Snowflake Table Structures

All data in Snowflake is stored in database tables, logically structured as collections of columns and rows. To best utilize Snowflake tables, particularly large tables, it is helpful to have an understanding of the physical structure behind the logical structure.

These topics describe micro-partitions and data clustering, two of the principal concepts utilized in Snowflake physical table structures. They also provide guidance for explicitly defining clustering keys for very large tables (in the multi-terabyte range) to help optimize table maintenance and query performance.

Query pruning, in the context of databases and data processing, refers to the optimization technique of eliminating unnecessary data access during query execution. It involves identifying and discarding partitions or segments of data that are irrelevant to the query based on filter conditions, thus reducing the amount of data that needs to be scanned or processed, which improves query performance and reduces resource consumption

Micro-partitions & Data Clustering

In contrast to a data warehouse, the Snowflake Data Platform implements a powerful and unique form of partitioning, called micro-partitioning, that delivers all the advantages of static partitioning without the known limitations, as well as providing additional significant benefits.

All data in Snowflake tables is automatically divided into micro-partitions, which are contiguous units of storage. Each micro-partition contains between 50 MB and 500 MB of uncompressed data (note that the actual size in Snowflake is smaller because data is always stored compressed). Groups of rows in tables are mapped into individual micro-partitions, organized in a columnar fashion. This size and structure allows for extremely granular pruning of very large tables, which can comprise millions, or even hundreds of millions, of micro-partitions.

Snowflake stores metadata about all rows stored in a micro-partition, including:

A. The range of values for each of the columns in the micro-partition.
B. The number of distinct values.
C. Additional properties used for both optimization and efficient query processing.

Benefits of Micro-partitioning

- ❖ In contrast to traditional static partitioning, Snowflake micro-partitions are **derived automatically**; they don’t need to be explicitly defined up-front or maintained by users.
- ❖ As the name suggests, micro-partitions are small in size (50 to 500 MB, before compression), which enables extremely efficient DML and fine-grained pruning for faster queries.
- ❖ Micro-partitions can overlap in their range of values, which, combined with their uniformly small size, helps prevent skew.
- ❖ Columns are stored independently within micro-partitions, often referred to as columnar storage. This enables efficient scanning of individual columns; only the columns referenced by a query are scanned.
- ❖ Columns are also compressed individually within micro-partitions. Snowflake automatically determines the most efficient compression algorithm for the columns in each micro-partition.

Impact of Micro-partitions

DML

All DML operations (e.g. DELETE, UPDATE, MERGE) take advantage of the underlying micro-partition metadata to facilitate and simplify table maintenance. For example, some operations, such as deleting all rows from a table, are metadata-only operations.

Dropping a Column in a Table

When a column in a table is dropped, the micro-partitions that contain the data for the dropped column are not re-written when the drop statement is executed. The data in the dropped column remains in storage. For more information, see the usage notes for ALTER TABLE.

Query Pruning

The micro-partition metadata maintained by Snowflake enables precise pruning of columns in micro-partitions at query run-time, including columns containing semi-structured data. In other words, a query that specifies a filter predicate on a range of values that accesses 10% of the values in the range should ideally only scan 10% of the micro-partitions.

For example, assume a large table contains one year of historical data with date and hour columns. Assuming uniform distribution of the data, a query targeting a particular hour would ideally scan 1/8760th of the micro-partitions in the table and then only scan the portion of the micro-partitions that contain the data for the hour column; Snowflake uses columnar scanning of partitions so that an entire partition is not scanned if a query only filters by one column.

In other words, the closer the ratio of scanned micro-partitions and columnar data is to the ratio of actual data selected, the more efficient is the pruning performed on the table.

For time-series data, this level of pruning enables potentially sub-second response times for queries within ranges (i.e. “slices”) as fine-grained as one hour or even less.

Not all predicate expressions can be used to prune. For example, Snowflake does not prune micro-partitions based on a predicate with a subquery, even if the subquery results in a constant.

🔍 What is a predicate?

In SQL, a predicate is a condition in a query that evaluates to true, false, or unknown. It's usually part of a WHERE clause or a JOIN condition. Example

```
age > 30
status = 'active'
order_date >= '2023-01-01'
```

📦 What is pruning in Snowflake?

Pruning in Snowflake refers to micro-partition pruning, a performance optimization where Snowflake skips scanning data that it knows won't match your predicate.

Snowflake stores table data in micro-partitions, and each partition has metadata (like min/max values for columns). If your predicate clearly rules out some micro-partitions, Snowflake skips reading them, making the query faster.

Example

```
SELECT * FROM orders
WHERE order_date >= '2023-01-01'
```

If some micro-partitions only contain data before 2023, Snowflake won't scan them. That's pruning.
❗ What the original statement means:

Not all predicate expressions can be used to prune. For example, Snowflake does not prune micro-partitions based on a predicate with a subquery, even if the subquery results in a constant.

Snowflake can't always "see through" a subquery to determine if the predicate is constant. So even if the subquery evaluates to a constant, pruning may not happen, leading to slower queries.
📘 Example: Let's say you have a table orders with millions of rows partitioned by order_date.
✅ Pruning works:

```
SELECT * FROM orders
WHERE order_date >= '2023-01-01'
```

Snowflake sees the date constant '2023-01-01' and skips partitions where all order_dates are earlier. Fast!
❌ Pruning may not work:

```
SELECT * FROM orders
WHERE order_date >= (SELECT MIN(created_at) FROM customers)
```

Even if the subquery returns '2023-01-01', Snowflake may not prune because:

The subquery hides the constant.

Snowflake treats it as dynamic at query planning time.

As a result, more partitions get scanned, and the query is slower.
✅ Tip: To enable pruning, move the subquery result into a constant using a common table expression (CTE) or run it separately and hard-code the value.

Example with CTE (sometimes helps):

```
WITH date_cutoff AS (
  SELECT MIN(created_at) AS cutoff FROM customers
)
SELECT * FROM orders, date_cutoff
WHERE order_date >= cutoff
```

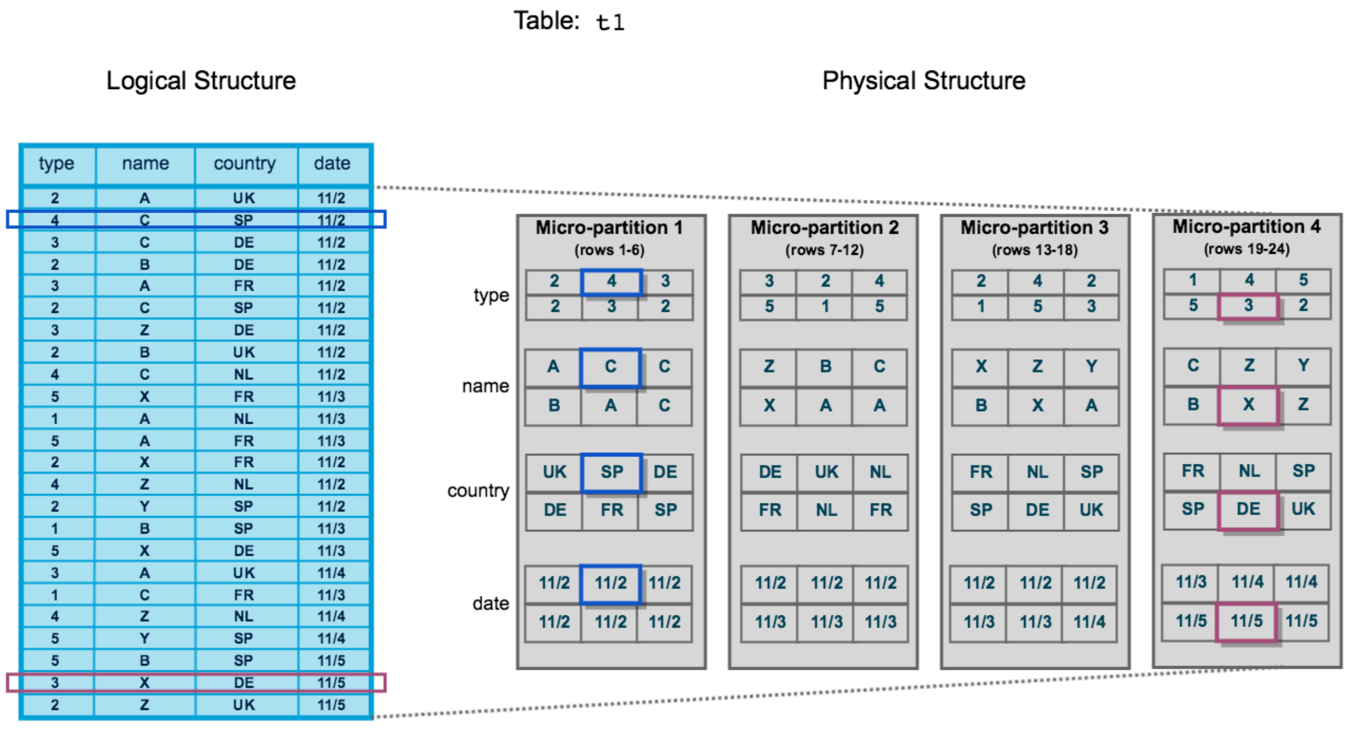
But best pruning happens when the value is explicit or parameterized and known at query compile time.

What is Data Clustering?

Typically, data stored in tables is sorted/ordered along natural dimensions (e.g. date and/or geographic regions). This “clustering” is a key factor in queries because table data that is not sorted or is only partially sorted may impact query performance, particularly on very large tables.

In Snowflake, as data is inserted/loaded into a table, clustering metadata is collected and recorded for each micro-partition created during the process. Snowflake then leverages this clustering information to avoid unnecessary scanning of micro-partitions during querying, significantly accelerating the performance of queries that reference these columns.

The following diagram illustrates a Snowflake table, t1, with four columns sorted by date:



type	name	country	date
2	A	UK	11/2
4	C	SP	11/2
3	C	DE	11/2
2	B	DE	11/2
3	A	FR	11/2
2	C	SP	11/2
3	Z	DE	11/2
2	B	UK	11/2
4	C	NL	11/2
5	X	FR	11/3
1	A	NL	11/3
5	A	FR	11/3
2	X	FR	11/2
4	Z	NL	11/2
2	Y	SP	11/2
1	B	SP	11/3
5	X	DE	11/3
3	A	UK	11/4
1	C	FR	11/3
4	Z	NL	11/4
5	Y	SP	11/4
5	B	SP	11/5
3	X	DE	11/5
2	Z	UK	11/5

The table consists of 24 rows stored across 4 micro-partitions, with the rows divided equally between each micro-partition. Within each micro-partition, the data is sorted and stored by column, which enables Snowflake to perform the following actions for queries on the table:

First, prune micro-partitions that are not needed for the query.

Then, prune by column within the remaining micro-partitions.

Note: This diagram is intended only as a small-scale conceptual representation of the data clustering that Snowflake utilizes in micro-partitions. A typical Snowflake table may consist of thousands, even millions, of micro-partitions.

Clustering Information Maintained for Micro-partitions

Snowflake maintains clustering metadata for the micro-partitions in a table, including:

- ❖ The total number of micro-partitions that comprise the table.
- ❖ The number of micro-partitions containing values that overlap with each other (in a specified subset of table columns).
- ❖ The depth of the overlapping micro-partitions.

Clustering Depth

The clustering depth for a populated table measures the average depth (1 or greater) of the overlapping micro-partitions for specified columns in a table. The smaller the average depth, the better clustered the table is with regards to the specified columns.

Clustering depth can be used for a variety of purposes, including:

1. Monitoring the clustering “health” of a large table, particularly over time as DML is performed on the table.
2. Determining whether a large table would benefit from explicitly defining a clustering key.
3. A table with no micro-partitions (i.e. an unpopulated/empty table) has a clustering depth of 0.

❓What this means:

When you cluster a table in Snowflake (either manually with clustering keys or automatically via automatic clustering), Snowflake tries to organize rows with similar values together in micro-partitions, especially for the clustering key columns.

However, in real-world data, values might span across multiple micro-partitions, creating overlap.

🔍 Overlap example:

Let’s say you're clustering a table on the column order_date.
Imagine Snowflake creates 3 micro-partitions with these ranges:

Micro-Partition	order_date Range
MP1	2024-01-01 to 2024-01-10
MP2	2024-01-05 to 2024-01-15
MP3	2024-01-12 to 2024-01-20

Notice how the date ranges overlap:
MP1 and MP2 overlap (2024-01-05 to 2024-01-10)
MP2 and MP3 overlap (2024-01-12 to 2024-01-15)

So Snowflake counts how many micro-partitions contain overlapping values on the order_date column.
More overlap = worse clustering = less efficient pruning.
✅ Point 3: Depth of overlapping micro-partitions

❓What is "depth"?

Depth refers to how many micro-partitions contain the same value (or overlapping ranges of values) for the clustering key.

Continuing the example:
If you query for order_date = '2024-01-06', both MP1 and MP2 have that date — the depth is 2.

If some values appear in 5+ micro-partitions, the depth is 5+, and that’s a sign of bad clustering.

In other words:
  Point 2: “How widespread is the overlap?”
  Point 3: “How dense is the overlap?”

🧠 **Why this matters**:

When micro-partitions overlap too much, Snowflake can’t prune effectively, because it must scan multiple partitions even for simple filters.

High overlap + high depth means your clustering is degraded and performance drops.

This metadata helps Snowflake (and you) decide when to recluster the table for better performance.

📈 **Visualization:**

```
order_date
|
|---------MP1-----------|
      |---------MP2-----------|
             |--------MP3---------|
```

A value like '2024-01-08' exists in MP1 and MP2 → depth = 2
A value like '2024-01-13' exists in MP2 and MP3 → depth = 2
So there's overlap, and a depth of 2 at minimum.

Example 1: '2024-01-08'

  MP1: ✅ contains it (range includes 2024-01-08)
  MP2: ✅ contains it
  MP3: ❌ doesn't (starts at 2024-01-12)
🔸 So depth = 2
\
Example 2: '2024-01-13'
  MP1: ❌ doesn’t
  MP2: ✅ contains it
  MP3: ✅ contains it
🔸 So depth = 2

Example 3: '2024-01-06'
  MP1: ✅
  MP2: ✅
  MP3: ❌
🔸 Again, depth = 2

Visual diagram:

```
Timeline:       01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20
MP1:               |--------------------|
MP2:                      |--------------------|
MP3:                               |--------------------|
```

Look at 2024-01-13 — it appears in both MP2 and MP3.
Hence, depth = 2 for that value.

Snowflake Views vs. Materialized Views

In Snowflake, both **Views** and **Materialized Views** allow users to define virtual tables based on SQL queries. However, their behavior, performance impact, and use cases differ significantly. This document explains the key differences using real-world scenarios from industry giants like **Meta**, **Amazon**, and **Google**.

🏢 Problem: Slow Analytical Queries on Real-Time Data

Industry Use Case: Meta's Real-Time Feed Recommendations

Meta runs constant analytics on user activity to tune real-time news feed ranking. The base tables (e.g., clicks, likes, views) are huge and updated constantly.

Challenge: Recomputing complex joins or aggregations on-the-fly becomes expensive when the view is queried repeatedly.

✅ Snowflake's Solution

- Use **Views** for ad-hoc exploration, lightweight reuse.
- Use **Materialized Views** for performance-sensitive use cases where precomputed results are beneficial.

🔒 What is a View?

🔍 Concept

A **View** is a stored SQL query that runs every time it is referenced. It does **not store** any data.

⚡ Characteristics

- Real-time reflection of base tables
- Low storage cost
- No performance gain (executes full query each time)

💡 Example (Amazon Redshift Migration Team):

```
CREATE VIEW ORDER_SUMMARY AS
SELECT CUSTOMER_ID, COUNT(*) AS TOTAL_ORDERS
FROM ORDERS
GROUP BY CUSTOMER_ID;
```

Used for: Simple reporting dashboards where data freshness is critical and query cost is acceptable.

🔒 What is a Materialized View?

🔍 Concept

A **Materialized View** stores the **results** of the SQL query physically. It is automatically refreshed in the background as base tables change.

⚡ Characteristics

- Faster query performance
- Consumes storage
- May lag slightly behind real-time (depends on refresh frequency)

💡 Example (Google Ads Click Aggregation):

```
CREATE MATERIALIZED VIEW CLICK_AGG_MV AS
SELECT CAMPAIGN_ID, COUNT(*) AS TOTAL_CLICKS
FROM AD_CLICKS
GROUP BY CAMPAIGN_ID;
```

Used for: Performance-intensive queries like dashboards, ML model input tables, and frequently queried metrics.

📊 Feature Comparison

Feature	View	Materialized View
Data Stored	No	Yes
Query Performance	No optimization	Fast (precomputed results)
Storage Cost	Minimal	Higher (depends on volume)
Reflects Real-Time Data	Yes	No (until refresh)
Auto Refresh on Base Change	Not applicable	Yes (automatic or manual)
Use Case	Exploration, lightweight BI	Dashboards, ML input tables

📆 Refresh Behavior

Materialized Views are **incrementally and automatically refreshed**:

```
-- Check refresh history
SELECT * FROM TABLE(INFORMATION_SCHEMA.MATERIALIZED_VIEW_REFRESH_HISTORY)
WHERE NAME = 'CLICK_AGG_MV';
```

Snowflake optimizes refresh using micro-partition changes and only recalculates impacted portions.

📊 Performance Benchmark

Meta Feed Analytics Benchmark

- **Base Table:** USER_ACTIONS (~5B rows)
- **Query:** Aggregation of reactions per post

Method	Time to Query
View	~18 seconds
Materialized View	~1.2 seconds

📊 When to Use What?

✉ Use Views When:

- You need real-time accuracy
- The query is rarely used
- Underlying data changes frequently
- You want to save storage

🚀 Use Materialized Views When:

- The same complex query runs often
- You need speed (dashboards, ML pipelines)
- Base table change frequency is low-to-moderate

💡 Pro Tips from Big Tech

- **Netflix**: Materialized views power their recommendation dashboards for quick filtering by user segments.
- **Apple**: Uses views during prototyping of internal HR analytics to avoid redundant data copies.
- **Amazon**: Runs materialized views on historical order data to speed up supplier performance metrics.

🎯 Summary

Use Case	Recommended Option
Ad-hoc data exploration	View
Dashboard for 10M+ rows	Materialized View
Historical aggregation	Materialized View
Quick insights during dev	View

Snowflake gives you the flexibility to balance cost, performance, and freshness by choosing between Views and Materialized Views.

Use **Views** for agility. Use **Materialized Views** for performance.

Monitoring Clustering Information for Tables

To view/monitor the clustering metadata for a table, Snowflake provides the following system functions:

- SYSTEM$CLUSTERING_DEPTH
- SYSTEM$CLUSTERING_INFORMATION (including clustering depth)

SYSTEM$CLUSTERING_INFORMATION

Returns clustering information, including average clustering depth, for a table based on one or more columns in the table.

Syntax
SYSTEM$CLUSTERING_INFORMATION('<table_name>'
  [, { '(<expr1> [, <expr2> ...])' | <number_of_errors> }])

Arguments:
table_name
Table for which you want to return clustering information.

(expr1 [, expr2 ...])
Column names or expressions for which clustering information is returned
Examples:

1. Return the 5 most recent clustering errors:

```
SELECT SYSTEM$CLUSTERING_INFORMATION('t1', 5);
```

2. Return the clustering information for a table using two columns in the table:

```
SELECT SYSTEM$CLUSTERING_INFORMATION('test2', '(col1, col3)');
+-----------------------------------------------------------------+
| SYSTEM$CLUSTERING_INFORMATION('TEST2', '(COL1, COL3)')          |
|-----------------------------------------------------------------|
| {                                                               |
|   "cluster_by_keys" : "LINEAR(COL1, COL3)",                     |
|   "total_partition_count" : 1156,                               |
|   "total_constant_partition_count" : 0,                         |
|   "average_overlaps" : 117.5484,                                |
|   "average_depth" : 64.0701,                                    |
|   "partition_depth_histogram" : {                               |
|     "00000" : 0,                                                |
|     "00001" : 0,                                                |
|     "00002" : 3,                                                |
|     "00003" : 3,                                                |
|     "00004" : 4,                                                |
|     "00005" : 6,                                                |
|     "00006" : 3,                                                |
|     "00007" : 5,                                                |
|     "00008" : 10,                                               |
|     "00009" : 5,                                                |
|     "00010" : 7,                                                |
|     "00011" : 6,                                                |
|     "00012" : 8,                                                |
|     "00013" : 8,                                                |
|     "00014" : 9,                                                |
|     "00015" : 8,                                                |
|     "00016" : 6,                                                |
|     "00032" : 98,                                               |
|     "00064" : 269,                                              |
|     "00128" : 698                                               |
|   },                                                                |
|   "clustering_errors" : [ {                                         |
|      "timestamp" : "2023-04-03 17:50:42 +0000",                     |
|      "error" : "(003325) Clustering service has been disabled.\n"  |
|      }                                                              |
|   ]                                                                 |
| }                                                                   |
+---------------------------------------------------------------------+
```

This example indicates that the `test2` table is ***not*** well-clustered for the following reasons:

- Zero (`0`) constant micro-partitions out of `1156` total micro-partitions.
- High average of overlapping micro-partitions.
- High average of overlap depth across micro-partitions.
- Most of the micro-partitions are grouped at the lower-end of the histogram, with the majority of micro-partitions having an overlap depth between `64` and `128`.
- Automatic clustering was previously disabled.

Returns

The function returns a value of type VARCHAR. The returned string is in JSON format and contains the following name/value pairs:

cluster_by_keys
Columns in table used to return clustering information; can be any columns in the table.

notes
This column can contain suggestions to make clustering more efficient. For example, this field might contain a warning if the cardinality of the clustering column is extremely high.

This column can be empty.

total_partition_count
Total number of micro-partitions that comprise the table.

total_constant_partition_count
Total number of micro-partitions for which the value of the specified columns have reached a constant state (i.e. the micro-partitions will not benefit significantly from reclustering). The number of constant micro-partitions in a table has an impact on pruning for queries. The higher the number, the more micro-partitions can be pruned from queries executed on the table, which has a corresponding impact on performance.

average_overlaps
Average number of overlapping micro-partitions for each micro-partition in the table. A high number indicates the table is not well-clustered.

average_depth
Average overlap depth of each micro-partition in the table. A high number indicates the table is not well-clustered.

This value is also returned by SYSTEM$CLUSTERING_DEPTH.

partition_depth_histogram

A histogram depicting the distribution of overlap depth for each micro-partition in the table. The histogram contains buckets with widths:

0 to 16 with increments of 1.

For buckets larger than 16, increments of twice the width of the previous bucket (e.g. 32, 64, 128, …).

clustering_errors
An array of JSON objects, each with a timestamp and error name/value pair. The error describes why automatic clustering was not able to recluster data.

By default, the 10 most recent errors are returned in the array. To return more or fewer errors, specify a number as the second argument of the function.

Limitations

If a table has more than 2 million partitions:

- The results of the function are based on a subset of the table's partitions.
- The value of the output's total_partition_count field is 2 million.

SYSTEM$CLUSTERING_DEPTH

Computes the average depth of the table according to the specified columns (or the clustering key defined for the table). The average depth of a populated table (i.e. a table containing data) is always `1` or more. The smaller the average depth, the better clustered the table is with regards to the specified columns.

```
SYSTEM$CLUSTERING_DEPTH( '<table_name>' , '( <col1> [ , <col2> ... ] )' [ , '<predicate>' ] )
```

Examples

Calculate the clustering depth for a table using the clustering key defined for the table:

```
SELECT SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS');

+----------------------------------------+
| SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS') |
|----------------------------------------+
| 2.4865                                 |
+----------------------------------------+
```

Calculate the clustering depth for a table using two columns in the table:

```
SELECT SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS', '(C2, C9)');

+---------------------------------------------------+
| SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS', '(C2, C9)') |
+---------------------------------------------------+
| 23.1351                                           |
+---------------------------------------------------+
```

Same as the previous example, but with a predicate on one of the columns:

```
SELECT SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS', '(C2, C9)', 'C2 = 25');

+-----------------------------------------------------+
| SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS', '(C2, C9)')  |
+-----------------------------------------------------+
| 11.2452                                             |
  +-----------------------------------------------------+
```

🔍 Example Analysis:

1. `SELECT SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS');`

- **No clustering key specified** (default behavior).
- Result: `2.4865`
- Interpretation: The table is **somewhat well-clustered** even without explicit clustering keys. Lower depth means fewer partitions need to be scanned during queries.

2. `SELECT SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS', '(C2, C9)');`

- Clustering depth is checked on composite key `(C2, C9)`.
- Result: `23.1351`
- Interpretation: This is a **high depth**, meaning that **queries filtering by `C2` and `C9` will be inefficient**, as data related to those columns is **spread across many micro-partitions**.
- Suggestion: Consider **reclustering the table** on `(C2, C9)` if queries often filter on those columns.

3. `SELECT SYSTEM$CLUSTERING_DEPTH('TPCH_ORDERS', '(C2, C9)', 'C2 = 25');`

- Clustering depth on `(C2, C9)` for a filtered subset (`C2 = 25`).
- Result: `11.2452`
- Interpretation: For rows where `C2 = 25`, clustering is **better**, but still not great. This subset has more organization, potentially due to data locality or insert patterns.

📈 Why Clustering Depth Matters for Performance:

- **Lower clustering depth → Fewer micro-partitions to scan → Faster query performance**
- **Higher clustering depth → More partitions to scan → Slower queries**, higher cost
- Helps in identifying when to **recluster** large tables for better **filtering and join performance**

Why Clustering Keys Matter in Modern Data Platforms

Problem: Degraded Performance in Real World at Scale

Large tech companies—like **Amazon**, **Google**, and **Meta**—routinely run multi-petabyte analytical workloads on Snowflake. As data is constantly loaded, updated, and merged, query performance degrades ▪ scans slow down ▪ costs skyrocket ▪ business insights lag.

For example, imagine **Uber's** trip logs table growing daily with DMLs; ad hoc queries filtering by date or region begin to scan excessive micro-partitions, increasing both compute time and cost.

Clustering Key

Definition: A *clustering key* in Snowflake is a defined subset of columns on which data is physically co-located in micro-partitions.

Why It Solves the Problem

- Reduces the number of micro-partitions scanned for queries filtering/sorting on the key.
- Improves columnar compression when key columns are highly correlated with others.
- Declutters table layout once clustering degrades from frequent DMLs.

Real-World Use Case: Netflix

Netflix often queries `view_date`, `customer_id`, and `geography`. By clustering the `streaming_logs` table on `(view_date, region)`, they reduce scan load and latency in weekly consumption reports—despite millions of records per day.

Clustered Tables & Maintenance

Definition: A clustered table is any table for which a clustering key has been defined.

Use as a Solution

- Automatically maintains micro-partition clustering: Snowflake adapts to data changes (merges, deletes, inserts).
- Transparent to users: No need to re-sort the entire table manually.
- Ideal for static-ish or append-heavy tables with frequent reads.

Example: Salesforce

Salesforce clusters its `opportunities` table using `(closed_date, account_id)`. Their sales analytics queries (`GROUP BY`, `ORDER BY`, and `JOIN`) on these columns run drastically faster because of how sorted and grouped data is stored physically.

Determining When to Cluster

Snowflake recommends clustering when:

1. Queries are slower than expected or performance is degrading.
2. **Clustering depth** (via `SYSTEM$CLUSTERING_DEPTH`) is high—indicating widespread, inefficient partition scans.

Real Example: Spotify

Spotify uses `SYSTEM$CLUSTERING_DEPTH('user_events','(event_date, user_id)')`:

- Pre-clustering depth ≈ 50 (inefficient).
- Post-clustering key defined and table rebuilt → depth dropped to ≈ 4.
- Result: Query scan time reduction by ~70%, greatly lowering credit usage.

Reclustering & Manual Intervention

Snowflake auto-maintains clustering, but heavy DML can erode its effectiveness.

Manual Cleanup Steps:

1. **Scan fragmentation**: Identify low clustering efficiency via `SYSTEM$CLUSTERING_DEPTH`.
2. **Recluster / Recreate**: Either:
   - Use **ALTER ... RECLUSTER** (when available),
   - Or: `CREATE TABLE ... CLUSTER BY ... AS SELECT * FROM old;` then swap tables.

Real-World Illustration: Airbnb

Airbnb found nightly upserts of room availability disrupted clustering on `(listing_id, date)`. They implemented a monthly `CREATE TABLE ... AS SELECT` for that table to restore performance—critical for query responsiveness on availability pages.

Trade-Offs & Cost-Benefit Analysis

Consideration	Clustering Benefit	Cost Consideration
Query Speed	Dramatically reduces scan time	Compute credits consumed during auto-recluster
Storage	Improves compression, saves storage costs	Organizing costs can sometimes offset compression gains
Data Change Velocity	Works best for read-heavy, static tables	Too many inserts/updates may offset benefits
Use-Case Relevance	Best for filter/sort on key columns	Analytical patterns must align with clustering

Happy learning
Regards
Saransh Jain