CLUSTERING IN SNOWFLAKE

Clustering in Snowflake is a technique used to improve **query performance** by organizing data within a table based on specific columns. Unlike traditional databases, Snowflake tables are naturally maintained as micro-partitions, which means data is automatically distributed and stored in a highly optimized manner. Clustering helps refine this organization further when working with **large datasets**, particularly for **repetitive queries that filter or join on specific columns**.

Clustering improves Snowflake's performance and cost-efficiency, especially for large datasets. It is particularly valuable in the following scenarios:

- **Date-based filtering**: e.g., sales reports.
- **Region-based filtering**: e.g., regional segmentation.
- **Type 1 & 2 dimension joins**: e.g., enhancing fact-dimension joins by clustering on relevant keys.

How Clustering Works in Snowflake

1. **Clustering Key**: A column or set of columns selected to organize the data physically. Snowflake stores data in micro-partitions, and clustering ensures that rows with similar values in the clustering key are stored closer together.
2. **Query Optimization**:
   - Without clustering, Snowflake might need to scan a large number of micro-partitions for a query.
   - With clustering, Snowflake can quickly locate relevant micro-partitions based on the clustering key, reducing scan time and improving performance.
3. **Automatic Maintenance**:
   - Snowflake continuously manages the clustering for smaller tables automatically.
   - For large tables, explicit clustering maintenance can be scheduled or manually triggered using the RECLUSTER operation.

Why Clustering Is Important

1. **Query Efficiency**:
   - Reduces micro-partition scans by targeting only the relevant partitions.
   - Particularly useful for selective queries (e.g., queries filtered by a specific date range or customer).
2. **Cost Optimization**:
   - Decreases compute costs by reducing the amount of data scanned.

3. **Improves Sorting**:
   - Helps maintain sorted order, which can be beneficial for window functions and aggregations.

Business Use Cases for Clustering

1. Date-Driven Reporting

Scenario: A retail business analyzing daily sales data.

- **Problem**: Queries frequently filter on the sale_date column. Without clustering, all partitions are scanned, even for a small date range.
- **Solution**: Cluster the table on the sale_date column. Queries for specific days, weeks, or months will scan only the relevant micro-partitions.

2. Customer Segmentation Analysis

Scenario: A marketing team segments customers based on regions.

- **Problem**: Frequent filtering by region leads to high partition scans.
- **Solution**: Cluster the table by region. This reduces scan time for region-specific queries.

Clustering with Type 1 and Type 2 Dimensions

Type 1 Dimensions

Definition: Overwrites old data with new data; no history is maintained.

- **Example**: Customer profile information where changes overwrite previous values (e.g., customer_address).

Use Case:

- **Scenario**: Queries frequently join fact tables with a Type 1 dimension using a customer ID.
- **Clustering**: Cluster the fact table on customer_id. This reduces the join cost by ensuring data is co-located for faster lookups.

Type 2 Dimensions

Definition: Maintains historical data by creating a new row for each change.

- **Example**: Customer membership levels with start and end dates for each change (membership_status, effective_date, end_date).

Use Case:

- **Scenario**: A sales fact table needs to join with a Type 2 dimension to determine customer status at the time of a transaction.

CLUSTERING IN SNOWFLAKE

- **Clustering**:
  - Cluster the fact table on transaction_date and the dimension table on effective_date.
  - This ensures queries that join based on date ranges (e.g., for status validity) perform efficiently.

Key Concepts in Clustering

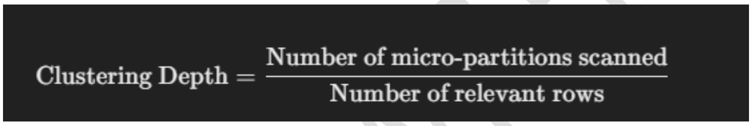
1. Clustering Keys

A clustering key is a column or set of columns used to group related data physically within micro-partitions. Clustering improves query performance by reducing the number of micro-partitions that need to be scanned.

2. Clustering Depth

Clustering depth measures how well the data within a table is organized based on the clustering key. It indicates the average number of micro-partitions scanned per query for a given clustering key.

Formula:



$$\text{Clustering Depth} = \frac{\text{Number of micro-partitions scanned}}{\text{Number of relevant rows}}$$

A lower clustering depth implies better clustering and fewer partitions scanned.

3. Reclustering

If the clustering key's effectiveness deteriorates over time due to data updates or inserts, reclustering reorganizes the data within the micro-partitions. Snowflake handles this process automatically using its **Automatic Clustering** feature.

When to Use Clustering

Clustering is most beneficial when:

- Queries frequently filter, aggregate, or join on specific columns.
- Tables are very large (e.g., millions or billions of rows).
- Query performance degrades due to excessive micro-partition scanning.

For smaller datasets (e.g., 100K records), clustering may not provide significant benefits because Snowflake’s native micro-partitioning and pruning are already efficient.

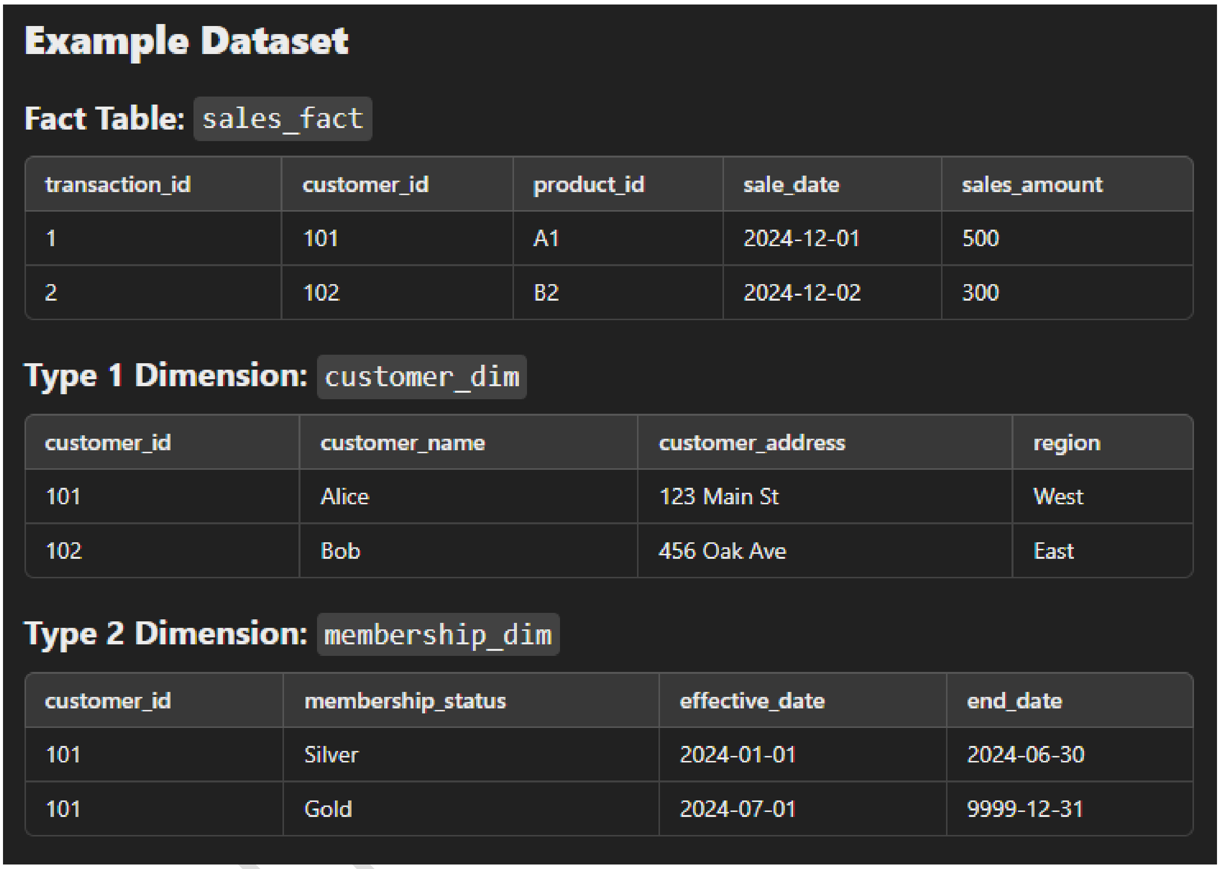
How to Choose Clustering Keys

When deciding clustering keys for a dataset with multiple fact and dimension tables, consider:

- **High-cardinality columns:** Columns with a wide range of unique values (e.g., OrderID, CustomerID).

- **Frequently filtered columns:** Columns that appear often in WHERE, GROUP BY, or JOIN clauses.
- **Temporal columns:** Date/time columns frequently used in range queries.
- **Composite keys:** Use a combination of keys if queries filter across multiple dimensions.

Here's an example of setting up **clustering keys** in Snowflake, with a focus on improving performance for large datasets. We'll cover scenarios for **Type 1** and **Type 2 dimensions**.



Example Dataset

Fact Table: `sales_fact`

transaction_id	customer_id	product_id	sale_date	sales_amount
1	101	A1	2024-12-01	500
2	102	B2	2024-12-02	300

Type 1 Dimension: `customer_dim`

customer_id	customer_name	customer_address	region
101	Alice	123 Main St	West
102	Bob	456 Oak Ave	East

Type 2 Dimension: `membership_dim`

customer_id	membership_status	effective_date	end_date
101	Silver	2024-01-01	2024-06-30
101	Gold	2024-07-01	9999-12-31

1. Creating and Clustering a Fact Table

In this example, clustering on **sale_date** optimizes date-range queries for reporting.

```
CREATE TABLE sales_fact (
   transaction_id INT,
   customer_id INT,
   product_id VARCHAR,
   sale_date DATE,
   sales_amount NUMERIC
)
CLUSTER BY (sale_date);
```

2. Clustering a Type 1 Dimension

For a **Type 1 dimension**, **clustering** by **customer_id** improves **join** performance with the **fact** table.

```
CREATE TABLE customer_dim (

   customer_id INT,

   customer_name VARCHAR,

   customer_address VARCHAR,

   region VARCHAR

)

CLUSTER BY (customer_id);
```

3. Clustering a Type 2 Dimension

For a **Type 2 dimension**, clustering by **customer_id** and **effective_date** optimizes range queries for joining on date ranges.

```
CREATE TABLE membership_dim (

   customer_id INT,

   membership_status VARCHAR,

   effective_date DATE,

   end_date DATE

)

CLUSTER BY (customer_id, effective_date);
```

Query Examples

1. Fact Table Query on sale_date

Querying sales for December 2024 will now efficiently scan only relevant partitions.

```
SELECT *

FROM sales_fact

WHERE sale_date BETWEEN '2024-12-01' AND '2024-12-31';
```

---

2. Join with Type 1 Dimension

Joining the fact table with the **Type 1 dimension** benefits from **clustering** on **customer_id**.

```
SELECT f.transaction_id, c.customer_name, f.sales_amount
```

```
FROM sales_fact f

JOIN customer_dim c

  ON f.customer_id = c.customer_id

WHERE f.sale_date = '2024-12-01';
```

---

3. Join with Type 2 Dimension

Joining the fact table with the **Type 2 dimension** for a valid membership status at the transaction time benefits from clustering on **customer_id** and **effective_date**.

```
SELECT f.transaction_id, m.membership_status, f.sales_amount

FROM sales_fact f

JOIN membership_dim m

  ON f.customer_id = m.customer_id

     AND f.sale_date BETWEEN m.effective_date AND m.end_date

WHERE f.sale_date = '2024-12-01';
```

Maintaining Clustering

Reclustering Large Tables

If a table becomes fragmented due to frequent inserts or updates, run **reclustering** to maintain optimal performance.

```
ALTER TABLE sales_fact RECLUSTER;
```

Automated Clustering

Enable Snowflake's automated clustering for dynamic updates:

```
ALTER TABLE sales_fact SET CLUSTERING KEY (sale_date);
```

Use Cases and Business Scenarios

1. Multi-Fact Table Scenario

- **Dataset:** Two fact tables: Sales and Inventory, and several dimension tables: Products, Customers, and Stores.
- **Query Pattern:** Regular reporting involves filtering Sales and Inventory by Date, Region, and ProductID.

Clustering Strategy:

- Cluster Sales and Inventory tables on (Date, Region, ProductID).

- This key ensures that micro-partitions are organized to optimize queries filtering on these columns.

2. Historical Data with Temporal Queries

- **Dataset:** A Transactions table with 100K daily records stored as CSV files, queried by TransactionDate and CustomerID.
- **Query Pattern:** Analysts frequently run reports for specific date ranges or customer cohorts.

Clustering Strategy:

- Use TransactionDate as the primary clustering key.
- If queries also filter by customer segments, consider (TransactionDate, CustomerID) as a composite key.

3. Geo-Spatial Analysis

- **Dataset:** A Shipments table containing ShipmentID, ShipDate, and DestinationRegion.
- **Query Pattern:** Analysts frequently filter by DestinationRegion and aggregate data by ShipDate.

Clustering Strategy:

- Cluster on (DestinationRegion, ShipDate) to optimize geo-spatial analysis.

4. Large-Scale Dimension Tables

- **Dataset:** A Customers table with customer demographics and purchase history.
- **Query Pattern:** Regular joins with fact tables like Sales and filters on Region and AgeGroup.

Clustering Strategy:

- Cluster on Region (if high cardinality) or (Region, AgeGroup) if queries involve demographic analysis.

Clustering Depth Calculation

1. **Analyze Clustering:** Use the SYSTEM$CLUSTERING_INFORMATION function to evaluate the clustering depth:

   SELECT SYSTEM$CLUSTERING_INFORMATION('DATABASE.SCHEMA.TABLE_NAME', '(Column1, Column2)');

2. Optimize Clustering: If clustering depth is high, consider reclustering:
   ALTER TABLE TABLE_NAME RECLUSTER;

Best Practices

1. **Test Before Applying:** Use query history to identify patterns before defining clustering keys.

2. **Monitor Query Performance:** Use Snowflake's Query Profile to understand the impact of clustering.
3. **Automatic Clustering:** Leverage automatic clustering for dynamic datasets where manual reclustering isn't feasible.
4. **Limit the Number of Keys:** Avoid overloading clustering keys to prevent excessive micro-partition scans.

1. Multi-Fact Table Scenario

Scenario: Reporting involves filtering **Sales** and **Inventory** tables by **Date**, **Region**, and **ProductID**.

Clustering Definition:

```
ALTER TABLE SALES CLUSTER BY (Date, Region, ProductID);

ALTER TABLE INVENTORY CLUSTER BY (Date, Region, ProductID);
```

Query:

```
SELECT
    s.Region, p.ProductName,
    SUM(s.SalesAmount) AS TotalSales,
    SUM(i.StockQuantity) AS AvailableStock
FROM Sales s
JOIN Inventory i ON s.ProductID = i.ProductID AND s.Date = i.Date
JOIN Products p ON s.ProductID = p.ProductID
WHERE s.Date BETWEEN '2024-01-01' AND '2024-01-31' AND s.Region = 'North'
GROUP BY s.Region, p.ProductName
ORDER BY TotalSales DESC;
```

2. Historical Data with Temporal Queries

Scenario: Analysts frequently filter **Transactions** by **TransactionDate** and **CustomerID**.

Clustering Definition:

```
ALTER TABLE TRANSACTIONS CLUSTER BY (TransactionDate, CustomerID);
```

Query :

```
SELECT
```

```
    CustomerID,

    COUNT(*) AS TotalTransactions,

    SUM(TransactionAmount) AS TotalAmount

FROM Transactions

WHERE TransactionDate BETWEEN '2024-01-01' AND '2024-03-31' AND CustomerID IN ('CUST123',
'CUST456')

GROUP BY CustomerID

ORDER BY  TotalAmount DESC;
```

3. Geo-Spatial Analysis

Scenario: Shipments are analyzed by DestinationRegion and ShipDate.

Clustering Definition:

```
ALTER TABLE SHIPMENTS CLUSTER BY (DestinationRegion, ShipDate);
```

Query:

```
SELECT

    DestinationRegion,

    COUNT(*) AS TotalShipments,

    MIN(ShipDate) AS FirstShipmentDate,

    MAX(ShipDate) AS LastShipmentDate


FROM Shipments

WHERE DestinationRegion = 'East Coast' AND ShipDate BETWEEN '2024-06-01' AND '2024-06-30'

GROUP BY  DestinationRegion

ORDER BY TotalShipments DESC;
```

4. Large-Scale Dimension Table

Scenario: The **Customers** table is frequently joined with fact tables, and filters are applied to **Region** and **AgeGroup**.

Clustering Definition:

```
ALTER TABLE CUSTOMERS CLUSTER BY (Region, AgeGroup);

SELECT c.Region, c.AgeGroup, SUM(s.SalesAmount) AS TotalSales
```

```
FROM Customers c

JOIN Sales s ON c.CustomerID = s.CustomerID

WHERE  c.Region = 'West' AND c.AgeGroup = '25-34' AND s.Date BETWEEN '2024-07-01' AND '2024-07-31'

GROUP BY c.Region, c.AgeGroup

ORDER BY TotalSales DESC;
```

5. Clustering Analysis and Optimization

Analyze Clustering Information:

To check the clustering depth and quality:

```
SELECT SYSTEM$CLUSTERING_INFORMATION('DATABASE.SCHEMA.SALES', '(Date, Region, ProductID)');
```

Reclustering a Table:

If clustering depth is high, trigger reclustering manually:

```
ALTER TABLE SALES RECLUSTER;
```

Additional Considerations

- **Monitoring Query Profile: Use the Query Profile in the Snowflake Web UI to evaluate the partition pruning impact. Look for reduced micro-partitions scanned.**
- Clustering with Automatic Clustering Enabled:

```
ALTER TABLE SALES SET AUTOMATIC_CLUSTERING = TRUE;
```

To analyze query performance using the **Query Profile** in **Snowflake**, follow these steps:

Step 1: Run a Query

Execute a sample query on a table with clustering keys to observe the performance impact.

```
SELECT Region, ProductID, SUM(SalesAmount) AS TotalSales

FROM Sales

WHERE Date BETWEEN '2024-01-01' AND '2024-01-31' AND Region = 'North'

GROUP BY Region, ProductID

ORDER BY  TotalSales DESC;
```

Step 2: Access the Query Profile

1. Open the Snowflake Web UI.
2. Navigate to History (left-hand menu).
3. Locate the query you executed. Use filters if necessary to find it quickly.
4. Click on the query's ID to open the Query Details page.

Step 3: Evaluate the Query Profile

The Query Profile provides a visual representation of the query execution process, including:

Key Areas to Observe

1. **Partition Pruning:**
   - Check how many micro-partitions were scanned versus how many were pruned.
   - If the clustering keys are effective, most irrelevant micro-partitions should be pruned.

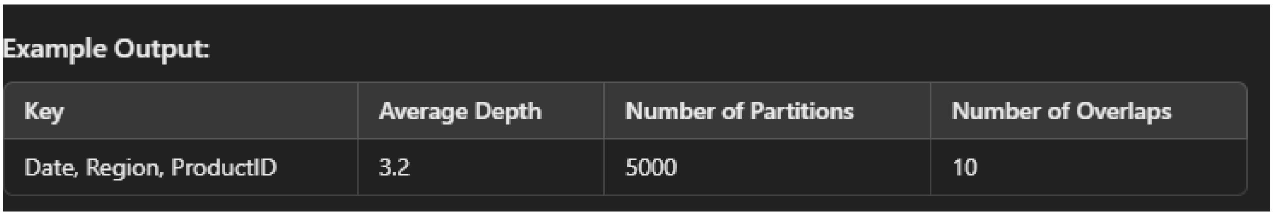
   Example:

   - Scanned Partitions: 100
   - Pruned Partitions: 900 (indicates good clustering).
2. **Execution Steps:**
   - Review the graphical breakdown of stages like filtering, scanning, and joining.
   - Ensure the Scan Nodes show efficient data retrieval.
3. **Performance Metrics:**
   - Total Time: Should decrease with good clustering.
   - Bytes Scanned: Should be minimal for optimized clustering.
4. **Row Count by Stage:**
   - Look for data reduction at each stage, particularly after partition pruning.

Step 4: Analyze Clustering Quality

To dive deeper, check the clustering information using:

SELECT SYSTEM$CLUSTERING_INFORMATION('DATABASE.SCHEMA.SALES', '(Date, Region, ProductID)');



Example Output:

Key	Average Depth	Number of Partitions	Number of Overlaps
Date, Region, ProductID	3.2	5000	10

- **Average Depth:** Should be close to 1 for optimal clustering.
- **Number of Overlaps:** Indicates how often data overlaps across micro-partitions. Fewer overlaps are better.

Step 5: Optimize Query if Necessary

If the query scans too many micro-partitions or the clustering depth is high:

1. **Revisit Clustering Keys:** Modify clustering keys to better align with query filters.
2. **Recluster Table:** Trigger manual reclustering.

```
ALTER TABLE SALES RECLUSTER;
```

Step 6: Rerun the Query and Compare

Run the same query after making adjustments and review the Query Profile again. Improved performance should show:

- **Fewer micro-partitions scanned.**
- **Reduced query execution time.**
- **Smaller bytes scanned.**

Setting Up Automatic Monitoring for Clustering Metrics and Query Optimization

Snowflake offers tools and techniques to monitor clustering performance and continuously optimize query execution.

1. Monitoring Clustering Metrics

Scheduled Clustering Depth Monitoring

You can create a task in Snowflake to regularly check clustering metrics and log results for analysis.

Step 1: Create a Table to Store Clustering Metrics

```
CREATE TABLE CLUSTERING_METRICS_LOG (
  TableName STRING,
  ClusteringKeys STRING,
  AvgDepth FLOAT,
  TotalPartitions INT,
  Overlaps INT,
  LogTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create a Stored Procedure to Log Clustering Metrics

```
CREATE OR REPLACE PROCEDURE LOG_CLUSTERING_METRICS()

RETURNS STRING

LANGUAGE SQL

AS

$$

BEGIN

   INSERT INTO CLUSTERING_METRICS_LOG

   SELECT

      'DATABASE.SCHEMA.SALES' AS TableName,

      'Date, Region, ProductID' AS ClusteringKeys,

      AVG_DEPTH AS AvgDepth,

      PARTITION_COUNT AS TotalPartitions,

      OVERLAPS AS Overlaps,

      CURRENT_TIMESTAMP AS LogTimestamp

   FROM

      TABLE(SYSTEM$CLUSTERING_INFORMATION('DATABASE.SCHEMA.SALES', '(Date, Region,
ProductID)'));

   RETURN 'Clustering Metrics Logged Successfully';

END;

$$;
```

Step 3: Schedule a Task

```
CREATE OR REPLACE TASK MONITOR_CLUSTERING

WAREHOUSE = DEMO_WAREHOUSE

SCHEDULE = 'USING CRON 0 0 * * *'  -- Runs daily at midnight

AS

CALL LOG_CLUSTERING_METRICS();
```

Step 4: Start the Task

```
ALTER TASK MONITOR_CLUSTERING RESUME;
```

2. Optimizing Query Plans

Use Query History and Profiling

Regularly review **Query History** to identify high-cost queries that could benefit from clustering or index optimization.

SQL to Extract High-Cost Queries:

```
SELECT
   QUERY_ID, EXECUTION_STATUS,
   TOTAL_ELAPSED_TIME / 1000 AS ExecutionTimeInSec,
   BYTES_SCANNED / (1024 * 1024 * 1024) AS GBScanned,
   ROWS_PROCESSED,ROWS_RETURNED,
   QUERY_TEXT
FROM
   SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
WHERE
   EXECUTION_STATUS = 'SUCCESS'
   AND TOTAL_ELAPSED_TIME > 10000  -- Queries taking longer than 10 seconds
   AND QUERY_START_TIME >= DATEADD(DAY, -7, CURRENT_DATE) -- Last 7 days
ORDER BY TOTAL_ELAPSED_TIME DESC;
```

Query Optimization Tips

1. **Index Frequent Filters:**
   - Identify columns frequently used in **WHERE**, **JOIN**, or **GROUP** BY clauses and define clustering keys on them.

   Example:

   ```
   ALTER TABLE SALES CLUSTER BY (Date, ProductID);
   ```

2. **Leverage Materialized Views:** For frequently queried aggregations or filters, use materialized views to pre-compute results.

   ```
   CREATE MATERIALIZED VIEW MV_SALES_SUMMARY AS
   SELECT Date, Region, ProductID,
   ```

```
         SUM(SalesAmount) AS TotalSales
FROM Sales
GROUP BY Date, Region, ProductID;
```

3. **Partition Elimination in Query Design:** Write queries that maximize Snowflake's partition pruning:
   - Avoid **non-SARGable** conditions (e.g., avoid FUNCTION(Column) in filters).
   - Example of **SARGable** query:
     SELECT * FROM Sales WHERE Date >= '2024-01-01' AND Date <= '2024-01-31';

4. **Automate Query Performance Alerts**
   Log Query Performance Metrics
   Use Snowflake's **Account Usage** schema to log and analyze query performance.

Step 1: Create a Table for Query Metrics

```
CREATE TABLE QUERY_METRICS_LOG (
  QueryID STRING,
  ExecutionTimeInSec FLOAT,
  BytesScannedGB FLOAT,
  RowsProcessed INT,
  RowsReturned INT,
  QueryText STRING,
  LogTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Stored Procedure to Log High-Cost Queries

```
CREATE OR REPLACE PROCEDURE LOG_HIGH_COST_QUERIES()
RETURNS STRING
LANGUAGE SQL
AS
$$

BEGIN
  INSERT INTO QUERY_METRICS_LOG
  SELECT
    QUERY_ID,
    TOTAL_ELAPSED_TIME / 1000 AS ExecutionTimeInSec,
    BYTES_SCANNED / (1024 * 1024 * 1024) AS GBScanned,
    ROWS_PROCESSED,
    ROWS_RETURNED,
    QUERY_TEXT,
    CURRENT_TIMESTAMP AS LogTimestamp
  FROM
    SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
  WHERE
    EXECUTION_STATUS = 'SUCCESS'
    AND TOTAL_ELAPSED_TIME > 10000  -- Queries taking longer than 10 seconds
    AND QUERY_START_TIME >= DATEADD(MINUTE, -60, CURRENT_TIMESTAMP); -- Last 1 hour
  RETURN 'High-Cost Queries Logged Successfully';
END;
$$;
```

Step 3: Automate with a Task

```
CREATE OR REPLACE TASK MONITOR_QUERY_PERFORMANCE
WAREHOUSE = DEMO_WAREHOUSE
SCHEDULE = 'USING CRON 0 * * * *'  -- Runs every hour
AS
CALL LOG_HIGH_COST_QUERIES();
```

Step 4: Start the Task

```
ALTER TASK MONITOR_QUERY_PERFORMANCE RESUME
```

By using **clustering metrics**, **query history analysis, and automation**, you can continuously improve your Snowflake environment for query performance and **cost-efficiency.**

Let's break down the implementation into manageable steps. I'll guide you through setting up **automatic monitoring for clustering and query performance** in Snowflake:

Step 1: Monitor Clustering Metrics

1. **Create a Metrics Log Table** This table will store historical clustering information for further analysis.

```
CREATE TABLE CLUSTERING_METRICS_LOG (
  TableName STRING,
  ClusteringKeys STRING,
  AvgDepth FLOAT,
  TotalPartitions INT,
  Overlaps INT,
  LogTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

2. **Create a Stored Procedure** The procedure will query clustering information and insert it into the log table.

```
CREATE OR REPLACE PROCEDURE LOG_CLUSTERING_METRICS()
RETURNS STRING
LANGUAGE SQL
AS
$$
BEGIN
  INSERT INTO CLUSTERING_METRICS_LOG
```

```
    SELECT
        'DATABASE.SCHEMA.SALES' AS TableName,
        'Date, Region, ProductID' AS ClusteringKeys,
        AVG_DEPTH AS AvgDepth,
        PARTITION_COUNT AS TotalPartitions,
        OVERLAPS AS Overlaps,
        CURRENT_TIMESTAMP AS LogTimestamp
    FROM
        TABLE(SYSTEM$CLUSTERING_INFORMATION('DATABASE.SCHEMA.SALES', '(Date,
Region, ProductID)'));

    RETURN 'Clustering Metrics Logged Successfully';
END;
$$;
```

3. Schedule a Task to Log Metrics Automate the stored procedure to run daily or weekly.

```
CREATE OR REPLACE TASK MONITOR_CLUSTERING
WAREHOUSE = COMPUTE_WH
SCHEDULE = 'USING CRON 0 0 * * *'  -- Daily at midnight
AS
CALL LOG_CLUSTERING_METRICS();
```

4. Activate the Task

```
ALTER TASK MONITOR_CLUSTERING RESUME;
```

Step 2: Monitor High-Cost Queries

1. **Create a Query Metrics Log Table** This table will track performance metrics for queries that take longer or scan significant data.

```
CREATE TABLE QUERY_METRICS_LOG (
    QueryID STRING,
    ExecutionTimeInSec FLOAT,
    BytesScannedGB FLOAT,
    RowsProcessed INT,
    RowsReturned INT,
     QueryText STRING,
    LogTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

2. **Create a Stored Procedure** This procedure logs high-cost queries from the Account Usage schema.

```
CREATE OR REPLACE PROCEDURE LOG_HIGH_COST_QUERIES()
RETURNS STRING
```

```
LANGUAGE SQL
AS
$$
BEGIN
  INSERT INTO QUERY_METRICS_LOG
  SELECT
    QUERY_ID,
    TOTAL_ELAPSED_TIME / 1000 AS ExecutionTimeInSec,
    BYTES_SCANNED / (1024 * 1024 * 1024) AS GBScanned,
    ROWS_PROCESSED,
    ROWS_RETURNED,
    QUERY_TEXT,
    CURRENT_TIMESTAMP AS LogTimestamp
  FROM
    SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY

 WHERE
    EXECUTION_STATUS = 'SUCCESS'
    AND TOTAL_ELAPSED_TIME > 10000  -- Queries taking longer than 10 seconds
    AND QUERY_START_TIME >= DATEADD(MINUTE, -60, CURRENT_TIMESTAMP); -- Last 1 hour

  RETURN 'High-Cost Queries Logged Successfully';
END;
$$;
```

3. **Schedule a Task for Query Monitoring** Automate the procedure to capture performance data hourly.

```
CREATE OR REPLACE TASK MONITOR_QUERY_PERFORMANCE
WAREHOUSE = DEMO_WAREHOUSE
SCHEDULE = 'USING CRON 0 * * * *'  -- Every hour
AS
CALL LOG_HIGH_COST_QUERIES();
```

4. **Activate the Task**

```
ALTER TASK MONITOR_QUERY_PERFORMANCE RESUME;
```

Step 3: Analyze and Optimize

Analyze Clustering and Query Performance

- **Clustering Metrics Analysis:**

```
SELECT *
FROM CLUSTERING_METRICS_LOG
ORDER BY LogTimestamp DESC;
```

Look for tables with:

  - High **AvgDepth** (inefficient clustering).
  - Many **Overlaps** (data poorly distributed).

- **High-Cost Query Analysis:**

```sql
SELECT *
FROM QUERY_METRICS_LOG
WHERE ExecutionTimeInSec > 20  -- High-cost threshold
ORDER BY ExecutionTimeInSec DESC;
```

Optimize Identified Tables/Queries

- **Reclustering Tables:** Improve clustering with:

```sql
ALTER TABLE SALES RECLUSTER;
```

- **Redefine Clustering Keys:** Adjust based on analysis:

```sql
ALTER TABLE SALES CLUSTER BY (Region, ProductID);
```

- **Materialized Views:** Create pre-aggregated views for repetitive, expensive queries:

```sql
CREATE MATERIALIZED VIEW MV_SALES_BY_REGION AS
SELECT
   Region, ProductID,
   SUM(SalesAmount) AS TotalSales
FROM SALES
GROUP BY Region, ProductID;
```

Step 4: Automate Alerts for Anomalies

1. **Create Alert for High AvgDepth** Use Snowflake **Streams** and **Tasks**:
   - Create a stream to detect high AvgDepth in **CLUSTERING_METRICS_LOG**.

```sql
CREATE STREAM CLUSTERING_ALERT_STREAM ON TABLE
CLUSTERING_METRICS_LOG;
```

   - Create a task to notify via a service (e.g., email, Slack):

```sql
CREATE OR REPLACE TASK SEND_CLUSTERING_ALERT
WAREHOUSE = DEMO_WAREHOUSE
AS
SELECT *
FROM CLUSTERING_ALERT_STREAM
WHERE AvgDepth > 5;  -- Custom threshold
```

2. **Activate Alerts Task**

```sql
ALTER TASK SEND_CLUSTERING_ALERT RESUME;
```

CLUSTERING IN SNOWFLAKE

Using these Stored Procedure & Tasks

1. **Monitor Regularly:** Run these views periodically to analyze trends in query performance.
2. **Identify Bottlenecks:**
   - Queries scanning large numbers of rows or bytes.
   - Long-running queries.
   - Tables with high clustering depths.
3. **Optimization Actions:**
   - Optimize queries with high scan rates.
   - Reclustering tables with inefficient clustering.
   - Adjust warehouse size for resource-intensive queries.

****