Data Pipelining: Retail

Sharan Shivamurthy

The design of this pipeline requires an interdisciplinary approach for requirement gathering. In order to ensure a robust and comprehensive solution, input from various departments—including marketing, data engineering, data science, IT security, and legal (for compliance with DSGVO regulations)—is essential. This collaborative effort will help align the CRM strategy with both business objectives and technical constraints. Additionally, the pipeline must incorporate cross-functional requirements such as data integrity, performance, security, and user experience to ensure a fully optimized system.

Data Architecture and Data Ingestion Flow

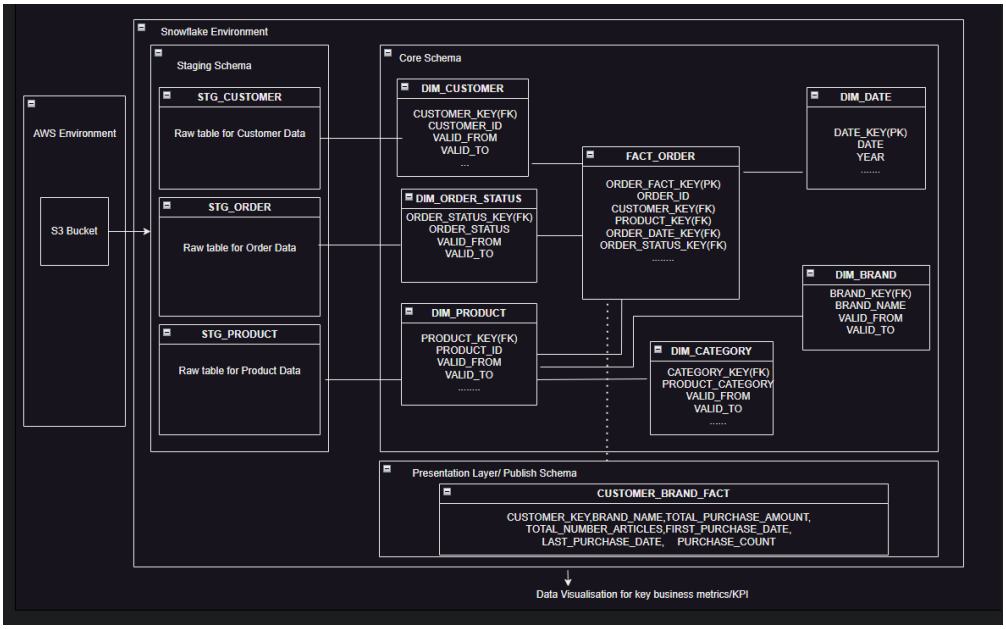

Figure 1: Data Architecture Diagram

The above diagram shows us the detailed Data Modelling / Architecture for the proposed pipeline.

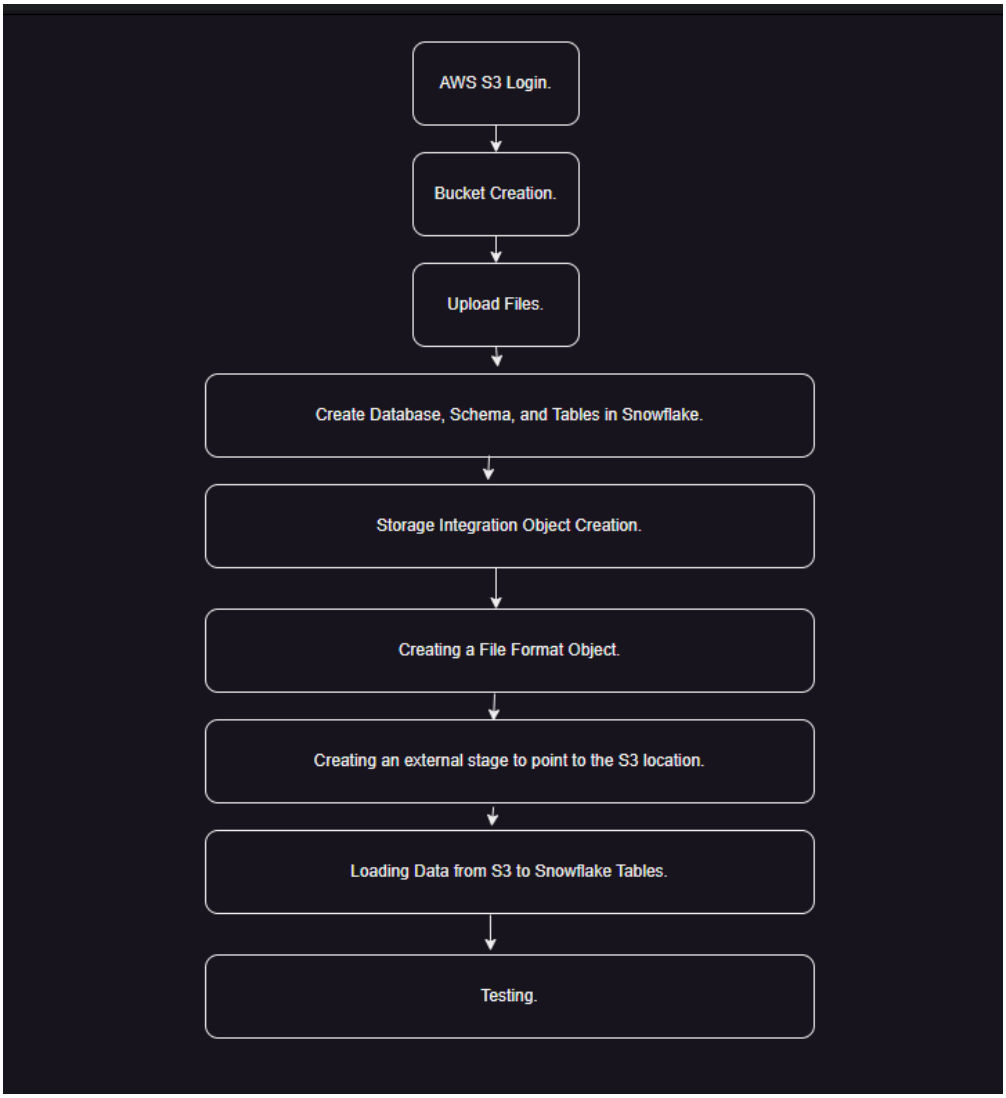

Figure 2: Data Ingestion Flow Diagram

1 Step 1: Loading the 3 files into the Snowflake Database

1.1 Step a: Uploading Files to AWS S3

First, upload the 3 data files to an AWS S3 bucket. Ensure that the necessary IAM policies are attached to the bucket, granting Snowflake access.

To upload files to S3, follow these steps:

1. **Login to AWS S3**:
   (a) Open AWS Management Console.
   (b) Go to S3.
2. **Create a Bucket**:
   (a) Click "Create Bucket" if we don't already have one.
   (b) Setting the bucket name and region.

3. **Upload Files**:

   (a) Inside the bucket, click "Upload" and choose the files we want to load into Snowflake.

1.2 Step b: Setting Up Snowflake Objects

1.2.1 Create Database, Schema, and Tables

First, log into Snowflake and create the necessary database, schema, and tables to store our data.

Set Role and Warehouse Context:

```
USE ROLE accountadmin;
USE WAREHOUSE compute_wh;
```

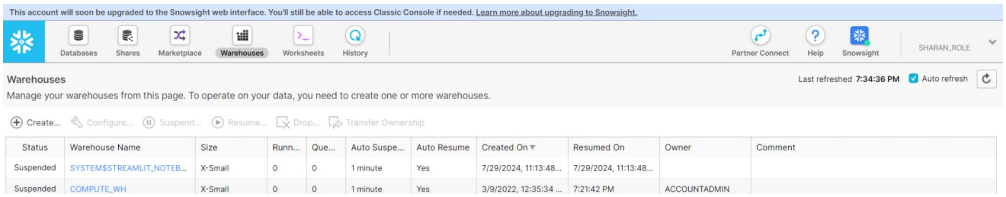

Figure 3: Snowflake Warehouse

Create the Database:

```
CREATE OR REPLACE DATABASE SPORT_RETAIL_CASE_STUDY
COMMENT = 'Database_for_analysis_of_Sports_Retail_Case_Study';
```

Create the Staging Schema:

```
CREATE OR REPLACE SCHEMA SPORT_RETAIL_CASE_STUDY.STAGING
COMMENT = 'Schema_hosting_the_raw_data';
```

Create the Staging Tables: Customer Table:

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.STAGING.
    STG_CUSTOMER (
CUSTOMER_ID NUMBER(38,0),
CUSTOMER_NAME VARCHAR,
GENDER VARCHAR,
BIRTH_DATE DATE,
EMAIL_ADDRESS VARCHAR,
COUNTRY VARCHAR,
ZIP_CODE NUMBER(38,0),
STREET VARCHAR,
STREET_NUMBER NUMBER(38,0)
) COMMENT = 'Raw_table_for_Customer_Data';
```

Order Table:

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
    (
ORDER_ID VARCHAR,
CUSTOMER_ID NUMBER(38,0),
PRODUCT_ID NUMBER(38,0),
ORDER_DATE DATE,
SALE_PRICE NUMBER(38,18),
SALE_PERCENTAGE NUMBER(38,1),
```

```
COUPON_VALUE NUMBER(38,0),
ORDER_STATUS VARCHAR,
NUMBER_ARTICLES NUMBER(38,0)
) COMMENT = 'Raw_table_for_Order_Data';
```

Product Table:

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.STAGING.
    STG_PRODUCT (
PRODUCT_ID NUMBER(38,0),
PRODUCT_NAME VARCHAR,
BRAND VARCHAR,
RRP NUMBER(38,2),
CURRENCY VARCHAR,
PRODUCT_CATEGORY VARCHAR,
PRODUCT_MAIN_CATEGORY VARCHAR
) COMMENT = 'Raw_table_for_Product_Data';
```

1.2.2 Create Storage Integration Object

The storage integration allows Snowflake to securely read data from the S3 bucket. We will need the AWS IAM role ARN for this step.

```
CREATE OR REPLACE STORAGE INTEGRATION s3_int
TYPE = EXTERNAL_STAGE
STORAGE_PROVIDER = 'S3'
STORAGE_AWS_ROLE_ARN = '<storage_aws_role_arn>'  -- Replace with
    AWS Role ARN
ENABLED = TRUE
STORAGE_ALLOWED_LOCATIONS = ('s3://sport-retail-case-study/')
COMMENT = 'Storage_Integration_Object_to_access_AWS_S3';
```

1.2.3 Create File Format Object

Define how files are formatted for Snowflake to read them properly:

```
CREATE OR REPLACE FILE FORMAT SPORT_RETAIL_CASE_STUDY.PUBLIC.
    CSV_FILEFORMAT
TYPE = 'CSV'
SKIP_HEADER = 1
FIELD_DELIMITER = ','
TRIM_SPACE = TRUE
FIELD_OPTIONALLY_ENCLOSED_BY = '"';
```

1.2.4 Create External Stage Object

Create an external stage pointing to the S3 location, configured with the appropriate file format and storage integration.

```
CREATE OR REPLACE STAGE SPORT_RETAIL_CASE_STUDY.PUBLIC.
    EXTERNAL_STAGE_CSV
URL = 's3://sport-retail-case-study/'
STORAGE_INTEGRATION = s3_int
FILE_FORMAT = SPORT_RETAIL_CASE_STUDY.PUBLIC.CSV_FILEFORMAT
COMMENT = '
    Stage_Object_pointing_to_external_stage_in_S3_holding_the_rawfiles
    ';
```

1.2.5 Load Data into Snowflake Using COPY Command

Before loading the data, verify that the files are accessible in the external stage:

```
LIST @SPORT_RETAIL_CASE_STUDY.PUBLIC.EXTERNAL_STAGE_CSV;
```

Load Customer Data:

```
COPY INTO SPORT_RETAIL_CASE_STUDY.STAGING.STG_CUSTOMER
FROM @SPORT_RETAIL_CASE_STUDY.PUBLIC.EXTERNAL_STAGE_CSV
FILES = ('customer.csv');
```

Load Order Data:

```
COPY INTO SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
FROM @SPORT_RETAIL_CASE_STUDY.PUBLIC.EXTERNAL_STAGE_CSV
FILES = ('order.csv');
```

Load Product Data:

```
COPY INTO SPORT_RETAIL_CASE_STUDY.STAGING.STG_PRODUCT
FROM @SPORT_RETAIL_CASE_STUDY.PUBLIC.EXTERNAL_STAGE_CSV
FILES = ('product.csv');
```

After the COPY INTO operation, we can check if the data was loaded correctly:

```
SELECT * FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_CUSTOMER;
```

This process outlines how to upload files to AWS S3, set up Snowflake for data loading, and load data from S3 into Snowflake using the COPY command.

2 Step 2: Creating the Data Warehouse Mid-Layer Model

2.1 Create a CORE Schema and Date Dimension Table

The CORE schema is where our clean, transformed data will reside. This step involves creating a CORE schema, which will host our cleaned and transformed data, and establishing dimension tables that follow a Slowly Changing Dimension Type 2 (SCD2) approach.

2.1.1 Create CORE Schema and DIM_DATE Table (SCD Type 0)

In this step, we will create the CORE schema, which will store dimension and fact tables. The DIM_DATE table is a common generic date dimension table, used for time-based analysis.

```
CREATE OR REPLACE SCHEMA SPORT_RETAIL_CASE_STUDY.CORE
COMMENT = 'Core_schema_for_cleaned_and_transformed_data';

CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.DIM_DATE (
DATE_KEY INT PRIMARY KEY, -- Unique date identifier (e.g. YYYYMMDD
    )
FULL_DATE DATE, -- Full date (e.g., 2023-09-28)
YEAR INT,
MONTH INT,
DAY INT,
QUARTER INT,
DAY_OF_YEAR INT,
WEEK_OF_YEAR INT
) COMMENT = 'Generic_Date_Dimension_Table';
```

2.1.2 Create Dimension Tables

Dimension tables describe the entities in our data warehouse, such as customers, brands, products, and order statuses. These tables will follow Slowly Changing Dimension Type 2 (SCD2) logic to track historical changes over time using VALID_FROM and VALID_TO fields.

Surrogate Key as the Primary Key Each dimension table will have a surrogate key (an auto-incremented integer) as the primary key, as opposed to using a business key like CUSTOMER_ID.

VALID_FROM and VALID_TO for SCD2 To implement SCD2, VALID_FROM and VALID_TO timestamps are used to track changes over time, with the current record having VALID_TO set to a distant future date (9999-12-31). This allows tracking changes in customer details, product information, etc., over time.

DIM_CUSTOMER (SCD Type 2): This table stores information about customers. Since the source data may contain multiple customers with the same name and email address, but with different details (like gender, birth date, or address), SCD2 will help track these changes as new records.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER (
CUSTOMER_KEY INT PRIMARY KEY AUTOINCREMENT,-- Surrogate key
CUSTOMER_ID NUMBER(38,0), -- Business key from source
CUSTOMER_NAME STRING,
EMAIL_ADDRESS STRING,
GENDER STRING,
BIRTH_DATE DATE,
ADDRESS STRING,
VALID_FROM TIMESTAMP, -- Start of validity for this record
VALID_TO TIMESTAMP DEFAULT '9999-12-31 00:00:00.000', -- End of validity
IS_ACTIVE BOOLEAN-- Active status for SCD2
) COMMENT = 'Dimension_table_for_Customer_Data_with_SCD2';
```

DIM_BRAND (SCD Type 2): This table stores information about product brands. SCD2 ensures that changes in brand information (e.g., name changes) are tracked.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.DIM_BRAND (
BRAND_KEY INT PRIMARY KEY AUTOINCREMENT,-- Surrogate key
BRAND_ID NUMBER(38,0),-- Business key from source
BRAND_NAME STRING,
VALID_FROM TIMESTAMP,
VALID_TO TIMESTAMP DEFAULT '9999-12-31 00:00:00.000',
IS_ACTIVE BOOLEAN
) COMMENT = 'Dimension_table_for_Product_Brands_with_SCD2';
```

DIM_CATEGORY (SCD Type 2): This table stores product category information.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.DIM_CATEGORY (
CATEGORY_KEY INT PRIMARY KEY AUTOINCREMENT, -- Surrogate key
CATEGORY_ID NUMBER(38,0), -- Business key from source
CATEGORY_NAME STRING,
VALID_FROM TIMESTAMP,
VALID_TO TIMESTAMP DEFAULT '9999-12-31 00:00:00.000',
IS_ACTIVE BOOLEAN
) COMMENT = 'Dimension_table_for_Product_Categories_with_SCD2';
```

DIM_PRODUCT (SCD Type 2): This table stores product information and has foreign key references to both DIM_BRAND and DIM_CATEGORY to maintain relationships between products, brands, and categories.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.DIM_PRODUCT (
PRODUCT_KEY INT PRIMARY KEY AUTOINCREMENT, -- Surrogate key
PRODUCT_ID NUMBER(38,0), -- Business key from source
PRODUCT_NAME STRING,
BRAND_KEY INT, -- Foreign key to DIM_BRAND
CATEGORY_KEY INT, -- Foreign key to DIM_CATEGORY
VALID_FROM TIMESTAMP,
VALID_TO TIMESTAMP DEFAULT '9999-12-31 00:00:00.000',
IS_ACTIVE BOOLEAN,
FOREIGN KEY (BRAND_KEY) REFERENCES SPORT_RETAIL_CASE_STUDY.CORE.
    DIM_BRAND(BRAND_KEY),
FOREIGN KEY (CATEGORY_KEY) REFERENCES SPORT_RETAIL_CASE_STUDY.CORE
    .DIM_CATEGORY(CATEGORY_KEY)
) COMMENT = '
    Dimension_table_for_Products_with_SCD2_and_foreign_keys_to_Brand_and_Categor
    ';
```

DIM_ORDER_STATUS (SCD Type 2): Order status details. This table stores the possible statuses of an order (e.g., "Pending", "Shipped", "Delivered"). It does not require as much historical tracking but follows the same SCD2 structure for consistency.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.
    DIM_ORDER_STATUS (
STATUS_KEY INT PRIMARY KEY AUTOINCREMENT,   -- Surrogate key
ORDER_STATUS STRING,
VALID_FROM TIMESTAMP,
VALID_TO TIMESTAMP DEFAULT '9999-12-31 00:00:00.000',
IS_ACTIVE BOOLEAN
) COMMENT = 'Dimension_table_for_Order_Status_with_SCD2';
```

2.1.3 Create FACT_ORDER Table (SCD Type 1)

The FACT_ORDER table will be the core fact table in our Data Warehouse (DWH) to store transactional information. This step involves creating a fact table with surrogate keys as the primary key, due to complications with the uniqueness of the business keys.

Why Use a Surrogate Key in the Fact Table?

- **ORDER_ID**: In our OLTP system, the ORDER_ID is not unique on its own, which means it cannot be used as a primary key.
- **Composite Primary Key**: Even combining ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, and ORDER_DATE_KEY does not guarantee uniqueness.
- **No Line Item Number**: There is no unique identifier (e.g., a line item number) to differentiate between multiple entries of the same product and the same order.
- **Measures Exclusion**: Measures like SALE_PRICE, SALE_PERCENTAGE, and COUPON_VALUE should not be part of a primary key.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER (
FACT_ORDER_KEY INT PRIMARY KEY AUTOINCREMENT, -- Surrogate key as
    primary key.
ORDER_ID STRING, -- Business key from source.
CUSTOMER_KEY INT, -- Foreign key reference to DIM_CUSTOMER, using
    the surrogate key from the customer dimension.
PRODUCT_KEY INT, -- Foreign key reference to DIM_PRODUCT, linking
    each order to the relevant product.
```

```
    ORDER_DATE DATE, -- The date the order was placed.
    SALE_PRICE NUMBER(10,2),-- Measure: sale price.
    SALE_PERCENTAGE NUMBER(5,2),-- Measure: sale percentage.
    COUPON_VALUE NUMBER(10,2), -- Measure: coupon value.
    NUMBER_ARTICLES INT,-- Number of articles in order.
    ORDER_STATUS_KEY INT,--  Foreign key reference to DIM_ORDER_STATUS
        , storing the current status of the order.
    ORDER_TIMESTAMP TIMESTAMP, -- Timestamp for when the order was
        placed
    LOAD_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Data load
        timestamp telling us when the data was loaded to the snowflake.
    ) COMMENT = '
        Fact_table_for_Orders_using_surrogate_key_and_foreign_keys_for_dimensions
        ';
```

2.1.4 Handling Data Entry Issues

There would be cases where multiple rows have identical attributes, suggesting data entry issues. We should have a data quality check in place in our ETL pipeline to correct these issues. While creating the table, we will still store these rows and handle them during transformation or reporting.

```
SELECT ORDER_ID, CUSTOMER_ID, PRODUCT_ID, COUNT(*) AS ROW_COUNT
FROM SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER
GROUP BY ORDER_ID, CUSTOMER_ID, PRODUCT_ID
HAVING COUNT(*) > 1;
```

This query will show any orders with multiple entries that need further inspection or cleanup.

2.1.5 Handling Differing Measures for the Same Order

We've noted that for some orders, attributes like SALE_PRICE, SALE_PERCENTAGE, and COUPON_VALUE may differ for the same ORDER_ID, CUSTOMER_ID, PRODUCT_ID, and ORDER_DATE. This likely indicates multiple discounts or promotions applied at different stages of the same order.

To handle this: These rows should still be stored separately in the fact table, allowing us to aggregate them later in analysis (e.g., summing the SALE_PRICE or counting the number of articles).

2.1.6 Source System Understanding

As we correctly mentioned, it's crucial to consult with source system experts to confirm the business logic and ensure that the data model accurately reflects real-world processes. A discussion with the source system team could help in adding a line item identifier in the source data to avoid this ambiguity in future data loads.

Key / Index Information:

- **Surrogate Keys:** All dimension tables use surrogate keys as primary keys (CUSTOMER_KEY, PRODUCT_KEY, etc.), which are auto-incrementing. These provide more stability and efficiency than business keys.
- **Foreign Keys:** Fact tables use foreign keys to link to dimension tables.
- **Indexes:** Implicit indexing is done on primary and foreign keys.
- **Clustering Keys:** For performance in Snowflake, we might add clustering based on commonly queried fields (e.g., ORDER_DATE_KEY).

2.1.7 Load Approach (Full, Delta):

- **Full Load:** Dimension tables like DIM_DATE, DIM_BRAND, and DIM_ORDER_STATUS can use full loads since they are relatively small and changes are rare or non-existent.
- **Delta Load:** Tables using SCD2 (like DIM_CUSTOMER and DIM_PRODUCT) should use delta loads. This involves identifying new records and changed records based on source updates, typically using a MERGE operation. The fact table FACT_ORDER would also use incremental (delta) loading, appending new transactions or updating existing ones.

3 Step 3: Loading Data into Mid-Layer Tables

3.1 Loading Dimension Tables

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER AS target
USING (
SELECT
CUSTOMER_ID,
CUSTOMER_NAME,
EMAIL_ADDRESS,
GENDER,
BIRTH_DATE,
ADDRESS
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_CUSTOMER
) AS source
ON target.CUSTOMER_ID = source.CUSTOMER_ID AND target.VALID_TO = '9999-12-31 00:00:00.000'
WHEN MATCHED AND (
target.CUSTOMER_NAME != source.CUSTOMER_NAME OR
target.EMAIL_ADDRESS != source.EMAIL_ADDRESS OR
target.GENDER != source.GENDER OR
target.BIRTH_DATE != source.BIRTH_DATE OR
target.ADDRESS != source.ADDRESS
) THEN
-- Mark existing record as inactive
UPDATE SET target.VALID_TO = CURRENT_TIMESTAMP, target.IS_ACTIVE = FALSE
WHEN NOT MATCHED THEN
-- Insert a new record with VALID_FROM as the current time and VALID_TO as the far future
INSERT (CUSTOMER_ID, CUSTOMER_NAME, EMAIL_ADDRESS, GENDER, BIRTH_DATE, ADDRESS, VALID_FROM, VALID_TO, IS
VALUES (source.CUSTOMER_ID, source.CUSTOMER_NAME, source.EMAIL_ADDRESS, source.GENDER, source.BIRTH_DATE
```

3.1.1 Source

The STG_CUSTOMER staging table contains the incoming raw customer data.

3.1.2 MATCHED

If a matching record exists in the dimension table (based on CUSTOMER_ID), but some of the attributes (e.g., CUSTOMER_NAME, EMAIL_ADDRESS) have changed, the existing record is marked as inactive (VALID_TO = CURRENT_TIMESTAMP), and IS_ACTIVE is set to FALSE.

3.1.3 NOT MATCHED

If no matching record exists or if there are changes, a new active record is inserted with VALID_FROM = CURRENT_TIMESTAMP and VALID_TO = '9999-12-31' to represent the current version of the data.

3.1.4 SCD2 Key Points

- **VALID_FROM and VALID_TO:** These fields track the time period when the record was valid.
- VALID_FROM = CURRENT_TIMESTAMP for new records.
- VALID_TO = '9999-12-31 00:00:00.000' for active records.
- **IS_ACTIVE:** This is used to flag whether the record is currently active. Only one record per entity should be active at a time.

Similarly, other dimension tables are also populated.

3.2 General Approach for All Dimension Tables

- **Check for Changes:** Each MERGE statement compares the incoming data with the current (active) record in the dimension table.
- If a change is detected in the attributes (e.g., name, status, category), the old record is updated by setting VALID_TO = CURRENT_TIMESTAMP and IS_ACTIVE = FALSE.
- **Insert New Records:** For records that don't exist or for which a change has been detected, a new row is inserted with VALID_FROM = CURRENT_TIMESTAMP and VALID_TO = '9999-12-31 00:00:00.000' to represent the active version.
- **SCD2 Implementation:** This pattern ensures that the history of changes is preserved, and only one active record exists per entity at any time.

3.3 DIM_PRODUCT

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.DIM_PRODUCT AS target
USING (
SELECT
PRODUCT_ID,
PRODUCT_NAME,
BRAND_ID,
CATEGORY_ID,
RRP,
CURRENCY
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_PRODUCT
) AS source
ON target.PRODUCT_ID = source.PRODUCT_ID AND target.VALID_TO = '9999-12-31 00:00:00.000'
WHEN MATCHED AND (
target.PRODUCT_NAME != source.PRODUCT_NAME OR
target.BRAND_ID != source.BRAND_ID OR
target.CATEGORY_ID != source.CATEGORY_ID OR
target.RRP != source.RRP OR
target.CURRENCY != source.CURRENCY
) THEN
-- Mark the existing record as inactive
UPDATE SET target.VALID_TO = CURRENT_TIMESTAMP, target.IS_ACTIVE = FALSE
WHEN NOT MATCHED THEN
-- Insert a new record with VALID_FROM as the current time and VALID_TO as the far future
INSERT (PRODUCT_ID, PRODUCT_NAME, BRAND_ID, CATEGORY_ID, RRP, CURRENCY, VALID_FROM, VALID_TO, IS_ACTIVE
VALUES (source.PRODUCT_ID, source.PRODUCT_NAME, source.BRAND_ID, source.CATEGORY_ID, source.RRP, source
```

3.4 DIM_BRAND

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.DIM_BRAND AS target
USING (
SELECT
BRAND_ID,
BRAND_NAME
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_BRAND
) AS source
ON target.BRAND_ID = source.BRAND_ID AND target.VALID_TO = '9999-12-31 00:00:00.000'
WHEN MATCHED AND target.BRAND_NAME != source.BRAND_NAME THEN
-- Mark existing record as inactive if the brand name has changed
UPDATE SET target.VALID_TO = CURRENT_TIMESTAMP, target.IS_ACTIVE = FALSE
WHEN NOT MATCHED THEN
-- Insert new record for the brand
INSERT (BRAND_ID, BRAND_NAME, VALID_FROM, VALID_TO, IS_ACTIVE)
VALUES (source.BRAND_ID, source.BRAND_NAME, CURRENT_TIMESTAMP, '9999-12-31 00:00:00.000', TRUE);
```

3.5 DIM_CATEGORY

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.DIM_CATEGORY AS target
USING (
SELECT
CATEGORY_ID,
CATEGORY_NAME,
PARENT_CATEGORY_ID
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_CATEGORY
) AS source
ON target.CATEGORY_ID = source.CATEGORY_ID AND target.VALID_TO = '9999-12-31 00:00:00.000'
WHEN MATCHED AND (
target.CATEGORY_NAME != source.CATEGORY_NAME OR
target.PARENT_CATEGORY_ID != source.PARENT_CATEGORY_ID
) THEN
-- Mark existing record as inactive if category info has changed
UPDATE SET target.VALID_TO = CURRENT_TIMESTAMP, target.IS_ACTIVE = FALSE
WHEN NOT MATCHED THEN
-- Insert new record for the category
INSERT (CATEGORY_ID, CATEGORY_NAME, PARENT_CATEGORY_ID, VALID_FROM, VALID_TO, IS_ACTIVE)
VALUES (source.CATEGORY_ID, source.CATEGORY_NAME, source.PARENT_CATEGORY_ID, CURRENT_TIMESTAMP, '9999-1
```

3.6 DIM_ORDER_STATUS

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.DIM_ORDER_STATUS AS target
USING (
SELECT
ORDER_STATUS_ID,
ORDER_STATUS
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER_STATUS
) AS source
ON target.ORDER_STATUS_ID = source.ORDER_STATUS_ID AND target.VALID_TO = '9999-12-31 00:00:00.000'
WHEN MATCHED AND target.ORDER_STATUS != source.ORDER_STATUS THEN
-- Mark existing record as inactive if the order status has changed
UPDATE SET target.VALID_TO = CURRENT_TIMESTAMP, target.IS_ACTIVE = FALSE
WHEN NOT MATCHED THEN
-- Insert new record for the order status
```

```
INSERT (ORDER_STATUS_ID, ORDER_STATUS, VALID_FROM, VALID_TO, IS_ACTIVE)
VALUES (source.ORDER_STATUS_ID, source.ORDER_STATUS, CURRENT_TIMESTAMP, '9999-12-31 00:00:00.000', TRUE
```

3.7 Loading Data into the FACT Table

The fact table contains transactional data (e.g., orders) and foreign key references to the dimension tables. Each time data is loaded into the fact table, it's important to:

- Append new records without duplicating data.
- Lookup foreign keys from the dimension tables, ensuring that only the active records from the dimension tables are referenced.

3.7.1 MERGE for FACT Table to Prevent Duplicates

To ensure that the fact table is loaded without inserting duplicates, a MERGE statement is used. The fact table's primary key is typically a surrogate key, while the MERGE condition uses the natural keys (e.g., ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, etc.).

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER AS target
USING (
SELECT
SO.ORDER_ID,
DC.CUSTOMER_KEY,
DP.PRODUCT_KEY,
DD.DATE_KEY AS ORDER_DATE_KEY,
SO.SALE_PRICE,
SO.SALE_PERCENTAGE,
SO.COUPON_VALUE,
DOS.ORDER_STATUS_KEY,
SO.NUMBER_ARTICLES
FROM
SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER SO
JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER DC
ON SO.CUSTOMER_ID = DC.CUSTOMER_ID AND DC.VALID_TO = '9999-12-31 00:00:00.000'
) AS source
ON target.ORDER_ID = source.ORDER_ID
WHEN MATCHED THEN
UPDATE SET target.SALE_PRICE = source.SALE_PRICE, target.SALE_PERCENTAGE = source.SALE_PERCENTAGE, targ
WHEN NOT MATCHED THEN
INSERT (ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, ORDER_DATE_KEY, SALE_PRICE, SALE_PERCENTAGE, COUPON_VALUE,
VALUES (source.ORDER_ID, source.CUSTOMER_KEY, source.PRODUCT_KEY, source.ORDER_DATE_KEY, source.SALE_PR
```

Source

The data is pulled from the STG_ORDER staging table, and the appropriate foreign keys are looked up from the dimension tables (e.g., DIM_CUSTOMER, DIM_PRODUCT, DIM_DATE, etc.).

JOINS

Each dimension table (DIM_CUSTOMER, DIM_PRODUCT, etc.) is joined to ensure that the current (active) records are referenced (VALID_TO = '9999-12-31 00:00:00.000').

MATCHING CONDITION

The MERGE condition checks whether a record with the same ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, and ORDER_DATE_KEY already exists in the fact table. If it exists, no action is taken (i.e., no duplicates are inserted). If no matching record exists, the new record is inserted.

Appending Data with Foreign Key Lookups

The key point when loading into the fact table is that foreign keys from the dimension tables are always current. This ensures that the fact table references the latest version of each entity in the dimension tables. The join condition filters by the VALID_TO = '9999-12-31 00:00:00.000' to select only active records from the dimension tables.

No Updates in Fact Tables

Unlike dimension tables, fact tables typically don't undergo updates. Instead, data is appended with each load, meaning that every new transaction (e.g., a new order) is inserted as a new row. The use of MERGE ensures that we avoid inserting duplicate records from the same load.

Importance of Loading Order in ETL Process

In a typical Data Warehouse ETL (Extract, Transform, Load) process, the dimension tables (dims) are loaded first, followed by the fact table. Here's why this order is important:

1. Dimension Tables Loaded First

- **Reason:** Dimension tables contain the descriptive or categorical attributes (e.g., customer, product, date, order status) that fact tables reference via foreign keys (surrogate keys).
- **Surrogate Keys:** When loading the fact table, we need the surrogate keys from the dimension tables to ensure proper relationships between the fact and dimension data.
- **SCD2 Handling:** If we're using Slowly Changing Dimensions Type 2 (SCD2), the dimensions must be up-to-date and active records must be correctly flagged before fact data can reference them.

2. Fact Table Loaded After Dimensions

- **Reason:** The fact table contains transactional data (e.g., orders, sales) and references the dimensions through foreign keys. To properly load the fact table, the dimensions must be already populated so that the fact data can link to the correct surrogate keys.
- **Lookups:** When loading the fact table, foreign key lookups are done against the dimension tables to ensure the fact table rows reference the most current dimension data (e.g., CUSTOMER_KEY, PRODUCT_KEY, DATE_KEY).

Next Steps

In this section, we outline the following steps to further enhance the ETL process:

1. **Monitoring and Logging:** Implement logging mechanisms to monitor the ETL process for errors and performance.
2. **Data Quality Checks:** Add data validation steps to ensure that data quality is maintained throughout the ETL pipeline.
3. **Performance Optimization:** Regularly review and optimize SQL queries for better performance, especially as the data volume grows.
4. **Documentation:** Keep documentation updated to reflect any changes in the ETL process or data structure.

Step 4: Ensuring High Data Quality in the Mid Layer Tables and Stability of the Load Process

a) Data Quality Checks in the Staging Layer

Data quality checks are crucial to ensuring that the data loaded into our Data Warehouse (DWH) is accurate, complete, and consistent. These checks are typically performed at the Staging Layer before the data is loaded into the dimension and fact tables.

Null Value Checks

Key fields like `customer_id`, `order_id`, and `product_id` should not have null values. Null values in these fields would break the relationships between fact and dimension tables, leading to incorrect analysis or missing data.

```
Check for nulls in STG_ORDER:
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE ORDER_ID IS NULL
OR CUSTOMER_ID IS NULL
OR PRODUCT_ID IS NULL;

Check for nulls in STG_CUSTOMER:
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_CUSTOMER
WHERE CUSTOMER_ID IS NULL;

Check for nulls in STG_PRODUCT:
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_PRODUCT
WHERE PRODUCT_ID IS NULL;
```

Solution: If null values are found, we may either reject the records with nulls or move them to another correction table before loading into the dimension or fact tables.

Duplicate Data Checks

Duplicate rows in staging tables can lead to over-counting in our fact tables and incorrect reporting. Ensuring that primary keys like `customer_id`, `order_id`, and `product_id` are unique is important.

```
Check for duplicate rows in STG_ORDER:
SELECT ORDER_ID, CUSTOMER_ID, PRODUCT_ID, COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
GROUP BY ORDER_ID, CUSTOMER_ID, PRODUCT_ID
HAVING COUNT(*) > 1;

Check for duplicate rows in STG_CUSTOMER:
SELECT CUSTOMER_ID, COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_CUSTOMER
GROUP BY CUSTOMER_ID
HAVING COUNT(*) > 1;

Check for duplicate rows in STG_PRODUCT:
SELECT PRODUCT_ID, COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_PRODUCT
GROUP BY PRODUCT_ID
HAVING COUNT(*) > 1;
```

Solution: If duplicates are found, investigate and deduplicate the records in the staging tables before loading them into the mid-layer or final tables.

Data Type Validation

Ensuring the data types in the staging tables match the expected types is critical to prevent errors during data transformations and loading into the final tables. For example, dates should be stored as `DATE` or `TIMESTAMP` types, and numeric fields should be properly formatted.

```
Check for invalid numeric values in STG_ORDER (e.g., SALE_PRICE, SALE_PERCENTAGE):
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE TRY_CAST(SALE_PRICE AS NUMBER) IS NULL
OR TRY_CAST(SALE_PERCENTAGE AS NUMBER) IS NULL;

Check for invalid date formats in STG_ORDER (e.g., ORDER_DATE):
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE TRY_CAST(ORDER_DATE AS DATE) IS NULL;
```

Solution: If any invalid data types are found, we can correct the invalid data or cast the data to the correct types as needed before proceeding with the data load.

Business Logic Checks

These checks ensure that the data follows the business rules and makes sense from a business perspective. For example, `SALE_PRICE` should always be positive, `SALE_PERCENTAGE` should not exceed 100%, and coupon values should be within expected ranges.

```
Check that SALE_PRICE is positive in STG_ORDER:
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE SALE_PRICE <= 0;

Check that SALE_PERCENTAGE is within the range [0, 100]:
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE SALE_PERCENTAGE < 0 OR SALE_PERCENTAGE > 100;
```

```
Check that COUPON_VALUE is within reasonable boundaries (between 0 and 70):
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE COUPON_VALUE < 0 OR COUPON_VALUE > 75;
```

Solution: If any rows violate business rules, we can reject, flag, or correct the records before loading them into the fact and dimension tables.

Automation of Data Quality Checks

To ensure the stability of the load process, these data quality checks should be automated and run as part of the ETL workflow:

- **Scheduled Checks:** Run these checks after data is loaded into the staging tables and before data is loaded into the mid-layer or fact tables.
- **Error Reporting:** If any of the checks fail, the system should flag the records, log the errors, and notify the relevant team to correct the issues.
- **Conditional Load:** Prevent the data load from continuing if data quality thresholds are not met (too many nulls or duplicates).

b) Load Process Stability

Ensuring the stability of the load process is critical in large-scale Data Warehouse projects. We want to minimize the amount of data being processed, log errors, and ensure that the load process can be re-run without causing issues such as data duplication or inconsistency.

Incremental Loads Using Change Data Capture (CDC)

Incremental loading ensures that only new or changed data is processed during the ETL load, improving performance and reducing unnecessary data processing. Instead of reloading the entire dataset, Change Data Capture (CDC) tracks only those records that have changed (inserted, updated, or deleted).

Approaches to Implement CDC in Snowflake:

- **Timestamp-based CDC:** Track new or modified rows based on a timestamp (e.g., `last_modified_date` or `load_date`).
- **Key-based CDC:** Track changes based on a unique key and compare it to the existing data.

Since our `STG_ORDER` table does not have a column like `LOAD_DATE` or `LAST_MODIFIED_DATE` to track changes, we will need to either:

- Add a timestamp column to track when each record was last modified or loaded, or
- Use an alternative approach for incremental loading, such as a comparison of existing data in the `FACT_ORDER` table.

Adding a LOAD_DATE Column to STG_ORDER

Add a `LOAD_DATE` column to capture when each record is loaded into the staging table.

```
ALTER TABLE SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
ADD LOAD_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
```

Incremental Load and Error Logging

Incremental Load Using LOAD_DATE

When new records are inserted into STG_ORDER, the LOAD_DATE column will automatically capture the timestamp of when the data was inserted.

We can then implement an incremental load by using LOAD_DATE to identify new or updated records:

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER AS target
USING (
SELECT *
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE LOAD_DATE > (SELECT MAX(LOAD_DATE) FROM SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER)
) AS source
ON target.ORDER_ID = source.ORDER_ID
AND target.CUSTOMER_ID = source.CUSTOMER_ID
AND target.PRODUCT_ID = source.PRODUCT_ID
AND target.ORDER_DATE = source.ORDER_DATE
WHEN MATCHED THEN
UPDATE SET target.SALE_PRICE = source.SALE_PRICE,
target.SALE_PERCENTAGE = source.SALE_PERCENTAGE,
target.COUPON_VALUE = source.COUPON_VALUE
WHEN NOT MATCHED THEN
INSERT (ORDER_ID, CUSTOMER_ID, PRODUCT_ID, ORDER_DATE, SALE_PRICE, SALE_PERCENTAGE, COUPON_VALUE, NUMBE
VALUES (source.ORDER_ID, source.CUSTOMER_ID, source.PRODUCT_ID, source.ORDER_DATE, source.SALE_PRICE, s
```

In this query, only records where the LOAD_DATE is greater than the maximum LOAD_DATE already in FACT_ORDER are processed. This ensures that only new or changed records are loaded incrementally.

Incremental Load Without LOAD_DATE and Using Key Comparison

If we cannot add a LOAD_DATE, we can implement incremental loads by comparing the staging table (STG_ORDER) with the existing data in the FACT_ORDER table based on the primary keys and key fields (ORDER_ID, CUSTOMER_ID, PRODUCT_ID, etc.).

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER AS target
USING (
SELECT *
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
) AS source
ON target.ORDER_ID = source.ORDER_ID
AND target.CUSTOMER_ID = source.CUSTOMER_ID
AND target.PRODUCT_ID = source.PRODUCT_ID
AND target.ORDER_DATE = source.ORDER_DATE
AND target.SALE_PRICE = source.SALE_PRICE
AND target.COUPON_VALUE = source.COUPON_VALUE
AND target.SALE_PERCENTAGE = source.SALE_PERCENTAGE
WHEN NOT MATCHED THEN
INSERT (ORDER_ID, CUSTOMER_ID, PRODUCT_ID, ORDER_DATE, SALE_PRICE, SALE_PERCENTAGE, COUPON_VALUE, NUMBE
VALUES (source.ORDER_ID, source.CUSTOMER_ID, source.PRODUCT_ID, source.ORDER_DATE, source.SALE_PRICE, s
```

In this case, the MERGE statement compares all relevant fields to detect changes and ensure only new or updated records are inserted.

Error Logging

To implement error logging for the STG_ORDER table, we can create an error log table and capture various types of errors, such as missing or invalid data.

Step A: Create Error Log Table

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.LOG.ERROR_LOG (
ERROR_TIMESTAMP TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
ERROR_TYPE STRING,
SOURCE_TABLE STRING,
RECORD_ID STRING,
ERROR_MESSAGE STRING
);
```

Step B: Capture Errors During the Load

Null Value Error Example (ORDER_ID, CUSTOMER_ID, or PRODUCT_ID should not be null):

```
INSERT INTO SPORT_RETAIL_CASE_STUDY.LOG.ERROR_LOG (
ERROR_TYPE, SOURCE_TABLE, RECORD_ID, ERROR_MESSAGE
)
SELECT
'Null Value Error' AS ERROR_TYPE,
'STG_ORDER' AS SOURCE_TABLE,
ORDER_ID AS RECORD_ID,
'ORDER_ID, CUSTOMER_ID, or PRODUCT_ID cannot be null'
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE ORDER_ID IS NULL OR CUSTOMER_ID IS NULL OR PRODUCT_ID IS NULL;
```

Invalid Data Type Error Example (SALE_PRICE should be a valid number):

```
INSERT INTO SPORT_RETAIL_CASE_STUDY.LOG.ERROR_LOG (
ERROR_TYPE, SOURCE_TABLE, RECORD_ID, ERROR_MESSAGE
)
SELECT
'Invalid Data Type Error' AS ERROR_TYPE,
'STG_ORDER' AS SOURCE_TABLE,
ORDER_ID AS RECORD_ID,
'Invalid SALE_PRICE format'
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
WHERE TRY_CAST(SALE_PRICE AS NUMBER) IS NULL;
```

With error logging in place, any invalid or missing data will be logged into the ERROR_LOG table for later review.

Batch Re-Runnability and Data Validations

Batch Re-Runnability (Rollback and Retry Mechanism)

For batch re-runnability, we need to ensure that the process can be re-run safely in case of failure without introducing duplicate data or inconsistency. This can be achieved using transactions, merge logic, and control tables.

Step A: Use Transactions for Safe Rollback

Start a transaction before the batch load:

```
BEGIN TRANSACTION;
```

Execute our batch load (similar to the `MERGE` statement used before):

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER AS target
USING (SELECT * FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER) AS source
ON target.ORDER_ID = source.ORDER_ID
WHEN NOT MATCHED THEN
INSERT (ORDER_ID, CUSTOMER_ID, PRODUCT_ID, ORDER_DATE, SALE_PRICE, SALE_PERCENTAGE, COUPON_VALUE, NUMBE
VALUES (source.ORDER_ID, source.CUSTOMER_ID, source.PRODUCT_ID, source.ORDER_DATE, source.SALE_PRICE, s
```

Commit the transaction if successful:

```
COMMIT;
```

Rollback the transaction in case of failure:

```
ROLLBACK;
```

Step B: Track Batch Status in a Control Table

Create a control table to track batch status:

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.LOG.BATCH_CONTROL (
BATCH_ID STRING,
BATCH_START_TIMESTAMP TIMESTAMP,
BATCH_END_TIMESTAMP TIMESTAMP,
STATUS STRING
);
```

Insert batch status before starting the load:

```
INSERT INTO SPORT_RETAIL_CASE_STUDY.LOG.BATCH_CONTROL (BATCH_ID, BATCH_START_TIMESTAMP, STATUS)
VALUES ('BATCH_2024_09_28', CURRENT_TIMESTAMP, 'IN_PROGRESS');
```

Update the batch status upon completion:

```
UPDATE SPORT_RETAIL_CASE_STUDY.LOG.BATCH_CONTROL
SET BATCH_END_TIMESTAMP = CURRENT_TIMESTAMP, STATUS = 'COMPLETED'
WHERE BATCH_ID = 'BATCH_2024_09_28';
```

This ensures that the batch process can track its progress and detect whether a batch has already been successfully completed or needs to be re-run due to a failure.

Data Validations in CORE Layer

After loading the data into the CORE layer (mid-layer), it is essential to validate that the data in both dimension (DIM) and fact (FACT) tables is accurate, without duplicates, and that the foreign keys in the fact tables correctly map to the corresponding dimension tables. This step ensures that:

- The dimension (DIM) and fact (FACT) tables do not contain duplicate records.
- Foreign key references in the fact tables correctly map to the corresponding dimension tables (e.g., FACT_ORDER to DIM_CUSTOMER, DIM_PRODUCT, etc.).

a. Ensuring No Duplicates in DIM and FACT Tables

Why Remove Duplicates? Duplicates in dimension and fact tables can cause incorrect aggregations and metrics during analysis and reporting. The uniqueness of records ensures consistency and correctness in the data model.

Approach 1: Use MERGE or UPSERT for Deduplication We can use `MERGE` operations to ensure that duplicate records are not introduced when inserting or updating data in the dimension and fact tables. This approach ensures that each row is unique based on a set of keys (such as CUSTOMER_ID, PRODUCT_ID, etc.).

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER AS target
USING (
SELECT DISTINCT CUSTOMER_ID, CUSTOMER_NAME, EMAIL_ADDRESS, GENDER, BIRTH_DATE, ADDRESS
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_CUSTOMER
) AS source
ON target.CUSTOMER_ID = source.CUSTOMER_ID
WHEN NOT MATCHED THEN
INSERT (CUSTOMER_ID, CUSTOMER_NAME, EMAIL_ADDRESS, GENDER, BIRTH_DATE, ADDRESS, VALID_FROM, VALID_TO, IS
VALUES (source.CUSTOMER_ID, source.CUSTOMER_NAME, source.EMAIL_ADDRESS, source.GENDER, source.BIRTH_DATE
```

This `MERGE` operation:

- Inserts new customer records if they don't already exist in `DIM_CUSTOMER`.
- Ensures no duplicates by checking the `CUSTOMER_ID` and inserting only if the record doesn't exist.

```
MERGE INTO SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER AS target
USING (
SELECT DISTINCT ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, ORDER_DATE, SALE_PRICE, COUPON_VALUE, SALE_PERCENTA
FROM SPORT_RETAIL_CASE_STUDY.STAGING.STG_ORDER
) AS source
ON target.ORDER_ID = source.ORDER_ID
AND target.CUSTOMER_KEY = source.CUSTOMER_KEY
AND target.PRODUCT_KEY = source.PRODUCT_KEY
AND target.ORDER_DATE = source.ORDER_DATE
WHEN NOT MATCHED THEN
INSERT (ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, ORDER_DATE, SALE_PRICE, COUPON_VALUE, SALE_PERCENTAGE, NUM
VALUES (source.ORDER_ID, source.CUSTOMER_KEY, source.PRODUCT_KEY, source.ORDER_DATE, source.SALE_PRICE,
```

This `MERGE` ensures that:

- Only unique rows are inserted into `FACT_ORDER` by checking for duplicates based on `ORDER_ID`, `CUSTOMER_KEY`, `PRODUCT_KEY`, and `ORDER_DATE`.

Approach 2: Enforcing Uniqueness with UNIQUE Constraints Snowflake allows us to define UNIQUE constraints on columns.

Adding UNIQUE Constraints to DIM_CUSTOMER:

```
ALTER TABLE SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER
ADD CONSTRAINT UNIQUE_CUSTOMER UNIQUE (CUSTOMER_ID);
```

Adding UNIQUE Constraints to FACT_ORDER:

```
ALTER TABLE SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER
ADD CONSTRAINT UNIQUE_ORDER UNIQUE (ORDER_ID, CUSTOMER_KEY, PRODUCT_KEY, ORDER_DATE);
```

Data Validation and Batch Monitoring

Validating Foreign Key Relationships in FACT Tables

Why Foreign Key Validation?

The fact table contains foreign keys that reference the dimension tables (e.g., `CUSTOMER_KEY` in `FACT_ORDER` references `CUSTOMER_KEY` in `DIM_CUSTOMER`). It is crucial to ensure that these foreign keys are valid and exist in the dimension tables to maintain data integrity and ensure correct join operations.

Step 1: Validate Foreign Keys Using JOINs

We can check whether the foreign keys in the fact table correctly reference the dimension tables by using JOINs to look for missing or invalid references.

Check `CUSTOMER_KEY` in `FACT_ORDER`:

```
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER f
LEFT JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER d
ON f.CUSTOMER_KEY = d.CUSTOMER_KEY
WHERE d.CUSTOMER_KEY IS NULL;
```

If this query returns any results, it means that there are foreign key violations where a `CUSTOMER_KEY` in `FACT_ORDER` does not have a corresponding entry in `DIM_CUSTOMER`.

Check `PRODUCT_KEY` in `FACT_ORDER`:

```
SELECT COUNT(*)
FROM SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER f
LEFT JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_PRODUCT p
ON f.PRODUCT_KEY = p.PRODUCT_KEY
WHERE p.PRODUCT_KEY IS NULL;
```

Step 2: Handle Missing Foreign Key References

If foreign key violations are found (e.g., orphaned records in the fact table), we need to take corrective action:

- **Log the errors:** Insert the invalid records into an error log table for further analysis.
- **Reject invalid records:** Prevent records with missing foreign keys from being inserted into the fact table by modifying the ETL logic.

Logging Foreign Key Violations:

```
INSERT INTO SPORT_RETAIL_CASE_STUDY.LOG.ERROR_LOG (
ERROR_TIMESTAMP, ERROR_TYPE, SOURCE_TABLE, RECORD_ID, ERROR_MESSAGE
)
SELECT CURRENT_TIMESTAMP, 'Foreign Key Violation', 'FACT_ORDER', ORDER_ID,
'Missing CUSTOMER_KEY in DIM_CUSTOMER'
FROM SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER f
LEFT JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER d
ON f.CUSTOMER_KEY = d.CUSTOMER_KEY;
```

Batch Monitoring

Monitoring the batch ETL process is crucial to ensure that the data pipeline meets Service Level Agreements (SLAs) for performance and reliability. Additionally, we should have automated systems in place to detect failures or long-running jobs and trigger alerts when necessary.

In this step, we focus on:

- Tracking batch performance, ensuring the process completes within the expected time frame.
- Implementing automated alerts in case of job failures or prolonged execution time.

a. Monitor Time Taken by the Batch

Why Monitor Batch Duration? Monitoring the time taken by each batch helps ensure that the ETL process is up to the agreed SLAs. If a batch takes too long, it can delay downstream processes, such as reporting or analytics, and may indicate inefficiencies or problems within the data pipeline.

Approach 1: Track Start and End Times for Each Batch We can track the start and end times of each batch and log this information in a batch control table.

Step 1: Create a Batch Control Table Create a control table that stores the batch's start and end times, along with the status of the batch.

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING (
BATCH_ID STRING,
BATCH_START_TIME TIMESTAMP,
BATCH_END_TIME TIMESTAMP,
DURATION_MINUTES NUMBER(10,2),
STATUS STRING,
ERROR_MESSAGE STRING
);
```

Step 2: Log the Batch Start Time At the beginning of the batch, insert the start time and set the status to `"IN_PROGRESS."`

```
INSERT INTO SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING (
BATCH_ID, BATCH_START_TIME, STATUS
)
VALUES ('BATCH_2024_09_28', CURRENT_TIMESTAMP, 'IN_PROGRESS');
```

Step 3: Log the Batch End Time and Calculate Duration At the end of the batch, update the record with the batch's end time, calculate the duration, and set the status to `"COMPLETED."`

```
UPDATE SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING
SET BATCH_END_TIME = CURRENT_TIMESTAMP,
DURATION_MINUTES = DATEDIFF(MINUTE, BATCH_START_TIME, CURRENT_TIMESTAMP),
STATUS = 'COMPLETED'
WHERE BATCH_ID = 'BATCH_2024_09_28';
```

Step 4: Identify Long-Running Batches We can query the `BATCH_MONITORING` table to identify any batches that exceed the acceptable time limit (based on SLA).

```
SELECT BATCH_ID, BATCH_START_TIME, BATCH_END_TIME, DURATION_MINUTES
FROM SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING
WHERE DURATION_MINUTES > 60; -- Example: SLA is 60 minutes
```

b. Automated Alerts for Failures or Long-Running Jobs

Why Automate Alerts? Automated alerts allow for proactive monitoring of the ETL process. If a job fails or runs longer than expected, the system can notify the relevant team to address the issue promptly. These alerts can help prevent disruptions to downstream processes, such as reporting or business analytics.

Approach 1: Error Alerts in Case of Failures We can trigger alerts when a batch fails or encounters an error during execution.

Step 1: Log Batch Failures If the batch fails at any point, update the `BATCH_MONITORING` table with an error message and set the status to `"FAILED."`

```
UPDATE SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING
SET STATUS = 'FAILED',
ERROR_MESSAGE = 'Data validation error in STG_ORDER'
WHERE BATCH_ID = 'BATCH_2024_09_28';
```

Step 2: Trigger Alerts via Snowflake Tasks Snowflake allows us to create tasks that can run SQL queries on a schedule or in response to certain events. We can set up a task that checks for failed batches and sends alerts.

Create a Task to Check for Failed Batches:

```
CREATE OR REPLACE TASK MONITOR_FAILED_BATCHES
WAREHOUSE = 'COMPUTE_WH'
SCHEDULE = '5 MINUTE' -- Runs every 5 minutes
AS
SELECT BATCH_ID, ERROR_MESSAGE
FROM SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING
WHERE STATUS = 'FAILED';
```

Step 3: Long-Running Job Alerts We can similarly trigger alerts for jobs that exceed the expected duration.

```
CREATE OR REPLACE TASK MONITOR_LONG_RUNNING_JOBS
WAREHOUSE = 'COMPUTE_WH'
SCHEDULE = '5 MINUTE' -- Runs every 5 minutes
AS
SELECT BATCH_ID, DURATION_MINUTES
FROM SPORT_RETAIL_CASE_STUDY.LOG.BATCH_MONITORING
WHERE STATUS = 'IN_PROGRESS'
AND DATEDIFF(MINUTE, BATCH_START_TIME, CURRENT_TIMESTAMP) > 60; -- Example: 60-minute SLA
```

This task will monitor for long-running jobs and can be configured to trigger an alert if a batch exceeds the SLA.

Approach 2: Use External Tools for Advanced Monitoring We can also integrate our Snowflake ETL process with external monitoring tools such as:

- **AWS CloudWatch:** For monitoring SQL execution times and failures.
- **Datadog:** For monitoring and alerts on job performance, failures, or delays.
- **Airflow:** For orchestrating and monitoring data pipelines with built-in alerting features.

By connecting Snowflake with these tools, we can set up more detailed monitoring, real-time alerts for our workflows.

Step 5: Designing the 3rd Layer (Presentation Layer) of the Data Warehouse

The `PUBLISH` schema serves as the final layer in the data warehouse optimized for reporting.

Creating the PUBLISH Schema

To begin, we will create the schema for our data mart tables:

```
CREATE OR REPLACE SCHEMA SPORT_RETAIL_CASE_STUDY.PUBLISH
COMMENT = 'Schema for data mart tables';
```

Creating the CUSTOMER_BRAND_FACT Table

The CUSTOMER_BRAND_FACT table will store aggregated data about customer purchases by brand. It is optimized for reporting, with pre-aggregated measures such as total purchase amount and the first/last purchase dates.

Table Definition

```
CREATE OR REPLACE TABLE SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT (
CUSTOMER_KEY         INT,               -- 'Surrogate key from DIM_CUSTOMER',
BRAND_NAME           VARCHAR,
TOTAL_PURCHASE_AMOUNT NUMBER(38,18),
TOTAL_NUMBER_ARTICLES NUMBER(38,0),
FIRST_PURCHASE_DATE  DATE,
LAST_PURCHASE_DATE   DATE,
PURCHASE_COUNT       INT,
PRIMARY KEY (CUSTOMER_KEY, BRAND_NAME)
);
```

Primary Key

We are using a composite primary key (CUSTOMER_KEY, BRAND_NAME). This design ensures uniqueness across this combination and makes sense as the fact table aggregates data by both customer and brand.

Column Descriptions

- CUSTOMER_KEY: References the customer dimension (DIM_CUSTOMER).
- BRAND_NAME: Comes from the brand dimension (DIM_BRAND).

Inserting Data into CUSTOMER_BRAND_FACT

The INSERT INTO query aggregates the relevant data from the FACT_ORDER table and joins it with the DIM_PRODUCT, DIM_BRAND, and DIM_DATE tables to summarize customer purchase behavior by brand.

```
INSERT INTO SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
(CUSTOMER_KEY, BRAND_NAME, TOTAL_PURCHASE_AMOUNT, TOTAL_NUMBER_ARTICLES, FIRST_PURCHASE_DATE, LAST_PURCH
SELECT
F.CUSTOMER_KEY,
B.BRAND_NAME,
SUM(F.SALE_PRICE) AS TOTAL_PURCHASE_AMOUNT,
SUM(F.NUMBER_ARTICLES) AS TOTAL_NUMBER_ARTICLES,
MIN(D.CALENDAR_DATE) AS FIRST_PURCHASE_DATE,
MAX(D.CALENDAR_DATE) AS LAST_PURCHASE_DATE,
COUNT(DISTINCT F.ORDER_ID) AS PURCHASE_COUNT
FROM
SPORT_RETAIL_CASE_STUDY.CORE.FACT_ORDER F
JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_PRODUCT P ON F.PRODUCT_KEY = P.PRODUCT_KEY
AND P.VALID_TO='9999-12-31 00:00:00.000'::timestamp_ntz
JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_BRAND B ON B.BRAND_KEY = P.BRAND_KEY
AND B.VALID_TO='9999-12-31 00:00:00.000'::timestamp_ntz
JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_DATE D ON D.DATE_KEY = F.ORDER_DATE_KEY
GROUP BY F.CUSTOMER_KEY, B.BRAND_NAME;
```

Aggregation Descriptions

- `SUM(F.SALE_PRICE)`: Computes the total purchase amount for each customer-brand combination.
- `SUM(F.NUMBER_ARTICLES)`: Total number of items purchased for the brand.
- `MIN(D.CALENDAR_DATE)`: Captures the first purchase date.
- `MAX(D.CALENDAR_DATE)`: Captures the last purchase date.
- `COUNT(DISTINCT F.ORDER_ID)`: Number of unique orders placed for that brand by the customer.

Table Relationships

The relationships between tables are as follows:

- `FACT_ORDER F`: The transactional data containing sales and order information.
- `DIM_PRODUCT P`: Product dimension to join on `PRODUCT_KEY` and access the related brand (`BRAND_KEY`).
- `DIM_BRAND B`: Brand dimension to get `BRAND_NAME`.
- `DIM_DATE D`: Date dimension to reference the order date (`ORDER_DATE_KEY`) and retrieve the calendar date for the first and last purchase.

Data Integrity Considerations

- `DISTINCT` in `COUNT(DISTINCT F.ORDER_ID)` ensures that multiple items in the same order don't result in double counting.
- The `GROUP BY` clause aggregates data at the customer-brand level, ensuring that the summarization is accurate.

Performance Considerations

Indexes and Clustering

To optimize queries that filter by customer or brand, we can cluster the `CUSTOMER_BRAND_FACT` table on the `CUSTOMER_KEY`.

```
ALTER TABLE SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
CLUSTER BY (CUSTOMER_KEY);
```

Materialized Views

If the query to populate the fact table is frequently run, we might consider creating a materialized view to store the aggregated results ahead of time. This would improve query performance when generating the data for `CUSTOMER_BRAND_FACT`.

Considerations for the Reporting Layer

Once the data is loaded into the `CUSTOMER_BRAND_FACT` table, it is ready to be utilized by reporting and BI tools. This table is optimized for queries such as:

- Customer behavior analysis: How much has each customer spent on different brands?
- Brand performance analysis: What are the top-performing brands in terms of revenue and orders?

Security Considerations

Since this is the PUBLISH schema, we need to implement proper role-based access control (RBAC). Only users or roles that need access to these data mart tables should have the necessary permissions.

```
GRANT SELECT ON SCHEMA SPORT_RETAIL_CASE_STUDY.PUBLISH TO ROLE REPORTING_USER;
```

Suggestions for the CRM Department

The CUSTOMER_BRAND_FACT table enables the CRM department to design and execute targeted marketing campaigns based on customer behavior, brand loyalty, and purchase trends. Here are some suggestions for CRM based on the insights from the CUSTOMER_BRAND_FACT table:

a. Identifying Brand-Loyal Customers for Targeted Offers

By analyzing the CUSTOMER_BRAND_FACT table, CRM teams can identify customers who have consistently purchased products from specific brands over time. These brand-loyal customers are excellent candidates for targeted offers related to the brands they prefer.

```
SELECT CUSTOMER_KEY, BRAND_NAME, PURCHASE_COUNT, TOTAL_PURCHASE_AMOUNT
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE PURCHASE_COUNT > 5 -- Customer made more than 5 purchases for this brand
AND TOTAL_PURCHASE_AMOUNT > 500; -- Customer spent more than €500 on this brand
```

Use Case: Offer exclusive brand-related promotions to customers who have shown a pattern of loyalty toward specific brands (like early access to new products or discounts on similar products).

b. Reactivating Inactive Customers Using LAST_PURCHASE_DATE

By querying the LAST_PURCHASE_DATE field, CRM can identify customers who have not made recent purchases from a brand and are potentially at risk of churn. Offering exclusive discounts or limited-time deals to these customers can help in reactivating them.

```
SELECT CUSTOMER_KEY, BRAND_NAME, LAST_PURCHASE_DATE
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE LAST_PURCHASE_DATE < CURRENT_DATE - INTERVAL '24' MONTH;
```

Use Case: Design customer retention campaigns for customers who haven't made a purchase in the last 24 months. Offer special deals or loyalty rewards to bring them back.

c. Identifying High-Value Customers for Personalized Campaigns

By analyzing the TOTAL_PURCHASE_AMOUNT column, CRM can identify high-value customers who have spent significant amounts on certain brands. These customers are ideal candidates for personalized marketing, offering early product releases, loyalty rewards, or personalized recommendations.

```
SELECT CUSTOMER_KEY, BRAND_NAME, TOTAL_PURCHASE_AMOUNT
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE TOTAL_PURCHASE_AMOUNT > 1000; -- High-value customers spending over €1000
```

Use Case: Run exclusive campaigns targeting high-value customers with offers like early access to new products, VIP events, or personalized product recommendations based on their purchase history.

d. Designing Campaigns for Bundle Deals and Discounts Based on Purchase Behavior

The PURCHASE_COUNT and TOTAL_PURCHASE_AMOUNT for specific brands provide insights into customer behavior. This can help CRM design campaigns offering bundle deals or discounts on related products from the same brand. This strategy encourages customers to purchase complementary or similar products.

```
SELECT CUSTOMER_KEY, BRAND_NAME, PURCHASE_COUNT, TOTAL_NUMBER_ARTICLES
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE PURCHASE_COUNT > 3 -- Customers who made more than 3 purchases for the brand
AND TOTAL_NUMBER_ARTICLES > 10; -- Purchased more than 10 articles from the brand
```

Use Case: Design bundle deals, offer discounts on related products, or promote new product launches from the same brand to customers with high engagement for specific brands.

This setup will help the CRM department create highly targeted marketing campaigns and maximize customer engagement based on purchase behavior and brand affinity.

e. Segmenting Customers Based on Purchase Frequency and Recency

By analyzing the FIRST_PURCHASE_DATE, LAST_PURCHASE_DATE, and PURCHASE_COUNT columns, CRM can classify customers into different segments such as new customers, repeat customers, and frequent buyers. This segmentation can help tailor marketing strategies for different customer groups.

- **New Customers:** Customers who made their first purchase in the last 6 months.
- **Repeat Customers:** Customers who made multiple purchases but are not frequent buyers.
- **Frequent Buyers:** Customers with a high purchase count over a long period.

```
SELECT CUSTOMER_KEY, BRAND_NAME, FIRST_PURCHASE_DATE, LAST_PURCHASE_DATE, PURCHASE_COUNT
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE FIRST_PURCHASE_DATE > CURRENT_DATE - INTERVAL '6' MONTH; -- New customers
```

Use Case: Offer loyalty programs or exclusive onboarding offers for new customers. For repeat and frequent customers, design campaigns that increase engagement, such as VIP programs or early access to sales.

f. Churn Prediction Using Purchase Patterns

Customers with decreasing purchase frequency or those whose last purchase date is far in the past are at risk of churn. CRM can analyze these patterns and offer incentives to prevent churn.

```
SELECT CUSTOMER_KEY, BRAND_NAME, LAST_PURCHASE_DATE, PURCHASE_COUNT
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE LAST_PURCHASE_DATE < CURRENT_DATE - INTERVAL '12' MONTH -- No purchase in the last 12 months
AND PURCHASE_COUNT < 5; -- Low purchase count
```

Use Case: Create a churn prevention strategy by targeting customers at risk of leaving with exclusive re-engagement offers, personalized emails, or limited-time promotions.

g. Cross-Selling and Up-Selling Opportunities

CRM can use purchase behavior from the CUSTOMER_BRAND_FACT table to identify cross-sell and up-sell opportunities. Customers who have purchased certain brands may be likely to purchase complementary products from related or premium brands.

```
SELECT CUSTOMER_KEY, BRAND_NAME, TOTAL_PURCHASE_AMOUNT, TOTAL_NUMBER_ARTICLES
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
WHERE TOTAL_PURCHASE_AMOUNT > 500 -- High spending on specific brand
AND TOTAL_NUMBER_ARTICLES > 3; -- Purchased multiple items from this brand
```

Use Case: Offer related products from complementary brands or encourage customers to buy higher-end or premium products through personalized product recommendations or bundle deals.

h. Customer Lifetime Value (CLTV) Calculation

Using the `TOTAL_PURCHASE_AMOUNT` from the `CUSTOMER_BRAND_FACT` table, CRM can calculate Customer Lifetime Value (CLTV) for each customer across different brands. This can help identify the most profitable customers and design high-value customer retention strategies.

```
SELECT CUSTOMER_KEY, SUM(TOTAL_PURCHASE_AMOUNT) AS LIFETIME_VALUE
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
GROUP BY CUSTOMER_KEY
HAVING SUM(TOTAL_PURCHASE_AMOUNT) > 1000; -- High-value customers (spent more than €1000)
```

Use Case: Design premium loyalty programs or early access promotions for high-CLTV customers. Focus on retaining these high-value customers by offering personalized experiences, exclusive discounts, or concierge services.

i. Brand Performance Analysis by Customer Segments

The `CUSTOMER_BRAND_FACT` table also enables the CRM team to evaluate brand performance by different customer segments (e.g., high spenders, frequent buyers, first-time buyers). This analysis helps tailor brand-specific campaigns or understand which brands resonate with specific customer groups.

```
SELECT BRAND_NAME, COUNT(CUSTOMER_KEY) AS CUSTOMER_COUNT, SUM(TOTAL_PURCHASE_AMOUNT) AS TOTAL_REVENUE
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
GROUP BY BRAND_NAME
ORDER BY TOTAL_REVENUE DESC;
```

Use Case: Identify top-performing brands and use this insight to focus marketing efforts on promoting those brands to different customer segments. Alternatively, create targeted campaigns for underperforming brands to boost their visibility and sales.

j. Geographical or Demographic Targeting

If additional demographic or geographical information is available in the `DIM_CUSTOMER` table (e.g., location, age group, or gender), CRM can target customers based on demographics or geography to design location-specific or demographically focused campaigns.

```
SELECT C.LOCATION, CB.BRAND_NAME, SUM(CB.TOTAL_PURCHASE_AMOUNT) AS TOTAL_PURCHASE_AMOUNT
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT CB
JOIN SPORT_RETAIL_CASE_STUDY.CORE.DIM_CUSTOMER C ON CB.CUSTOMER_KEY = C.CUSTOMER_KEY
GROUP BY C.LOCATION, CB.BRAND_NAME;
```

Use Case: Tailor marketing campaigns based on location (e.g., specific offers for urban vs. rural areas) or age groups (e.g., youth-targeted promotions for brands popular with younger customers).

k. Analyzing Purchase Patterns for Seasonal Campaigns

Using the FIRST_PURCHASE_DATE and LAST_PURCHASE_DATE from the CUSTOMER_BRAND_FACT table, CRM can identify seasonal purchase patterns. This information can be used to design seasonal promotions and plan marketing campaigns around peak buying seasons.

```
SELECT BRAND_NAME, EXTRACT(MONTH FROM FIRST_PURCHASE_DATE) AS PURCHASE_MONTH, SUM(TOTAL_PURCHASE_AMOUNT)
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
GROUP BY BRAND_NAME, EXTRACT(MONTH FROM FIRST_PURCHASE_DATE);
```

Use Case: Identify peak purchasing months for each brand and run seasonal campaigns or discounts to drive sales during high-demand periods. For example, launch specific promotions during the holiday season or summer months.

l. Identifying Product Preferences for Product Development

By analyzing the brand preferences and total purchase data in CUSTOMER_BRAND_FACT, CRM can provide feedback to the product development team. This can help in developing products that align with customer preferences and trends.

```
SELECT BRAND_NAME, SUM(TOTAL_PURCHASE_AMOUNT) AS TOTAL_REVENUE, COUNT(DISTINCT CUSTOMER_KEY) AS CUSTOMER
FROM SPORT_RETAIL_CASE_STUDY.PUBLISH.CUSTOMER_BRAND_FACT
GROUP BY BRAND_NAME
ORDER BY CUSTOMER_COUNT DESC;
```

Use Case: Use this data to identify popular brands and high-demand product categories, and collaborate with the product development team to design or launch new products that align with customer preferences.

Shortcuts in this Design

- **Error Handling and Logging:** In practice, data quality issues (e.g., invalid data types, null values, outliers) require extensive validation during the ETL process. This includes capturing and logging errors, notifying teams, and handling exceptions. For simplicity, I haven't included the full implementation of error logging or alerting mechanisms (which may involve external tools like Airflow, CloudWatch, or Snowflake Tasks).
- **Comprehensive Data Quality Rules:** The case study covers null checks, duplicate checks, and basic validation, but in a real project, we'd likely add additional rules like referential integrity checks, ensuring foreign keys are always valid, and business logic validation (e.g., verifying price ranges, ensuring product availability).
- **Optimized Performance Tuning:** In a real project, performance tuning (e.g., partitioning large tables, clustering keys in Snowflake) is critical to ensure scalability. For simplicity, I've not implemented optimizations like clustering, materialized views, or specific indexing strategies, which can significantly speed up complex queries.
- **Full ETL/ELT Pipelines:** The entire ETL/ELT pipeline (using tools like Airflow, DBT, or Matillion) would include scheduling, dependency management, and pipeline orchestration. I've described some batch processes in the case study but didn't provide the full automation.
- **Data Archiving and Auditing:** Real projects often involve data archiving, data retention policies, and audit trails to track the movement and transformation of data. I did not implement full data lineage and archiving mechanisms in this case study.
- **Detailed Role-Based Access Control (RBAC):** In a real project, security around who can access, read, or write to each schema or table would be enforced using role-based access control (RBAC). While I mention the importance of security, I didn't implement a full RBAC setup.

All the above mentioned architecture in this pipeline has to be in accordance to the DSGVO guidelines and regulations.