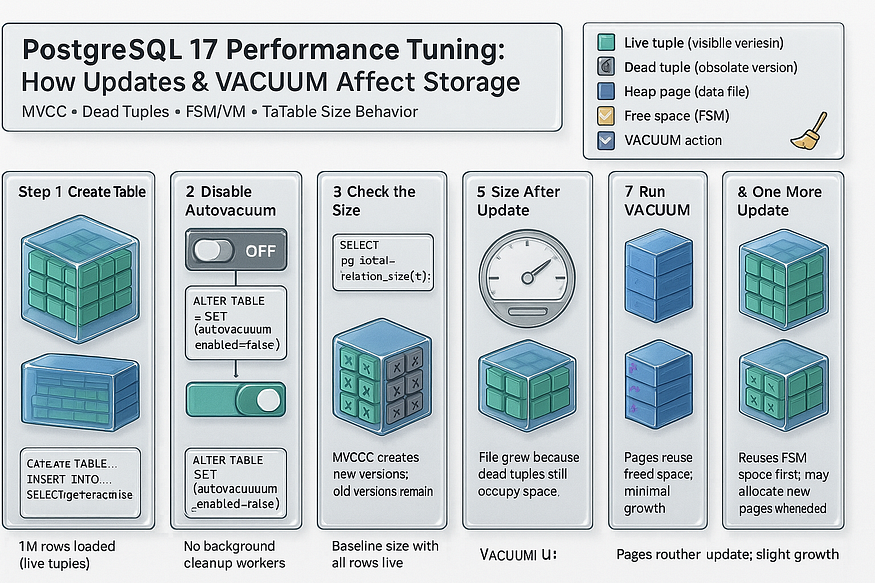
# **09 - PostgreSQL 17 Performance Tuning: How Updates and VACUUM Affect Storage**



PostgreSQL’s storage model is different from many other databases. Instead of overwriting rows in place, PostgreSQL creates new row versions whenever data is updated. This design is great for transaction safety and concurrency, but it also impacts table size and makes ****VACUUM**** essential for performance.

Let’s go through a test example step by step to see how this works in practice.

## **Turning Off Autovacuum (For Testing Only)**

PostgreSQL normally runs a background process called ****autovacuum**** to clean up dead tuples and keep tables from bloating.

* For real applications, turning autovacuum off is ****not recommended****, because it is critical for database health.
* But for ****testing purposes****, it can be turned off for a specific table to make sure nothing happens automatically in the background.

This allows us to observe exactly how PostgreSQL manages storage.

## **Step 1: Create the Table**

First, create a table with one million rows using the generate\_series function.

psql -c "CREATE TABLE project\_tasks\_1 AS SELECT generate\_series AS task\_id FROM generate\_series(1,1000000);

[postgres@ip-172-31-25-61 ~]$ psql -c "CREATE TABLE project\_tasks\_1 AS SELECT generate\_series AS task\_id FROM generate\_series(1,1000000)";  
SELECT 1000000  
[postgres@ip-172-31-25-61 ~]$

This creates a table named ****project\_tasks\_1**** with values from 1 to 1,000,000 in the column task\_id.

## **Step 2: Disable Autovacuum**

Although we added the storage parameter while creating the table, we can explicitly disable autovacuum using the following command:

ALTER TABLE project\_tasks\_1 SET (autovacuum\_enabled = off);

postgres=# ALTER TABLE project\_tasks\_1 SET (autovacuum\_enabled = off);  
ALTER TABLE  
postgres=#

Now, autovacuum will not run on this table, and we can test updates and VACUUM behavior manually.

## **Step 3: Check the Size of the Table**

Before making any changes, let’s see how large the table is on disk:

SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));  
 pg\_size\_pretty  
----------------  
 35 MB  
(1 row)  
  
postgres=#

* pg\_relation\_size → gives the table size in bytes.
* pg\_size\_pretty → makes it easier to read (e.g., MB, GB).

At this stage, the size represents only the original one million rows.

## **Step 4: Update All Rows**

Now update every row in the table:

UPDATE project\_tasks\_1 SET task\_id = task\_id + 1;

postgres=# UPDATE project\_tasks\_1 SET task\_id = task\_id + 1;  
UPDATE 1000000  
postgres=#

What happens behind the scenes is crucial:

* PostgreSQL does ****not overwrite rows**** in place.
* Instead, it creates ****new versions of all rows**** and marks the old ones as ****dead tuples****.
* Why does it copy rows?

1. Because the transaction may fail, and PostgreSQL needs the old version for rollback.
2. Because other concurrent transactions may still need to read the old version.

## **Step 5: Table Size After Update**

Check the table size again:

SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));  
 pg\_size\_pretty  
----------------  
 70 MB  
(1 row)  
  
postgres=#

👉 You will notice that the table has ****doubled in size**** because it now contains both old and new row versions.

## **Step 6: Run VACUUM**

At this point, many people expect that running VACUUM will shrink the table size:

VACUUM project\_tasks\_1;  
SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));

postgres=# VACUUM project\_tasks\_1;  
VACUUM  
postgres=#

postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));  
 pg\_size\_pretty  
----------------  
 70 MB  
(1 row)  
  
postgres=#

But here’s the reality:

* ****VACUUM does not return space to the operating system.****
* It simply marks the dead rows as ****reusable free space inside the table****.
* The table size remains the same on disk.

## **Step 7: Run Another Update**

Now let’s update all rows again:

UPDATE project\_tasks\_1 SET task\_id = task\_id + 1;  
SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));

postgres=# UPDATE project\_tasks\_1 SET task\_id = task\_id + 1;  
UPDATE 1000000  
postgres=#  
postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));  
 pg\_size\_pretty  
----------------  
 70 MB  
(1 row)  
  
postgres=#

👉 This time, the table may ****not grow further**** because PostgreSQL reuses the free space tracked by the ****Free Space Map (FSM)****.

## **Step 8: Run One More Update**

If we update all rows yet again:

UPDATE project\_tasks\_1 SET task\_id = task\_id + 1;  
SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));

postgres=# UPDATE project\_tasks\_1 SET task\_id = task\_id + 1;  
UPDATE 1000000  
postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));  
 pg\_size\_pretty  
----------------  
 104 MB  
(1 row)  
  
postgres=#

👉 At this point, if all reusable space is exhausted, the table will grow again, because PostgreSQL must allocate new disk space.

## **Key Takeaway**

* PostgreSQL updates always create ****new row versions****.
* VACUUM does ****not shrink table files****; it only allows space to be reused.
* Free space is tracked in the ****FSM****, and PostgreSQL uses it before allocating new disk pages.
* If no free space remains, further updates will cause table growth.

✅ Understanding this storage behavior is critical for PostgreSQL performance tuning. There is no performance tuning without first understanding how storage works.

## **Shrinking a Table with**VACUUM FULL**(locks the table)**

VACUUM FULL project\_tasks\_1;

SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));

-- 1) Compact the table and return space to the OS  
postgres=# VACUUM FULL project\_tasks\_1;  
VACUUM

-- 2) Verify the table's heap size after compaction  
postgres=# SELECT pg\_size\_pretty(pg\_relation\_size('project\_tasks\_1'));  
 pg\_size\_pretty  
----------------  
 35 MB  
(1 row)

****Notes & best practices****

* VACUUM FULL rewrites the table; plan for downtime on busy systems.
* It needs free disk space roughly equal to the table size (plus indexes) during the rewrite.
* To see ****table + indexes**** together, check total size:

SELECT pg\_size\_pretty(pg\_total\_relation\_size('project\_tasks\_1'));

* Follow up with ANALYZE to refresh statistics:

ANALYZE project\_tasks\_1;

* If you need to reduce locking, consider alternatives like CLUSTER (also exclusive lock) or the pg\_repack extension (online, minimal blocking).