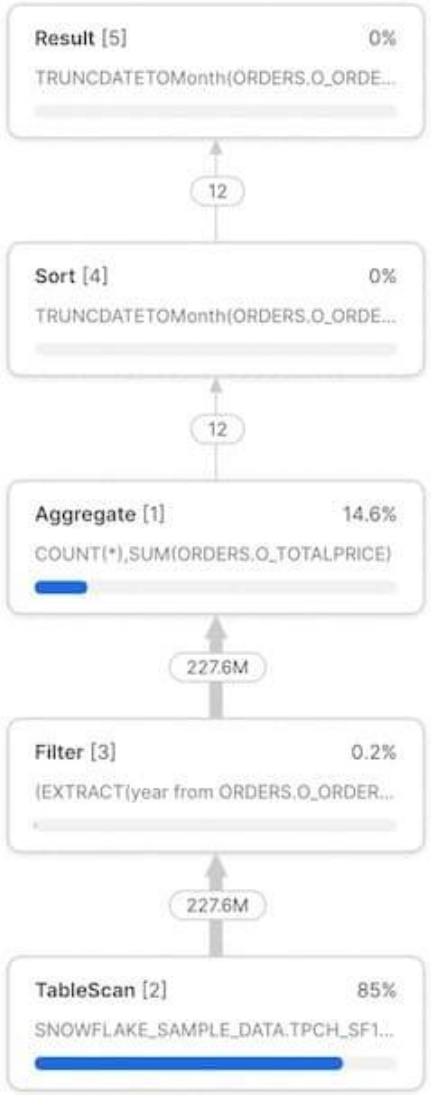
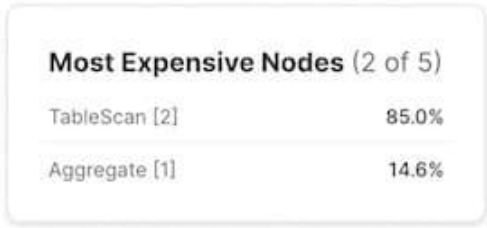
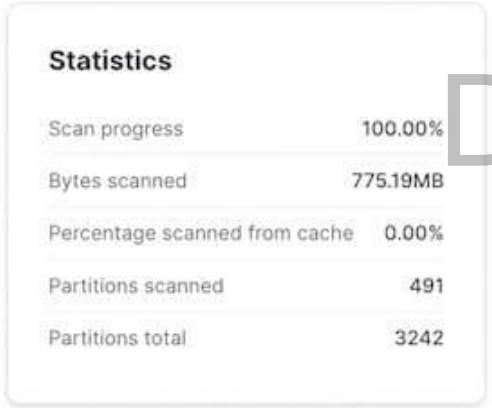
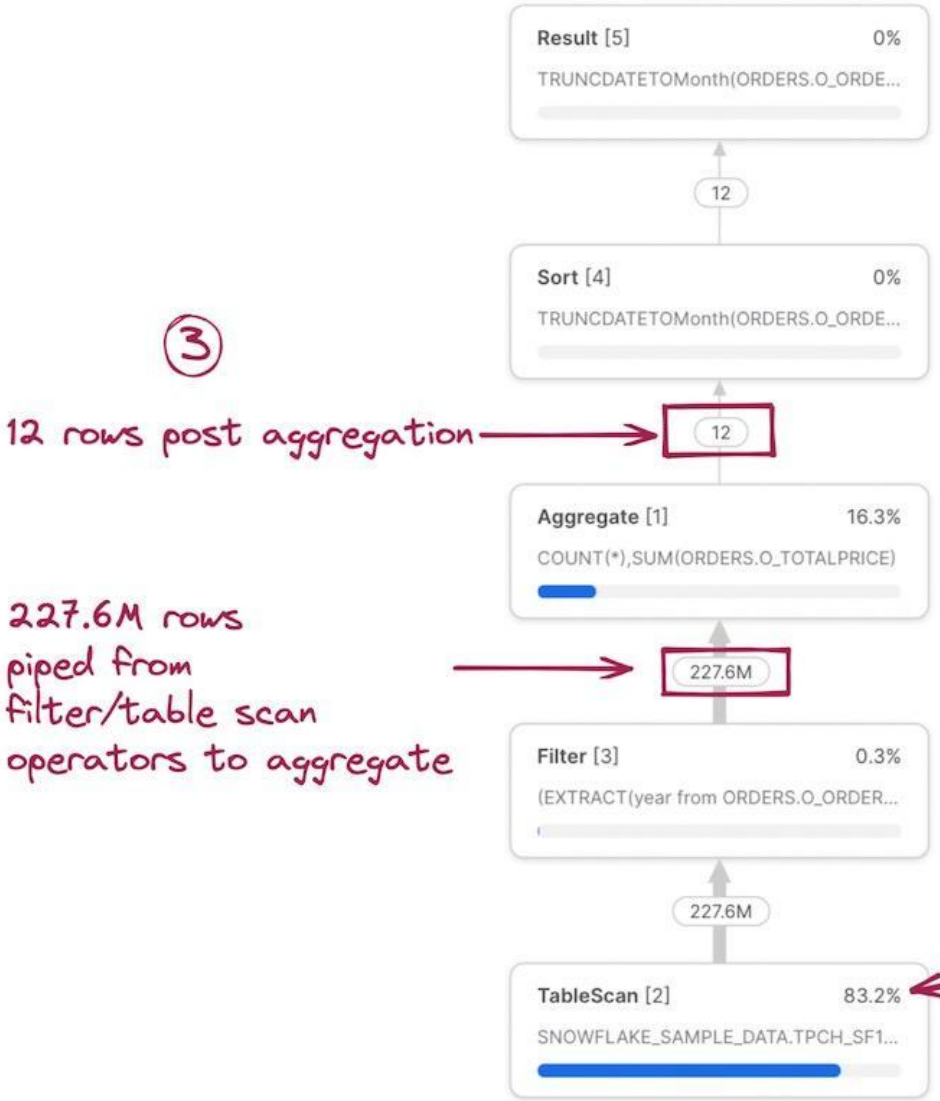
Query Profile

Query Profile is a feature in the Snowflake UI that gives you detailed insights into the execution of a query. It contains a visual representation of the query plan, with all nodes and links represented. Execution details and statistics are provided for each node as well as the overall query.





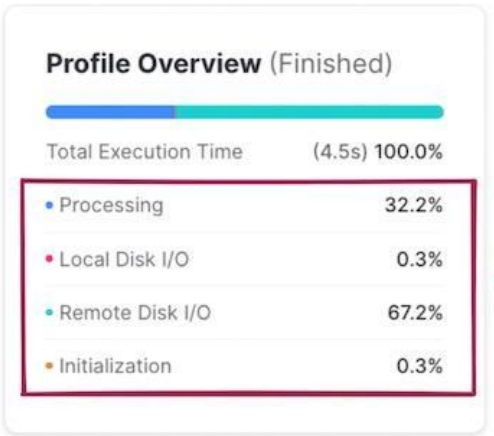




3

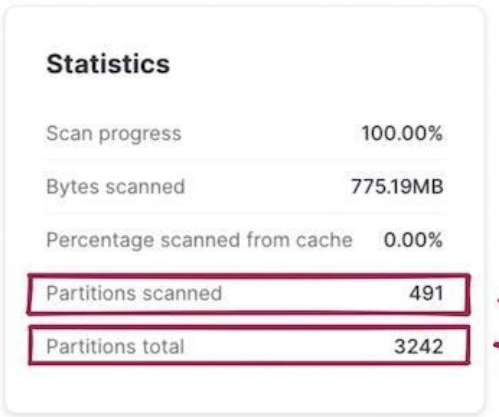
12 rows post aggregation

227.6M rows piped from filter/table scan operators to aggregate



1

Execution time summary



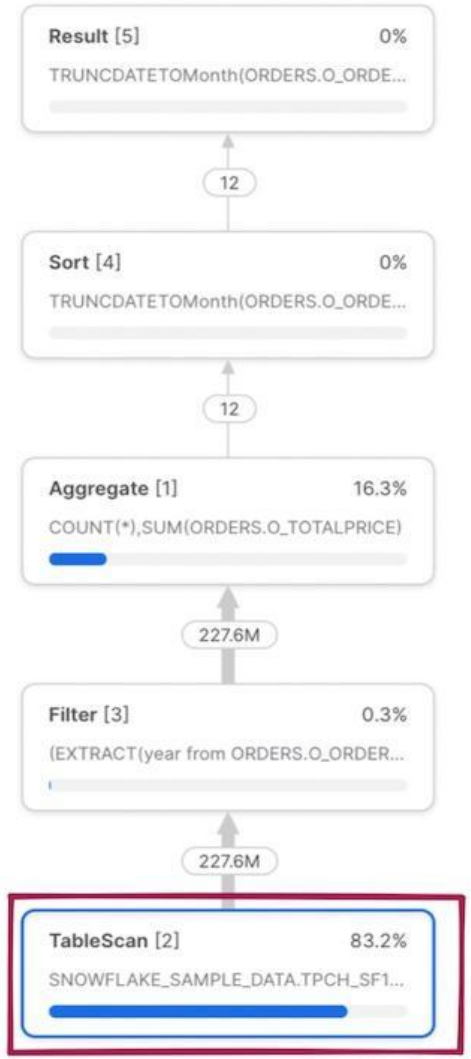
2

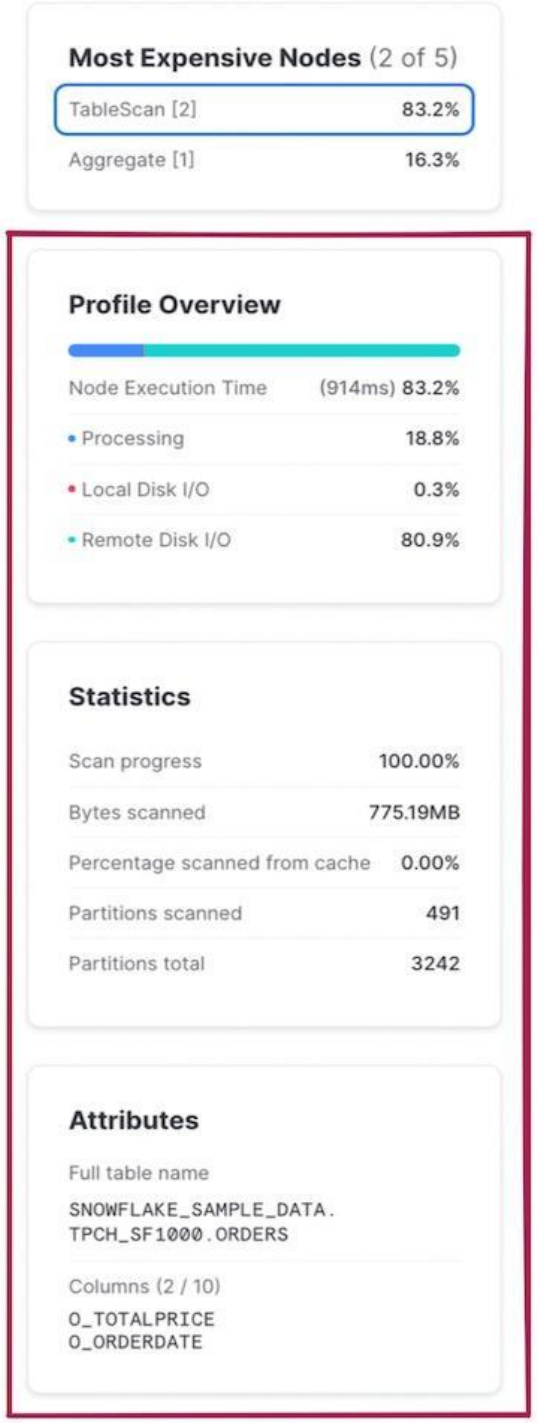
Actual partitions read

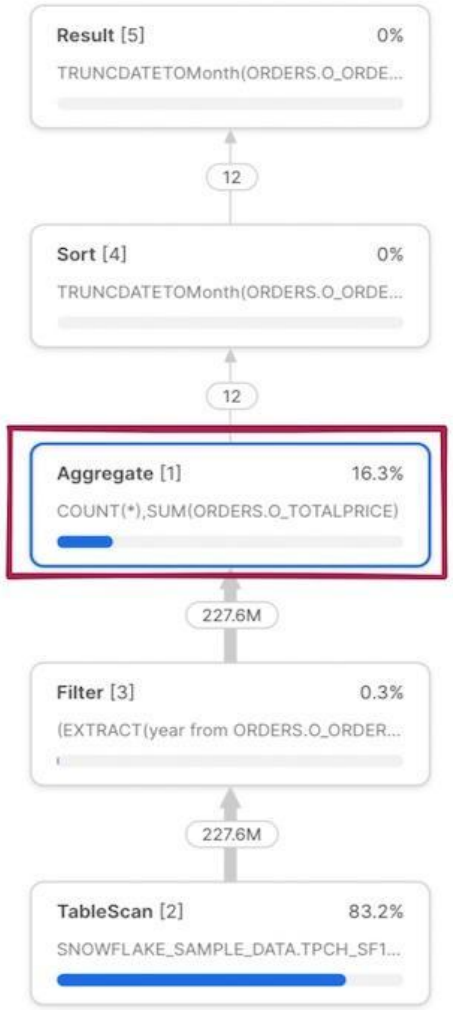
All partitions available (across all tables in query)

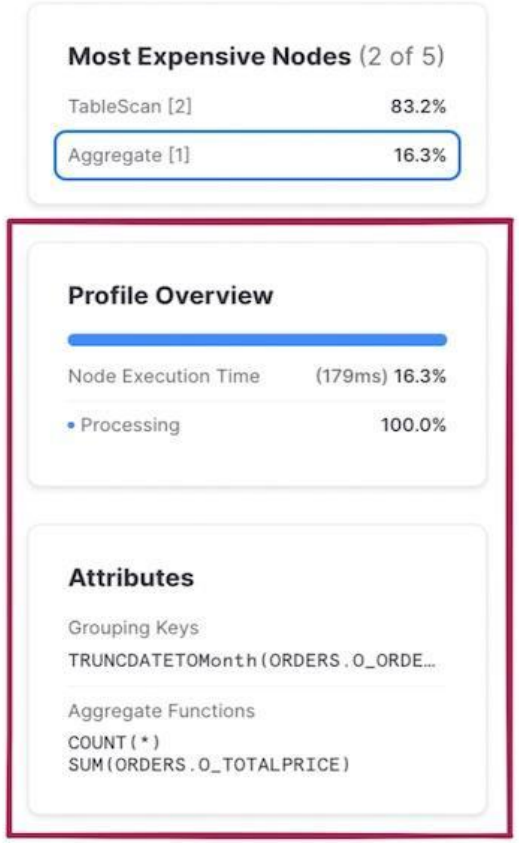
83.2% of total execution time spent on TableScan operator

4









Query Insights

Query Insights is a feature that builds on the Query Profile by automatically detecting conditions that might affect performance and offering actionable recommendations.

QUERY_INSIGHTS is account_usage view and can be used same way as other usage views. This view have specific columns that share information about queries, warehouse, and three key columns — insight_type_id, message, and suggestion.

We can use query_id to check if there any insights available in the table and get recommendations to improve performance.

For Example

```
SELECT query_id, insight_instance_id,insight_type_id, message,
suggestions
FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_INSIGHTS
WHERE query_id = '01bd1a3p-0203-9072-0000-
06172103b023';
```


Bytes spilled to storage

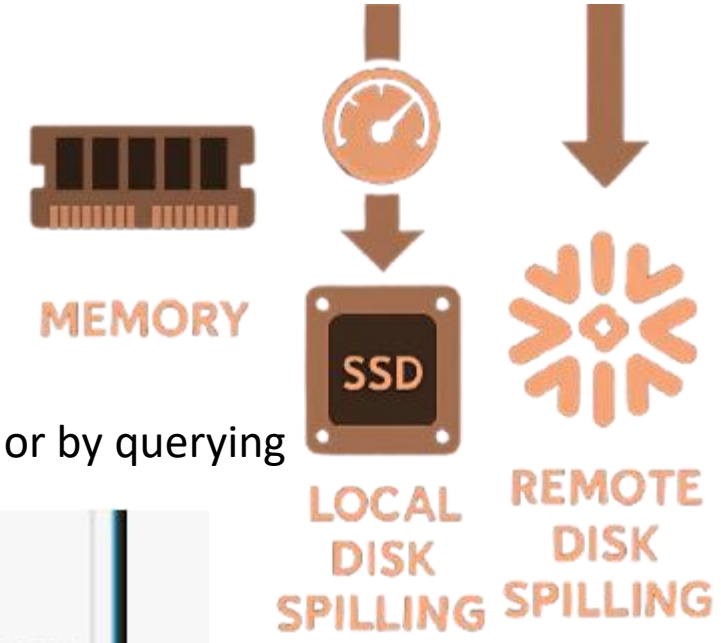
"Bytes spilled to storage" in Snowflake refers to the temporary data that a virtual warehouse writes to disk (local SSD or remote cloud storage) when an operation (like a large sort or join) runs out of available memory.
This spilling significantly degrades query performance because disk I/O is much slower than memory access.

Why Does Spilling Happen

Disk spilling is almost always about **resource pressure** — the data involved in operations like large `ORDER BY`, `GROUP BY`, or joins simply doesn't fit in memory.

Major causes include:

- **Large data operations:** Sorting, grouping, or joining massive datasets.
- **Warehouse size:** Using a small or overloaded virtual warehouse.
- **Concurrency:** Many queries run at once, competing for memory.
- **Suboptimal SQL:** Complex or inefficient query designs.



You can identify queries experiencing spillage using the Query Profile in Snowsight or by querying the ACCOUNT_USAGE.QUERY_HISTORY view.

Inefficient pruning

Inefficient pruning in Snowflake occurs when queries scan too many micro-partitions, often due to poor clustering, improper data types in filters (e.g., using `TO_CHAR` on dates), or using complex expressions in `WHERE` clauses. This causes high I/O, increased query time, and higher costs.

Common Causes and Solutions for Poor Pruning:

Weak Clustering: If table data is not sorted or is only partially sorted, Snowflake cannot efficiently skip irrelevant partitions.
Solution: Implement Automatic Clustering on frequently filtered columns.

Unsupported Filter Expressions: Functions or expressions applied directly to filter columns (e.g., WHERE YEAR(date_col) = 2024) prevent optimization.
Solution: Use direct column comparisons, such as WHERE date_col >= '2024-01-01' AND date_col < '2025-01-01'.

Subquery Predicates: Queries using WHERE x IN (SELECT ...) may not always prune effectively.
Solution: Use JOIN clauses or CTEs to allow for better partition elimination.

Unsupported Data Types: Using specific data types can sometimes limit metadata usage.

Frequent Small Updates: Frequent updates and deletes fragment data, making pruning less effective.

Without

Profile Overview (Finished)

Total Execution Time	(20m 58s)	100.0%
Processing		15.1%
Local Disk I/O		5.5%
Remote Disk I/O		79.3%
Synchronization		0.0%
Initialization		0.1%

Statistics

Scan progress	100.00%
Bytes scanned	348.36GB
Percentage scanned from cache	0.67%
Partitions scanned	300112
Partitions total	300112

With Pruning

Profile Overview (Finished)

Total Execution Time	(1.7s)	100.0%
Processing		4.3%
Remote Disk I/O		87.0%
Initialization		8.7%

Statistics

Scan progress	100.00%
Bytes scanned	23.56MB
Percentage scanned from cache	45.37%
Partitions scanned	4
Partitions total	300112

Exploding joins

In Snowflake, a **join explosion** (or row explosion) occurs when a join operation produces an unexpectedly and excessively large number of rows, far exceeding the input data size. This issue often leads to severe performance degradation, long-running queries, memory spillage, increased compute costs, and potentially query failures.

Common Causes of Join Explosions

Missing or Incomplete Join Conditions | **Duplicate Values in Join Keys** | **Non-Equi Joins** | **Complex Queries/Flipped Joins**

Queuing

Queuing in Snowflake occurs when a virtual warehouse has insufficient compute resources to handle the volume of incoming queries, causing them to wait in a FIFO (first-in-first-out) queue until capacity frees up. It is primarily caused by excessive concurrency or resource exhaustion, resulting in higher latency

Solutions:

- **Multi-cluster Warehouses:** Enable auto-scaling to add compute clusters automatically during high load.
- **Resize Warehouse:** Increase warehouse size to provide more resources per query.
- **Query Optimization:** Review query history to optimize poorly performing SQL.

SNOWFLAKE.ACCOUNT_USAGE views

QUERY_ATTRIBUTION_HISTORY view, tracks query-specific costs. This allows users to accurately measure and analyze the resource consumption of their queries, enabling more effective cost management and optimization.

The QUERY_ATTRIBUTION_HISTORY view in Snowflake is a valuable tool for gaining insights into the cost of individual queries executed within your account. It provides detailed information about the resource consumption of each query, enabling you to optimize performance and manage costs effectively.

Important Considerations:

When using the QUERY_ATTRIBUTION_HISTORY view, keep the following factors in mind:

Latency: The QUERY_ATTRIBUTION_HISTORY view may have a latency of up to six hours.

Role Requirements: You must have the USAGE_VIEWER database role to access this view.

Short-Running Queries: Queries that are shorter than approximately 100 milliseconds may not be included in the view due to the current limitations of per query cost attribution.

QUERY HISTORY

Query history : the **Snowsight UI**, the **Information Schema** (for real-time data), and **Account Usage** (for long-term auditing).

Feature	Information Schema (SQL)	Account Usage (SQL)	Snowsight UI
Retention	7 days	**365 days (1 year)**	14 days
Latency	Real-time (None)	45 min – 3 hours	Real-time
Best For	Troubleshooting current issues	Long-term auditing/billing	Visualizing query plans

Information Schema uses these table functions for **real-time** monitoring of queries from the last 7 days. Note that these require the `TABLE()` keyword.

For Example:

```
SELECT * FROM TABLE(INFORMATION_SCHEMA.QUERY_HISTORY(
    END_TIME_RANGE_START => DATEADD('hour',-1,CURRENT_TIMESTAMP()),
    RESULT_LIMIT => 100
    ));
```


Account Usage

For historical analysis up to **1 year**, use the `SNOWFLAKE` shared database. This is a view, so it doesn't need the `TABLE()` syntax, but keep in mind the data might be delayed by a couple of hours.

For Example:

```
SELECT query_id, query_text, user_name, total_elapsed_time / 1000 AS seconds
FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
WHERE start_time >= DATEADD('day', -30, CURRENT_DATE())
AND total_elapsed_time > 10000 -- more than 10 seconds
ORDER BY total_elapsed_time DESC;
```

Grouping similar workloads

1. The Functional Grouping Model

Loading (ETL/ELT): Group all COPY INTO or INSERT tasks here. These are usually credit-heavy and can run on a smaller warehouse for a longer time.

Business Intelligence (BI): Group tools like Tableau, PowerBI, or ThoughtSpot. These require low latency and high concurrency.

Transformation (dbt/Tasks): Group your scheduled SQL transformations. This allows you to track exactly how much it costs to "build" your data models.

Ad-Hoc/Sandboxing: Group your analysts and data scientists. Their queries are unpredictable and often need "Large" warehouses for heavy joins.

2. Technical Grouping Model

T-Shirt Sizing (Vertical Scaling)

Group workloads that need to process **massive volumes of data** (large scans/joins) onto a larger warehouse (e.g., Large or X-Large). This provides more memory and local SSD space to prevent "spilling" to remote storage.

Multi-Cluster (Horizontal Scaling)

Group workloads with **high user concurrency** (many small queries at once) onto a Multi-Cluster Warehouse.

- **Standard Scaling:** Adds clusters immediately when a queue forms. Best for BI groups.
- **Economy Scaling:** Only adds clusters if there’s enough work to keep it busy for 6 minutes. Best for background "batch" groups.

Query acceleration service

Queued

Queued

Queued

Queuing

Data-2-Dollar$

The Query Acceleration Service (QAS) in Snowflake is a feature that improves the performance of large, scan-intensive queries by offloading portions of the processing work to separate, shared compute resources. This allows the primary warehouse to focus on other tasks and can significantly reduce the execution time of eligible queries without requiring a full warehouse resize.

How It Works

Automatic Detection: Snowflake's optimizer automatically analyzes a query plan at runtime to determine if parts of the query (such as large table scans, filters, and aggregations) are eligible for acceleration.

Offloading Work: If eligible, these portions are offloaded to dedicated QAS serverless compute resources, which process the data in parallel.

Pay-per-Use: The service is billed separately from the main warehouse on a per-second basis, only consuming credits when actually used.

No Code Changes: It integrates seamlessly with existing setups, requiring no changes to SQL code or pipeline redesign.

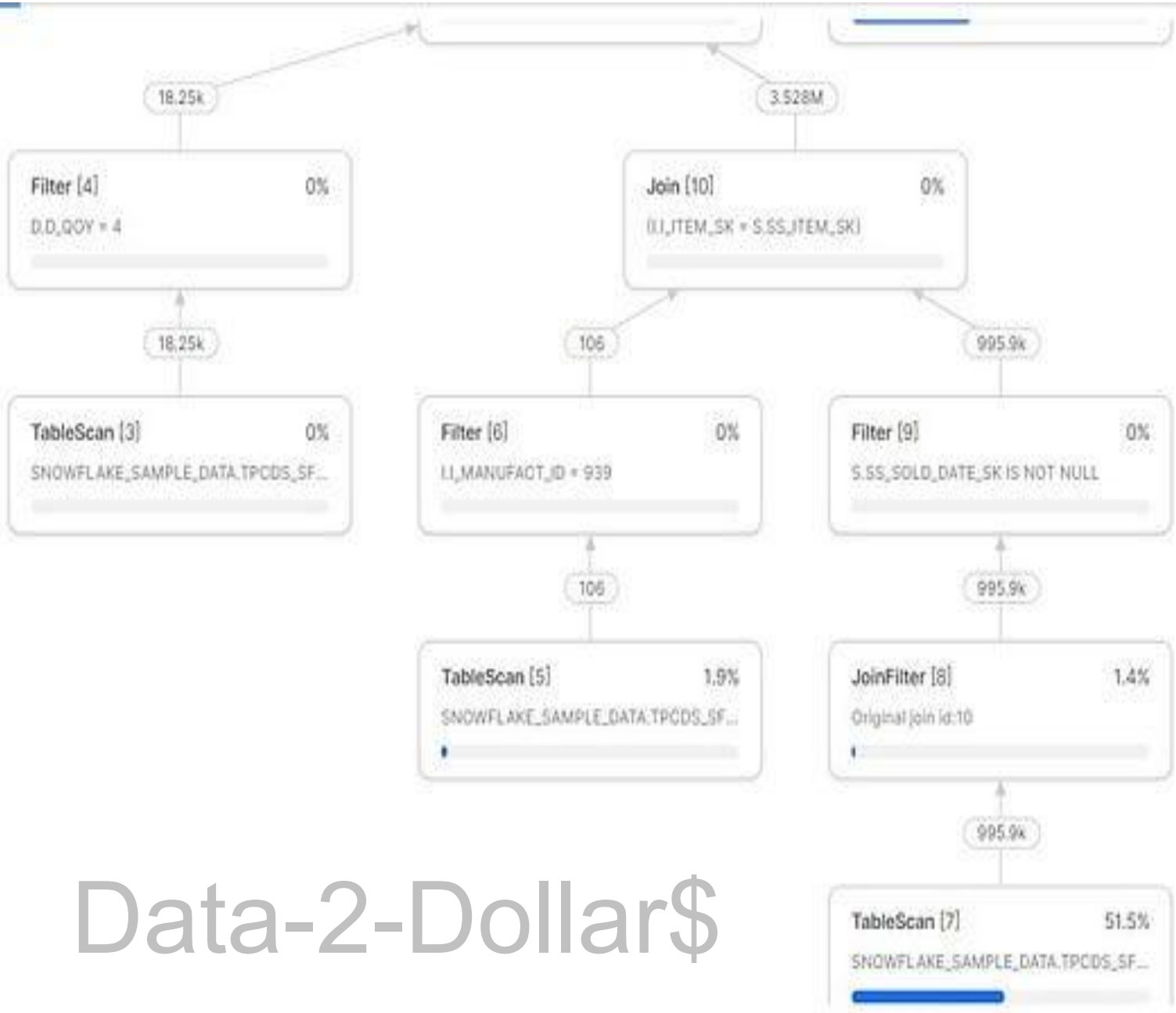
Enabling and Configuration

The service requires the Snowflake Enterprise Edition (or higher). It is enabled at the warehouse level using SQL commands:
Ex:

```
CREATE WAREHOUSE my_wh WITH ENABLE_QUERY_ACCELERATION = true;
ALTER WAREHOUSE my_wh SET ENABLE_QUERY_ACCELERATION = true;
```

Control costs with a scale factor: The **QUERY_ACCELERATION_MAX_SCALE_FACTOR** property limits the maximum amount of additional compute resources the service can lease. The default is 8 (max 16).
Ex:

```
ALTER WAREHOUSE my_wh SET QUERY_ACCELERATION_MAX_SCALE_FACTOR = 4;
```



Total Execution Time	(59s) 100.0%
• Processing	18.8%
• Local Disk I/O	10.8%
• Remote Disk I/O	62.5%
• Synchronization	7.8%
• Initialization	0.2%

Statistics

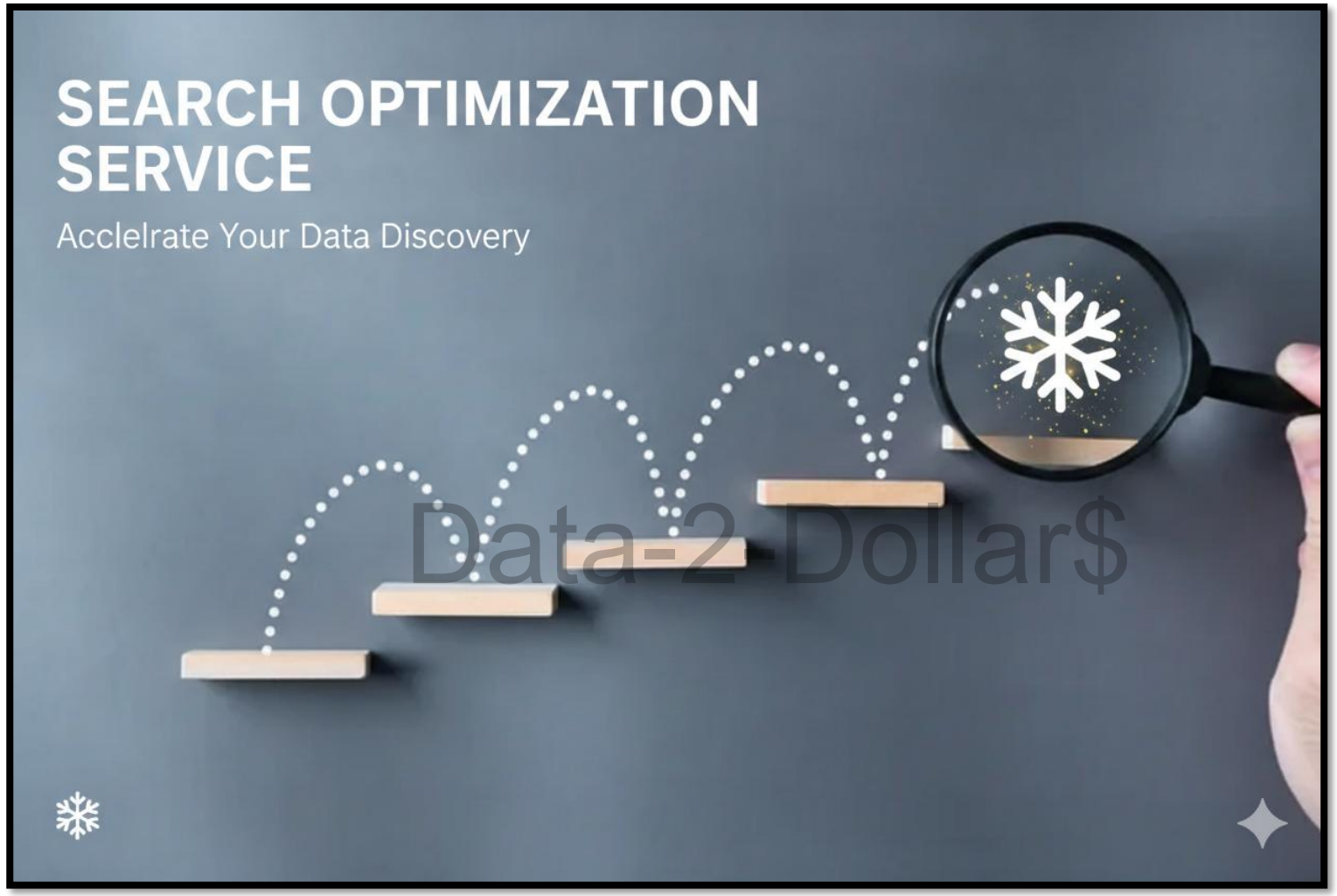
Scan progress	15.03%
Bytes scanned	23.28GB
Percentage scanned from cache	0.00%
Partitions scanned	13861
Partitions total	94560
Partitions scanned by service	57080
Scans selected for acceleration	2
Bytes spilled to remote storage	12.47MB

After QAS,Partition scan only 13K compared to 70K

QAS enabled

Data-2-Dollar$

Search optimization service



Snowflake's **Search Optimization Service (SOS)** is a feature designed to significantly improve the performance of specific, highly selective queries, often referred to as "point lookups" or "needle in a haystack" searches, on large tables.

```
ALTER TABLE customer_SOS ADD search OPTIMIZATION;
```

How It Works

The service operates by creating and maintaining a persistent, hidden data structure called a search access path. This path keeps track of which data values reside in which micro-partitions. During a query, the optimizer uses this path to prune (eliminate) unnecessary micro-partitions from the scan, dramatically reducing the amount of data read and speeding up the query.

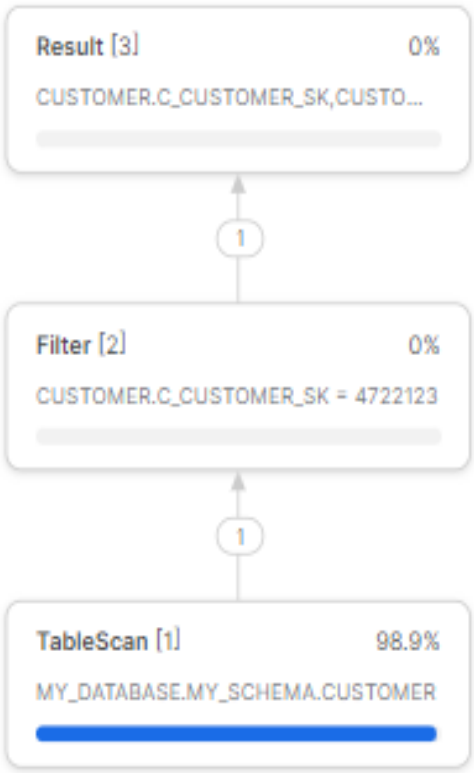
Key Features and Benefits

Improved Query Performance: It transforms slow, full table scans for specific values into fast, targeted lookups.
Automatic Maintenance: A background service automatically updates the search access path as data in the table is modified (INSERT, UPDATE, DELETE), requiring no manual index management from the user.
Broad Data Type Support: It works with structured data and semi-structured data (fields in VARIANT, OBJECT, and ARRAY columns).
Substring and Text Search: It can optimize queries using LIKE, ILIKE, RLIKE, CONTAINS, and other regular expression functions, including full-text searches with the SEARCH function.

Use Case Criteria

The service is most effective for:

- Queries that run for several seconds or longer.
- Highly selective filters that return a small number of rows (e.g., finding a single order by a specific, high-cardinality ID).
- Columns with a high number of distinct values (at least 100k-200k recommended).
- Large tables that are not clustered or are frequently queried on columns other than the clustering key.



Most Expensive Nodes (1 of 3)

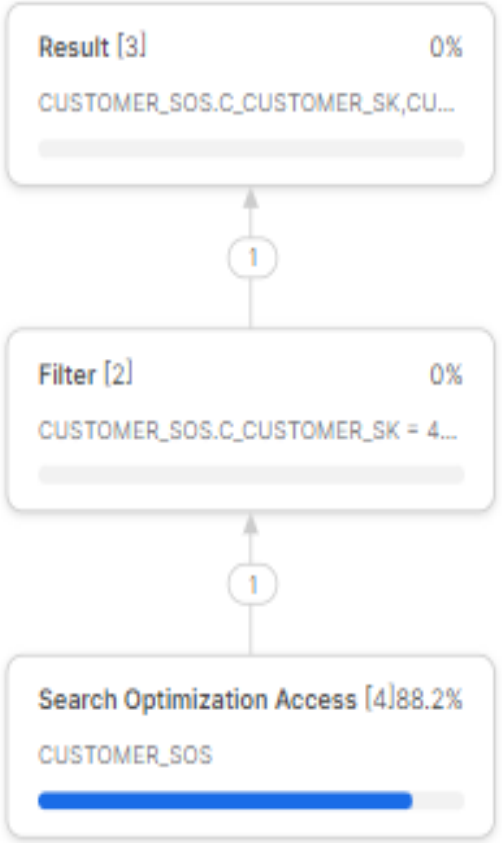
TableScan [1]	98.9%

Profile Overview (Finished)

Total Execution Time	(1.3s) 100.0%
• Processing	3.4%
• Remote Disk I/O	95.4%
• Initialization	1.1%

Statistics

Scan progress	100.00%
Bytes scanned	113.56MB
Percentage scanned from cache	0.00%
Partitions scanned	103
Partitions total	214



Most Expensive Nodes (1 of 3)

Search Optimization Access [4]	88.2%

Profile Overview (Finished)

Total Execution Time	(799ms) 100.0%
• Remote Disk I/O	88.2%
• Initialization	11.8%

Statistics

Scan progress	0.97%
Bytes scanned	8.38MB
Percentage scanned from cache	0.00%
Partitions scanned	1
Partitions total	214

Conclusion Difference

Feature	Primary Goal	Primary Query Type	User Effort
Search Optimization	Find specific rows faster	WHERE ID = 123 (Point Lookups)	Low (Enable on table)
Query Acceleration	Scale compute for huge scans	Massive SELECT * or large filters	Medium (Set scale factor)
Clustering Keys	Organize data for pruning	WHERE Date BETWEEN... (Range scans)	High (Requires design)

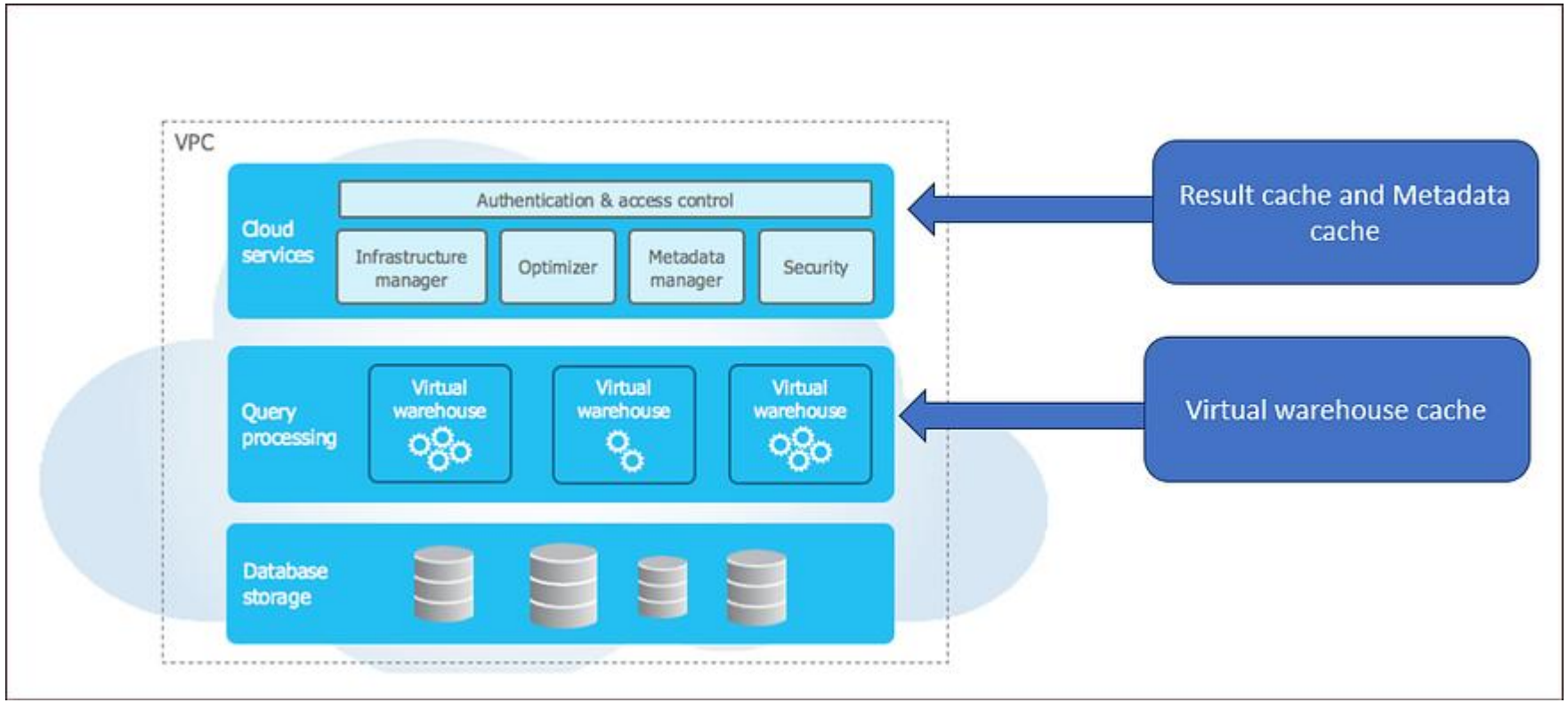
Use Snowflake caching

Three core caches in Snowflake :

1.Metadata Cache

2.Query Result Cache

3.Virtual Warehouse (Data) Cache



1. 🧠 Metadata Cache

Metadata Cache in Snowflake is a **system-managed cache** that stores **file-level and table-level metadata**. It resides in the **Cloud Services Layer** (control plane) and is automatically populated and maintained by Snowflake.

This cache includes metadata such as:

Micro-partition information	File locations in cloud storage	Row counts	Null value indicators	Column-level min/max values

How does it help?

When a query is submitted, Snowflake first consults the metadata cache before scanning any actual data. This allows it to:

1.Prune micro-partitions: If the metadata shows that certain partitions don't contain relevant data (e.g., for a specific date range or filter), Snowflake can skip reading those.

2.Avoid unnecessary I/O: Instead of scanning all files, Snowflake reads only what's necessary.

3.Speed up query planning: Metadata helps Snowflake's optimizer make intelligent decisions without accessing storage.

Key Benefits:

Performance Boost : Faster queries due to reduced data scanned

Reduced Costs : Less compute time = lower credits consumed

Zero Overhead : Managed by Snowflake, no tuning or memory allocation needed

Early Filtering : Enables partition pruning and filtering before compute kicks in

Always-on & Automatic : Built into the system — works across all queries without user action

2. 📦 Query Result Cache

The **Query Result Cache** in Snowflake stores the **final output** of a query — the actual result set returned to the user. When the **exact same query** is run again, Snowflake can **serve the results directly from this cache** instead of re-executing the query, **provided certain conditions are met**. It lives in the **Cloud Services Layer**, which means **no compute resources (virtual warehouses)** are involved when serving from this cache.

When is it used?

Here's what Snowflake checks before using the Query Result Cache:

- The **SQL text is identical**
- The **user role and permissions** are the same
- The **underlying data** (tables, views) **has not changed**
- The **query is run within 24 hours** of the last execution (default Time-to-Live)
- The query is not using **non-deterministic functions** like CURRENT_TIMESTAMP()

Key Benefits:

Zero Compute Cost : No virtual warehouse is used — results come from the control plane
Instantaneous Response : Time Results are returned almost instantly for cached queries
Cost Savings : No billing for compute means direct cost reduction
BI Tool Optimization : Great for dashboards repeatedly running the same queries
Reduced Load : Prevents reprocessing, freeing up resources for other tasks

3. ⚙️ Virtual Warehouse (Data) Cache

The **Virtual Warehouse Cache** — also known as the **Query Cache within the Virtual Warehouse** — is a **compute-level, in-memory cache** that stores **micro-partitions** recently read from **remote Snowflake storage (S3, Azure Blob, or GCS)** during query execution.This cache **resides within the virtual warehouse itself**, which is the compute engine running your SQL queries.

When is it used?

•When a query accesses table data, the warehouse fetches that data from storage.
•These data blocks are **temporarily stored in memory** on the **local nodes** of the virtual warehouse.
•If a subsequent query on the **same warehouse** accesses **overlapping data**, Snowflake can **re-use the in-memory data** instead of re-fetching it from storage.
•This reduces I/O latency and boosts performance.

❗ **Note:** This cache is **ephemeral** — it is lost if the virtual warehouse is:

•**Suspended**
•**Scaled up/down**
•**Restarted**

Key Benefits:

Faster repeated queries — Reduces the time for subsequent queries on overlapping datasets
Reduces storage I/O costs — Less data needs to be read from remote storage (Snowflake storage tier)
Boosts performance — Especially valuable for large table scans or aggregation workloads
Great for batch jobs — Ideal in ETL jobs or scheduled workflows using same compute node

Comparison Table: Snowflake Cache Types

Feature	Metadata Cache	Query Result Cache	Virtual Warehouse Cache
Location	Cloud Services Layer	Cloud Services Layer	Virtual Warehouse (Compute Layer)
What it stores	Table/file metadata (e.g. min/max)	Final query results	Recently accessed table data in memory
Used when	Table scan is needed	Same query re-run	Same warehouse, same data queried again
Valid if	Table hasn't changed	Query, role, and underlying table data haven't changed	Warehouse is still running (warm)
Compute needed?	❌ No	❌ No	✅ Yes
Performance impact	Prunes data scanned	Skips query execution entirely	Speeds up re-scans, avoids I/O
Persistence duration	Varies; managed by Snowflake	24 hours (default)	As long as warehouse is running
Benefit type	Reduced scan scope	Faster results, zero compute	Faster query execution
Best suited for	Any query involving large tables	Repeated queries from BI/dashboard	Transformation-heavy ETL workloads

Data transformation techniques using Structured, Semi-structured and Unstructured data

1. Structured Data (Traditional SQL): Transformation here usually involves **Standard SQL** and **DML** operations.

2. Semi-Structured Data (JSON, Avro, Parquet, XML)

Snowflake uses a special data type called **VARIANT** to store semi-structured data. You don't need to define a schema before loading it.

Transformation Techniques:

- **Dot Notation:** Access nested fields directly (e.g., `src:customer:id`).
- **FLATTEN:** A table function used to explode arrays into individual rows.

3. Unstructured Data (PDFs, Images, Audio, Logs)

Snowflake does not store the "content" of unstructured files in a table cell. Instead, it stores **Metadata** about the file and provides a **Directory Table** to manage them.

Transformation Techniques:

- **Scoped URLs:** Generate temporary, secure links to files for processing by external tools or Snowflake's **Snowpark**.
- **Snowpark (Python/Java/Scala):** Since SQL can't "read" a PDF, you use Snowpark to run Python libraries (like `PyPDF2` or `OpenCV`) directly inside Snowflake.
- **User-Defined Functions (UDFs):** Write a Python UDF to extract text from a blob and return it as a structured string.

Aggregate functions

Aggregate functions in Snowflake perform a calculation on a set of values and return a single value. These are essential for summarizing data and are almost always used in conjunction with a `GROUP BY` clause.

Category	Functions
Basic	SUM, AVG, MIN, MAX, COUNT
Statistical	MEDIAN, STDDEV, VAR_SAMP, CORR
String	LISTAGG
Estimation	APPROX_COUNT_DISTINCT, APPROX_PERCENTILE
Semi-Structured	ARRAY_AGG, OBJECT_AGG

Applying SQL for query optimization

1. The "Golden Rule": Pruning

Snowflake stores data in micro-partitions. Your goal is to write SQL that allows Snowflake to ignore as many partitions as possible. This is called **Pruning**.

🚫 The Optimizer's Enemy: Functions on Columns

If you wrap a column in a function in your `WHERE` clause, Snowflake cannot prune partitions efficiently.

- **Slow:** `SELECT * FROM sales WHERE YEAR(transaction_date) = 2024;`
- **Fast:** `SELECT * FROM sales WHERE transaction_date >= '2024-01-01' AND transaction_date <= '2024-12-31';`

2. Reduce the "Working Set"

Snowflake is a columnar database. Every extra column you select adds to the processing cost.

- **Avoid `SELECT *`**: Only pull the columns you need.
- **Filter Early**: Place your `WHERE` clauses as early as possible (especially in subqueries or CTEs) to reduce the volume of data passed to the next step.

3. Join Optimization & Deduplication(`DISTINCT` vs. `GROUP BY`)

Joins are often the most "expensive" part of a query. So prefer to use Hash Joins, also see the possibility of Join based on Cluster keys and avoid Cartesian products.

While **Deduplication** often yield the same result, `GROUP BY` is generally more efficient in Snowflake's engine for complex datasets because it allows for better parallel execution.

- **Instead of:** `SELECT DISTINCT user_id FROM logs;`
- **Try:** `SELECT user_id FROM logs GROUP BY 1;`

Applying SQL for query optimization

4. Use Result Caching

Snowflake has a **24-hour Result Cache**. If you run the exact same query and the underlying data hasn't changed, Snowflake returns the result instantly for **$0**.

•**Optimization Tip:** Avoid using non-deterministic functions like `CURRENT_TIMESTAMP()` or `UUID_STRING()` inside your query if you want to benefit from the cache. These functions make the query "new" every time it's run.

5. Sorting with `ORDER BY`

Sorting is a memory-intensive operation.

•**Only sort at the very end**: Don't use `ORDER BY` inside subqueries or CTEs unless you are using a `LIMIT`.

•**Use Top-N**: If you only need the top 10 rows, always use `LIMIT 10`. Snowflake will optimize the sort to stop once it finds the top 10.

Window Functions

Window functions are analytic functions that you can use for various calculations such as running totals, moving averages, and rankings.

Snowflake have a complete category of window functions like LAG,LEAD,ANY_VALUE,COUNT,LIST_AGG,RANK,DENSE_RANK,NTH_VALUE, and so on….

For More details: https://docs.snowflake.com/en/sql-reference/functions-window