Maintenance activities for PostgreSQL databases in Amazon RDS and Amazon Aurora to avoid performance issues

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Maintenance activities for PostgreSQL databases in Amazon RDS and Amazon Aurora to avoid performance issues

Table of Contents

Maintenance activities for PostgreSQL databases in Amazon RDS and Amazon Aurora to avoid performance issues

Anuradha Chintha, Rajesh Madiwale, and Srinivas Potlachervoo, Amazon Web Services (AWS)

August 2025 (document history)

Amazon Aurora PostgreSQL-Compatible Edition and Amazon Relational Database Service (Amazon RDS) for PostgreSQL are fully managed relational database services for PostgreSQL databases. These managed services free the database administrator from many maintenance and management tasks. However, some maintenance tasks, such as `VACUUM`, require close monitoring and configuration based on your database usage. This guide describes PostgreSQL maintenance activities in Amazon RDS and Aurora.

Targeted business outcomes

Database performance is a key measure that underlies the success of a business. Performing maintenance activities on your Aurora PostgreSQL-Compatible and Amazon RDS for PostgreSQL databases provides these benefits:

- Helps achieve optimal query performance
- Frees up bloated space for reuse by future transactions
- Prevents transaction wraparound
- Helps the optimizer generate good plans
- Ensures proper index usage

Multiversion concurrency control (MVCC)

PostgreSQL database maintenance requires an understanding of *multiversion concurrency control (MVCC)*, which is a mechanism of PostgreSQL. When multiple transactions are processed concurrently in the database, MVCC ensures that atomicity and isolation, which are two characteristics of atomicity, consistency, isolation, durability (ACID) transactions, are maintained. In MVCC, every write operation generates a new version of data and stores the previous version.

Readers and writers don't block one another. When a transaction reads data, the system chooses one of the versions to provide transaction isolation. PostgreSQL and some relational databases use an adaptation of MVCC called *snapshot isolation (SI)*. For example, Oracle implements SI by using rollback segments. During a write operation, Oracle writes the old version of data to a rollback segment and overwrites the data area with the new version. PostgreSQL databases implement SI by using *visibility check rules* to evaluate versions. When new data is placed into a table page, PostgreSQL uses these rules to select the appropriate version of the data for a read operation.

When you modify data in a table row, PostgreSQL uses MVCC to maintain multiple versions of the row. During `UPDATE` and `DELETE` operations on the table, the database keeps the old versions of the rows for other running transactions that might need a consistent view of the data. These old versions are called *dead rows* (*tuples*). A collection of dead tuples produces bloat. A large amount of bloat in the database can cause a number of problems, including poor query plan generation, slow query performance, and increased disk space usage to store the older versions.

Removing bloat and keeping a database healthy requires periodic maintenance, which includes these activities, which are discussed in the following sections:

- Vacuuming and analyzing tables automatically
- Vacuuming and analyzing tables manually
- Removing bloat with pg_repack
- Rebuilding indexes

Vacuuming and analyzing tables automatically

Autovacuum is a daemon (that is, it runs in the background) that automatically *vacuums* (cleans up) dead tuples, reclaims storage, and gathers statistics. It checks for bloated tables in the database and clears the bloat to reuse the space. It monitors database tables and indexes and adds them to a vacuum job after they reach a specific threshold of update or delete operations.

Autovacuum manages vacuuming by automating the PostgreSQL `VACUUM` and `ANALYZE` commands. `VACUUM` removes bloat from tables and reclaims the space, whereas `ANALYZE` updates the statistics that enable the optimizer to produce efficient plans. `VACUUM` also performs a major task called *vacuum freezing* to prevent transaction ID wraparound issues in the database. Every row that is updated in the database receives a transaction ID from the PostgreSQL transaction control mechanism. These IDs control the row's visibility to other concurrent transactions. The transaction ID is a 32-bit number. Two billion IDs are always kept in the visible past. The remaining (about 2.2 billion) IDs are preserved for transactions that will take place in the future and are hidden from the current transaction. PostgreSQL requires an occasional cleaning and *freezing* of old rows in order to prevent transactions from wrapping around and making old, existing rows invisible when new transactions are created. For more information, see Preventing Transaction ID Wraparound Failures in the PostgreSQL documentation.

Autovacuum is recommended and enabled by default. Its parameters include the following.

Parameter	Description	Default for Amazon RDS	Default for Aurora
`autovacuum_vacuum_threshold`	The minimum number of tuple update or delete operations that must occur on a table before autovacuum vacuums it.	50 operations	50 operations
`autovacuum_analyze_threshold`	The minimum number of tuple inserts, updates, or deletes that must	50 operations	50 operations

	occur on a table before autovacuum analyzes it.		
`autovacuum_vacuum_scale_factor`	The percentage of tuples that must be modified in a table before autovacuum vacuums it.	0.1	0.1
`autovacuum_analyze_scale_factor`	The percentage of tuples that must be modified in a table before autovacuum analyzes it.	0.05	0.05
`autovacuum_freeze_max_age`	The maximum age of frozen IDs before a table is vacuumed to prevent transaction ID wraparound issues.	200,000,000 transactions	200,000,000 transactions

Autovacuum makes a list of tables to process based on specific threshold formulas, as follows.

- Threshold for running `VACUUM` on a table:

```
vacuum threshold = autovacuum_vacuum_threshold + (autovacuum_vacuum_scale_factor *
 Total row count of table)
```

- Threshold for running `ANALYZE` on a table:

```
analyze threshold = autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor *
 Total row count of table)
```

For small to medium-sized tables, the default values might be sufficient. However, a large table that has frequent data modifications will have a higher number of dead tuples. In this case, autovacuum might process the table frequently for maintenance, and the maintenance of other

tables might get delayed or ignored until the large table finishes. To avoid this, you can tune the autovacuum parameters described in the following section.

Autovacuum memory-related parameters

`autovacuum_max_workers`

Specifies the maximum number of autovacuum processes (other than the autovacuum launcher) that can run at the same time. This parameter can be set only when you start the server. If the autovacuum process is busy with a large table, this parameter helps run the cleanup for other tables.

`maintenance_work_mem`

Specifies the maximum amount of memory to be used by maintenance operations such as `VACUUM`, `CREATE INDEX`, and `ALTER`. In Amazon RDS and Aurora, memory is allocated based on the instance class by using the formula `GREATEST({DBInstanceClassMemory/63963136*1024},65536)`. When autovacuum runs, up to `autovacuum_max_workers` times that calculated value can be allocated, so be careful not to set the value too high. To control this, you can set `autovacuum_work_mem` separately.

`autovacuum_work_mem`

Specifies the maximum amount of memory to be used by each autovacuum worker process. This parameter defaults to -1, which indicates that you should use the value of `maintenance_work_mem` instead.

For more information about autovacuum memory parameters, see Allocating memory for autovacuum in the Amazon RDS documentation.

Tuning autovacuum parameters

Users might need to tune autovacuum parameters depending on the their update and delete operations. The settings for the following parameters can be set at the table, instance, or cluster level.

Cluster or instance level

As an example, let's look at a banking database where continuous data manipulation language (DML) operations are expected. To maintain the database's health, you should tune autovacuum

parameters at the cluster level for Aurora and at the instance level for Amazon RDS, and apply the same parameter group to the reader as well. In the case of a failover, the same parameters should apply to the new writer.

Table level

For example, in a database for food delivery where continuous DML operations are expected on a single table called `orders`, you should consider tuning the `autovacuum_analyze_threshold` parameter at the table level by using the following command:

```
ALTER TABLE <table_name> SET (autovacuum_analyze_threshold = <threshold rows>)
```

Using aggressive autovacuum settings at the table level

The example `orders` table that has continuous update and delete operations becomes a candidate for vacuuming because of default autovacuum settings. This leads to bad plan generation and slow queries. Clearing the bloat and updating statistics requires table-level aggressive autovaccum settings.

To determine settings, keep track of the duration of queries running on this table and identify the percentage of DML operations that result in plan changes. The `pg_stat_user_tables` view helps you track insert, update, and delete operations.

Example:

Let's assume that the optimizer generates bad plans whenever 5 percent of the `orders` table changes. In this case, you should change the scale factor threshold to 2 percent as follows:

```
ALTER TABLE orders SET (autovacuum_vacuum_scale_factor = 0.02)
```

> **Tip**
>
> Select aggressive autovacuum settings carefully to avoid high consumption of resources.

For more information, see the following:

- Understanding autovacuum in Amazon RDS for PostgreSQL environments (AWS blog post)

- Automatic Vacuuming (PostgreSQL documentation)
- Tuning PostgreSQL parameters in Amazon RDS and Amazon Aurora (AWS Prescriptive Guidance)

To make sure that autovacuum works effectively, monitor dead rows, disk usage, and the last time autovacuum or `ANALYZE` ran on a regular basis. The `pg_stat_all_tables` view provides information on each table (`relname`) and how many dead tuples (`n_dead_tup`) are in the table.

Monitoring the number of dead tuples in each table, especially in frequently updated tables, helps you determine if the autovacuum processes are periodically removing the dead tuples so their disk space can be reused for better performance. You can use the following query to check the number of dead tuples and when the last autovacuum ran on the tables:

```
SELECT
relname AS TableName,n_live_tup AS LiveTuples,n_dead_tup AS DeadTuples,
last_autovacuum AS Autovacuum,last_autoanalyze AS Autoanalyze_FROM
pg_stat_user_tables;
```

Advantages and limitations

Autovacuum provides the following advantages:

- It removes bloat from tables automatically.
- It prevents transaction ID wraparound.
- It keeps database statistics up to date.

Limitations:

- If queries use parallel processing, the number of worker processes might not be enough for autovacuum.
- If autovacuum runs during peak hours, resource utilization might increase. You should tune parameters to handle this issue.
- If table pages are occupied in another session, autovacuum might skip those pages.
- Autovacuum can't access temporary tables.

Vacuuming and analyzing tables manually

If your database is vacuumed by the autovacuum process, it's best practice to avoid running manual vacuums on the entire database too frequently. A manual vacuum might result in unnecessary I/O loads or CPU spikes, and might also fail to remove any dead tuples. Run manual vacuums on a table-by-table basis only if it's really necessary, such as when the ratio of live to dead tuples is low, or when there are long gaps between autovacuums. In addition, you should run manual vacuums when there's minimal user activity.

Autovacuum also keeps a table's statistics up to date. When you run the `ANALYZE` command manually, it rebuilds these statistics instead of updating them. Rebuilding statistics when they are already updated by the regular autovacuum process might cause system resource utilization.

We recommend that you run the VACUUM and ANALYZE commands manually in the following scenarios:

- During low peak hours on busier tables, when autovacuuming might not be sufficient.
- Immediately after you bulk load data into the target table. In this case, running `ANALYZE` manually completely rebuilds statistics, which is a better option than waiting for autovacuum to begin.
- To vacuum temporary tables (autovacuum can't access these).

To reduce the I/O impact when you run the `VACUUM` and `ANALYZE` commands on concurrent database activity, you can use the `vacuum_cost_delay` parameter. In many situations, maintenance commands such as `VACUUM` and `ANALYZE` don't have to finish quickly. However, these commands shouldn't interfere with the system's ability to perform other database operations. To prevent this, you can enable cost-based vacuum delays by using the `vacuum_cost_delay` parameter. This parameter is disabled by default for manually issued `VACUUM` commands. To enable it, set it to a nonzero value.

Running vacuum and cleanup operations in parallel

The `VACUUM` command PARALLEL option uses parallel workers for the index vacuum and index cleanup phases and is disabled by default. The number of parallel workers (the degree of parallelism) is determined by the number of indexes in the table and can be specified by the user. If

you're running parallel VACUUM operations without an integer argument, the degree of parallelism is calculated based on the number of indexes in the table.

The following parameters help you configure parallel vacuuming in Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible:

- max_worker_processes sets the maximum number of concurrent worker processes.
- min_parallel_index_scan_size sets the minimum amount of index data that must be scanned in order for a parallel scan to be considered.
- max_parallel_maintenance_workers sets the maximum number of parallel workers that can be started by a single utility command.

> **Note**
>
> The PARALLEL option is used only for vacuuming purposes. It doesn't affect the ANALYZE command.

The following example illustrates database behavior when you use manual VACUUM and ANALYZE on a database.

Here's a sample table where autovacuum has been disabled (for illustration purposes only; disabling autovacuum isn't recommended):

```
create table t1 ( a int, b int, c int );
alter table t1 set (autovacuum_enabled=false);
```

```
apgl=> \d+ t1
Table "public.t1"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------
+-------------
a | integer | | | | plain | |
b | integer | | | | plain | |
c | integer | | | | plain | |
Access method: heap
Options: autovacuum_enabled=false
```

Add 1 million rows to table t1:

```
apgl=> select count(*) from t1;
count
1000000
(1 row)
```

Statistics of table t1:

```
select * from pg_stat_all_tables where relname='t1';
-[ RECORD 1 ]-------+--------
relid         | 914744
schemaname    | public
relname       | t1
seq_scan      | 0
seq_tup_read  | 0
idx_scan      |
idx_tup_fetch |
n_tup_ins     | 1000000
n_tup_upd     | 0
n_tup_del     | 0
n_tup_hot_upd | 0
n_live_tup    | 1000000
n_dead_tup    | 0
n_mod_since_analyze | 1000000
last_vacuum      |
last_autovacuum  |
last_analyze     |
last_autoanalyze |
vacuum_count     | 0
autovacuum_count | 0
analyze_count    | 0
autoanalyze_count | 0
```

Add an index:

```
create index i2 on t1 (b,a);
```

Run the EXPLAIN command (Plan 1):

```
Bitmap Heap Scan on t1 (cost=10521.17..14072.67 rows=5000 width=4)
```

```
Recheck Cond: (a = 5)
# Bitmap Index Scan on i2 (cost=0.00..10519.92 rows=5000 width=0)
Index Cond: (a = 5)
(4 rows)
```

Run the `EXPLAIN ANALYZE` command (Plan 2):

```
explain (analyze,buffers,costs off) select a from t1 where b = 5;
QUERY PLAN
Bitmap Heap Scan on t1 (actual time=0.023..0.024 rows=1 loops=1)
Recheck Cond: (b = 5)
Heap Blocks: exact=1
Buffers: shared hit=4
# Bitmap Index Scan on i2 (actual time=0.016..0.016 rows=1 loops=1)
Index Cond: (b = 5)
Buffers: shared hit=3
Planning Time: 0.054 ms
Execution Time: 0.076 ms
(9 rows)
```

The `EXPLAIN` and `EXPLAIN ANALYZE` commands display different plans, because autovacuum was disabled on the table and the `ANALYZE` command wasn't performed manually. Now let's update a value in the table and regenerate the `EXPLAIN ANALYZE` plan:

```
update t1 set a=8 where b=5;
explain (analyze,buffers,costs off) select a from t1 where b = 5;
```

The `EXPLAIN ANALYZE` command (Plan 3) now displays:

```
apgl=> explain (analyze,buffers,costs off) select a from t1 where b = 5;
QUERY PLAN
Bitmap Heap Scan on t1 (actual time=0.075..0.076 rows=1 loops=1)
Recheck Cond: (b = 5)
Heap Blocks: exact=1
Buffers: shared hit=5
# Bitmap Index Scan on i2 (actual time=0.017..0.017 rows=2 loops=1)
Index Cond: (b = 5)
Buffers: shared hit=3
Planning Time: 0.053 ms
Execution Time: 0.125 ms
```

If you compare the costs between Plan 2 and Plan 3 you will see the differences in planning and execution time, because we haven't collected statistics yet.

Now let's run a manual ANALYZE on the table, and then check the statistics and regenerate the plan:

```
apgl=> analyze t1
apgl# ;
ANALYZE
Time: 212.223 ms

apgl=> select * from pg_stat_all_tables where relname='t1';
-[ RECORD 1 ]-------+------------------------------
relid          | 914744
schemaname     | public
relname        | t1
seq_scan       | 3
seq_tup_read   | 1000000
idx_scan       | 3
idx_tup_fetch  | 3
n_tup_ins      | 1000000
n_tup_upd      | 1
n_tup_del      | 0
n_tup_hot_upd  | 0
n_live_tup     | 1000000
n_dead_tup     | 1
n_mod_since_analyze | 0
last_vacuum       |
last_autovacuum   |
last_analyze      | 2023-04-15 11:39:02.075089+00
last_autoanalyze  |
vacuum_count      | 0
autovacuum_count  | 0
analyze_count     | 1
autoanalyze_count | 0

Time: 148.347 ms
```

Run the EXPLAIN ANALYZE command (Plan 4):

```
apgl=> explain (analyze,buffers,costs off) select a from t1 where b = 5;
QUERY PLAN
```

```
Index Only Scan using i2 on t1 (actual time=0.022..0.023 rows=1 loops=1)
Index Cond: (b = 5)
Heap Fetches: 1
Buffers: shared hit=4
Planning Time: 0.056 ms
Execution Time: 0.068 ms
(6 rows)

Time: 138.462 ms
```

If you compare all plan results after you manually analyze the table and collect statistics, you'll notice that the optimizer's Plan 4 is better than the others and also decreases query execution time. This example shows how important it is to run maintenance activities on the database.

Rewriting an entire table with VACUUM FULL

Running the `VACUUM` command with the `FULL` parameter rewrites the entire contents of a table into a new disk file with no extra space, and returns unused space to the operating system. This operation is much slower and requires an `ACCESS EXCLUSIVE` lock on each table. It also requires extra disk space, because it writes a new copy of the table and doesn't release the old copy until the operation is complete.

`VACUUM FULL` can be useful in the following cases:

- When you want to reclaim a significant amount of space from the tables.
- When you want to reclaim bloat space in non-primary key tables.

We recommend that you use `VACUUM FULL` when you have non-primary key tables, if your database can tolerate downtime.

Because `VACUUM FULL` requires more locking than other operations, it is more expensive to run on crucial databases. To replace this method, you can use the `pg_repack` extension, which is described in the next section. This option is similar to `VACUUM FULL` but requires minimal locking and is supported by both Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible.

Removing bloat with pg_repack

You can use the pg_repack extension to remove table and index bloat with minimal database locking. You can create this extension in the database instance and run the pg_repack client (where the client version matches the extension version) from Amazon Elastic Compute Cloud (Amazon EC2) or from a computer that can connect to your database.

Unlike VACUUM FULL, pg_repack doesn't require downtime or a maintenance window, and won't block other sessions.

pg_repack is helpful in situations where VACUUM FULL, CLUSTER, or REINDEX might not work. It creates a new table that contains the data of the bloated table, tracks the changes from the original table, and then replaces the original table with the new one. It doesn't lock the original table for read or write operations while it's building the new table.

You can use pg_repack for a full table or for an index. To see a list of tasks, see the pg_repack documentation.

Limitations:

- To run pg_repack, your table must have a primary key or a unique index.
- pg_repack won't work with temporary tables.
- pg_repack won't work on tables that have global indexes.
- When pg_repack is in progress, you can't perform DDL operations on tables.

The following table describes the differences between pg_repack and VACUUM FULL.

VACUUM FULL	pg_repack
Built-in command	An extension that you run from Amazon EC2 or your local computer
Requires an ACCESS EXCLUSIVE lock while it's working on a table	Requires an ACCESS EXCLUSIVE lock only for a short time
Works with all tables	Works on tables that have primary and unique keys only

Requires double the storage that's consumed by the table and indexes	Requires double the storage that's consumed by the table and indexes

To run `pg_repack` on a table, use the command:

```
pg_repack -h <host> -d <dbname> --table <tablename> -k
```

To run `pg_repack` on an index, use the command:

```
pg_repack -h <host> -d <dbname> --index <index name>
```

For more information, see the AWS blog post Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack.

Caveat

The `error-on-invalid-index` error in `pg_repack` usually means that one or more indexes on the table are corrupt or invalid. `pg_repack` cannot safely operate on tables that have invalid indexes, because it relies on the indexes for data consistency during the repack process.
This error occurs when:

- The index is marked as invalid (for example, because of a failed `CREATE INDEX CONCURRENTLY` statement).
- The index is corrupted (possibly due to hardware issues or abrupt shutdowns).

Use the following query to identify invalid indexes and to drop them first if you find them.

```
SELECT indexrelid::regclass, indisvalid FROM pg_index WHERE indrelid =
 'orders'::regclass AND NOT indisvalid; Drop the invalid index: DROP INDEX
 index_name;
```

Rebuilding indexes

The PostgreSQL REINDEX command rebuilds an index by using the data that's stored in the index's table and replacing the old copy of the index. We recommend that you use `REINDEX` in the following scenarios:

- When an index becomes corrupted and no longer contains valid data. This can happen as a result of software or hardware failures.
- When queries that previously used the index stop using it.
- When the index becomes bloated with a large number of empty or nearly empty pages. You should run `REINDEX` when the bloat percentage (`bloat_pct`) is greater than 20.

The following query helps you find `bloat_pct`:

```
SELECT current_database(), nspname AS schemaname, tblname, idxname,
 bs*(relpages)::bigint AS real_size,
  bs*(relpages-est_pages)::bigint AS extra_size,
  100 * (relpages-est_pages)::float / relpages AS extra_pct,
  fillfactor,
  CASE WHEN relpages > est_pages_ff
    THEN bs*(relpages-est_pages_ff)
    ELSE 0
  END AS bloat_size,
  100 * (relpages-est_pages_ff)::float / relpages AS bloat_pct,
  is_na
  -- , 100-(pst).avg_leaf_density AS pst_avg_bloat, est_pages, index_tuple_hdr_bm,
 maxalign, pagehdr, nulldatawidth, nulldatahdrwidth, reltuples, relpages -- (DEBUG
 INFO)
FROM (
  SELECT coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)/(4+nulldatahdrwidth)::float)), 0
 -- ItemIdData size + computed avg size of a tuple (nulldatahdrwidth)
      ) AS est_pages,
      coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)*fillfactor/
(100*(4+nulldatahdrwidth)::float))), 0
      ) AS est_pages_ff,
      bs, nspname, tblname, idxname, relpages, fillfactor, is_na
```

```
      -- , pgstatindex(idxoid) AS pst, index_tuple_hdr_bm, maxalign, pagehdr,
 nulldatawidth, nulldatahdrwidth, reltuples -- (DEBUG INFO)
  FROM (
      SELECT maxalign, bs, nspname, tblname, idxname, reltuples, relpages, idxoid,
 fillfactor,
            ( index_tuple_hdr_bm +
                maxalign - CASE -- Add padding to the index tuple header to align on
 MAXALIGN
                  WHEN index_tuple_hdr_bm%maxalign = 0 THEN maxalign
                  ELSE index_tuple_hdr_bm%maxalign
                END
              + nulldatawidth + maxalign - CASE -- Add padding to the data to align on
 MAXALIGN
                  WHEN nulldatawidth = 0 THEN 0
                  WHEN nulldatawidth::integer%maxalign = 0 THEN maxalign
                  ELSE nulldatawidth::integer%maxalign
                END
            )::numeric AS nulldatahdrwidth, pagehdr, pageopqdata, is_na
            -- , index_tuple_hdr_bm, nulldatawidth -- (DEBUG INFO)
      FROM (
          SELECT n.nspname, i.tblname, i.idxname, i.reltuples, i.relpages,
              i.idxoid, i.fillfactor, current_setting('block_size')::numeric AS bs,
              CASE -- MAXALIGN: 4 on 32bits, 8 on 64bits (and mingw32 ?)
                WHEN version() ~ 'mingw32' OR version() ~ '64-bit|x86_64|ppc64|ia64|
amd64' THEN 8
                ELSE 4
              END AS maxalign,
              /* per page header, fixed size: 20 for 7.X, 24 for others */
              24 AS pagehdr,
              /* per page btree opaque data */
              16 AS pageopqdata,
              /* per tuple header: add IndexAttributeBitMapData if some cols are null-
able */
              CASE WHEN max(coalesce(s.null_frac,0)) = 0
                  THEN 8 -- IndexTupleData size
                  ELSE 8 + (( 32 + 8 - 1 ) / 8) -- IndexTupleData size +
 IndexAttributeBitMapData size ( max num filed per index + 8 - 1 /8)
              END AS index_tuple_hdr_bm,
              /* data len: we remove null values save space using it fractionnal part
 from stats */
              sum( (1-coalesce(s.null_frac, 0)) * coalesce(s.avg_width, 1024)) AS
 nulldatawidth,
```

```
            max( CASE WHEN i.atttypid = 'pg_catalog.name'::regtype THEN 1 ELSE 0
END ) > 0 AS is_na
         FROM (
            SELECT ct.relname AS tblname, ct.relnamespace, ic.idxname, ic.attpos,
ic.indkey, ic.indkey[ic.attpos], ic.reltuples, ic.relpages, ic.tbloid, ic.idxoid,
ic.fillfactor,
                coalesce(a1.attnum, a2.attnum) AS attnum, coalesce(a1.attname,
a2.attname) AS attname, coalesce(a1.atttypid, a2.atttypid) AS atttypid,
                CASE WHEN a1.attnum IS NULL
                THEN ic.idxname
                ELSE ct.relname
                END AS attrelname
            FROM (
                SELECT idxname, reltuples, relpages, tbloid, idxoid, fillfactor,
indkey,
                    pg_catalog.generate_series(1,indnatts) AS attpos
                FROM (
                    SELECT ci.relname AS idxname, ci.reltuples, ci.relpages,
i.indrelid AS tbloid,
                        i.indexrelid AS idxoid,
                        coalesce(substring(
                            array_to_string(ci.reloptions, ' ')
                            from 'fillfactor=([0-9]+)')::smallint, 90) AS fillfactor,
                        i.indnatts,
                        pg_catalog.string_to_array(pg_catalog.textin(
                            pg_catalog.int2vectorout(i.indkey)),' ')::int[] AS indkey
                    FROM pg_catalog.pg_index i
                    JOIN pg_catalog.pg_class ci ON ci.oid = i.indexrelid
                    WHERE ci.relam=(SELECT oid FROM pg_am WHERE amname = 'btree')
                    AND ci.relpages > 0
                ) AS idx_data
            ) AS ic
            JOIN pg_catalog.pg_class ct ON ct.oid = ic.tbloid
            LEFT JOIN pg_catalog.pg_attribute a1 ON
                ic.indkey[ic.attpos] <> 0
                AND a1.attrelid = ic.tbloid
                AND a1.attnum = ic.indkey[ic.attpos]
            LEFT JOIN pg_catalog.pg_attribute a2 ON
                ic.indkey[ic.attpos] = 0
                AND a2.attrelid = ic.idxoid
                AND a2.attnum = ic.attpos
        ) i
        JOIN pg_catalog.pg_namespace n ON n.oid = i.relnamespace
```

```
            JOIN pg_catalog.pg_stats s ON s.schemaname = n.nspname
                                      AND s.tablename = i.attrelname
                                      AND s.attname = i.attname
            GROUP BY 1,2,3,4,5,6,7,8,9,10,11
      ) AS rows_data_stats
  ) AS rows_hdr_pdg_stats
) AS relation_stats
ORDER BY nspname, tblname, idxname;
```

Index pages that are completely empty are reclaimed for reuse. However, we recommend periodic reindexing if the index keys on a page have been deleted but space remains allocated.

Recreating the index helps provide better query performance. You can recreate an index in three ways, as described in the following table.

Method	Description	Limitations
`CREATE INDEX` and `DROP INDEX` with the `CONCURRENTLY` option	Builds a new index and removes the old index. The optimizer generates plans by using the newly created index instead of the old index. During low peak hours, you can drop the old index.	Index creation take more time when you use the `CONCURRENTLY` option, because it has to track all incoming changes. When changes are frozen, the process is marked as complete.
`REINDEX` with the `CONCURRENTLY` option	Locks write operations during the rebuild process. PostgreSQL version 12 and later versions provide the `CONCURRENTLY` option, which avoids these locks.	Using `CONCURRENTLY` requires a longer time to rebuild the index.
`pg_repack` extension	Cleans the bloat from a table and rebuilds the index.	You must run this extension from an EC2 instance or your local computer that is connected to the database.

Creating a new index

The DROP INDEX and CREATE INDEX commands, when used together, rebuild an index:

```
DROP INDEX <index_name>
CREATE INDEX <index_name> ON TABLE <table_name> (<column1>[,<column2>])
```

The disadvantage of this approach is its exclusive lock on the table, which impacts performance during this activity. The DROP INDEX command acquires an exclusive lock, which blocks both read and write operations on the table. The CREATE INDEX command blocks the write operations on the table. It allows read operations, but these are expensive during index creation.

Rebuilding an index

The REINDEX command helps you maintain consistent database performance. When you perform a large number of DML operations on a table, these result in both table and index bloat. Indexes are used to speed lookup on tables to improve query performance. Index bloating affects lookups and query performance. Therefore, we recommend that you perform reindexing on tables that have a high volume of DML operations to maintain consistency in the performance of queries.

The REINDEX command rebuilds the index from scratch by locking the write operations on the underlying table, but it allows read operations on the table. However, it does block the read operations on the index. Queries that use the corresponding index are blocked, but other queries aren't.

PostgreSQL version 12 introduced a new optional parameter, CONCURRENTLY, which rebuilds the index from scratch but doesn't lock the write or read operations on the table or on queries that use the index. However, it takes a longer time to complete the process when you use this option.

Examples

Creating and dropping an index

Create a new index with the CONCURRENTLY option:

```
create index CONCURRENTLY on table(columns) ;
```

Drop the old index with the CONCURRENTLY option:

```
drop index CONCURRENTLY <index name> ;
```

Rebuilding an index

To rebuild a single index:

```
reindex index <index name> ;
```

To rebuild all indexes in a table:

```
reindex table <table name> ;
```

To rebuild all indexes in a schema:

```
reindex schema <schema name> ;
```

Rebuilding an index concurrently

To rebuild a single index:

```
reindex index CONCURRENTLY <indexname> ;
```

To rebuild all indexes in a table:

```
reindex table CONCURRENTLY <tablename> ;
```

To rebuild all indexes in a schema:

```
reindex schema CONCURRENTLY <schemaname> ;
```

Rebuilding or relocating indexes only

To rebuild a single index:

```
pg_repack -h <hostname> -d <dbname> -i <indexname> -k
```

To rebuild all indexes:

```
pg_repack -h <hostname> -d <dbname> -x <indexname> -t <tablename> -k
```

Example: Reclaiming space by using autovacuum and VACUUM FULL

As an example, let's create an emp table table with 500,000 rows, and then update the rows with new values. Autovacuum is enabled, so it will run both VACUUM and ANALYZE commands on this table to remove bloat and reclaim space. The reclaimed space can be reused but it won't be returned to the operating system.

The following query determines the bloat on the table:

```
-- WARNING: When run with a non-superuser role, the query inspects only indexes on
 tables you are granted to read.
-- WARNING: Rows with is_na = 't' are known to have bad statistics ("name" type is not
 supported).
-- This query is compatible with PostgreSQL 8.2 and later.
SELECT current_database(), nspname AS schemaname, tblname, idxname,
 bs*(relpages)::bigint AS real_size,
  bs*(relpages-est_pages)::bigint AS extra_size,
  100 * (relpages-est_pages)::float / relpages AS extra_pct,
  fillfactor,
  CASE WHEN relpages > est_pages_ff
    THEN bs*(relpages-est_pages_ff)
    ELSE 0
  END AS bloat_size,
  100 * (relpages-est_pages_ff)::float / relpages AS bloat_pct,
  is_na
  -- , 100-(pst).avg_leaf_density AS pst_avg_bloat, est_pages, index_tuple_hdr_bm,
 maxalign, pagehdr, nulldatawidth, nulldatahdrwidth, reltuples, relpages -- (DEBUG
 INFO)
FROM (
  SELECT coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)/(4+nulldatahdrwidth)::float)), 0
 -- ItemIdData size + computed avg size of a tuple (nulldatahdrwidth)
      ) AS est_pages,
      coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)*fillfactor/
(100*(4+nulldatahdrwidth)::float))), 0
      ) AS est_pages_ff,
      bs, nspname, tblname, idxname, relpages, fillfactor, is_na
```

```
      -- , pgstatindex(idxoid) AS pst, index_tuple_hdr_bm, maxalign, pagehdr,
 nulldatawidth, nulldatahdrwidth, reltuples -- (DEBUG INFO)
  FROM (
      SELECT maxalign, bs, nspname, tblname, idxname, reltuples, relpages, idxoid,
 fillfactor,
            ( index_tuple_hdr_bm +
                maxalign - CASE -- Add padding to the index tuple header to align on
 MAXALIGN
                  WHEN index_tuple_hdr_bm%maxalign = 0 THEN maxalign
                  ELSE index_tuple_hdr_bm%maxalign
                END
              + nulldatawidth + maxalign - CASE -- Add padding to the data to align on
 MAXALIGN
                  WHEN nulldatawidth = 0 THEN 0
                  WHEN nulldatawidth::integer%maxalign = 0 THEN maxalign
                  ELSE nulldatawidth::integer%maxalign
                END
            )::numeric AS nulldatahdrwidth, pagehdr, pageopqdata, is_na
            -- , index_tuple_hdr_bm, nulldatawidth -- (DEBUG INFO)
      FROM (
          SELECT n.nspname, i.tblname, i.idxname, i.reltuples, i.relpages,
              i.idxoid, i.fillfactor, current_setting('block_size')::numeric AS bs,
              CASE -- MAXALIGN: 4 on 32bits, 8 on 64bits (and mingw32 ?)
                WHEN version() ~ 'mingw32' OR version() ~ '64-bit|x86_64|ppc64|ia64|
amd64' THEN 8
                ELSE 4
              END AS maxalign,
              /* per page header, fixed size: 20 for 7.X, 24 for others */
              24 AS pagehdr,
              /* per page btree opaque data */
              16 AS pageopqdata,
              /* per tuple header: add IndexAttributeBitMapData if some cols are null-
able */
              CASE WHEN max(coalesce(s.null_frac,0)) = 0
                  THEN 8 -- IndexTupleData size
                  ELSE 8 + (( 32 + 8 - 1 ) / 8) -- IndexTupleData size +
 IndexAttributeBitMapData size ( max num filed per index + 8 - 1 /8)
              END AS index_tuple_hdr_bm,
              /* data len: we remove null values save space using it fractionnal part
 from stats */
              sum( (1-coalesce(s.null_frac, 0)) * coalesce(s.avg_width, 1024)) AS
 nulldatawidth,
```

```
            max( CASE WHEN i.atttypid = 'pg_catalog.name'::regtype THEN 1 ELSE 0
END ) > 0 AS is_na
         FROM (
            SELECT ct.relname AS tblname, ct.relnamespace, ic.idxname, ic.attpos,
ic.indkey, ic.indkey[ic.attpos], ic.reltuples, ic.relpages, ic.tbloid, ic.idxoid,
ic.fillfactor,
                coalesce(a1.attnum, a2.attnum) AS attnum, coalesce(a1.attname,
a2.attname) AS attname, coalesce(a1.atttypid, a2.atttypid) AS atttypid,
                CASE WHEN a1.attnum IS NULL
                THEN ic.idxname
                ELSE ct.relname
                END AS attrelname
            FROM (
                SELECT idxname, reltuples, relpages, tbloid, idxoid, fillfactor,
indkey,
                    pg_catalog.generate_series(1,indnatts) AS attpos
                FROM (
                    SELECT ci.relname AS idxname, ci.reltuples, ci.relpages,
i.indrelid AS tbloid,
                        i.indexrelid AS idxoid,
                        coalesce(substring(
                            array_to_string(ci.reloptions, ' ')
                            from 'fillfactor=([0-9]+)')::smallint, 90) AS fillfactor,
                        i.indnatts,
                        pg_catalog.string_to_array(pg_catalog.textin(
                            pg_catalog.int2vectorout(i.indkey)),' ')::int[] AS indkey
                    FROM pg_catalog.pg_index i
                    JOIN pg_catalog.pg_class ci ON ci.oid = i.indexrelid
                    WHERE ci.relam=(SELECT oid FROM pg_am WHERE amname = 'btree')
                    AND ci.relpages > 0
                ) AS idx_data
            ) AS ic
            JOIN pg_catalog.pg_class ct ON ct.oid = ic.tbloid
            LEFT JOIN pg_catalog.pg_attribute a1 ON
                ic.indkey[ic.attpos] <> 0
                AND a1.attrelid = ic.tbloid
                AND a1.attnum = ic.indkey[ic.attpos]
            LEFT JOIN pg_catalog.pg_attribute a2 ON
                ic.indkey[ic.attpos] = 0
                AND a2.attrelid = ic.idxoid
                AND a2.attnum = ic.attpos
          ) i
          JOIN pg_catalog.pg_namespace n ON n.oid = i.relnamespace
```

```
            JOIN pg_catalog.pg_stats s ON s.schemaname = n.nspname
                                      AND s.tablename = i.attrelname
                                      AND s.attname = i.attname
            GROUP BY 1,2,3,4,5,6,7,8,9,10,11
      ) AS rows_data_stats
  ) AS rows_hdr_pdg_stats
) AS relation_stats
ORDER BY nspname, tblname, idxname;
```

The results of the query show that the table has a bloat of around 51 percent:

```
current_database | schemaname | tblname | real_size | extra_size | extra_pct |
 fillfactor | bloat_size | bloat_pct | is_na
------------------+------------+---------+-----------+------------+------------------
+------------+------------+-------------------+-------
apgl | public | emp | 60383232 | 30744576 | 50.91575091575091 | 100 | 30744576 |
 50.91575091575091 | f
```

Here are the statistics from the `pg_stat_all_tables` view:

```
relid           | 914748
schemaname      | public
relname         | emp
seq_scan        | 5
seq_tup_read    | 1500000
idx_scan        | 0
idx_tup_fetch   | 0
n_tup_ins       | 600000
n_tup_upd       | 500000
n_tup_del       | 0
n_tup_hot_upd   | 0
n_live_tup      | 500000
n_dead_tup      | 0
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     | 2023-04-15 11:59:54.957449+00
last_analyze        |
last_autoanalyze    | 2023-04-15 11:59:55.016352+00
vacuum_count        | 0
autovacuum_count    | 2
analyze_count       | 0
autoanalyze_count   | 3
```

Notice that autovacuum updated the `last_autovacuum` and `last_autoanalyze` columns after it ran.

Now, let's insert some rows into the table and check `extra_size(bloat_size)`, because the empty space is also considered bloat.

```
apgl=> select count(*) from emp;
count | 900000

current_database | schemaname | tblname | real_size | extra_size | extra_pct |
 fillfactor | bloat_size | bloat_pct | is_na
------------------+------------+---------+-----------+------------+-------------------
+------------+------------+--------------------+-------
apgl | public | emp | 61349888 | 327680 | 0.5341167044999332 | 100 | 327680 |
 0.5341167044999332 | f
(1 row)
```

The `bloat_pct` column in the output indicates that the cleaned space is been occupied by new inserts. Let's run `VACUUM FULL`:

```
apgl=> vacuum full emp ;
VACUUM

current_database | schemaname | tblname | real_size | extra_size | extra_pct |
 fillfactor | bloat_size | bloat_pct | is_na
------------------+------------+---------+-----------+------------+-----------
+------------+------------+-----------+-------
apgl | public | emp | 60792832 | -229376 | 0 | 100 | 0 | 0 | f
(1 row)
```

From this output, you can see that the the empty space and the bloat has been removed and the space has been returned to the operating system.

> **Note**
>
> Instead of `VACUUM FULL`, you could run `pg_repack` to get the same results.

Resources

- Understanding autovacuum in Amazon RDS for PostgreSQL environments (AWS blog post)
- Automatic vacuuming (PostgreSQL documentation)
- Allocating memory for autovacuum (Amazon RDS documentation)
- Preventing Transaction ID Wraparound Failures; (PostgreSQL documentation)
- Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack (AWS blog post)

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Updates	Corrected errors and added information to the Vacuuming and analyzing tables automatically and Removing bloat with pg_repack sections.	August 22, 2025
Corrected `reindex` syntax	In the section for rebuilding an index concurrently, corrected the `reindex` examples.	June 30, 2025
Initial publication	—	December 22, 2023