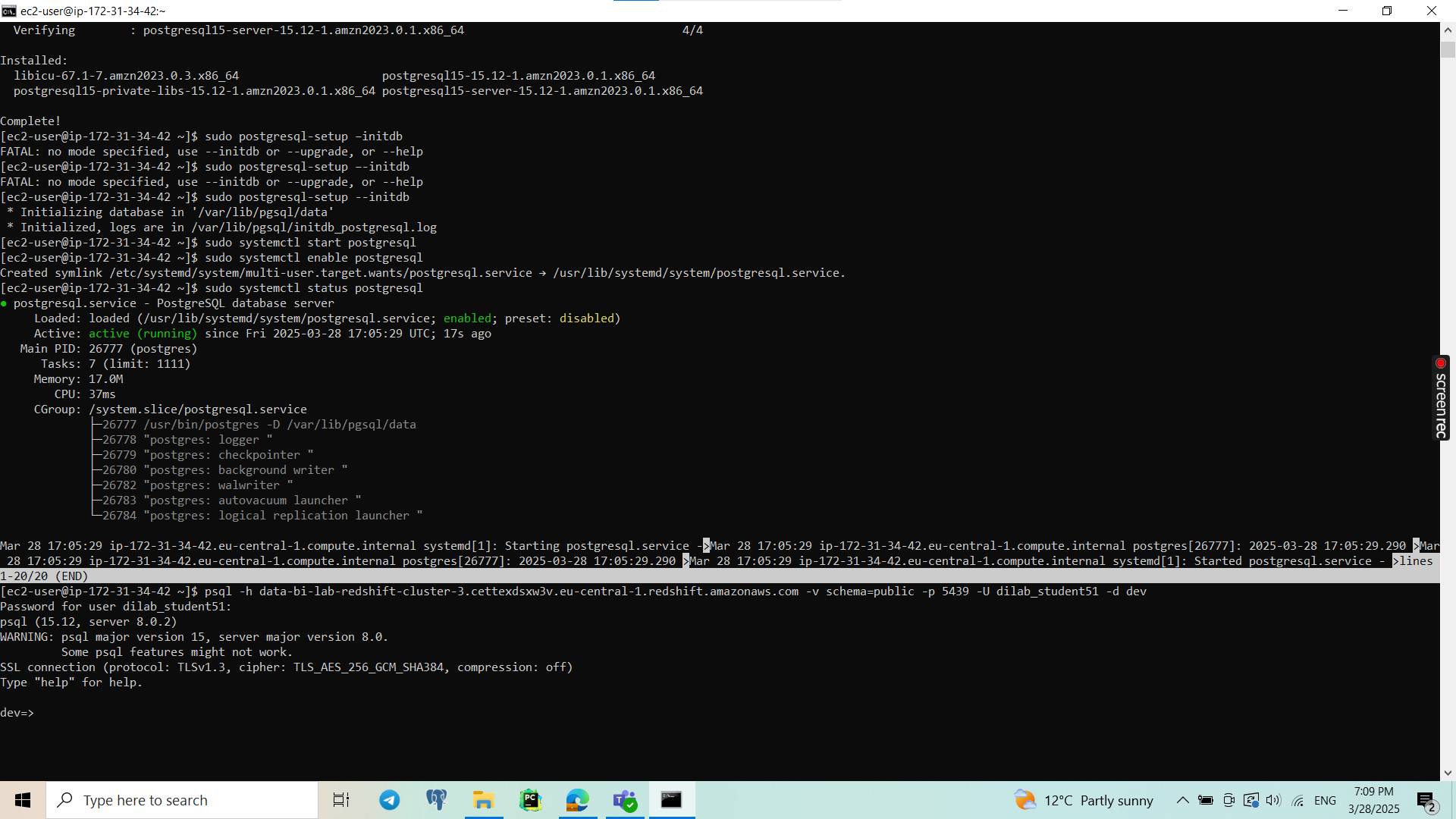
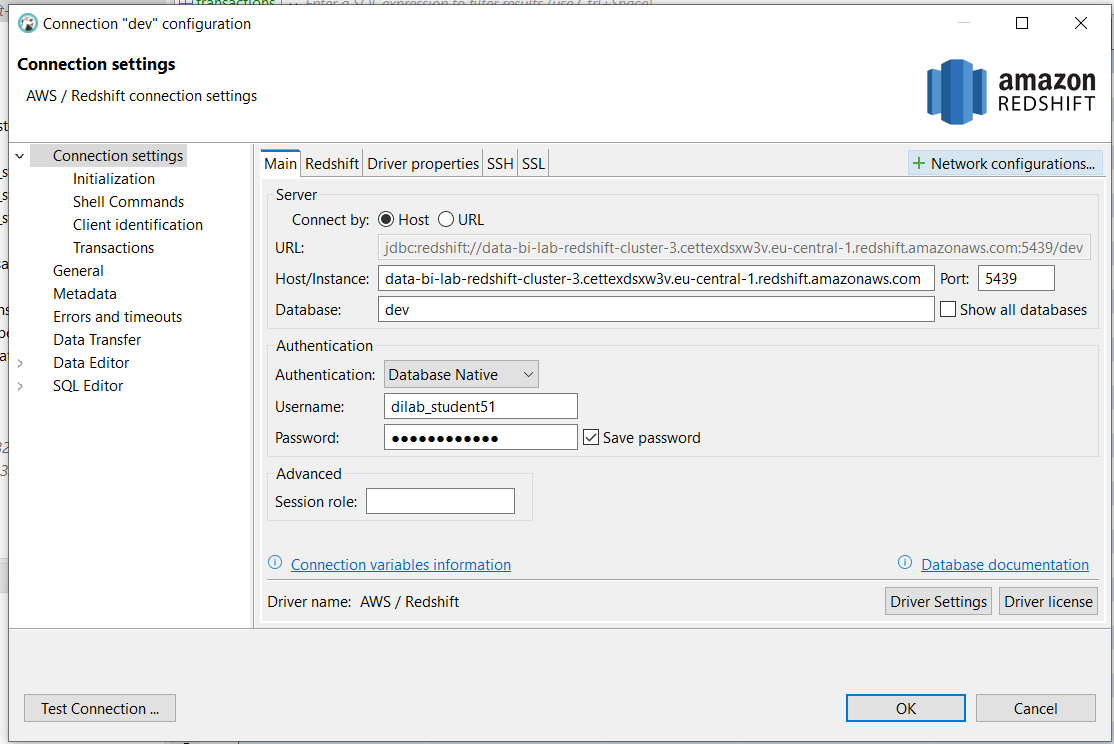
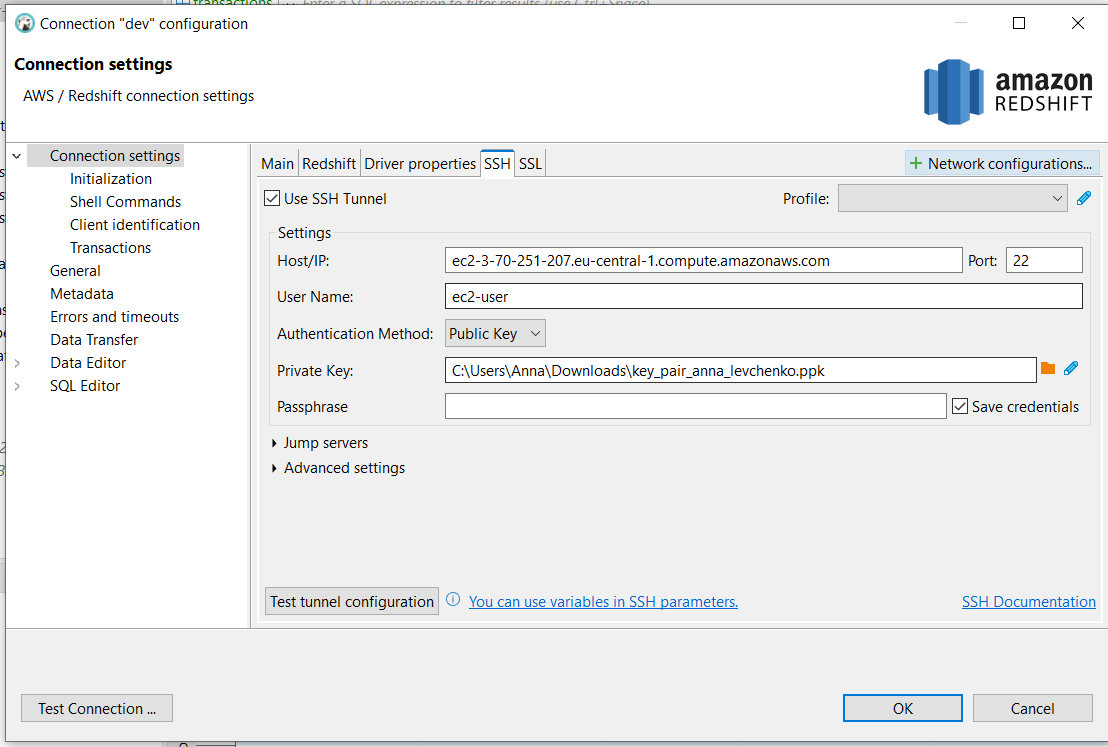
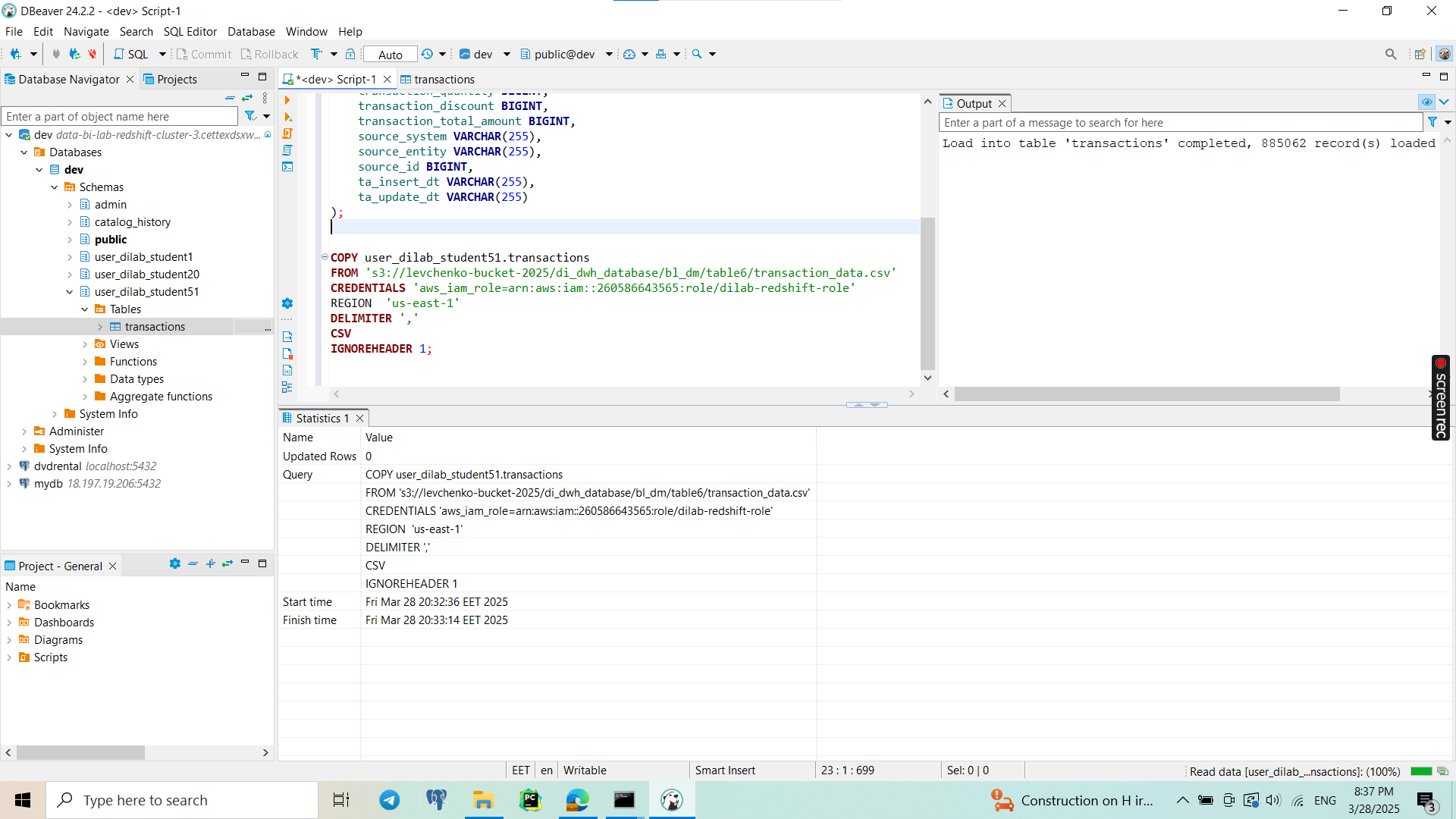
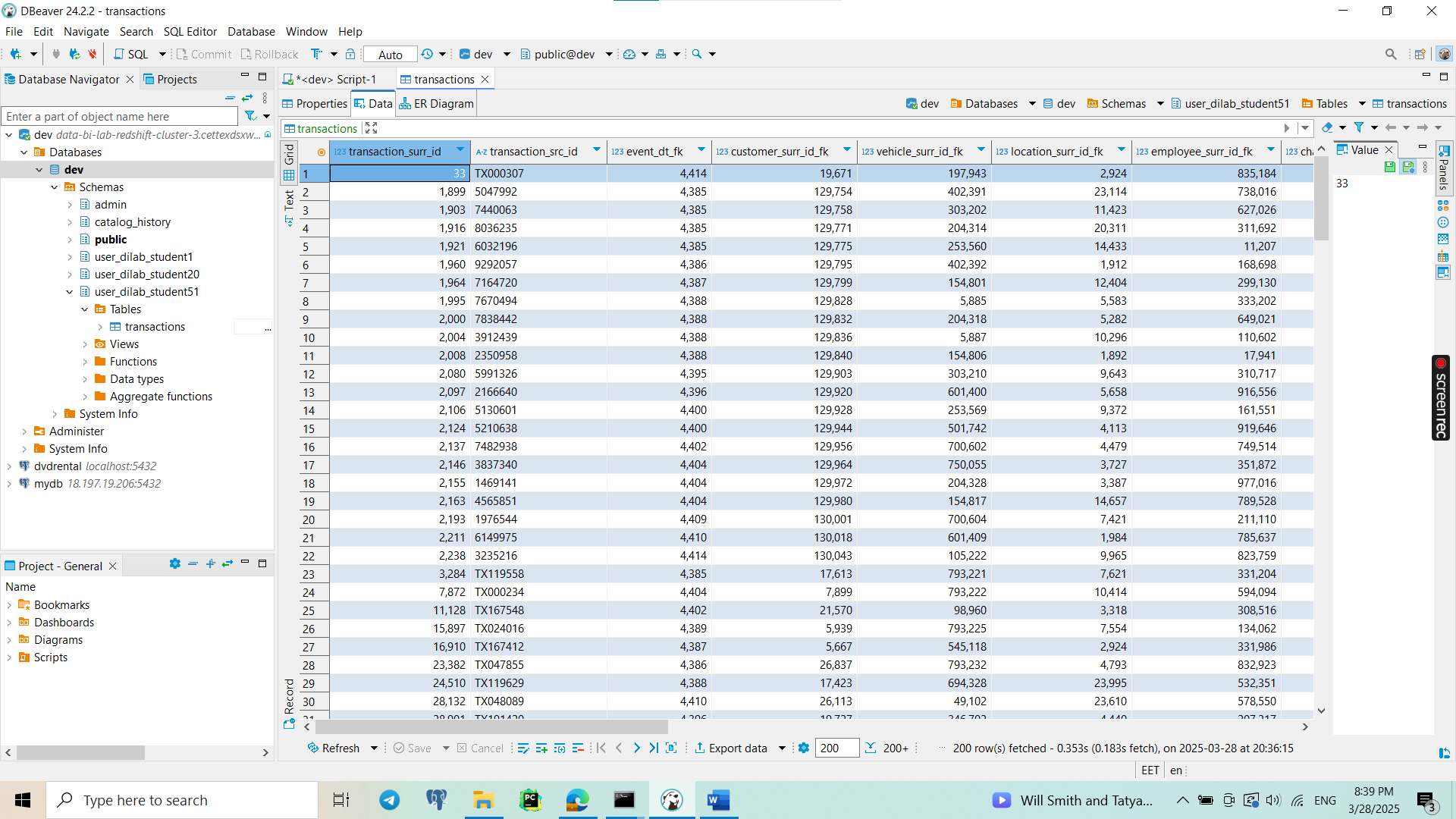
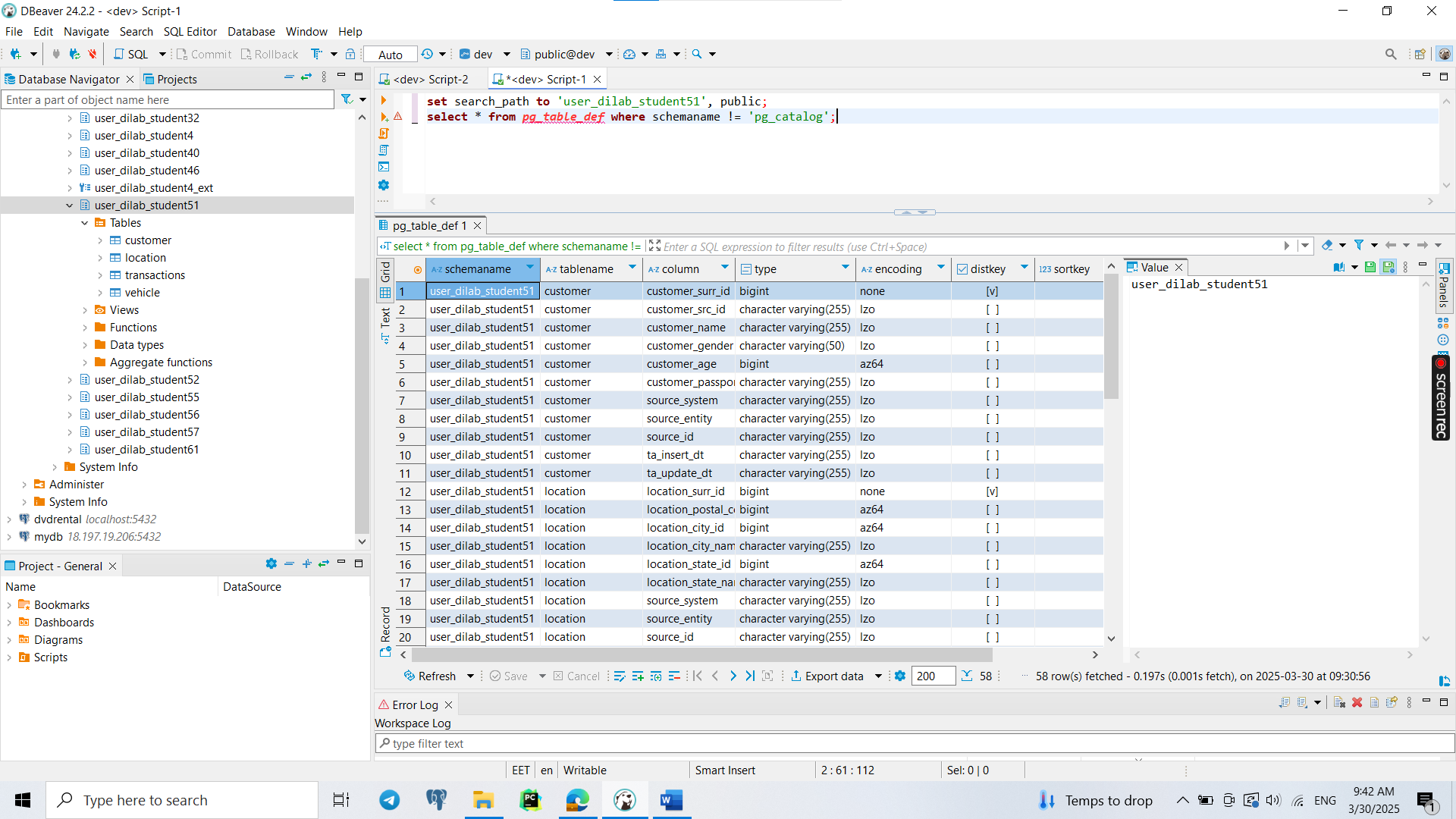
1.1-1.6

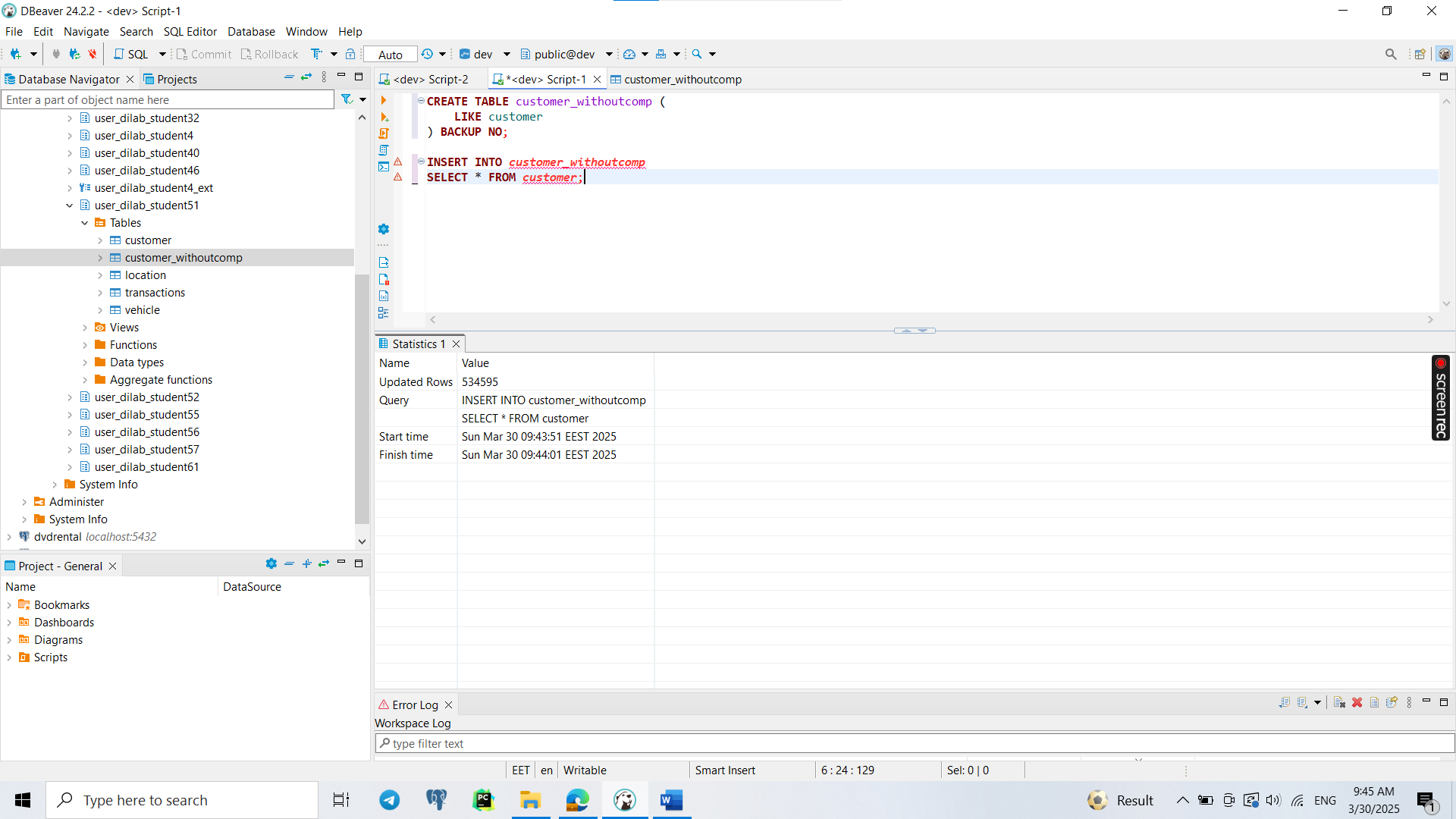
1.7 

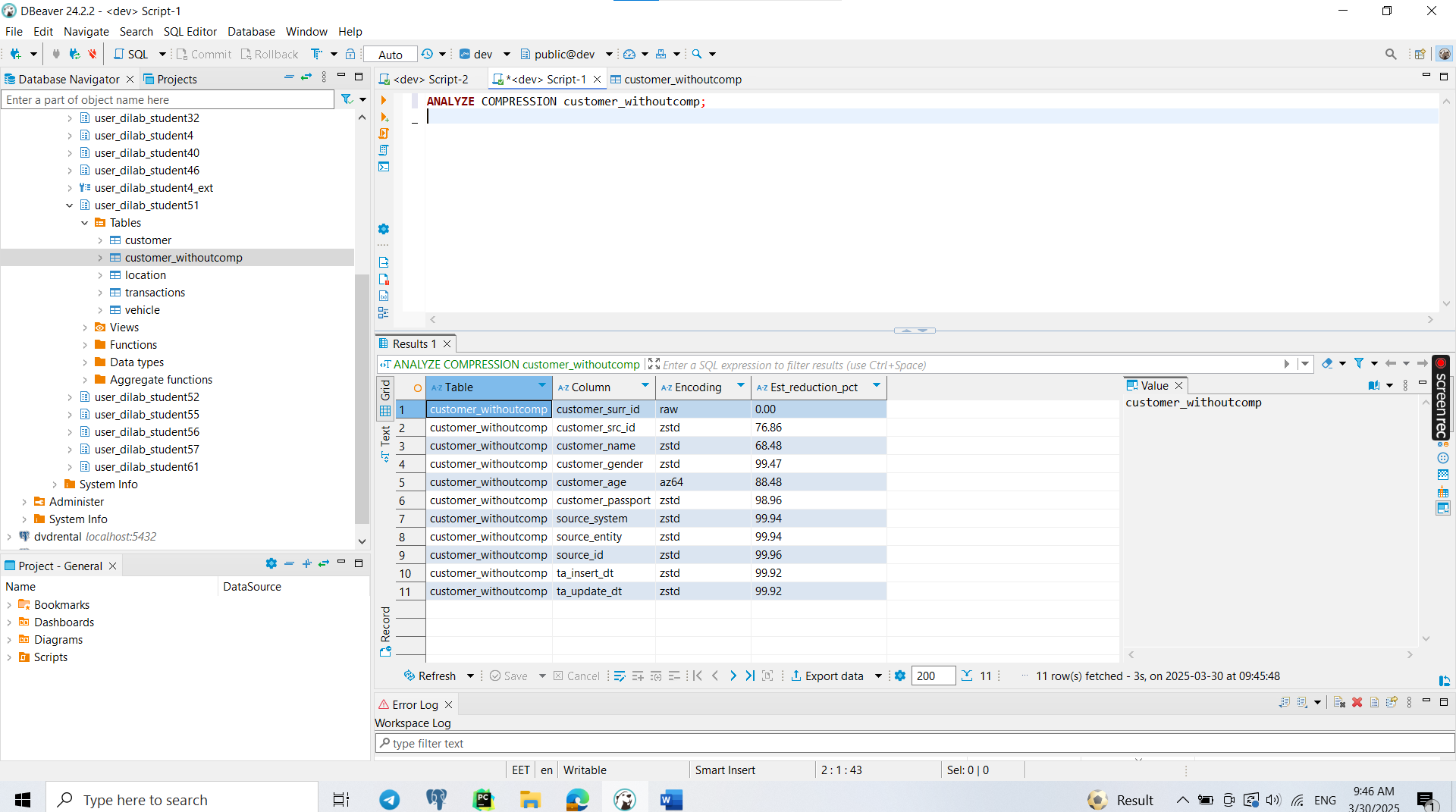


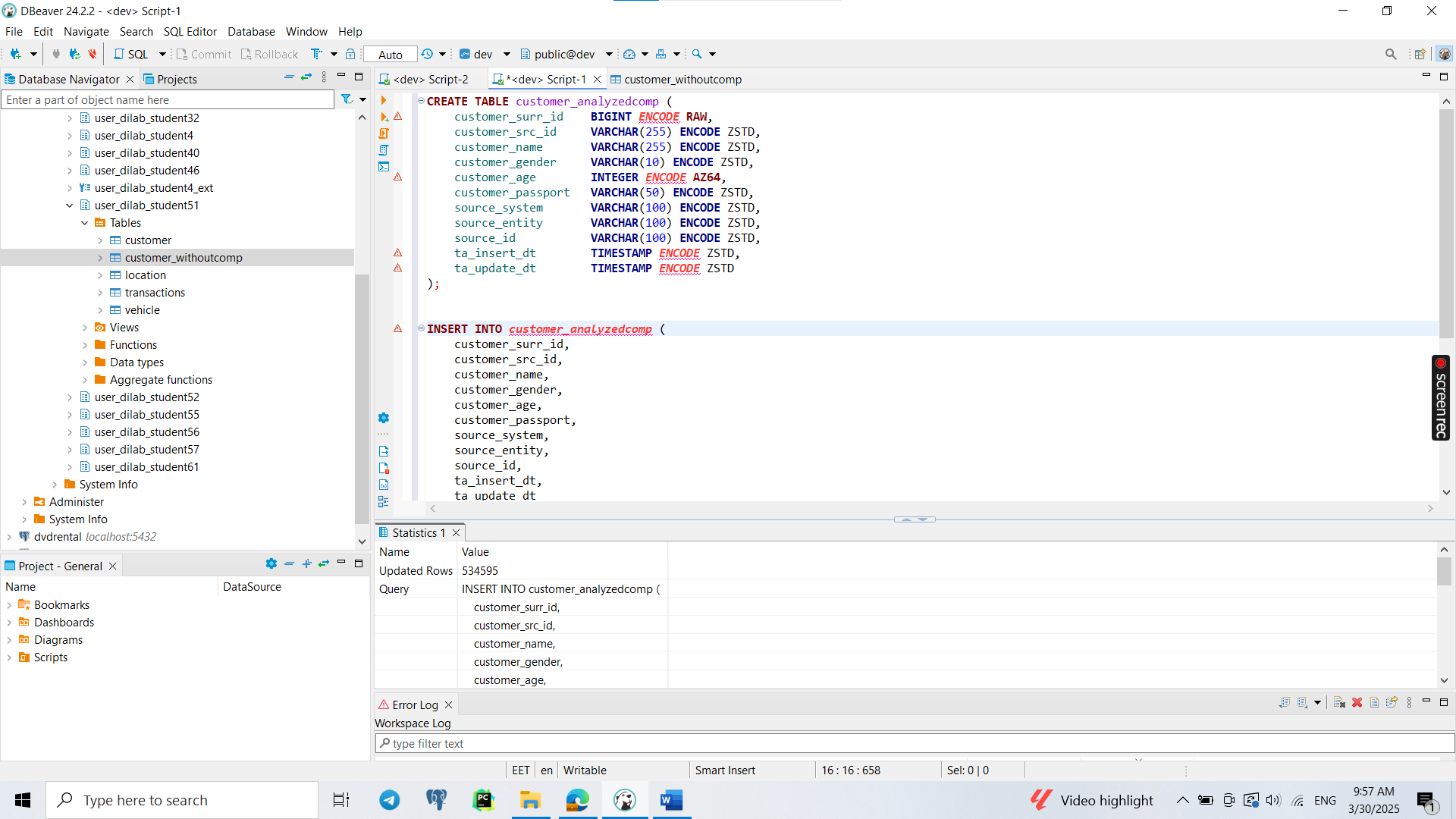
2. 

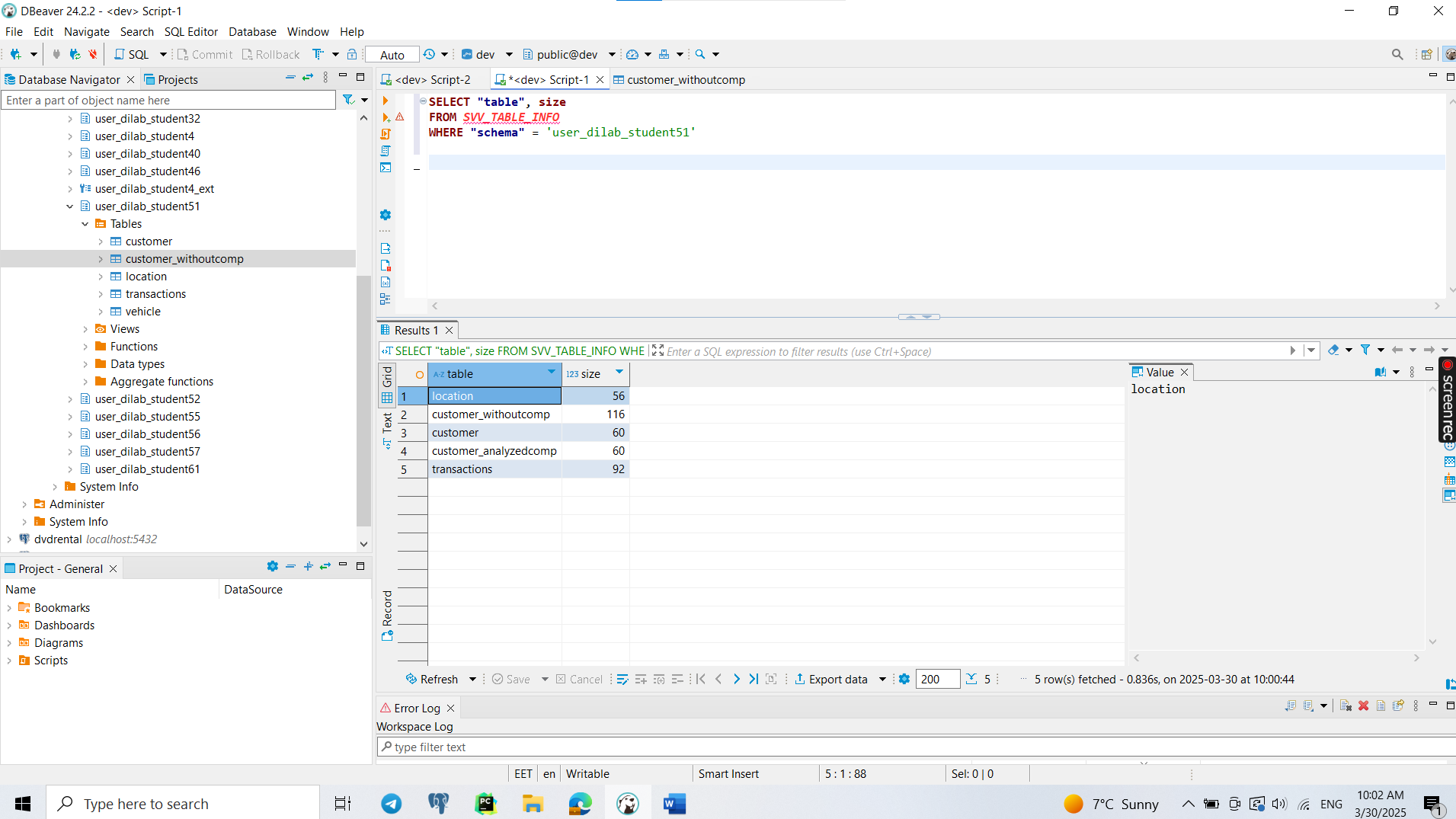


3. 



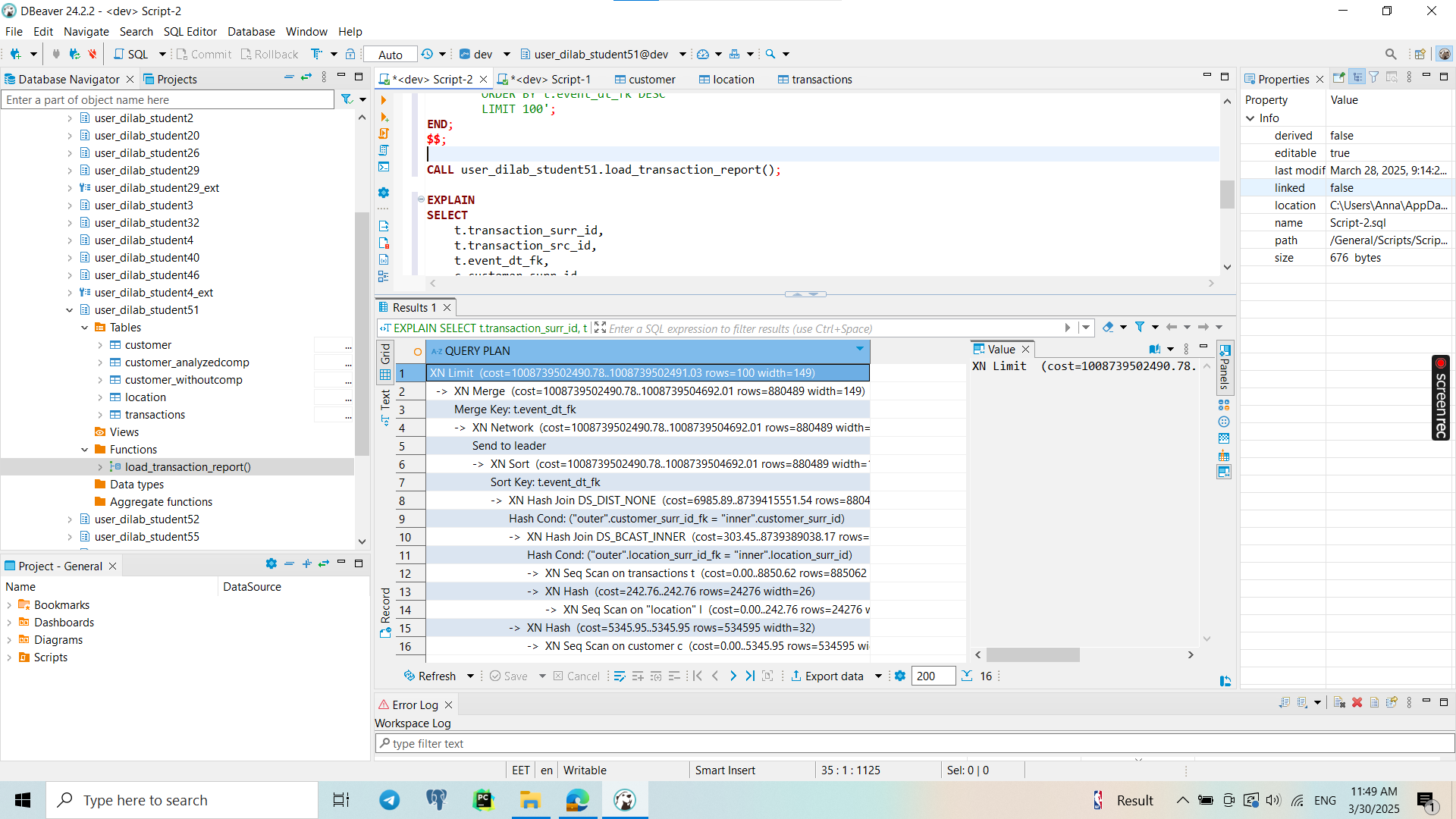






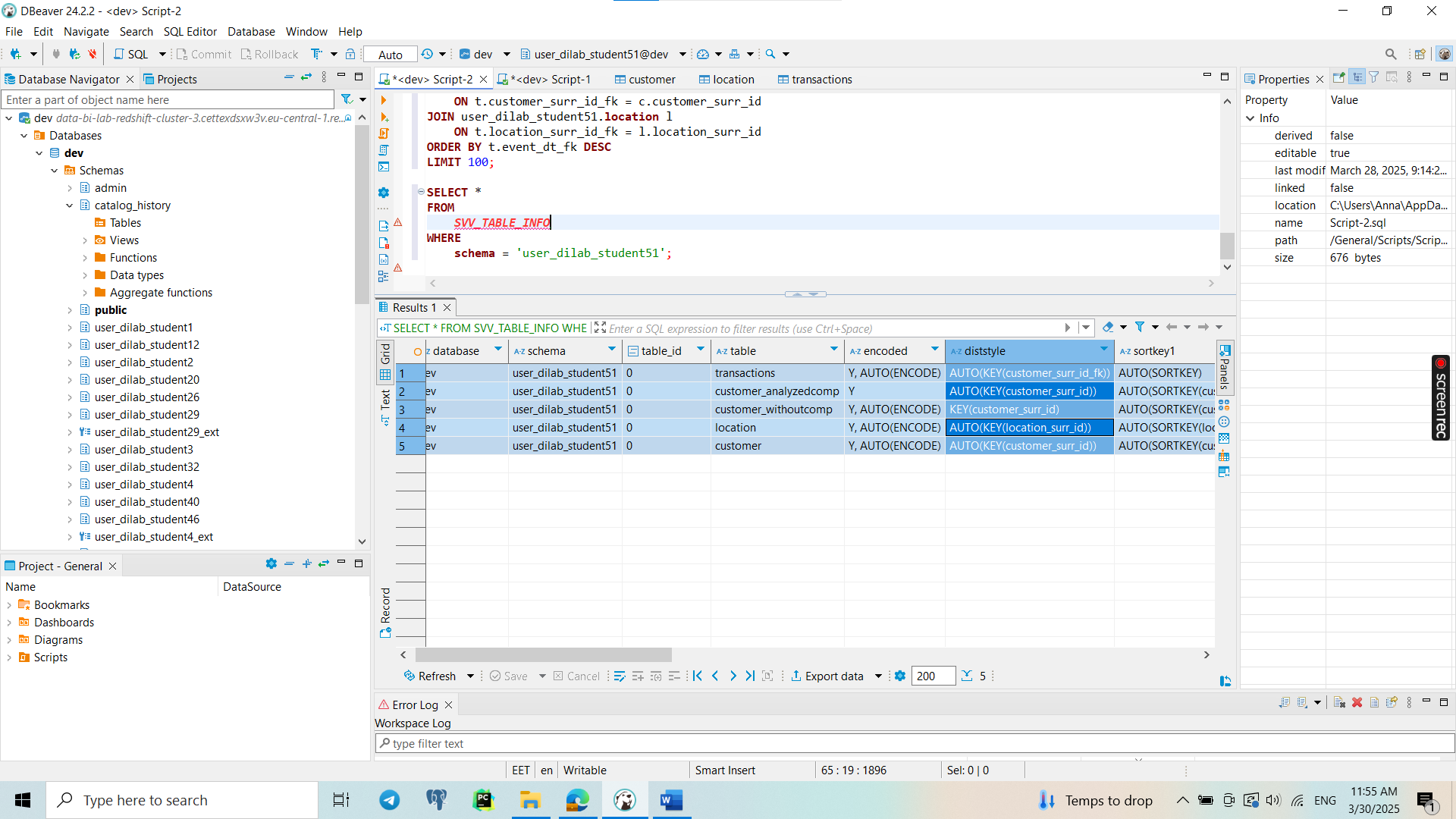
The analysis of table sizes shows that customer\_withoutcomp is the largest because it lacks compression, leading to excessive storage usage. The customer\_analyzedcomp table achieves the best compression, significantly reducing storage size while maintaining performance. And my default customer is the same, because I think Redshift automatically tries to compress in the best way.

4.



Why the First Execution Is Not Reliable?

* First Execution is Slow Because:
  + Redshift compiles the query execution plan for the first time.
  + Data is fetched from disk to memory (instead of cache).
  + Sorting and Joins might cause temporary storage usage.
* Subsequent Executions are Faster Because:
  + Redshift optimizes execution with cached data.
  + The query execution plan is already prepared.



Distribution Style: All tables use AUTO distribution style. Redshift automatically determines the best distribution method, which could be based on the data size and query patterns. For larger tables, this typically results in choosing a distribution key that optimizes join performance. For smaller tables, EVEN or ALL might be used, but Redshift likely chooses KEY distribution when the schema has appropriate keys defined.

Sort Key: Most tables use AUTO(SORTKEY), which means Redshift automatically chooses a column to optimize query performance based on query patterns. In some tables, the sort key explicitly includes specific columns, such as customer\_surr\_id and location\_surr\_id.

5.

Distribution Style: The distribution style chosen for the tables is important to optimize how the data is distributed across Redshift nodes. Since the joins between transactions, customer, and location are expected to happen frequently, choosing the correct distribution key ensures that the data is collocated, minimizing the need for data shuffling during query execution.

Based on the columns involved in the JOIN conditions:

* transactions.customer\_surr\_id\_fk = customer.customer\_surr\_id: The customer\_surr\_id in both transactions and customer should be the distribution key for both of those tables.
* transactions.location\_surr\_id\_fk = location.location\_surr\_id: Similarly, for the location table, location\_surr\_id should be the distribution key.

By distributing the tables on these columns, Redshift will collocate the related rows in memory, which reduces the need for data shuffling between compute nodes.

For the distribution style:

* KEY distribution was chosen since it works best when tables are joined on specific keys. It ensures that rows with the same key values are stored on the same node, which reduces network traffic during joins.
* AUTO distribution is used in some cases when the data is small, or the query patterns are not well defined. For optimal performance, however, KEY distribution should be used.

Sort Keys: The sort key defines the order of data in each table. Since the query is filtering by the column event\_dt\_fk in the transactions table, it is useful to create a compound sort key on that column to make range scans more efficient.

* For the transactions table, a compound sort key on event\_dt\_fk is beneficial for efficient filtering and ordering by date.
* For the customer and location tables, since the query joins on their respective IDs (customer\_surr\_id and location\_surr\_id), these columns should be set as sort keys as well. This allows for efficient scans on those columns during join operations.

By setting the sort key, Redshift can perform more efficient range-based scans when filtering or sorting on the event\_dt\_fk, customer\_surr\_id, or location\_surr\_id columns, significantly improving performance.

#### Before Optimization:

The **initial execution plan** would likely show the following characteristics:

* **Data shuffling** due to the lack of distribution on the join columns. Since the tables have AUTO distribution, Redshift would need to move data between nodes to perform the joins, which can increase the execution time.
* **Disk I/O**: Without appropriate sort keys, the system may perform full scans or suboptimal scans when filtering or sorting by event\_dt\_fk, customer\_surr\_id, or location\_surr\_id.
* **High cost**: You may see higher costs in the execution plan due to unnecessary data shuffling and inefficient scans.

Example of the execution plan before optimization might show:

* **Large data movement** (Shuffle Join operations) across the compute nodes.
* **Full table scans** or **inefficient scans** on the transactions, customer, or location tables.

#### After Optimization:

* **Efficient Join Operations**: Since all the tables are distributed by their **join keys** (customer\_surr\_id and location\_surr\_id), the query will benefit from **colocated joins**, significantly reducing data shuffling.
* **Efficient Sort Scans**: The **sort key** on event\_dt\_fk, customer\_surr\_id, and location\_surr\_id will allow the system to scan the data more efficiently, particularly when filtering or sorting by event\_dt\_fk.
* **Reduced Cost**: The execution plan should now show **fewer data shuffling operations** and a **lower overall cost** for the query.

Conclusion:

* Optimization Impact: With the correct distribution style (KEY) and sort keys, you should see a significant reduction in query execution time, as fewer data shuffling operations will be required and the scans will be more efficient.
* Execution Plan: After optimization, expect more efficient joins and faster scans due to colocated data and optimized disk layout.

6.

**CREATE** **TABLE** **IF** **NOT** **EXISTS** user\_dilab\_student51.transaction\_report (

transaction\_surr\_id **BIGINT**,

transaction\_src\_id **VARCHAR**(255),

event\_dt\_fk **BIGINT**,

customer\_surr\_id **BIGINT**,

customer\_name **VARCHAR**(255),

customer\_gender **VARCHAR**(50),

customer\_age **BIGINT**,

location\_surr\_id **BIGINT**,

location\_city\_name **VARCHAR**(255),

location\_state\_name **VARCHAR**(255),

transaction\_payment\_method\_name **VARCHAR**(255),

transaction\_amount **BIGINT**,

transaction\_discount **BIGINT**,

transaction\_total\_amount **BIGINT**,

ta\_insert\_dt **VARCHAR**(255)

);

**CREATE** **OR** **REPLACE** **PROCEDURE** user\_dilab\_student51.load\_transaction\_report()

LANGUAGE plpgsql

**AS** **$$**

**BEGIN**

-- Disable result caching to ensure accurate timing

**SET** enable\_result\_cache\_for\_session **TO** **OFF**;

-- Insert the main query results into a report table

**INSERT** **INTO** user\_dilab\_student51.transaction\_report (

transaction\_surr\_id,

transaction\_src\_id,

event\_dt\_fk,

customer\_surr\_id,

customer\_name,

customer\_gender,

customer\_age,

location\_surr\_id,

location\_city\_name,

location\_state\_name,

transaction\_payment\_method\_name,

transaction\_amount,

transaction\_discount,

transaction\_total\_amount,

ta\_insert\_dt

)

**SELECT**

t.transaction\_surr\_id,

t.transaction\_src\_id,

t.event\_dt\_fk,

c.customer\_surr\_id,

c.customer\_name,

c.customer\_gender,

c.customer\_age,

l.location\_surr\_id,

l.location\_city\_name,

l.location\_state\_name,

t.transaction\_payment\_method\_name,

t.transaction\_amount,

t.transaction\_discount,

t.transaction\_total\_amount,

t.ta\_insert\_dt

**FROM** user\_dilab\_student51.transactions t

**JOIN** user\_dilab\_student51.customer c

**ON** t.customer\_surr\_id\_fk = c.customer\_surr\_id

**JOIN** user\_dilab\_student51.**location** l

**ON** t.location\_surr\_id\_fk = l.location\_surr\_id

**ORDER** **BY** t.event\_dt\_fk **DESC**

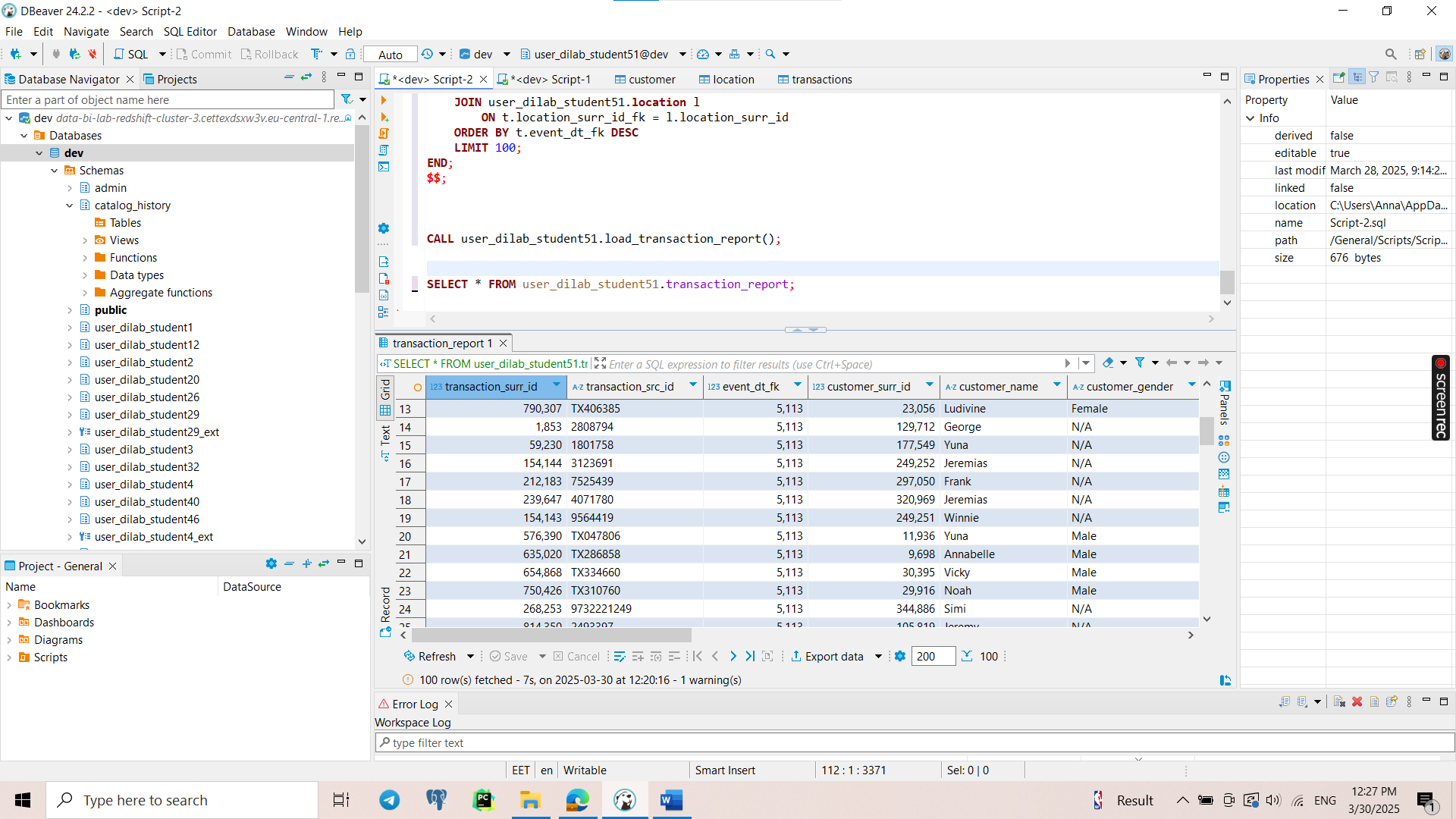
**LIMIT** 100;

**END**;

**$$**;

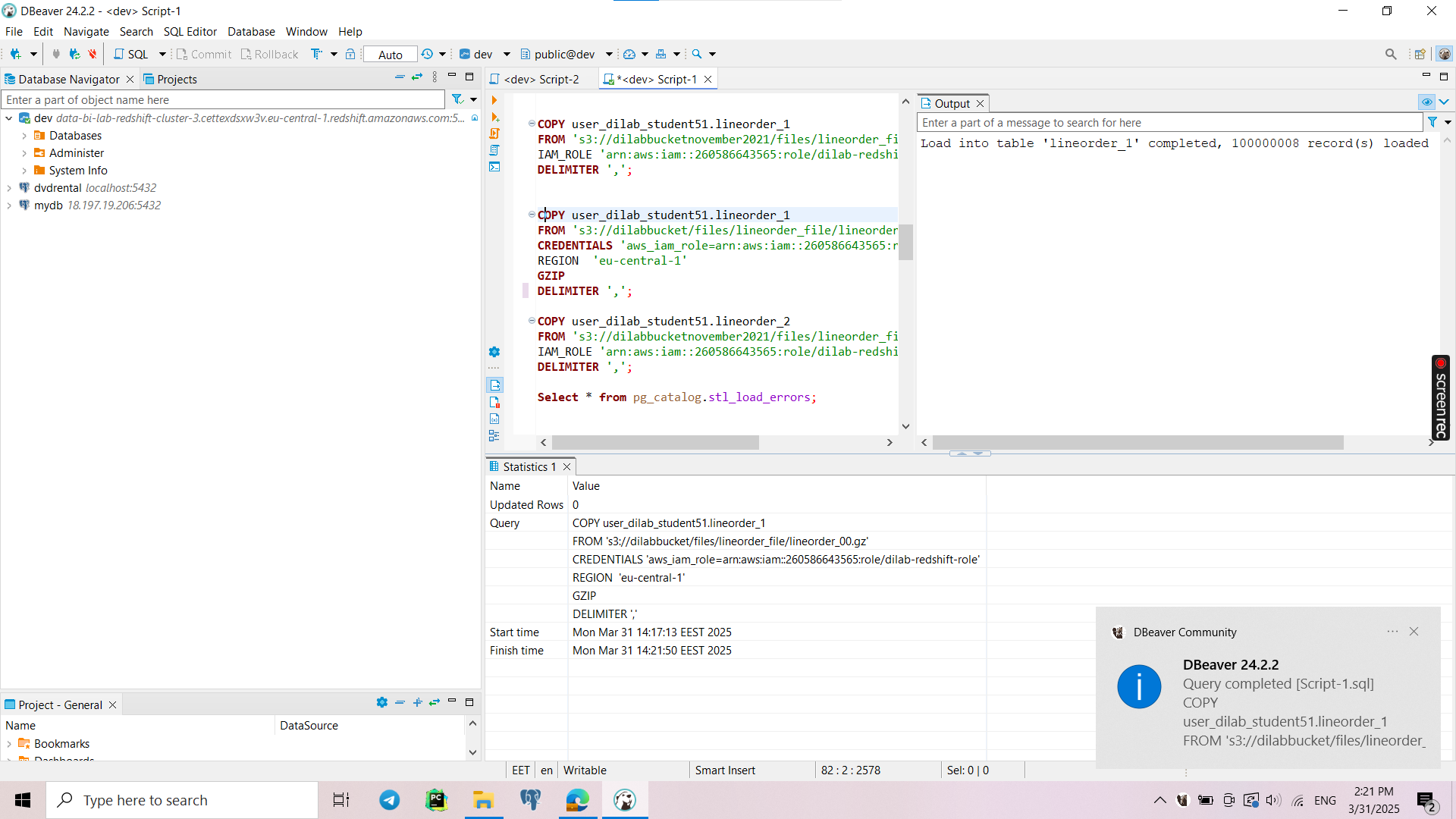
**CALL** user\_dilab\_student51.load\_transaction\_report();

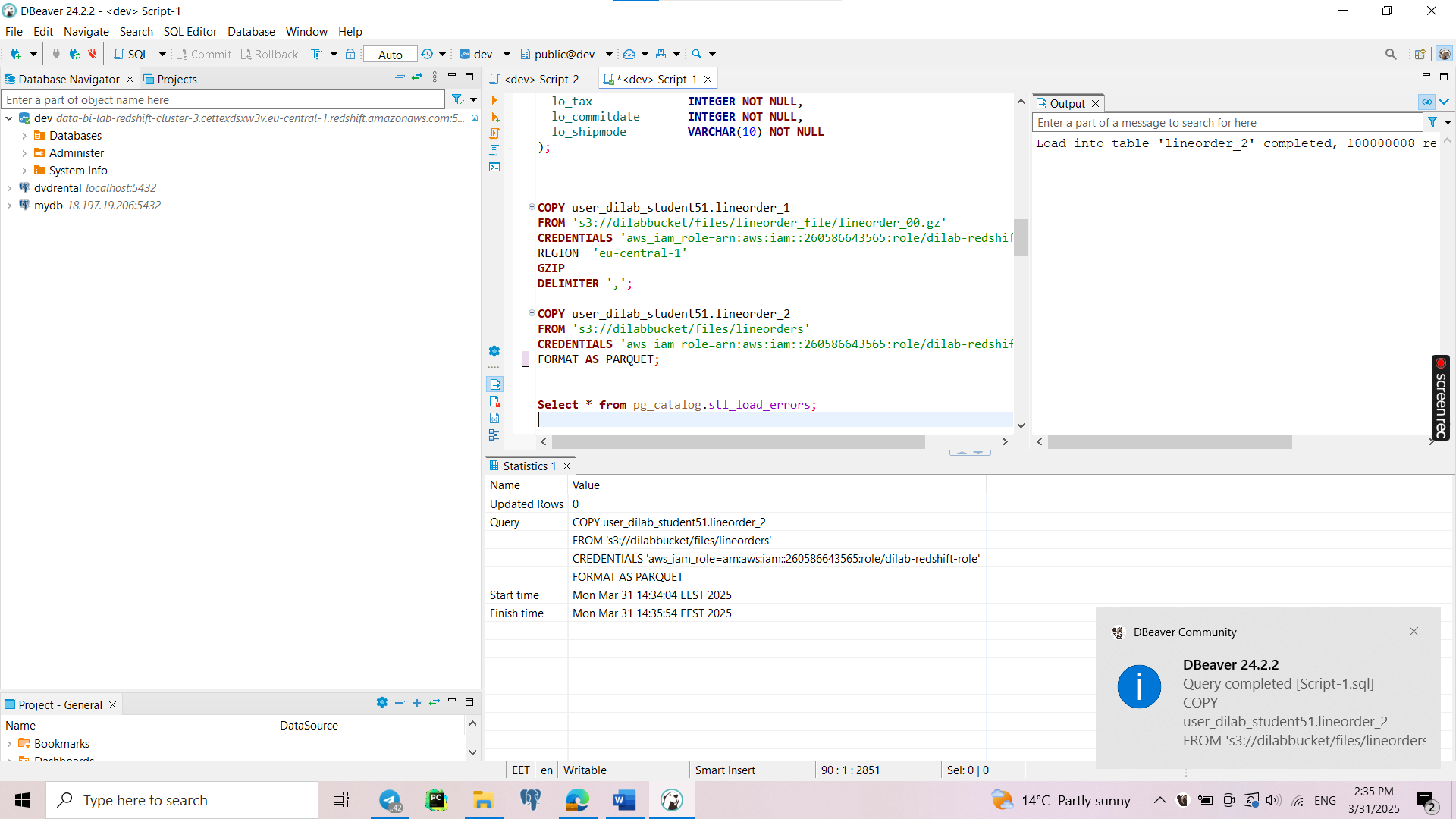
**SELECT** \* **FROM** user\_dilab\_student51.transaction\_report;



7. I tried to connect to PowerBI, but everytime error ‘connection timed out’, also in group chat was written “it's impossible to connect to Redshift via PBI as it's a private network now and connection has to be established via PBI - EC2 – Redshift”, so I did not do this task.

**Copy question**

****

****

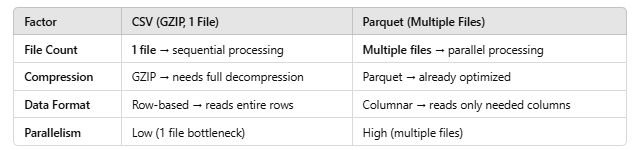
The second COPY command (lineorder\_2 with Parquet) executed **faster** than the first (lineorder\_1 with GZIP CSV) because of **parallel processing differences and data format efficiency**.

**Single File** (lineorder\_00.gz):

* Since there's only one file, Redshift cannot parallelize the load effectively.
* The leader node has to read and distribute data sequentially across compute nodes.

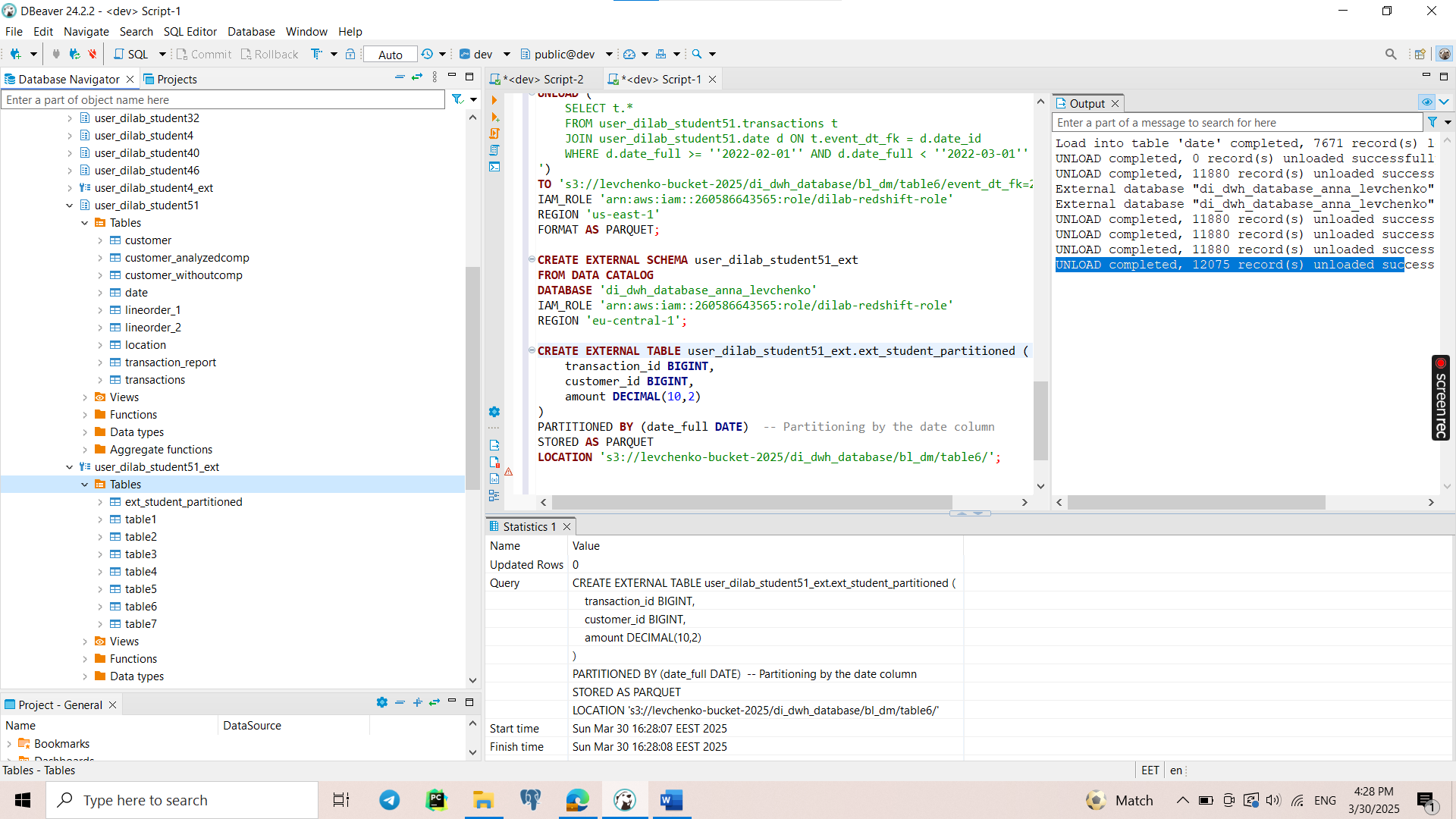
**Multiple Parquet Files** (0000\_part\_00.parquet, 0001\_part\_00.parquet, etc.):

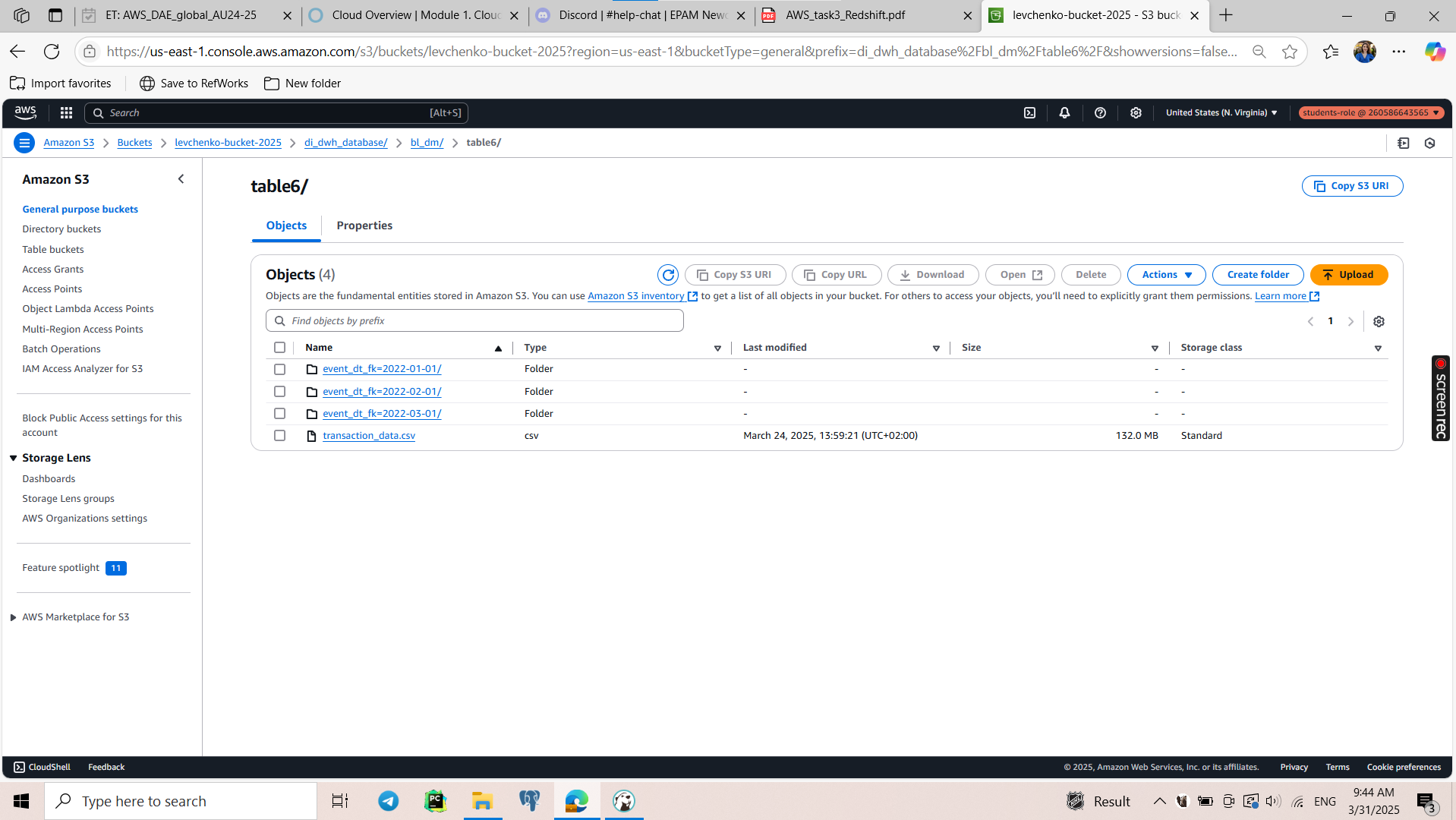
* Each file can be read independently by different Redshift nodes.
* This allows parallel loading, meaning multiple slices of the cluster work at the same time.
* Parquet files are columnar, meaning Redshift loads only the needed columns, reducing I/O and improving speed.

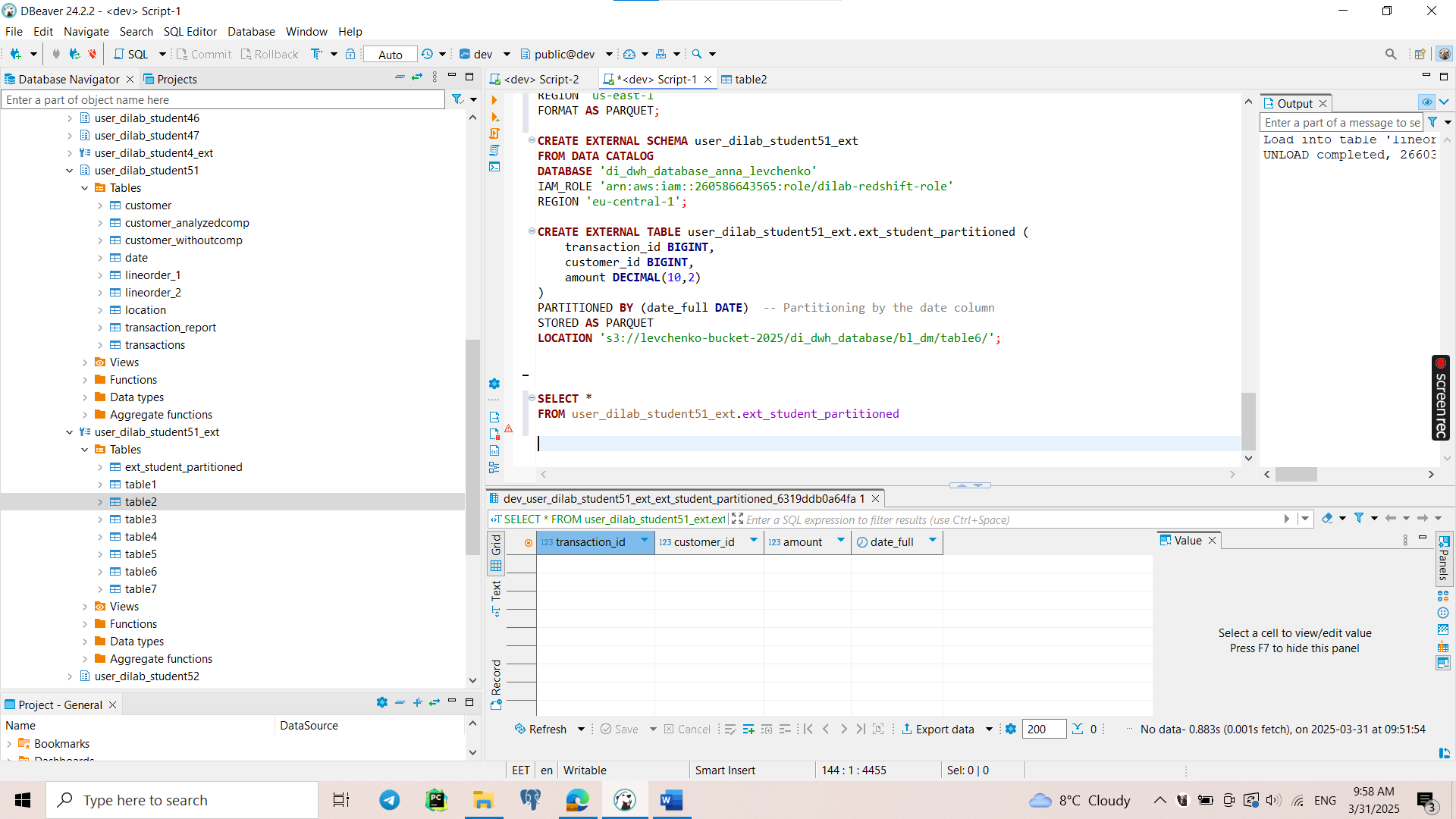


**External tables**

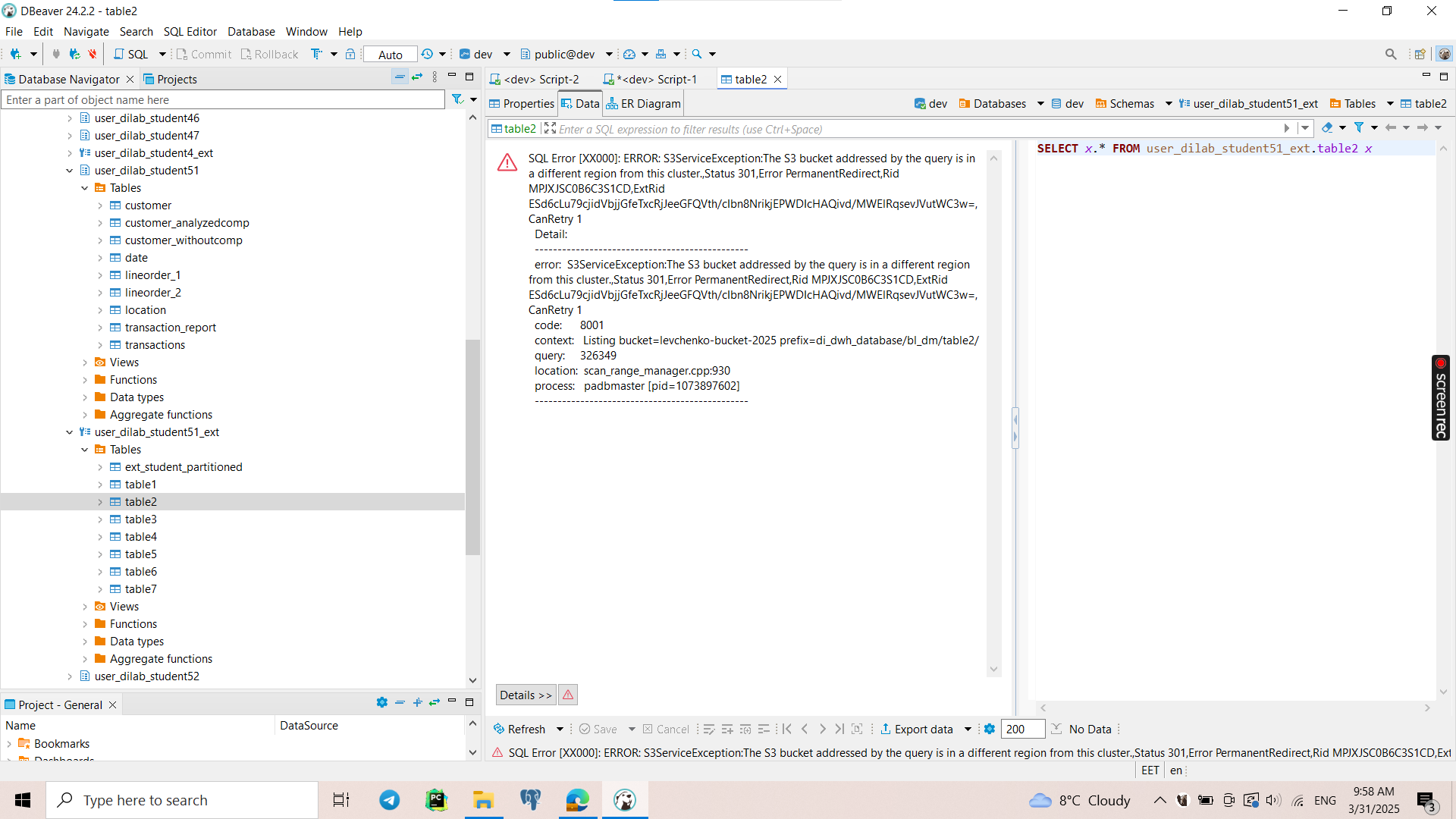
1-2.

****

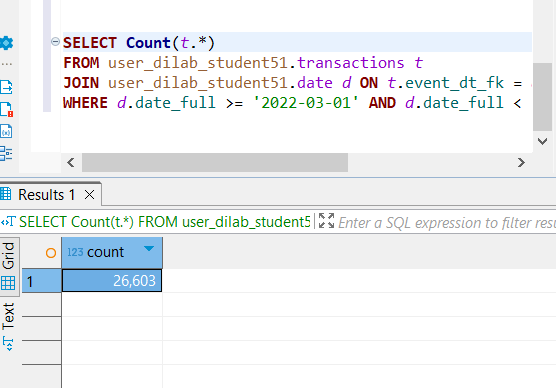
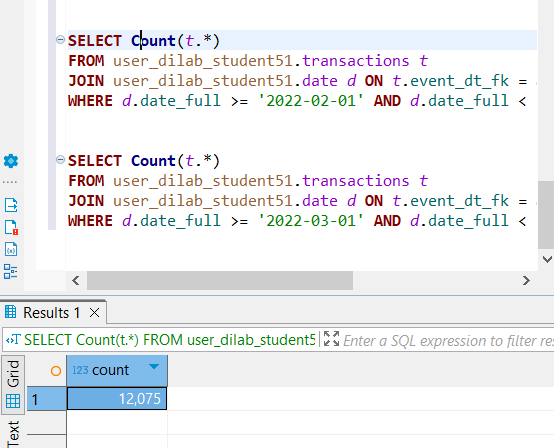
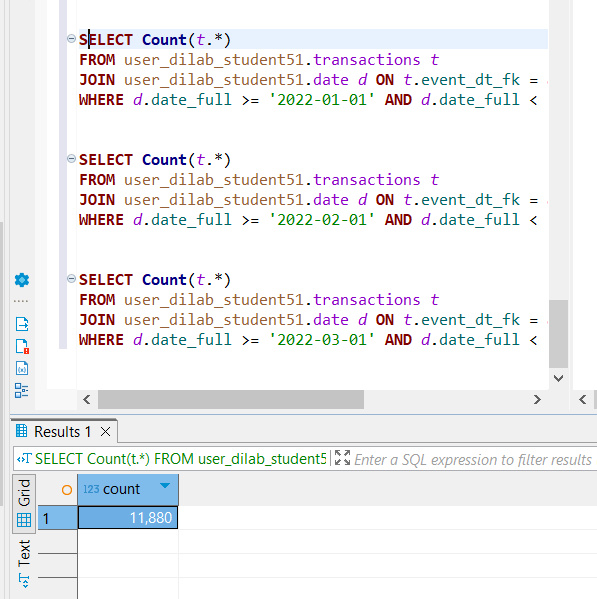
****

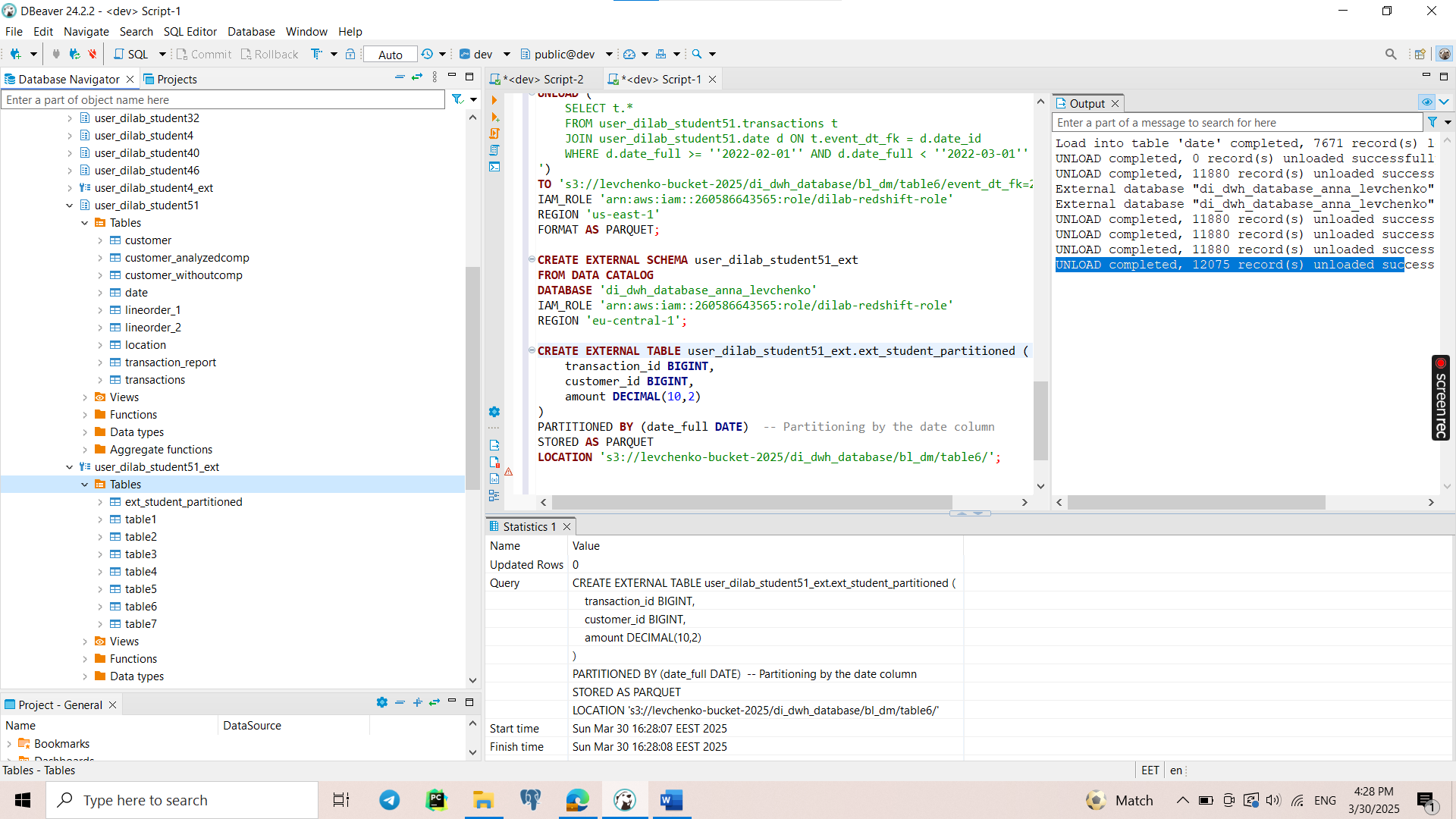
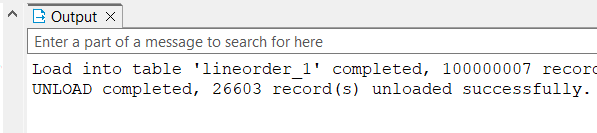
****

My partitioned table is empty, and then I figured out that all tables in schema *user\_dilab\_student51\_ext* are not opened and this error occurs.

****

3.



****

In this case, I checked only that the number of unloaded records to bucket is the same as selecting those rows from not partitioned shema. Of course, query with compering will not show 0, because partition table is empty.

4.

