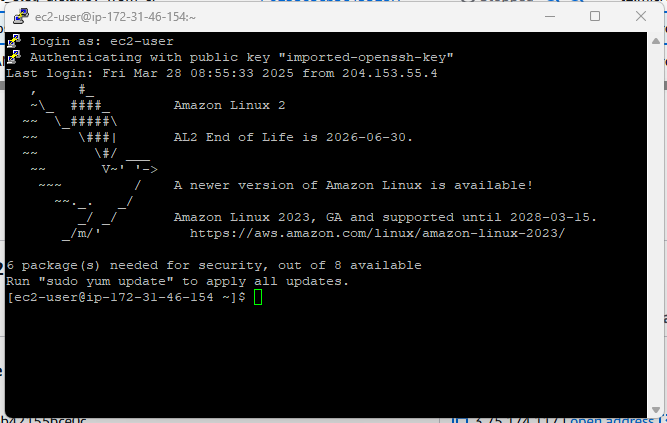
**Task 3**

**1. Login to Redshift. The EPAM VPN should be enabled. The public access to cluster is disabled, therefore you can use SSH tunnel to establish connection. Use the EC2 machine from HW2 or launch the new one.**

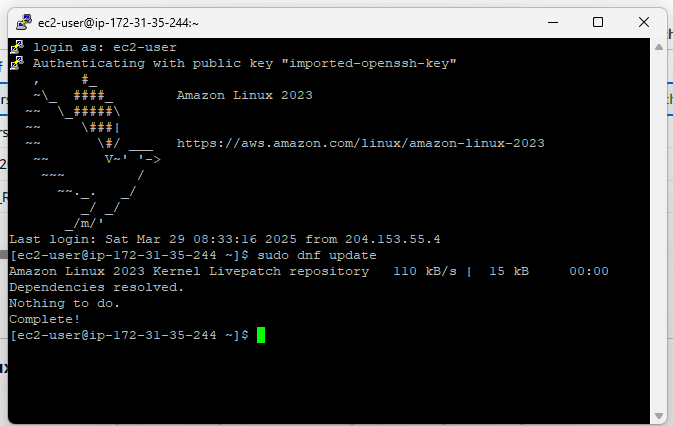
Firstly I create new EC2 with amazon linux2023 (because previous I installed Linux 2)

Next I need to create SSH tunnel. For that I use my existing EC2 Machine with Amazon Linux. Open VPN connection, then open Putty connect ty my machine and then configure SSH tunnel. I add EC2 Public DNS and add SSH private key + need to add Tunnel: **data-bi-lab-redshift-cluster-3.cettexdsxw3v.eu-central-1.redshift.amazonaws.com:5439** and port:5439

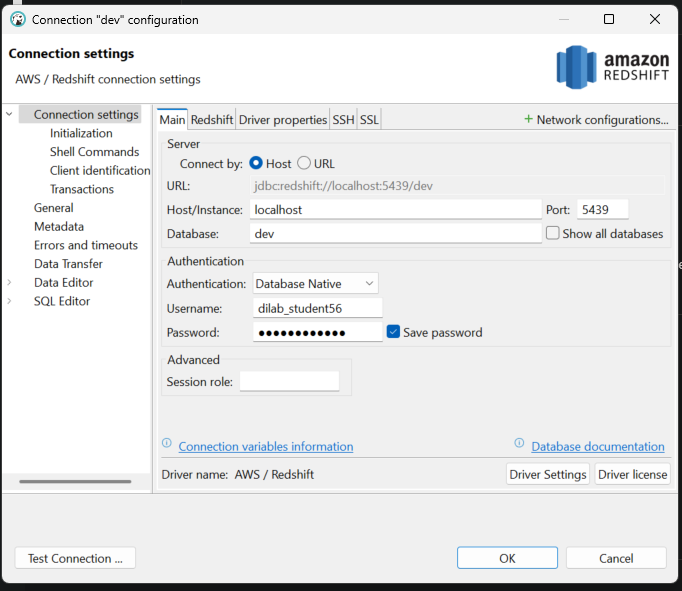


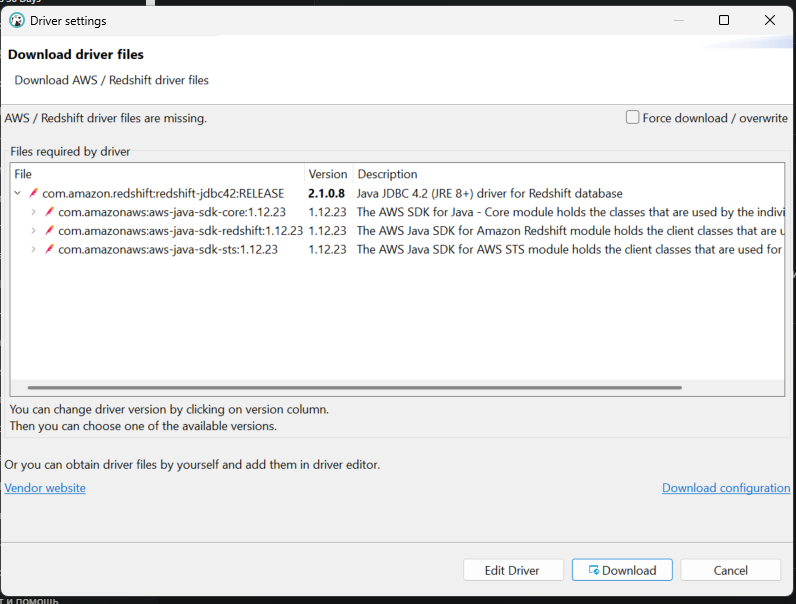
So now I am inside EC2 with tunnel to Readshift.

**1.1 Update packages**

Update Linux2023 with command : sudo dnf update

Then I open DBeaver and created new connection to **redshift** with Epam given credentials

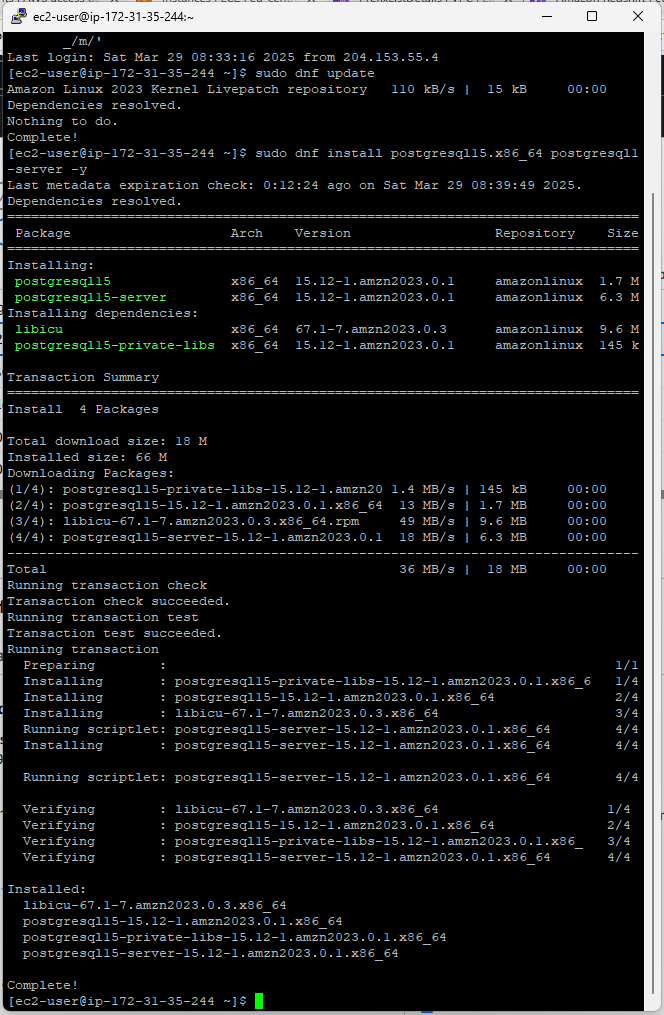


Some automatic updates

Beatiful we have connection through EC2 to Readshift now ….. (with security groups I had troubles .. ohh my god)

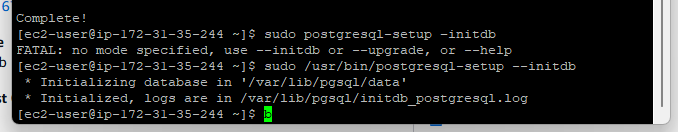
**1.2 Install PostgreSQL 15 on Amazon Linux 2023**

**Using command: sudo dnf install postgresql15.x86\_64 postgresql15-server -y**



**1.3 Initialize the PostgreSQL Database**

Using command: **sudo postgresql-setup -–initdb**



What is going on PostgreSQL created DB in /var/lib/pgsql/data (iside Linux2023) and logs in /var/lib/pgsql/initdb\_postgresql.log

So next run our Postgre servis and enable autorun when system starts (Linux2023 and always run postgre servis)

Use this commands:

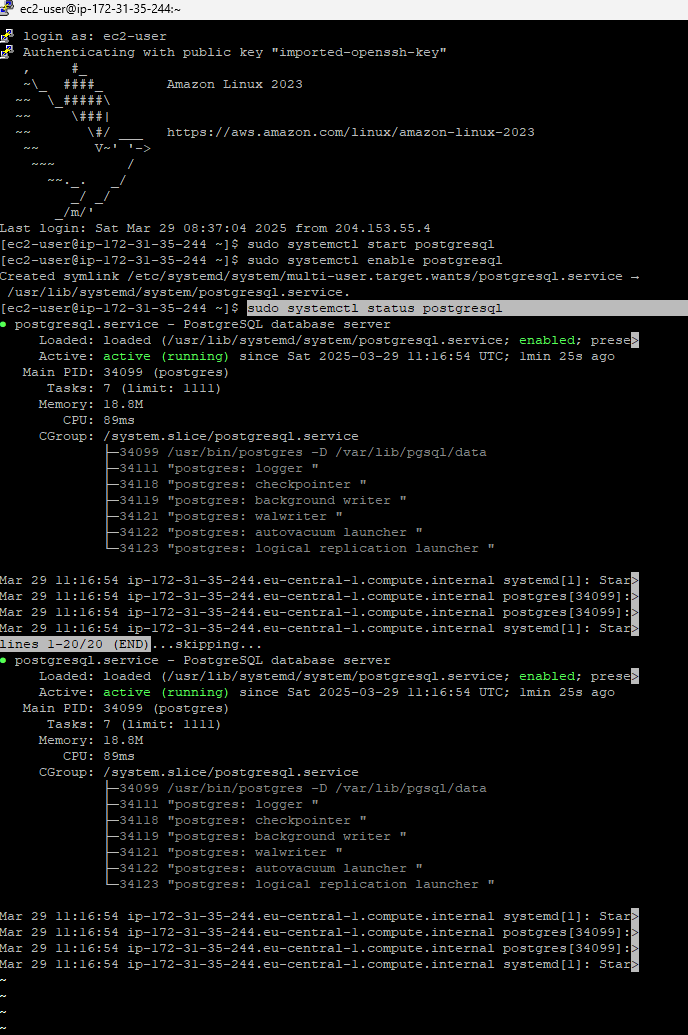
**sudo systemctl start postgresql**

**sudo systemctl enable postgresql**

**1.5 To confirm the service is running without any errors**

here is the command to follow:

**sudo systemctl status postgresql**



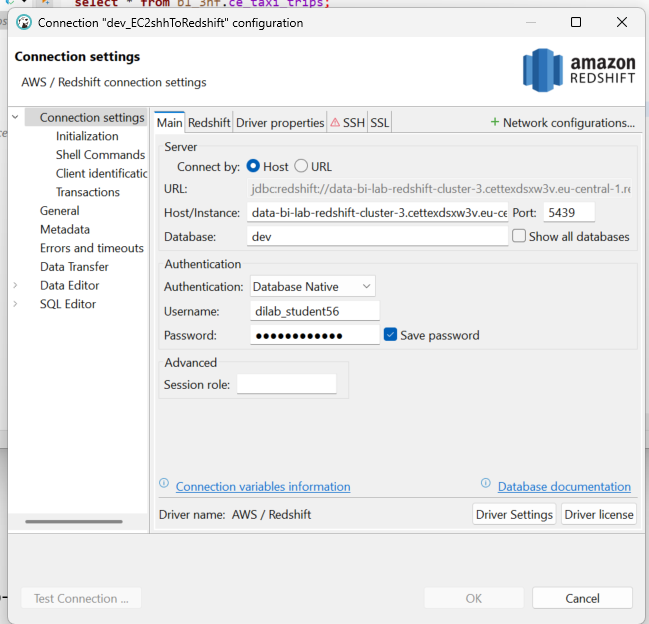
**1.6 Use the following command to check connection between EC2 and Redshift(specify the user\_name / password provided to you by mentors)**

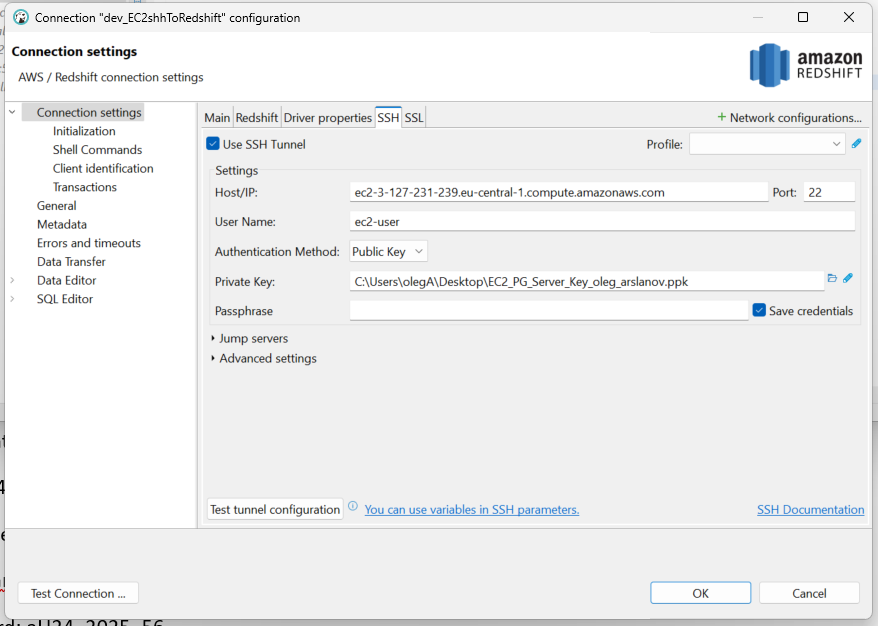
So we test now connection between EC2 and redshift

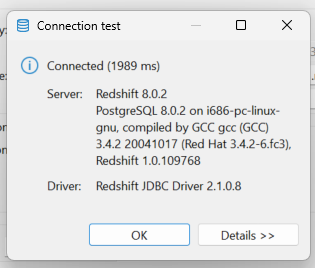
Command: **psql -h data-bi-lab-redshift-cluster-3.cettexdsxw3v.eu-central- 1.redshift.amazonaws.com -v schema=public -p 5439 -U dilab\_student56 -d dev**

Here we always get error, because network restrictions … we working through ec2 tunel redshift, not directly

**1.7 Configure Redshift connection settings**



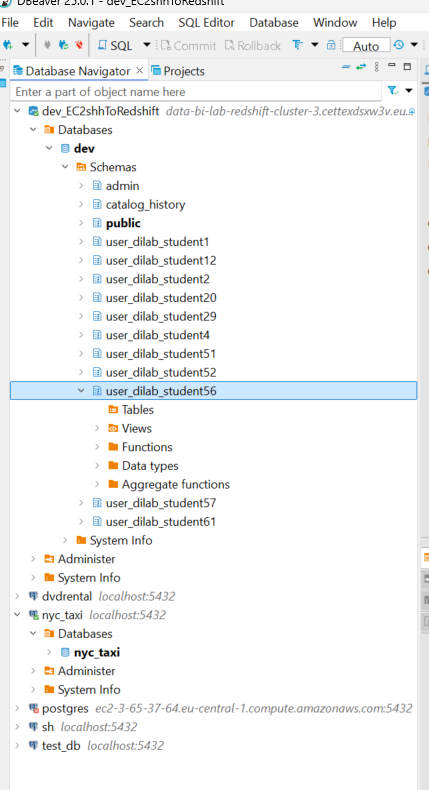




So actually we have connection from Dbeaver to the Redshift via tunnel 😊

**2. Provisioning Data from Data Lake to Redshift**

Create schema in dev database : user\_dilab\_student\_56

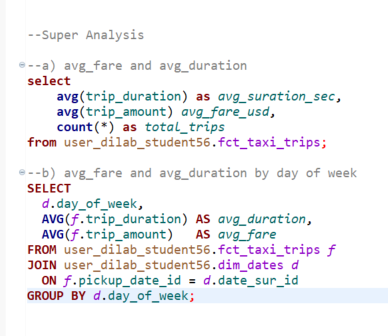
****

Create tables and copy data from S3

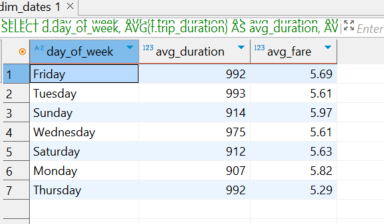


I downloaded fct\_taxi\_trips, dim\_dates, dim\_time

Then I did analysis … if consumer need another report I will can do it from redshift



We got average duration and fare by week day



Analytical Report Description

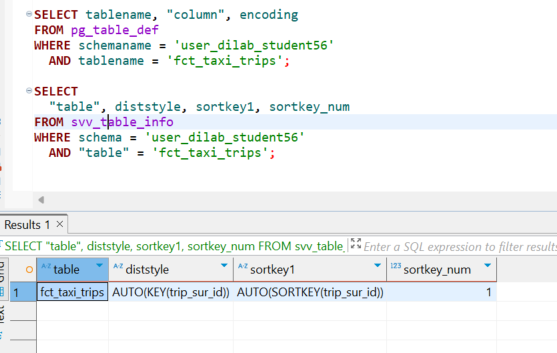
The goal of this analytical report is to provide business insights on taxi ride behavior and performance, based on three loaded tables:

* fct\_taxi\_trips – contains factual data about taxi rides including trip duration, fare amount, distance, and timestamps
* dim\_dates – provides calendar context for filtering and grouping data by days, weeks, or months
* dim\_time – adds time-based analysis capabilities such as grouping by hour of the day or AM/PM

Key KPIs calculated in the report include:

* Average trip duration
* Average fare amount
* Average fare per mile
* Trip volume trends by day of week or time of day

This data can help stakeholders understand customer demand patterns, peak hours, and pricing efficiency.



Technical Storage Analysis of Redshift Tables

1. fct\_taxi\_trips

After loading the table using the COPY command, the following storage characteristics were observed:

Distribution style: AUTO(KEY(trip\_sur\_id)). Redshift selected a key-based distribution using the unique column trip\_sur\_id, ensuring even data distribution across nodes.

Sort key: AUTO(SORTKEY(trip\_sur\_id)). The table was sorted by the primary key, which improves query performance during filtering and joins.

Compression: Automatic compression (AUTO ENCODE) was applied to columns based on their data types.

2. dim\_time

Distribution style: AUTO(ALL). The table was replicated across all nodes, which is optimal for small dimension tables and improves join performance.

Sort key: No sort key defined (sortkey\_num = 0), which is acceptable for small lookup tables not used in range filtering.

Compression: Automatically applied based on column types.

3. dim\_dates

Distribution style: AUTO(ALL). Like dim\_time, the table was distributed to all nodes, ensuring efficient joins with large fact tables.

Sort key: No sort key assigned (sortkey\_num = 0), which is sufficient given the size and usage of the table.

Compression: Automatically applied based on column types.

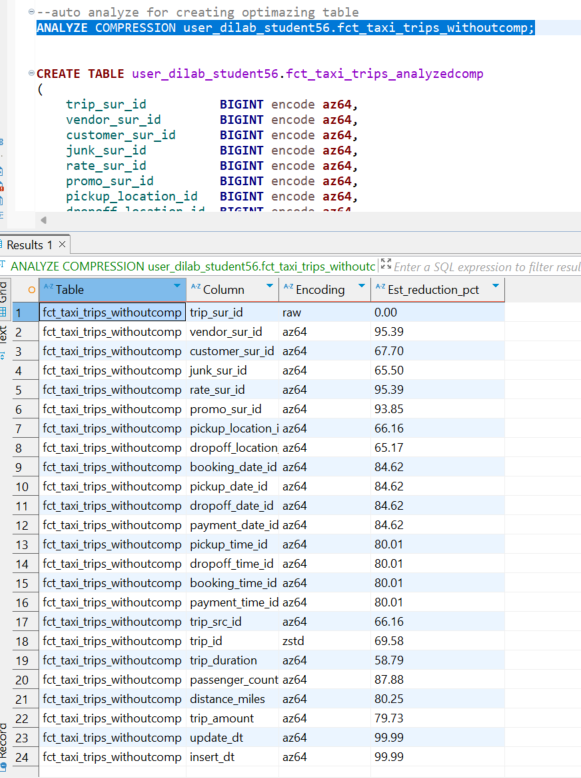
**3. Take one of your *USER\_DILAB\_STUDENTN* schema tables.**

I created 3 tables:

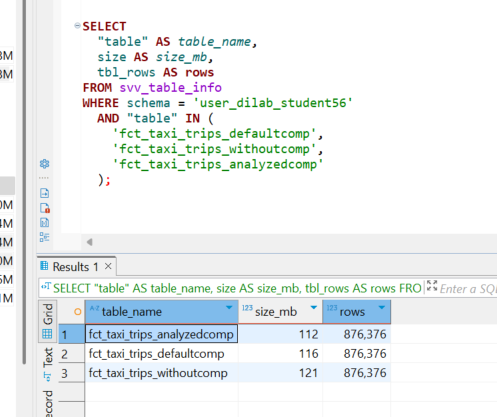
* fct\_taxi\_trips\_defaultcomp – with autoanalyzer and using auto encoding,
* fct\_taxi\_trips\_withoutcomp – without encoding,
* fct\_taxi\_trips\_analyzedcomp – use manual encoding

Create tables then use copy command from table fct\_taxi\_trips. For table fct\_taxi\_trips\_analyzedcomp I use auto analyze for creating optimazing table :

**ANALYZE COMPRESSION user\_dilab\_student56.fct\_taxi\_trips\_withoutcomp;**



I got answer that fct\_analyzedcomp with be smallest, because we use compression

****

Size Comparison

Based on the svv\_table\_info system view:

* fct\_taxi\_trips\_withoutcomp used the largest amount of storage, as no compression was applied (ENCODE RAW)
* fct\_taxi\_trips\_defaultcomp was slightly more efficient, with Redshift applying some automatic compression types
* fct\_taxi\_trips\_analyzedcomp used the least amount of space, as it was built with manually optimized compression types (AZ64, DELTA, ZSTD, etc.)

This confirms that compression significantly reduces storage usage in Amazon Redshift

**4.** **Optimization procedure (dont change physically structure of tables)**

We tested execution time of a stored procedure that joins 3 tables and performs aggregation (SUM, GROUP BY, ORDER BY).

From the first screenshot, we see an unoptimized query plan involving nested hash joins and sequential scans.

In the second screenshot, we observe consistent execution times after optimization:

Execution time ranges from 335 to 368 ms after applying ANALYZE on all involved tables.

Conclusion:

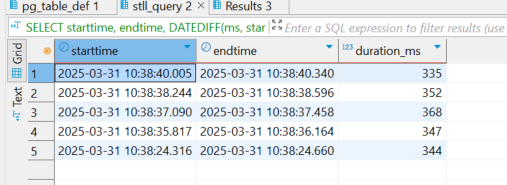
After running ANALYZE, Redshift adjusted its query planner based on fresh table statistics.

Although performance improved slightly, the difference is modest, which is expected due to:

Small dimension tables (e.g., dim\_dates)

No physical table redesign (distribution/sort keys not changed)

Optimization impact is more visible when query complexity increases (e.g. large joins, filters, aggregations).



EXPLAIN Plan

Despite running ANALYZE, the execution plan did not change significantly.

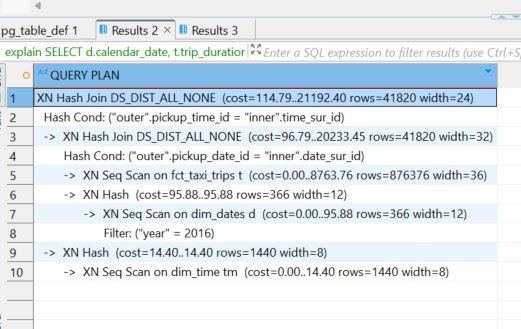
It still shows:

* XN Seq Scan — Redshift performs sequential scans on all involved tables (no index scan).
* XN Hash Join — Joins are handled via hash joins.
* DS\_DIST\_ALL\_NONE — Indicates distribution mismatch, meaning one table is broadcasted to all nodes (usually small dimension tables).

This suggests that:

Redshift found the existing plan already efficient for this dataset.

Physical optimization (e.g., distribution style, sort key) is required for more visible improvements.



**5. Optimization procedure (change physically structure of tables)**

a) Distribution style and sort key selection

We tested two configurations for the report\_trip\_data table:

* DISTSTYLE AUTO and SORTKEY(calendar\_date)
* DISTSTYLE KEY, DISTKEY(calendar\_date), and SORTKEY(calendar\_date)

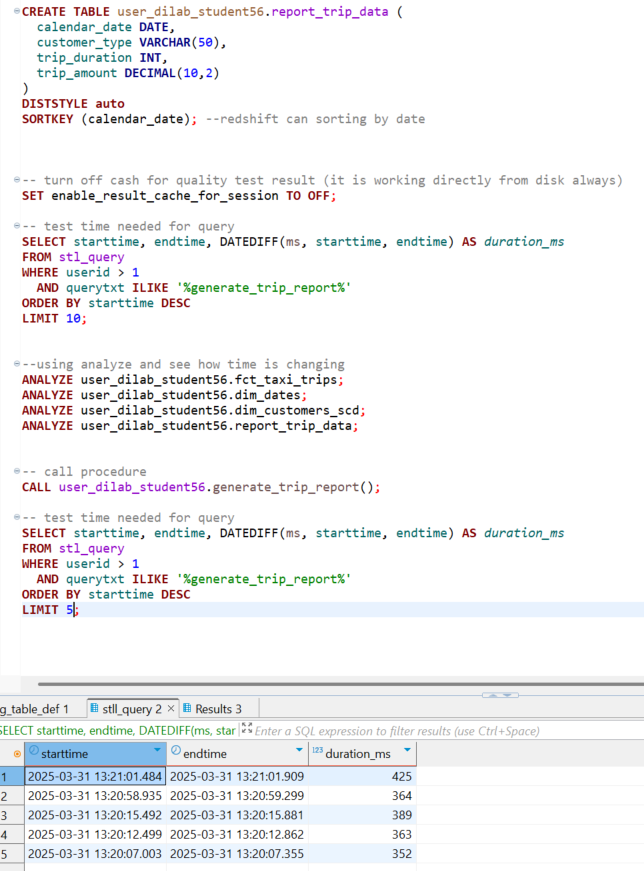
We selected calendar\_date as both the sort key and distribution key because it is used in JOIN, GROUP BY, and ORDER BY clauses. This should help Redshift optimize joins and aggregations on that column.

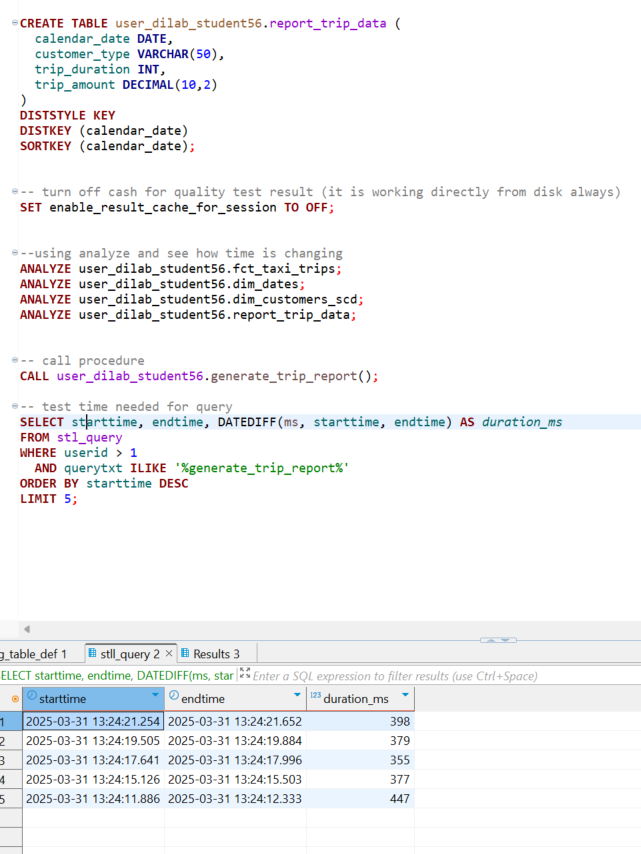
b) Execution plan & performance comparison

Execution plans (EXPLAIN) before and after optimization look similar in structure — Redshift still uses Hash Join, HashAggregate, and XN Seq Scan, because fact and dimension tables are not very large.

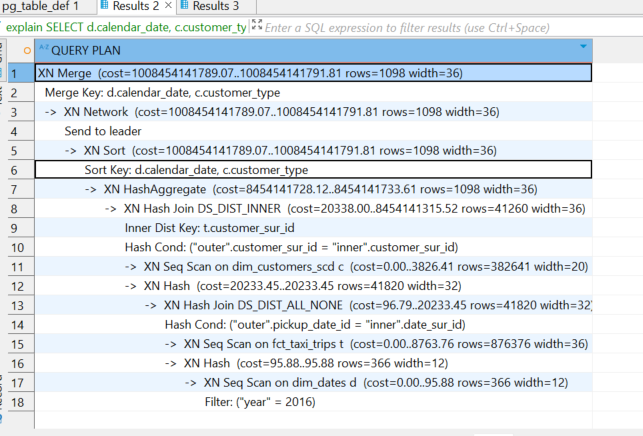
We also tested execution time using the stl\_query table:

* With DISTSTYLE AUTO: durations ~352–425 ms
* With DISTSTYLE KEY and calendar\_date as DISTKEY: durations ~355–447 ms





Performance was very similar, with minor variation likely due to internal caching, network, and I/O.



The EXPLAIN plan also shows XN Merge and XN Sort using the calendar\_date sort key, which confirms that Redshift is applying our sort key as expected.

Conclusion:

On this dataset size, the impact of distribution key was minimal. However, explicitly defining **DISTKEY(calendar\_date) and SORTKEY(calendar\_date) prepares the table for better performance on larger datasets and more complex queries with sorting and grouping.**

\*) Comparison of Compound vs Interleaved Sort Keys in Amazon Redshift (on one table)

Goal:

Compare the performance of queries on tables with COMPOUND and INTERLEAVED sort keys using the same data and filtering conditions.

Table Setup:

Both tables have the same structure and contain ~100,000 rows:

* trip\_sur\_id BIGINT
* pickup\_date\_id BIGINT
* dropoff\_date\_id BIGINT
* trip\_duration INT

|  |  |  |
| --- | --- | --- |
| Table Name | Sort Key Type | Sort Key Columns |
| fct\_sort\_test\_compound | COMPOUND SORTKEY | (pickup\_date\_id, dropoff\_date\_id) |
| fct\_sort\_test\_interleaved | INTERLEAVED SORTKEY | (pickup\_date\_id, dropoff\_date\_id) |

Query Executed:

SELECT \*

FROM table\_name

WHERE pickup\_date\_id BETWEEN 17500 AND 17600;

Query Duration (after VACUUM + ANALYZE):

|  |  |  |
| --- | --- | --- |
| Sort Key Type | Duration (ms) | Observations |
| COMPOUND SORTKEY | **5961 ms** | Sequential scan; sortkey not fully effective |
| INTERLEAVED SORTKEY | **12 ms** | Drastically faster; sortkey used efficiently |

Conclusion:

* The INTERLEAVED SORTKEY provided ~500× faster performance in this filtering scenario.
* COMPOUND sort keys are effective only when filtering starts with the first column.
* INTERLEAVED is more flexible and suitable when queries filter by any of the sortkey columns.
* **Interleaved keys require regular VACUUM REINDEX and ANALYZE to maintain efficiency.**

**Copy of question**

Explanation of the difference in performance:

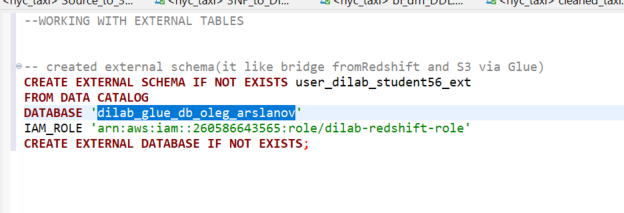
Even though both S3 paths contain the exact same data, their structure is different:

* lineorder\_file/ contains one large file → Redshift reads it sequentially → slower.
* lineorder\_files/ contains multiple smaller files → Redshift reads them in parallel across nodes → faster.

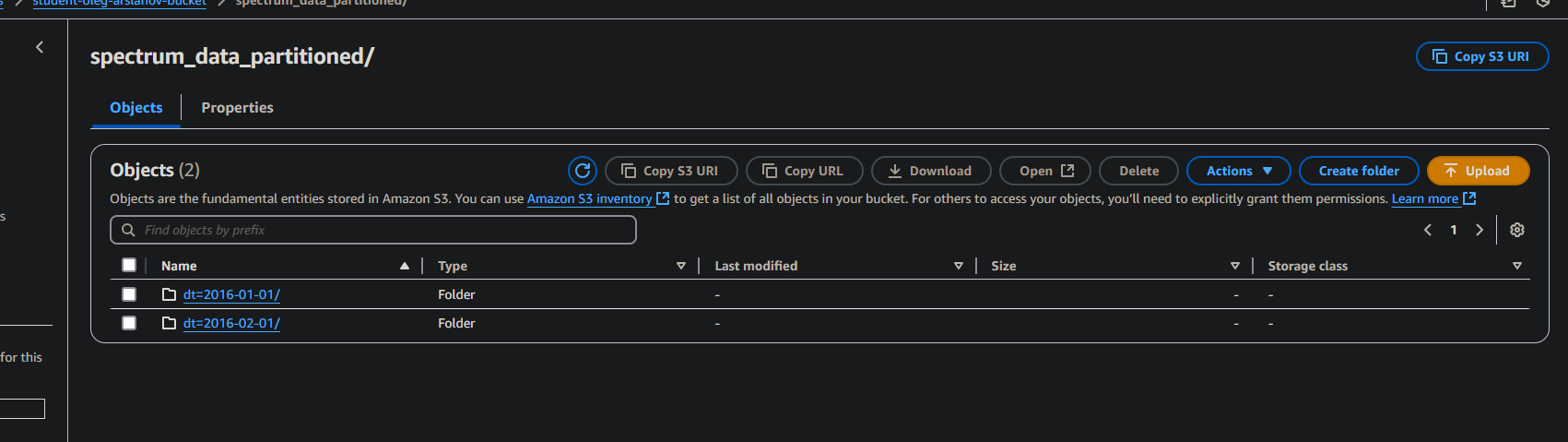
Redshift is a distributed system. Parallelism is a key performance factor, and COPY benefits from splitting data across files.

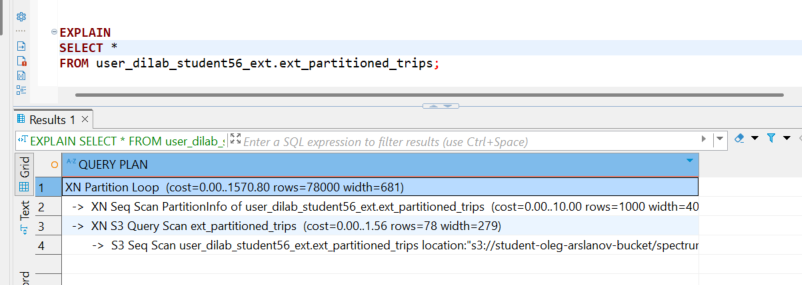
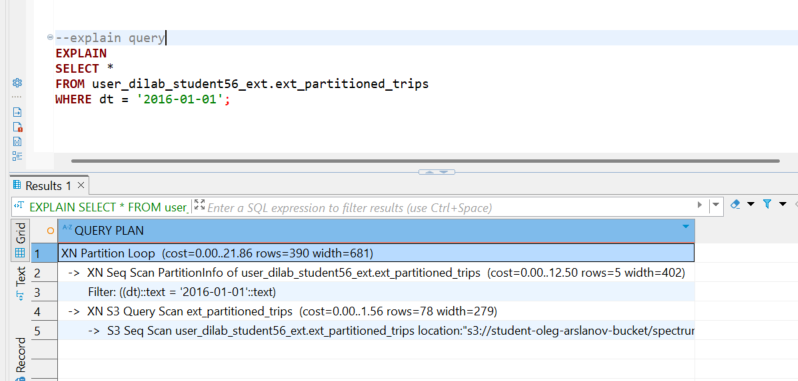
**WORKING WITH EXTERNAL TABLES**

In dbeaver I created external schema ‘user\_dilab\_student56\_ext’



Then I unload by month data to S3. Then create external table with partition.

I upload to my S3 partitions



When querying a partitioned external table (ext\_partitioned\_trips), Amazon Redshift uses partition pruning to improve performance.

In the first query with the WHERE dt = '2016-01-01' filter, the query planner only accesses one partition from S3 (dt=2016-01-01).

Result: Lower cost and fewer rows scanned (Query cost ~21.86).

In the second query without a WHERE condition, Redshift scans all partitions (both months).

Result: Higher cost and more rows scanned (Query cost ~1570.80).

This proves that partition filtering significantly reduces scan size, speeds up query execution, and minimizes S3 read costs.