Extracted from:

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

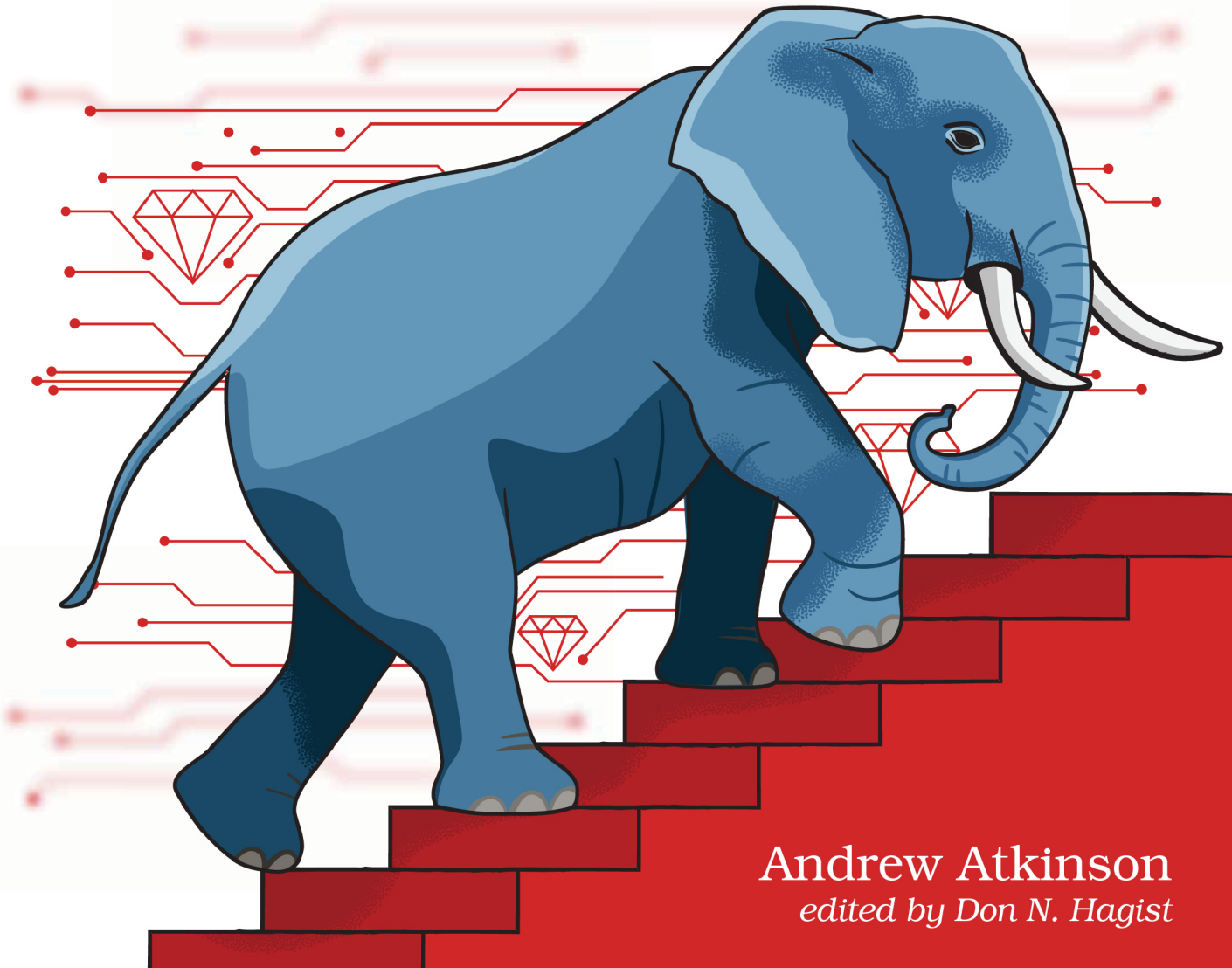
The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications



Andrew Atkinson

edited by Don N. Hagist

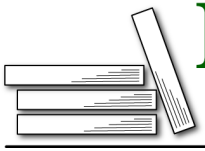
High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

Andrew Atkinson

The Pragmatic Bookshelf

Dallas, Texas



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 979-8-88865-038-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—August 30, 2023

Rebuilding Indexes Without Downtime

Periodically rebuilding an index in PostgreSQL ensures it only has pointers to live tuples. As a refresher, Indexes are secondary data stores that are owned by tables.

To rebuild Indexes in PostgreSQL you use the `REINDEX` statement. Dropping and creating the index again achieves the same outcome as it builds the index based on live tuples. But on a live system, this is unsafe because it means the index won't be available for active queries that use it.

On older versions of PostgreSQL, rebuilding indexes while the server was running was not possible. An index could be safely rebuilt (Reindexed) if the database was taken down without disrupting any queries.

Fortunately PostgreSQL has made it possible to rebuild indexes while the server is running. The index remains available to live queries that are using it.

If you're running PostgreSQL version 12 or older you won't yet have access to this capability. Upgrading to PostgreSQL 13 or newer is the best option because there are many improvements besides online index rebuilds. If upgrading is not an option, you've got alternatives. Use the extension and command line program `pg_repack`[7] to perform online index rebuilds.

The post *Using 'pg_repack' to Rebuild Indexes*[8] provides an overview of how to use pg_repack to replace Indexes online. Briefly, pg_repack can be used to rebuild an index. The technique it uses is to rebuild a replacement index and then swap in the replacement while dropping the old index in a transaction. pg_repack works well for that purpose but it is a separate client program from PostgreSQL. What if there was a built-in way to rebuild indexes without disrupting any queries that are using the Indexes?

Fortunately from PostgreSQL 13 onward, you can use the CONCURRENTLY keyword with the REINDEX command to accomplish this, rebuilding Indexes online without requiring a third party tool like pg_repack.

Try putting this into action. From psql, run the following command to reindex the `index_trips_on_driver_id` index. The Verbose is used below and it's optional but shows more information about what's happening.

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

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

Instead of Reindex with an Index, try specifying a Table to reindex all the indexes for the table. Run this statement from psql:

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

Running Manual Vacuums

PostgreSQL lays out data on disk in pages that are 8kb in size. The pages are not completely filled by default; space is left available to store new tuples from row modifications. Dead tuples take up space in those pages as well. When Autovacuum runs a VACUUM worker for each table in PostgreSQL, one of the jobs it performs is to reclaim the space used by dead tuples. Once completed, the space that was used by those dead tuples is marked as available again for new row modifications.

As a refresher, performing a manual VACUUM for a table like users is done by running this command from psql:

sql/vacuum_table.sql
```
VACUUM users;
```

The ANALYZE option can be added, and this updates table statistics for the table. The statistics include estimates that are used by the query planner as you saw earlier.

sql/vacuum_analyze_table.sql
```
VACUUM ANALYZE users;
```

New versions of PostgreSQL continue to add capabilities to Vacuum and Analyze. Version 11 made it possible to specify multiple tables at once. Version 12 added SKIP_LOCKED support which skips locked tables (See: *Postgres 12 highlight - SKIP_LOCKED for VACUUM and ANALYZE*[9]). Try running this statement from psql:

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

PostgreSQL keeps track of when Vacuum and Analyze are run both automatically and manually.

The following query shows a last_analyzed timestamp for the users table, which is when it had been last vacuumed manually. The timestamp last_auto_analyzed_at shows when the table was analyzed last by Autovacuum.

sql/last_analyzed_pg_stat_all_tables.sql
```
SELECT
```

9. https://paquier.xyz/postgresql-2/postgres-12-vacuum-skip-locked/

```
  schemaname,
  relname,
  last_autoanalyze,
  last_analyze
FROM pg_stat_all_tables
WHERE relname = 'vehicles';
```

Try running it for the vehicles table. The value for last_analyze may be empty. Run ANALYZE vehicles; and then check the last_analyze value again.

Since PostgreSQL 13, Parallel Vacuum Workers can be configured (See: *Parallel Vacuum in Upcoming PostgreSQL 13*[10]).

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

sql/set_max_parallel_maintenance_workers.sql

```
SET MAX_PARALLEL_MAINTENANCE_WORKERS=4;

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

Let's explore Bloat in greater depth.

Simulating Bloat and Understanding Impact

As you saw earlier, Bloat refers to the amount of dead tuples that are present in your system. Dead tuples consume space that could be made available for live tuples. When bloat is very high it can contribute to worse performance.

Very high bloat is not normally observed in local development databases. Dead row versions might accumulate in a high volume production system with a very high amount of Updates and Deletes and with Autovacuum not tuned and using default conservative values. This problem has lessened in newer versions of PostgreSQL and may continue to lessen in versions in the future.

Bloat management is nonetheless a common challenge, so you'll work with it a bit more in a hands on fashion.

To better understand bloat in your local development PostgreSQL server, you'll disable Autovacuum and create a very high amount of updates and deletes. By measuring the estimated bloat percentage before and after these updates, you can start to see how bloat accumulates.

10. https://www.highgo.ca/2020/02/28/parallel-vacuum-in-upcoming-postgresql-13/

To get started, collect a bloat estimate of the users table. Use the pgsql-bloat-estimation/table/table_bloat.sql[11] script from GitHub by running the statement from psql. Add the line WHERE schemaname = 'public' AND tblname = 'users' just before the ORDER BY, to limit the rows to the public.users table. The bloat_pct shows the current estimated bloat percentage. Take note of the current value.

Next, simulate bloat by creating a lot of Updates to the table. Run the following statement from psql to create 10 million updates to the users table with unique values for each update.

sql/bloat_simulate_user_updates.sql

```
UPDATE users
SET first_name =
  CASE (seq % 2)
    WHEN 0 THEN 'Bill' || FLOOR(RANDOM() * 10)
    ELSE 'Jane' || FLOOR(RANDOM() * 10)
  END
FROM GENERATE_SERIES(1,1000000) seq
WHERE id = seq;
```

This statement generates one update per row, matching the Primary Key id column to a sequence value. If you've got around 20 thousand rows, you'll see around 20 thousand updates. Each row gets a unique string that's either Bill or Jane with a number added on. Run the statement a few times to generate more updates.

Periodically check the bloat_pct column by running the query from earlier. Run some more updates. You'll see the bloat_pct growing. With these heavy amounts of updates, you're causing bloat in the table and in the Indexes.

To reclaim the space in the table, you'll need to run VACUUUM FULL users; from psql. Running VACUUM FULL on a live system is not recommended because it will lock the table while it runs.

Indexes for the users table will also become bloated and should be rebuilt. This keeps them optimized for queries.

As you saw earlier, you may run REINDEX for a single Index, or for all Indexes on a table. Run this statement from psql to rebuild one index:

sql/reindex_concurrently.sql

```
REINDEX INDEX CONCURRENTLY index_users_on_email;
```

You've now seen how to use Vacuum, Analyze, and Reindex to help with your database maintenance. What other types of Index maintenance are there?

11. https://github.com/ioguix/pgsql-bloat-estimation