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database.

The age column measures the number of transactions from the freeze cutoff XID to the current transaction's XID.

```
age(datfrozenxid) = txid_current() - datfrozenxid
age(relfrozenxid) = txid_current() - relfrozenxid
```

If age(relfrozenxid) exceeds autovacuum_freeze_max_age, an autovacuum will soon be forced for that table.

View the age of each table

```
SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r';
```

View the age of each database

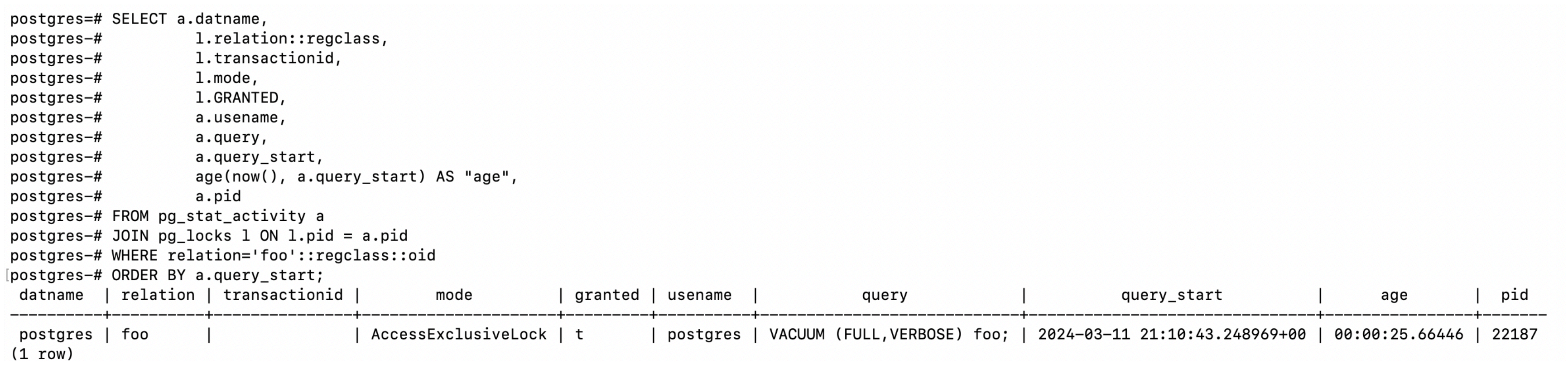
```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

Other types of vacuum

So far, we've discussed standard VACUUM (without FULL). The following tabs cover Vacuum Full and Vacuum Freeze

A standard VACUUM simply reclaims space and makes it available for reuse within the same table by future data inserts. This space is not returned back to the OS.

A VACUUM FULL rewrites the entire contents of the table, indexes into new files, and releases unused space back to OS. As a result, it is important that the database has enough storage equivalent to at least the size of the table, and indexes for VACUUM FULL to complete successfully. This form of vacuum is very restrictive, as it takes an AccessExclusive lock on the table for its entire duration (versus a ShareUpdateExclusive by plain VACUUM). This means that the table is locked during its duration, and no other operations are permitted on the table, as shown in the following image:



```
postgres=# SELECT a.datname,
postgres-#          l.relation::regclass,
postgres-#          l.transactionid,
postgres-#          l.mode,
postgres-#          l.GRANTED,
postgres-#          a.usename,
postgres-#          a.query,
postgres-#          a.query_start,
postgres-#          age(now(), a.query_start) AS "age",
postgres-#          a.pid
postgres-# FROM pg_stat_activity a
postgres-# JOIN pg_locks l ON l.pid = a.pid
postgres-# WHERE relation='foo'::regclass::oid
postgres-# ORDER BY a.query_start;
 datname  | relation | transactionid |        mode         | granted | usename  |           query            |          query_start           |       age       |  pid
----------+----------+---------------+---------------------+---------+----------+----------------------------+--------------------------------+-----------------+-------
 postgres | foo      |               | AccessExclusiveLock | t       | postgres | VACUUM (FULL,VERBOSE) foo; | 2024-03-11 21:10:43.248969+00 | 00:00:25.66446 | 22187
(1 row)
```

An alternative to VACUUM FULL is the pg_repack extension that helps to remove bloat from tables and indexes, and reclaim space. Unlike VACUUM FULL, pg_repack does not require an AccessExclusive lock for its entire duration, meaning INSERTs, UPDATEs, DELETEs, SELECTs are permitted on the table while pg_repack works on the table to reclaim space.

The following AWS Documentation explains how to remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack:
[Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack]

Autovacuum

Autovacuum is a daemon that automates the execution of (standard) VACUUM and ANALYZE (to gather statistics) commands.

Autovacuum parameters

Here, you'll find some parameters that control autovacuum/vacuum behavior in PostgreSQL:

autovacuum_vacuum_threshold	Specifies the minimum number of updates or deletes that must occur on a table before autovacuum vacuums it. This threshold helps prevent autovacuum from unnecessarily triggering for tables that do not have a high rate of these operations. When set to 0, autovacuum triggers for the table based on autovacuum_vacuum_scale_factor setting. To have autovacuum trigger for the table when a specific number of dead tuples have been reached, set autovacuum_vacuum_scale_factor to 0 and autovacuum_vacuum_threshold to the desired number.
autovacuum_vacuum_scale_factor	Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a VACUUM for the table. For a table that is highly write intensive, lower the value of this parameter at a table level using the ALTER TABLE…SET command, to allow autovacuum to trigger more frequently and handle bloat on the table.
autovacuum_analyze_threshold	Specifies the minimum number of inserted, updated or deleted tuples needed to trigger an ANALYZE in any one table.
autovacuum_analyze_scale_factor	Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a ANALYZE for the table. This parameter in conjunction with autovacuum_analyze_threshold determines how often statistics are gathered for the table. Setting these values very high can lead to stale statistics and poor performance.
autovacuum_vacuum_insert_scale_factor	Specifies a fraction of the table size to add to autovacuum_vacuum_insert_threshold when deciding whether to trigger a VACUUM.
autovacuum_vacuum_insert_threshold	Specifies the number of inserted tuples needed to trigger a VACUUM in any one table.
autovacuum_naptime	Specifies the minimum delay between autovacuum runs on any given database. In each round the daemon examines the database and issues VACUUM and ANALYZE commands as needed for tables in that database.

autovacuum_max_workers	Maximum number of autovacuum processes which may be running at any one time. Each worker process works on a single table at a time. With a database that has multiple tables eligible for vacuum, increasing **autovacuum_max_workers** can allow for multiple tables to be autovacuumed simultaneously. However, note that this can drive increased resource usage on the instance.
autovacuum_vacuum_cost_limit	Total IO cost limit Autovacuum could reach. Note that the value is distributed proportionally among the running autovacuum workers if there is more than one, so that the sum of the limits for each worker does not exceed the value of this variable.
autovacuum_vacuum_cost_delay	Autovacuum will sleep for these many milliseconds after reaching the cost limit.
vacuum_freeze_min_age	Specifies the cutoff age (in transactions) that VACUUM should use to decide whether to freeze row versions while scanning a table. The default is 50,000,000 transactions. vacuum_freeze_min_age governs whether or not a tuple will be frozen while vacuum is already looking at a page, to see if it has dead tuples that can be cleaned up. Tuples older than vacuum_freeze_min_age will be frozen in this case.
vacuum_freeze_table_age	VACUUM performs an aggressive scan if the table's pg_class.relfrozenxid field has reached the age specified by this setting. An aggressive scan differs from a regular VACUUM in that it visits every page that might contain unfrozen XIDs or MXIDs, not just those that might contain dead tuples. The default is 150 million transactions. Setting vacuum_freeze_table_age to 0 forces VACUUM to use this more aggressive strategy for all scans.
autovacuum_freeze_max_age	Specifies the maximum age (in transactions) that a table's pg_class.relfrozenxid field can attain before a VACUUM operation is forced to prevent transaction ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled. Default is 200,000,000 transactions.

Adaptive autovacuum(rds.adaptive_autovacuum)

Adaptive autovacuum parameter tuning is a feature for RDS PostgreSQL and Aurora PostgreSQL. This feature is enabled by default with the dynamic parameter rds.adaptive_autovacuum set to ON. It is strongly recommended to have this enabled.

When adaptive autovacuum parameter tuning is turned on, Amazon RDS begins adjusting autovacuum parameters when the CloudWatch metric MaximumUsedTransactionIDs reaches the value of the autovacuum_freeze_max_age parameter, or 500,000,000, whichever is greater.

Amazon RDS updates the following autovacuum-related parameters:

- autovacuum_vacuum_cost_delay
- autovacuum_vacuum_cost_limit
- autovacuum_work_mem
- autovacuum_naptime

Amazon RDS continues adjusting autovacuum parameters, and dedicates more resources to autovacuum to avoid wraparound if a table continues to trend toward transaction ID wraparound. The parameters are modified in memory of the DB instance. The values in the parameter group aren't changed.

After the MaximumUsedTransactionIDs CloudWatch metric returns below the threshold, Amazon RDS resets the autovacuum-related parameters in memory back to the values specified in the parameter group.

Monitoring and Dynamic Actions

Information about autovacuum activities is sent to the postgresql.log based on various parameter configurations, which are discussed below.

log_autovacuum_min_duration:
log_autovacuum_min_duration parameter's value is the threshold (in milliseconds) above which autovacuum actions get logged. A setting of -1 logs nothing, while a setting of 0 logs all actions.

rds.force_autovacuum_logging_level:

rds.force_autovacuum_logging_level parameter defines the verbosity of the autovacuum messages logged to the postgres error log file. Values include disabled (PostgreSQL 10, PostgreSQL 9.6), debug5, debug4, debug3, debug2, debug1, info (PostgreSQL 12, PostgreSQL 11), notice, warning (PostgreSQL 13 and above), error, log, fatal, panic.

When troubleshooting autovacuum related issues, set rds.force_autovacuum_logging_level parameter to WARNING or one of the debug levels, from debug1 up to debug5 for the most verbose information, and set log_autovacuum_min_duration to a value other than -1 to log messages, such as when the autovacuum action is skipped because of a conflicting lock or concurrently dropped relations.

Use debug settings for short periods of time and for troubleshooting purposes only, as this can increase the size of error log files and exhaust local storage space on Aurora clusters and allocated storage on RDS instances.

The following AWS Documentation explains logging autovacuum and vacuum activities in further detail:

Working with the PostgreSQL autovacuum on Amazon RDS for PostgreSQL - Logging autovacuum and vacuum activities

How to troubleshoot common scenarios

Autovacuum/vacuum running long for table (Slow vacuum)

When autovacuum runs long for a table, it’s typically because the maintenance_work_mem (or autovacuum_work_mem) is insufficient for the size of the table and/or table has large indexes.

To approach how to troubleshoot, the following queries can be used:

Check which table autovacuum is working on from pg_stat_activity using the following query:

```
SELECT datname, usename, pid, state, wait_event, current_timestamp - xact_start AS xact_runtime,
query
FROM pg_stat_activity
```

```
WHERE upper(query) LIKE '%VACUUM%'
ORDER BY xact_start;
```

After the table is identified, check if any table level settings are in place using the following query:

```
SELECT relname, reloptions FROM pg_class WHERE relname='tablename';
```

Check the value of maintenance_work_mem, and increase the value to 1GB if the instance has enough memory. This critical parameter determines how much tuple information the autovacuum worker can hold in memory while it's processing tables. If this parameter is too small for a table, it will require multiple passes on the table to complete.

Note: Note that for the collection of dead tuple identifiers, VACUUM is only able to utilize up to a maximum of 1GB of memory.

Consider setting autovacuum_cost_delay to 0. Autovacuum will take a nap of autovacuum_vacuum_cost_delay ms every time the internal counter hits auto_vacuum_cost_limit. For a large table, it can be very beneficial to have autovacuum take no "breaks" at the expense of additional resource consumption. You could set it to 0 at table level using the ALTER TABLE command:

```
ALTER TABLE mytable SET (autovacuum_vacuum_cost_delay=0);
```

Use the following query to identify vacuum progress:

```
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;
```

If you see that autovacuum is spending most of its time vacuuming indexes, you could consider dropping these indexes to speed up autovacuum on the table.

Note:

For VACUUM FULL progress reporting, consider using the pg_stat_progress_cluster view.

To check the size of the table and its indexes:

```
postgres=> select pg_size_pretty(pg_relation_size('pgbench_accounts'));
postgres=> select pg_size_pretty(pg_indexes_size('pgbench_accounts'));
```

For RDS PostgreSQL 11 and lower versions, the only way to allow vacuum to complete faster is to reduce the number of indexes on a table. Dropping an index can affect query plans. We recommend that you drop unused indexes first, then drop the indexes when XID wraparound is very near. After the vacuum process completes, you can recreate these indexes.

For RDS for PostgreSQL 12 and higher - you can use VACUUM with the INDEX_CLEANUP clause. INDEX_CLEANUP can be set to OFF to force VACUUM to *always* skip index vacuuming, even when there are many dead tuples in the table. This may be useful when it is necessary to make VACUUM run as quickly as possible to avoid imminent transaction ID wraparound.

```
postgres=> VACUUM (INDEX_CLEANUP FALSE, VERBOSE TRUE) tablename;
```

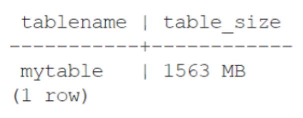
Troubleshooting

Slow vacuum due to low maintenance_work_mem

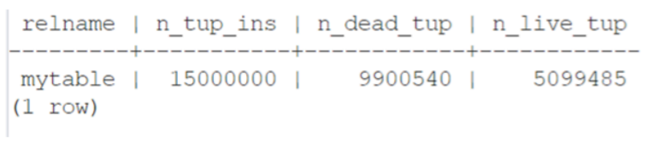
Now, let's take a look at a scenario that deals with slow vacuum due to low maintenance_work_mem.

Step 1: Understand the scenario

- The table 'mytable' has a size of 1563 MB with 9900540 dead rows.
- The current value of the maintenance_work_mem parameter is set to 5 MB.



```
 tablename | table_size
-----------+------------
 mytable   | 1563 MB
(1 row)
```



```
 relname | n_tup_ins | n_dead_tup | n_live_tup
---------+-----------+------------+------------
 mytable |  15000000 |    9900540 |    5099485
(1 row)
```

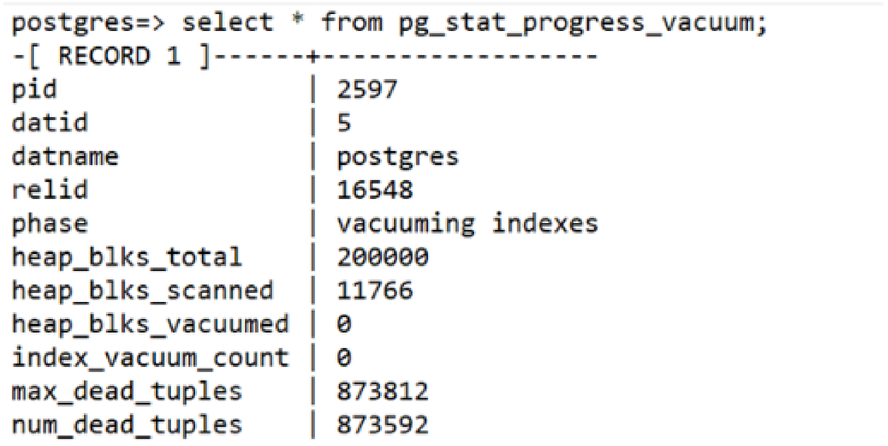
Step 2: Understand the maintenance_work_mem parameter

- The maintenance_work_mem parameter determines the maximum memory that can be used for maintenance operations like VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY.
- PostgreSQL splits the data requiring vacuuming into smaller work batches during the VACUUM process and processes each batch independently.
- The maintenance_work_mem parameter specifies the maximum amount of memory that can be used per work batch.
- A smaller value for maintenance_work_mem can cause less data processing and sorting to happen in memory, slowing down the overall vacuum operation.

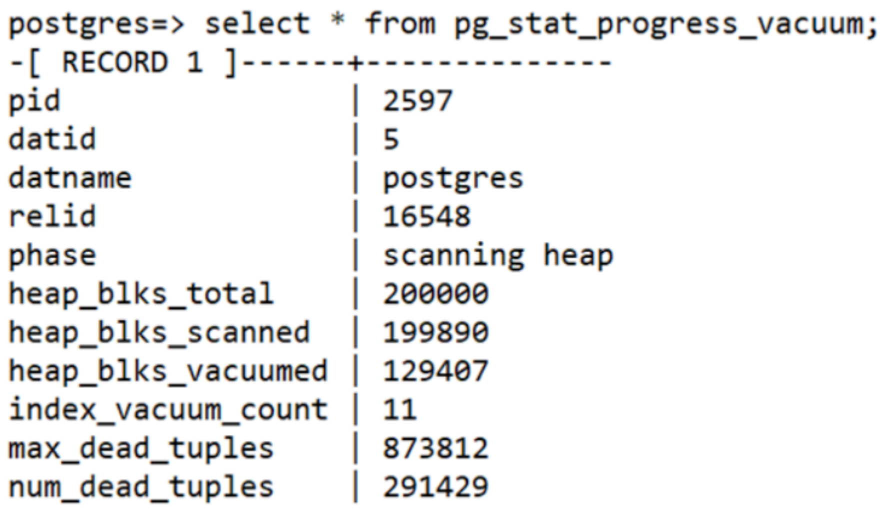
Step 3: Observe the initial vacuum behavior

- With maintenance_work_mem set to 5 MB, the pg_stat_progress_vacuum view shows a higher index_vacuum_count value of 11.
- The "vacuuming indexes phase" indicates that the VACUUM process is currently cleaning up the indexes associated with a table. If a table has one or more indexes defined, this phase will occur at least once during each VACUUM operation, after the main data file (heap) has been fully scanned. However, if the allocated memory specified by the maintenance_work_mem parameter (or autovacuum_work_mem for autovacuum) is insufficient to hold all the dead or obsolete entries found in the indexes, the "vacuuming indexes phase" may need to happen multiple times within a single VACUUM operation to complete the index cleanup process, which can increase overall vacuum time.
- The vacuum took 171038.295 ms (approximately 2.86 minutes) to complete, and the overall resource usage was high.

```
postgres=> SET maintenance_work_mem= '5 MB';
SET
Time: 83.715 ms
postgres=> VACUUM VERBOSE mytable;
INFO:  vacuuming "postgres.public.mytable"
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  finished vacuuming "postgres.public.mytable": index scans: 12
pages: 0 removed, 200000 remain, 200000 scanned (100.00% of total)
tuples: 9900540 removed, 5099460 remain, 0 are dead but not yet removable
removable cutoff: 12903, which was 2 XIDs old when operation ended
new relfrozenxid: 12903, which is 58 XIDs ahead of previous value
frozen: 111855 pages from table (55.93% of total) had 5064150 tuples frozen
index scan needed: 133334 pages from table (66.67% of total) had 9900540 dead item identifiers removed
index "mytable_col1_idx": pages: 141250 in total, 0 newly deleted, 0 currently deleted, 0 reusable
index "mytable_col2_idx": pages: 141013 in total, 0 newly deleted, 0 currently deleted, 0 reusable
index "mytable_col3_idx": pages: 11848 in total, 7684 newly deleted, 7684 currently deleted, 6095 reusable
index "mytable_col4_idx": pages: 59555 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 48409.654 ms, write: 9011.155 ms
avg read rate: 44.338 MB/s, avg write rate: 48.833 MB/s
buffer usage: 1258814 hits, 969717 misses, 1068037 dirtied
WAL usage: 4822263 records, 546838 full page images, 2061222519 bytes
system usage: CPU: user: 43.93 s, system: 7.83 s, elapsed: 170.86 s
VACUUM
Time: 171038.295 ms (02:51.038)
```



```
postgres=> select * from pg_stat_progress_vacuum;
-[ RECORD 1 ]------+-----------------
pid                | 2597
datid              | 5
datname            | postgres
relid              | 16548
phase              | vacuuming indexes
heap_blks_total    | 200000
heap_blks_scanned  | 11766
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuples    | 873812
num_dead_tuples    | 873592
```



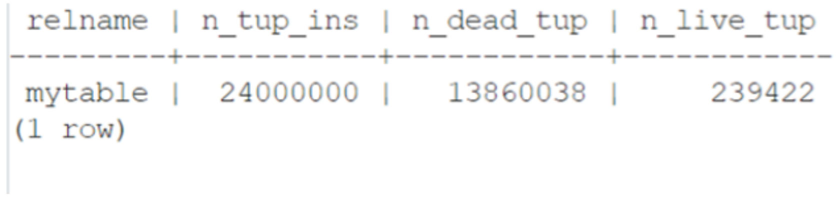
```
postgres=> select * from pg_stat_progress_vacuum;
-[ RECORD 1 ]------+--------------
pid                | 2597
datid              | 5
datname            | postgres
relid              | 16548
phase              | scanning heap
heap_blks_total    | 200000
heap_blks_scanned  | 199890
heap_blks_vacuumed | 129407
index_vacuum_count | 11
max_dead_tuples    | 873812
num_dead_tuples    | 291429
```

Step 4: Increase the maintenance_work_mem value

- Increase the value of maintenance_work_mem to 500 MB.

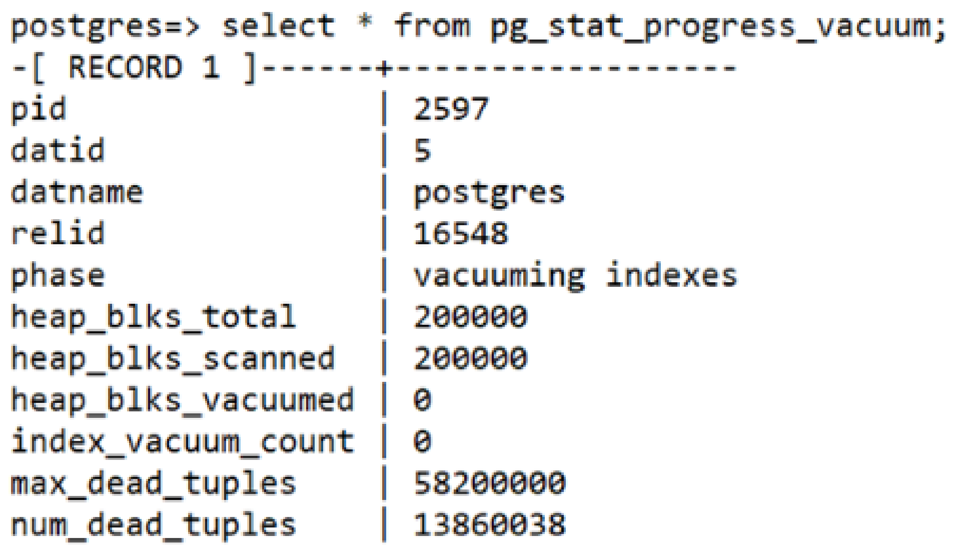
Step 5: Observe the vacuum behavior after increasing maintenance_work_mem

- After increasing maintenance_work_mem to 500 MB, the vacuum got completed in 44110.197 ms (approximately 0.73 minutes), which is significantly faster than the initial run.
- The index_vacuum_count was reduced to 1, indicating that vacuum stored more dead tuples in memory per cycle.

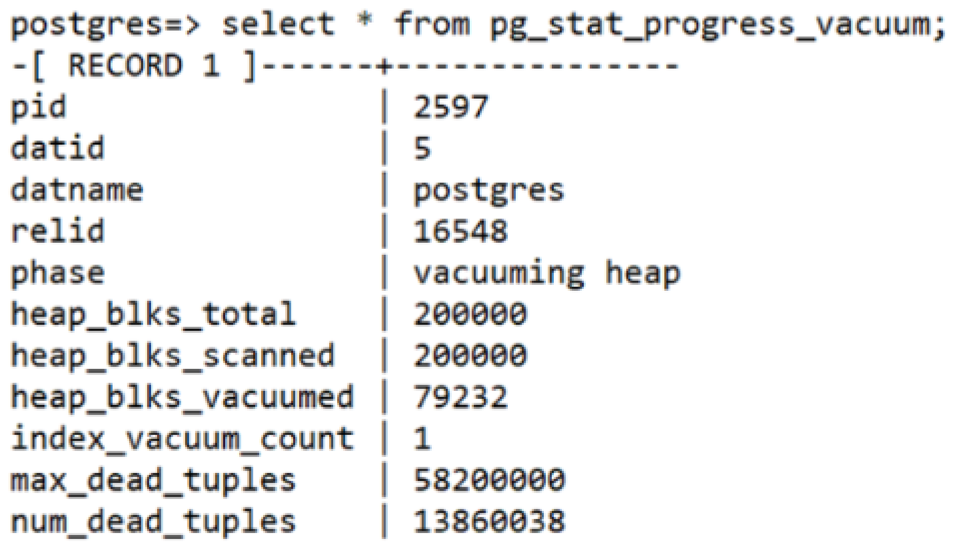


```
 relname | n_tup_ins | n_dead_tup | n_live_tup
---------+-----------+------------+------------
 mytable |  24000000 |   13860038 |     239422
(1 row)
```

```
postgres=> SET maintenance_work_mem= '500 MB';
SET
Time: 105.951 ms
postgres=>  VACUUM VERBOSE mytable;
INFO:  vacuuming "postgres.public.mytable"
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  finished vacuuming "postgres.public.mytable": index scans: 1
pages: 0 removed, 200000 remain, 187872 scanned (93.94% of total)
tuples: 13860038 removed, 539544 remain, 0 are dead but not yet removable
removable cutoff: 12931, which was 2 XIDs old when operation ended
new relfrozenxid: 12911, which is 8 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 187872 pages from table (93.94% of total) had 13860038 dead item identifiers removed
index "mytable_col1_idx": pages: 143207 in total, 27294 newly deleted, 27294 currently deleted, 16580 reusable
index "mytable_col2_idx": pages: 142936 in total, 26981 newly deleted, 26981 currently deleted, 16404 reusable
index "mytable_col3_idx": pages: 19243 in total, 17390 newly deleted, 18979 currently deleted, 1589 reusable
index "mytable_col4_idx": pages: 59768 in total, 2008 newly deleted, 2008 currently deleted, 1143 reusable
I/O timings: read: 22025.520 ms, write: 1457.785 ms
avg read rate: 47.622 MB/s, avg write rate: 61.929 MB/s
buffer usage: 702727 hits, 268045 misses, 348571 dirtied
WAL usage: 1071226 records, 77347 full page images, 271230746 bytes
system usage: CPU: user: 14.27 s, system: 2.55 s, elapsed: 43.97 s
VACUUM
Time: 44110.197 ms (00:44.110)
postgres=>
```



```
postgres=> select * from pg_stat_progress_vacuum;
-[ RECORD 1 ]------+-----------------
pid                | 2597
datid              | 5
datname            | postgres
relid              | 16548
phase              | vacuuming indexes
heap_blks_total    | 200000
heap_blks_scanned  | 200000
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuples    | 58200000
num_dead_tuples    | 13860038
```



```
postgres=> select * from pg_stat_progress_vacuum;
-[ RECORD 1 ]------+--------------
pid                | 2597
datid              | 5
datname            | postgres
relid              | 16548
phase              | vacuuming heap
heap_blks_total    | 200000
heap_blks_scanned  | 200000
heap_blks_vacuumed | 79232
index_vacuum_count | 1
max_dead_tuples    | 58200000
num_dead_tuples    | 13860038
```

Key Points:

By increasing the maintenance_work_mem parameter, PostgreSQL can allocate more memory for each work batch during the VACUUM process. This allows more data processing and sorting to happen in memory, resulting in a faster and more efficient vacuum operation. It's important to note that setting maintenance_work_mem too high can lead to excessive memory usage and potentially degrade overall system performance. Therefore, it's recommended to monitor the system's memory usage and adjust the maintenance_work_mem value accordingly, considering the available system resources and the specific workload requirements.

Test Your Knowledge

Troubleshooting Slow Vacuum

Vacuum Not Removing Dead Tuples

Autovacuum Not Triggering for Table

Troubleshooting Transaction Wraparound Issues

Resources

Working with the PostgreSQL autovacuum on Amazon RDS for PostgreSQL

Troubleshooting Autovauum "Not Runing", "Slow Running", "Blocked" issues in RDS PostgreSQL

Title: Vacuum/Autovacuum not cleaning bloat (dead row versions cannot be removed yet)

Autovacuum not deleting dead tuples in Aurora Postgres

Implement an Early Warning System for Transaction ID Wraparound in Amazon RDS for PostgreSQL

Understanding autovacuum in Amazon RDS for PostgreSQL environments

PostgreSQL Concurrency with MVCC

20.10. Automatic Vacuuming

VACUUM - SQL Commands

20.8. Error Reporting and Logging - Server Configuration

Logging autovacuum and vacuum activities

VACUUM Phases

[Implement an Early Warning System for Transaction ID Wraparound in Amazon RDS for PostgreSQL]

[Managing autovacuum with large indexes]

[Reindexing a table when autovacuum is running]

[Four reasons why vacuum won't remove dead rows from a table]

[When autovacuum does not vacuum]

[Implement an Early Warning System for Transaction ID Wraparound in Amazon RDS for PostgreSQL]