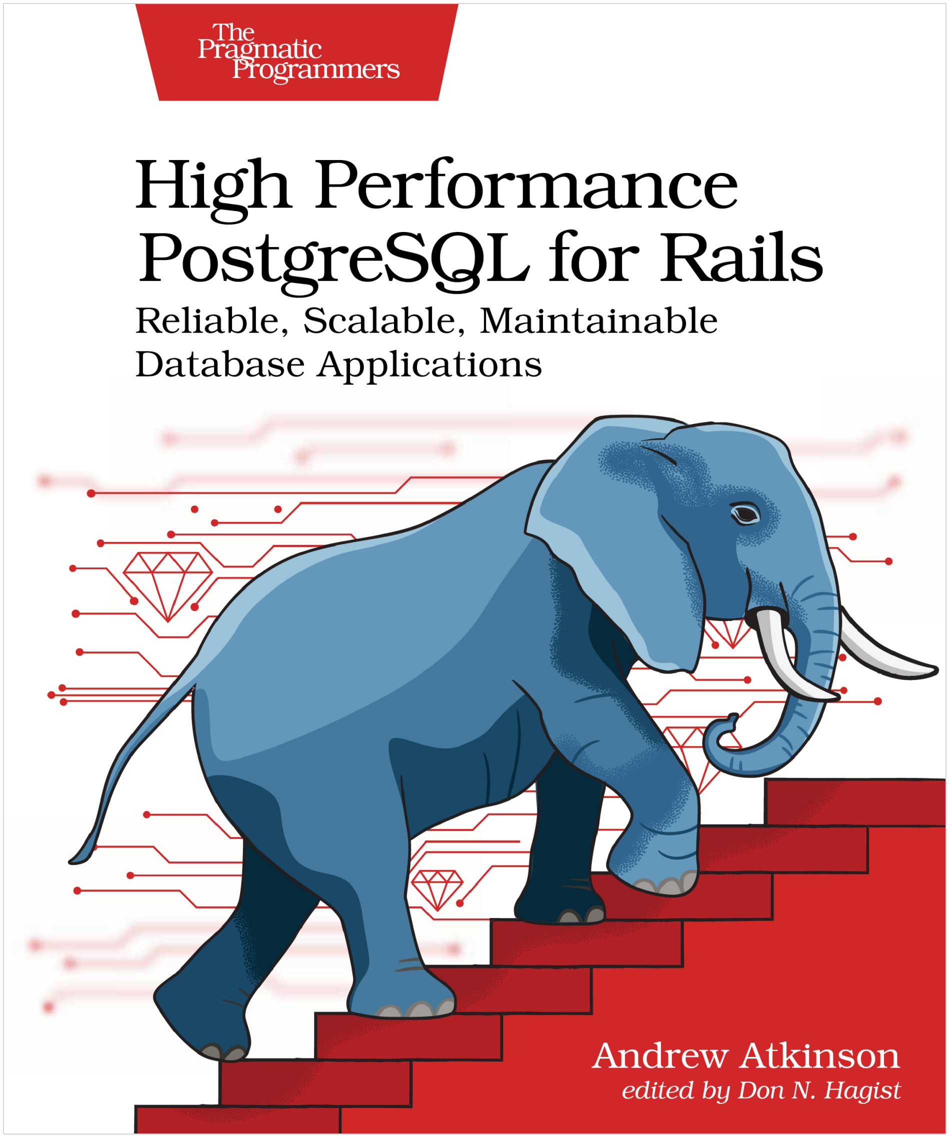

The Pragmatic Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

Andrew Atkinson

edited by Don N. Hagist

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

Rebuilding Indexes Without Downtime

Besides table data, indexes become bloated over time with entries to refer to dead tuples. To optimize your indexes, periodically rebuild the important ones.

Fortunately, PostgreSQL made it possible to rebuild indexes while they're being used concurrently, starting in version 12.

If you're on an earlier version of PostgreSQL, to get this functionality, the best bet is to upgrade to PostgreSQL 12 or newer. If upgrading is not an option, third-party tools like pg_repack[9] can be used to achieve similar results.

The post, "Using pg_repack to Rebuild Indexes,"[10] shows how to install and use it. This approach builds a replacement index in the background and then swaps the names between the original index and new index with a short exclusive lock.

In PostgreSQL 12, REINDEX gained support for the CONCURRENTLY option. This works similarly to how pg_repack does, but it's native to PostgreSQL. By being native, it continues to receive improvements in newer versions.

Let's try putting this into action. From psql, run the following command to reindex the index_trips_on_driver_id index. Use the VERBOSE option to show more information.

sql/reindex_concurrently_verbose.sql

```
REINDEX (VERBOSE) INDEX CONCURRENTLY index_trips_on_driver_id;
```

Providing a table name as an option to REINDEX performs a reindex for *all* indexes that are on the specified table. Try out the following example.

sql/reindex_table.sql

```
REINDEX (VERBOSE) TABLE CONCURRENTLY trips;
```

Great. You've now learned some basics with Autovacuum and how to tune it. You've seen how to manually rebuild indexes using native tools and third-party tools. Sometimes, you'll want to manually perform Vacuum operations. How do you do that?

Running Manual Vacuums

As you learned earlier, PostgreSQL stores data in pages that are 8 KB in size by default. The pages are not completely filled; space is left available to store new tuples from row modifications. Dead tuples take up space in those

9. https://reorg.github.io/pg_repack/
10. https://andyatkinson.com/blog/2021/09/28/pg-repack

pages. PostgreSQL also considers unused free space part of what makes up bloat.[11]

When you want to make sure row visibility is updated and space from dead tuples is reclaimed, you can perform a manual VACUUM operation. Optionally, include the ANALYZE keyword to update table statistics at the same time. Try that out by running the following from psql:

sql/vacuum_table.sql
```
VACUUM (ANALYZE, VERBOSE) users;
```

New versions of PostgreSQL continue to add capabilities to Vacuum and Analyze. Version 11 added support for Vacuuming multiple tables at once. Version 12 added SKIP_LOCKED support, which skips locked tables (see "Postgres 12 Highlight—SKIP_LOCKED for VACUUM and ANALYZE"[12]).

Try out multiple tables and SKIP_LOCKED by running the following statement:

sql/vacuum_multiple_skip_locked.sql
```
VACUUM (SKIP_LOCKED) trip_requests, trips;
```

PostgreSQL tracks when Vacuum and Analyze are run, both when run from the automatic process and when they've been run manually.

The last_analyzed timestamp for users shows the time of the last manual vacuum and the last_auto_analyzed_at timestamp shows the last "automatic" analyze that was run from Autovacuum.

sql/last_analyzed_pg_stat_all_tables.sql
```
SELECT
  schemaname,
  relname,
  last_autoanalyze,
  last_analyze
FROM pg_stat_all_tables
WHERE relname = 'vehicles';
```

Try running ANALYZE on the vehicles table listed in the previous query. Prior to running ANALYZE, the last_analyze timestamp may have been empty. After running ANALYZE vehicles; verify that last_analyze is now set.

What else is available related to Vacuum? Since PostgreSQL 13, parallel Vacuum workers can be configured (see "Parallel Vacuum in Upcoming PostgreSQL 13"[13]).

11. https://www.postgresql.org/docs/current/glossary.html
12. https://paquier.xyz/postgresql-2/postgres-12-vacuum-skip-locked/
13. https://www.highgo.ca/2020/02/28/parallel-vacuum-in-upcoming-postgresql-13/

The default value for MAX_PARALLEL_MAINTENANCE_WORKERS is 2. Try increasing it to 4. Specify the PARALLEL option with a value of 4, and set the VERBOSE flag for more information. Up to four workers will be started.

sql/set_max_parallel_maintenance_workers.sql

```
SET MAX_PARALLEL_MAINTENANCE_WORKERS=4;

VACUUM (PARALLEL 4, VERBOSE) users;
```

We're exploring some of the ways to add more resources to maintenance operations. To learn more about the effects of bloat, it can be helpful to simulate it locally so that you have a test environment to use for learning.

How do we do that? Let's configure that in the next section.

Simulating Bloat and Understanding Impact

Since high estimated bloat levels aren't normal in local development or for new databases, we'll simulate high bloat locally so we can see the effect.

High bloat occurs for tables with a high amount of updates and deletes. Autovacuum can fall behind in processing the bloat. Check out the page "Bloat and Vacuum"[14] for a nice overview, and explore which of your application tables get the most updates. Find those tables using the query top_updated_tables.sql.[15]

Run the code/sql/insert_users_generate_series.sql SQL statement that you saw earlier, which creates a temp.users table with ten million rows. The fragment WITH (autovacuum_enabled = FALSE) is used to disable Autovacuum for the table.

After it's created, let's collect an estimate of the bloat percentage.

Run the table_bloat.sql[16] script from psql, which can be found at https://github.com/ioguix/pgsql-bloat-estimation. Navigate to the SQL script, view the raw version, and copy and paste it into an editor. Add WHERE schemaname = 'temp' AND tblname = 'users' just before the ORDER BY to narrow down the results to the temp.users table you just made.

View the bloat_pct result row column, which shows an estimated bloat percentage. Since you haven't made many updates, this should start off as a low value, for example, less than 5 percent. This is mainly unused space left available in pages.

14. https://docs.crunchybridge.com/insights-metrics/bloat-and-vacuum#
15. https://github.com/andyatkinson/pg_scripts/blob/main/top_updated_tables.sql
16. https://github.com/ioguix/pgsql-bloat-estimation

Next, simulate bloat by creating a lot of updates for temp.users. Run the following statement a few times, which creates up to one million updates, to increase the amount of dead tuples:

sql/bloat_simulate_user_updates.sql

```
UPDATE temp.users
SET first_name =
  CASE (seq % 2)
    WHEN 0 THEN 'Bill' || FLOOR(RANDOM() * 10)
    ELSE 'Jane' || FLOOR(RANDOM() * 10)
  END
FROM GENERATE_SERIES(1, 1_000_000) seq
WHERE id = seq;

-- Vacuum and Analyze the table
VACUUM (ANALYZE, VERBOSE) temp.users;
```

The statement has one update per row, matching id values up to one million. Each row gets a unique string that's either "Bill" or "Jane", based on id being even or odd.

After the update completes, check the bloat_pct column value. The estimated bloat percentage will start increasing.

The space used by the dead tuples can be marked for re-use by running the VACUUM command for the table. Remember that running VACUUM only (without FULL, discussed next) only marks the space available for re-use, but does *not* fully reclaim or free up the space.

To fully reclaim or free it up, we'd need to "rewrite" the table content behind the scenes. PostgreSQL provides the VACUUM FULL command to do that. This reclaims more space. Unfortunately, VACUUM FULL requires an exclusive table lock. The lock length is generally too long for this operation to be "online" if there's concurrent activity for the table.

With that said, you may be able to schedule a VACUUM FULL during a low activity period, or better yet, during a downtime window. Test how long it takes to run on a separate instance, although depending on how you create separate instances, you may not be able to fully retain the real bloat levels to test with. Otherwise, you'll be limited to maintenance operations that can be performed safely while "online."

What other options are there? The pg_repack tool you saw used for indexes can also be used to rebuild tables online (which includes the indexes). The post, "Using pg_repack to Rebuild PostgreSQL Database Objects Online,"[17]

17. https://www.percona.com/blog/pg_repack-rebuild-postgresql-database-objects-online

has examples. The post, “pg_squeeze—Shrinks Tables Better Than Vacuum,”[18] describes another tool called pg_squeeze that’s an alternative way to rebuild (and shrink) tables.

You’ve now seen how to use the Vacuum, Analyze, and Reindex (VAR) commands to perform manual and automatic maintenance on your database. Regular maintenance helps keep performance optimal and predictable.

What other types of maintenance operations are available?

18. https://www.cybertec-postgresql.com/en/products/pg_squeeze