**Snowflake Optimization:**

When optimizing SQL queries, start with row-level operations, then move on to group-level operations, analytic functions, and finally, result generation. Focusing on the initial stages of the query process will have the most significant impact.

Order of Query Operations

**Now of Rows Extracted**

* From
* Join
* Where

**Grouping of Rows**

* Group by
* Having

**Analytic**

* Create Partitions (Clustered table,
* Order rows in partitions (Row\_number
* Function)
* Perform analytic calculation(sum,avg along with the order by )

**Result**

* Order by
* Select

**Optimize Query Logic:**

* Use CTEs (Common Table Expressions) or temporary tables for complex subqueries to simplify and potentially reuse results.
* Avoid unnecessary joins and instead use semi-joins or exists where possible.
* Use window functions judiciously; they are powerful but can be costly in terms of performance.
* Optimize Joins and Subqueries
* Use JOINs effectively by ensuring they are on indexed or clustered columns where possible.

**Best practices:**

* Utilize the ‘explain’ function to understand Snowflake’s query execution strategy. Data retrieval options include tabular format or JSON.
* Avoid using SELECT \* without a LIMIT clause for exploration queries about table data.
* Opt for selecting only the necessary columns rather than using “SELECT \*”.
* When using Sliding Window functions, avoid querying large volumes of data.
* Avoid using Complex Functions and UDFs in WHERE Clauses, as they can impact performance.Built-in functions and UDFs are useful but can impact performance when used in query predicates.
* Equality filter performs best because it looks for an exact match.
* Wildcard at the end of a string works better compared to a wildcard at the start of the string.
* Avoid using ORDER BY clauses in CTEs or Subqueries as they waste resources.
* Understand the relationships between your tables’ data before joining. Ensure keys are distinct (deduplicate) on joins to avoid Join Explosion and Cartesian Joins.
* Steer clear of Disjunctive Joins, where a query involves an OR clause for multiple join paths between two data objects (e.g., T1.COL1=T2.COL1 OR T1.COL1=T2.COL2). Instead, rectify this by crafting two separate queries that are subsequently combined using UNION
* Ensure that when joining two tables on a related column, the columns should possess identical data types. If one column has a numeric data type while the other is character, a data type translation issue may arise, leading to an unresolved join and continual resource consumption
* Use the QUALIFY clause in a SELECT statement to filter results of window functions, similar to how HAVING does with aggregate functions and GROUP BY clauses. QUALIFY is recommended for getting the most recent record based on key value.
* If you have a complex query with a large volume of data, materialize the intermediate result using a temporary table .It will help in most cases.
* Design CTEs and Subqueries to use caches to reduce performance issues, keeping in mind how the Result and Data Caches work.
* If an identical subquery exists within a transaction, leverage CTE for reuse.
* ORDER BY is an expensive operation. Use ORDER BY with LIMIT clause when looking for TOP N rows and validate through Explain.
* Query has group by columns with small number of distinct values is less memory intensive .Query has group by columns with a large number of distinct values and is more memory intensive. Try to reduce the distinct groups if possible