**Vacuuming & Autovacuum Configuration in PostgreSQL**

**1. Introduction**

PostgreSQL uses **Multi-Version Concurrency Control (MVCC)** to handle concurrent transactions. This results in the creation of **dead tuples**—obsolete row versions that are no longer visible to any active transaction.

To reclaim space and maintain performance, PostgreSQL uses two primary mechanisms:

* **Manual VACUUM**: User-initiated cleanup.
* **Autovacuum Daemon**: Background process that automatically vacuums tables based on configuration.

**2. What Does VACUUM Do?**

**VACUUM Functions**

* Reclaims storage by marking dead tuples as reusable.
* Updates statistics for the query planner (with ANALYZE).
* Prevents transaction ID (XID) wraparound.

**Types of VACUUM**

| **Type** | **Description** |
| --- | --- |
| VACUUM | Removes dead tuples and marks space for reuse. |
| VACUUM FULL | Rewrites the entire table, reclaiming disk space. Requires exclusive lock. |

**3. Autovacuum Daemon**

**What is Autovacuum?**

Autovacuum is a background process that:

* Runs VACUUM and optionally ANALYZE on tables.
* Automatically chooses when and where to run based on thresholds.
* Helps prevent transaction ID wraparound.

**Autovacuum Must Be Enabled (default)**

Check with:

SHOW autovacuum;

**Threshold Formula**

vacuum threshold = autovacuum\_vacuum\_threshold + autovacuum\_vacuum\_scale\_factor \* reltuples

**Per-Table Configuration**

Use ALTER TABLE to customize autovacuum per table:

ALTER TABLE logs

SET (autovacuum\_vacuum\_threshold = 1000,

autovacuum\_vacuum\_scale\_factor = 0.05);

**5. Monitoring Vacuum Activity**

**Check vacuum stats**

SELECT \* FROM pg\_stat\_user\_tables;

Useful columns:

* last\_vacuum
* last\_autovacuum
* n\_dead\_tup (number of dead tuples)
* vacuum\_count, autovacuum\_count

**Logging Autovacuum**

Enable logging in postgresql.conf:

log\_autovacuum\_min\_duration = 0 # Log all autovacuums

**6. When to Tune Autovacuum**

You may need to tune autovacuum if:

* Tables grow large rapidly (e.g., event logs).
* You notice bloated tables or slow queries.
* You get wraparound warnings.
* Too many dead tuples accumulate (n\_dead\_tup).

**7. VACUUM FULL and Table Bloat**

**What is Table Bloat?**

If dead tuples are not cleaned up, tables can become bloated—using more disk and memory than necessary.

**When to Use VACUUM FULL**

* When reclaiming disk space is a priority.
* After massive deletions or updates.
* As part of scheduled maintenance.

VACUUM FULL tablename;

It locks the table and rewrites it, so avoid using it in high-availability periods.

**8. Best Practices**

* Monitor regularly using pg\_stat\_user\_tables and pg\_stat\_activity.
* Log and review autovacuum activity.
* Customize autovacuum for high-write tables.
* Use VACUUM ANALYZE after batch inserts or updates.
* Schedule VACUUM FULL only when needed and during low-traffic periods.
* Keep autovacuum\_freeze\_max\_age under control to prevent wraparound risks.

**10. Example Scenario: High Insert Table**

**Problem:**

Table logs grows rapidly. Autovacuum isn’t keeping up.

**Solution:**

ALTER TABLE logs

SET (

autovacuum\_vacuum\_threshold = 100,

autovacuum\_vacuum\_scale\_factor = 0.01,

autovacuum\_naptime = '30s'

);

Add index for performance and schedule occasional VACUUM FULL during maintenance.

**Understanding Dead Tuples in PostgreSQL**

**What Are Dead Tuples?**

PostgreSQL uses **MVCC (Multi-Version Concurrency Control)** to ensure data consistency during concurrent transactions.

* When you **UPDATE** or **DELETE** a row, PostgreSQL **does not immediately remove it**.
* Instead, it **marks the old row as dead** and creates a new row version (for UPDATE) or leaves it as-is (for DELETE).
* These obsolete rows are **dead tuples**—they’re invisible to new transactions but still occupy space.

**Example of Dead Tuples**

**Step 1: Create a Table**

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name TEXT,

department TEXT

);

**Step 2: Insert Some Rows**

INSERT INTO employees (name, department)

VALUES

('Alice', 'HR'),

('Bob', 'Finance'),

('Charlie', 'IT');

**Step 3: Update a Row**

UPDATE employees SET department = 'Legal' WHERE name = 'Bob';

* PostgreSQL creates a **new row version** with department = 'Legal'.
* The old version (department = 'Finance') becomes a **dead tuple**.

**Step 4: Delete a Row**

DELETE FROM employees WHERE name = 'Charlie';

* The row is **not physically deleted**.
* It becomes a **dead tuple** until vacuumed.

**Checking for Dead Tuples**

Run the following to inspect tuple stats:

SELECT

relname AS table,

n\_tup\_ins AS inserts,

n\_tup\_upd AS updates,

n\_tup\_del AS deletes,

n\_live\_tup AS live,

n\_dead\_tup AS dead

FROM pg\_stat\_user\_tables

WHERE relname = 'employees';

**Sample Output:**

| **table** | **inserts** | **updates** | **deletes** | **live** | **dead** |
| --- | --- | --- | --- | --- | --- |
| employees | 3 | 1 | 1 | 2 | 2 |

* 2 live rows (Alice and updated Bob)
* 2 dead tuples (old Bob and deleted Charlie)

**Reclaiming Dead Tuples with VACUUM**

VACUUM employees;

* Marks dead tuples' space as **available for reuse**.
* Doesn’t shrink disk size (use VACUUM FULL for that).

**VACUUM FULL**

VACUUM FULL employees;

* **Physically removes** dead tuples.
* **Rewrites the table**, releasing disk space.
* Requires an **exclusive lock** on the table.

**🛠 Autovacuum to the Rescue**

Autovacuum will automatically vacuum the table **when enough dead tuples accumulate**, based on:

autovacuum\_vacuum\_threshold + autovacuum\_vacuum\_scale\_factor \* reltuples

You can tune it for high-write tables:

ALTER TABLE employees SET (

autovacuum\_vacuum\_threshold = 10,

autovacuum\_vacuum\_scale\_factor = 0.05

);

**Summary**

| **Action** | **Result** |
| --- | --- |
| UPDATE | Old row becomes a dead tuple; new row created |
| DELETE | Row becomes a dead tuple |
| VACUUM | Frees space inside the table file |
| VACUUM FULL | Shrinks table size on disk |
| n\_dead\_tup | Helps monitor dead rows accumulation |