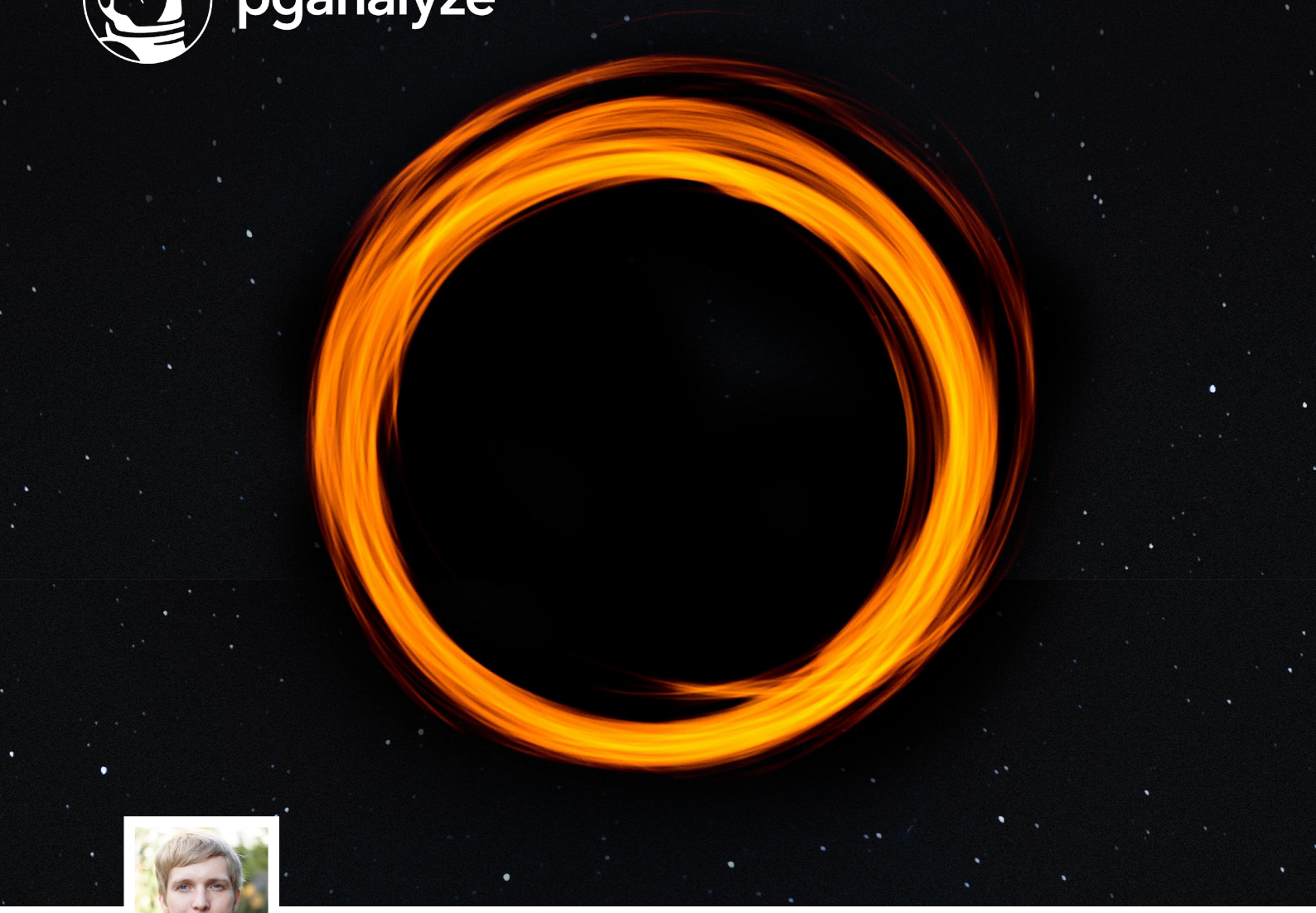
pganalyze



LUKAS FITTL

Tuning autovacuum for best Postgres performance

Tuning autovacuum for best Postgres performance

VACUUM in Postgres is a fact of life - you can't run Postgres without it, and every Postgres installation out there has to vacuum each table in the database at least once every 2 billion transactions.

Luckily we don't have to do this vacuuming ourselves. Postgres will automatically schedule autovacuum to run based on certain triggers, either based on tuple thresholds, or the age of transaction IDs referenced in pages of the table.

At the end of this book we'll also discuss when it may make sense to run manual vacuums in addition to autovacuum, and other ways to reduce the impact of autovacuum during business hours.

In this eBook we'll walk you through the most important concepts of why and when Postgres has to vacuum, why that matters, and when it's best to tune the default settings that control autovacuum scheduling, vacuum overhead, and more. We'll provide actionable tuning tips you can use in your specific setup.

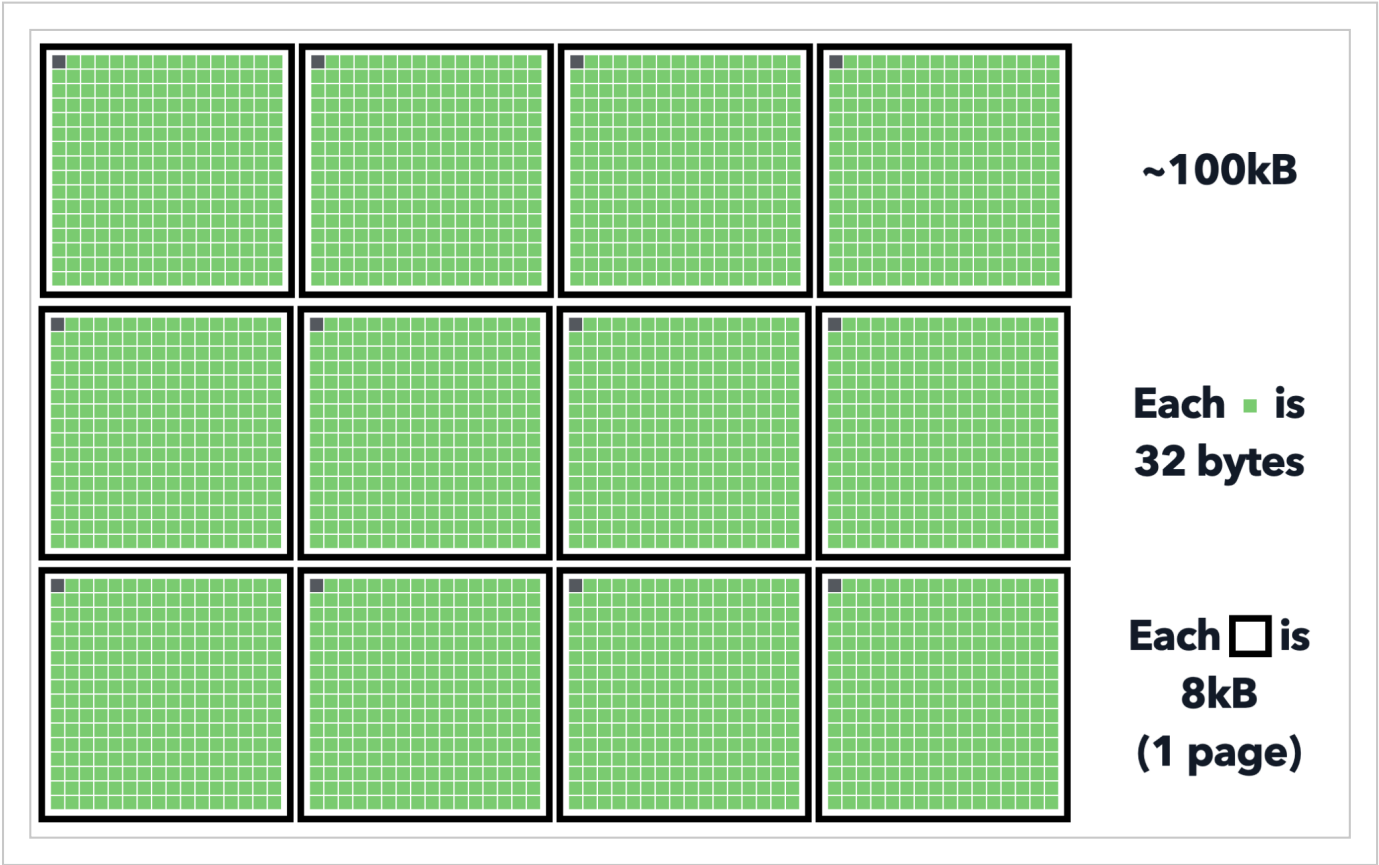
ABOUT

All of these concepts are what led us at pganalyze to create the pganalyze VACUUM Advisor - an automated tool that detects common autovacuum problems, and points out potential configuration tuning opportunities. The advice contained in this eBook is applicable whether you use pganalyze or not - but a lot of the manual work, data collection and identifying problems can be done by pganalyze for you.

But now, let's dive right in. One of the challenges when working with VACUUM is making sense of why we have it in the first place. Let's start by looking at **bloat**.

What is (Table) Bloat?

In the most basic sense, bloat refers to empty space in a table. This is oftentimes further qualified by saying, bloat is empty space that is not actually needed - i.e. our table is larger than it should be.



Let's take an example. Imagine you insert 10000 rows of data into Postgres, and each has about 1 kB of data in them. That will cause Postgres to extend the table to be 10 MB in size, in order to fit this data. Now imagine we delete 5000 of these rows.

Effectively we are only using 5 MB of our table now. The deleted 5 MB of this data can be considered "unvacuumed bloat", also referred to as "dead tuples".

Let's say we're about to insert another 10000 rows of data (of similar size). But, depending on whether you run VACUUM before that, one of two things can happen:

Situation 1: No VACUUM runs, and we insert another 10000 rows of data. Because the space taken up by the dead tuples isn't marked as being truly empty yet (another session in Postgres may technically still have visibility of the data), the inserts cannot reuse the space. Instead, Postgres has to extend the total table to 20 MB in size.

Situation 2: VACUUM runs, and after that we insert the additional 10000 rows of data. Because VACUUM was able to confirm that the dead tuples are truly not visible to any other session in Postgres

(and no other reason to retain the data existed), the space is marked as empty. This allows half of the new data to reuse this empty space, causing Postgres to only extend the table to 15 MB in total size.

These same situations can happen on much larger tables, with a similar pattern. Ultimately it comes down to this: If you don't vacuum in time, a new insert (or updates, in many cases) will have to grow the size of the table, instead of being able to reuse the space taken up by "unvacuumed bloat". Later when a VACUUM occurs, it will be able to mark that space as reusable, but because Postgres doesn't rewrite the full table on vacuum (only parts of it), the table will not shrink in size. This causes bloat that persists with the physical table structure, until you perform a full table rewrite (e.g. as done through a VACUUM FULL, or with online approaches like pg_repack, etc).

We'll get back to some of the tunable settings that affect the frequency of vacuum, but generally speaking, we want vacuum to run more often, in order to reduce bloat. There are a few other causes of bloat, such as blocked vacuum, which we will discuss separately.

Now talking about all of that, you may wonder - what about index bloat? Everything we said thus far mainly talked about the table itself. Similarly, vacuuming too late will have negative effects on indexes as well, but there are a few other aspects to consider (page splits, inefficient sorting due to random values, etc) that this is best left to explore in detail in a future eBook.

The other main reason for VACUUM: Transaction IDs and Freezing

With bloat out of the way, there is one more big reason we need to vacuum our tables. Before we dive into that, let's add some background on how transaction IDs work in Postgres.

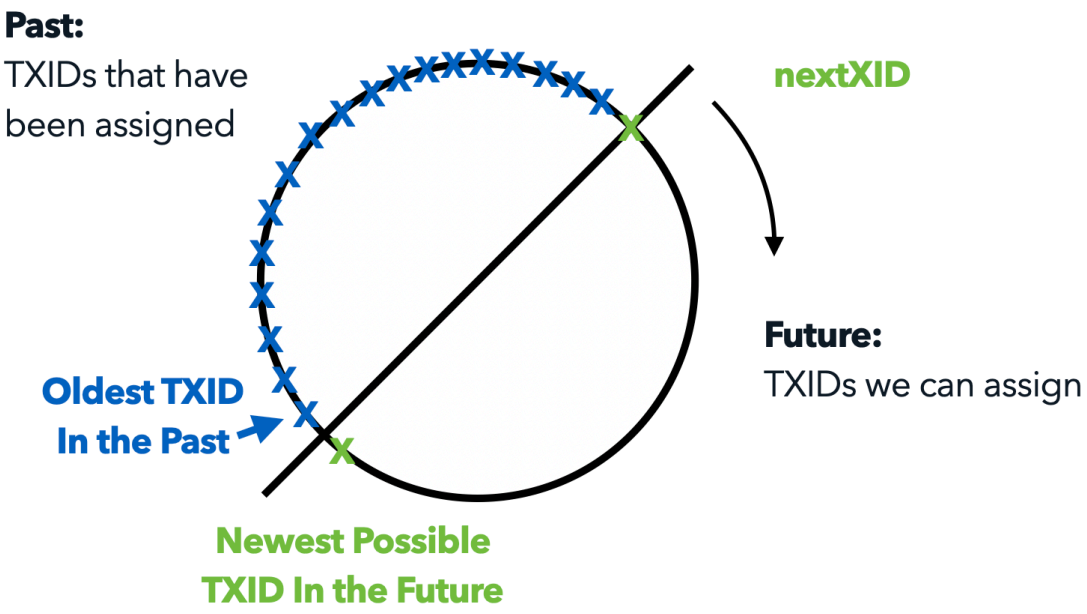
When we perform an insert, or another write operation (updates, etc) in Postgres, it will happen as part of a transaction. This transaction might be explicit, because we use a "BEGIN" and a "COMMIT", or Postgres might create one for us, because we're issuing a write statement. Each transaction gets a unique ID assigned. When the row that's being inserted or updated gets added to a table, this transaction ID is stored as well, in the internal "xmin" field, representing the first transaction where this data became visible.

Similarly, when a row gets deleted, the transaction ID is stored in the internal "xmax" field, representing the last transaction where this data was visible.

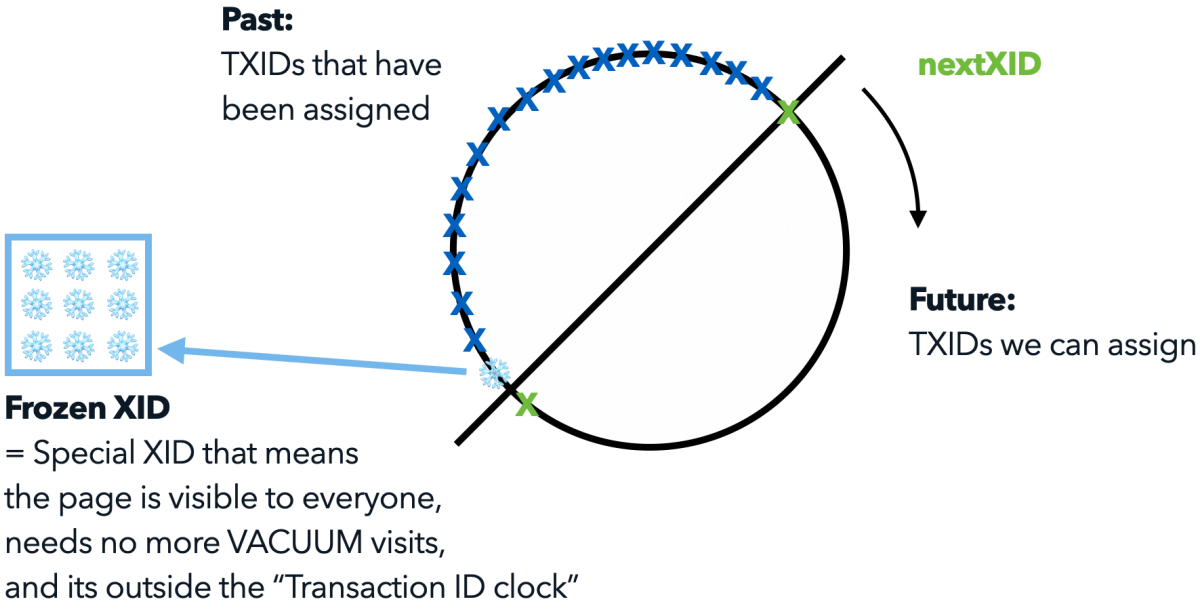
Concurrent sessions in a Postgres database have visibility of the rows based on what's called a "snapshot" (and the xmin/xmax fields of any rows being accessed), which we won't have time to dig into here - but suffice to say that these internal fields must be correct - or we have data corruption, since we may see data that was deleted, or data that should be visible is not.

Now, comes the problem with all of this: Transaction IDs in these internal fields are stored as 32-bit integers. That means there is a maximum of about 4 billion unique IDs. And because today's Postgres servers have a lot more activity in their lifetime than 4 billion transactions, the IDs are re-used. This re-use can take years, months, weeks, or on very very busy systems, even days. But it means that the same transaction ID can have two different meanings, depending on whether we have wrapped around the 32-bit integer number.

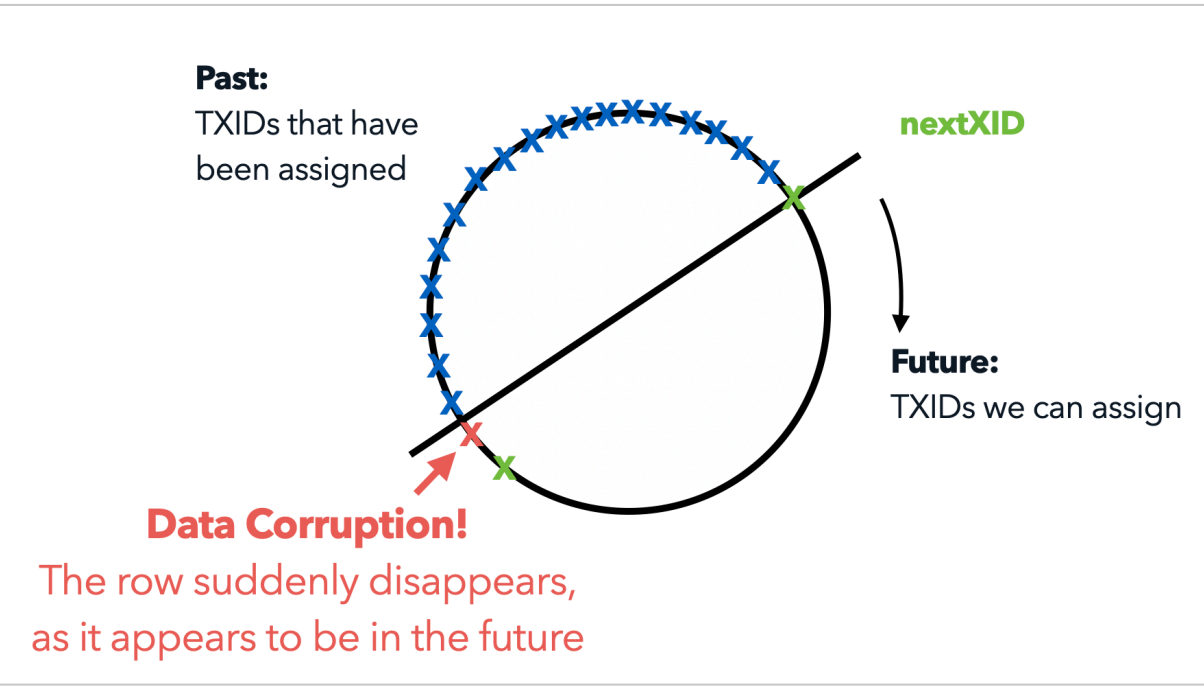
To deal with this problem, and avoid data corruption, Postgres has a concept called **"freezing".**



Freezing means instead of retaining the actual value of a transaction ID, Postgres marks it as "frozen", or put differently, to be infinitely in the past. This can only occur once no other session cares about the visibility being one way or the other - all sessions consider the actual ID value to be irrelevant, since it's clearly old enough. When we freeze values, we effectively take them out of the 32-bit range altogether:



If we didn't, we'd have data corruption after sufficient new transaction IDs have been consumed:



How do we freeze? **With VACUUM of course.** When VACUUM works a table, and when it looks at a given page, it first verifies that all row versions (tuples) on a table's page are old enough, i.e. their transaction IDs are visible to all sessions, and older than vacuum_freeze_min_age, typically 50 million IDs. If that is the case, the complete page will be recorded as frozen (only containing rows that have frozen transaction IDs), in the table's freeze map.

Doing this too often will cause unnecessary table writes and an increase in WAL activity by Postgres, which is why very recent row versions are not frozen right away.

Because freezing is such a critical operation, Postgres will automatically trigger "anti-wraparound" vacuums as needed, if a regular VACUUM based on tuple thresholds was not triggered in time.

In newer Postgres versions (Postgres 14 and newer), there is an additional concept of "failsafe" VACUUMs, whose sole purpose is to perform the freezing operation on a table, skipping any other activity that vacuum may need to do, such as dead tuple cleanup.

In the rare chance that vacuuming does not occur in time, **Postgres will shut down in order**

to prevent data corruption. This means the database will be completely unavailable whilst the necessary vacuuming can complete. Historically there have been multiple horror stories of production outages caused by insufficient vacuuming, but these days, thanks to improved defaults and failsafe vacuums, this is mostly a thing of the past.

Instead, we care about freezing because anti-wraparound vacuums typically have higher overhead, and thus we want to avoid having this high impact too often, or at the wrong time.

Performance benefits of frequent vacuuming: All-visible pages

There is one more motivation for vacuuming: The benefit of having a page in a table be marked as "all-visible". This is a special flag that gets set by the Postgres vacuum process, when all rows on the page are visible to all other sessions in Postgres.

Until the page gets modified again, it will retain this "all-visible" flag. A page that is all-visible, has two main benefits:

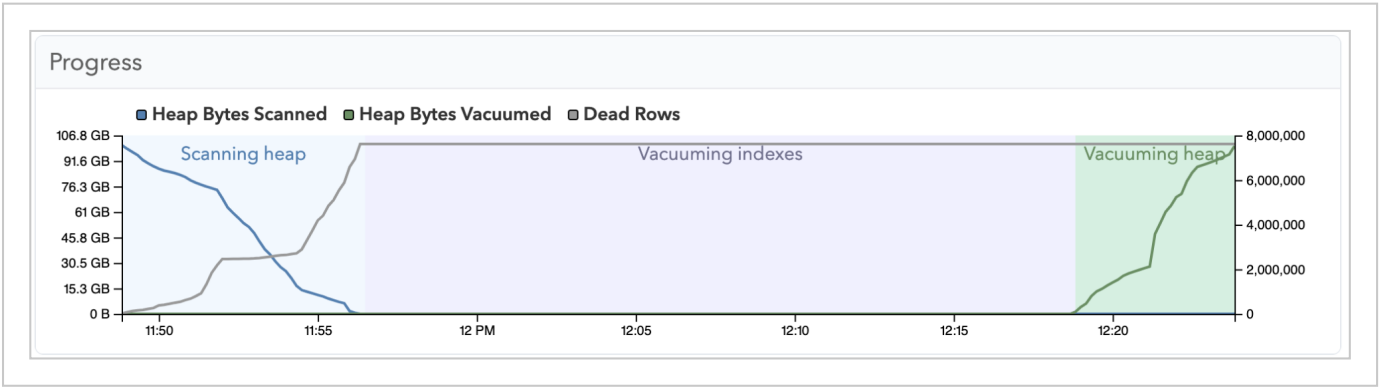
1. Certain visibility checks can be skipped, that otherwise require looking at the actual table page. The clearest example of this are Index Only Scans. When querying for data through an index, an Index Only Scan can return the full result without ever going to the table. But that is only true to the extent that the pages that the data is on are marked all-visible. If they are not, despite it being an Index Only Scan, Postgres will have to perform a "heap fetch" to access the table to confirm whether the data in the result is actually visible to the session. If Postgres estimates a lot of such heap fetches, it may opt not to do an Index Only Scan at all, since alternative scan methods might be more performant. If you're not getting Index Only Scans when you expect them, its often worth running an explicit VACUUM on the table. And if that fixes it, you may need more frequent autovacuums to increase the overall % of all-visible pages, to cause the planner to choose your preferred plan.
2. Subsequent vacuums may be able to skip a page that is marked as all-visible. If the page is also marked all-frozen (i.e. freezing already occurred), all vacuums will be able to skip the page, until it gets modified. If the page is not frozen yet, Postgres will freeze it depending on the age of the oldest transaction IDs on the table, specifically, once vacuum_freeze_table_age is reached (by default 150 million transaction IDs).

The biggest enemy of all-visible pages are frequent modifications. Even a single update on a row contained within a page, is enough for that page to lose its all-visible (and all-frozen, if applicable) attribute. Having a table with lots of updates is often times an anti-pattern in Postgres, but one of the reasons it causes issues with vacuum is that it effectively causes Postgres to have to do a lot of extra work all the time, because it cannot be sure that a page only contains visible rows.

The anatomy of a vacuum

Before we look at specific tuning settings, let's review what VACUUM actually does when it runs to achieve the goals discussed.

You can observe the phases of a running vacuum through the pg_stat_progress_vacuum view, and inside pganalyze we visualize these phases as well on each vacuum's detail page:



VACUUM starts by scanning the table, also referred to as the "heap". As it encounters row versions it checks whether they either are (1) dead to all sessions, based on an xmax that has passed the xmin horizon (more on that in a dedicated section later), (2) or visible to all sessions, which allows it to potentially mark the page as all-visible.

For each dead row version that was found, vacuum remembers its location (page number and tuple number on the page), but does not yet clean it up.

After either all pages that need to be visited have been visited (which may not be all pages, depending on the kind of vacuum running, and the state of the all-visible/all-frozen flags), or if autovacuum_work_mem has been reached, VACUUM switches to an index vacuuming phase.

That last part is important - because vacuum remembers each dead row location in memory, it can only keep a limited number of such dead rows. Once the limit is reached, it has to perform an index vacuuming phase, and resume scanning the remaining pages later.

In an index vacuuming phase, Postgres will visit each index, and all of the pages in the index, in order to remove index entries that point to dead rows. This is an important operation that prevents index bloat,

but it's very expensive. Postgres has no method to skip over parts of the index, which means that all of each table's index has to be read, and potentially large portions of it written as well.

What's worse: If the initial heap scanning phase did not have enough memory to complete its work, Postgres will resume after the index vacuuming, only to perform a second (or third, etc) index vacuum phase with the newly discovered dead row versions.

This leads us to our first tuning recommendation:

If you find your vacuums have multiple index phases, and your autovacuum_work_mem is not yet at the maximum (currently 1GB), it very likely makes sense to increase it. Unless you are very memory constrained! The additional overhead of the extra index phases almost always makes using more memory the better choice.

TUNING TIP

But, how do we find out whether we have multi-index phase vacuums?

If you're not using pganalyze, the easiest way to get this information is by parsing Postgres log events, and looking for the relevant information:

LOG

```
LOG: automatic vacuum of table "mydb.public.vac_test": index scans: 1
 pages: 0 removed, 1 remain, 0 skipped due to pins, 0 skipped frozen
 tuples: 3 removed, 6 remain, 0 are dead but not yet removable
 buffer usage: 70 hits, 4 misses, 4 dirtied
 avg read rate: 62.877 MB/s, avg write rate: 62.877 MB/s
 system usage: CPU 0.00s/0.00u sec elapsed 0.00 sec
```

What you'd expect to see is that this either says "index scans: 1", or notes that index scans were skipped (which can happen when there are very few dead rows). If it says a number higher than 1, that means that multiple index-phases had to be executed, which likely almost doubled (or worse) the effort of the vacuum.
If you're using pganalyze, there is a built-in check for this that runs in the background

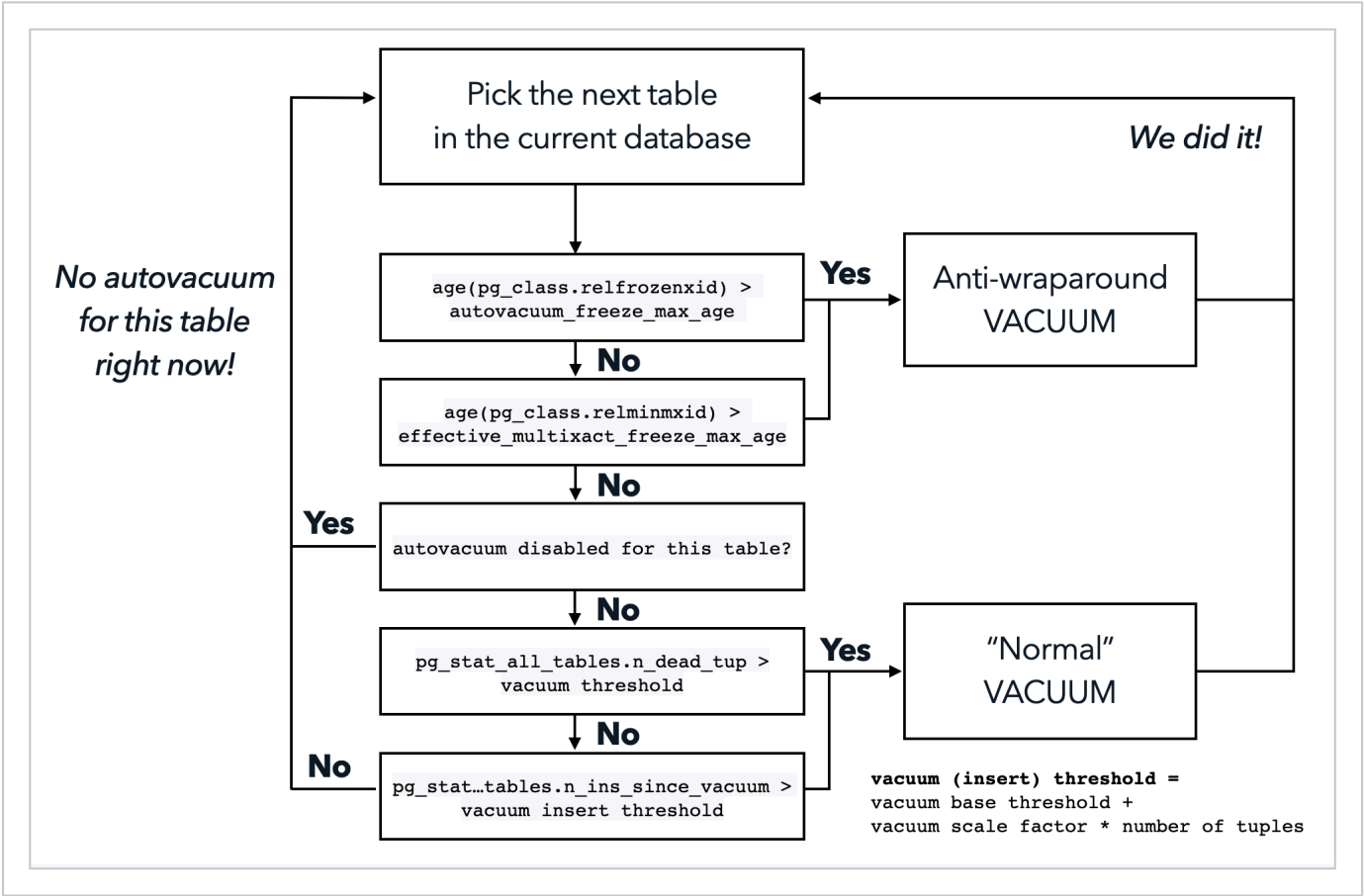
Let's talk more about what else we can tune!

Tuning when autovacuum is scheduled

Autovacuum is generally scheduled for one of two reasons:

- A row threshold is reached, based on either dead rows or inserted rows
- The oldest transaction ID on the table is considered too old

Or, more comprehensively represented as a diagram:



Scheduling matters because if vacuum runs too infrequently, bloat can accumulate, freezing may not happen early (making later anti-wraparound vacuums more expensive), and performance is worse because of pages not being all-visible.

In the most common case worth tuning, we can control the row thresholds for dead rows based on these two parameters, the **autovacuum_vacuum_threshold** and **autovacuum_vacuum_scale_factor**.

These are used in combination, in the following way, to decide when to vacuum a table based on the number of dead rows (pg_stat_user_tables.n_dead_tup):

*dead row threshold = number of rows (tuples) * scale factor + threshold*

On busy tables, it often makes sense to lower the scale factor, to increase the frequency of vacuuming, by having the dead tuple threshold be reached more often.

If you have a large table, but only a small portion of it is being updated (e.g. recent data is modified, but combined with a lot of older inactive data), its often a good idea to set autovacuum_vacuum_scale_factor to 0, and autovacuum_vacuum_threshold to a fixed number of dead rows (e.g. 100k, 1 million, etc). This ensures that despite the total number of rows being quite high, a comparatively smaller number of dead rows will trigger autovacuum.

TUNING TIP

A similar logic exists to vacuum a table based on the number of inserted rows. This is mainly aimed at append-only tables, and is controlled through autovacuum_vacuum_insert_threshold and autovacuum_vacuum_insert_scale_factor.

The formula for the threshold will vacuum based on the number of inserted rows since the last vacuum (pg_stat_user_tables.n_ins_since_vacuum):

*insert row threshold = number of rows (tuples) * scale factor + threshold*

The benefit of vacuuming based on inserts is that it enables subsequent vacuums to do less work (thanks to pages being all-visible, and some of them all-frozen), and it enables Index Only Scans on such append-only tables.

If a table is truly append-only, as well as large, it may also make sense to lower the vacuum_freeze_min_age used by autovacuum, on a per-table basis.

On an append-only table that never receives updates or deletes, its worth considering to set vacuum_freeze_min_age to 0, by utilizing the relevant table storage option:

ALTER TABLE large_append_only SET autovacuum_freeze_min_age = 0;

This ensures that if a vacuum that's triggered based on the insert threshold visits a page, it not only marks the page as all-visible, but also marks it as all-frozen, allowing all subsequent vacuums to skip over the page. In other workload patterns this is not advisable due to the extra overhead this causes when a page does get modified afterwards (only to be frozen again shortly after).

TUNING TIP

Speaking of freezing, we come to the second reason that autovacuum gets scheduled: For anti-wraparound vacuums.

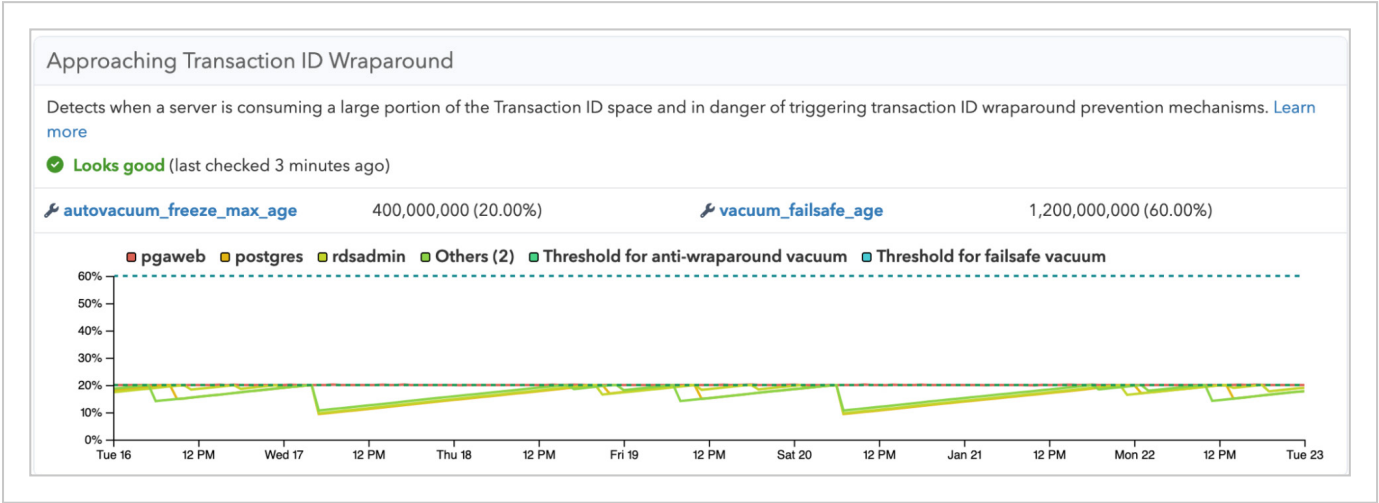
This is controlled through the autovacuum_freeze_max_age parameter, which sets a threshold that defines how many transaction IDs can be consumed, before a table needs to be revisited to perform freezing (if necessary). The default of this parameter is a quite conservative 200 million, or 10% of the maximum of visible transaction IDs (2 billion).

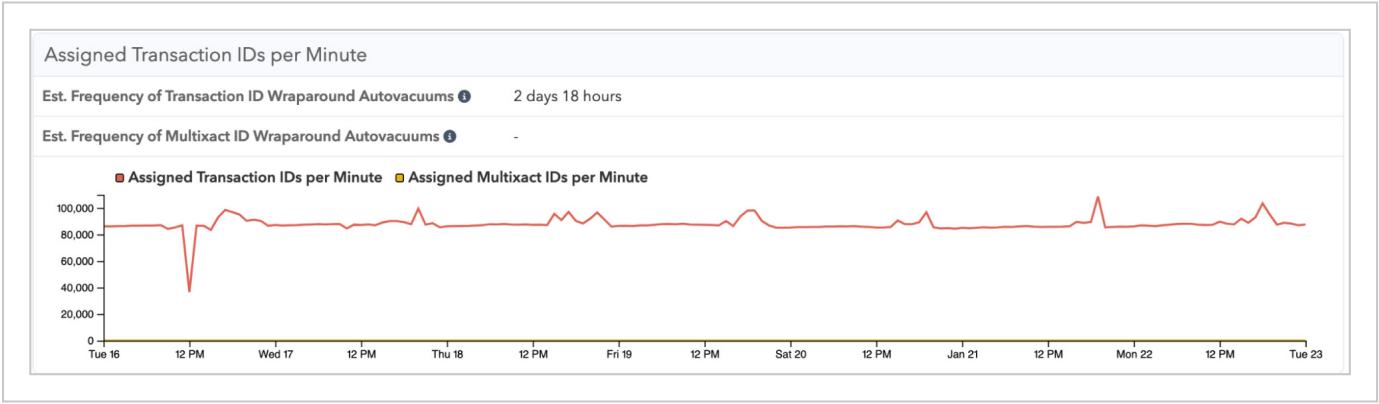
For most workloads, it makes sense to raise the default **autovacuum_freeze_max_age** from 200 million to 400 million (or possibly a bit higher), to decrease the frequency of anti-wraparound vacuums. This assumes an otherwise functioning vacuum cycle triggered based on row thresholds, which takes care of the bulk of the vacuuming workload. Raising this parameter reduces the occurrence of unnecessary anti-wraparound vacuums.

Note that depending on a table's workload, it may also make sense to lower **vacuum_freeze_min_age** (which controls when to freeze any pages that were visited), and increase **vacuum_freeze_table_age** (which controls when a "normal" vacuum visited pages marked all-visible but not yet frozen).

TUNING TIP

Especially when tuning the **autovacuum_freeze_max_age** threshold, it makes sense to keep a close eye on both the rate of transaction ID consumption, as well as the progress in advancing the transaction IDs on all tables and databases. This can be graphed and tracked through monitoring tools such as pganalyze:



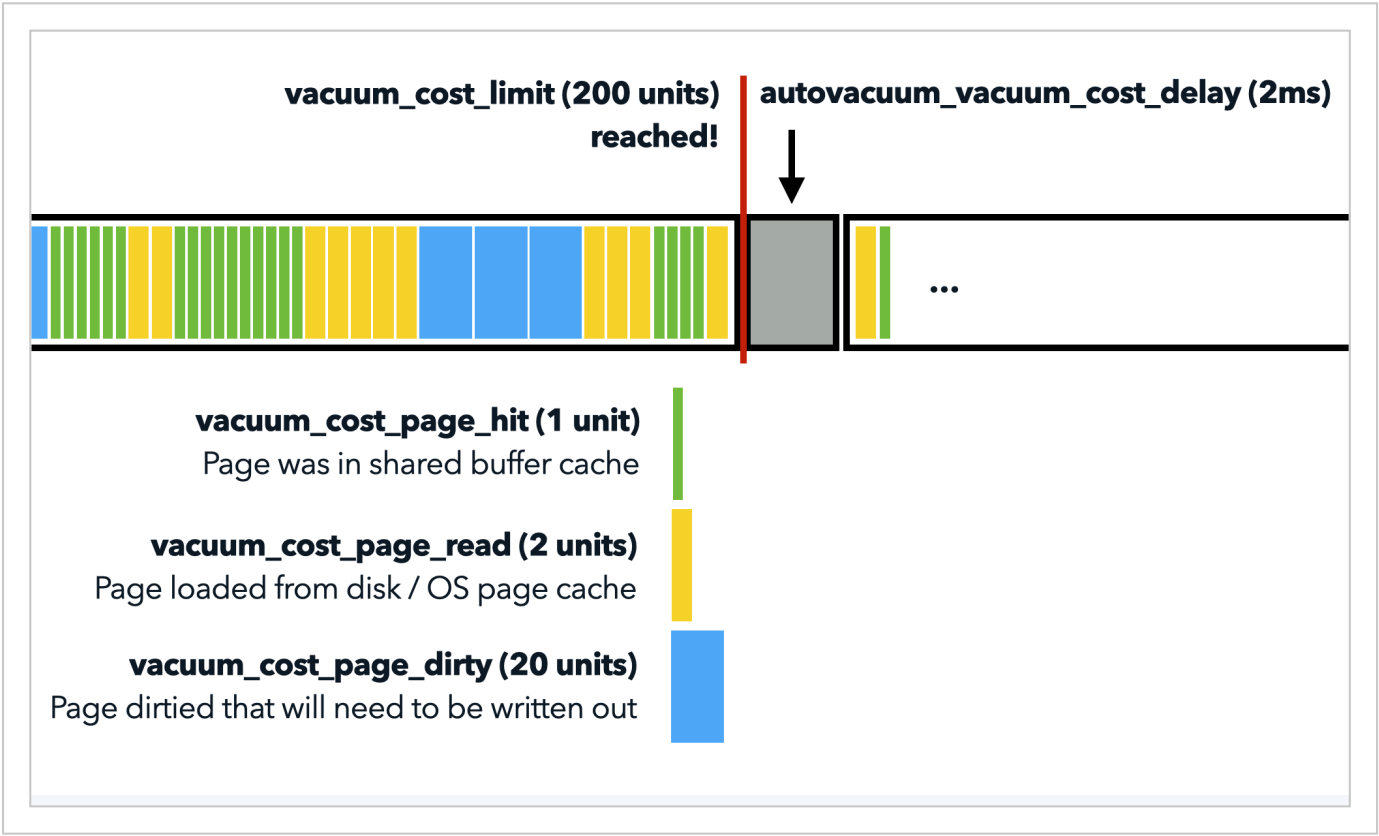


Tuning how fast autovacuum is running

Assuming that autovacuum has started, there are a few additional controls on how fast its running. If you were to run a vacuum directly on your Postgres database (e.g. “VACUUM mytable”), that would typically run at full speed.

But we don't necessarily want autovacuums that run in the background to run at full speed. By default Postgres will reduce impact on the rest of the system, by estimating the I/O activity that a vacuum has produced, and then sleeping after a certain number of operations.

Visually this can be represented as follows:



Different activities are costed differently. A page that is already in the Postgres shared buffer cache is estimated to be cheaper (vacuum_cost_page_hit, by default 1) vs a page that needs to be retrieved

from disk (vacuum_cost_page_miss, by default 2). Additionally, writes have an associated cost as well (vacuum_cost_page_dirty, by default 20). The activities get added up together into a total, and once that exceeds a threshold (vacuum_cost_limit, by default 200), the autovacuum process will pause for a moment (autovacuum_vacuum_cost_delay, by default 2ms).

Many years ago, common wisdom was to increase the speed of autovacuum, because it was just too slow. Luckily, Postgres 12 changed the defaults (changing autovacuum_vacuum_cost_delay from 20ms to 2ms), and Postgres 14 did it again (changing vacuum_cost_page_miss from 10 to 2) which made autovacuum run a lot faster for more workloads.

However, one common misconfiguration remains. And this relates to a different configuration setting, autovacuum_max_workers. Generally, if you have a large and busy database server, with either many databases, and/or many tables per database, it makes sense to increase how many autovacuum processes are running, by raising autovacuum_max_workers.

This seems pretty straightforward, but what's commonly missed is that the cost limit is not assigned per worker, but is rather considered a total. If

you add more workers, that effectively gives each worker a smaller portion of the pie, which could cause individual vacuums to take a lot longer.

When raising **autovacuum_max_workers**, consider raising **autovacuum_vacuum_cost_limit** as well, as the total cost limit is divided by the number of workers. This will avoid having individual workers take too long for a given vacuum, which may have unintended consequences.

TUNING TIP

Now that we've talked about the most commonly tuned settings, lets review one other frequent consideration:

Reducing the impact of autovacuum during business hours

Generally speaking, autovacuum has no concept of time. All it cares about are thresholds, be they related to rows or to transaction IDs. But in practice, when we have a product, we often have times during the day when additional I/O can be problematic (e.g. during the business day when our users most actively use the product), and other times when there is a lot of free capacity (e.g. on weekends).

There is no direct way of teaching Postgres to vacuum differently during the week vs on weekends, but broadly there are two strategies that can be employed to reduce business impact:

- Run autovacuum more often. This may be counterintuitive, but the more impactful vacuums are usually those that have to visit the full table (since very little pages are marked all-visible/all-frozen), potentially have to do multiple index phases because of the amount of dead tuples, and so forth. Lowering thresholds, so that busy tables get vacuumed more frequently, will lead to individual vacuums being faster, reducing the impact that is noticed.
- Run explicit VACUUM during off hours. Since the thresholds that autovacuum is triggered on are reset when it runs successfully, we can simply do a manual VACUUM outside business hours, and that will automatically reduce the chance that it runs when it's inconvenient. Details of course depend on the workload, but the simplest case, this can be helpful if an impactful vacuum is already the preferred method, and instead of having that occur during business hours every 2-3 weeks, it's intentionally run on the weekend, through a manual VACUUM command.

That's it with the tuning tips!

But there is one more thing that I've often seen cause production issues, that we shouldn't forget about.

When VACUUM runs, yet it doesn't do any work

If we think back to vacuums that are triggered to clean up dead rows, it's important to mention something: A VACUUM may see a dead row, yet not be able to clean it up.

This can occur for a number of reasons, the most important ones being:

- There is a currently running transaction that was started before the row was deleted (the transaction may still be able to see the deleted row version)
- There is a replica that's executing a query that was started before the row was deleted (if the row was removed by vacuum, that would cause the standby to replicate that removal, and then no longer be able to answer its query correctly)
- There is an open replication slot that's not progressing (Postgres can't know whether the slot needs a certain row to still be visible, and thus will not clean up rows that were deleted after the relevant transaction ID)
- There is a prepared two-phase commit transaction

that was created before the row was deleted (the transaction may still be able to see the deleted row version)

The relevant limit of maximum transaction ID that can be cleaned up is also referred to as the "xmin horizon".

If this applies to your table, you will see the seemingly innocent log message indicating that vacuum found "dead tuples not yet removable":

LOG

```
1   LOG: automatic vacuum of table "mydb.public.mytable":
2    pages: 0 removed, 21146 remain, 0 skipped due to pins
3    tuples: 0 removed, 152873 remain, 26585 are dead but not yet removable
4    buffer usage: 104893 hits, 54167 misses, 2165 dirtied
5    avg read rate: 3.548 MB/s, avg write rate: 0.142 MB/s
6    system usage: CPU 0.28s/0.97u sec elapsed 119.26 sec
```

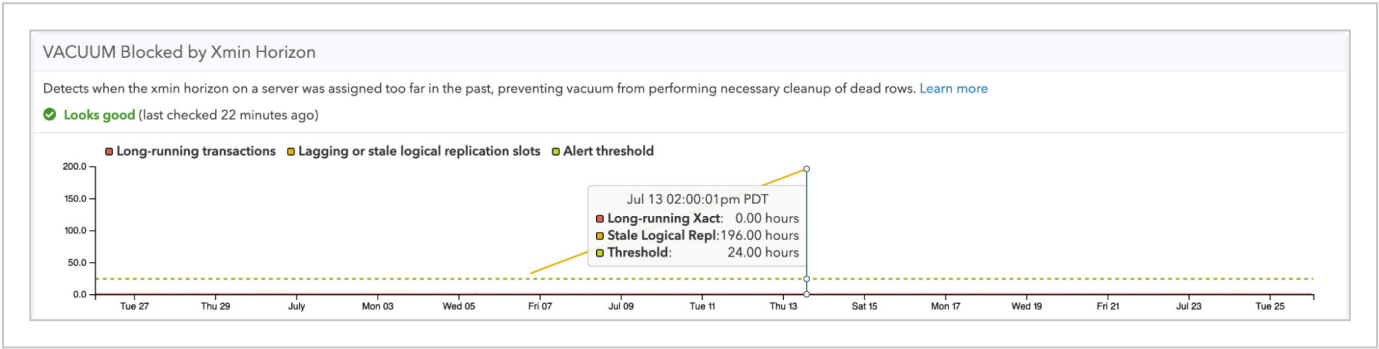
But this small detail is a much bigger problem than it seems at first. Imagine a long-running transaction that's kept open for 24 hours. During those 24 hours, none of the deleted data can be reclaimed by vacuum, and thus tables will increase in size, ultimately causing bloat. Further, freezing cannot succeed either, and so more work needs to be done later, once the transaction is finished.

Now imagine that the transaction runs for 1 week.

Or you have a replication slot that was forgotten, and it blocks cleanup the exact same way. This can quickly spiral into a problem that could ultimately take your database down.

And until you cancel the problematic transaction, cancel the standby query, or drop the inactive replication slot, the main symptom you will see is that (1) the autovacuum log output will warn you about this, (2) autovacuum will keep vacuuming the same table, over and over again, right after it finished vacuuming it.

Monitoring for this is critical. You can write a lengthy SQL query for this, or have pganalyze take care of this for you, and alert you when this becomes a problem:

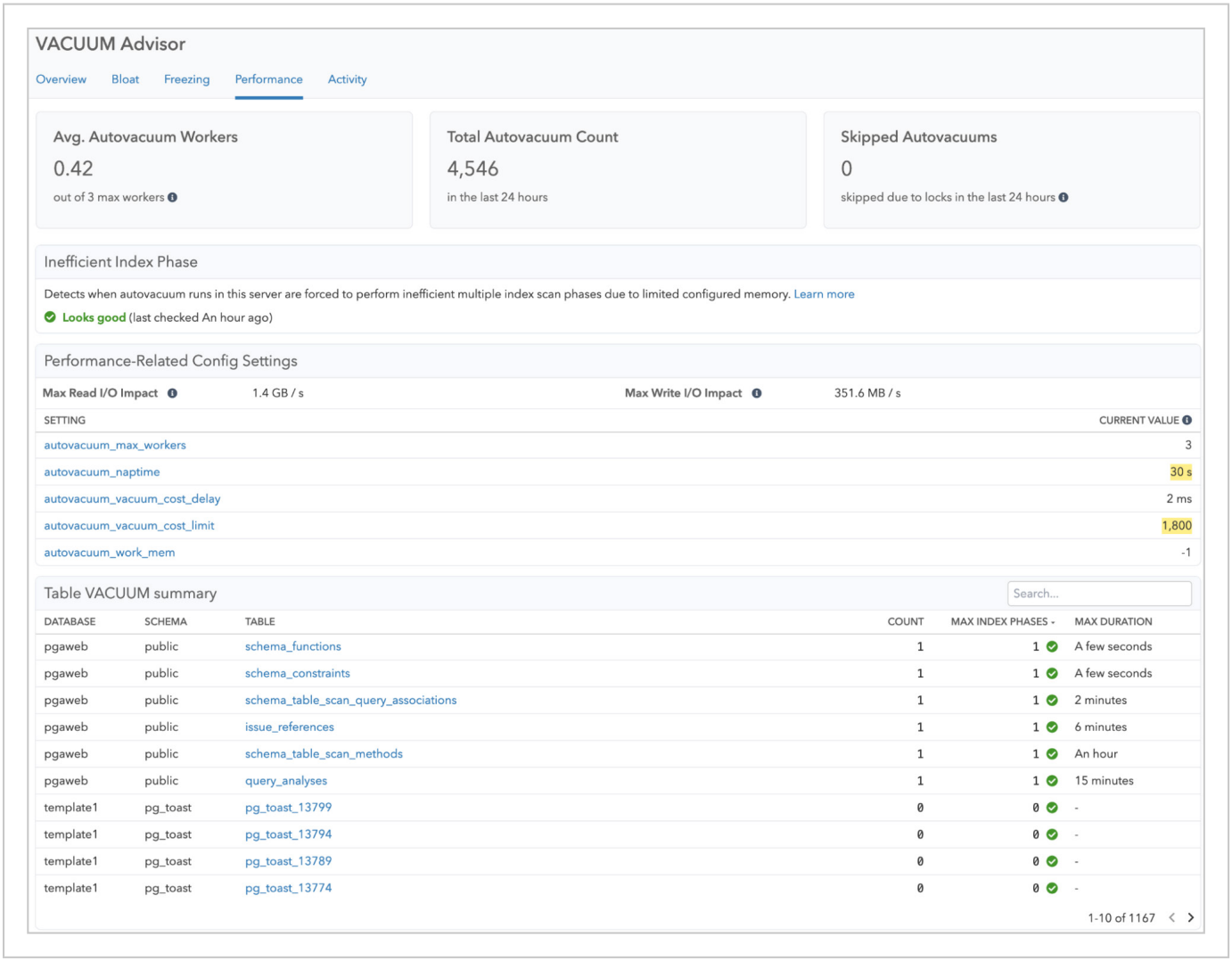


The pganalyze VACUUM Advisor

All of what we have discussed in this eBook led us to design a better way to tune autovacuum for Postgres. The pganalyze VACUUM Advisor combines the best and most in-depth analysis

of Postgres autovacuum metrics, with tuning recommendations for adjusting autovacuum settings.

With pganalyze VACUUM Advisor you will get proactive notifications when VACUUM is blocked due to a long-running transaction, get workload-specific tuning insights, for example to tune the dead tuple threshold and scale factor, and be able to collaborate with your team in understanding Postgres autovacuum behavior.



As part of pganalyze VACUUM Advisor, you can also use the VACUUM Simulator, which lets you simulate different autovacuum schedules, based on different settings that you choose, for your whole server, or for a particular table.

Try out pganalyze VACUUM Advisor today, to make sure your autovacuum is tuned right.

Try pganalyze for free

Have performance issues with your database, and looking for a way to improve indexes, vacuum and query plans?

pganalyze helps deliver consistent database performance and availability through intelligent tuning advisors and continuous database profiling.

It provides deep insights for your database, with detailed performance analysis, and automated EXPLAIN plans, to quickly understand what is slow with your database. pganalyze integrates directly with major cloud providers, as well as self-managed Postgres installations.

Get started easily with a free 14-day trial, or learn more about our Enterprise product.

If you want, you can also request a personal demo.

pganalyze

Share this with your peers:

Postgres performance at any scale.

Take control of your Postgres workload and query plans, discover tuning opportunities with automatic advisors, and empower all development teams to fix slow queries.

Specializing in PostgreSQL database monitoring and optimization, pganalyze gives automatic insights into Postgres query plans, helps improve Postgres query performance with its Index Advisor and VACUUM Advisor, and lets you perform query drill-down analysis, observe per-query statistics and conduct trend analysis - all in one platform that integrates with both self-managed Postgres servers as well as Database-as-a-service providers like Amazon RDS.

Hundreds of companies monitor their production PostgreSQL databases with pganalyze.

Be one of them.
Sign up for a free trial today!

"Our overall usage of Postgres is growing, as is the amount of data we're storing and the number of users that interact with our products. pganalyze is essential to making our Postgres databases run faster, and makes sure end-users have the best experience possible."

Robin Fernandes, Software Development Manager
Atlassian