PostgreSQL's `VACUUM` command has to process each table on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To update the visibility map, which speeds up index-only scans.
4. To protect against loss of very old data due to *transaction ID wraparound* or *multixact ID wraparound*.

- There are two variants of `VACUUM`: standard `VACUUM` and `VACUUM FULL`. `VACUUM FULL` can reclaim more disk space but runs much more slowly.
- Also, the standard form of `VACUUM` can run in parallel with production database operations. (Commands such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` will continue to function normally, though you will not be able to modify the definition of a table with commands such as `ALTER TABLE` while it is being vacuumed.)
- `VACUUM FULL` requires an `ACCESS EXCLUSIVE` lock on the table it is working on, and therefore cannot be done in parallel with other use of the table. Generally, therefore, administrators should strive to use standard `VACUUM` and avoid `VACUUM FULL`.
- `VACUUM` creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions.
- In PostgreSQL, an `UPDATE` or `DELETE` of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multi-version concurrency control, the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction.

As tables become bloated with dead tuples, PostgreSQL queries — especially **sequential scans** — become slower. Why?

- PostgreSQL reads data **block by block** (each block is usually 8 KB).
- Even if most blocks contain only dead tuples, they are still read during a scan.
- The **I/O cost increases**, as more blocks need to be read and evaluated.
- The **CPU must filter out dead tuples** during query execution.

Table bloat is often accompanied by **index bloat**. When a row is deleted or updated:

- The index entry pointing to that row isn't immediately removed.
- Index pages begin to include pointers to dead or outdated rows.
- Over time, the index grows in size but becomes **less efficient**.

This leads to:

- **Longer index scans**, as PostgreSQL has to check visibility for each pointer.
- **Increased CPU usage** during queries involving index lookups.
- **Replanning of queries**, as the query planner may choose sequential scans over bloated indexes.

- The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running `VACUUM`
- If you have a table whose entire contents are deleted on a periodic basis, consider doing it with `TRUNCATE` rather than using `DELETE` followed by `VACUUM`. `TRUNCATE` removes the entire content of the table immediately, without requiring a subsequent `VACUUM` or `VACUUM FULL` to reclaim the now-unused disk space. The disadvantage is that strict MVCC semantics are violated.
- The PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the `ANALYZE` command, which can be invoked by itself or as an optional step in `VACUUM`. It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.
- The autovacuum daemon, if enabled, will automatically issue `ANALYZE` commands whenever the content of a table has changed sufficiently. The daemon schedules `ANALYZE` strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes.
- It is possible to run `ANALYZE` on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database, because it is a fast operation. `ANALYZE` uses a statistically random sampling of the rows of a table rather than reading every single row.
- Columns that are heavily used in `WHERE` clauses and have highly irregular data distributions might require a finer-grain data histogram than other columns. See `ALTER TABLE SET STATISTICS`, or change the database-wide default using the default_statistics_target configuration parameter.

Tuning based on AUTOVACUUM

For cleanup, you need to be looking at least at these values:

- `pg_stat_all_tables.n_dead_tup` - number of dead rows in each table (both user tables and system catalogs)
- `(n_dead_tup / n_live_tup)` - ratio of dead/live rows in each table
- `(pg_class.relpages / pg_class.relrows)` - space "per row"

Thresholds and Scale Factors

Naturally, the first thing you may tweak is when the cleanup gets triggered, which is affected by two parameters (with these default values):

- `autovacuum_vacuum_threshold = 50`
- `autovacuum_vacuum_scale_factor = 0.2`

The cleanup is triggered whenever the number of dead rows for a table (which you can see as `pg_stat_all_tables.n_dead_tup`) exceeds

```
threshold + pg_class.relrows * scale_factor
```

This formula says that up to 20% of a table may be dead rows before it gets cleaned up (the threshold of 50 rows is there to prevent very frequent cleanups of tiny tables, but for large tables it's dwarfed by the scale factor).

So, what's wrong with these defaults, particularly with the scale factor? This value essentially determines what fraction of a table can be "wasted", and 20% works pretty well for small and medium sized tables - on a 10GB table this allows up to 2GB of dead rows. But for a 1TB table this means we can accumulate up to 200GB of dead rows, and then when the cleanup finally happens it will have to do a lot of work at once.

This is an example of accumulating a lot of dead rows, and having to clean up all of it at once, which is going to hurt - it's going to use a lot of I/O and CPU, generate WAL and so on. Which is exactly the disruption of other backends that we'd like to prevent.

This can be done by significantly decreasing the scale factor, perhaps like this:

```
autovacuum_vacuum_scale_factor = 0.01
```

This decreases the limit to only 1% of the table, to ~10GB of the 1TB table. Alternatively you could abandon the scale factor entirely, and rely solely on the threshold:

```
autovacuum_vacuum_scale_factor = 0
```

```
autovacuum_vacuum_threshold = 10000
```

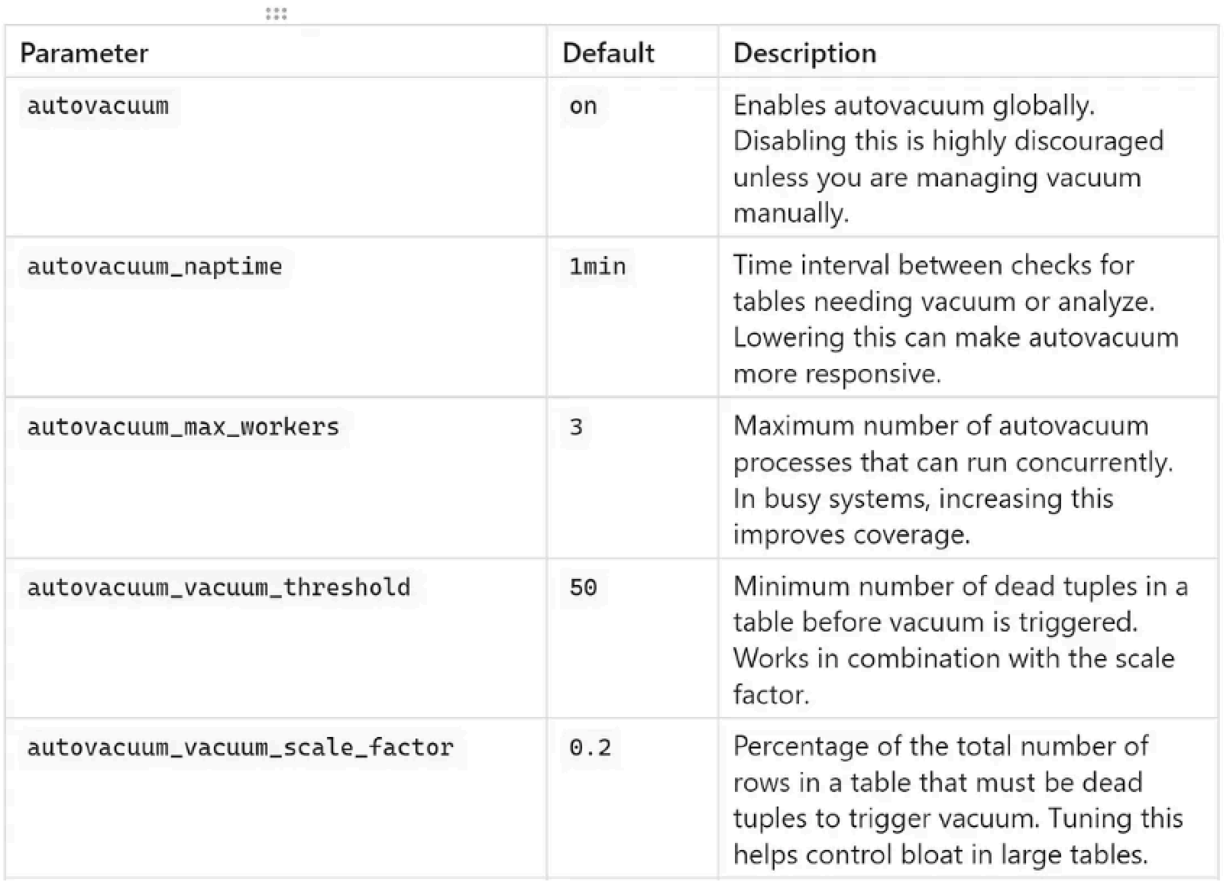
This would trigger the cleanup after generating 10000 dead rows.

Autovacuum Throttling

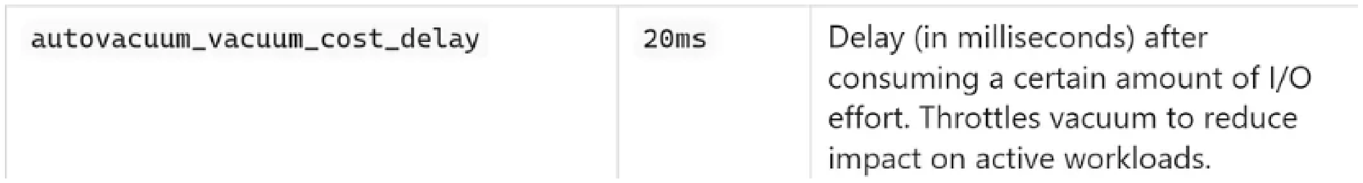
- The cleanup process is fairly simple - it reads pages (8kB chunks of data) from data files, and checks if the page needs cleaning up. If there are no dead rows, the page is simply thrown away without any changes. Otherwise it's cleaned up (dead rows are removed), marked as "dirty" and eventually written out (to cache and eventually to disk).

We certainly don't want the cleanup to consume so much resources (CPU and disk I/O) to affect regular user activity (e.g. by making queries much slower). That requires limiting the amount of resources the cleanup can utilize over time.
The throttling is then done by limiting the amount of work that can be done in one go, which is by default set to 200, and every time the cleanup does this much work it'll sleep for a little bit. The sleep used to be 20ms, but in PG 12 it was reduced to 2ms.
`autovacuum_vacuum_cost_delay = 2ms`
`autovacuum_vacuum_cost_limit = 200`

Let's compute how much work that actually allows. The delay changed in PG 12, which means we have three groups of releases, with different parameter parameter values.



Parameter	Default	Description
`autovacuum`	`on`	Enables autovacuum globally. Disabling this is highly discouraged unless you are managing vacuum manually.
`autovacuum_naptime`	`1min`	Time interval between checks for tables needing vacuum or analyze. Lowering this can make autovacuum more responsive.
`autovacuum_max_workers`	`3`	Maximum number of autovacuum processes that can run concurrently. In busy systems, increasing this improves coverage.
`autovacuum_vacuum_threshold`	`50`	Minimum number of dead tuples in a table before vacuum is triggered. Works in combination with the scale factor.
`autovacuum_vacuum_scale_factor`	`0.2`	Percentage of the total number of rows in a table that must be dead tuples to trigger vacuum. Tuning this helps control bloat in large tables.

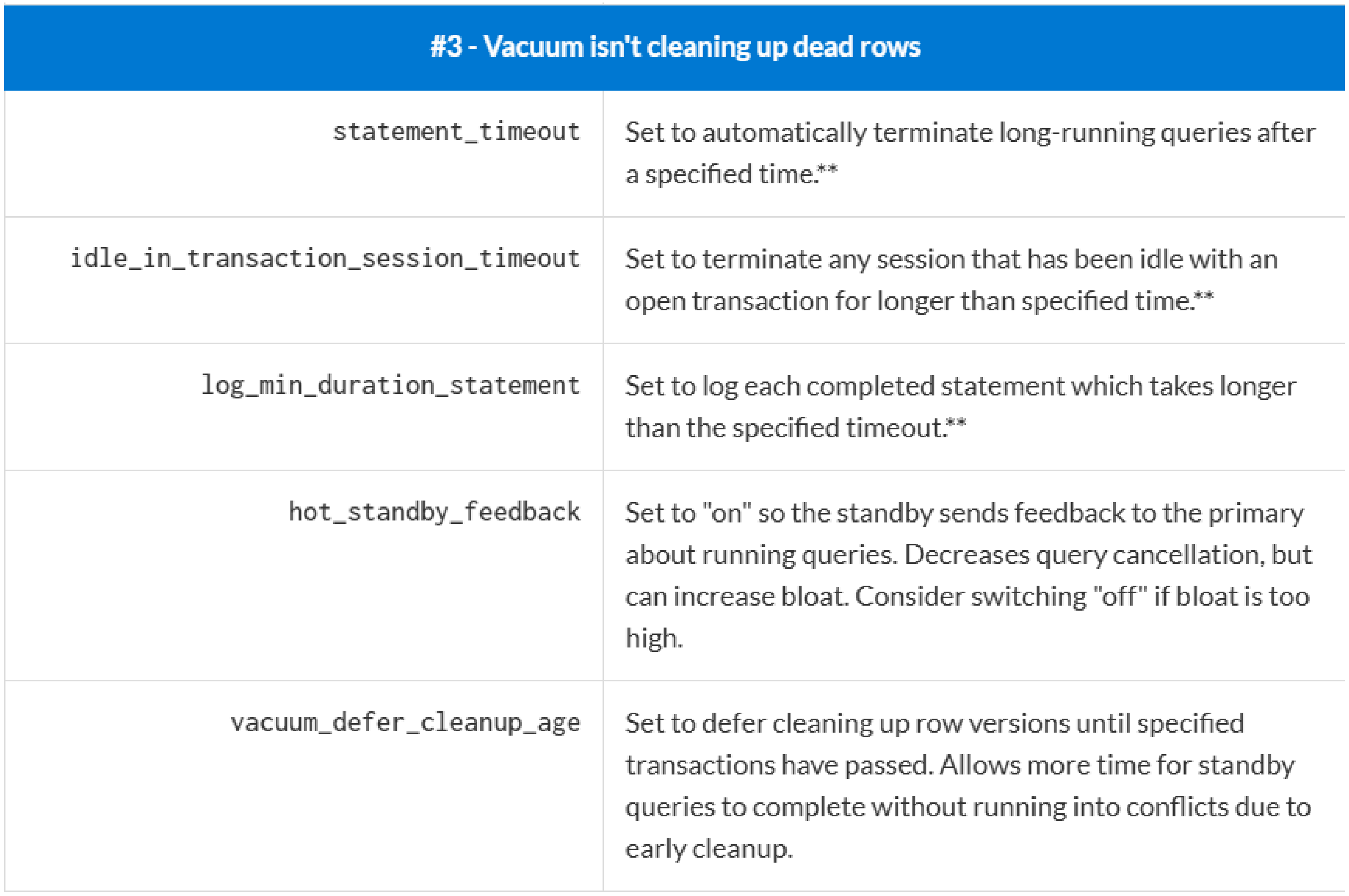


`autovacuum_vacuum_cost_delay`	`20ms`	Delay (in milliseconds) after consuming a certain amount of I/O effort. Throttles vacuum to reduce impact on active workloads.

Increasing the number of parallel workers

- Autovacuum can only vacuum `autovacuum_max_workers` tables in parallel. So, if you have hundreds of tables being actively written to (and needing to be vacuumed), doing them 3 at a time might take a while (3 is the default value for `autovacuum_max_workers`).

- Therefore, in scenarios with a large number of active tables, it might be worth increasing autovacuum_max_workers to a higher value—assuming you have enough compute to support running more autovacuum workers.
- Before increasing the number of autovacuum workers, make sure that you are not being limited by cost limiting. Cost limits are shared among all active autovacuum workers, so just increasing the number of parallel workers may not help, as each of them will then start doing lesser work.
- Each worker process only gets a fraction of the cost limit (roughly 1/ `autovacuum_max_workers`) of the global limit. So increasing the number of workers only makes them go slower, it doesn't increase the cleanup throughput.



#3 - Vacuum isn't cleaning up dead rows	
`statement_timeout`	Set to automatically terminate long-running queries after a specified time.**
`idle_in_transaction_session_timeout`	Set to terminate any session that has been idle with an open transaction for longer than specified time.**
`log_min_duration_statement`	Set to log each completed statement which takes longer than the specified timeout.**
`hot_standby_feedback`	Set to "on" so the standby sends feedback to the primary about running queries. Decreases query cancellation, but can increase bloat. Consider switching "off" if bloat is too high.
`vacuum_defer_cleanup_age`	Set to defer cleaning up row versions until specified transactions have passed. Allows more time for standby queries to complete without running into conflicts due to early cleanup.