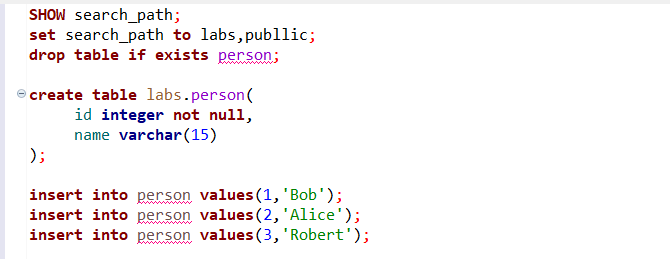


| Business Template  **Relational Structures** |
| --- |
|  |

## 1. Prerequisite Task

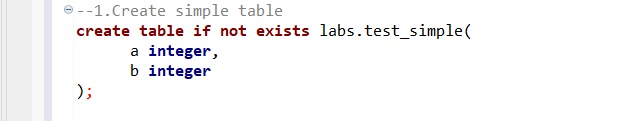
* Connect to the test\_db database and the lab schema that you created in the previous module.
* Check the search\_path parameter to ensure your schema is included. If it is not, please add it to the path.
* Drop the person table from the previous module, if it exists, and recreate it.
* Populate the person table using the provided CREATE statement.

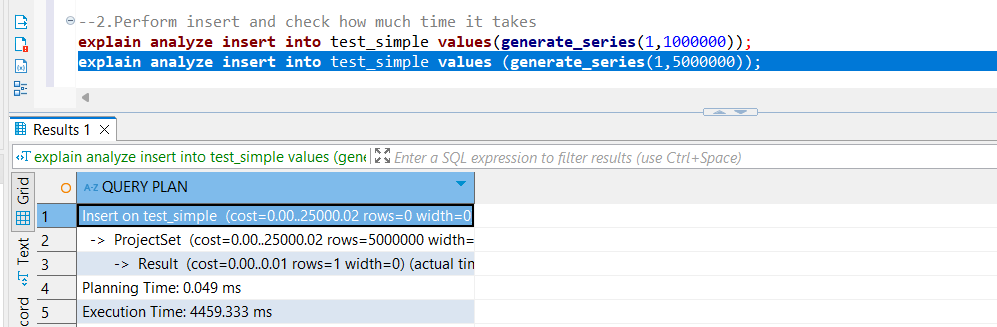
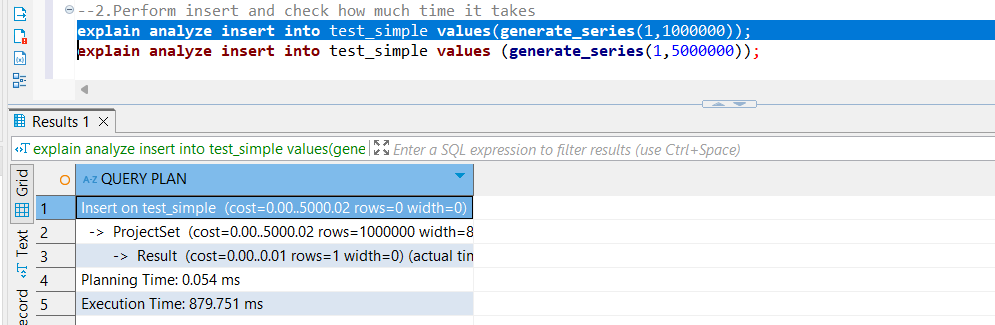


## 2. Tables

### 2.1 Task 1 – Performance with Unlogged Tables

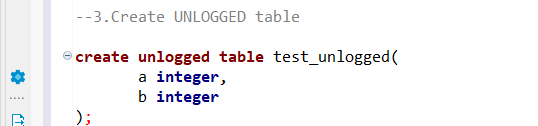
**Step 1:** Create a simple table  
 Define a standard table named test\_simple with two integer columns: a and b.



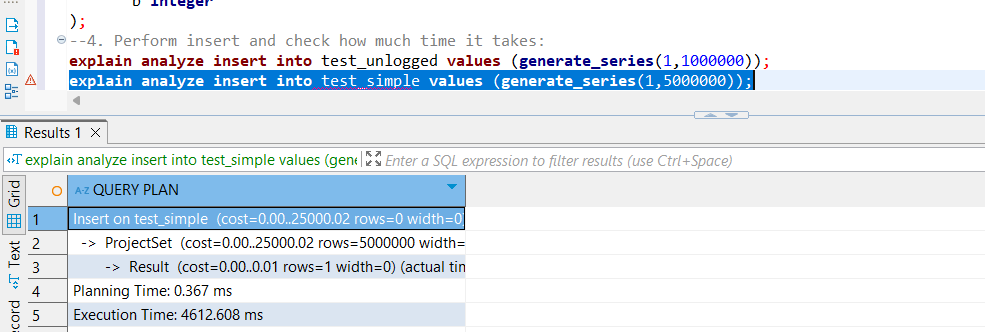
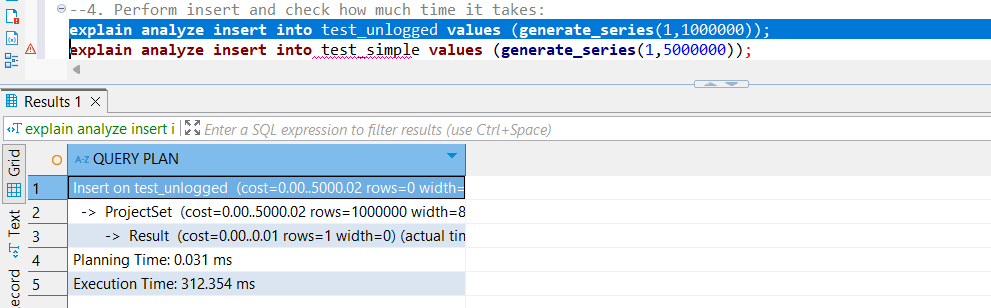
**Step 2:** Insert data into the simple table and measure performance  


Based in the results the performance statistics of this query obtained vua the explain analyze command show around 879.71 milliseconds for the first insert while a whooping 4459.333 milliseconds for the second one, which given that the number of variables to be inserted is 5 times larger seems fair.

**Step 3:** Create an unlogged table  
 Define a similar table named test\_unlogged, also with two integer columns.



**Step 4:** Insert data into the unlogged table and measure performance



Based on the results we can observe that the pattern persists, basically when we have a way larger number of rows the time will way larger. Firstit takes only 312 millisecond which was way faster than bore but for the insertion of the second generated number it takes longer than before which seems unexpcted given that unlogged tables are deemed to perform better.

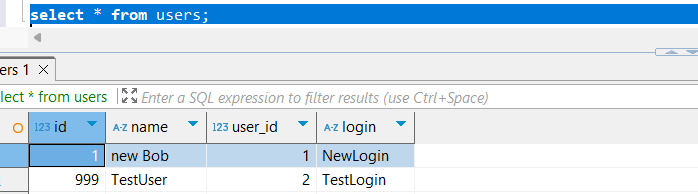
### 2.2 TASK 2 – UNDERSTANDING INHERITED TABLES

### Step 1: Create Inherited Table for person

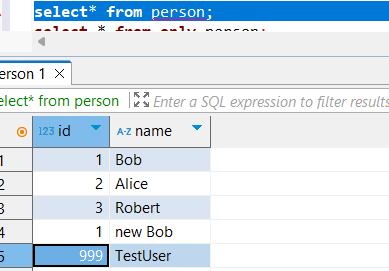
### Step 2: Insert into New Table

### Step 3: Perform SELECT Queries

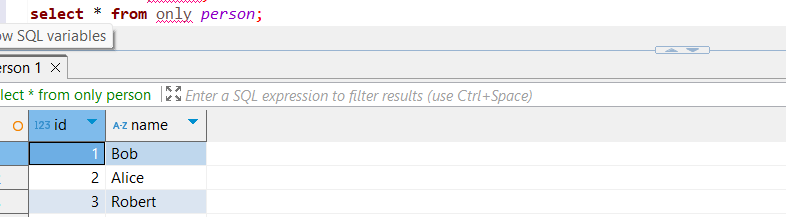
-- View all data from the child table (includes inherited columns)



-- View all data from the parent table (includes data inserted into child as well)



-- View only data directly inserted into the parent table

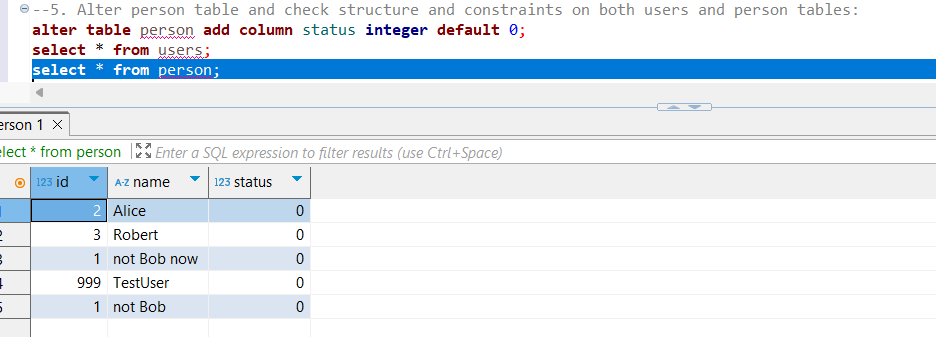
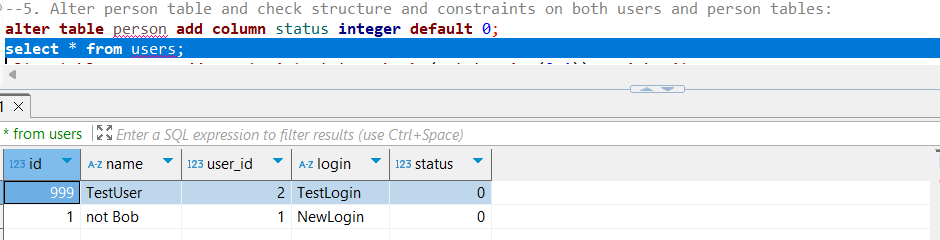


As we can see if we add data to the child table, this will be transferred into the parent table as well. More than that, we can manage to say that as well the id gets transferred, besides when we make a table to inherit another we basically apply the same structure to the first one and all other columns added by us in the child table will be additional to the ones already existing.

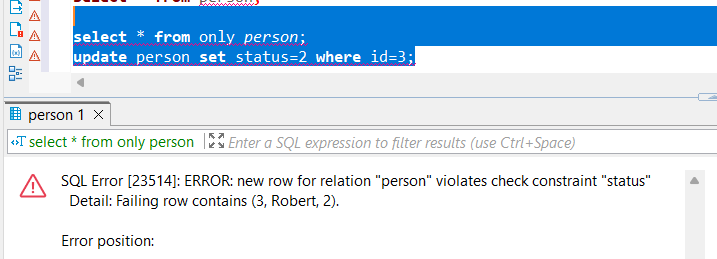
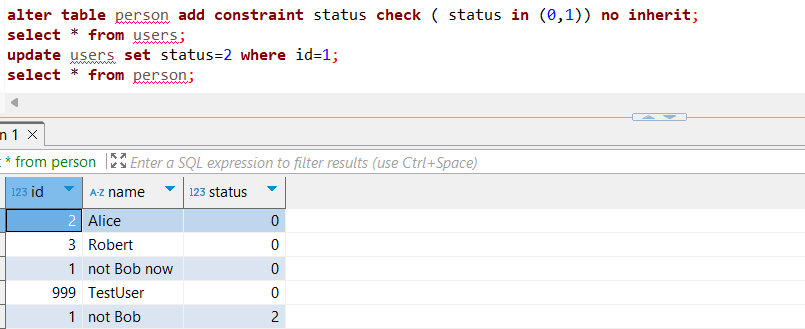
### Step 4: Modify Data in person Table.**Rewrite to update ONLY person (not children):**

As we cn see in the screenshot when we modify the name of bob without restricting to the specific parent table this change will propagate into the child table, bute when we restrict our update, this will only modify the entry in the parent table.

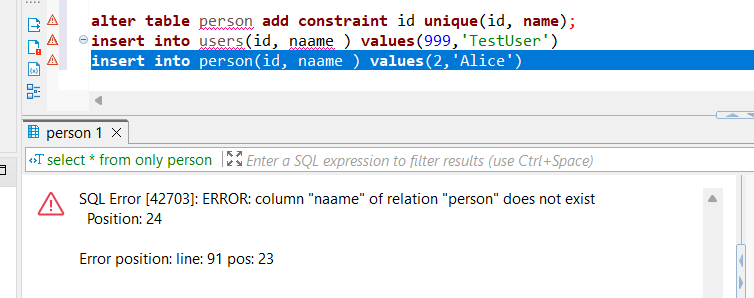
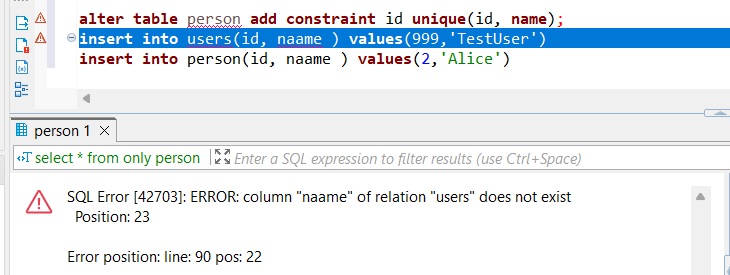
### Step 5: Alter person Table and Check Impact on Both Tables



As we can altering the parent table by adding a new column with a new default value, this will be added in the child table as well.



While running these queries I noticed that if a user is updated with a status value larger than 1 in the child table will not throw an error, but if the same is done in the parent table for a person created during this session, the program will throw an error. Interestingly enough, if we do change the status for a person created in the user table but doing so in the parent table no error will be thrown



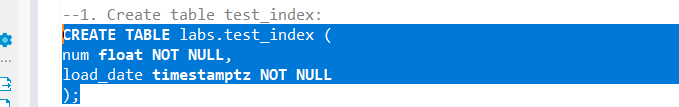
As we inserting a tuple that does not have a unique combination of id and name to the previous tuples already existing in either peron table or user, an error will be thrown immediately in both situations as withouthout setting NO INHERIT the constraint will be propagated from the parent to the child.

## 3. Indexes

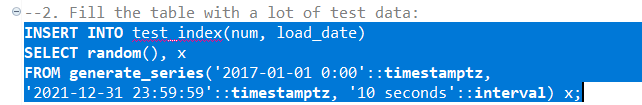
### 3.1 TASK 3 – BRIN vs B-TREE

**Task Result:** Provide queries where needed. Add your personal description of what happened and why, with screenshots where applicable.  
 *You may also use EXPLAIN ANALYZE for SELECT queries to inspect and compare execution plans.*

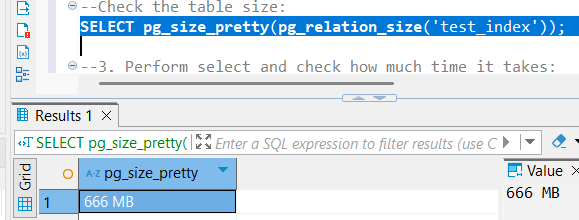
#### **Step 1: Create the test\_index Table**



#### **Step 2: Populate the Table with a Large Volume of Test Data**

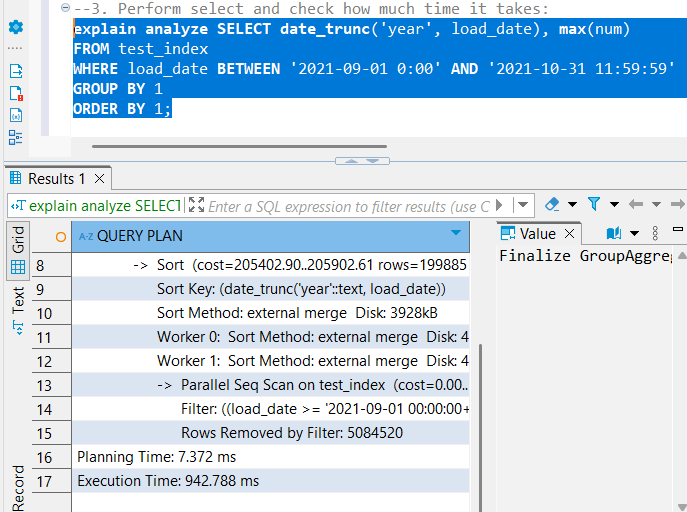


**Check the Table Size**



The size is 666 MB

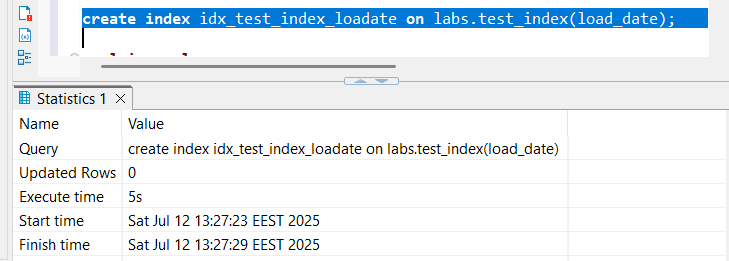
#### **Step 4: Run the SELECT Query and Measure Execution Time**



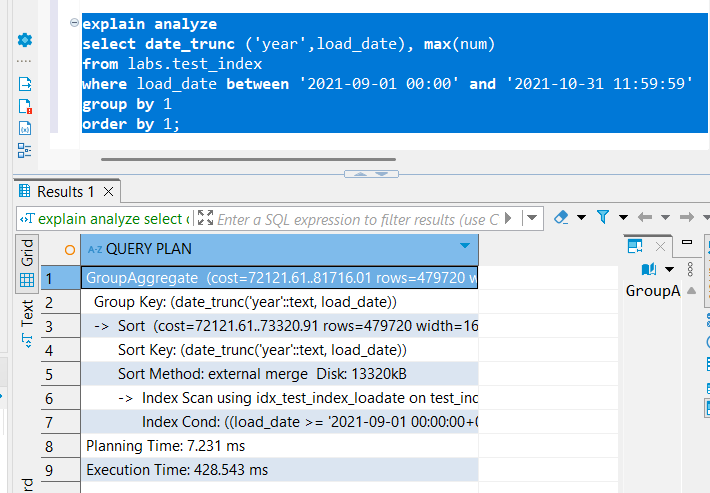
It took around 943 ms to run this query.

#### **Step 5: Create a B-Tree Index on load\_date**

* Measure how long the index creation takes.

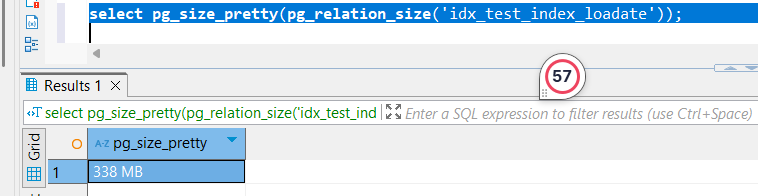


It took in total 5 seconds to create the index

* Repeat the SELECT query from Step 4.  
  

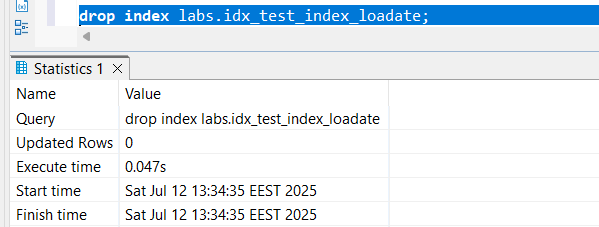
As we can see, performing the same SELECT operation as before nearly halved the execution time. This improvement makes sense because a B-tree index organizes data in a balanced tree structure, where each branch evenly splits the data. As a result, search operations have consistent performance, since traversing the tree involves moving through equidistant and evenly distributed branches, allowing efficient access to the relevant data range.

* Check the index size:



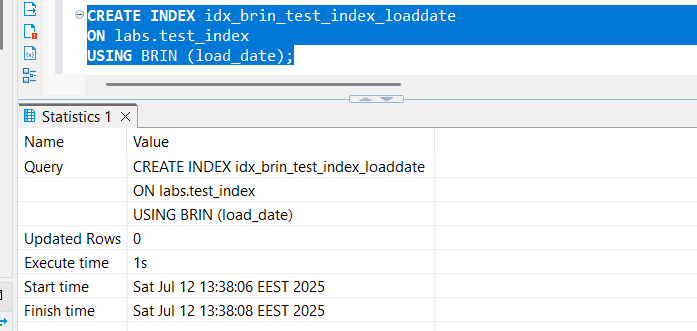
The index size is 338 MB

* Drop the B-Tree index:



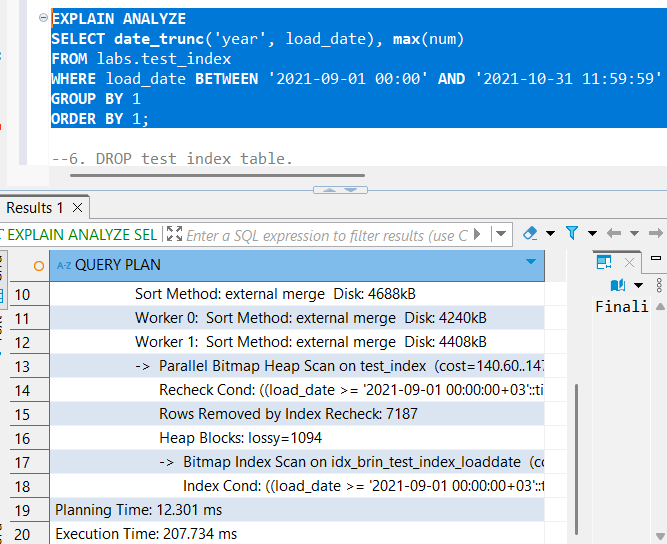
#### **Step 6: Create a BRIN Index on load\_date**

* Measure how long this index creation takes.



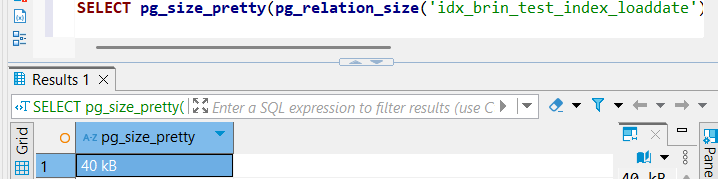
It took in total 1 seconds to create the index, which makes sense as this index is usually faster to build and takes up less space than b tree

* Repeat the SELECT query from Step 4.



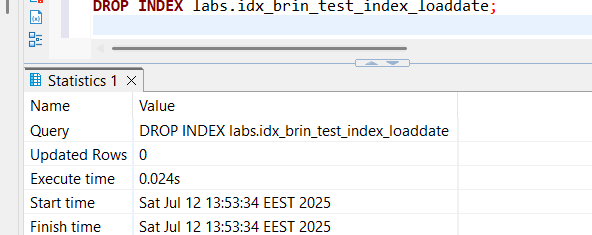
The time it took was around 208 milliseconds, which was two times faster than the B tree index. This ,ay make sense as BRIN (Block Range Index) is optimized for large, sequentially ordered data (like timestamps).

* Check the BRIN index size:



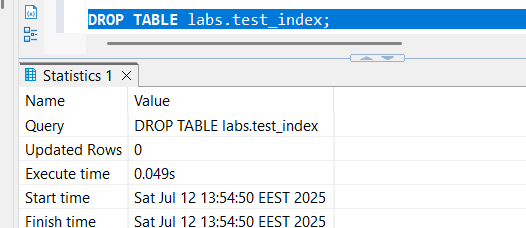
The size of the index is pretty small comare to the Btree index which again makes sense as BRIN index is more compact.

* Drop the BRIN index:



The index was dropped pretty fast compared to the b tree which makes sense given that it occupies less space.

#### **Step 7: Drop the test\_index Table**



### 3.2 TASK 4 – GIN vs GIST

**Task Result:** Provide queries where needed. Add your personal description of what happened and why, with screenshots where needed.

#### **Step 1: Create a Table and Fill with Test Data**

The size of the index is 652 MB .

#### **Step 2: Perform a SELECT Query and Measure Time**

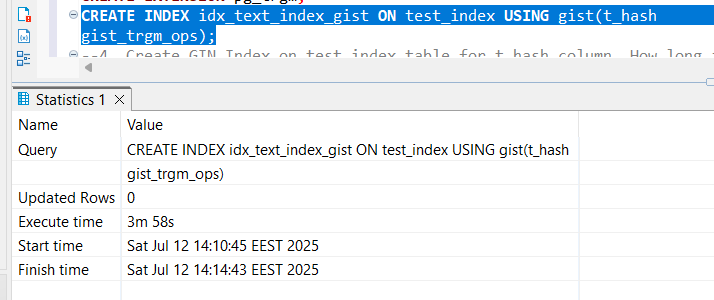
The execution time was 856.306 milliseconds.

#### **Step 3: Create a GIST Index on t\_hash Column**

1. Enable the required extension:CREATE EXTENSION pg\_trgm;

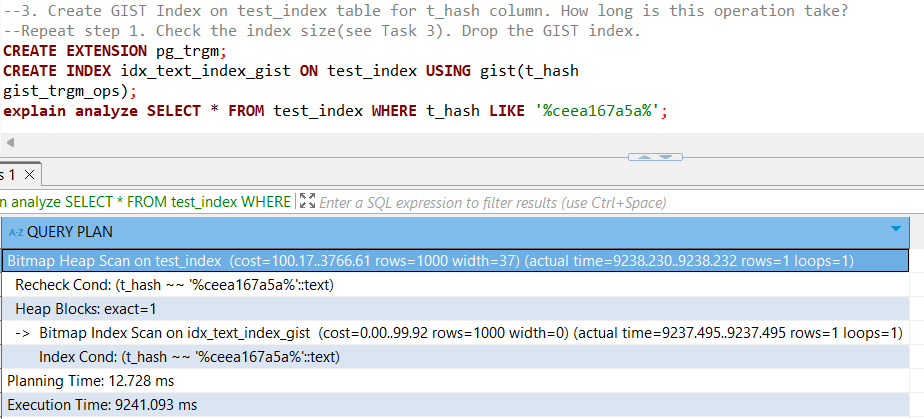


1. Create the GIST index and measure how long it took



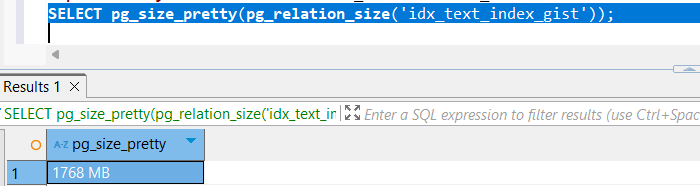
It took almost 4 minutes to create the index which makes sense logically as while the GIST index has a similar structure to the B tree is used in more complex queries that go beyoonf equality or range comparisons handled by the B tree.

1. Repeat Step 2(select)



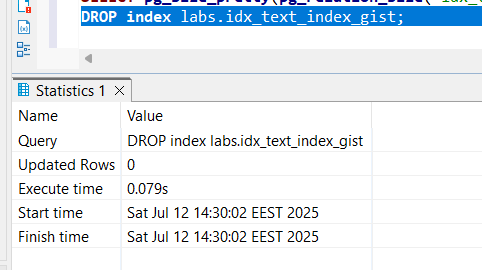
It took a whooping 9241 milliseconds, ehich unexpectedly is way longer than the table without this index. The issue might be that while this index is highly flexible, it is not optimized for simple equality or range queries, and might not be used efficiently by the planner — or used at all.

1. Check index size(see task 3.1)



The size as well is pretty large as I previously mentioned the operations acheived by this index are pretty complex so it has to store more complex indexes compared to B tree.

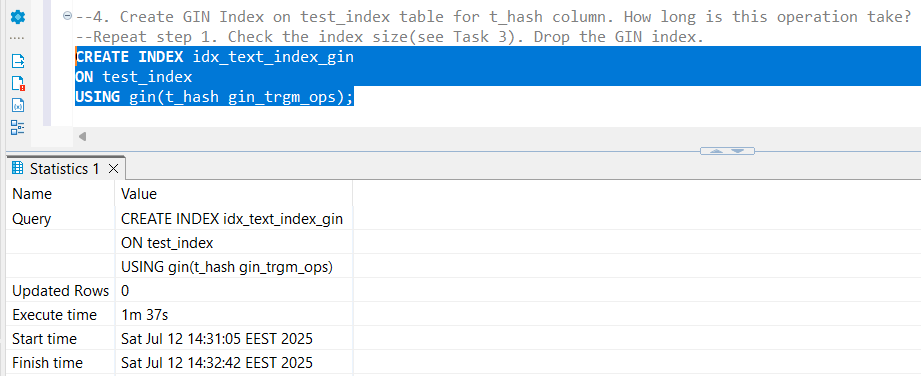
1. Drop the GIST index



#### **Step 4: Create a GIN Index on t\_hash Column**

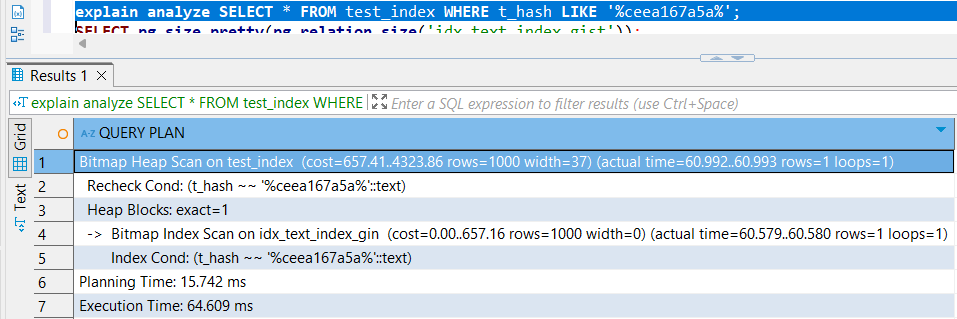
It took almost 4 minutes to create the index which makes sense logically as while the GIST index has a similar structure to the B tree is used in more complex queries that go beyoonf equality or range comparisons handled by the B tree.

1. Create the GIN index and measure how long it took



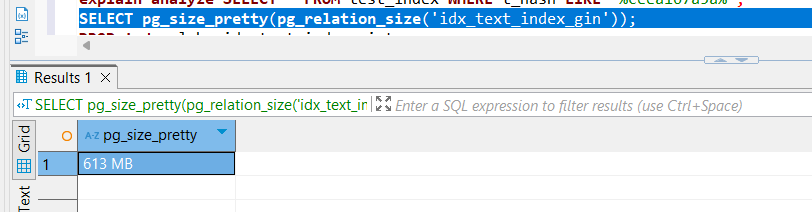
It took a 1minute and 37 seconds time, which while still big compared to BRIN or B tree is still more reasonable than GIST, which makes sense because the main purpose of gin (generaized inverted index) is to store a list of keys with what’s called a posting list of rows, each of which contain a key. It basically has a role of lookup while its indexes are based on a tree based structure. It does not inherit the ability to perform very complex queries , but is useful for indexing array values, , one of the ways to implement full text search.

1. Repeat Step 2(select)

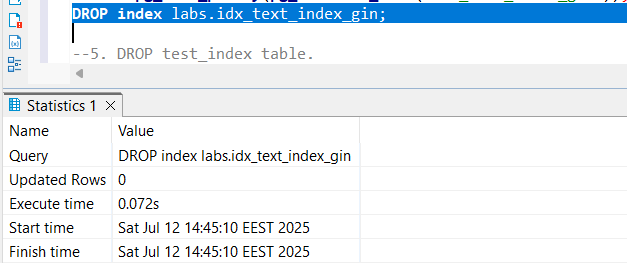


Now, we can see that the time it took to execute this select query is way shorter than before, making sense because the main purpose of this index is lookup which is eaxctly what this query is doing looking up for sr spme values that match a string.

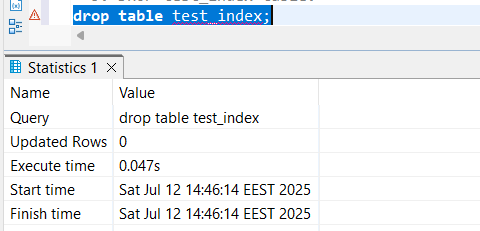
1. Check index size(see task 3.1)



This time the size is quite smaller which makes snese in context,

1. Drop the GIN index  
   
2. The dropping time was a bit smaller but still similar.

#### **Step 5: Drop the Test Table**

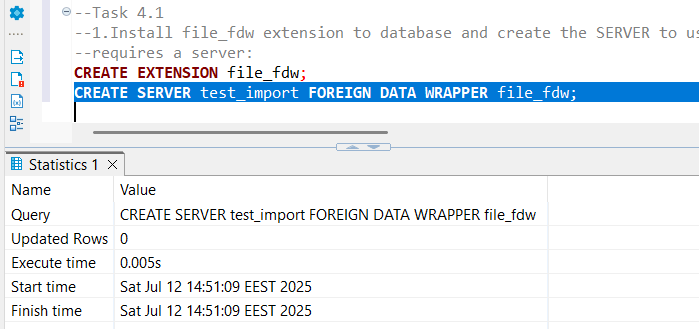


## 4. FOREIGN DATA WRAPPERS

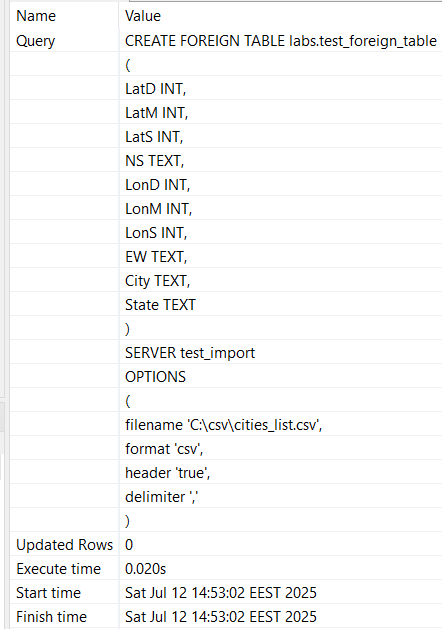
### 4.1 TASK 5 – CSV FILE AS A TABLE

**Task Result:** Provide queries where needed. Add your personal description of what happened and why, with screenshots where applicable.

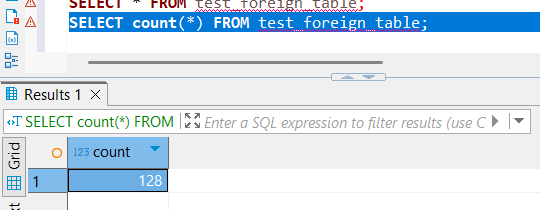
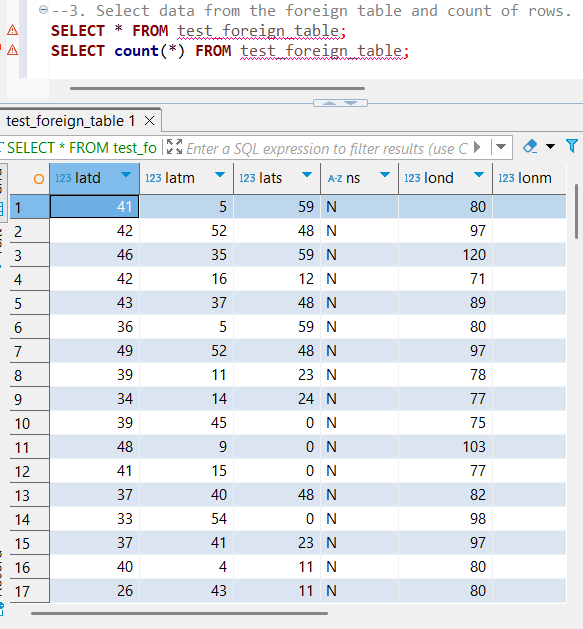
### Step 1: Install file\_fdw Extension and Create Server



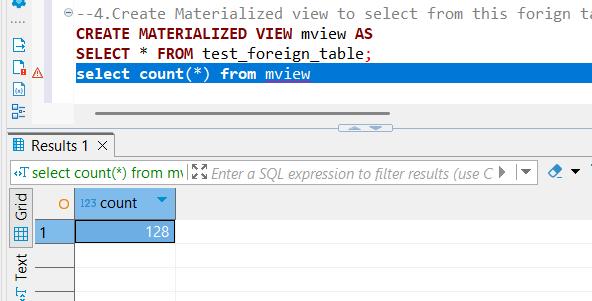
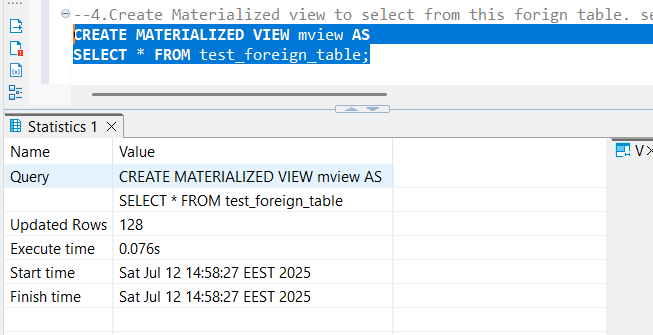
### Step 2: Create a Foreign Table for the CSV File



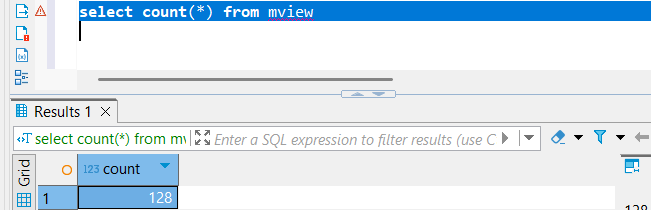
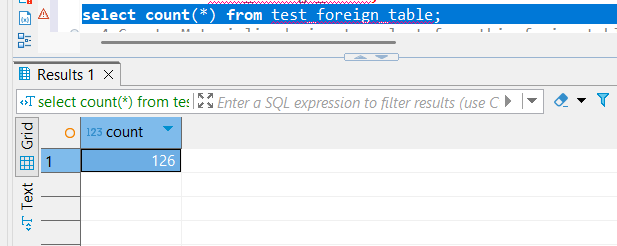
### Step 3:Select data from the foreign table and count of rows



### Step 4:Create Materialized view to select from this foreign table. Select from MView.

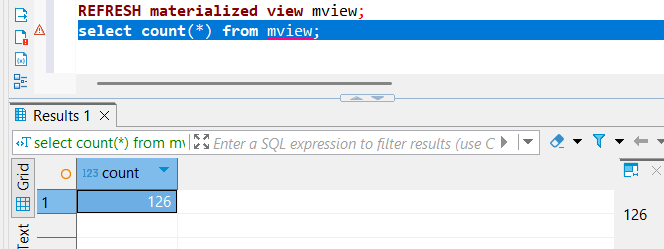


### Step 5: Delete from file some rows. Select count again.



Note: The materialized view still shows the **old** count until refreshed.

### Step 6: Refresh the Materialized View



After I refreshed the number was updated which makes sense as comapred to normal views the materialized views until refreshed can shoe stale data, so we need to run refresh mateirlaed view to see the latest updates and rebuild cache.