MAINTENANCE TASKS

VACUUM

-- The VACUUM command is an important maintenance tool for stable and optimal database performance.
-- It improves query performance by recovering space occupied by "dead tuples" or dead rows in a table.
-- Whenever there is an update or delete, Postgres doesn't remove those tuples but it only marks them as deleted.
-- This leads to table bloat.
-- It has impact on query performance ,as Postgres has to scan all the tuples ,live and dead.

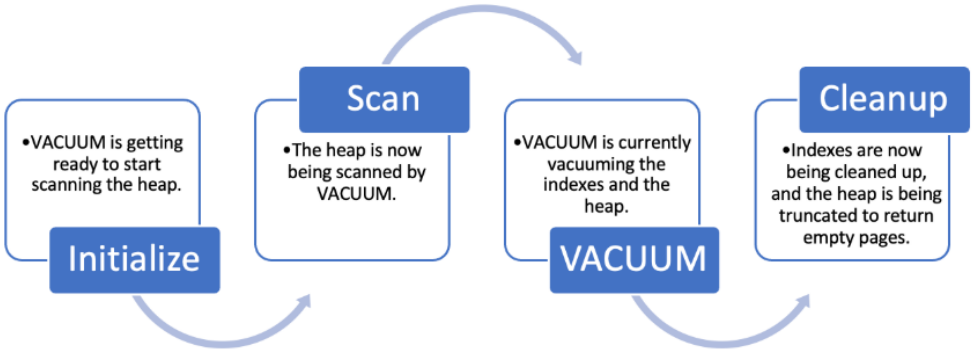
- **TYPES**
  1) VACUUM
  2) VACUUM FULL
  3) VACUUM ANALYZE
  4) AUTOVACUUM
  5) PARALLEL VACUUM
  6) VACUUM FREEZE

1) VACUUM

-- Cleans up dead tuples and recovers space.
-- Improves query speed and decreases disk I/O by reducing table bloat.
-- Frequently and consistently to minimize table bloat.
-- Dosn't lock the table
-- Can manually run the command through *Vacuum table_name;*
-- VM gets updated, which speeds up index only scans.

5 Phases of VACUUM :-

1) Heap scan -- scans the table, identifies the dead tuples
2) Index vacuum -- Vacuum the indexes one by one
3) Heap vacuum -- Vacuum the table
4) Index cleanup -- reclaims and marks empty index pages as reusable.
5) Heap truncation -- truncate empty pages at the end of the table



There is an exception in the VACUUM, as we usually know Vacuum helps to reclaim the space by removing the dead tuples and makes it reusable,but in the below scenario, when the records are deleted at the bottom of the table and Vacuum is run on it, it gives space back to the OS.

This is because of the **Heap Truncation Phase (final phase of VACUUM)**

-- If PostgreSQL sees completely empty pages at the end of the heap,
-- And no other session is using them
-- Then it will truncate them and physically return that space to the OS.

Example

1	Create test table	CREATE TABLE test_table (id SERIAL PRIMARY KEY,data TEXT);
2	Insert around 4 lakh records	INSERT INTO test_table (data) SELECT 'Row number ' || generate_series(1, 400000);
3	delete random records	DELETE FROM test_table WHERE id IN (SELECT id FROM test_table ORDER BY random() LIMIT 10000);
4	delete records from end of the table.	DELETE FROM test_table WHERE id > 300000;

1) Create test table

```
test=# CREATE TABLE test_table (
test(#     id SERIAL PRIMARY KEY,
test(#     data TEXT
test(# );
CREATE TABLE
```

2) Insert around 4 lakh records

```
test=# INSERT INTO test_table (data)
test-# SELECT 'Row number ' || generate_series(1, 400000);


   INSERT 0 400000
```

3) delete random records

```
test=# DELETE FROM test_table WHERE id IN (
    SELECT id FROM test_table
    ORDER BY random()
    LIMIT 100000
);
DELETE 100000
test=# vacuum test_table;
VACUUM
```

-- When we delete the random records and run Vacuum manually, file size is not affected.

-- Below is the snip of the refilnode, after vacuum and before vacuum.
-- It is observed that there is no space released to OS.

```
[postgres@localhost 17150]$ ls -ltrh 17152*
-rw-------. 1 postgres postgres  24K Apr 16 02:14 17152_fsm
-rw-------. 1 postgres postgres 8.0K Apr 16 02:14 17152_vm
-rw-------. 1 postgres postgres  20M Apr 16 02:22 17152
[postgres@localhost 17150]$
[postgres@localhost 17150]$
[postgres@localhost 17150]$ ls -ltrh 17152*
-rw-------. 1 postgres postgres  24K Apr 16 02:14 17152_fsm
-rw-------. 1 postgres postgres 8.0K Apr 16 02:14 17152_vm
-rw-------. 1 postgres postgres  20M Apr 16 02:23 17152
```

4) delete records from end of the table.

```
test=# DELETE FROM test_table
test-# WHERE id > 300000;
DELETE 72458
test=#
test=#
test=# vacuum test_table;
VACUUM
```

-- When there is deletion of the records at the bottom of the table, and when we run Vacuum manually, space is released to the OS.
-- Below is the snip where space is released to the OS after vacuum.
-- The file size is changed from 20MB to 15MB.

```
[postgres@localhost 17150]$ ls -ltrh 17152*
-rw-------. 1 postgres postgres  24K Apr 16 02:14 17152_fsm
-rw-------. 1 postgres postgres 8.0K Apr 16 02:14 17152_vm
-rw-------. 1 postgres postgres  20M Apr 16 02:23 17152
[postgres@localhost 17150]$ ls -ltrh 17152*
-rw-------. 1 postgres postgres  24K Apr 16 02:14 17152_fsm
-rw-------. 1 postgres postgres 8.0K Apr 16 02:26 17152_vm
-rw-------. 1 postgres postgres  15M Apr 16 02:26 17152
```

2) VACUUM FULL

-- When we run Vacuum Full , table is exclusively locked.
-- other transactions cannot read from or write to the table, potentially impacting database availability.
-- takes longer time.
-- ensures that all dead tuples are completely removed.
-- helps in reclaiming the space and improving the query performance.
-- it rewrites the table

3) VACUUM ANALYZE

-- both vacuuming and analyzing are done together
-- convenient as running them together frees up disk space while providing updated statistics.
-- Longer duration, as both operations are performed simultaneously.

-- Potential locking as both operations do require some level of lock (not exclusive locks), which can impact database operations.
-- Updates statistics on a table to help efficient query execution,accurate statistics will help planner to choose the appropriate query plan.

Best Time to use VACUUM ANALYZE is after bulk operations or at initial database setup, to ensure everything is optimized.

4) PARALLEL VACUUM (PG13+)

-- Runs index vacuuming and index cleanup cycles of VACUUM in parallel using integer background worker.
-- allows to use multiple CPUs
-- When we run VACUUM with the PARALLEL option, parallel workers are used for the index VACUUM and index cleanup phase.
-- The number of parallel workers (degree of parallelism) is decided based on the number of indexes in the table or as specified by the user.
-- Heap (table data) cleanup is done by the leader process,
-- If you're running parallel VACUUM without an integer argument, it calculates the degree of parallelism based on the number of indexes in the table.

--Logic that Postgres determines :-

A) degree_of_parallelism =
min(number_of_indexes,max_parallel_maintenance_workers)

B) **max_possible_workers** =
min(number_of_indexes - 1, max_parallel_maintenance_workers)

The following are important parameters for parallel vacuuming in for PostgreSQL;

1) **max_worker_processes**
-- Sets the maximum number of concurrent worker processes.
--It includes parallel query workers, autovacuum workers, logical replication workers

2) **min_parallel_index_scan_size**
--The minimum amount of index data that must be scanned in order for a parallel scan to be considered.

3) **max_parallel_maintenance_workers**
-- The maximum number of parallel workers that can be started by a single utility command.
--More workers = faster vacuum on large tables.

For parallel VACUUM to work:
-- the table must have more than one index
-- and the index size must be greater than min_parallel_index_scan_size.

4) Autovacuum

-- PostgreSQL's autovacuum system involves two types of processes
1. autovacuum launcher
2. autovacuum workers

-- The autovacuum launcher's job is to start autovacuum workers
-- job of the workers is to perform whatever VACUUM and ANALYZE operations are needed in order to maintain good database performance.
-- Prevents table bloat by removing dead tuples..
-- Runs automatically in the background.
-- We can also set Autovacuum on Individual tables too by using ALTER Table command.

```
ALTER TABLE table_name SET (
  autovacuum_vacuum_scale_factor = 0.05,
  autovacuum_vacuum_threshold = 20,
  autovacuum_freeze_max_age = 100000000
);
```

--Vacuuming is typically triggered for a table if

1) **Dead tuples** > autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * number of tuples

OR

2) **the number of inserted tuples** > autovacuum_vacuum_insert_threshold + autovacuum_vacuum_insert_scale_factor * number of tuples.

-- Autovacuum uses below parameters together to decide when to start cleaning and how quickly to do it, balancing performance and system load.
- -- autovacuum_vacuum_scale_factor
- -- autovacuum_vacuum_threshold
- -- autovacuum_vacuum_cost_limit
- -- autovacuum_naptime.

-- To Analyze it uses below parameters together to decide
- -- autovacuum_analyze_scale_factor
- -- autovacuum_analyze_threshold
- -- autovacuum_analyze_cost_limit
- -- autovacuum_naptime

Important Parameters:-

A. autovacuum = on

-- Controls whether the server should run the autovacuum launcher daemon.
-- This is on by default
-- track_counts must also be enabled for autovacuum to work.
(track_counts informs postgres to track table activity)
--autovacuuming can be disabled for individual tables by changing table storage parameters.

Query :-

-- ALTER TABLE employees SET (autovacuum_enabled = false);

```
sampledb_1=# select relname,n_tup_upd,n_tup_del ,n_dead_tup, last_vacuum, last_autovacuum from pg_stat_user_tables;
  relname  | n_tup_upd | n_tup_del | n_dead_tup | last_vacuum |        last_autovacuum
-----------+-----------+-----------+------------+-------------+-------------------------------
 employees |    234002 |     52000 |     104001 |             | 2025-04-21 03:19:42.886396-07
(1 row)

sampledb_1=# select relname,n_tup_upd,n_tup_del ,n_dead_tup, last_vacuum, last_autovacuum from pg_stat_user_tables;
  relname  | n_tup_upd | n_tup_del | n_dead_tup | last_vacuum |        last_autovacuum
-----------+-----------+-----------+------------+-------------+-------------------------------
 employees |    234002 |     52000 |          0 |             | 2025-04-21 03:23:42.871667-07
(1 row)
```

B. autovacuum_vacuum_threshold = 50

-- parameter controls the minimum number of tuple update or delete operations that must occur on a table before autovacuum vacuums it.
-- useful for the tables to prevent unnecessary vacuumin on tables, which dont have high rate of these operations.
-- default value is set to 50, it means if tables has less than 50 updates or deletes, autovacuum wont vacuum it.
-- it works in combination with autovacuum_vacuum_scale_factor

Recommendations :
-- Set this higher to prevent constant triggering of autovacuum on small catalog tables.
-- setting this value too high can grow dead tuples, which will eventually degrade performance.

C. autovacuum_vacuum_scale_factor = 0.2

-- The default is 0.2 (20% of table size).
--Vacuum scale factor controls how often autovacuum runs on a table.
--depends on the value of autovacuum_vacuum_threshold as a controlling factor.
--lower values like 0.005 or 0.035 can significantly improve autovacuum performance on large tables, preventing excessive accumulation of dead tuples.,

Recommendations :-

Lower value= autovacuum runs more frequently, which is good for busy tables with lots of updates/deletes.
--Frequent autovacuum keeps tables small and performance high by removing dead rows quickly.

Higher Value= autovacuum runs less frequently, which saves CPU and I/O on low-activity tables.

--*Tune it per table based on how often that table is written to*
high writes = lower value, low writes = higher value.

Example :-
I have set the parameters as

--autovacuum_vacuum_threshold = 50
--autovacuum_vacuum_scale_factor = 0.2
--number of tuples = 208003

```
sampledb_1=# show autovacuum_vacuum_threshold ;
 autovacuum_vacuum_threshold
-----------------------------
 50
(1 row)

sampledb_1=# show autovacuum_vacuum_scale_factor ;
 autovacuum_vacuum_scale_factor
--------------------------------
 0.2
(1 row)

sampledb_1=# select count(*) from employees;
 count
--------
 208003
(1 row)
```

Calculations :-

-- autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * number of tuples
- 50 + (208003 * 0.2)= 41650.6

Based on parameters, Autovacuum will run when there are more than 41,650.

```
sampledb_1=# SELECT relname, last_autovacuum, n_dead_tup
FROM pg_stat_user_tables
WHERE relname = 'employees';
  relname  |        last_autovacuum        | n_dead_tup
-----------+-------------------------------+------------
 employees | 2025-04-21 03:57:08.472922-07 |      42000
(1 row)

sampledb_1=# SELECT relname, last_autovacuum, n_dead_tup
FROM pg_stat_user_tables
WHERE relname = 'employees';
  relname  |        last_autovacuum        | n_dead_tup
-----------+-------------------------------+------------
 employees | 2025-04-21 03:58:07.748836-07 |          0
(1 row)
```

Autovacuum has run on table.

```
2025-04-21 03:58:07.749 PDT [6102] LOG:  automatic vacuum of table "sampledb_1.public.employees": index scans: 1
        pages: 0 removed, 3389 remain, 0 skipped due to pins, 182 skipped frozen
        tuples: 42000 removed, 235660 remain, 0 are dead but not yet removable, oldest xmin: 841
        index scan needed: 3121 pages from table (92.09% of total) had 42000 dead item identifiers removed
        index "employees_pkey": pages: 1702 in total, 0 newly deleted, 0 currently deleted, 0 reusable
        avg read rate: 0.000 MB/s, avg write rate: 5.619 MB/s
        buffer usage: 11315 hits, 0 misses, 198 dirtied
        WAL usage: 11059 records, 0 full page images, 844163 bytes
        system usage: CPU: user: 0.09 s, system: 0.00 s, elapsed: 0.27 s
2025-04-21 03:58:07.959 PDT [6102] LOG:  automatic analyze of table "sampledb_1.public.employees"
        avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
        buffer usage: 3578 hits, 0 misses, 0 dirtied
        system usage: CPU: user: 0.15 s, system: 0.01 s, elapsed: 0.20 s
[postgres@localhost log]$
```

D. autovacuum_analyze_threshold = 50

-- minimum no. of updates, or deltes that must ocuur before autovacuum analyzes it.
-- If the number of tuple inserts, updates, or deletes exceeds this threshold, autovacuum analyzes the table.
-- It is important for query planner to get the updated statistics to access the latest data
-- autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor * number_of_rows)

It triggers when it exceeds the below value

autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor * number_of_rows)

E. autovacuum_analyze_scale_factor = 0.1

-- The autovacuum process uses this parameter to calculate a threshold based on the number of tuples in a table.
-- If the number of tuple inserts, updates, or deletes exceeds this threshold, autovacuum analyzes the table.
-- it works similar to the vaccum

Example :-

autovacuum_analyze_threshold = 50
autovacuum_analyze_scale_factor = 0.1
number_of_rows = 178003

By calculations :-

50 + (178003*0.1) = **17,850.3**

So, Analyze will run after the dead tuples will reach 17,850.

F. autovacuum_vacuum_cost_limit = -1

-- This parameter controls the amount of CPU and I/O resources that an autovacuum worker can consume.
-- If the cost exceeds this limit, autovacuum will wait and resume after a delay.
-- It sets a work limit, like 2,000 units, after which the vacuum pauses to avoid overloading the system.

Lowering the limit helps prevent autovacuum from impacting the performance of other queries.

Example:

-- If set to a positive number (e.g., 1000), autovacuum will stop if it uses more than that amount of resources during the operation.
-- If set to -1, autovacuum will not have its own limit and will inherit the default system limit.
-- parameter affects only autovacuum processes for vacuum , not for analyze.

Recommendations :

-- If you set the value of autovacuum_vacuum_cost_limit too high, the autovacuum process might consume too many resources and slow down other queries.
-- If you set it too low, the autovacuum process might not reclaim enough space, which causes the table to become larger over time.

G. autovacuum_vacuum_cost_delay = 2ms

-- If this value is specified without units, it is taken as milliseconds
-- The default value is 2 milliseconds.
-- Autovacuum will sleep for the time when it reached the cost limit.
-- helps to reduce the I/O process
-- This value should be balanced based on your disk speed, workload, and how aggressive you want autovacuum to be.

H. autovacuum_max_workers = 3

-- Increasing the autovacuum_max_workers setting can increase the load on the server, which can impactperformance if you don't have enough resources.
-- The optimal setting depends on the specific requirements of your database, its size, and the number of tables it contains.
-- Increasing this workers would increase the load on the server.

Recommendations:

This no should be decided by considering the database, its size and the number of table it contains.

I. autovacuum_naptime = 1min

-- The default setting of 1 minute means that the autovacuum launcher should wait to launch an autovacuum worker for each database once per minute.

Example

-- To check each database less frequently, you will tend to increase the amount of time that passes between the point at which a table needs a VACUUM or ANALYZE operation to maintain good performance.

Impact:-
-- In the case of VACUUM, this will typically lead to table bloat and more space will have to be allocated in order to store the new row versions created by UPDATE or INSERT operations performed on that table.
-- In the case of ANALYZE, this will typically lead to poor query performance.
-- If the data in your table changes enough that the statistics are needed to be updated, and by increasing autovacuum naptime we delay that statistics update, then you will be running with inaccurate statistics for a longer period of time, perhaps resulting in poor query plans.

J. autovacuum_freeze_max_age = 200000000

-- It sets the maximum number of transactions a table can go through before it's forced to be vacuumed.
-- Default is 200,000,000 transactions — this helps prevent transaction ID wraparound.
-- If the table isn’t vacuumed before this limit, PostgreSQL automatically runs a vacuum to "freeze" old data and keep things safe.

TABLE

Parameter	Default Value	Description
autovacuum	on	enables or disables the autovacuum background process
autovacuum_vacuum_threshold	50	minimum number of dead rows that must be present in a table before it is vacuumed.
autovacuum_analyze_threshold	50	minimum number of live rows that must be present in a table before it is analyzed.
autovacuum_vacuum_scale_factor	0.2	setting is a multiplier that determines how many dead rows are needed to trigger a vacuum based on the table size.
autovacuum_analyze_scale_factor	0.1	setting is a multiplier that determines how many live rows are needed to trigger an analyze based on the size of the table.
autovacuum_vacuum_cost_delay	2ms	determines the time (in milliseconds) the autovacuum will wait before starting a vacuum operation.
autovacuum_vacuum_cost_limit	-1	determines the maximum number of rows that can be vacuumed in a single vacuum operation.
autovacuum_max_workers	3	Specifies the maximum number of autovacuum processes (other than the autovacuum launcher) that may be running at any one time.
autovacuum_naptime	1min	Specifies the minimum delay between autovacuum runs on any given database.
autovacuum_vacuum_scale_factor	0.2	Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a VACUUM.
autovacuum_freeze_max_age	200000000	It sets the maximum number of transactions a table can go through before it's forced to be vacuumed.

Monitoring Queries

1) Last Vacuum for all tables

```
SELECT relname,
    last_vacuum,
    last_autovacuum,
    last_analyze,
    last_autoanalyze
FROM pg_stat_user_tables
ORDER BY last_autovacuum DESC NULLS LAST;
```

```
test=# SELECT relname,
      last_vacuum,
      last_autovacuum,
      last_analyze,
      last_autoanalyze
FROM pg_stat_user_tables
ORDER BY last_autovacuum DESC NULLS LAST;
     relname     |          last_vacuum          |        last_autovacuum        |         last_analyze          |       last_autoanalyze
-----------------+-------------------------------+-------------------------------+-------------------------------+-------------------------------
 autovac_test    | 2025-04-16 21:21:37.254822-07 | 2025-04-16 09:12:01.508698-07 |                               | 2025-04-16 09:12:02.981311-07
 test_table      | 2025-04-16 02:31:30.062587-07 | 2025-04-16 02:31:27.949438-07 | 2025-04-16 02:09:48.375646-07 | 2025-04-16 02:31:28.319581-07
 projects        |                               |                               |                               |
```

2) Last vacuum for particular table

```
SELECT relname,last_vacuum,last_autovacuum,last_analyze,last_autoanalyze
FROM pg_stat_user_tables
WHERE relname = 'autovac_test';
```

```
test=# SELECT relname,last_vacuum,last_autovacuum,last_analyze,last_autoanalyze
FROM pg_stat_user_tables
WHERE relname = 'autovac_test';
   relname    |          last_vacuum          |        last_autovacuum        |         last_analyze          |       last_autoanalyze
--------------+-------------------------------+-------------------------------+-------------------------------+-------------------------------
 autovac_test | 2025-04-16 21:21:37.254822-07 | 2025-04-16 09:12:01.508698-07 | 2025-04-16 21:24:38.875203-07 | 2025-04-16 09:12:02.981311-07
(1 row)
```

3)To check tables with high dead tuples

```
SELECT relname,
    n_dead_tup,
    n_live_tup,
    pg_size_pretty(pg_total_relation_size(relid)) AS table_size
FROM pg_stat_user_tables
WHERE n_dead_tup > 1000
ORDER BY n_dead_tup DESC;
```

```
test=# SELECT relname,
test-#        n_dead_tup,
test-#        n_live_tup,
test-#        pg_size_pretty(pg_total_relation_size(relid)) AS table_size
test-# FROM pg_stat_user_tables
test-# WHERE n_dead_tup > 1000
test-# ORDER BY n_dead_tup DESC;
     relname      | n_dead_tup | n_live_tup | table_size
------------------+------------+------------+------------
 employees        |     200000 |     100000 | 29 MB
 employee_project |     150000 |          0 | 9824 kB
(2 rows)
```

4) Bloat query

```
SELECT * FROM (
 SELECT
  current_database(), schemaname, tablename, /*reltuples::bigint, relpages::bigint, otta,*/
  ROUND((CASE WHEN otta=0 THEN 0.0 ELSE sml.relpages::float/otta END)::numeric,1) AS tbloat,
  CASE WHEN relpages < otta THEN 0 ELSE bs*(sml.relpages-otta)::BIGINT END AS wastedbytes,
  iname, /*ituples::bigint, ipages::bigint, iotta,*/
  ROUND((CASE WHEN iotta=0 OR ipages=0 THEN 0.0 ELSE ipages::float/iotta END)::numeric,1) AS ibloat,
  CASE WHEN ipages < iotta THEN 0 ELSE bs*(ipages-iotta) END AS wastedibytes
FROM (
  SELECT
    schemaname, tablename, cc.reltuples, cc.relpages, bs,
    CEIL((cc.reltuples*((datahdr+ma-
      (CASE WHEN datahdr%ma=0 THEN ma ELSE datahdr%ma END))+nullhdr2+4))/(bs-20::float)) AS otta,
    COALESCE(c2.relname,'?') AS iname, COALESCE(c2.reltuples,0) AS ituples, COALESCE(c2.relpages,0) AS ipages,
    COALESCE(CEIL((c2.reltuples*(datahdr-12))/(bs-20::float)),0) AS iotta -- very rough approximation, assumes all cols
  FROM (
    SELECT
      ma,bs,schemaname,tablename,
      (datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%ma END)))::numeric AS datahdr,
      (maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma ELSE nullhdr%ma END))) AS nullhdr2
    FROM (
      SELECT
        schemaname, tablename, hdr, ma, bs,
        SUM((1-null_frac)*avg_width) AS datawidth,
        MAX(null_frac) AS maxfracsum,
        hdr+(
          SELECT 1+count(*)/8
          FROM pg_stats s2
          WHERE null_frac<>0 AND s2.schemaname = s.schemaname AND s2.tablename = s.tablename
        ) AS nullhdr
      FROM pg_stats s, (
        SELECT
          (SELECT current_setting('block_size')::numeric) AS bs,
          CASE WHEN substring(v,12,3) IN ('8.0','8.1','8.2') THEN 27 ELSE 23 END AS hdr,
          CASE WHEN v ~ 'mingw32' THEN 8 ELSE 4 END AS ma
        FROM (SELECT version() AS v) AS foo
      ) AS constants
      GROUP BY 1,2,3,4,5
    ) AS foo
  ) AS rs
  JOIN pg_class cc ON cc.relname = rs.tablename
  JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.nspname = rs.schemaname AND nn.nspname <> 'information_schema'
  LEFT JOIN pg_index i ON indrelid = cc.oid
  LEFT JOIN pg_class c2 ON c2.oid = i.indexrelid
) AS sml
ORDER BY wastedbytes DESC
) AS bloat_data
WHERE tablename = 'test_bloat';
```

```
 current_database | schemaname | tablename  | tbloat | wastedbytes |      iname      | ibloat | wastedibytes
------------------+------------+------------+--------+-------------+-----------------+--------+--------------
 bank             | public     | test_bloat |    2.3 |    97648640 | test_bloat_pkey |    0.0 |            0
(1 row)
```

Observation :-

Column	What it Means
current_database	The name of the database (bank)
schemaname	The schema the table belongs to (public)
tablename	The name of the table analyzed (test_bloat)
tbloat	**Table Bloat Ratio** = 2.3 → The table is **2.3 times larger** than it needs to be
wastedbytes	**Bytes of wasted space** = 97,648,640 bytes ≈ **93 MB** of unnecessary disk usage
iname	Name of the primary index = test_bloat_pkey
ibloat	**Index Bloat Ratio** = 0.0 → Index is perfectly optimized, no waste
wastedibytes	**Wasted space in index** = 0 bytes

After running Vacuum Full :-

```
current_database | schemaname | tablename  | tbloat | wastedbytes |      iname      | ibloat | wastedibytes
-----------------+------------+------------+--------+-------------+-----------------+--------+-------------
bank             | public     | test_bloat |    1.1 |     9879552 | test_bloat_pkey |    0.0 |           0
(1 row)
```

Column	Value	Meaning
current_database	bank	You're running this query in the bank database
schemaname	public	The schema where the table lives
tablename	test_bloat	Target table you're analyzing
tbloat	1.1	**Table bloat ratio** — this means the table is estimated to be 1.1x (or 10%) bigger than ideal
wastedbytes	9879552	≈ 9.42 MB of **extra disk space** used (compared to ideal size)
iname	test_bloat_pkey	This is the **primary key index** on the table
ibloat	0	**Index bloat ratio** — index is clean (perfect, no bloat)
wastedibytes	0	No space is wasted in the index

https://blogs.vmware.com/tanzu/autovacuum-tuning-in-gpdb7/
https://www.citusdata.com/blog/2022/07/28/debugging-postgres-autovacuum-problems-13-tips/
https://www.enterprisedb.com/blog/parallelism-comes-vacuum
https://www.enterprisedb.com/blog/postgresql-vacuum-and-analyze-best-practice-tips#section5
https://www.percona.com/blog/postgresql-vacuuming-to-optimize-database-performance-and-reclaim-space/
https://www.enterprisedb.com/blog/postgresql-vacuum-and-analyze-best-practice-tips
https://wiki.postgresql.org/wiki/Show_database_bloat