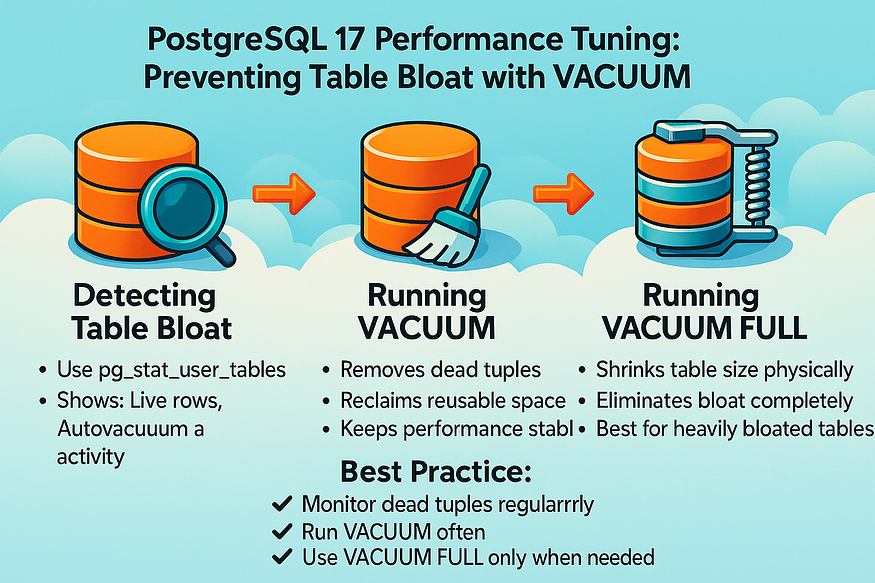
# **10 - PostgreSQL 17 Performance Tuning: Why Vacuum More Often Prevents Table Bloat**



One of the most common problems in PostgreSQL is ****table bloat**** — when dead tuples (old row versions) pile up inside a table and indexes grow unnecessarily large. Many administrators assume VACUUM should be avoided because it is resource-intensive, but the truth is the opposite:

👉 The real solution is to ****vacuum more often****.

Running VACUUM regularly means each cycle does less work and keeps tables from growing too large in the first place. Let’s see how to detect bloat and why frequent vacuuming matters.

## **Example: Detecting Table Bloat**

To investigate table health, PostgreSQL provides system views such as pg\_stat\_user\_tables. This view shows live rows, dead rows, and autovacuum activity.

Here’s a query that helps identify tables with high numbers of dead tuples:

SELECT   
 relname AS table\_name,  
 n\_live\_tup AS live\_rows,  
 n\_dead\_tup AS dead\_rows,  
 last\_autovacuum,  
 last\_vacuum  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC  
LIMIT 10;

postgres=# SELECT  
 relname AS table\_name,  
 n\_live\_tup AS live\_rows,  
 n\_dead\_tup AS dead\_rows,  
 last\_autovacuum,  
 last\_vacuum  
FROM pg\_stat\_user\_tables  
ORDER BY n\_dead\_tup DESC  
LIMIT 10;  
 table\_name | live\_rows | dead\_rows | last\_autovacuum | last\_vacuum  
------------+-----------+-----------+-------------------------------+-------------  
 orders | 1948839 | 1051161 | 2025-08-31 19:56:27.311642+00 |  
(1 row)  
  
postgres=#

👉 In this example:

* The ****orders**** table has ****1,051, 161 dead rows****, which is a sign of bloat.
* Autovacuum last ran on orders recently, but not frequently enough to prevent dead tuple buildup.

## **Example: Running VACUUM**

If we notice a table with excessive dead tuples, we can run a manual VACUUM:

VACUUM orders;

postgres=# VACUUM orders;  
VACUUM  
postgres=#

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

This will clean up space for reuse but will ****not shrink the file size on disk****. Instead, it records free space in the Free Space Map (FSM), so new inserts or updates reuse existing pages.

## **Example: Running VACUUM FULL**

If the table is badly bloated and we need to physically shrink it, we can use:

VACUUM FULL orders;

postgres=# VACUUM FULL orders;  
VACUUM  
postgres=#

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

* This rewrites the entire table, removing all dead space.
* The size on disk will shrink.
* But it requires an ****exclusive lock****, which means no other queries can write to the table during the process.

## **Why Autovacuum Matters**

To avoid disruptive operations like VACUUM FULL, it’s best to let ****autovacuum**** run frequently:

* It keeps dead tuples under control.
* It reduces the risk of severe table bloat.
* It prevents performance drops caused by overly large tables and indexes.

For example, if we notice that orders keeps accumulating dead tuples, we can adjust autovacuum for that table:

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=#  
postgres=#

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

ALTER TABLE orders  
SET (autovacuum\_vacuum\_scale\_factor = 0.05, -- trigger cleanup at 5% dead rows  
 autovacuum\_analyze\_scale\_factor = 0.02); -- update stats more often

postgres=# ALTER TABLE orders  
SET (autovacuum\_vacuum\_scale\_factor = 0.05, -- trigger cleanup at 5% dead rows  
 autovacuum\_analyze\_scale\_factor = 0.02); -- update stats more often  
ALTER TABLE  
postgres=#

This makes autovacuum run more aggressively for that table, keeping it lean.

DELETE FROM orders WHERE status = 'DELIVERED';

postgres=# DELETE FROM orders WHERE status = 'DELIVERED';  
DELETE 1051039  
postgres=#  
postgres=#

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

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

✅ ****Key takeaway:****  
Avoiding VACUUM creates larger problems. The right approach is to ****vacuum more often**** — either manually or by tuning ****auto vacuum**** — so tables stay small, indexes stay efficient, and PostgreSQL performance remains stable.