**1.Advanced Data Transformations in Snowflake**

In Snowflake, advanced data transformations involve converting raw data into meaningful, structured formats suitable for analytics and reporting. These transformations can include filtering, aggregating, joining, pivoting, and applying business logic. Snowflake provides several SQL functions and features, such as CTEs (Common Table Expressions), window functions, and JSON manipulation, to perform these transformations efficiently.

**Scenario: Sales Data Transformation**

Suppose you have a table SALES\_DATA that contains raw sales transactions from different regions with the following structure:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sale\_ID | Sale\_Date | Region | Product | Quantity | Unit\_Price | Customer |
| 1 | 2024-01-01 | North | A | 10 | 20.0 | John |
| 2 | 2024-01-01 | South | B | 5 | 50.0 | Sarah |
| 3 | 2024-01-02 | North | A | 15 | 20.0 | Emily |
| 4 | 2024-01-02 | East | C | 8 | 40.0 | Mark |

**Step-by-Step Transformation**

**1. Basic Aggregation: Total Sales and Revenue by Region and Product**

First, you want to calculate the total sales (SUM(Quantity)) and revenue (SUM(Quantity \* Unit\_Price)) for each region and product.

SQL Query:  
```sql  
SELECT   
 Region,  
 Product,  
 SUM(Quantity) AS Total\_Sales,  
 SUM(Quantity \* Unit\_Price) AS Total\_Revenue  
FROM   
 SALES\_DATA  
GROUP BY   
 Region, Product  
ORDER BY   
 Region, Product;  
```

**2. Window Function: Running Total of Sales by Region**

Next, you want to add a running total of sales for each region. To do this, you can use the SUM() window function along with the PARTITION BY clause.

SQL Query:  
```sql  
WITH Sales\_Aggregated AS (  
 SELECT   
 Region,  
 Product,  
 SUM(Quantity) AS Total\_Sales,  
 SUM(Quantity \* Unit\_Price) AS Total\_Revenue  
 FROM   
 SALES\_DATA  
 GROUP BY   
 Region, Product  
)  
SELECT  
 Region,  
 Product,  
 Total\_Sales,  
 Total\_Revenue,  
 SUM(Total\_Sales) OVER (PARTITION BY Region ORDER BY Product) AS Running\_Total\_Sales  
FROM  
 Sales\_Aggregated  
ORDER BY  
 Region, Product;  
```

**3. Pivoting Data: Creating a Product Sales Matrix**

You can pivot the data to create a sales matrix where each row represents a region, and each column represents the total sales of a specific product.

SQL Query:  
```sql  
SELECT   
 Region,  
 SUM(CASE WHEN Product = 'A' THEN Quantity ELSE 0 END) AS Product\_A\_Sales,  
 SUM(CASE WHEN Product = 'B' THEN Quantity ELSE 0 END) AS Product\_B\_Sales,  
 SUM(CASE WHEN Product = 'C' THEN Quantity ELSE 0 END) AS Product\_C\_Sales  
FROM   
 SALES\_DATA  
GROUP BY   
 Region  
ORDER BY   
 Region;  
```

**4. JSON Transformation: Extracting Customer Information**

If customer data is stored in a JSON format in a column, you can use Snowflake’s FLATTEN() and OBJECT functions to transform the data.

Assume there’s a Customer\_Info column in JSON format:  
SQL Query:  
```sql  
SELECT   
 Sale\_ID,  
 Sale\_Date,  
 Region,  
 Product,  
 Quantity,  
 Unit\_Price,  
 Customer\_Info:Name::STRING AS Customer\_Name,  
 Customer\_Info:Email::STRING AS Customer\_Email  
FROM   
 SALES\_DATA  
WHERE   
 Region = 'North';  
```

**5. Time-Series Data Transformation: Rolling Average Sales**

If you want to calculate a rolling average of sales over a 7-day window, you can use window functions like AVG().

SQL Query:  
```sql  
SELECT   
 Sale\_Date,  
 Region,  
 Product,  
 Quantity,  
 AVG(Quantity) OVER (PARTITION BY Region ORDER BY Sale\_Date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS Rolling\_Avg\_Sales  
FROM   
 SALES\_DATA  
ORDER BY   
 Sale\_Date, Region;  
```

**2.Optimization Techniques**

We will use several optimization techniques that can help improve query performance on the SALES\_DATA table. The techniques we will cover include:

1. Clustering Keys: For better performance when filtering on specific columns.

2. Materialized Views: For frequently run aggregation queries.

3. Result Caching: To take advantage of cached query results.

4. Data Compression: Ensures efficient storage.

**Scenario: Optimizing Queries for Sales Data**

Suppose we frequently run a query to analyze total sales by region and product for a given date range. This query may look like:

SELECT   
 Region,   
 Product,   
 SUM(Quantity) AS Total\_Quantity,   
 SUM(Quantity \* Unit\_Price) AS Total\_Sales  
FROM SALES\_DATA  
WHERE Sale\_Date BETWEEN '2024-01-01' AND '2024-01-02'  
GROUP BY Region, Product;

**Step-by-Step Optimization**

**1. Analyzing Query Performance**

Before applying optimizations, use the EXPLAIN command to analyze the performance of the query:

EXPLAIN  
SELECT   
 Region,   
 Product,   
 SUM(Quantity) AS Total\_Quantity,   
 SUM(Quantity \* Unit\_Price) AS Total\_Sales  
FROM SALES\_DATA  
WHERE Sale\_Date BETWEEN '2024-01-01' AND '2024-01-02'  
GROUP BY Region, Product;

This will provide insights into the query execution plan and how many rows are scanned.

**2. Implementing Clustering Keys**

To optimize queries that filter by Sale\_Date, we can set up a clustering key on this column:

ALTER TABLE SALES\_DATA  
CLUSTER BY (Sale\_Date);

Clustering the table by Sale\_Date helps Snowflake organize the data physically in a way that minimizes the number of micro-partitions scanned during queries.

**3. Creating Materialized Views**

If you often query total sales by region and product, consider creating a materialized view to pre-compute these aggregations:

CREATE MATERIALIZED VIEW mv\_sales\_summary AS  
SELECT   
 Region,   
 Product,   
 SUM(Quantity) AS Total\_Quantity,   
 SUM(Quantity \* Unit\_Price) AS Total\_Sales  
FROM SALES\_DATA  
GROUP BY Region, Product;

When you query the materialized view instead of the base table, it will be faster as the results are pre-computed.

**4. Using Result Caching**

Snowflake automatically caches the results of queries for 24 hours. If the same query is executed multiple times, it can leverage this cache. To ensure your query takes advantage of this feature, simply rerun the query after running it once. If the data in the underlying table hasn’t changed, Snowflake will return the cached results:

SELECT   
 Region,   
 Product,   
 SUM(Quantity) AS Total\_Quantity,   
 SUM(Quantity \* Unit\_Price) AS Total\_Sales  
FROM SALES\_DATA  
WHERE Sale\_Date BETWEEN '2024-01-01' AND '2024-01-02'  
GROUP BY Region, Product;

**5. Data Compression**

Snowflake automatically compresses data during storage, which helps in reducing costs and improving performance. You don’t need to manually manage this, but it’s important to ensure that your warehouse has adequate storage for larger datasets.

**Monitoring and Maintenance**

To ensure that the optimizations are effective, you should monitor the performance of your queries and the state of your materialized views and clustering. Use the following commands:

**Check the status of materialized views:**

SELECT \* FROM TABLE(information\_schema.materialized\_views);

**Monitor clustering depth:**

SELECT SYSTEM$CLUSTERING\_DEPTH('SALES\_DATA');

**3.Data Sharing and Collaboration in Snowflake**

Snowflake provides robust data sharing capabilities that allow organizations to share data securely and seamlessly between accounts without requiring the data to be copied or moved. This feature is known as **Secure Data Sharing** and enables users to create **Shares** and provide access to specific data sets to other Snowflake accounts. Data sharing in Snowflake is read-only, ensuring that the consumer cannot alter the shared data.

Below, I will explain the process of data sharing and collaboration in Snowflake using the provided SALES\_DATA table.

**1. Scenario Overview**

Suppose there is a company named **TechyMart** that wants to share its SALES\_DATA with its regional partners to provide insights into their sales performance. The company has created a Snowflake table named SALES\_DATA, which contains the following columns:

* **Sale\_ID**: Unique identifier for each sale.
* **Sale\_Date**: Date of the sale.
* **Region**: Geographic region of the sale.
* **Product**: Product type.
* **Quantity**: Quantity sold.
* **Unit\_Price**: Price per unit of the product.
* **Customer\_Info**: JSON column that stores customer information such as name and email.

**2. Creating the Table and Inserting Sample Data**

We will use the provided SQL commands to create and populate the SALES\_DATA table:

sql

-- Create the SALES\_DATA table

CREATE OR REPLACE TABLE SALES\_DATA (

Sale\_ID INT AUTOINCREMENT,

Sale\_Date DATE,

Region STRING,

Product STRING,

Quantity INT,

Unit\_Price DECIMAL(10, 2),

Customer\_Info VARIANT

);

-- Insert sample data into SALES\_DATA table

INSERT INTO SALES\_DATA (Sale\_Date, Region, Product, Quantity, Unit\_Price, Customer\_Info)

SELECT

'2024-01-01', 'North', 'A', 10, 20.00, PARSE\_JSON('{"Name": "John", "Email": "john@example.com"}')

UNION ALL

SELECT

'2024-01-01', 'South', 'B', 5, 50.00, PARSE\_JSON('{"Name": "Sarah", "Email": "sarah@example.com"}')

UNION ALL

SELECT

'2024-01-02', 'North', 'A', 15, 20.00, PARSE\_JSON('{"Name": "Emily", "Email": "emily@example.com"}')

UNION ALL

SELECT

'2024-01-02', 'East', 'C', 8, 40.00, PARSE\_JSON('{"Name": "Mark", "Email": "mark@example.com"}');

**3. Creating and Granting Access to a Share**

TechyMart wants to share its SALES\_DATA with a partner company named **NorthRegionPartner**. This partner only needs to see the sales data for the **North** region.

To achieve this, TechyMart will create a **Secure Share** and include only the relevant table data in it.

**Step 1: Create a Share**

sql

-- Create a share named NORTH\_REGION\_SHARE

CREATE SHARE NORTH\_REGION\_SHARE;

**Step 2: Grant Access to the Table**

sql

-- Grant usage on the database and schema to the share

GRANT USAGE ON DATABASE MY\_DATABASE TO SHARE NORTH\_REGION\_SHARE;

GRANT USAGE ON SCHEMA MY\_DATABASE.PUBLIC TO SHARE NORTH\_REGION\_SHARE;

-- Grant select access on the SALES\_DATA table to the share

GRANT SELECT ON TABLE MY\_DATABASE.PUBLIC.SALES\_DATA TO SHARE NORTH\_REGION\_SHARE;

**Step 3: Restrict Access to North Region Data (Optional)**

If TechyMart wants to share only the North region data, they can create a secure view:

sql

-- Create a secure view for North region data only

CREATE SECURE VIEW MY\_DATABASE.PUBLIC.NORTH\_REGION\_SALES AS

SELECT \*

FROM MY\_DATABASE.PUBLIC.SALES\_DATA

WHERE Region = 'North';

-- Grant select access on the secure view to the share

GRANT SELECT ON VIEW MY\_DATABASE.PUBLIC.NORTH\_REGION\_SALES TO SHARE NORTH\_REGION\_SHARE;

**Step 4: Add Consumer Account**

TechyMart needs to add the consumer’s Snowflake account identifier to the share to complete the sharing process:

sql

-- Add the consumer account to the share

ALTER SHARE NORTH\_REGION\_SHARE ADD ACCOUNTS = ('partner\_account\_identifier');

**4. Accessing the Shared Data (Consumer Perspective)**

After the share is set up, the **NorthRegionPartner** can access the shared data from their Snowflake account:

**Step 1: Create a Database from the Share**

sql

-- Create a database from the shared data

CREATE DATABASE SALES\_SHARED\_DB FROM SHARE techymart\_account\_name.NORTH\_REGION\_SHARE;

**Step 2: Query the Shared Data**

The consumer can now query the shared data:

sql

-- Query the shared North region sales data

SELECT \* FROM SALES\_SHARED\_DB.PUBLIC.NORTH\_REGION\_SALES;

**5. Benefits of Snowflake Data Sharing**

* **Real-Time Data Sharing**: Partners can access the shared data in real-time, without requiring additional ETL processes or file transfers.
* **Secure and Controlled Access**: Data sharing is read-only, ensuring that consumers cannot alter the shared data.
* **Cost Efficiency**: Sharing does not involve copying or moving data, thus saving storage costs.
* **Centralized Management**: Data providers can manage and revoke access through centralized controls.

**6. Example Query Output**

The NORTH\_REGION\_SALES secure view will display the following results when queried:

| **Sale\_ID** | **Sale\_Date** | **Region** | **Product** | **Quantity** | **Unit\_Price** | **Customer\_Info** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2024-01-01 | North | A | 10 | 20.00 | {"Name": "John", "Email": "john@example.com"} |
| 3 | 2024-01-02 | North | A | 15 | 20.00 | {"Name": "Emily", "Email": "emily@example.com"} |

With this setup, NorthRegionPartner can perform analytics and reporting on the North region sales without accessing other regions’ data.

**7. Additional Considerations**

* **Data Sharing with External Organizations**: Data sharing can be done across Snowflake accounts even if the accounts are in different regions or cloud platforms.
* **Revoking Access**: Access to shared data can be revoked at any time using the ALTER SHARE command, removing specific accounts or the entire share.

This example demonstrates how TechyMart can share its sales data securely and efficiently with its regional partner using Snowflake’s Secure Data Sharing.

**4.Snow SQL**

**SnowSQL** is a command-line interface (CLI) client for interacting with Snowflake, a cloud-based data warehousing service. It provides a convenient way to execute SQL queries, manage database objects, and automate various operations in Snowflake directly from your terminal.

To use SnowSQL from your Windows PC, follow the steps below:

**Step 1: Download and Install SnowSQL**

1. **Go to the Snowflake Downloads Page:**
   * Visit the Snowflake Downloads page and download the **SnowSQL** installer for Windows.
2. **Install SnowSQL:**
   * Run the installer and follow the prompts.
   * By default, it installs SnowSQL in the directory: C:\Program Files\Snowflake SnowSQL\. You can choose a different location if needed.
3. **Verify the Installation:**
   * Open the Command Prompt (cmd) or Windows PowerShell and type:

snowsql --version

* + This command should display the installed version of SnowSQL, confirming that it's installed correctly.

**Step 2: Configure SnowSQL**

Before connecting to Snowflake, configure SnowSQL by creating a configuration file or using command-line parameters.

1. **Create a Configuration File:**
   * SnowSQL uses a configuration file located at %USERPROFILE%\.snowsql\config (for example: C:\Users\YourUsername\.snowsql\config).
   * Open or create the file and add the following details:

[connections.my\_account]

accountname = <your\_account\_name>

username = <your\_username>

password = <your\_password>

region = <region\_identifier>

* + Replace <your\_account\_name>, <your\_username>, <your\_password>, and <region\_identifier> with your Snowflake account details.

1. **Optional Configuration:**
   * You can also configure the default warehouse, role, and database in the configuration file:

warehouse = <your\_warehouse>

role = <your\_role>

database = <your\_database>

schema = <your\_schema>

**Step 3: Connecting to Snowflake**

1. **Connect Using Command-Line Parameters:**
   * Open the Command Prompt or PowerShell and run the following command:

snowsql -a <account\_name> -u <username> -r <role> -w <warehouse> -d <database> -s <schema>

Example:

snowsql -a xy12345.us-east-1 -u my\_user -r my\_role -w my\_warehouse -d my\_database -s public

* + You will be prompted for the password.

1. **Connect Using Configuration File:**
   * If you configured the .snowsql/config file with connection details, you can connect using the profile name:

snowsql -c my\_account

* + This command will connect using the profile my\_account specified in the configuration file.

**Step 4: Running SQL Queries**

Once connected, you can start running SQL queries. For example:

SELECT CURRENT\_VERSION();

This query checks the version of Snowflake you are connected to.

**Step 5: Additional SnowSQL Commands**

Here are some useful SnowSQL commands:

* **Help:** View all available commands and options.

snowsql --help

* **Run a SQL file:**

snowsql -f path\_to\_file.sql

This command executes the SQL commands stored in a .sql file.

* **Set Variables in SnowSQL:** You can set variables using !set within the SnowSQL CLI:

!set variable\_name='value';

**Step 6: Disconnecting and Exiting**

When you are done, simply type:

!exit

This command will disconnect you from the Snowflake session and exit the SnowSQL CLI.

**5.Snowpark in Snowflake**

Snowpark is a developer framework available in Snowflake that allows data engineers, data scientists, and developers to write scalable and efficient data transformations directly within Snowflake using languages like Python, Scala, and Java. This means you can execute data processing and machine learning tasks using familiar programming languages while leveraging the power of Snowflake's elastic compute resources.

**Why Use Snowpark?**

Snowpark is particularly useful in scenarios where complex transformations, data cleansing, or advanced analytics need to be performed. With Snowpark, developers can write complex data transformation logic in Python, Scala, or Java without moving data out of Snowflake, ensuring that data security, governance, and performance are maintained.  
  
  
**Additional Features of Snowpark**

1. **User-Defined Functions (UDFs)**: Snowpark allows you to define custom UDFs to perform complex transformations.
2. **DataFrame API**: Snowpark’s DataFrame API is similar to Pandas and Spark DataFrames, providing familiar functions like filter, select, groupBy, agg, and more.
3. **Machine Learning Integration**: Snowpark integrates well with Snowflake’s data science capabilities, making it easy to build and deploy machine learning models.

**6.Time Travel in Snowflake**

Snowflake’s Time Travel feature allows users to query, restore, and clone historical data at any point within a defined retention period (up to 90 days depending on the account configuration). This capability is beneficial when dealing with accidental deletions, updates, or other changes to the data.

Time Travel helps in scenarios such as:

1. Recovering data that was accidentally deleted or modified.
2. Creating reports based on historical snapshots of the data.
3. Analyzing changes to your data over time.

**Key Concepts of Time Travel:**

* **Historical Queries:** Querying a table at a specific point in time.
* **Data Recovery:** Restoring a table, schema, or database to a previous state.
* **Zero-Copy Cloning:** Creating clones of tables or databases at a specific point in time without consuming additional storage.

**Scenario: Recovering Data Using Time Travel**

Imagine that we accidentally deleted all data from the SALES\_DATA table using a DELETE statement. We can recover the data using Time Travel by referencing a past state of the table.

1. **Step 1: Check Current Data** Let's see what the current data looks like before the deletion:

SELECT \* FROM SALES\_DATA;

This will show the initial data in the table.

1. **Step 2: Perform an Accidental Deletion** We delete all rows from the SALES\_DATA table:

DELETE FROM SALES\_DATA;

Now, if we query the table again, it will show no records:

SELECT \* FROM SALES\_DATA;

1. **Step 3: Recover Data Using Time Travel** To recover the deleted data, we can query the table as it existed before the DELETE operation. We’ll use the AT clause with the BEFORE statement to specify the state before the deletion:

SELECT \*

FROM SALES\_DATA AT (BEFORE STATEMENT => (SELECT QUERY\_ID FROM TABLE(INFORMATION\_SCHEMA.QUERY\_HISTORY())

WHERE QUERY\_TEXT LIKE 'DELETE FROM SALES\_DATA%' LIMIT 1));

This query will show the data as it was before the delete operation.

1. **Step 4: Restore Table to Previous State** If we want to permanently restore the table to the state before the deletion, we can use the UNDROP command:

UNDROP TABLE SALES\_DATA;

Alternatively, we can restore to a specific timestamp or query ID:

CREATE OR REPLACE TABLE SALES\_DATA AS

SELECT \*

FROM SALES\_DATA AT (BEFORE STATEMENT => (SELECT QUERY\_ID FROM TABLE(INFORMATION\_SCHEMA.QUERY\_HISTORY())

WHERE QUERY\_TEXT LIKE 'DELETE FROM SALES\_DATA%' LIMIT 1));

1. **Step 5: Verify Restored Data** Finally, we can verify the restored data by querying the table again:

SELECT \* FROM SALES\_DATA;

This will display the original data before the accidental deletion.

**Using Time Travel for Historical Analysis**

Let's say we want to analyze how the sales data looked like at a particular point in time. We can use the AT clause with a specific timestamp.

SELECT \*

FROM SALES\_DATA AT (TIMESTAMP => '2024-01-02T00:00:00');

**7.Integration with Power BI**

Integrating Snowflake with Power BI allows you to leverage the powerful data warehousing capabilities of Snowflake and the visualization and business intelligence features of Power BI. This integration enables users to create meaningful visualizations, dashboards, and reports based on data stored in Snowflake, facilitating better decision-making and insights. Below is a detailed overview of how to set up and use Snowflake with Power BI:  
There are two primary ways to connect Snowflake to Power BI:

1. **Native Connector**: Power BI has a native connector for Snowflake that simplifies the process of connecting. This method is recommended for most users due to its ease of use.
2. **ODBC Connection**: For more advanced scenarios or when additional customization is required, you can use the Snowflake ODBC driver to connect. This method provides more flexibility in controlling the connection settings.

**Steps for Connecting Snowflake to Power BI**

**Method 1: Using the Native Snowflake Connector**

1. **Open Power BI Desktop** and click on **Home** → **Get Data** → **More...**.
2. In the **Get Data** window, search for "Snowflake" and select **Snowflake**.
3. In the **Snowflake Connection** dialog:
   * **Server**: Enter your Snowflake account URL. This should be in the format: account\_name.region.cloudprovider.snowflakecomputing.com.
   * **Warehouse**: Select the warehouse to use for running queries.
   * **Database** and **Schema**: Choose the database and schema where the required tables reside.
4. **Sign in with your Snowflake credentials** (username and password) or use the **SAML Authentication** method if configured.
5. Once authenticated, navigate to the desired tables and **load** or **transform** the data using the Power Query Editor.
6. Once the data is loaded into Power BI, you can start creating visualizations and reports.



A screenshot of a computer

Description automatically generated