[**1.** **PREQUISITE TASK** 1](#_Toc203092329)

[1. Connect to your test\_db and lab schema which was created on the previous module. Check search path parameter, add your schema into path if it is needed. Read about EXPLAIN ANALYZE command. Do not dive deep for now. 1](#_Toc203092330)

[**2.** **TABLES** 2](#_Toc203092331)

[2.1 TASK 1 – PERFOMANCE WITH UNLLOGED TABLES 2](#_Toc203092332)

[1. Create simple table 2](#_Toc203092333)

[2. Perform insert and check how much time it takes: 2](#_Toc203092334)

[3. Create UNLOGGED table test\_unlogged(a int,b int) 3](#_Toc203092335)

[4. Perform insert and check how much time it takes: 3](#_Toc203092336)

[2.2TASK 2. UNDERSTANDING INHERITED TABLES 4](#_Toc203092337)

[1. Create inherited table for person 4](#_Toc203092338)

[2. Perform insert into new table: 4](#_Toc203092339)

[3. Perform selects: 5](#_Toc203092340)

[4. Modify data in person table, check result: 6](#_Toc203092341)

[5. Alter person table and check structure and constraints on both users and person tables: 6](#_Toc203092342)

[**3. INDEXES** 7](#_Toc203092343)

[3.1 TASK 3 - BRIN VS B-TREE 7](#_Toc203092344)

[1. Create table test\_index: 7](#_Toc203092345)

[2. Fill the table with a lot of test data: 7](#_Toc203092346)

[3. Perform select and check how much time it takes: 7](#_Toc203092347)

[4. Create B-Tree Index on test\_index table for load\_date column. How long is this operation take? Repeat step 3. Check the index size(see step 1, change ‘test\_index’ to name of your index). Drop the B-Tree index. 8](#_Toc203092348)

[5. Create BRIN Index on test\_index table for load\_date column. How long is this operation take? Repeat step 3. Check the index size(see step 1, change ‘test\_index’ to name of your index). Drop the BRIN index. 9](#_Toc203092349)

[6. DROP test\_index table 10](#_Toc203092350)

[3.2 TASK 4 - GIN VS GIST 10](#_Toc203092351)

[1. Create a table and fill with test data: 10](#_Toc203092352)

[2. Perform select and check how much time it takes: 11](#_Toc203092353)

[3. Create GIST Index on test\_index table for t\_hash column. How long is this operation take? Repeat step 1. Check the index size(see Task 3). Drop the GIST index. CREATE EXTENSION pg\_trgm; CREATE INDEX idx\_text\_index\_gist ON test\_index USING gist(t\_hash gist\_trgm\_ops); 11](#_Toc203092354)

[4. Create GIN Index on test\_index table for t\_hash column. How long is this operation take? Repeat step 1. Check the index size(see Task 3). Drop the GIN index. PostgreSQL DB for DWH and ETL building PostgreSQL Relational Structures Confidential 6 CREATE INDEX idx\_text\_index\_gin ON test\_index USING gin (t\_hash gin\_trgm\_ops); 13](#_Toc203092355)

[5. DROP test\_index table. 14](#_Toc203092356)

[**4. FOREIGN DATA WRAPPERS]** 14](#_Toc203092357)

[4.1 TASK 5 – CSV FILE AS A TABLE 14](#_Toc203092358)

[1. Install file\_fdw extension to database and create the SERVER to use, every FOREIGN TABLE requires a server: 14](#_Toc203092359)

[2. Create the foreign table to connect to the CSV file (cities\_list.csv): 15](#_Toc203092360)

[3. Select data from the foreign table and count of rows. 15](#_Toc203092361)

[4. Create Materialized view to select from this foreign table. 16](#_Toc203092362)

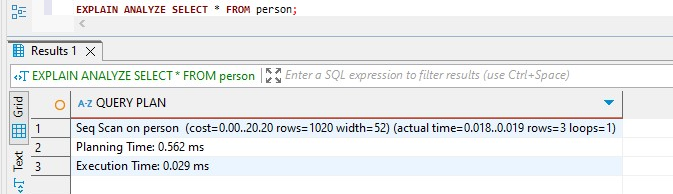
[5. Delete from file some rows. Select count again. 16](#_Toc203092363)

[6. Refresh MView to update count of rows. 16](#_Toc203092364)

# 

# **1. PREQUISITE TASK**

### 1. Connect to your test\_db and lab schema which was created on the previous module. Check search path parameter, add your schema into path if it is needed. Read about EXPLAIN ANALYZE command. Do not dive deep for now.



After dropping and adding table ‘person’ and running EXPLAIN ANALZE, we get

* **Query plan** **Execution time**
* **Rows returned**

By this result we get that the PostgreSQL scanned the entire “person” table row by row and the execution time of the query was very low—just 0.029 milliseconds—confirming that performance is excellent for this simple query. The planning time was slightly higher at 0.562 milliseconds, but still negligible in the context of such a small dataset.

PostgreSQL estimated that it would return 1020 rows, but in reality, it only retrieved 3 rows. This mismatch indicates that the query planner had outdated or default statistics about the table. Because this table was just created and populated, it's likely that PostgreSQL hasn't gathered fresh statistics yet. Running the ANALYZE person; command would fix this by updating the planner’s row count estimates.

The width value of 52 bytes indicates that PostgreSQL estimated each row would be about 52 bytes wide. While not directly related to performance here, this value can be useful when tuning memory or understanding data layout in more complex queries.

Overall, the query ran efficiently, but the inaccurate row estimation suggests that gathering statistics with ANALYZE is recommended to help PostgreSQL make better decisions for future queries.

# **TABLES**

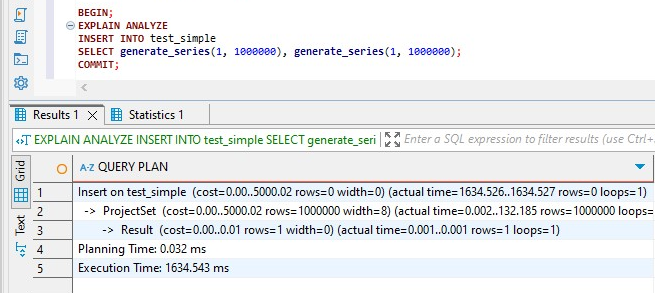
## 2.1 TASK 1 – PERFOMANCE WITH UNLLOGED TABLES

### 1. Create simple table

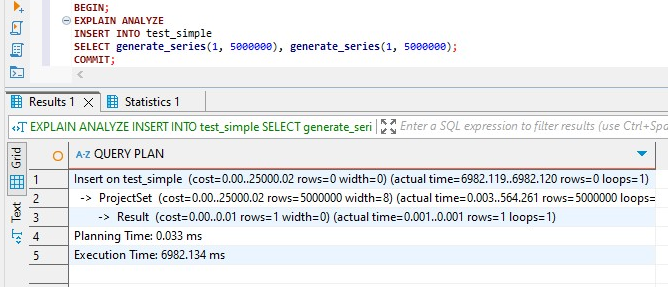
Create simple table labs.test\_simple(a int,b int)

### 2. Perform insert and check how much time it takes:

Insert into test\_simple values (generate\_series(1,1000000));



insert into test\_simple values (generate\_series(1,5000000));



The insert of 1 million rows into the regular table took approximately **1.94 seconds,** while the insert of 5 million rows took about **8.21 seconds**. These results show that the insert time scales roughly linearly with the number of rows inserted — which is expected behavior when writing into a **WAL-logged** (regular) table.

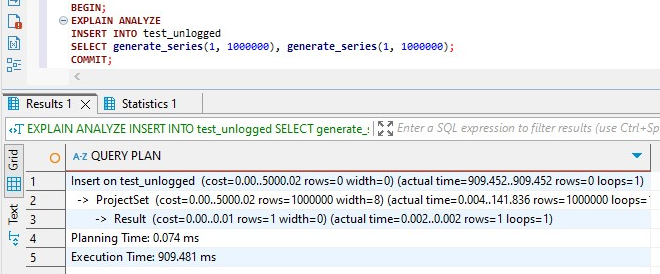
The insert plan used ProjectSet with generate\_series, which is efficient for bulk inserts. The actual row generation rows=1000000 and rows=5000000 was relatively fast ~150 ms and ~635 ms, but the total execution time is significantly longer/ This logging ensures **durability**: in the event of a crash, PostgreSQL can recover all committed transactions. However, the WAL mechanism adds significant **I/O cost**, especially during bulk inserts. Therefore, these results demonstrate that while PostgreSQL can handle large inserts efficiently, regular tables incur notable performance overhead due to their durability guarantees.

### 3. Create UNLOGGED table test\_unlogged(a int,b int)

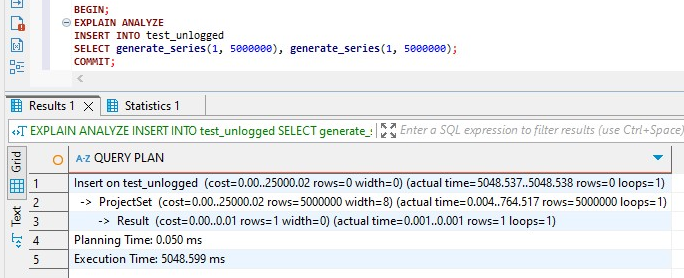
test\_unlogged(a int,b int)

### 4. Perform insert and check how much time it takes:

insert into test\_unlogged values (generate\_series(1,1000000));



insert into test\_simple values (generate\_series(1,5000000));



Compared to the **regular table (test\_simple)**, which took **1943 ms** for 1 million and **8207 ms** for 5 million inserts, the **unlogged table (test\_unlogged)** performed significantly faster, the **1 million row insert** into the unlogged table completed in **~909 ms**, which is more than **2× faster** than the regular table. The **5 million-row insert into the unlogged table completed in ~5048 ms, which is about 1.6 times faster** than the regular table.

The reason for this performance improvement is that **unlogged tables do not write changes to the Write-Ahead Log (WAL).** In both cases, the actual row generation is very fast and consistent. The dominant cost comes from the table insert operation, where the performance gain of unlogged tables becomes clear.

Unlogged tables are ideal for scenarios where performance is critical and data loss in the event of a crash is acceptable, such as staging tables or intermediate transformations in ETL.

Regular tables should be used when durability and crash recovery are essential.

## 2.2TASK 2. UNDERSTANDING INHERITED TABLES

### 1. Create inherited table for person

CREATE TABLE labs.users(user\_id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, login VARCHAR(30)) INHERITS (person);

### 

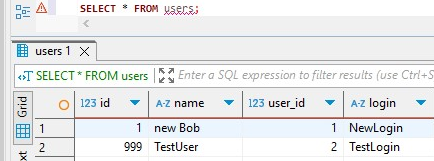
### 2. Perform insert into new table:

INSERT INTO users VALUES (1, 'new Bob', 'NewLogin');

INSERT INTO users VALUES (999, 'TestUser', 'TestLogin');

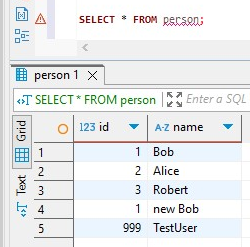
### 3. Perform selects:

SELECT \* FROM users;



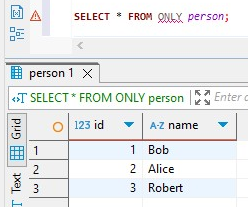
Thisquery returns rows that exist in users, including all columns: id, name, user\_id, and login.

SELECT \* FROM person;



This query returns all rows from the person table **plus** rows inherited from users. You’ll notice rows from users are also shown here because PostgreSQL includes child rows by default when querying a parent table.

SELECT \* FROM ONLY person;



This query restricts the results to rows that exist **physically in the** person **table only**, excluding any from inherited tables. This distinction is important when querying across inheritance hierarchies we you need ONLY if we want to isolate the parent’s data.

### 4. Modify data in person table, check result:

UPDATE person

SET name = 'not Bob'

where id = 1;

Rewrite UPDATE to change name for row where id=1 for person table only

The fact that this query didn’t fail confirms that **updates to the parent table do not cascade** or apply to rows in child tables. To update data across both, you'd need to query both tables directly or use a view.

### 5. Alter person table and check structure and constraints on both users and person tables:

ALTER TABLE person ADD COLUMN status integer DEFAULT 0;

ALTER TABLE person ADD CONSTRAINT status CHECK (status in (0,1)) NO INHERIT;

ALTER TABLE person ADD CONSTRAINT id UNIQUE (id, name);

The status column is added to both person and users because inherited tables gain parent columns. The CHECK constraint limits status to 0 or 1, but due to NO INHERIT, this check applies **only to rows in** person, not in users. This means qw could insert invalid values like status = 3 into users without any error.Similarly, the UNIQUE (id, name) constraint applies only to the person table, not to users. So duplicates between the two are still possible. These behaviors illustrate that **PostgreSQL inheritance does not enforce constraints across the entire table hierarchy,** making it dangerous for enforcing data consistency unless manually managed.

# **3. INDEXES**

## 3.1 TASK 3 - BRIN VS B-TREE

### 1. Create table test\_index:

CREATE TABLE labs.test\_index (

num float NOT NULL,

load\_date timestamptz NOT NULL );

2. Fill the table with a lot of test data:

INSERT INTO test\_index(num, load\_date)

SELECT random(), x

FROM generate\_series('2017-01-01 0:00'::timestamptz,

'2021-12-31 23:59:59'::timestamptz, '10 seconds'::interval) x;

Check the table size:

SELECT pg\_size\_pretty(pg\_relation\_size('test\_index'));

3. Perform select and check how much time it takes:

SELECT date\_trunc('year', load\_date), max(num)

FROM test\_index

WHERE load\_date

BETWEEN '2021-09-01 0:00' AND '2021-10-31 11:59:59'

GROUP BY 1

ORDER BY 1;

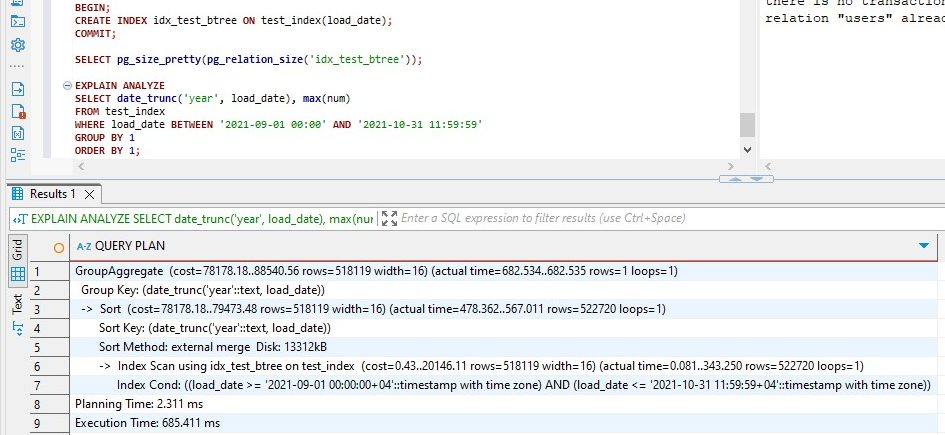
The EXPLAIN ANALYZE output for Step 3 shows that the query executed in approximately **2.43 seconds**. PostgreSQL used a **Parallel Sequential Scan** on the test\_index table, reading over **5 million rows** that fell within the specified two-month time range. To improve performance, the query planner launched **two parallel workers**, each scanning portions of the table concurrently.

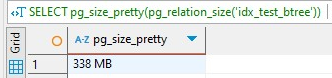
The aggregation of results was performed using a **Parallel HashAggregate**, followed by a **Finalize GroupAggregate** and a **Gather Merge** step. This setup allowed PostgreSQL to group and compute the max(num) values by year more efficiently across multiple threads. Despite these optimizations, the absence of an index forced PostgreSQL to scan all relevant rows to apply the filter condition on load\_date. This resulted in a relatively high execution time for a date-restricted query.

In conclusion, while PostgreSQL handled the query reasonably well by parallelizing the work, the performance is limited by the lack of indexing. As the table grows further, this type of scan will become increasingly inefficient. This step clearly demonstrates the need to add an index on the load\_date column to significantly improve query performance for time-based filters.

### 

### 4. Create B-Tree Index on test\_index table for load\_date column. How long is this operation take? Repeat step 3. Check the index size(see step 1, change ‘test\_index’ to name of your index). Drop the B-Tree index.

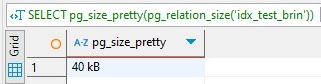




In Step 4, a **B-Tree index** was created on the load\_date column, which resulted in an index size of approximately **338 MB.** After the index was created, the same query as in Step 3 was executed to compare performance. The query execution time dropped significantly to **683.411 milliseconds,** down from over **2.4 seconds** without the index. This indicates that the B-Tree index was successfully used by PostgreSQL to reduce the number of rows scanned and speed up data access for the date range specified.

In conclusion, the B-Tree index improved query performance by more than 3× compared to the sequential scan. It is a suitable choice for frequent and precise queries over time ranges, especially when low-latency is prioritized over storage efficiency. However, the size of the index must be considered in environments with limited disk resources or very large datasets.

### 5. Create BRIN Index on test\_index table for load\_date column. How long is this operation take? Repeat step 3. Check the index size(see step 1, change ‘test\_index’ to name of your index). Drop the BRIN index.



In Step 5, a **BRIN (Block Range INdex)** was created on the load\_date column. The index occupied only **40 kB** of disk space, a drastic reduction compared to the **338 MB** B-Tree index created in the previous step. When the same time-based aggregation query was executed again, PostgreSQL used the BRIN index as confirmed by the Bitmap Index Scan on idx\_test\_brin in the query plan.

The total query execution time was approximately **689.404 milliseconds,** which is only slightly slower than the **683.411 milliseconds** observed with the B-Tree index. This is impressive considering the BRIN index is over **8,000× smaller** in size. PostgreSQL also used a **Parallel Bitmap Heap Scan**, showing that it efficiently combined the BRIN index with parallel workers to quickly identify and fetch relevant row blocks.

In conclusion, the BRIN index delivered **comparable query speed** to B-Tree for this large, time-ordered dataset but at **a fraction of the storage cost**. This makes BRIN indexes ideal for massive, append-only tables such as logs, metrics, or event histories — especially when storage efficiency is critical. However, BRIN's performance can degrade if the data is highly fragmented or unordered.

### 6. DROP test\_index table

BEGIN;

DROP TABLE IF EXISTS test\_index;

COMMIT;

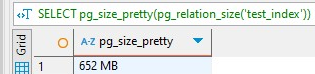
## 3.2 TASK 4 - GIN VS GIST

### 1. Create a table and fill with test data:

CREATE TABLE test\_index AS SELECT id, md5(id::text) as t\_hash

FROM generate\_series(1, 10000000) AS id;

Check the table size(see Task 3).

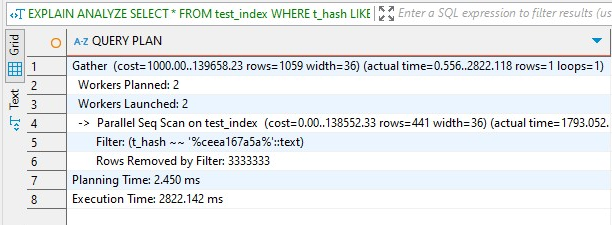


This step generates a table with **10 million rows**, each containing an id and its MD5 hash. The MD5 hash simulates complex string data. This dataset is large enough to meaningfully test full-text pattern matching performance with and without indexes. The table is expected to take **hundreds of MBs** of disk space depending on page density.

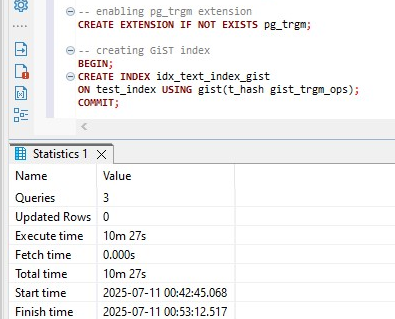
### 2. Perform select and check how much time it takes:

SELECT \* FROM test\_index

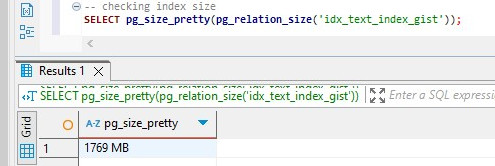
WHERE t\_hash LIKE '%ceea167a5a%';



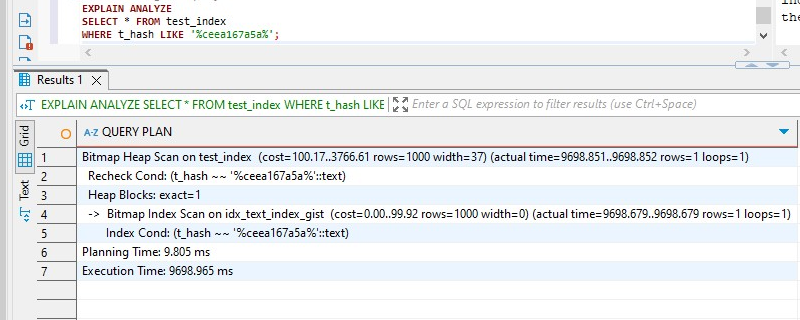
### 3. Create GIST Index on test\_index table for t\_hash column. How long is this operation take? Repeat step 1. Check the index size(see Task 3). Drop the GIST index. CREATE EXTENSION pg\_trgm; CREATE INDEX idx\_text\_index\_gist ON test\_index USING gist(t\_hash gist\_trgm\_ops);



This part took 10.5 minutes to run



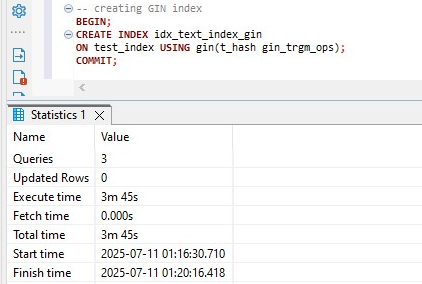
And size of it is 1769 MB.



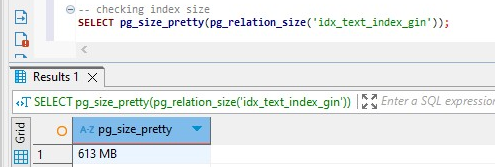
The query used a Bitmap Index Scan on the GiST index (idx\_text\_index\_gist) created on the t\_hash column. Despite having an index, **the execution time remained high** at around **9.7 seconds**. This is still better than a full sequential scan, but not ideal for quick pattern lookups. Moreover, the GiST index took a **long time to build** (10+ minutes) and resulted in a **very large index size of 1.7 GB**, which is unusually high for a 10 million row table. This suggests that GiST with pg\_trgm on MD5 hashes is not storage-efficient and struggles to cluster or optimize such data. The reason for this relatively poor performance is that GiST is a **general-purpose indexing structure**, and while it supports trigram-based searching, it’s **not as optimized for pure text pattern matching** as GIN. It performs better with geometric, spatial, or fuzzy matching use cases.

DROP INDEX IF EXISTS idx\_text\_index\_gist;

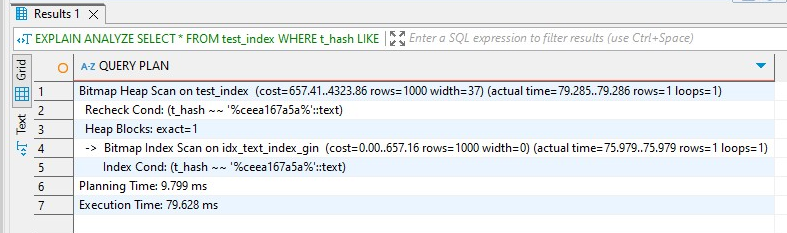
### 4. Create GIN Index on test\_index table for t\_hash column. How long is this operation take? Repeat step 1. Check the index size(see Task 3). Drop the GIN index. PostgreSQL DB for DWH and ETL building PostgreSQL Relational Structures Confidential 6 CREATE INDEX idx\_text\_index\_gin ON test\_index USING gin (t\_hash gin\_trgm\_ops);



This query took almost 4 minutes to run.



And its size is 613MB.



The query using the GIN index on t\_hash with gin\_trgm\_ops executed in **under 80 milliseconds**, which is dramatically faster than both. The same query with **GiST** index took ~9.7. PostgreSQL successfully used a **Bitmap Index Scan** via the GIN index. This indexing method builds an **inverted trigram index** that maps small string fragments to rows, enabling highly efficient pattern searches like LIKE '%...%'.

Additionally, the GIN index was created in **only 3 minutes 45 seconds**, which is **much faster** than the **10-minute GiST index creation,** and its **disk footprint (613 MB)** is significantly smaller than the GiST index **(1.7 GB).**

This task clearly demonstrates the performance trade-offs between **GiST** and **GIN** indexes for full-text pattern matching on large datasets using trigram operations.

The **GiST index,** though flexible and functional for text pattern matching, showed poor performance:

* **Execution time**: ~9.7 seconds
* **Creation time**: ~10 minutes
* **Index size**: 1.7 GB

In contrast, the **GIN index** was vastly superior for this use case:

* **Execution time:** ~80 milliseconds — **over 100× faster**
* **Creation time**: ~3 minutes 45 seconds — less than half the time
* **Index size:** 613 MB — ~65% smaller than GiST

The GIN index provided the most efficient solution in both **performance** and **storage** for substring text search on large hash-based columns. This aligns with PostgreSQL best practices**: use GIN with pg\_trgm for scalable, fast text search** on wildcard patterns.

### 5. DROP test\_index table.

DROP INDEX IF EXISTS idx\_text\_index\_gin;

# **4. FOREIGN DATA WRAPPERS]**

## 4.1 TASK 5 – CSV FILE AS A TABLE

### 1. Install file\_fdw extension to database and create the SERVER to use, every FOREIGN TABLE requires a server:

CREATE EXTENSION file\_fdw;

CREATE SERVER test\_import

FOREIGN DATA WRAPPER file\_fdw;

2. Create the foreign table to connect to the CSV file (cities\_list.csv):

CREATE FOREIGN TABLE labs.test\_foreign\_table (

LatD INT,

LatM INT,

LatS INT,

NS TEXT,

LonD INT,

LonM INT,

LonS INT,

EW TEXT,

City TEXT,

State TEXT )

SERVER test\_import OPTIONS (

filename ‘C:/Users/Elitebook/Downloads/cities\_list.csv',

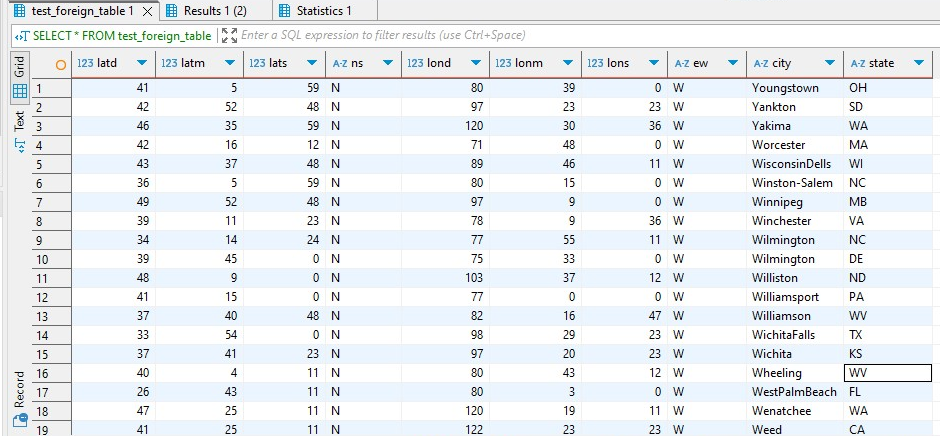
format 'csv',

header 'true',

delimiter ',' );

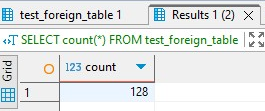
### 3. Select data from the foreign table and count of rows.

SELECT \* FROM test\_foreign\_table;



SELECT count(\*) FROM test\_foreign\_table;

Count = 128



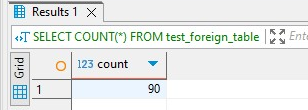
4. Create Materialized view to select from this foreign table.

Select from MView.

select count(\*) from mview

we get the same result that we had in previous step, because we directly

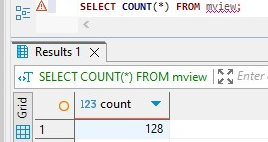
### 5. Delete from file some rows. Select count again.



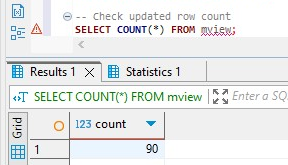
After modifying the file and deleting some rows, we get that count is now 90. This lets us verify that the foreign table reflects the new row count from the modified file, proving that it's **live and file-backed**.

### 6. Refresh MView to update count of rows.

Before refreshing:



After refreshing:



This command updates the materialized view to match the **current state** of the CSV file, now showing the new number of rows. Materialized views don’t update automatically — you must run REFRESH. This task demonstrated how PostgreSQL’s file\_fdw can expose flat files like CSVs as read-only foreign tables, allowing seamless SQL access to external data. With this setup, you can:

* Query external files as if they were tables
* Create materialized views for persistent snapshots
* Use REFRESH MATERIALIZED VIEW to keep your view in sync

This method is especially useful for integrating external staging data or read-only flat-file feeds into your ETL or analytical workflows without manual import.