**Snowflake Optimization**

**Snowflake:**

Snowflake stands out from traditional databases and other cloud-based solutions due to its cloud-native design, scalable architecture, ease of use, and unique features like data sharing, time travel, and zero maintenance.

**SQL Query Tunning**

Focus on minimizing data scans, optimizing joins, using Snowflake's caching and built-in functions, and continuously monitoring query performance, you can significantly improve the efficiency and speed of your queries.

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

**JOIN**

**SELECT**

**WHERE**

**From**

**GROUP**

**HAVING**

**DISTINCT** 

**ORDER BY**

**Optimize Query Logic:**

* Use CTEs (Common Table Expressions) or temporary tables for complex subqueries to simplify and potentially reuse results.
* Use EXISTS or IN instead of JOIN for semi-joins

Example semi join

select pat\_enc.pat\_enc\_csn\_id, pat\_enc.pat\_id

from clarity.pat\_enc

where exists (

select 1

from clarity.pat\_enc\_hsp

pat\_enc.pat\_enc\_csn\_id = pat\_enc\_hsp.pat\_enc\_csn\_id)

* Use JOINs effectively by ensuring they are on indexed or clustered columns where possible.

**Optimize Join Order**

* Place the most selective tables first in the join order
* Use the JOIN ORDER hint to specify the join order

**Limit Subquery Results**

* Use LIMIT or TOP to restrict subquery results
* Avoid using SELECT \* in subqueries

**Use Window Functions**

* Use window functions judiciously; they are powerful but can be costly in terms of performance. Examples (SUM() , MAX() , COUNT() )- row-level calculations that would be less efficient with traditional aggregation methods.
* Use window functions instead of self-joins or subqueries
* Improve performance with efficient window function usage

**Avoid Using SELECT \***

* Only retrieve the necessary columns even in DEV or PROD
* Avoid unnecessary data retrieval and processing

**Best practices:**

* Utilize the ‘explain’ function to understand Snowflake’s query execution strategy. Data retrieval options include tabular format or JSON.

explain using text select pat\_enc.pat\_enc\_csn\_id, pat\_enc.pat\_id

from clarity.pat\_enc

where exists (

select 1

from clarity.pat\_enc\_hsp where pat\_enc.pat\_enc\_csn\_id = pat\_enc\_hsp.pat\_enc\_csn\_id);

* 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.
* An equality filter performs best because it searches for an exact match, making it more efficient compared to other types of filters

Example : where department\_id = 5

~~where department\_id like ‘% 5%’~~

* This type of filter is generally faster because it directly compares the value in the column to a specific value, making it straightforward for the database to process.
* Wildcard at the end of a string works better compared to a wildcard at the start of the string.

Example End: (LIKE 'App%') Start (LIKE '%Phone')

* 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.
* Refrain of Disjunctive Joins, where a query involves an **OR** clause for multiple join paths between two data objects
  + ~~(e.g., Pat\_enc.Pat\_id=Pat\_enc\_hsp. Pat\_id OR pat\_enc.pat\_enc\_csn\_id = pat\_enc\_hsp.pat\_enc\_csn\_id);~~
* Instead, rectify this by crafting two separate queries that are subsequently combined using UNION

select distinct pat\_enc.pat\_enc\_csn\_id, pat\_enc.pat\_id

from pat\_enc

join pat\_enc\_hsp on pat\_enc.pat\_enc\_csn\_id = pat\_enc\_hsp.pat\_enc\_csn\_id

union

select distinct pat\_enc.pat\_enc\_csn\_id, pat\_enc.pat\_id

from pat\_enc

join pat\_enc\_hsp on pat\_enc.pat\_id = pat\_enc\_hsp.pat\_id

* 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 Cast Function CAST(Pat\_ID as numeric)
* 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.

select dx\_id, pat\_id, sale\_amount

from Diagnosiseventfact **qualify** row\_number()

over (partition by product\_id order by sale\_amount desc) = 1;

* 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