Date: 21 June 2021

ASHNIK

Everything you need to know about PostgreSQL Database Maintenance Getting Inside 'VACUUM'

Introduction

We often hear the term 'database maintenance'. So what exactly is it?
Everything requires effective maintenance, including your database. Regular maintenance helps it run and perform efficiently to meet your business expectations. Database Maintenance describes a set of tasks that are run with the intention to improve your database. There are routines meant to help performance, free up disk space, check for data errors, hardware faults, update internal stats, and many other indistinct (but generally important) things.

Database maintenance is a highly neglected topic in daily PostgreSQL operation. While it is a common understanding that database backups need to be done regularly and essentially, only a few users are aware of the extra work that goes into it. One of the main reasons, you don't see many database maintenance works is the lack of in-depth knowledge of SQL itself, and being able to carry out the tasks with efficient timelines.

At times, a database maintenance not done properly may go unnoticed, but when the database performance is hit; is when it turns into a real issue.

THIS WHITEPAPER WILL TAKE YOU THROUGH THE COMMON DATABASE MAINTENANCE TASKS IN POSTGRESQL

Key Highlights:

1. Vacuum is processed in PostgreSQL which does a clean-up job of dead rows or dead tuples. It is like defragmentation activity of dead rows/tuples or commonly known as bloat.

2. PostgreSQL maintains the old tuples versioning for the visibility in transaction via MVCC. Hence, it will not remove immediately and due to the MVCC functionality it keeps those versions of data unless someone tells it to remove.

3. There is an automatic vacuum process in PostgreSQL, but many times for specific load, DBA prefers to schedule this for better performance.

4. If someone does not vacuum frequently, then database performance will go down as versions of data tuples increase. In fact, after a few days/months, it may crash due to transaction wraparound.

About Postgres VACUUM

PostgreSQL is a low-maintenance database compared to most other database management systems. Nonetheless, appropriate attention to maintenance tasks will go far towards ensuring a pleasant and productive experience with the system.

In PostgreSQL, an operation like UPDATE or DELETE on a row does not immediately remove the old version of the row. If you have an application that performs many UPDATE/DELETE operations frequently in your database, it can grow quickly and need maintenance activity periodically.

Generally, in the PostgreSQL database, maintenance activity is performed periodically, known as vacuuming. Its two main tasks are removing dead rows and freezing transaction IDs.

The vacuum process provides two modes to remove the dead rows; one is **VACUUM** and the other is **VACUUM FULL.**

VACUUM operation is responsible for removing the dead rows for each page of the table file, and other transactions can read the table while this process is running.

While on the other hand, Full VACUUM removes dead rows and defragments live rows of the whole file, and other transactions like any DDL, DML, and Select cannot access those tables while Full VACUUM is running.

VACUUM:

A vacuum is used for recovering space occupied by “dead tuples” in a table. A dead tuple is created when a record is either deleted or updated (a delete followed by an insert). PostgreSQL doesn’t physically remove the old row from the table but puts a “marker” on it so that queries don’t return that row. When a vacuum process runs, the space occupied by these dead tuples is marked reusable by other tuples.

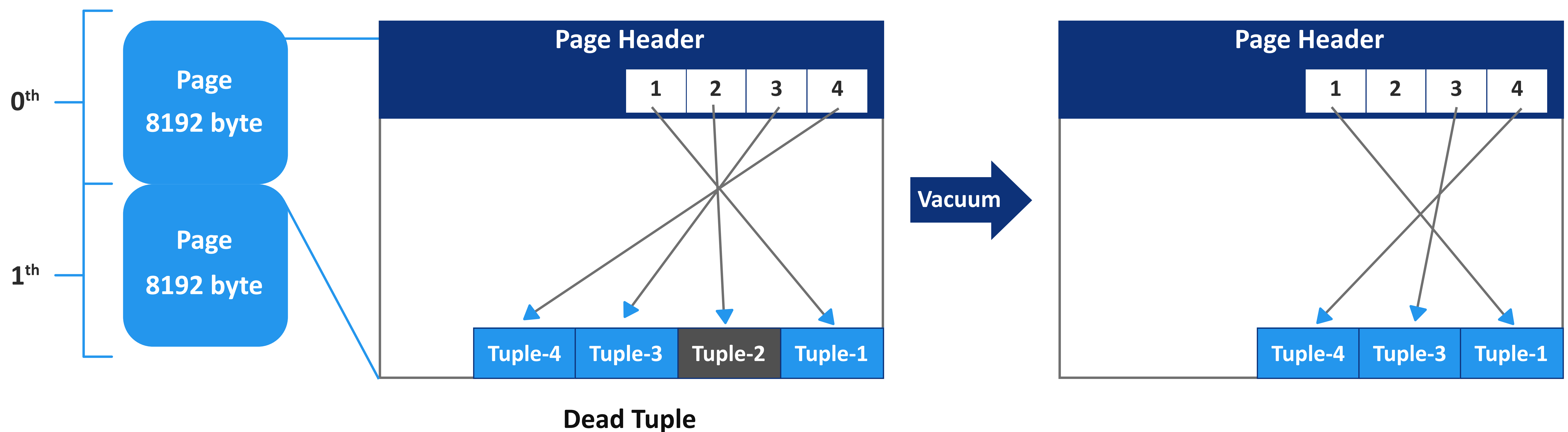
Vacuum processing performs the following tasks for specified tables or all tables in a database.

1. Get each table from the specified tables.
2. Acquire “**ShareUpdateExclusiveLock**” lock for the table. This lock allows reading from other transactions.
3. Scan all pages to get all dead tuples and freeze old tuples if necessary.
4. Remove the index tuples that point to the respective dead tuples if they exist.
5. Perform the following tasks, steps (6) and (7), for each page of the table.
6. Remove the dead tuples and defragment the live tuples on the page.
7. Update both the respective FSM and VM of the target table.
8. Clean up the indexes using the in-built function.
9. Truncate the last page if the last one does not have any tuple.
10. Update both the statistics and the system catalogs related to vacuum processing for the target table.
11. Release “**ShareUpdateExclusiveLock**” lock.
12. Update both the statistics and the system catalogs related to vacuum processing.
13. Remove both unnecessary files and pages of the clog, if possible.

There are two new terms for you: Free Space Map (FSM) and Visibility Map (VM). Let me give you a brief explanation of these terms.

- **Free Space Map (FSM):**
  In PostgreSQL, each table and index have an FSM to keep track of available space. It stores all free space-related information alongside primary relation, and that relationship starts with the file node number plus the suffix _fsm.

- **Visibility Map (VM):**
  The Visibility Map associates with each table and index and uses to keep track of which pages contain only tuples that are known to be visible to all active transactions. It stores in separate relation alongside the main relation, and it starts with the file node number plus a suffix _vm.



For example, as you can see in the above diagram, one table has two pages. Let us concentrate on the 0th page, which is the first page. This page has four tuples. Tuple-2 is a dead tuple. In this case, PostgreSQL removes Tuple-2 and reorders the remaining tuples to repair fragmentation/bloat, and then updates both the FSM and VM of this page. PostgreSQL continues this process until the last page.

PostgreSQL-13 onward for VACCUM operation PARALLEL option is introduced, which helps to run index vacuuming and cleaning parallelly.

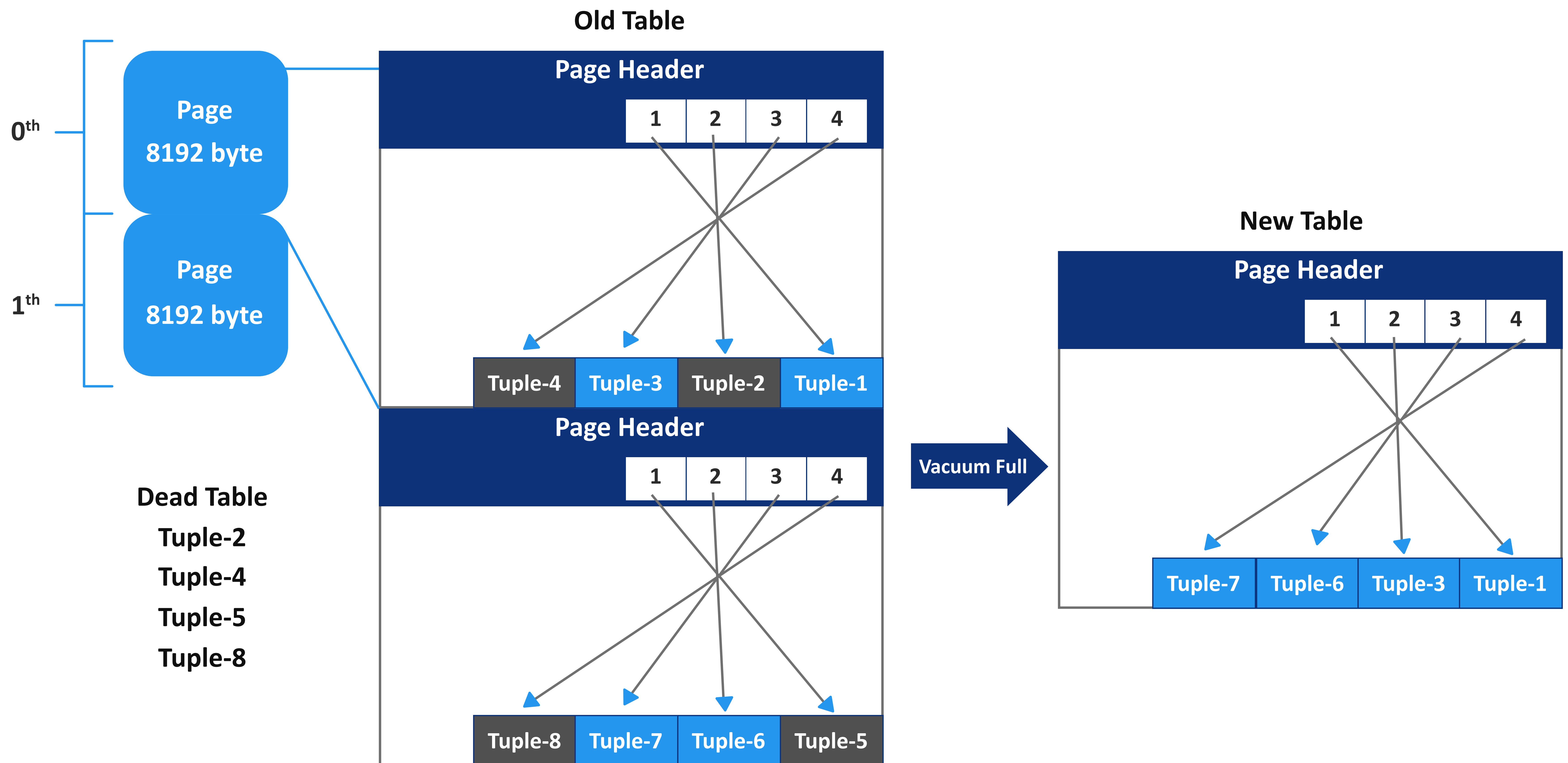
Please note that the PARALLEL option is only useful when there are at least two indexes in the table.

We will explain the freezing process later in this paper.

VACUUM FULL:

Vacuum Full process performs the following tasks for specified tables or all tables in the database.

1. Get each table from the specified tables.
2. Acquire an "**AccessExclusiveLock**" lock for the table. This lock does not allow reading and writing from any other transactions.
3. Creates a new table file whose size is 8 KB.
4. PostgreSQL copies only live tuples within the old table file to the new table.
5. Remove the old table file.
6. Rebuild all indexes
7. Update both the statistics and the system catalogs.
8. Release "**AccessExclusiveLock**" lock
9. Remove unnecessary clog files and pages if possible.



For example, as you can see in the above diagram, one table have two pages. 0th page has four tuples. Tuple-2 & Tuple-4 are dead tuples. 1st page also has another four tuples, and Tuple-5 & 8 are dead tuples. When we execute the Vacuum Full command, PostgreSQL starts removing the dead tuples, like first acquiring the "**AccessExclusiveLock**" lock for that table and creates a new table file whose size is 8 KB. After that, PostgreSQL copies only live tuples within the old table file to the new table, and after copying all live tuples, PostgreSQL removes the old file, rebuilds all associated table indexes, updates associated statistics and system catalogs.

Please note that due to "**AccessExclusiveLock**", no one can access the table when Full VACUUM is running, and this method also requires extra disk space temporarily since it writes a new copy of the table and does not release the old copy until the operation is complete. Disk space required the double size of the table. Vacuum Full helps you to reclaim free space from a table to a disk.

Now immediately question comes to mind: What if disk space is about to the full and cannot increase the size quickly. So, how we can proceed in that situation? Then I would recommend the below approaches.

- **Approach-1:**
  It is always good to start with small-sized tables which can easily accommodate the available disk space. So, slowly you will get freer disk space, and then there is a high possibility of getting enough disk space to accommodate those tables that were initially not eligible for VACUUM FULL due to less disk space and high table size.

- **Approach-2:**
  For VACUUM FULL, drop all the indexes of that table on which you want to perform VACUUM FULL and then execute the VACUUM FULL. Once this operation is completed successfully, then recreate all the dropped indexes on that table.

If the above two approaches are not workable, then, in that case, you must increase the disk space, which is the only feasible option.

Now, let's understand the concept of Vacuum Freeze, but before proceeding with this, we should know about is Transaction ID Wraparound issue in PostgreSQL.

What is the Transaction ID Wraparound issue in PostgreSQL?

In PostgreSQL, the transaction control mechanism assigns a transaction ID (Txid) to every row that is modified in the database; these IDs control the visibility of that row to other concurrent transactions.

Transaction Wraparound is a problem due to Multi-Version Concurrency Control (MVCC). MVCC relies on taking the Txids of two transactions and determining which of the transactions came first. In Postgres, Txids are only 32-bit integers long. That means there are only 232, or we can say about four billion, possible Txids only. Honestly, four billion may sound like a lot, but workloads can reach four billion transactions within a few days or weeks with very high write volume.

Now, an instant question comes to mind that what will happen once PostgreSQL reaches 4 billion transactions?

So, the answer is that PostgreSQL assigns Txids sequentially from a cycle. The cycle goes back to zero and looks something like 0, 1, 2, ..., 232-1, 0, 1, ... To determine which two Txids are older. So, under this logic, Postgres has to make sure that all Txids currently in use is within a range of 231 of each other. That way, all of the Txids in use form a consistent ordering.

PostgreSQL also ensures that only a valid range of Txids is used by regularly removing all the old Txids. If the old Txids are not cleared regularly, then there will be a new Txid that is newer than the newest Txids and simultaneously appears older than the oldest Txids then; this is known as Transaction Wraparound.

In this case, PostgreSQL will terminate normal operations to prevent data corruption and stop accepting new transactions request, which causes downtime.

To clear out old txids automatically, Postgres uses a special vacuum. Autovacuum adds this "**to prevent wraparound**" message to the process as below.

Please note that if you see this message in your environment, you cannot stop the autovacuum process even if you
set **autovacuum=off** in **postgresql.conf** or **postgresql.auto.conf** files.

VACUUM FREEZE:

Now, we back to our original point, which is Vacuum Freeze. Autovacuum process does take care of freezing the table transaction ID and replace it with Frozen Txid to avoid Transaction ID Wraparound failure. In simple words, we can say the frozen Txid is always inactive and visible.

As a proactive measure, DBA needs to monitor the tables to ensure that Txid does not get exhausted for the large tables where autovacuum process cannot keep up with vacuuming the frequently accessed tables.

Below SQL command helps the DBA monitor the oldest Txid age of databases and the current setting for the database.

```
SELECT datname, age(datfrozenxid), current_setting('autovacuum_freeze_max_age')FROM pg_database
ORDER BY 2 DESC;
```

If any specific database approaches the **freeze_max_age** value, then the below query should be executed connecting to that specific database. The query gives the list of tables with the oldest transaction ID that should be vacuum freeze manually.

```
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;
```

Now, as you know, if you want to perform the Vacuum Full in your heavily busy production environment, you have to pay for the long overhead of “AccessExclusiveLock”. So, to overcome this, many enterprise-level customers are using the “pg_repack” extension.

pg_repack:

pg_repack is a PostgreSQL extension tool that can do pretty much what FULL VACUUM does, like remove bloat from tables and indexes and restore the physical order of clustered indexes. Unlike VACUUM FULL, it works online without holding

an “**AccessExclusiveLock**” lock on the processed tables during processing.
pg_repack process performs the following tasks for specified tables or all tables in the database.

1. Create a log table to record changes made to the original table
2. Add a trigger onto the original table, logging INSERTs, UPDATEs and DELETEs into our log table
3. Create a new table containing all the rows in the old table
4. Build indexes on this new table
5. Apply all changes which have accrued in the log table to the new table
6. Swap the tables, including indexes and toast tables, using the system catalogs
7. Drop the original table

pg_repack will only hold an “**ACCESSEXCLUSIVELOCK**” lock for a short period during initial steps (steps 1 and 2 as above) and during the final swap-and-drop phase (steps 6 and 7).
For the rest of its time, pg_repack only needs to hold an “**ACCESSSHARELLOCK**” lock on the original table, meaning SELECTs, INSERTs, UPDATEs, and DELETEs may proceed as usual.

I hope this paper helps you to understand the concept of vacuum and to planned maintenance activity effectively.

If you have any queries or concerns regarding your Postgres Database management, reach us on success@ashnik.com and our technical experts will be happy to guide you.

Visit: www.ashnik.com to know more