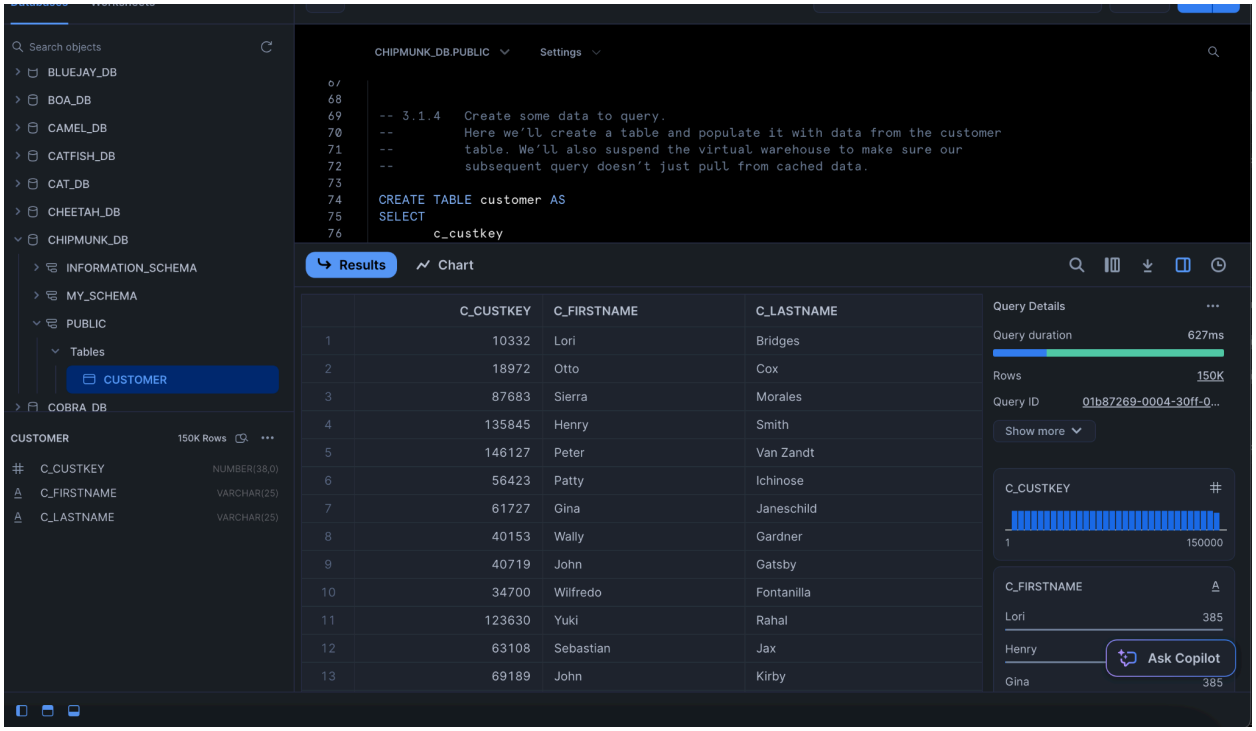
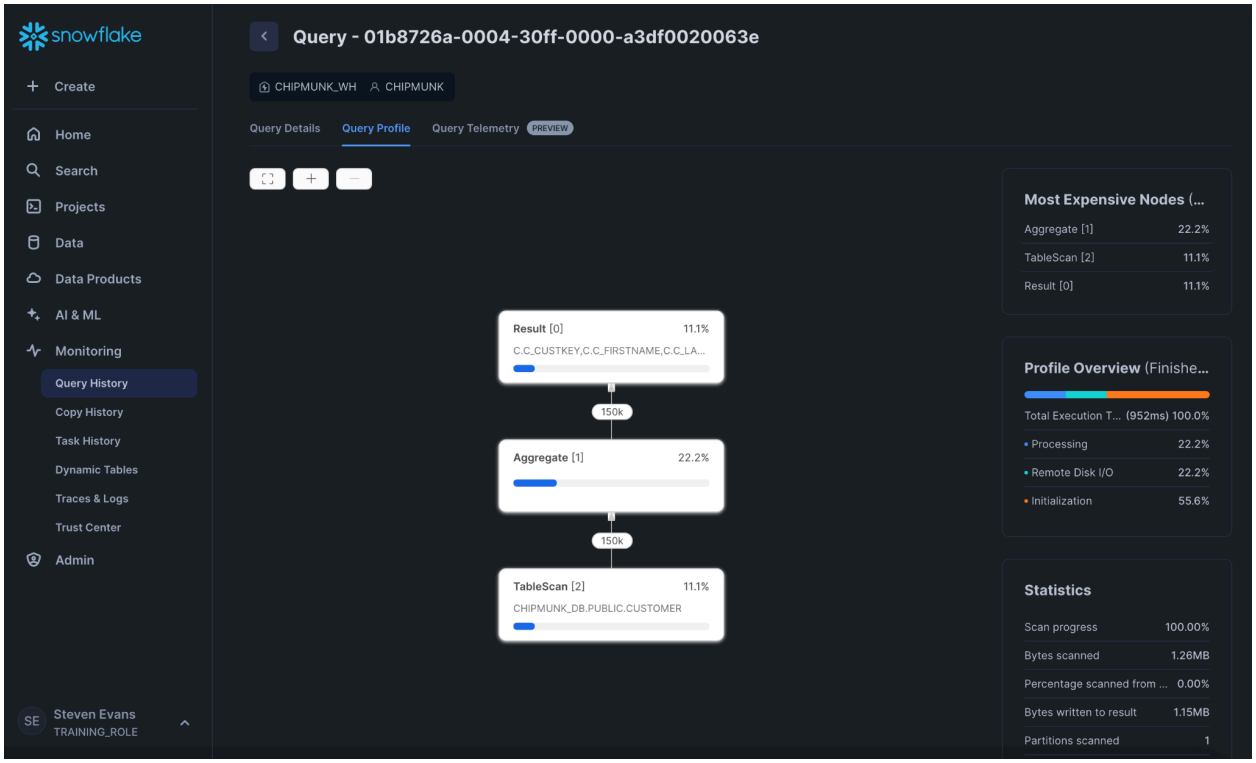
Steven Evans
SRE220000
BUAN 6335
Snowflake Lab 3

CUSTOMER TABLE CREATED IN SNOWSIGHT



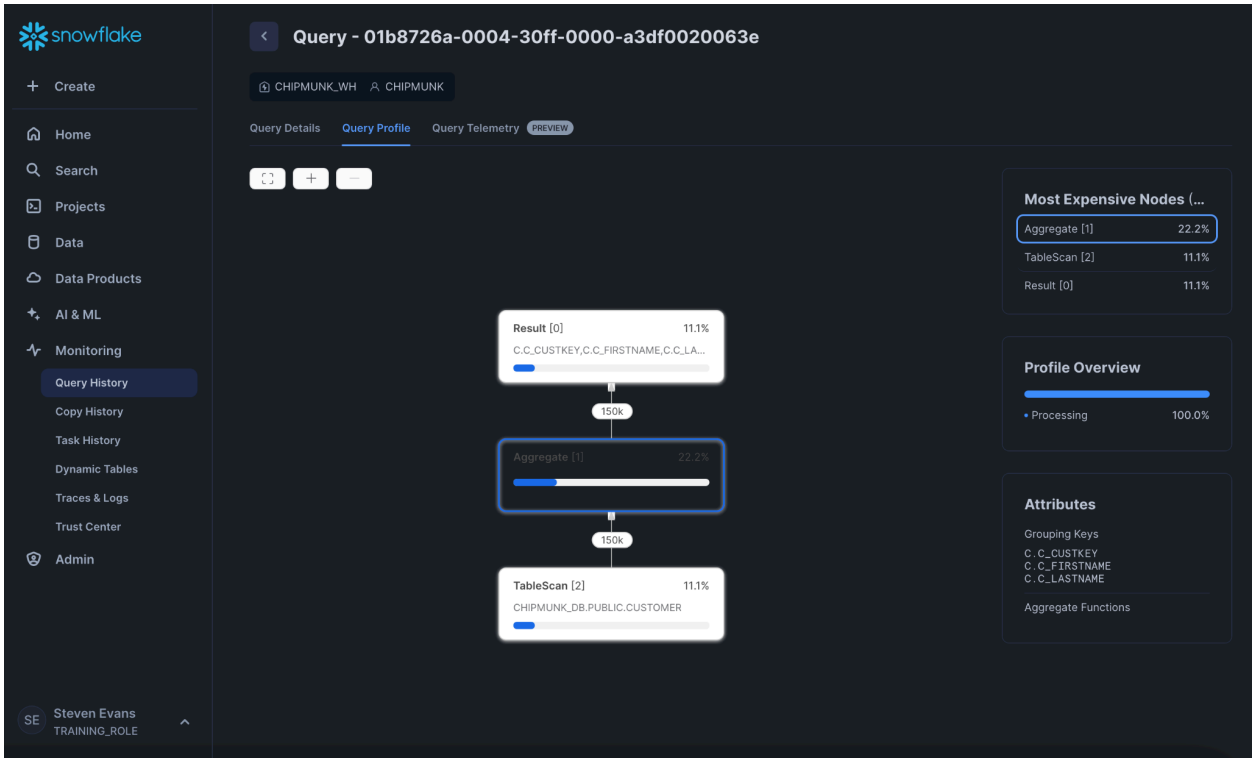
VIEWING THE QUERY PROFILE FOR SELECT DISTINCT * FROM CUSTOMER C;



Important Noticings:
22.2 % of the execution timer was done via processing the data, 22.2% was completed via Remote Disk I/O, and 55.6% was executed throughout the initialization. 1 partition was scanned and 1.15 Bytes were written to the result.

- **It is important to note that the Dick I/O is fairly low here indicating that this is the reason why the query did not take as long to run, if a more complex query were run and the time to complete were high, having a high Disk I/O time would be indicative that we need to do more filtering in the query to reduce the runtime.**

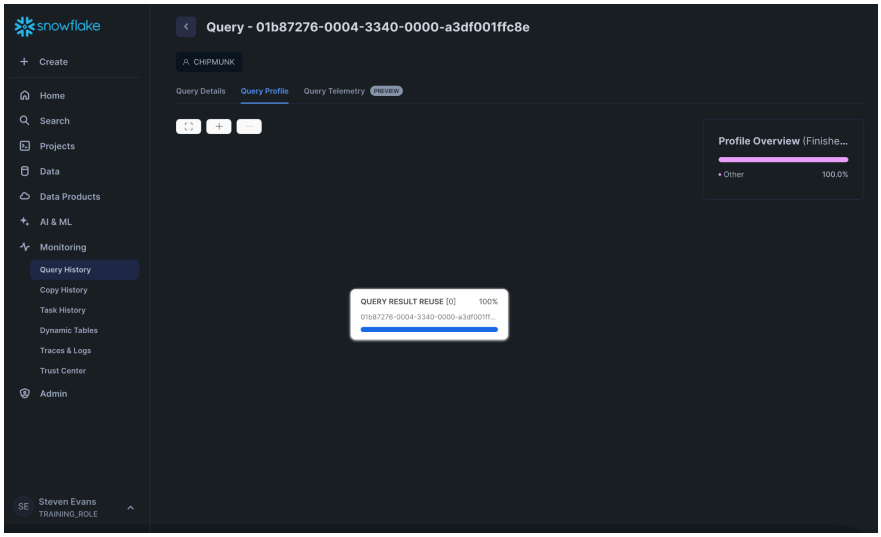
3.1 LOOKING AT THE DIFFERENT TABS IN THE QUERY PROFILE TO DISCOVER HOW THE QUERY WAS PROCESSED.



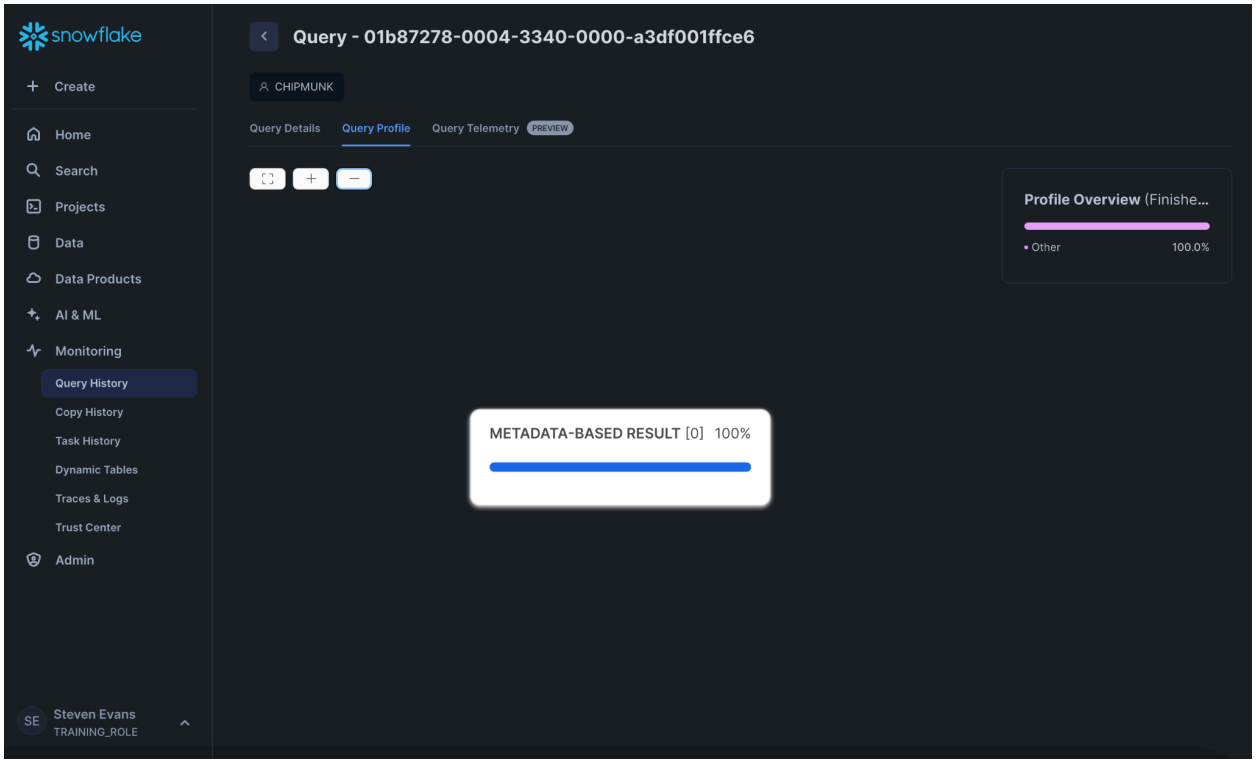
Important Noticings:

- **In the Aggregate tab 100% was done via processing, indicating that the 22.2% of the overall execution time is due to this single step**
- **In the Result and TableScan tabs 100% is due to Disk I/O indicating that the Disk I/O usage is due to producing results for the query and scanning the necessary table to find the correct information.**

After running the query again I can see that the metadat for the query was cached and reused. The faster execution time makes sense here



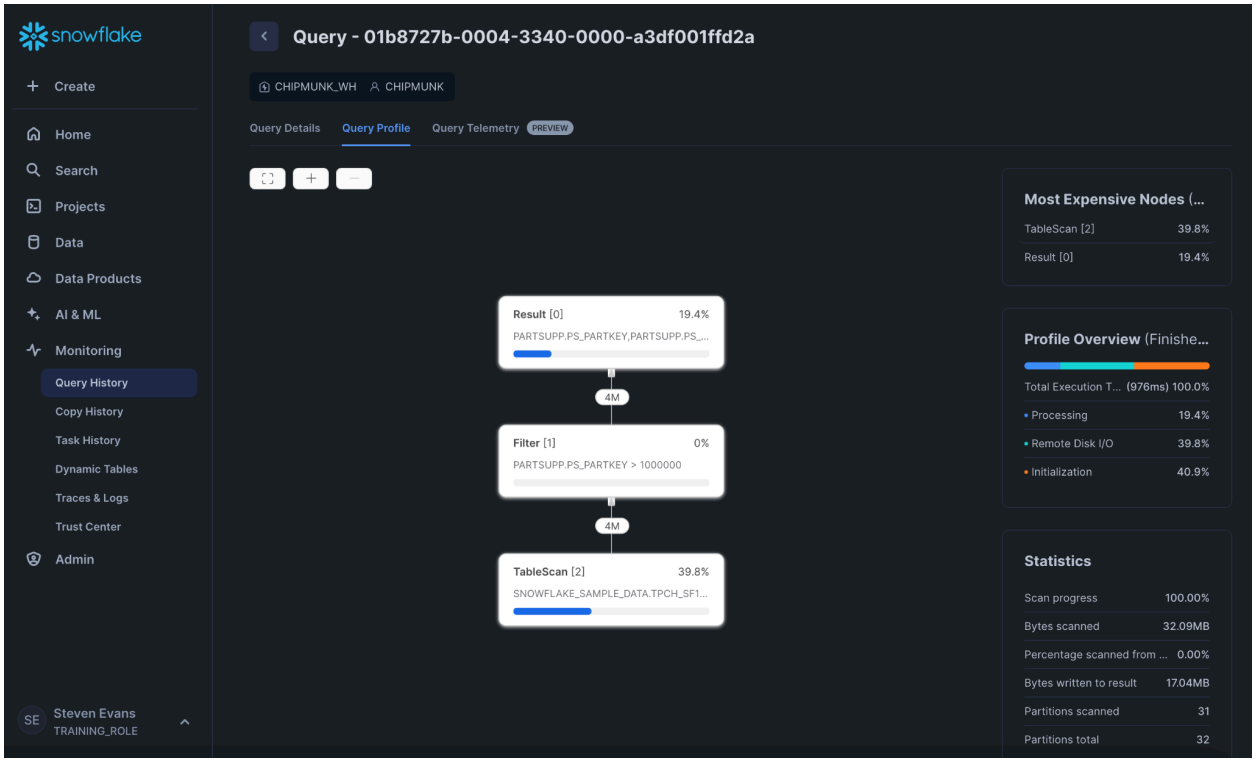
3.2 QUERY PROFILE FOR MIN AND MAX USAGE IN QUERIES



Important Noticings:

When running a query of this nature Snowflake executes the command based on the metadata stored in the cache. This shows that this information is stored within the database and utilizes no disk space.

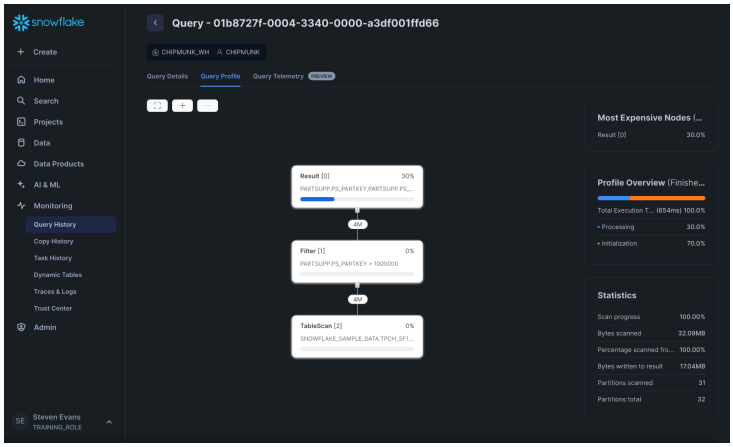
3.3 Running a new query using a where specific identifer



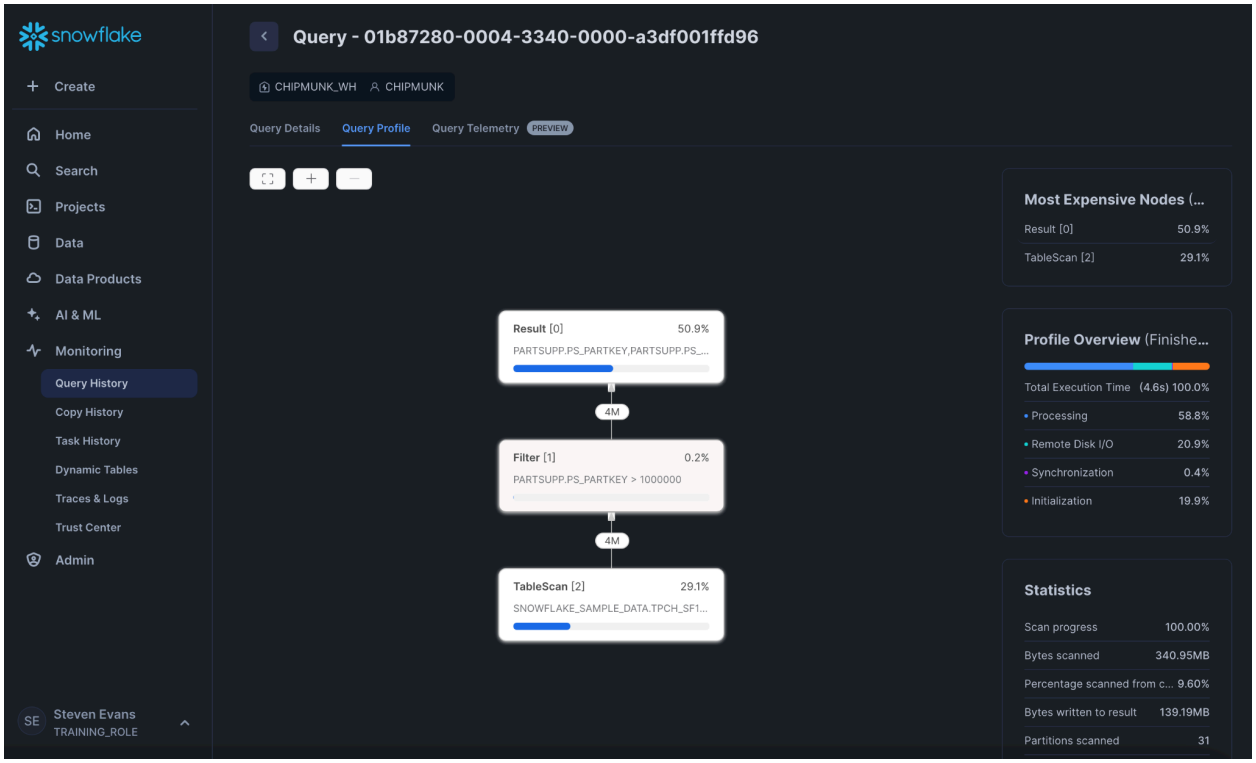
Important Noticings:

Here we can see that because this is a new query on a newly resumed virtual warehouse there was no caching used for its execution, There was some processing ran during the Result phase and Disk I/O used to scan the table.

- **After running the query again, there was no disk I/O used to scan the table only processing done on the result tab**



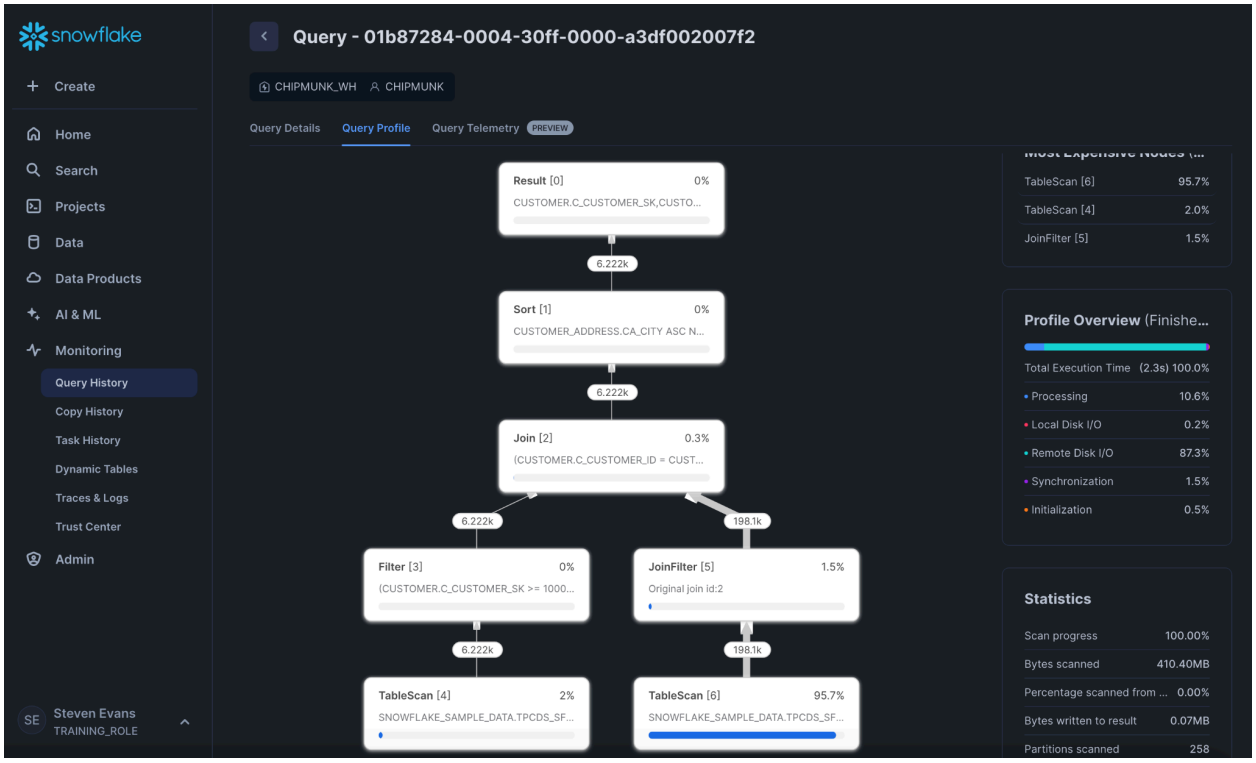
3.3.6 Running the same query but adding some columns



Important Noticings:

Upon adding some columns we can see that some Disk I/O space was used to execute the query. This explains why the query took much longer to run as opposed to the second run of the initial query where we pulled info from the metadata cache.

3.4 Running a query including joins of multiple tables and where clauses

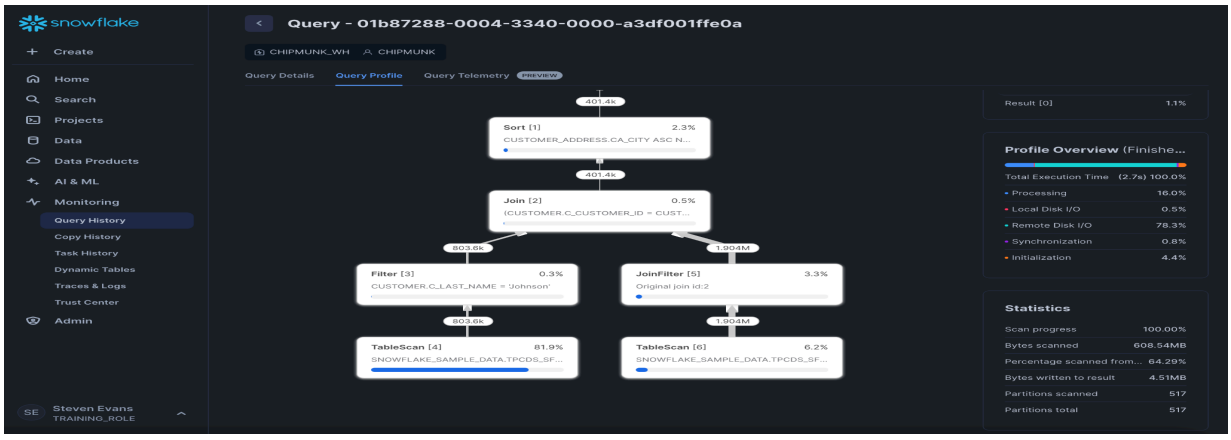


Important Noticings:

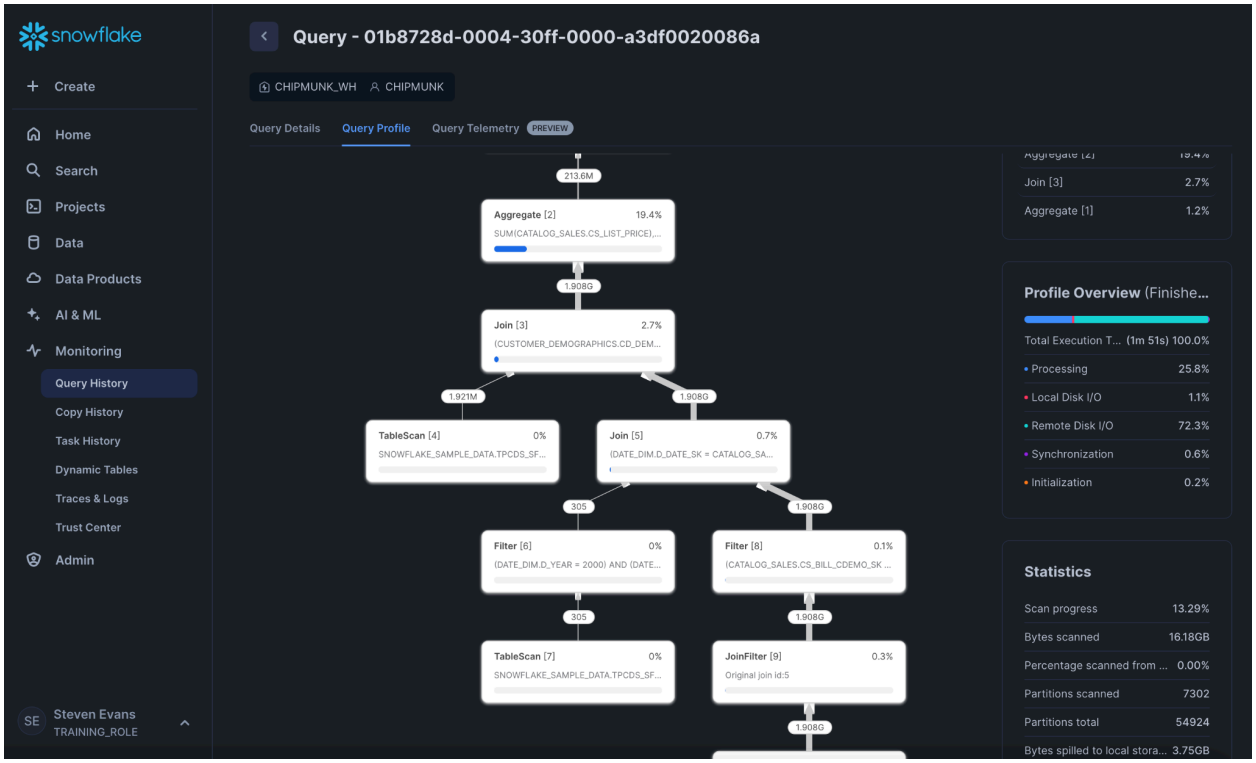
Only about half of the total partitions were scanned indicating that this query did not need to scan the entire database to find the necessary information to run the query.

If we look below at the second query without partition pruning using where clauses we can see that every partition was scanned to run the query and more remote disk I/O space was used in the process.

- **This demonstrates the importance of partition pruning and understanding how the table's data is organized so that when running queries we can optimize and reduce runtime and remote disk usage. This in turn will lower the overall cost of compute for an organization.**



3.5.3 Running a query that generates spillage

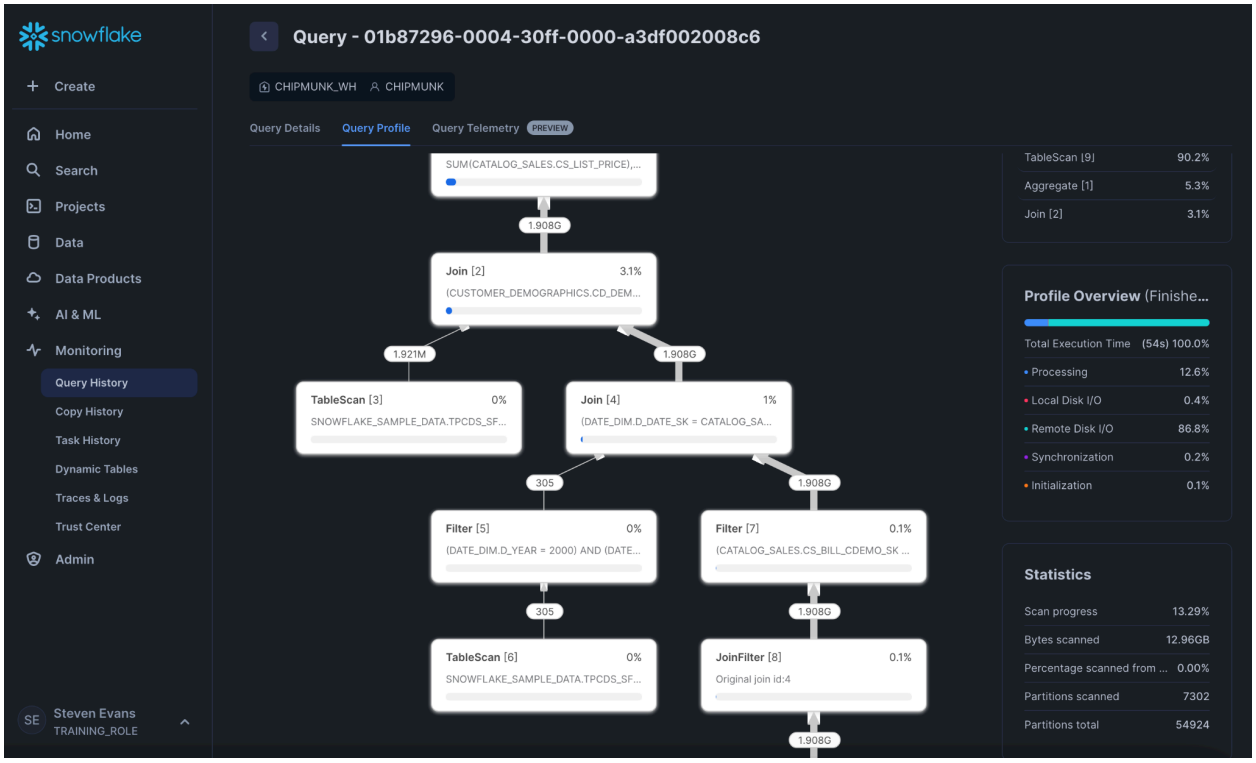


Important Noticings:

Because of the number of partitions scanned during this query we can see that 3.75 GB of bytes were spilled into local storage. This explains why the query took much longer to run than prior queries. The primary cause of this spillage within the queries are the aggregate nodes where multiple sum and counts aggregate functions were run for execution. Throughout the lab I will be focusing on different methods used to remedy this.

- **This query doesn't really need the outer query, we can remove it and remove the culprit of the spillage which is the cs_order_number column**

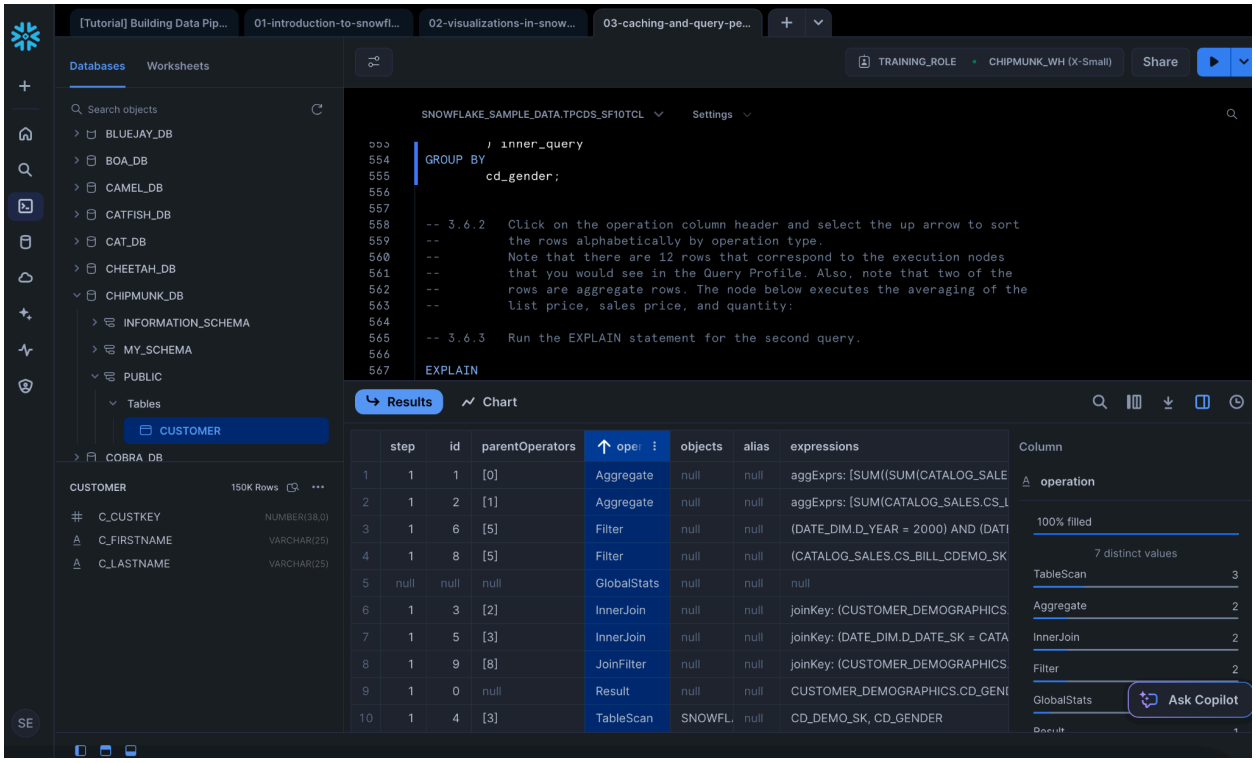
3.5.6 Remedying the spillage by removing the outer query and the cs_order_number column



Important Noticings:

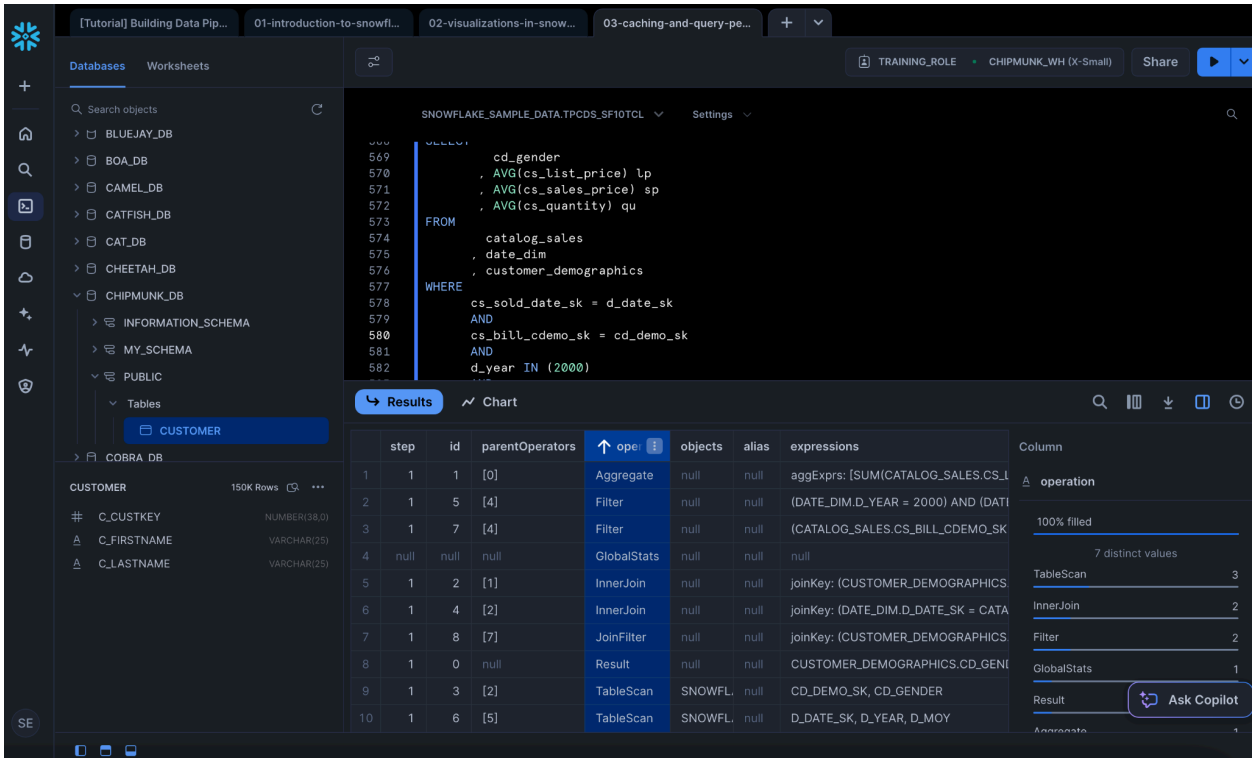
I can see how the removal of these two pieces of the query resulted in there being no spillage and still getting the same output from the query. The query also took around a minute less time to execute than the first. It is important also to note that there is only one aggregate node present in this query profile.

3.6 Running EXAPLIN on the query profiles to look at the different query nodes and understand their significance.



Important Noticings- First Query: Spillage

- **Based on the information presented in the output of the explain query I can see that there are two rows with aggregate expressions. This aligns with what I saw earlier in the query profile showing an unnecessary aggregate node from getting the average a second time.**



Important Noticings - Second Query: No Spillage

- **Here I can see that the aggregate node running the average a second time is no longer present. This shows the importance and functionality of the explain command to show the different nodes of the query and how it can be leveraged to optimize queries by removing bottlenecks in the query writing process.**