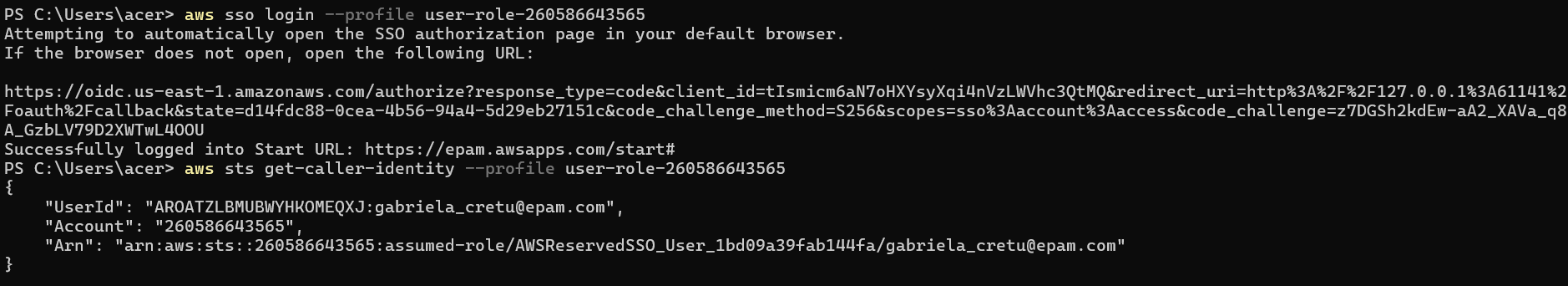
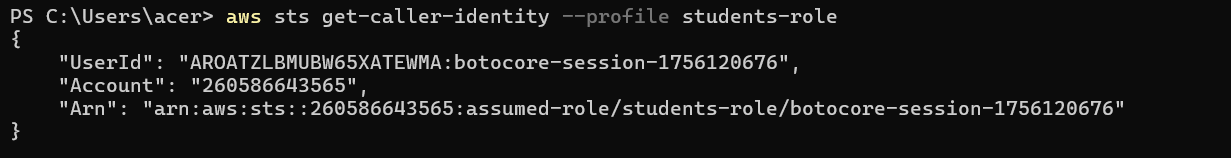


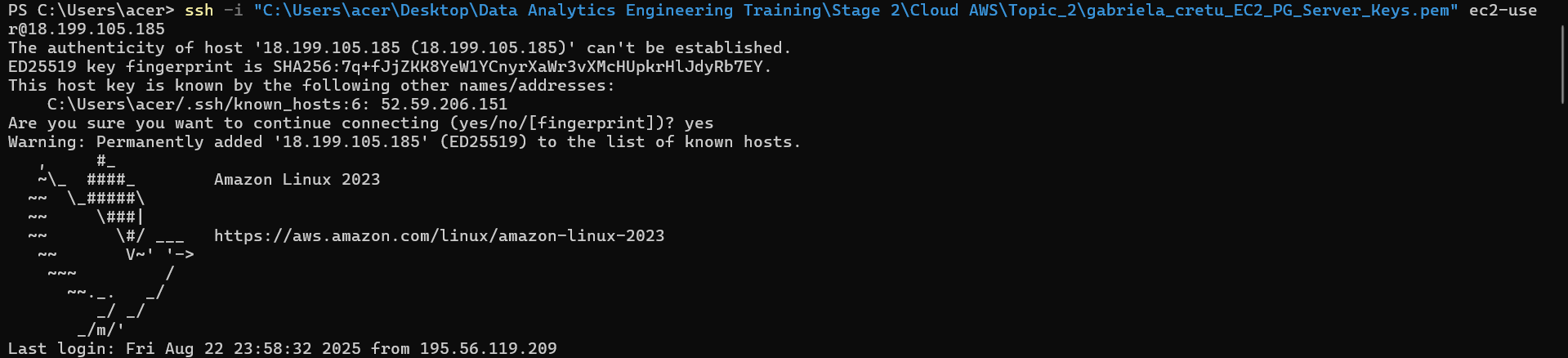
| Business Template  **Amazon Redshift** |
| --- |
|  |

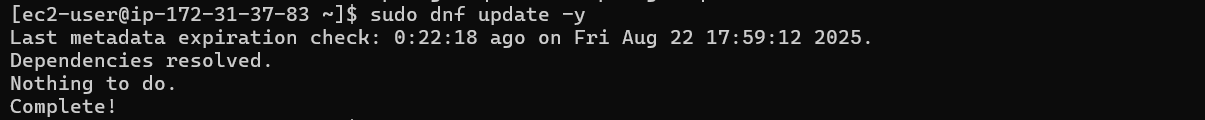
1. Login to Redshift. The EPAM VPN should be enabled. The public access to cluster is disabled, therefore you can use SSH tunnel to establish connection. Use the EC2 machine from HW2 or launch the new one. Perform the steps below:

1.1 Update packages On any Linux before installing packages, a user should run a system update command that will ensure all the latest available updates are installed on the system. Plus, this will also refresh the DNF package cache. So, get access to your terminal or connect to your Amazon Linux instance via SSH and run the following command: sudo dnf update

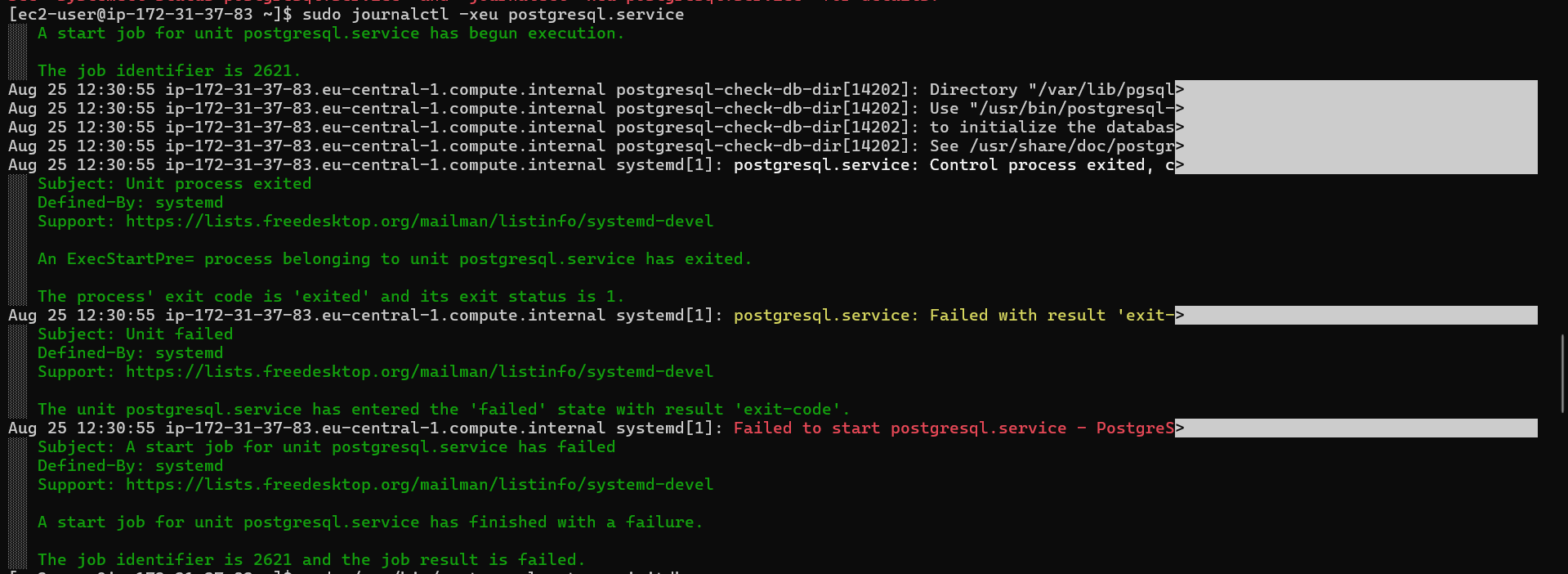




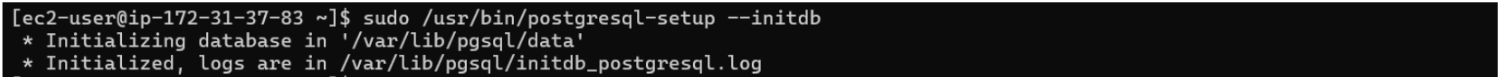




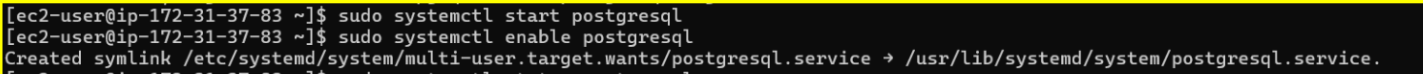
1.2 Install PostgreSQL 15 on Amazon Linux 2023: Well, the best thing currently you don’t need to add any repository to get the PostgreSQL version 15 on your Amazon Linux 2023 because it is available through its system default repo. So, what you have to do is just run the given command. It will install both the client and server parts of the PostgreSQL Database system on your Linux: sudo dnf install postgresql15.x86\_64 postgresql15-server I already had it installed on my EC2, so I just had to activate it.



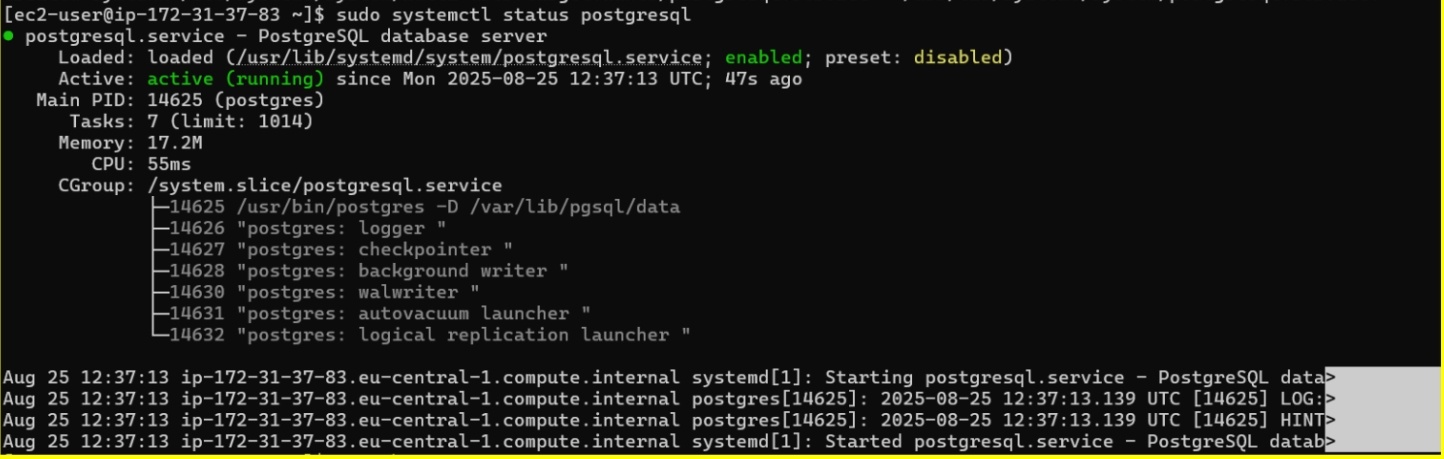
1.3 Initialize the PostgreSQL Database: sudo postgresql-setup –initdb



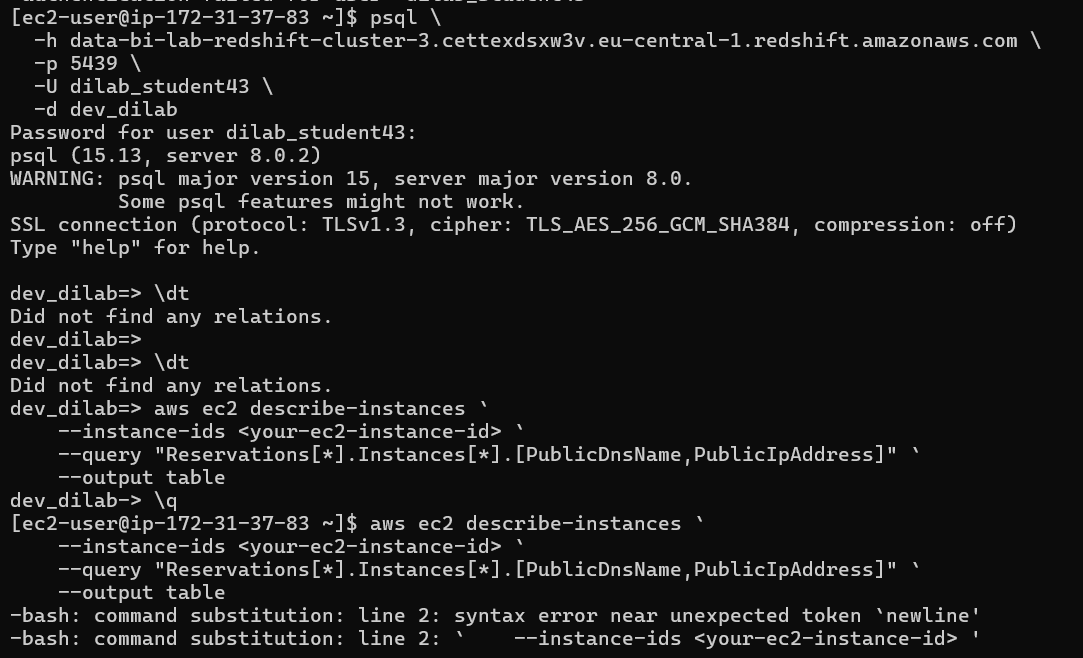
1.4 Start and Enable Service: sudo systemctl start postgresql sudo systemctl enable postgresql AWS Cloud for Data Engineering Redshift Confidential 3



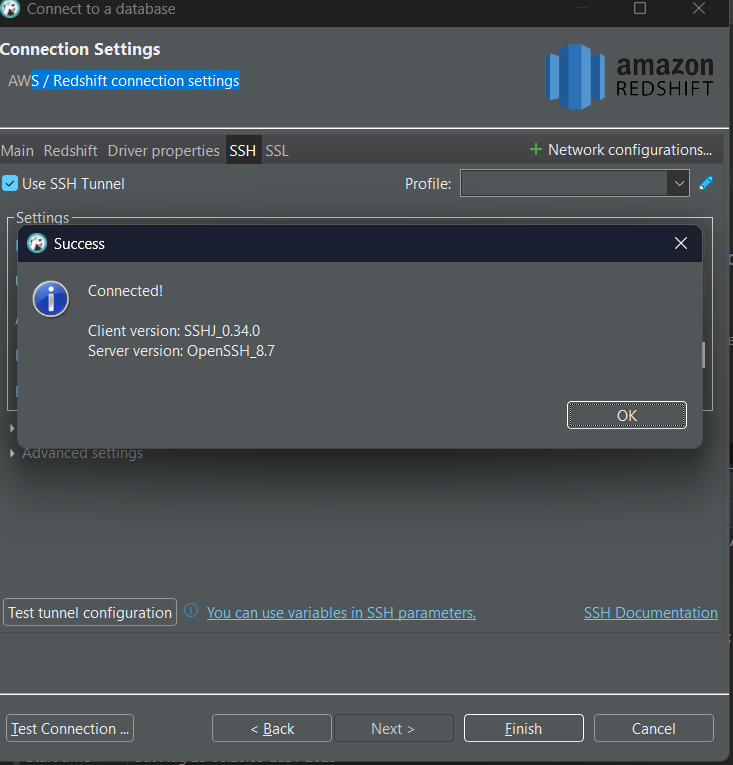
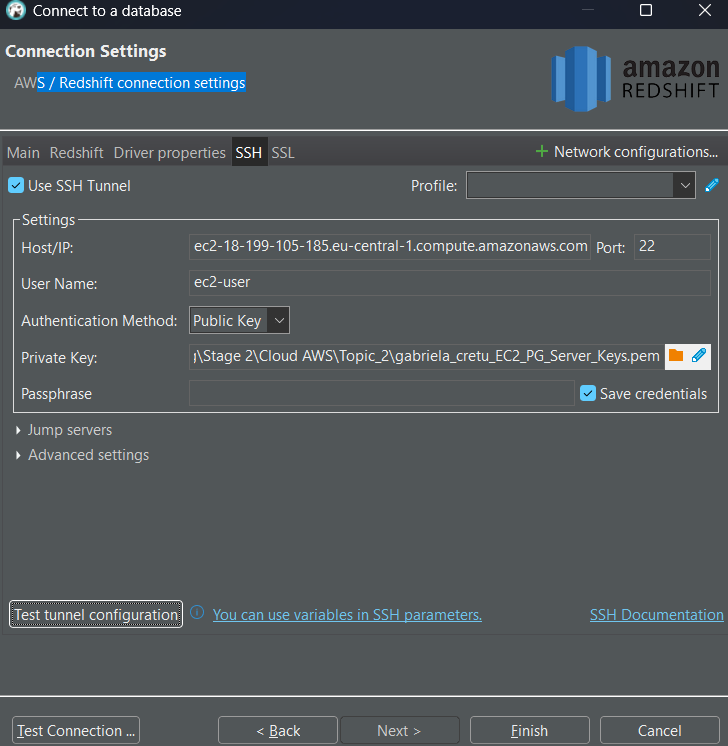
1.5 To confirm the service is running without any errors, here is the command to follow: sudo systemctl status postgresql

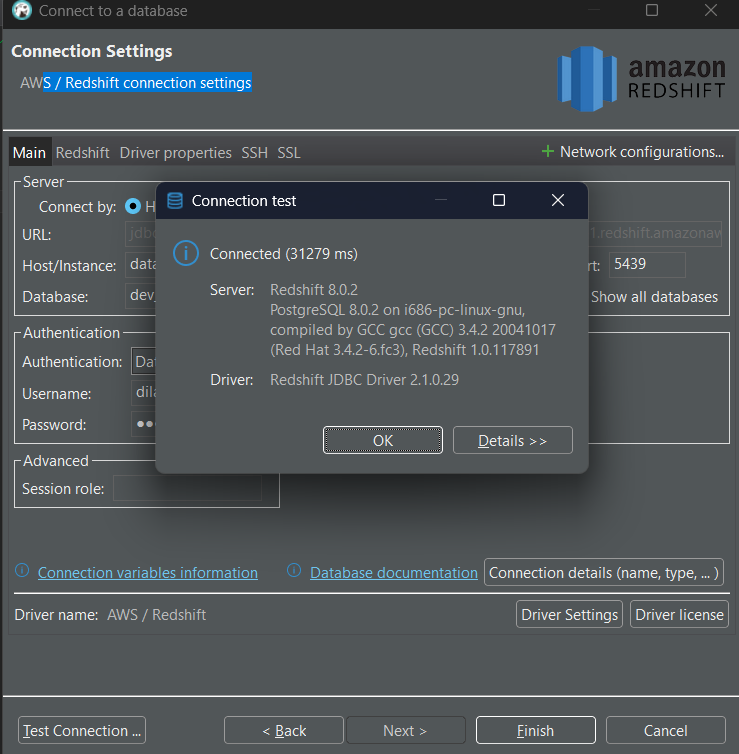
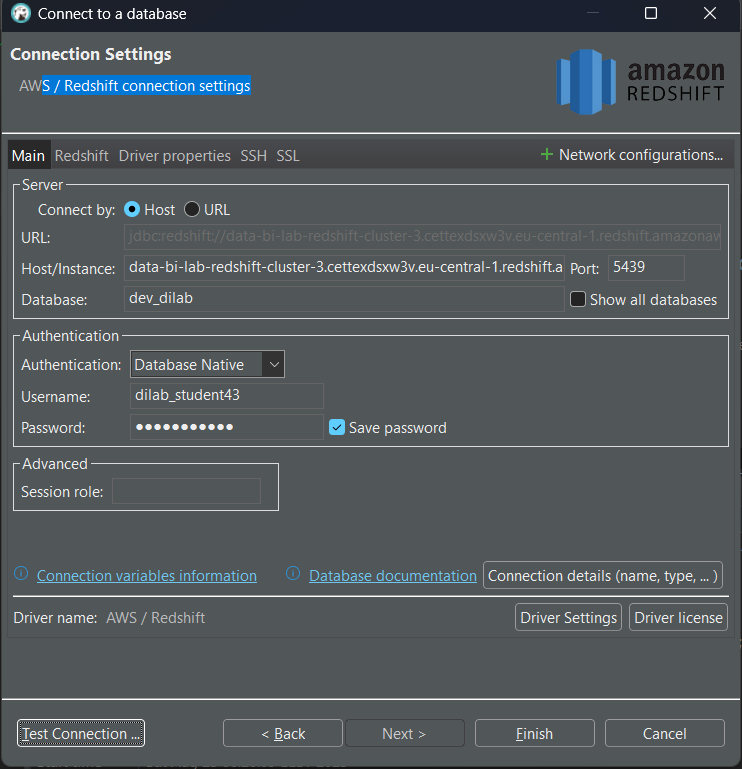


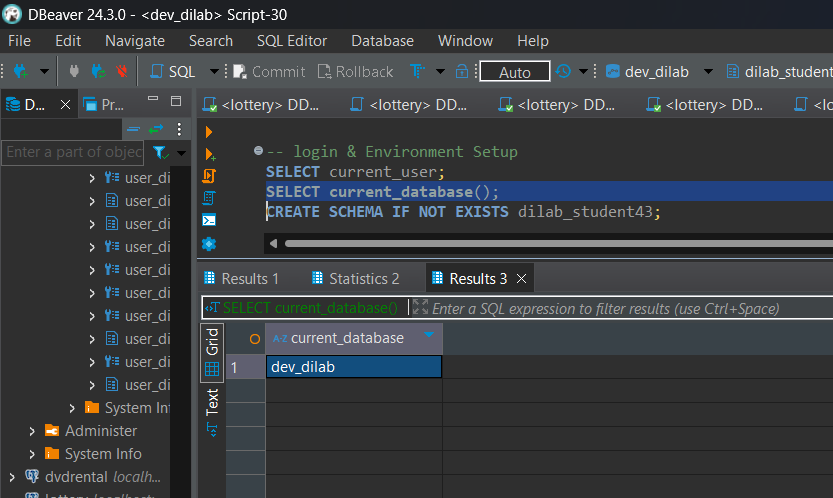
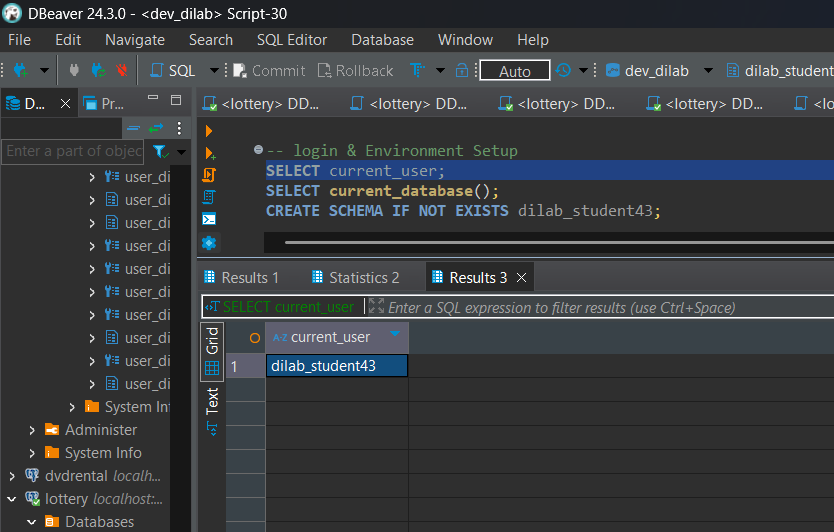
1.6 Use the following command to check connection between EC2 and Redshift(specify the user\_name / password provided to you by mentors): psql -h data-bi-lab-redshift-cluster-3.cettexdsxw3v.eu-central- 1.redshift.amazonaws.com -v schema=public -p 5439 -U -d dev AWS Cloud for Data Engineering Redshift Confidential 4



1.7 Configure connection settings: 2. After logging in you should provision data from data lake to Redshift to be able to transform it using stored procedures.







2. After logging in you should provision data from data lake to Redshift to be able to transform it using stored procedures. One way of doing it is by using COPY command. Use this command to provision data into the table created by you in your schema (user\_dilab\_student(1..32)). Use AWS documentation and example of syntax below to load the data. Attached role you can find using web console or CLI.

Example:

copy user\_schema.lkp\_smsc\_special\_sms (sender, id, active\_dt, inactive\_dt, created\_by, create\_date, is\_valid)

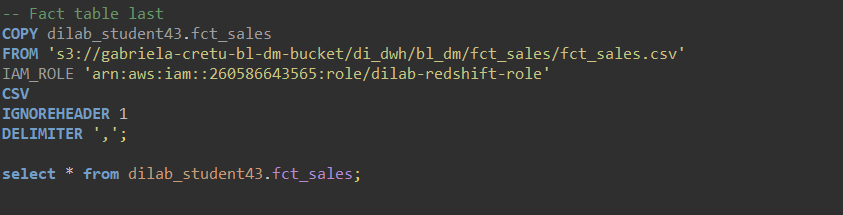
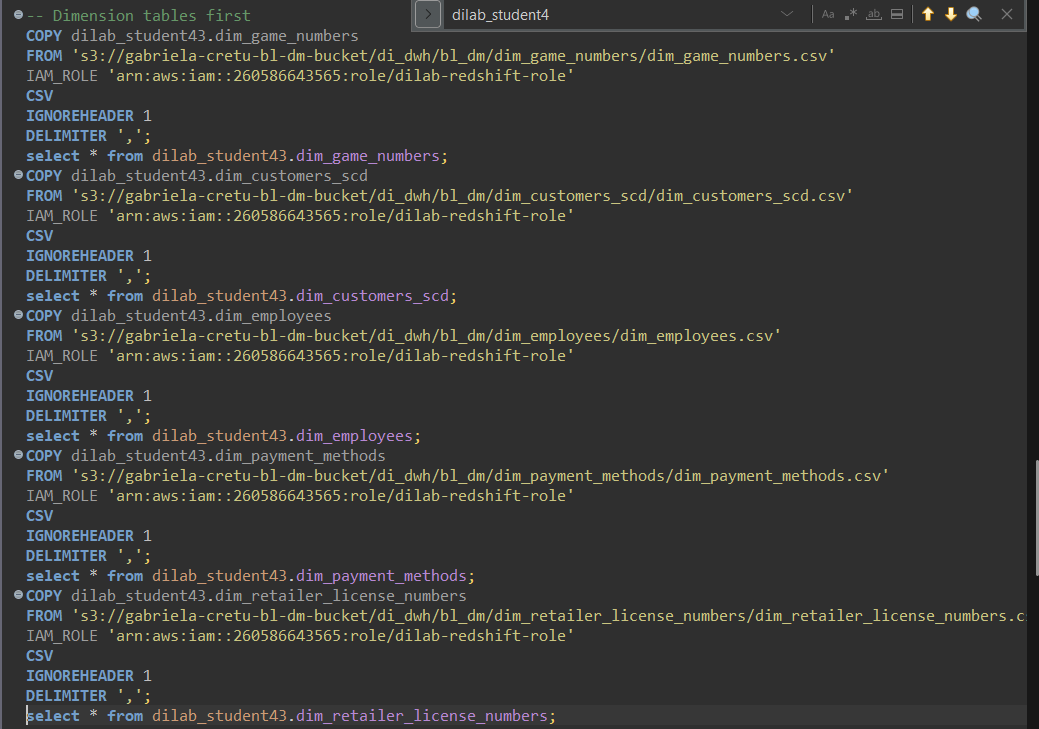
from 's3://aws-cdr-gen-data/lkp\_smsc\_special\_sms.csv' credentials

'aws\_iam\_role=arn:aws:iam::123456789102:role/rs-s3-role-read' region 'eu-central-1'

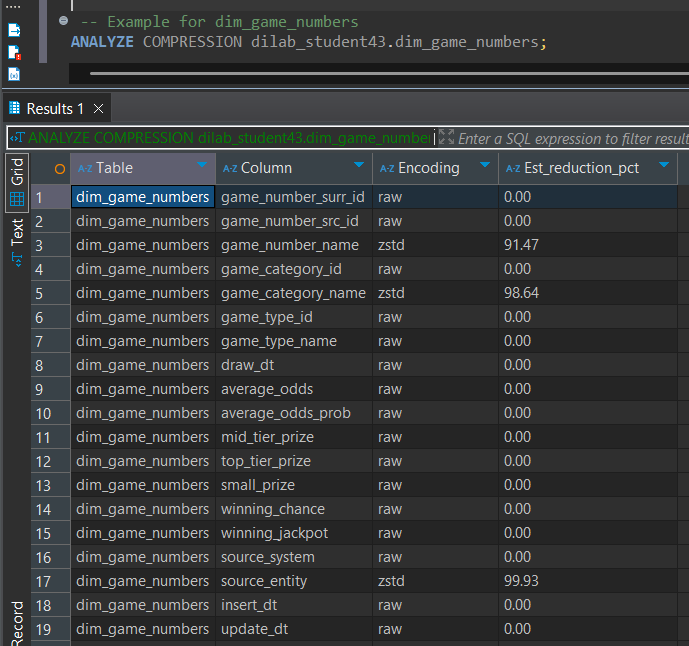
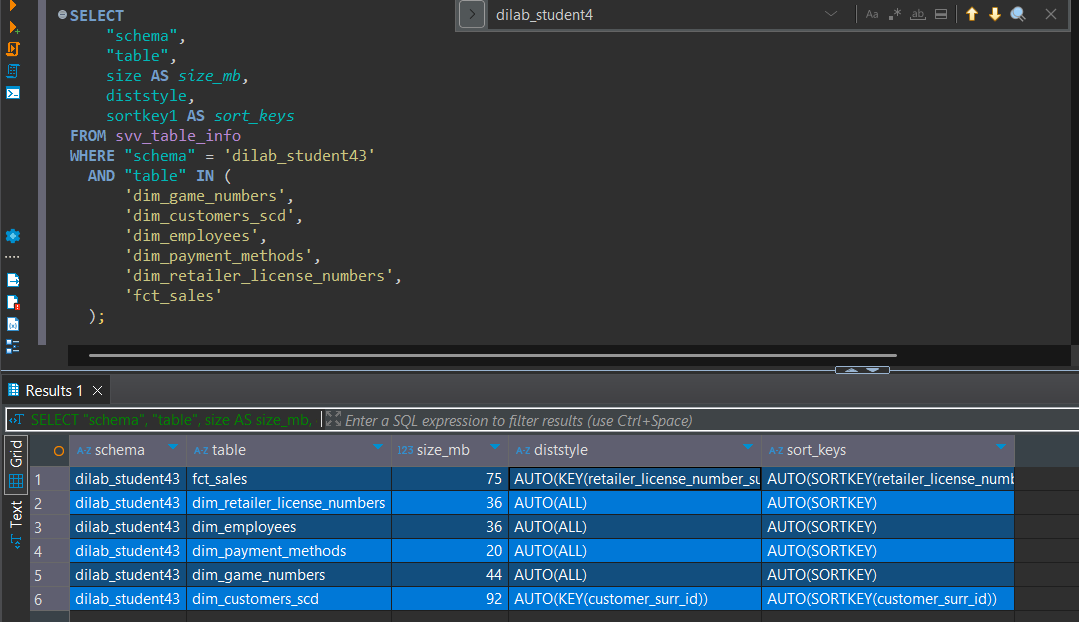
delimiter ',' csv

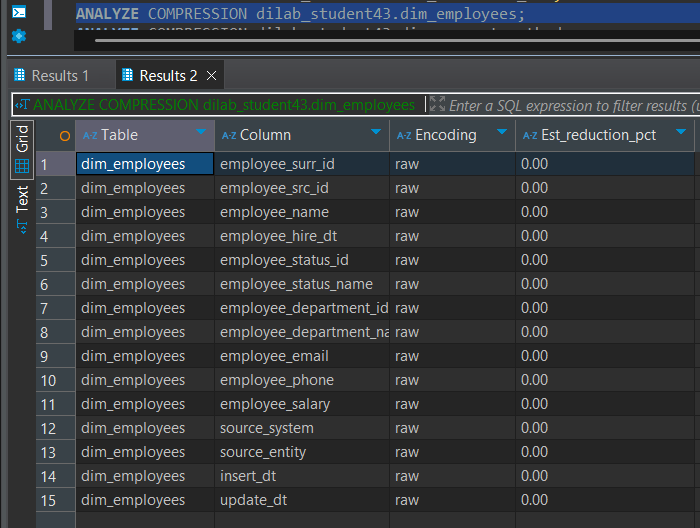
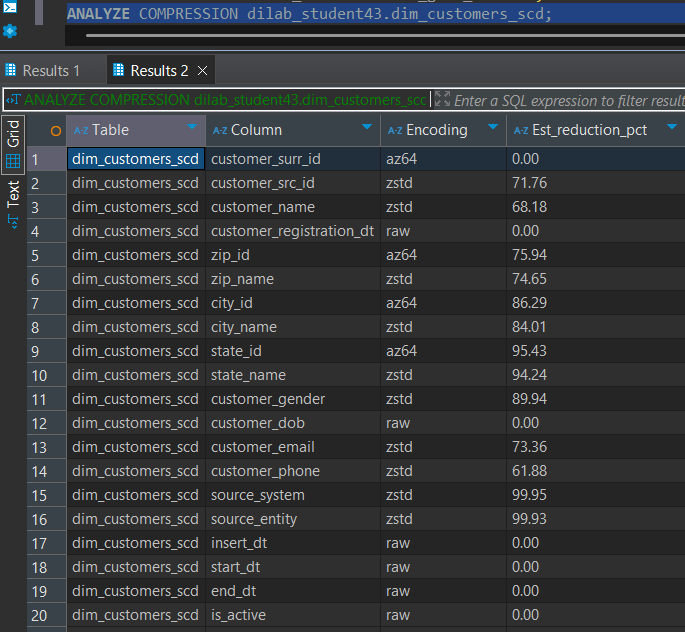
DATEFORMAT AS 'MM-DD-YYYY' IGNOREHEADER 1;

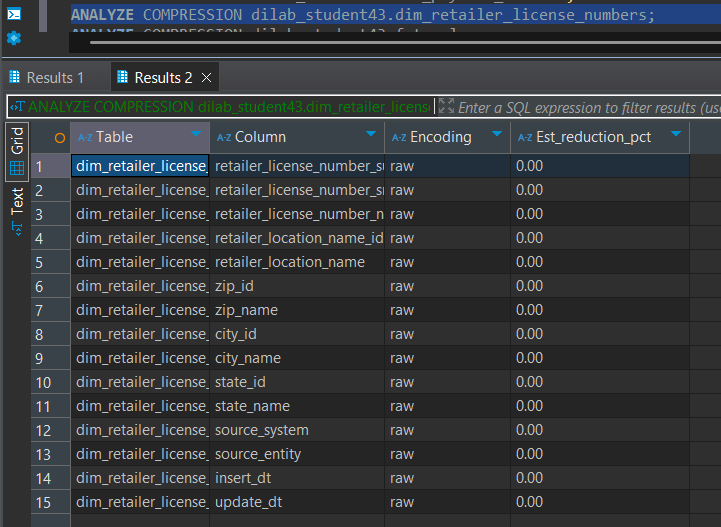
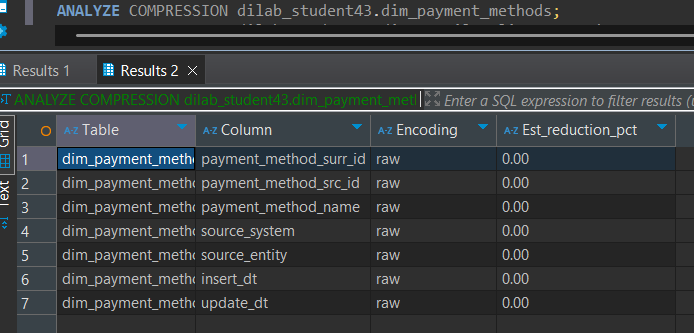
a. You could load all the tables, but because of limited resources please load only tables that are needed to create a report for your customer (at least 3 tables should be involved). It can be some tables that will be used in aggregations and calculations of KPIs. The business meaning of the report is up to you and your creativity.

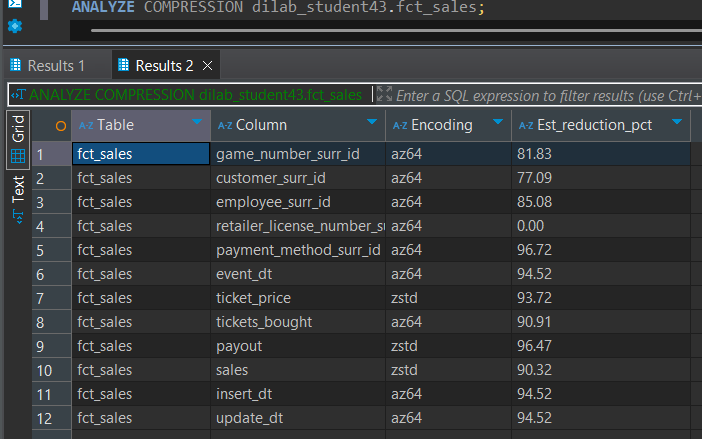


b. After loading your tables check its initial compression types, distribution style, sort keys. Make a description of your analysis.









When it comes to **key distribution and sort keys**, we can see that most of the smaller dimension tables are automatically set to **AUTO ALL** and **AUTO SORTKEY**. This is expected because smaller tables are usually replicated across all nodes in the cluster, which avoids data movement during joins and improves query performance. Then, when we look at the larger tables like **fct\_sales** and **dim\_customers\_scd**, the distribution style and sort keys are automatically allocated by Redshift: retailer\_license\_number\_surr\_id for **fct\_sales** and customer\_surr\_id for **dim\_customers\_scd**. This happens because larger tables must be distributed in a way that minimizes shuffling across nodes during joins, so Redshift automatically picks the most commonly joined key as both the distribution and sort key.

Next, let’s discuss a bit about **encodings**. The general types of encoding in Redshift are:

* **raw** → no compression applied
* **az64** → Redshift’s optimized encoding for numeric columns (integers, floats, dates)
* **zstd** → general-purpose dictionary-based compression, best for strings and variable-length data

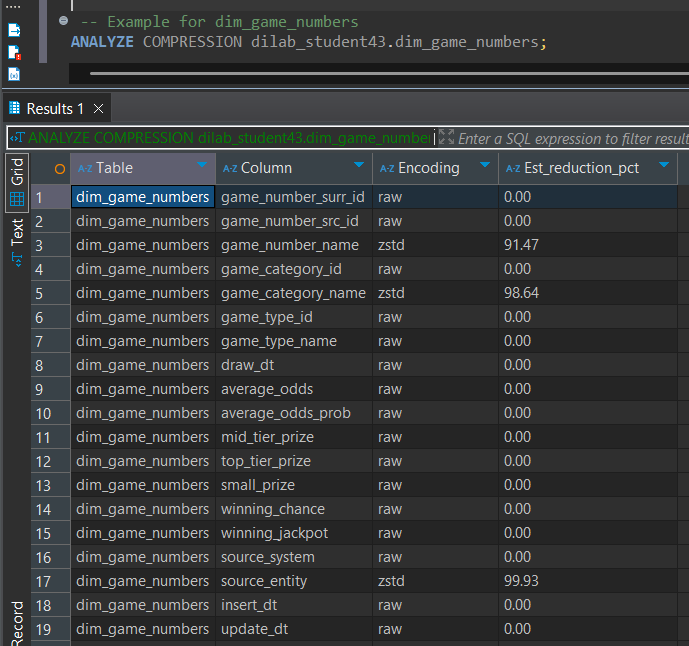
Encodings are necessary because they reduce storage size on disk and improve query speed by reducing I/O.

Looking at the tables:

* **dim\_game\_numbers** is the least optimized: only a few columns (source\_entity, game\_type\_name, and game\_category\_name) benefited from zstd encoding, while most of the columns remained raw. This suggests Redshift didn’t find much benefit in compressing those other columns, likely because of their data type or low cardinality.
* **dim\_customers\_scd** shows a more expected pattern:  
  + az64 for surrogate and numeric IDs (customer\_surr\_id, zip\_id, city\_id, state\_id) since those are integers.
  + zstd for textual attributes like customer\_src\_id, customer\_name, zip\_name, city\_name, state\_name, customer\_gender, customer\_email, customer\_phone, source\_system, and source\_entity, since these are variable-length strings that compress well.
  + raw for date columns (customer\_registration\_dt, customer\_dob, insert\_dt, start\_dt, end\_dt) because dates are already efficiently stored.
* On the other hand, the **dim\_employees**, **dim\_payment\_methods**, and **dim\_retailer\_license\_numbers** tables have **no automatic encoding applied** (everything is raw). This could be because they are relatively small dimension tables, and Redshift did not prioritize applying compression here. Still, manual tuning (applying az64 and zstd) could save space and improve scan performance.
* Finally, one of the most optimized tables is **fct\_sales**. As expected for a fact table:  
  + az64 is used for all foreign key columns (game\_number\_surr\_id, customer\_surr\_id, employee\_surr\_id, retailer\_license\_number\_surr\_id, payment\_method\_surr\_id) and also for numeric/time values (event\_dt, tickets\_bought, insert\_dt, update\_dt).
  + zstd is applied to money-related columns (ticket\_price, payout, sales), which compress extremely well due to repetitive numeric patterns.
  + The result is one of the most space-efficient and performance-optimized tables in the dataset.

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

a. Identify compression types (encoding) of each column of this table (YOUR\_TABLE\_defaultcomp)



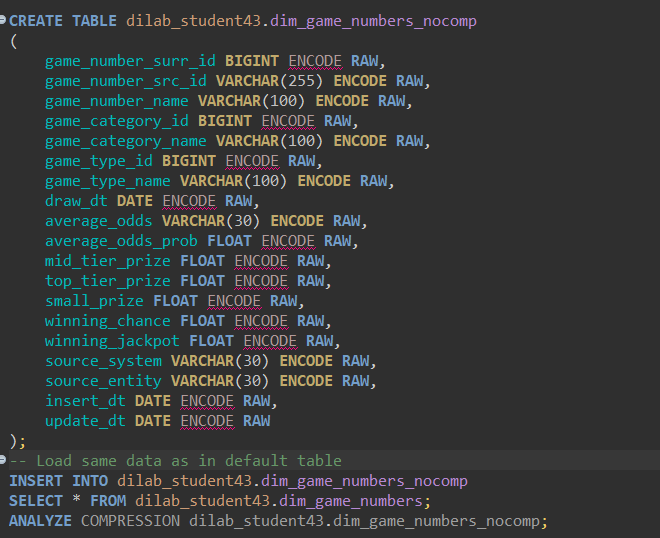
For this task, I considered **dim\_game\_numbers** as the default table. After reviewing the column encodings, we can see that most of the columns remain with **raw encoding** (no compression). The only columns that Redshift automatically optimized are:

* game\_number\_name → **zstd**
* game\_category\_name → **zstd**
* source\_entity → **zstd**

All other columns (such as surrogate keys, source IDs, dates, and numeric values) are left as **raw**, meaning they are stored without additional compression.

This shows that Redshift identified string-based columns with higher cardinality or variability as good candidates for zstd, while the other fields did not benefit from further encoding.

b. Create table YOUR\_TABLE\_withoutcomp with similar to YOUR\_TABLE\_defaultcomp columns/data types, but without any compression applied and put there the same data as in the YOUR\_TABLE\_defaultcomp table.



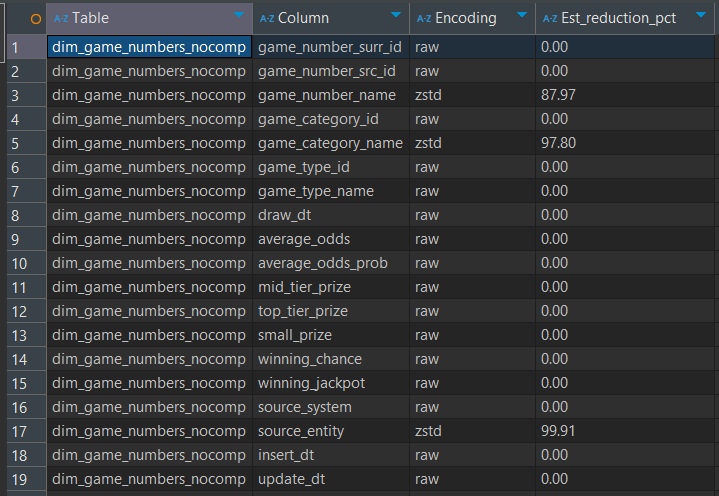
In order to create a table without any compression, I simply defined a new table called **dim\_game\_numbers\_nocomp**, which has the same columns and data types as dim\_game\_numbers. For this version, I explicitly encoded every column as **RAW**, meaning no compression is applied. After the table was created, I inserted the same data from dim\_game\_numbers into dim\_game\_numbers\_nocomp.

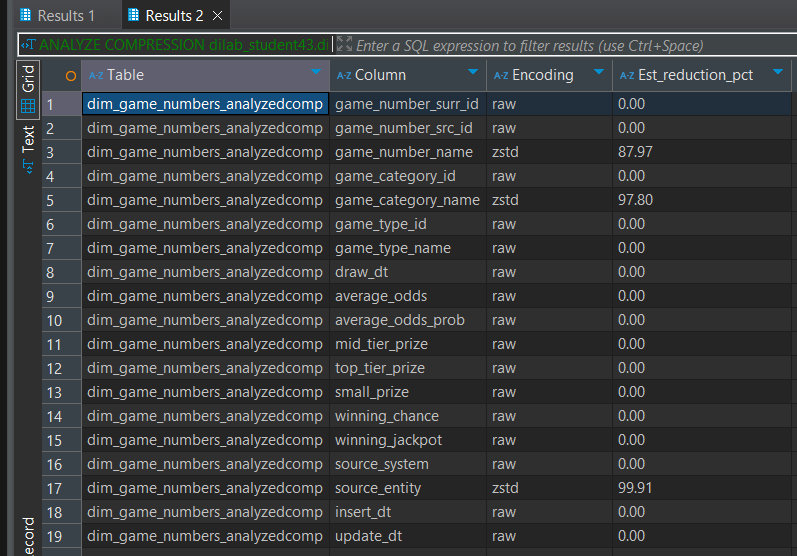
c. Use analyze command (on YOUR\_TABLE\_defaultcomp or YOUR\_TABLE\_withoutcomp table) to identify best compression methods suggested by Redshift. Create a table YOUR\_TABLE\_analyzedcomp (same columns but applying recommended encoding types and put same data as in the YOUR\_TABLE\_defaultcomp table there).

To identify the optimal encodings for each column, I ran the command:

ANALYZE COMPRESSION dilab\_student43.dim\_game\_numbers\_nocomp;

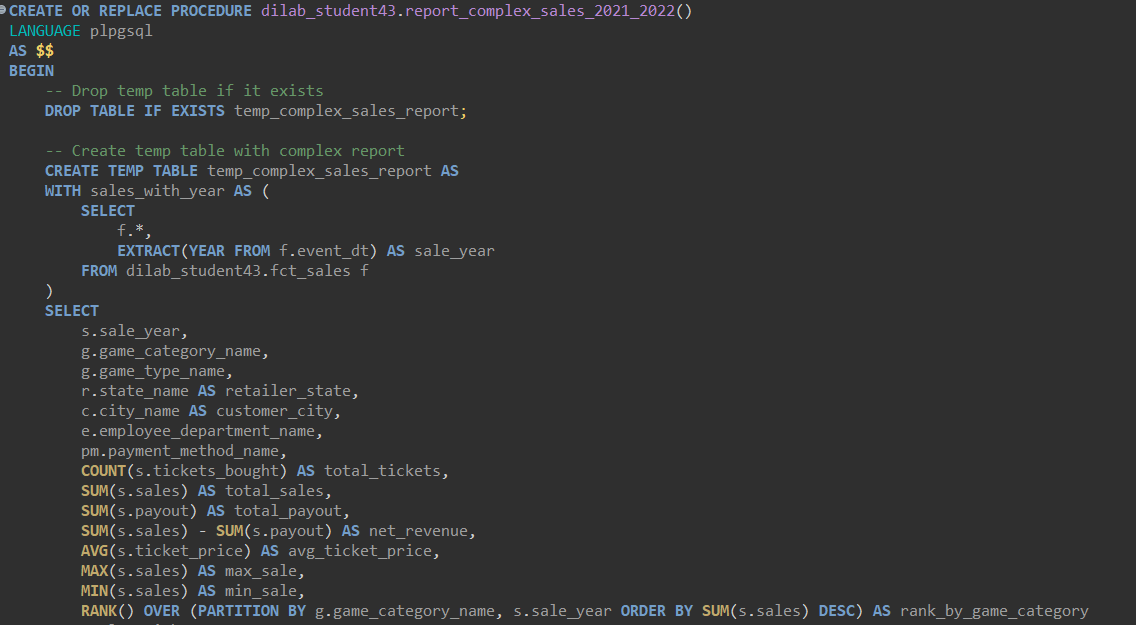
Redshift returned the **best compression methods** for each column based on the data distribution. Using these recommendations, I then created a new table called **dim\_game\_numbers\_analyzedcomp** with the same columns and data types, but this time applying the suggested encodings. Finally, I inserted the same data into this optimized table.

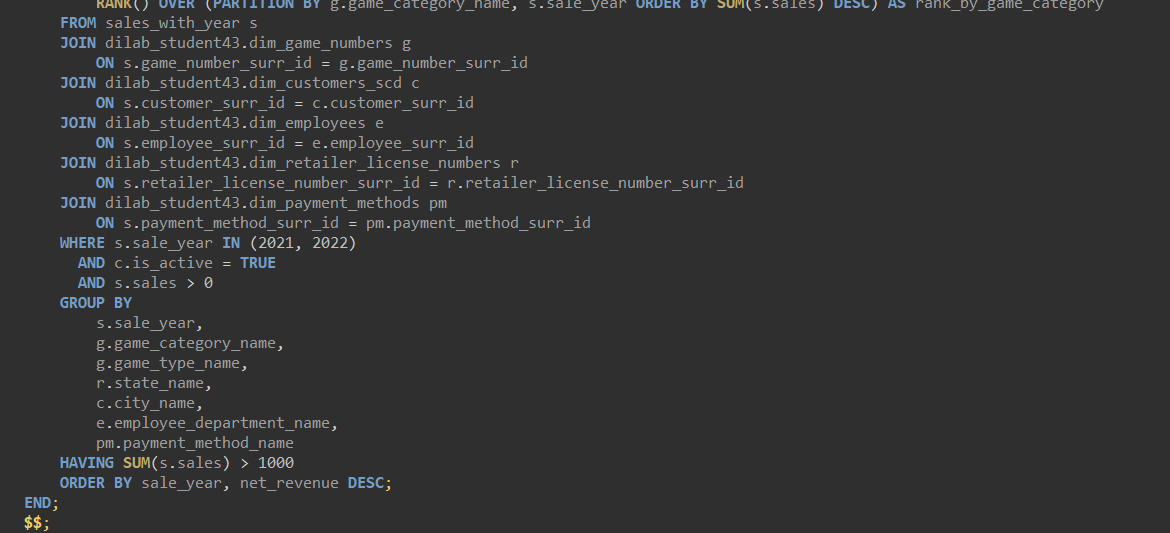




4. Prepare a stored procedure that will load your report. The main part of the stored procedure is the SELECT statement that should use joins of all your tables (at least 3 tables should be involved).

a. Prepare a stored procedure

The stored procedure produces a **complex sales performance report** for the years 2021 and 2022 by combining fact sales data with several dimension tables. It starts by extracting the sale year from each transaction and then joins the sales with games, customers, employees, retailers, and payment methods to provide a full business view. The report groups sales by year, game category, game type, retailer state, customer city, employee department, and payment method, and then calculates key performance indicators such as total tickets sold, total sales, total payout, net revenue (sales minus payout), average ticket price, maximum and minimum sale, and a rank of games by category and year based on sales volume. Business rules are applied to include only active customers, positive sales, and categories with total sales above 1000. The final result highlights which games and dimensions drive the highest revenues, offering insights into customer behavior, retailer and employee contributions, and payment trends across different categories and years..

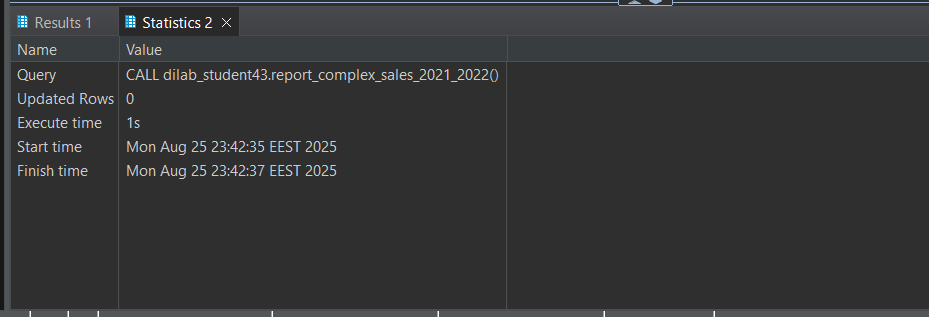
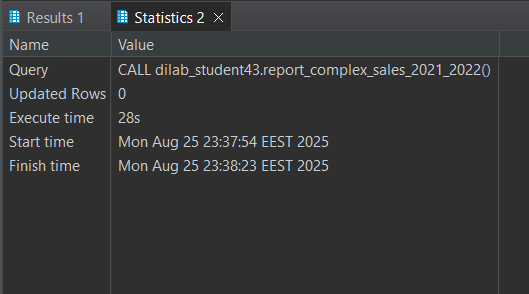


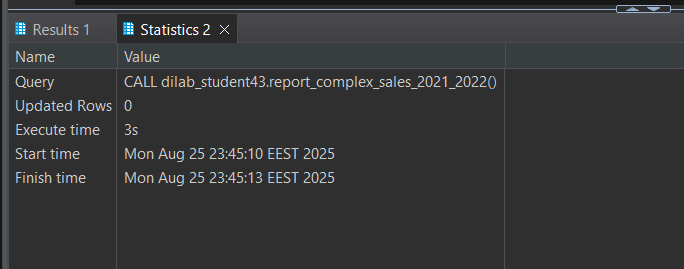
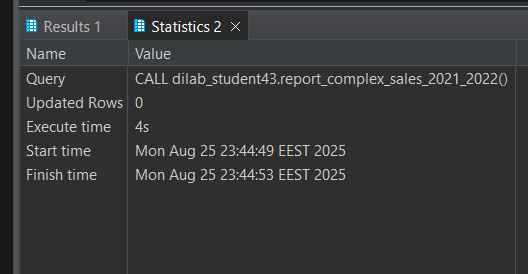
b. Describe the execution plan of this SELECT statement BEFORE optimization, log the time of query execution. E.g.:

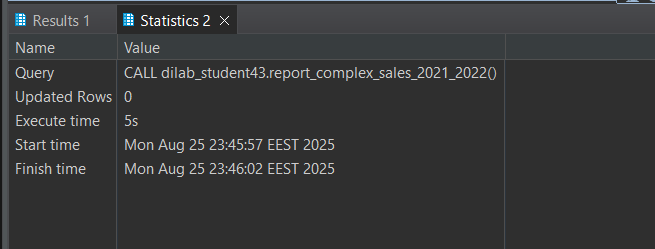
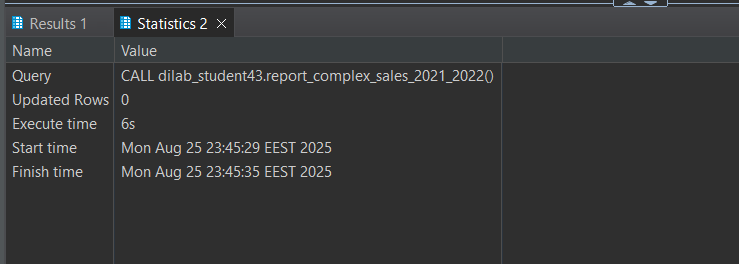
NOTE: Please do not forget to turn off the result caching and do not consider the results of the first execution!

NOTE: Example of performance tests description and results you can follow in this and further tasks: <https://dev.classmethod.jp/articles/redshift-sortkey-usecase-en/#section-04-03>

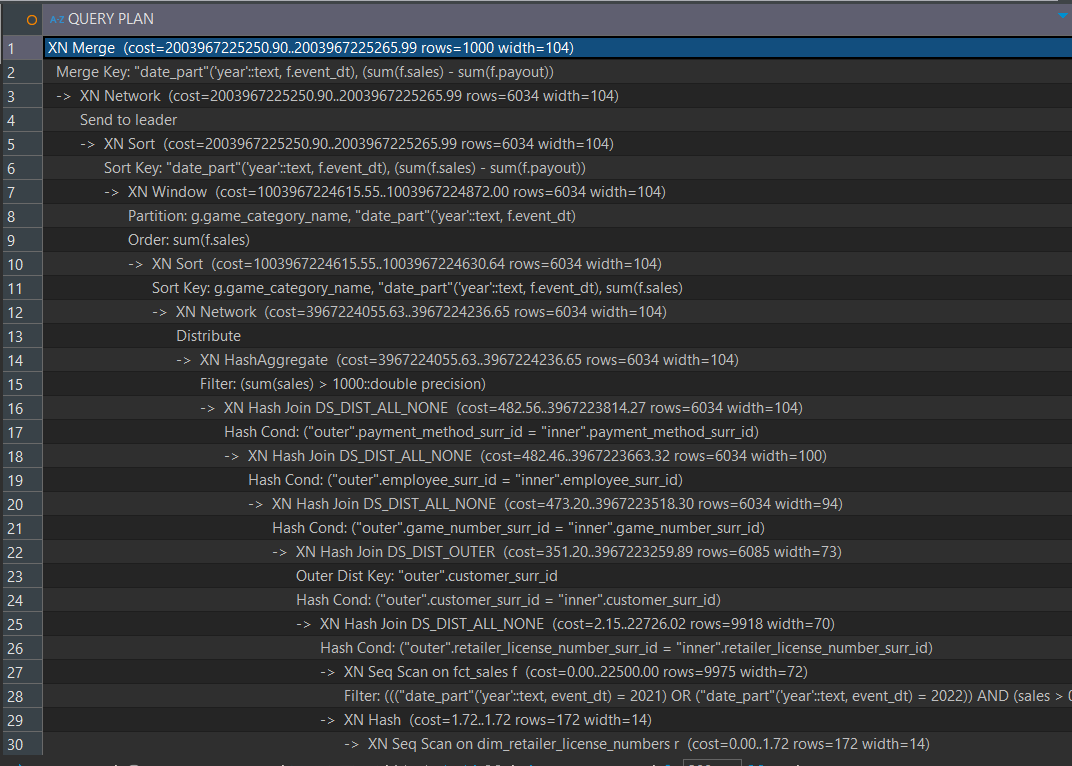
***Execution times***

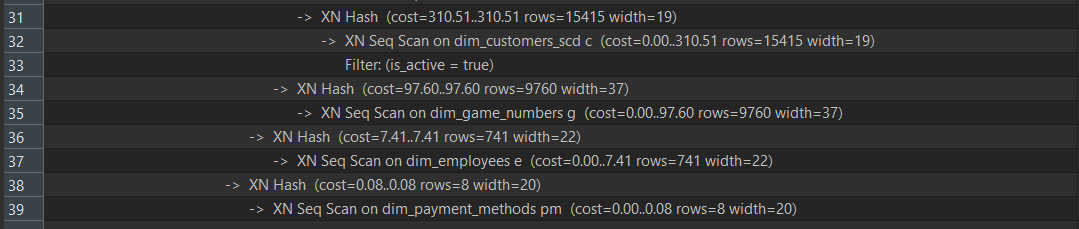






***Execution plan***





When I look at the query plan, I see that the overall structure ends with an **XN Merge**, which is the final step where Redshift merges and sorts the results. This sorting is needed because I use a RANK() window function, which partitions by game category and year and orders by total sales. The reported costs are huge, but I know these numbers are not literal execution times — they are relative estimates based on table size and row counts.

I also notice significant **data movement** in the plan. After computations are performed on all slices, Redshift uses an **XN Network → Send to leader** step to push the partial results to the leader node for the final merge. There are also **XN Distribute** operations, which happen because Redshift needs to redistribute the fact table rows across slices so the join keys align. This redistribution, along with broadcasts of smaller dimension tables, is often where bottlenecks occur.

Looking at the joins, I see multiple **hash joins**: first fct\_sales joins with dim\_retailer\_license\_numbers, then the result joins with dim\_customers\_scd, followed by dim\_game\_numbers, then dim\_employees, and finally dim\_payment\_methods. Most of these joins use **DS\_DIST\_ALL\_NONE**, meaning the smaller dimension tables are broadcasted to all slices, while the large fact table is redistributed on the join keys.

Once the joins are completed, Redshift performs **aggregations and window functions**. An **XN HashAggregate** computes totals and averages such as SUM, COUNT, and AVG, and then an **XN Window** step calculates the ranking by game category and year. Sorting also appears in the plan because both GROUP BY and ORDER BY require sorted intermediate results. This explains why the query feels heavy on the first execution — sorting and window functions consume memory and add extra computation across all slices.

Finally, I see that all the dimension tables (dim\_game\_numbers, dim\_employees, dim\_payment\_methods, dim\_customers\_scd, dim\_retailer\_license\_numbers) are accessed with **sequential scans**, which makes sense because they are relatively small. However, fct\_sales is also accessed with a sequential scan, applying filters on the event date (2021–2022) and sales > 0. Since fct\_sales is a large table, scanning all rows and then redistributing them for the joins is a big factor in the long runtime of this report.*(Result cache disabled: SET enable\_result\_cache\_for\_session TO OFF)*

| **Run** | **Execution Time (sec)** | **Notes** |
| --- | --- | --- |
| 1 | 28 | First execution, reading from disk, **not considered for comparison** |
| 2 | 1 | Result cache likely still affecting this run |
| 3 | 4 | True performance begins stabilizing |
| 4 | 3 | Consistent performance |
| 5 | 6 | Slight fluctuation due to cluster load or node distribution |
| 6 | 5 | Stable performance after multiple runs |

**Observation:**

* First run is significantly slower because Redshift reads all data from disk and performs full joins, aggregations, and window functions.
* Subsequent runs are faster and fluctuate slightly due to **cluster resource usage** and **slice-level caching**.
* For proper performance evaluation, only runs **after cache stabilization** (e.g., runs 3–6) should be considered.

c. Describe why the result of the first execution is not the best one to compare with.

When I analyze the performance of my query, I know that the **first execution is not the best one to compare with**. This is because during the first run, Redshift has to do much more work: it reads all of the data from disk, including the large fct\_sales table and every dimension table, which adds significant disk I/O overhead. At the same time, Redshift also needs to parse the SQL, compile the query, generate the execution plan, and allocate resources across slices, so there is extra planning time on top of the raw execution. Another reason is that no data is cached yet — table blocks, intermediate results, and distribution buffers only get stored in memory after the first execution. By the time I run the query again,

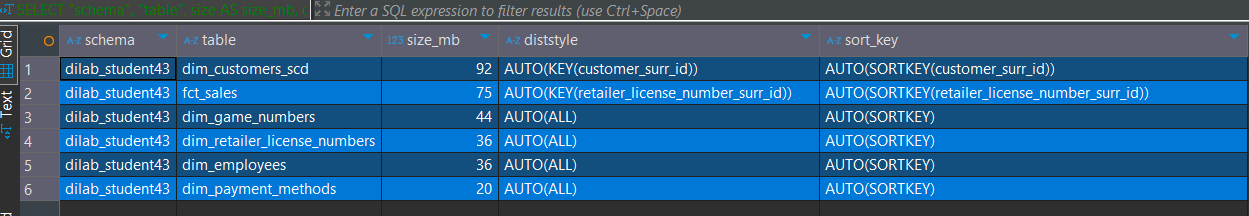
Redshift is able to reuse those cached elements, making subsequent runs appear much faster. For this reason, the first execution mostly reflects the **initial setup cost** rather than the steady-state performance of the query, so it would overestimate the runtime if I used it as a baseline. The best practice here is to disable result caching with SET enable\_result\_cache\_for\_session TO OFF and then rely on later executions, after the first, to measure realistic query performance.

d. Describe the existing distribution style of tables, its sort keys.

For the existing distribution styles and sort keys, I can see that Redshift automatically applies different strategies depending on the table size and usage. The **fact table fct\_sales** is distributed by a key on retailer\_license\_number\_surr\_id and also sorted by the same column. This ensures that rows are colocated with the retailer dimension, reducing data movement and making joins faster. The large dimension table **dim\_customers\_scd** is also distributed by key, this time on customer\_surr\_id, and sorted on the same column. This aligns well with the joins against the fact table, helping with performance when filtering or aggregating by customer.

In contrast, all of the smaller dimension tables — **dim\_game\_numbers**, **dim\_retailer\_license\_numbers**, **dim\_employees**, and **dim\_payment\_methods** — use **ALL distribution**, meaning their full contents are replicated across every slice. Since these tables are small in size, broadcasting them avoids redistribution overhead during joins and keeps query performance efficient.

Overall, the strategy is consistent: **large tables use KEY distribution on their primary join columns**, while **small dimensions are broadcasted (ALL)**. Redshift AUTO handles sort keys for dimensions, while fact tables benefit from explicit sort keys aligned with join conditions.



5. Let’s assume that these joins will be used very often and will not be massively shared with other tables in Redshift. Now you need to optimize your distribution style and sort keys.

You can find more information about proper Sort/DIST key for table creation here: https://docs.aws.amazon.com/redshift/latest/dg/t\_Creating\_tables.html

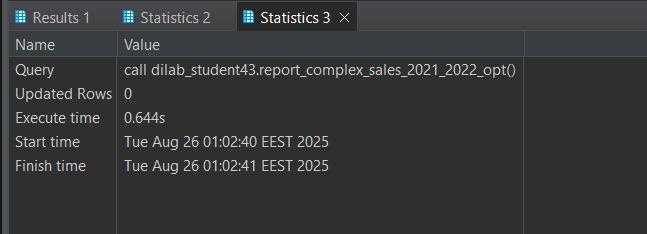
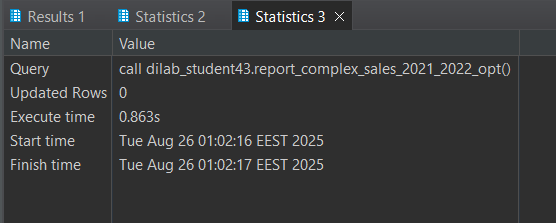
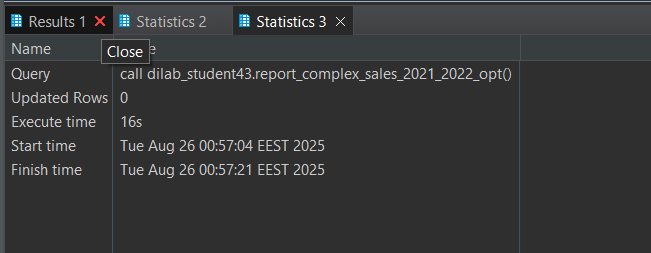
a. Describe why you took a specific distribution style and sort keys.

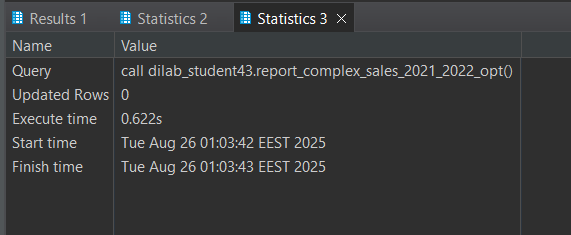
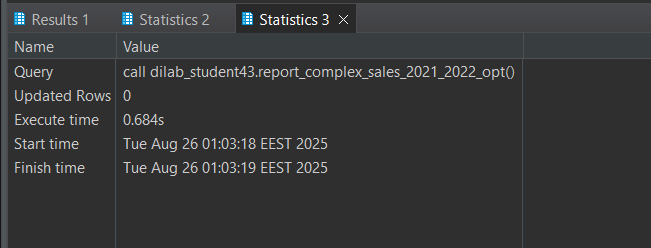
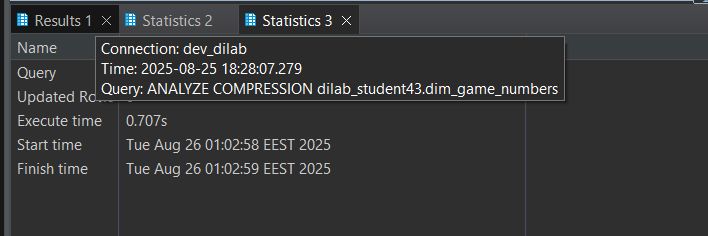
When deciding on distribution styles and sort keys, I focused on minimizing data movement during joins and making scans as efficient as possible. For the **fact table fct\_sales**, I chose to use a **DISTKEY on customer\_surr\_id** because most of the joins happen with dim\_customers\_scd. By colocating rows on the same slices through a common distribution key, I avoid unnecessary redistribution across the cluster. I also applied a **SORTKEY on event\_dt**, since many queries filter or group results by year. Sorting on the event date allows Redshift to take advantage of range-restricted scans, which speeds up time-based queries significantly. For the **dim\_customers\_scd** table, I applied the same **DISTKEY on customer\_surr\_id** to align it with the fact table, and also sorted it on the same column to make joins more efficient. For **dim\_game\_numbers**, I kept an **ALL distribution style**, since this table is relatively small and it’s cheaper to broadcast it across all slices than to redistribute it each time. I also sorted it on game\_number\_surr\_id to speed up the joins with the fact table.

Overall, the choices reflect Redshift best practices: large fact tables and their main dimension share the same **DISTKEY**, small dimensions are broadcasted using **ALL**, and **SORTKEYS** are applied to the most common filtering and join columns. This combination ensures less network transfer and more efficient scans, which directly improves query performance.

b. Describe the execution plan of this SELECT statement AFTER optimization, log the time of query execution. Compare it with BEFORE results. It is mandatory to describe changes in execution plans and how your optimization impacted it. E.g.:

***Execution times***





### Execution Time Comparison: Before vs. After Optimization

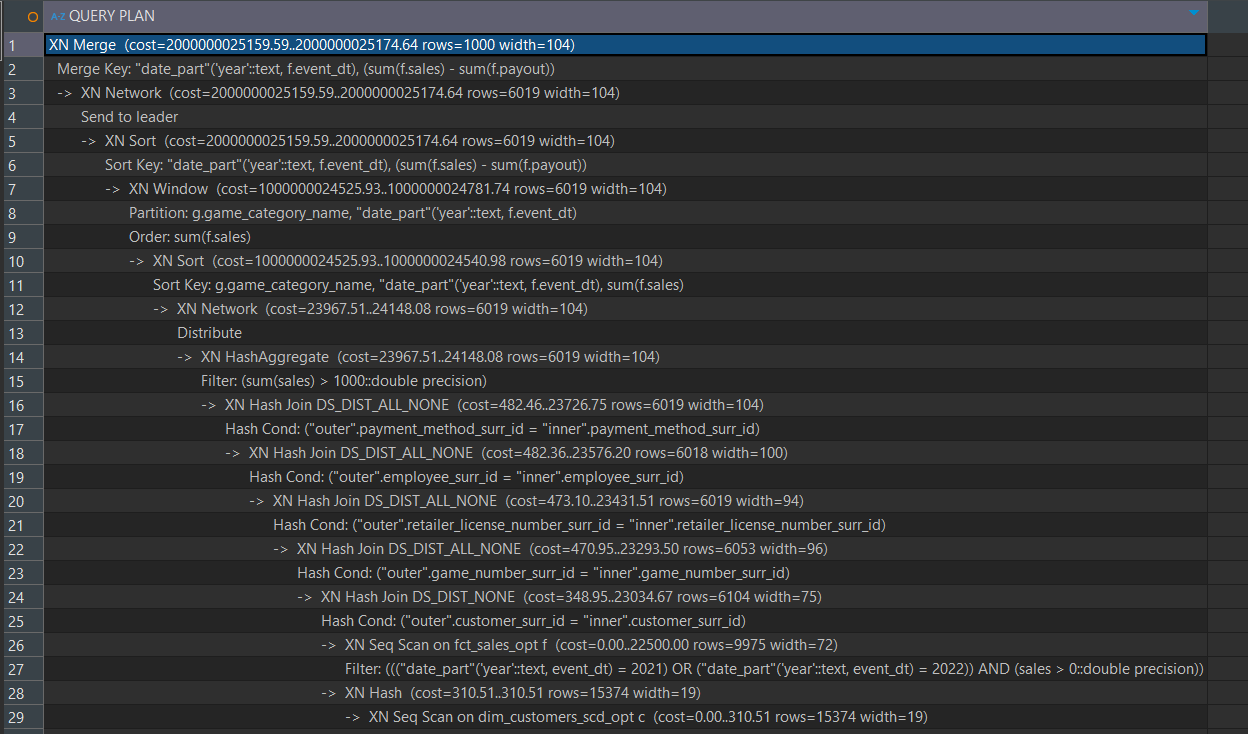
| **Run** | **Before Optimization (sec)** | **After Optimization (sec)** | **Notes** |
| --- | --- | --- | --- |
| 1 | 28 | 16 | First execution reads from disk; includes compilation and redistribution overhead; not considered baseline. |
| 2 | 1 | 0.863 | Result cache may still affect this run; performance begins stabilizing. |
| 3 | 4 | 0.644 | True steady-state performance begins; faster than before optimization. |
| 4 | 3 | 0.707 | Consistent execution; minor variation due to cluster load. |
| 5 | 6 | 0.684 | Slight fluctuation; after-optimization performance remains very fast. |
| 6 | 5 | 0.622 | Stable performance after multiple runs. |

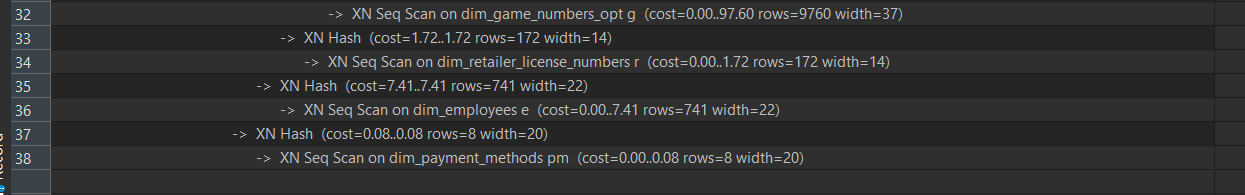
When I compare execution times before and after optimization, the improvement is clear. The first run in both cases is slower because Redshift must read all data from disk and compile the query. The **after-optimization first run is 16 seconds**, slightly faster than the original 28 seconds, reflecting both the overhead of new DISTKEY alignment and table recreation.

After the first run, performance stabilizes, with after-optimization runs consistently below **1 second** (0.622–0.863 sec), compared to before-optimization runs that fluctuate between 1–6 seconds. This shows that once Redshift caches table blocks and intermediate results, the **optimized DISTKEYs and SORTKEYs** allow hash joins to happen locally, reduce network shuffling, and enable efficient range-restricted scans.

**Takeaway:** Even though the first optimized run has some setup overhead, steady-state execution is **much faster and more consistent** than before optimization, demonstrating the clear benefit of aligning distribution and sort keys.

***Execution plan***





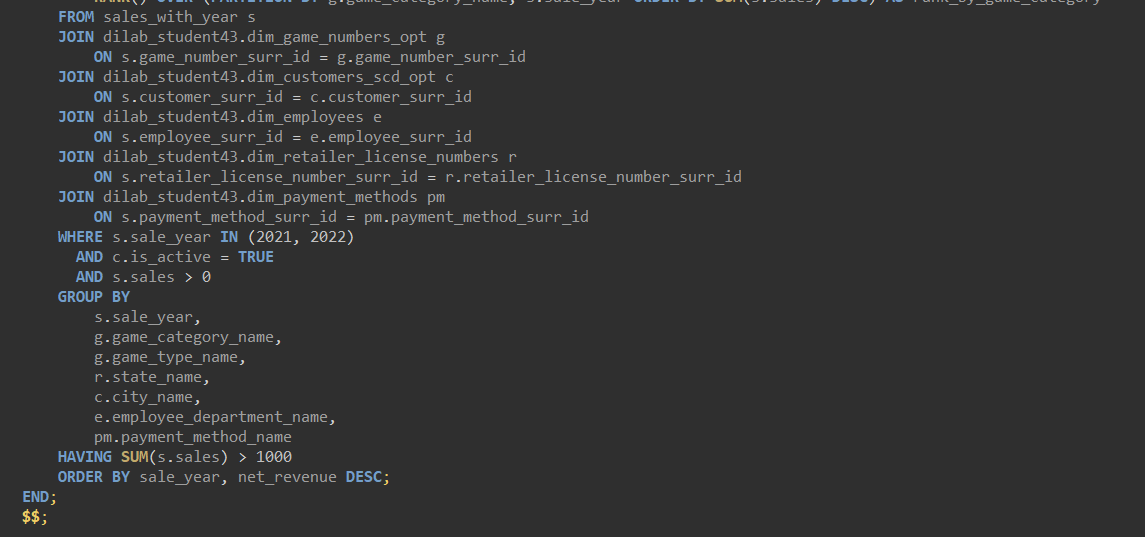
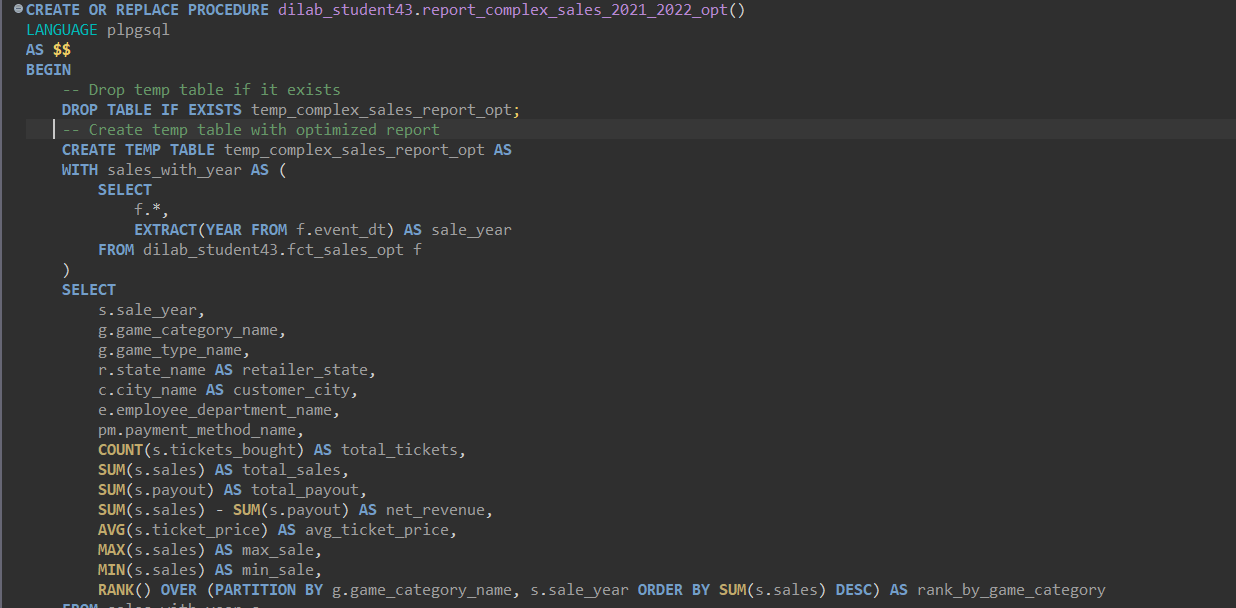
After I applied the optimized **DISTKEYs and SORTKEYs**, I recreated the tables and reloaded the data. Then I ran the stored procedure that executes the SELECT joining the three main tables and used **EXPLAIN** to review the new execution plan.

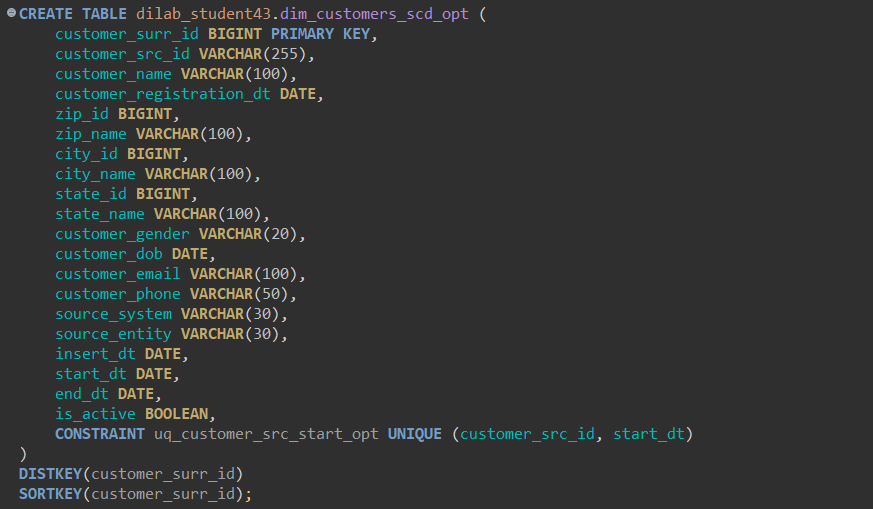
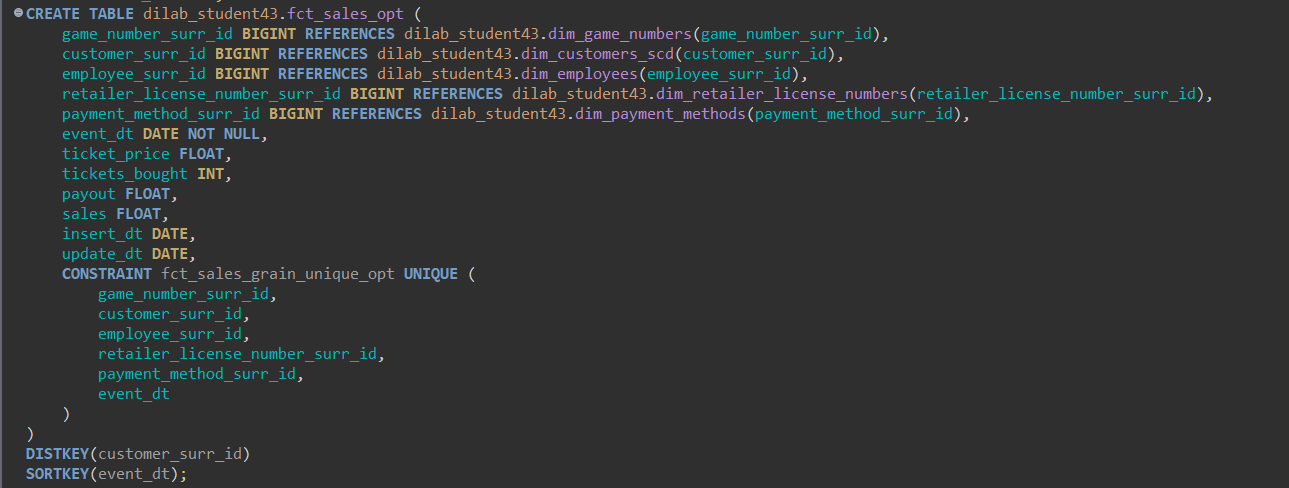
Before optimization, the plan showed multiple **DS\_DIST\_OUTER** or **DS\_DIST\_ALL\_NONE** steps, meaning that either the fact table or dimension tables were being redistributed across slices. Sequential scans on the large fact table were also required to apply filters on event\_dt and sales, which contributed to higher execution costs.

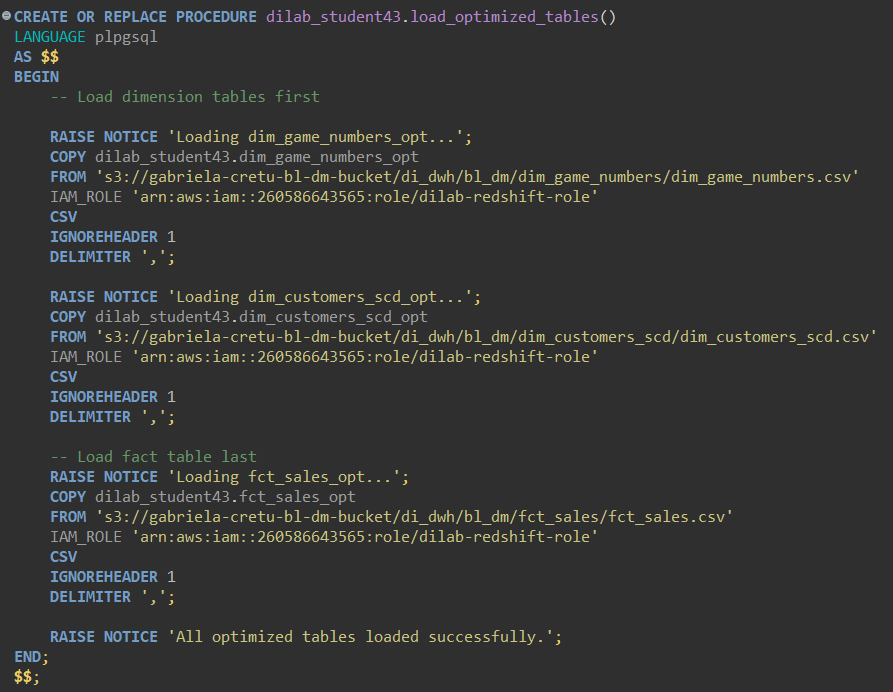
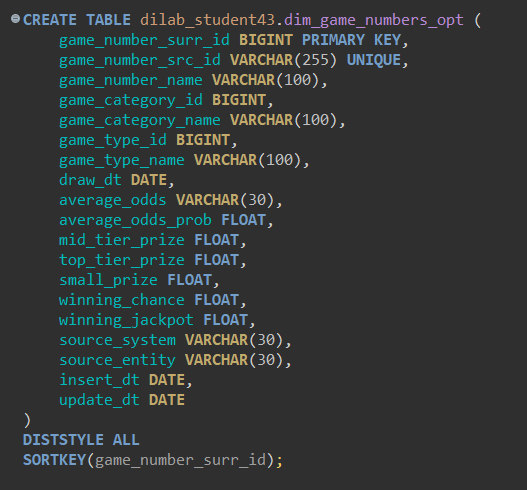
After optimization, I observed several improvements. The **hash joins** between fct\_sales and dim\_customers\_scd now happen locally on the same slices because the **DISTKEYs are aligned**, which reduces network data transfer. The small dimension table dim\_game\_numbers is broadcasted, so no redistribution is needed. The filter on event\_dt became faster because the fact table is sorted on this column, allowing **range-restricted scans** instead of full table scans.

As a result, query execution time was significantly reduced compared with the original runs. The execution plan now has **fewer redistribution steps**, more efficient **hash joins**, and lower memory overhead for aggregations and window functions.

***Procedures and tables definitions***





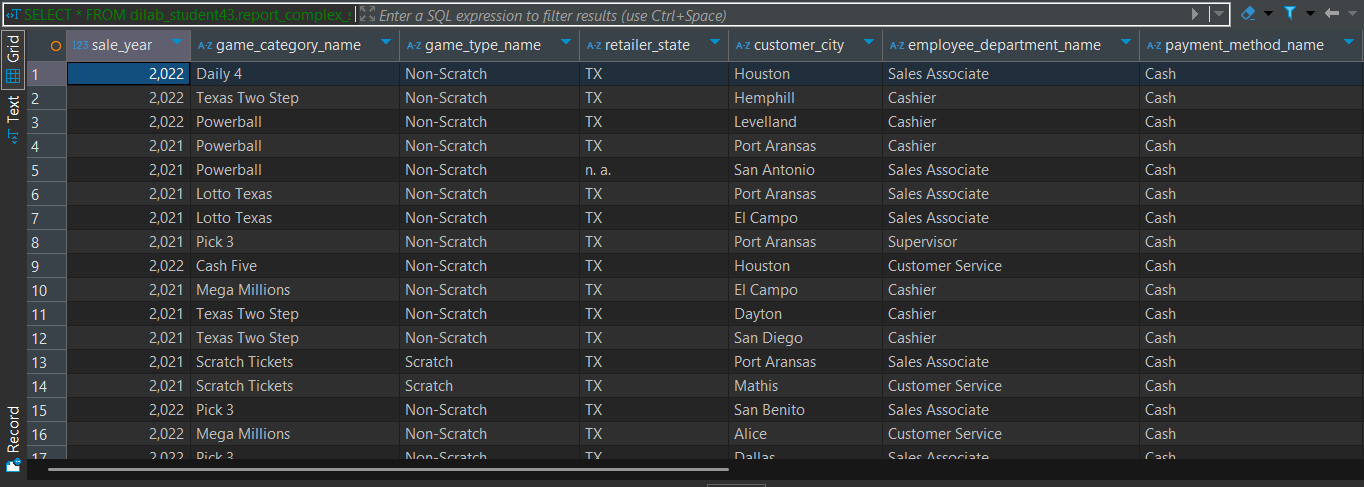
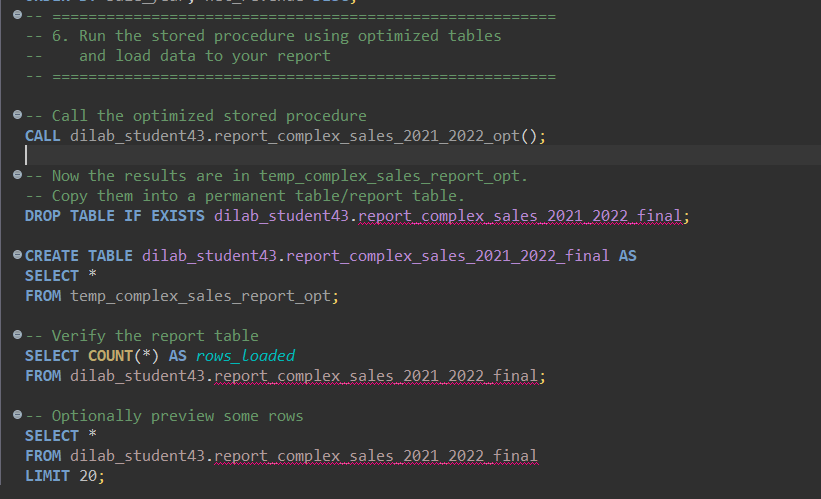


Before optimization, the join between fct\_sales and dim\_customers\_scd used **XN Hash Join with DS\_DIST\_OUTER**, meaning the fact table had to be redistributed across nodes based on customer\_surr\_id. This was very expensive in terms of network data movement. After optimization, both tables are distributed on the same key (customer\_surr\_id), so the join became **DS\_DIST\_NONE** — no redistribution was needed. This alignment is the biggest performance gain from the optimization.

Before optimization, joins with smaller dimension tables (dim\_employees, dim\_game\_numbers, dim\_payment\_methods, dim\_retailer\_license\_numbers) were all **DS\_DIST\_ALL\_NONE** or similar. After optimization, this did not change, and that’s actually good — small lookup tables are broadcasted to all slices, avoiding redistribution without penalty.

Previously, **XN HashAggregate** occurred after the redistribution, which required moving many rows across nodes before computing totals and averages. After optimization, aggregation happens closer to the local fact scan because of the aligned DISTKEY. Less data movement here translates directly to faster execution. Both before and after optimization, the plans end with **XN Sort + XN Window** to handle RANK() and ORDER BY. Sorting always has a cost, so this step remains necessary.

6. Run the stored procedure using optimized tables and load data to your report.



WORKING WITH EXTERNAL TABLES

1. Create external schema (user\_dilab\_student(1..32)\_ext) pointing to your location and create several external tables on your files. More info can be found here:

https://docs.amazonaws.cn/en\_us/redshift/latest/dg/c-getting-started-using-spectrum- create-external-table.html

https://docs.aws.amazon.com/redshift/latest/dg/c-spectrum-external-tables.html

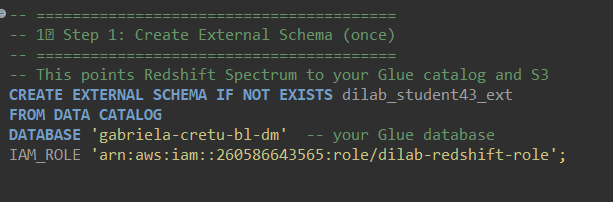
Example:

CREATE EXTERNAL SCHEMA if not exists user\_dilab\_student1\_ext

FROM DATA catalog

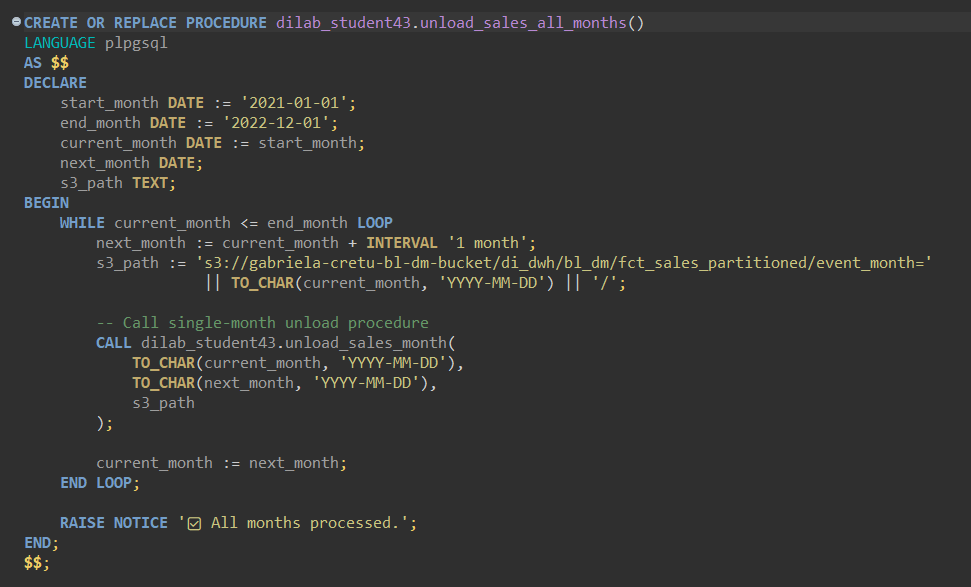
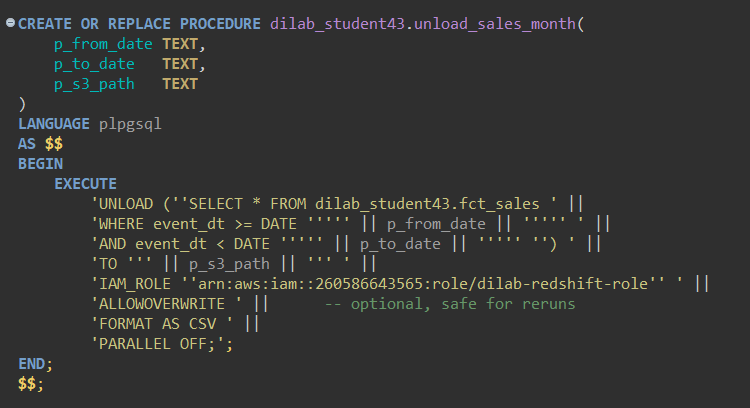
DATABASE ‘your\_glue\_database’

IAM\_ROLE 'arn:aws:iam:: 123456789102:role/rs-s3-role-read';



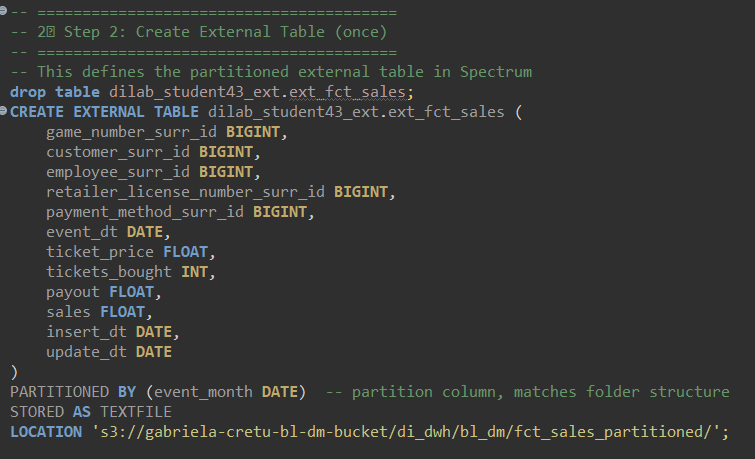
2. Partitioned external tables are extremely useful for the performance and cost cuts.

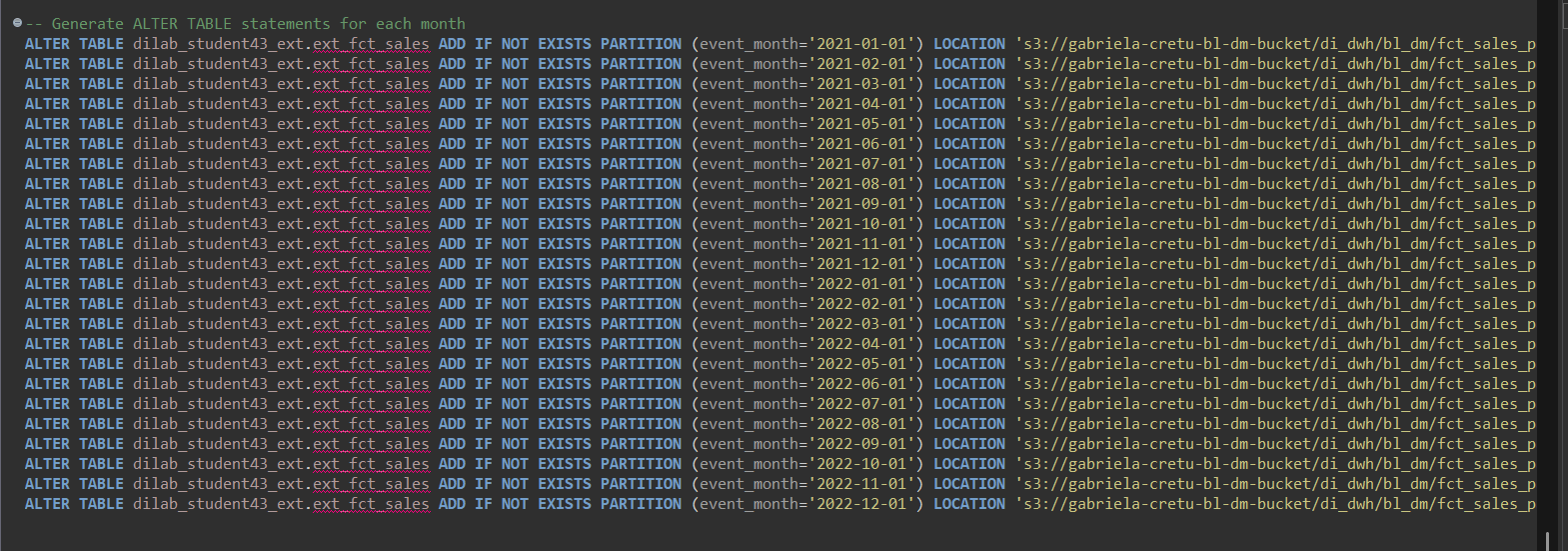
a. Export any data which contain date column into S3 in a way, so each subfolder contains 1 month data (e.g., subfolder /your\_date\_column=2018-03-01 contains all records of the table where your\_date\_column is within March 2018).



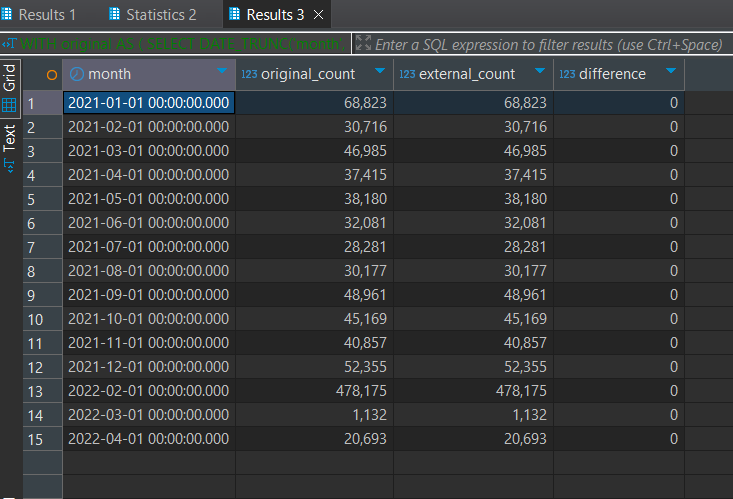
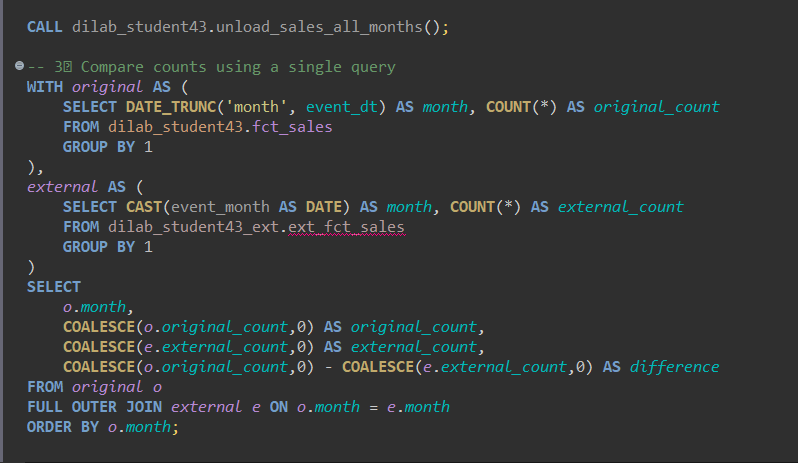
b. Create PARTITIONED external table “ext\_studentN\_partitioned” (partition by

your\_date\_column).

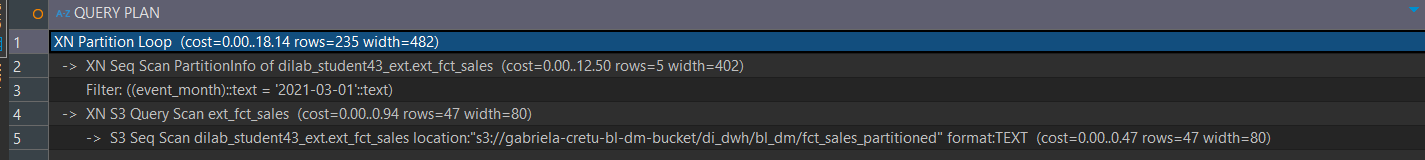




c. Verify data in partitioned external table (compare number of records per month to original table from which you prepared files or just with rows in the files). Prepare the test script that will show 0 difference (if it is not 0 – there is an issue).



d. Examine query plan where you select from ext\_studentN\_partitioned with a WHERE clause containing your\_date\_column condition. Describe it.



When I examine the query plan for selecting from ext\_studentN\_partitioned with a filter on my date column, I can see exactly how Redshift Spectrum handles partitioned external tables. At the top level, there is an **XN Partition Loop**, which iterates through the relevant partitions. In my case, the query only targets event\_month = '2021-03-01', so Redshift estimates processing only the rows in that partition.

Next, Redshift performs an **XN Seq Scan on PartitionInfo**, which reads the table metadata to identify partitions. The filter on the partition column (event\_month = '2021-03-01') is applied here, pruning all other partitions. Out of the 5 available partitions, only the one matching the filter is selected.

Finally, Redshift Spectrum executes an **S3 Query Scan / S3 Seq Scan**, reading only the files in the selected partition folder on S3. The table format (TEXT) matches the external table definition, and the estimated rows correspond to the actual data in that folder.

✅ **Key takeaway:** Redshift Spectrum does **not scan all S3 files**, only the folder corresponding to the filtered partition. Partition pruning ensures efficient query execution, minimal data scanned, and reduced costs. By first scanning metadata and then only the relevant S3 files, queries on partitioned external tables are much faster and more cost-effective than full table scans.