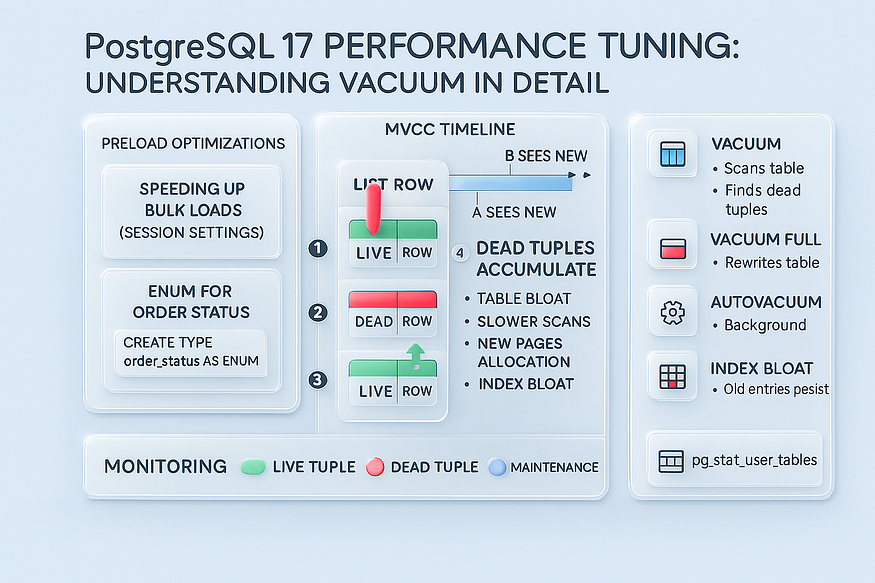
# **07 - PostgreSQL 17 Performance Tuning: Understanding VACUUM in Detail**



When it comes to PostgreSQL performance tuning, one of the most important maintenance tasks is ****VACUUM****. Without it, tables grow unnecessarily large, queries slow down, and indexes become inefficient. Many people hear about vacuuming but don’t fully understand why it is needed or how it works. Let’s explore this concept step by step, in an easy-to-understand format, with examples.

## **Bulk Loading 3 Million Orders in PostgreSQL**

When working with PostgreSQL, one of the most common tasks for testing and benchmarking is generating large datasets quickly. Let’s walk through how to create an ****orders table with 3 million rows****, complete with a status column and realistic values.

## **1. Speeding Up Bulk Loads (Session Settings)**

Before inserting millions of rows, we can temporarily adjust some PostgreSQL settings for faster performance:

SET synchronous\_commit = OFF;  
SET maintenance\_work\_mem = '1GB';  
SET temp\_buffers = '256MB';

Output:

postgres=# SET synchronous\_commit = OFF;  
SET maintenance\_work\_mem = '1GB';  
SET temp\_buffers = '256MB';  
SET  
SET  
SET  
postgres=#

* ****synchronous\_commit = OFF**** → avoids waiting for WAL flush after each commit.
* ****maintenance\_work\_mem**** → gives more memory for maintenance tasks like index creation.
* ****temp\_buffers**** → increases the memory used for temporary tables.

⚠️ These are ****session-only changes****. Once done, reset them back to safe defaults.

## **2. Creating an Enum for Order Status**

Instead of using free-text, we define an enum to keep status values clean and consistent:

DO $$  
BEGIN  
 IF NOT EXISTS (SELECT 1 FROM pg\_type WHERE typname = 'order\_status') THEN  
 CREATE TYPE order\_status AS ENUM ('PENDING','PROCESSING','SHIPPED','DELIVERED','CANCELLED');  
 END IF;  
END$$;

Output

postgres=# DO $$  
BEGIN  
 IF NOT EXISTS (SELECT 1 FROM pg\_type WHERE typname = 'order\_status') THEN  
 CREATE TYPE order\_status AS ENUM ('PENDING','PROCESSING','SHIPPED','DELIVERED','CANCELLED');  
 END IF;  
END$$;  
DO  
postgres=#

## **3. Creating the Orders Table**

We’ll create the table as ****UNLOGGED**** first. This skips WAL logging for faster inserts. After the bulk load, we’ll switch it back to LOGGED for durability.

DROP TABLE IF EXISTS orders CASCADE;

postgres=#  
postgres=# DROP TABLE IF EXISTS orders CASCADE;  
NOTICE: table "orders" does not exist, skipping  
DROP TABLE  
postgres=#

CREATE UNLOGGED TABLE orders (  
 order\_id BIGSERIAL PRIMARY KEY,  
 customer\_id BIGINT NOT NULL,  
 status order\_status NOT NULL,  
 amount\_usd NUMERIC(10,2) NOT NULL,  
 item\_count INT NOT NULL,  
 created\_at TIMESTAMPTZ NOT NULL,  
 updated\_at TIMESTAMPTZ NOT NULL  
);

postgres=# CREATE UNLOGGED TABLE orders (  
 order\_id BIGSERIAL PRIMARY KEY,  
 customer\_id BIGINT NOT NULL,  
 status order\_status NOT NULL,  
 amount\_usd NUMERIC(10,2) NOT NULL,  
 item\_count INT NOT NULL,  
 created\_at TIMESTAMPTZ NOT NULL,  
 updated\_at TIMESTAMPTZ NOT NULL  
);  
CREATE TABLE  
postgres=#

## **4. Disable Autovacuum (Optional)**

When bulk-loading millions of rows, PostgreSQL’s ****autovacuum**** process might kick in and slow things down. To avoid unnecessary background cleanup during the initial load, you can temporarily disable autovacuum on the table:

ALTER TABLE orders SET (autovacuum\_enabled = off);

postgres=# ALTER TABLE orders SET (autovacuum\_enabled = off);  
ALTER TABLE  
postgres=#

This ensures that PostgreSQL doesn’t try to vacuum the table while you’re still inserting massive amounts of data.

⚠️ ****Important:**** Don’t leave autovacuum off permanently. After your bulk load and indexing are done, it’s good practice to re-enable it:

ALTER TABLE orders SET (autovacuum\_enabled = on);

## **5. Bulk-Loading 3 Million Rows**

We generate rows using generate\_series and random(). Each row simulates:

* A ****customer\_id**** (random between 1 and ~1,000,000).
* A ****status**** distributed realistically:
* PENDING ≈ 12%
* PROCESSING ≈ 18%
* SHIPPED ≈ 30%
* DELIVERED ≈ 35%
* CANCELLED ≈ 5%
* An ****amount\_usd**** between $5 and $500.
* An ****item\_count**** between 1 and 5.
* ****created\_at**** spread over the past two years.
* ****updated\_at**** within two weeks of created\_at.

WITH src AS (  
 SELECT  
 (1 + (random()\*999999)::BIGINT) AS customer\_id,  
 random() AS r, -- used for status distribution  
 ROUND((5 + random()\*495)::NUMERIC, 2) AS amount\_usd,  
 GREATEST(1, (random()\*5)::INT) AS item\_count,  
 now() - ((random()\*730)::INT || ' days')::INTERVAL AS created\_at\_rand,  
 random() AS ro -- used for updated\_at offset  
 FROM generate\_series(1, 3000000) AS gs  
)  
INSERT INTO orders (customer\_id, status, amount\_usd, item\_count, created\_at, updated\_at)  
SELECT  
 customer\_id,  
 CASE  
 WHEN r < 0.12 THEN 'PENDING'::order\_status  
 WHEN r < 0.30 THEN 'PROCESSING'::order\_status  
 WHEN r < 0.60 THEN 'SHIPPED'::order\_status  
 WHEN r < 0.95 THEN 'DELIVERED'::order\_status  
 ELSE 'CANCELLED'::order\_status  
 END AS status,  
 amount\_usd,  
 item\_count,  
 created\_at\_rand,  
 created\_at\_rand + ((ro\*14)::INT || ' days')::INTERVAL  
FROM src;

postgres=# WITH src AS (  
 SELECT  
 (1 + (random()\*999999)::BIGINT) AS customer\_id,  
 random() AS r, -- used for status distribution  
 ROUND((5 + random()\*495)::NUMERIC, 2) AS amount\_usd,  
 GREATEST(1, (random()\*5)::INT) AS item\_count,  
 now() - ((random()\*730)::INT || ' days')::INTERVAL AS created\_at\_rand,  
 random() AS ro -- used for updated\_at offset  
 FROM generate\_series(1, 3000000) AS gs  
)  
INSERT INTO orders (customer\_id, status, amount\_usd, item\_count, created\_at, updated\_at)  
SELECT  
 customer\_id,  
 CASE  
 WHEN r < 0.12 THEN 'PENDING'::order\_status  
 WHEN r < 0.30 THEN 'PROCESSING'::order\_status  
 WHEN r < 0.60 THEN 'SHIPPED'::order\_status  
 WHEN r < 0.95 THEN 'DELIVERED'::order\_status  
 ELSE 'CANCELLED'::order\_status  
 END AS status,  
 amount\_usd,  
 item\_count,  
 created\_at\_rand,  
 created\_at\_rand + ((ro\*14)::INT || ' days')::INTERVAL  
FROM src;  
  
INSERT 0 3000000  
postgres=#

SELECT pg\_size\_pretty(pg\_relation\_size('orders'));

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('orders'));  
 pg\_size\_pretty  
----------------  
 219 MB  
(1 row)  
  
postgres=#

## **6. Switch Table Back to Logged**

After the bulk load, switch the table back to normal logging:

ALTER TABLE orders SET LOGGED;

postgres=# ALTER TABLE orders SET LOGGED;  
ALTER TABLE  
postgres=#

## **7. Create Indexes After the Load**

Always create indexes after bulk inserts — it’s much faster.

CREATE INDEX orders\_status\_idx ON orders (status);  
CREATE INDEX orders\_created\_at\_idx ON orders (created\_at);  
CREATE INDEX orders\_customer\_id\_idx ON orders (customer\_id);

postgres=# CREATE INDEX orders\_status\_idx ON orders (status);  
CREATE INDEX  
postgres=# CREATE INDEX orders\_created\_at\_idx ON orders (created\_at);  
CREATE INDEX  
postgres=# CREATE INDEX orders\_customer\_id\_idx ON orders (customer\_id);  
CREATE INDEX  
postgres=#

## **8. Analyze for the Query Planner**

Run ANALYZE so PostgreSQL collects fresh statistics:

ANALYZE orders;

postgres=# ANALYZE orders;  
ANALYZE  
postgres=#

## **9. Reset Session Settings**

Put things back to normal:

RESET synchronous\_commit;  
RESET maintenance\_work\_mem;  
RESET temp\_buffers;

postgres=# ANALYZE orders;  
ANALYZE  
postgres=# RESET synchronous\_commit;  
RESET  
postgres=# RESET maintenance\_work\_mem;  
RESET  
postgres=# RESET temp\_buffers;  
RESET  
postgres=#

## **Quick Sanity Checks**

SELECT COUNT(\*) AS total\_rows FROM orders;  
SELECT status, COUNT(\*) AS c FROM orders GROUP BY status ORDER BY c DESC;

postgres=# SELECT COUNT(\*) AS total\_rows FROM orders;  
 total\_rows  
------------  
 3000000  
(1 row)  
  
postgres=# SELECT status, COUNT(\*) AS c FROM orders GROUP BY status ORDER BY c DESC;  
 status | c  
------------+---------  
 DELIVERED | 1051338  
 SHIPPED | 899705  
 PROCESSING | 540512  
 PENDING | 358959  
 CANCELLED | 149486  
(5 rows)  
  
postgres=#

✅ And that’s it! You now have a fully populated orders table with ****3 million realistic records****, perfect for performance tuning, query optimization tests, or demo dashboards.

## **Step 1: What Happens When You Delete a Record?**

In PostgreSQL, deleting a row does ****not**** immediately remove it from disk.

* Instead, the row is simply ****marked as deleted****.
* It is no longer visible to queries, but the storage space is still occupied.
* Such rows are called ****dead tuples****.

👉 ****Example:****

DELETE FROM orders WHERE status = 'DELIVERED';

postgres=# DELETE FROM orders WHERE status = 'DELIVERED';  
DELETE 1050792  
postgres=#

SELECT pg\_size\_pretty(pg\_relation\_size('orders'));

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('orders'));  
 pg\_size\_pretty  
----------------  
 219 MB  
(1 row)  
  
postgres=#

* Suppose the orders table had 3,000,000 rows.
* After deleting 1,050,792, PostgreSQL still stores space for 3,000,000 rows.
* 1,949,208 rows are live; 1,050,792 are dead tuples.

📌 ****Analogy:**** Imagine writing in a notebook and crossing out a paragraph. The text is still there on the page, but you’ve decided to ignore it.

## **Step 2: Updates Work Like Delete + Insert**

In PostgreSQL, ****UPDATE = DELETE + INSERT**** under the hood.

1. The old row version is marked as deleted.
2. A new row version is inserted.

👉 ****Example:****

UPDATE orders  
SET customer\_id = '7967210'  
WHERE order\_id = 3000010;

postgres=# UPDATE orders  
SET customer\_id = '7967210'  
WHERE order\_id = 3000010;  
UPDATE 1  
postgres=#

* If you update the same customer 5 times, PostgreSQL keeps 5 versions of that row.
* Only the newest version is visible; the other 4 are dead tuples.

📌 ****Analogy:**** Think of sticky notes. Every time you change something, you paste a new sticky note on top. The old ones are still stuck under there, piling up.

## **Step 3: Why PostgreSQL Keeps Old Versions**

This happens because PostgreSQL uses ****MVCC (Multi-Version Concurrency Control)****.

* Multiple transactions may run at the same time.
* One transaction might need to see the old version while another uses the new version.

👉 Example scenario:

* Transaction A starts and queries a row.
* Transaction B updates that row.
* Transaction A still sees the old row, while Transaction B and later transactions see the updated row.

📌 ****Analogy:**** Imagine a shared Google Doc. You opened it at 2:00 PM. Even if your friend makes edits at 2:05 PM, you still see the older version you opened.

## **Step 4: Dead Tuples — The Hidden Problem**

Over time, dead tuples accumulate. This causes:

* ****Table bloat:**** tables get bigger on disk.
* ****Slower queries:**** scans take longer.
* ****Inefficient storage:**** PostgreSQL allocates new pages even though space is available.
* ****Index bloat:**** indexes also keep references to deleted rows.

👉 ****Check dead tuples:****

SELECT relname AS table\_name,  
 n\_live\_tup AS live\_rows,  
 n\_dead\_tup AS dead\_rows  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC;

postgres=# SELECT relname AS table\_name,  
 n\_live\_tup AS live\_rows,  
 n\_dead\_tup AS dead\_rows  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC;  
 table\_name | live\_rows | dead\_rows  
-----------------+-----------+-----------  
 orders | 1949208 | 1050793  
 project\_tasks\_1 | 1000000 | 0  
 customers | 5 | 0  
 audit\_log | 3 | 0  
 project\_tasks\_2 | 1000000 | 0  
 employee | 1 | 0  
(6 rows)  
  
postgres=#

📌 ****Analogy:**** Imagine a warehouse filled with empty boxes. New shipments arrive, but instead of reusing space, the warehouse keeps expanding — wasting resources.

## **Monitor Dead Tuples in Your Tables**

After loading millions of rows, it’s a good idea to ****check how many dead tuples**** (deleted/expired row versions) exist in your tables. PostgreSQL tracks this information in the pg\_stat\_user\_tables view.

You can run the following query:

SELECT relname, n\_dead\_tup, n\_live\_tup,  
 round(100.0\*n\_dead\_tup/GREATEST(n\_live\_tup,1),2) AS dead\_pct  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC  
LIMIT 20;

postgres=# SELECT relname, n\_dead\_tup, n\_live\_tup,  
 round(100.0\*n\_dead\_tup/GREATEST(n\_live\_tup,1),2) AS dead\_pct  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC  
LIMIT 20;  
 relname | n\_dead\_tup | n\_live\_tup | dead\_pct  
-----------------+------------+------------+----------  
 orders | 1050793 | 1949208 | 53.91  
 project\_tasks\_1 | 0 | 1000000 | 0.00  
 customers | 0 | 5 | 0.00  
 audit\_log | 0 | 3 | 0.00  
 project\_tasks\_2 | 0 | 1000000 | 0.00  
 employee | 0 | 1 | 0.00  
(6 rows)  
  
postgres=#

This will:

* Show the ****table name (relname)****.
* Display the number of ****dead tuples**** and ****live tuples****.
* Calculate the percentage of dead rows (dead\_pct).
* Order the results so the tables with the most dead tuples appear first.

This query helps you quickly identify which tables are suffering the most from bloat and may need ****VACUUM**** or ****VACUUM FULL****.

## **Step 5: How VACUUM Fixes It**

The ****VACUUM**** command solves this.

* It scans the table.
* Finds dead tuples.
* Marks their space as ****available for reuse****.

👉 ****Example:****

VACUUM orders;

postgres=# VACUUM orders;  
VACUUM  
postgres=#

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('orders'));  
 pg\_size\_pretty  
----------------  
 219 MB  
(1 row)  
  
postgres=#

* The 1,050,792 deleted rows in orders don’t vanish physically.
* But PostgreSQL now knows it can reuse that space for future inserts.

✅ Key Points:

* Normal VACUUM does not shrink the physical size of the table.
* It runs without locking the table — reads and writes can continue.

📌 ****Analogy:**** It’s like clearing shelves in a warehouse so new products can be stored in the same space.

## **Step 6: VACUUM FULL — The Heavy Cleaner**

Sometimes you don’t just want reusable space — you want to ****shrink the table file**** on disk. That’s when you use ****VACUUM FULL****.

* It rewrites the entire table.
* Physically removes dead tuples.
* Compacts the rows, reducing file size.

👉 ****Example:****

VACUUM FULL orders;

postgres=# VACUUM FULL orders;  
VACUUM  
postgres=#

SELECT pg\_size\_pretty(pg\_relation\_size('orders'));

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('orders'));  
 pg\_size\_pretty  
----------------  
 142 MB  
(1 row)  
  
postgres=#

* If your orders table grew to 219 MB but only had 142 MB of real data, VACUUM FULL could shrink it back to 142 MB.

📌 ****Analogy:**** Regular vacuuming is like sweeping the floor. VACUUM FULL is like moving the furniture, scrubbing, and reorganizing the room completely.

## **Step 7: Drawbacks of VACUUM FULL**

While powerful, VACUUM FULL has two major downsides:

1. ****Exclusive Lock****

* The table is locked for the entire duration.
* No queries can run during this time.

2. ****Resource Intensive****

* On large tables, it can take hours.
* Uses a lot of CPU and disk I/O.

👉 ****Bad scenario:**** Running VACUUM FULL on a 100-million-row production table during peak hours. Users will see downtime.

## **Step 8: Autovacuum — PostgreSQL’s Automatic Cleaner**

Manually vacuuming isn’t practical. PostgreSQL has ****autovacuum****, a background process that:

* Automatically vacuums tables when dead tuples exceed a threshold.
* Runs continuously to keep tables healthy.

👉 ****Configuration Example (postgresql.conf):****

autovacuum = on  
autovacuum\_naptime = 60  
autovacuum\_vacuum\_scale\_factor = 0.2  
autovacuum\_analyze\_scale\_factor = 0.1

📌 In PostgreSQL 17, autovacuum improvements include:

* Better prioritization of busy tables.
* Smarter scheduling.
* More detailed logging.

📌 ****Analogy:**** Autovacuum is like a robotic vacuum cleaner that quietly runs in the background, keeping your house from getting too messy.

## **Step 9: Index Bloat — Another Challenge**

Even after vacuuming, indexes can remain bloated.

* Updates and deletes leave behind old index entries.
* This slows down searches and joins.

👉 ****Solution — REINDEX:****

REINDEX TABLE orders;

📌 ****Analogy:**** If vacuuming is cleaning shelves, reindexing is like reorganizing the product catalog to make searches faster and more efficient.

## **Step 10: Best Practices in PostgreSQL 17**

1. Keep ****autovacuum enabled**** and tuned for your workload.
2. Run ****manual VACUUM**** on high-update tables when needed.
3. Avoid frequent VACUUM FULL — only use it for extreme bloat.
4. Regularly ****monitor dead tuples**** with pg\_stat\_user\_tables.
5. Handle ****index bloat separately**** with REINDEX.

👉 ****Quick check for dead tuples:****

SELECT relname, n\_dead\_tup, n\_live\_tup  
FROM pg\_stat\_user\_tables  
WHERE n\_dead\_tup > 100000;

## **Final Thoughts**

* ****Deletes and updates**** don’t immediately free space — they create dead tuples.
* ****VACUUM**** reclaims space for reuse, keeps performance stable, and is safe to run while queries execute.
* ****VACUUM FULL**** shrinks tables but locks them and is expensive.
* ****Autovacuum**** is your ongoing housekeeper.
* ****Indexes**** need separate care with REINDEX.
* In ****PostgreSQL 17****, vacuuming is smarter, faster, and easier to monitor, but still requires attention from DBAs.

✅ Think of vacuuming as ****housekeeping for your database****. Ignore it, and performance suffers. Manage it well, and your PostgreSQL 17 instance will run efficiently for years to come.