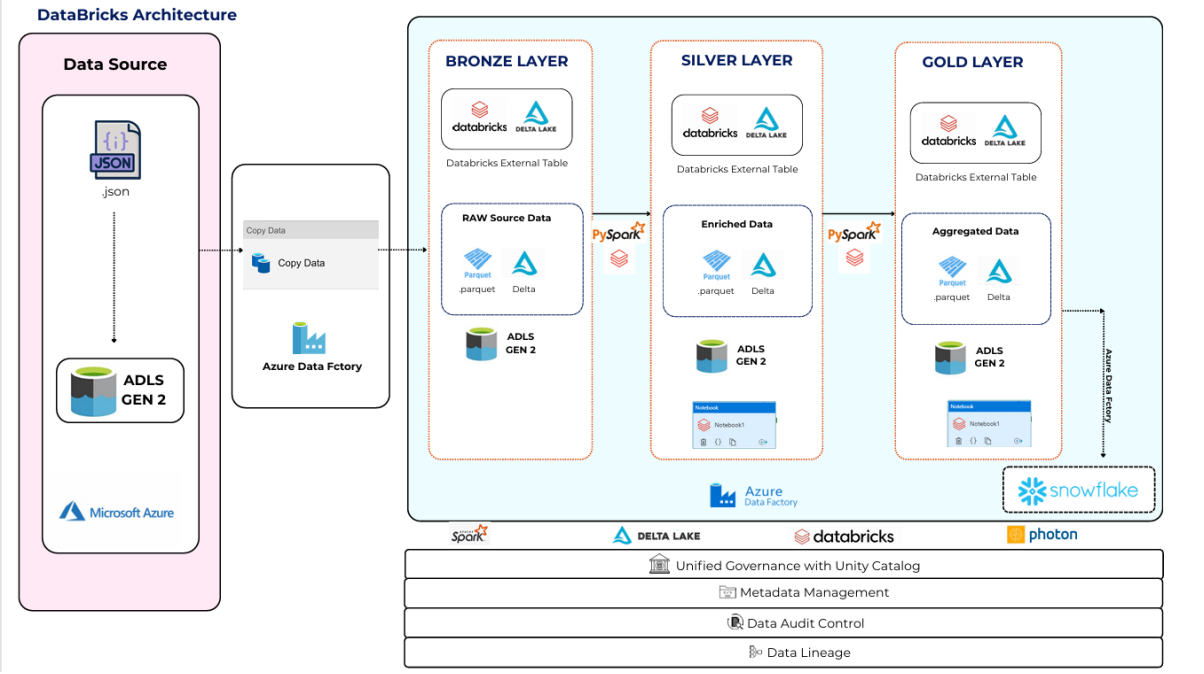
Databricks Internal Project

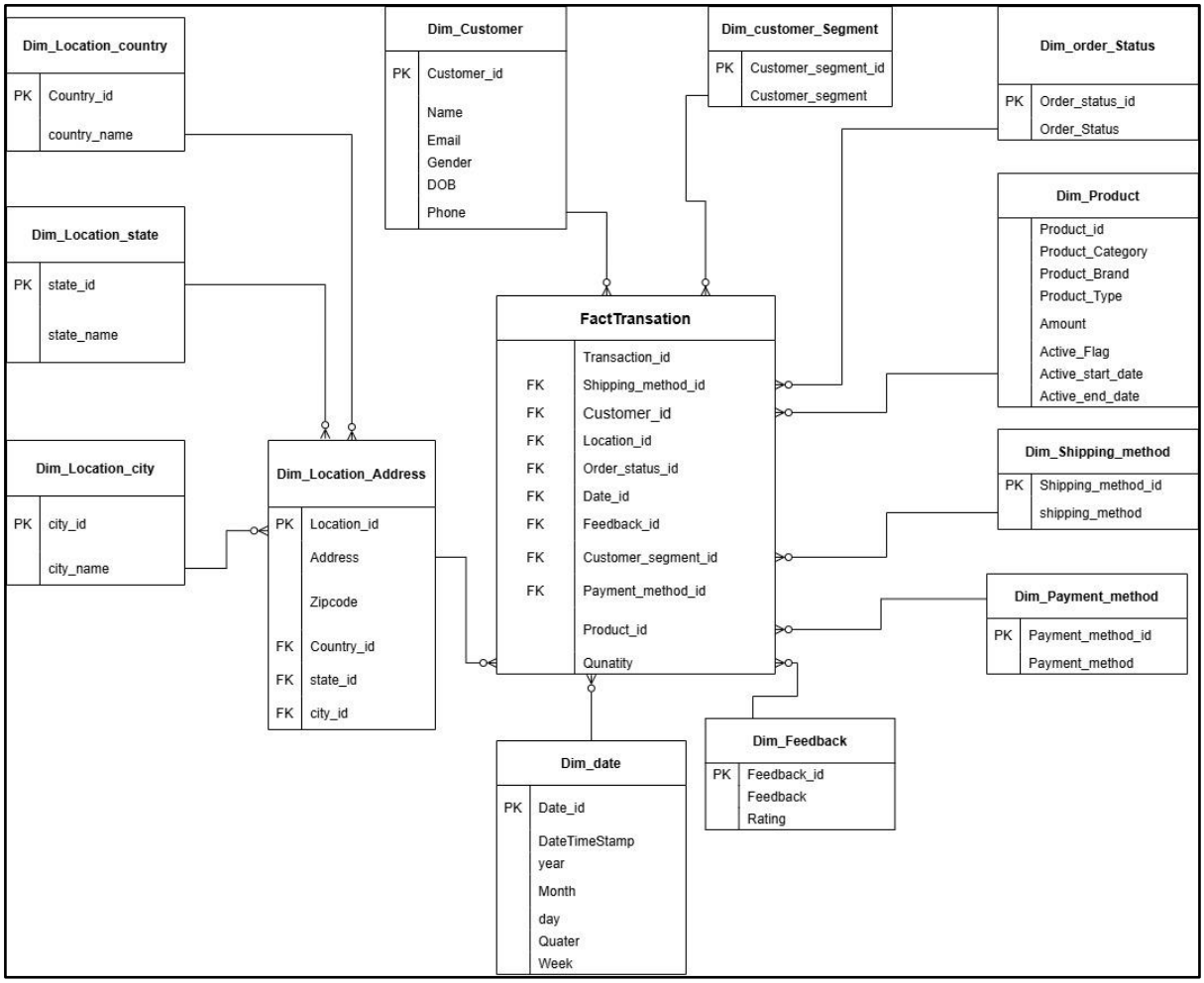
Project Overview

This project outlines the end-to-end architecture and implementation of a data processing pipeline using **Azure Databricks**, **Azure Data Lake Storage Gen2 (ADLS Gen2)**, **Snowflake**, and **Azure Data Factory (ADF)**.
The goal is to process raw data, clean and transform it, and generate **Key Performance Indicators (KPIs)** for business reporting and analytics.

Project Architecture



Data Model



Step 1: Data Ingestion using ADF (Copy Activity)

- Used **Azure Data Factory Copy Activity** to move data from another person's **ADLS Gen2** account to my own **ADLS Gen2** account.
- Source data is in **JSON format** and stored in a **folder path** (no specific file name provided).
- The Copy Activity reads all JSON files from the **source folder** and writes them to the **target folder**:
    - **Source Folder**: External ADLS Gen2 → retaildata/
    - **Target Folder**: My ADLS Gen2 → /bronze_layer/raw_data/
- No transformations applied at this stage. files are just moved as-is for further processing.
- This forms the **Bronze Layer** of the data lake.

Step 2: Processing Raw JSON and Creating Delta Table (Bronze Layer)

- Configured **Service Principal** and **External Location** in Azure Databricks to access the ADLS Gen2 storage folder.
- Set the folder path in the Bronze Layer where raw JSON files are stored
- Loaded the JSON files directly from this folder into a Spark DataFrame.
- Added two columns for incremental processing:
    - **Batch_id**: Unique identifier for each data batch
    - **Insert_timestamp**: Timestamp when data is loaded
- Created a **Delta table** named **retail_data_external** in the Bronze Layer from this DataFrame.
- Maintained a **batch tracking table** (**Batch_tracking_tbl**) to log batch information, insert timestamps, and processing status flags (Processed_flag).

Step 3: Silver Layer – Data Cleaning

- In the Silver Layer, only the data with Processed_flag = 'N' from the Bronze Delta table (retail_data_external) is selected for transformation.
- The following cleaning and transformation rules were applied to the data:
    - **Multiple Genders for One Customer:**
      For customers having multiple gender values, the most frequent gender (maximum count) was assigned uniformly to all records of that customer.
    - **Null Email Addresses:**
      Where customer email was null, the first non-null email recorded for that customer was propagated to those null entries.
    - **Customer Name Errors:**
      Removed any numeric characters appended to customer names using regular expressions to keep only valid alphabetic characters.
    - **Multiple Phone Numbers for One Customer:**
      Aggregated all unique phone numbers into a list per customer, removing duplicates.
    - **Missing Ratings but Present Feedback:**
      Where feedback exists but rating is missing, rating was updated based on feedback content.
    - **Missing Feedback but Present Rating:**
      Where rating exists but feedback is missing, feedback was inferred or updated based on the rating.
    - **Product Naming Errors:**
      Corrected product type naming inconsistencies by referencing product brand and category.
    - **Invalid Records Removal:**
      Removed 55 rows where both datetime stamp and amount were null, as these were invalid entries.

- The cleaned and transformed data was stored into a new DataFrame ready for further processing.

Step 3: Silver Layer – Dimension & Fact Table Creation

After the **Bronze Layer cleaning**, the **Silver Layer** focuses on structured transformation and loading of cleaned data into **Dimension** and **Fact** tables. Each dimension table is created by selecting relevant columns from the cleaned DataFrame and applying necessary transformations (e.g., hash token generation, SCD logic). To avoid duplicates and ensure upsert functionality, a reusable merge function is used.

The merge function checks whether a Delta table exists:

- If **not exists**, the data is **overwritten**.
- If **exists**, it performs a **merge operation** based on key columns to **upsert** data.

Table Name	Table Type	Source Cols	Transformation Logic	Merge Logic	Target Columns
Dim_Location_Country	Dimension	country_name	Lowercase + Hash on country_name	Merge on country_id	country_id, country_name
Dim_Location_State	Dimension	state_name	Lowercase + Hash	Merge on state_id	state_id, state_name
Dim_Location_City	Dimension	city_name	Lowercase + Hash	Merge on city_id	city_id, city_name
Dim_Customer_Segment	Dimension	customer_segment	Lowercase + Hash	Merge on customer_segment_id	customer_segment_id, customer_segment
Dim_Order_Status	Dimension	order_status	Lowercase + Hash	Merge on order_status_id	order_status_id, order_status
Dim_Shipping_Method	Dimension	shipping_method	Lowercase + Hash	Merge on shipping_method_id	shipping_method_id, shipping_method
Dim_Payment_Method	Dimension	payment_method	Lowercase + Hash	Merge on payment_method_id	payment_method_id, payment_method
Dim_Customer	Dimension	customer_id, name, email, gender, birthdate, phone_number	Merge: Update phone list if phone changes	Merge on customer_id	customer_id, name, email, gender, birthdate, phone_number_list
Dim_Location_Address	Dimension	address, zip, state, city, country	Hash: address+zip → location_id, state/city/country hash	Merge on location_id	location_id, state_id, city_id, country_id, address, zip

Dim_Feedback	Dimension	feedback, rating	Hash(feedback + rating) → feedback_id	Merge on feedback_id	feedback_id, feedback, rating
Dim_Product	Dimension	product_id, name, price, etc.	SCD Type 2: Maintain version history	Merge on product_id and end_date	product_id, name, price, valid_from, valid_to, is_current
Dim_Date	Dimension	datetime_timestamp	Extract: date, time, year, month, etc. + hash(datetime) → date_id	Merge on date_id	date_id, date, time, year, month, day, weekday, quarter
Fact_Transaction	Fact	All foreign keys and transaction details	Generate hash on lowercased values, reference all dim tables	Append only	All dimension IDs + quantity, transaction_id, date_id, feedback_id

Step:4 Silver Layer – Batch Tracking Table Update

After all Silver transformations are completed, the batch tracking table is updated using **merge**, setting the processing_flag = 'Y'. This helps maintain a consistent incremental load strategy and avoids reprocessing already transformed data.

Step: 5 Gold Layer – KPI logics

The **Gold Layer** in this Databricks project represents the final stage of data transformation, built upon the **Silver Layer**. It consolidates cleaned and enriched datasets from various sources to support business intelligence, reporting, and KPI generation. This layer uses **complete transactional data**, not just batched snapshots, ensuring that all KPIs are calculated on the most accurate and updated data available.

KPI 1: Customer Retention Rate (Returning Customers)

Description

This KPI tracks the **percentage of customers** who made repeat purchases in a year, compared to those who made purchases in the **previous year**. It helps evaluate how well the business is retaining customers year-over-year.

Approach

1. Extract distinct combinations of customer_id and year from the transaction data.
2. Group customers by year to get a yearly customer list.
3. Join current year customer lists with the previous year's list.
4. Identify how many customers from the previous year made purchases in the current year.
5. Calculate retention rate using the formula:

Retention Rate= (Previous Year Customers/Returning Customers) ×100

Pyspark code

```
fact_with_year = Fact_Transation.alias("f").join(
    Dim_Date.alias("d"),
    col("f.date_Id") == col("d.date_Id"),
    "inner"
).select(
    col("f.customer_id"),
    col("d.year")
).distinct()

#fact_with_year.show()
# list of customers per yeaR
customers_per_year = fact_with_year.groupBy("year") \
    .agg(collect_set("customer_id").alias("customers"))

#customers_per_year.show()

# Self join
current = customers_per_year.alias("curr")
previous = customers_per_year.alias("prev")

#Join on year difference one
retention_join = current.join(
    previous,
    col("curr.year") == col("prev.year") + 1
)
retention_join.show()

#Calculate retention rate
retention_rate = retention_join.select(
    col("curr.year").alias("year"),
    (size(array_intersect("curr.customers", "prev.customers")) /
size("prev.customers") * 100).alias("retention_rate_percent")
).orderBy("year")
retention_rate.show()
```

SQL Code

```
%sql

WITH fact_with_year AS (
   SELECT DISTINCT
      f.customer_id,
      d.year
```

```
    FROM dbx_internalproc.sliver_layer.Fact_Transation f
    JOIN dbx_internalproc.sliver_layer.dim_date d
      ON f.date_Id = d.date_Id
),
customers_per_year AS (
    SELECT
        year,
        COLLECT_SET(customer_id) AS customers
    FROM fact_with_year
    GROUP BY year
),
retention_join AS (
    SELECT
        curr.year AS year,
        array_intersect(curr.customers, prev.customers) AS returning_customers,
        prev.customers AS prev_customers
    FROM customers_per_year curr
    JOIN customers_per_year prev
      ON curr.year = prev.year + 1
)
SELECT
    year,
    SIZE(returning_customers) * 100.0 / SIZE(prev_customers) AS retention_rate_percent
FROM retention_join
ORDER BY year;
```

Output

	1²₃ year	.00 retention_rate_percent
1	2025	14.098073555166

KPI 2: Monthly Churn Rate (Customers Who Didn't Come Back)

Description

This KPI tracks the **percentage of customers** who made purchases in the **previous month** but did **not return** in the current month. It helps understand short-term customer retention trends and identify churn patterns.

Expected Output

- Month
- Churn Rate (in percentage)

Approach

1. Extract unique combinations of customer_id, year, and month from the transaction records.
2. Group customers by each month to build monthly customer sets.
3. Join current month with the previous month using both year and month logic.
4. Identify customers who existed in the previous month but not in the current one.
5. Calculate churn rate as:

   Churn Rate = (Lost Customers/ Previous Month Customers) *100

Pyspak code:

```
#Join Fact with Date to get customer_id and month
fact_with_month = Fact_Transation.alias("f").join(
    Dim_Date.alias("d"),
    col("f.date_Id") == col("d.date_Id"),
    "inner"
).select(
    col("f.customer_id"),
    col("d.year"),
    col("d.month")
).distinct()
fact_with_month.show()

# Group by year and month to collect customers
customers_per_month = fact_with_month.groupBy("year", "month") \
    .agg(collect_set("customer_id").alias("customers"))

customers_per_month.show()
#Self join
curr = customers_per_month.alias("curr")
prev = customers_per_month.alias("prev")

monthly_join = curr.join(
    prev,
    (
        (col("curr.year") == col("prev.year")) & (col("curr.month") ==
col("prev.month") + 1)
    ) |
```

```
    (
        (col("curr.year") == col("prev.year") + 1) & (col("curr.month") == 1) &
(col("prev.month") == 12)
    )
)
monthly_join.show()
# Calculate Churn Rate
from pyspark.sql.functions import array_except

churn_rate = monthly_join.select(
    col("curr.month").alias("month"),
    (size(array_except("prev.customers", "curr.customers")) / size("prev.customers")
* 100).alias("churn_rate_percent")
).orderBy("month")

churn_rate.show()
```

SQL Code:

```
%sql
WITH fact_with_year AS (
   SELECT DISTINCT
      f.customer_id,
      d.year,
      d.month
   FROM dbx_internalproc.sliver_layer.Fact_Transation f
   JOIN dbx_internalproc.sliver_layer.dim_date d
     ON f.date_Id = d.date_Id
),
customers_per_month AS (
   SELECT
      year,
      month,
      COLLECT_SET(customer_id) AS customers
   FROM fact_with_year
   GROUP BY year,month
),
churn_join AS (
   SELECT
      curr.year AS year,
      curr.month AS month,
      prev.customers AS prev_customers,
      curr.customers AS curr_customers
   FROM customers_per_month curr
```

```
   JOIN customers_per_month prev
     ON (curr.year = prev.year AND curr.month = prev.month + 1)
     OR (curr.year = prev.year + 1 AND curr.month = 1 AND prev.month = 12)

)
SELECT
   month,
   100 * (size(array_except(prev_customers, curr_customers)) / size(prev_customers)) AS
churn_rate_percent
FROM churn_join
ORDER BY month;
```

Output:

	month	churn_rate_percent
1	1	84.56909520739308
2	2	86.24225592822047
3	3	84.0726757904672
4	4	85.95920138888889
5	5	57.03764320785597
6	5	99.84704802691955
7	6	86.25239412640987
8	7	84.60842009959258
9	8	85.37592351151673
10	9	85.03487634749524
11	10	84.51063829787235
12	11	85.46099290780141
13	12	85.31928900592494

KPI: 3 : Product Return/ Cancelled Rate by Brand

Description

This KPI calculates the **percentage of orders that were either returned or cancelled**, grouped by **product brand**. It helps assess brand-level customer satisfaction and logistics efficiency.

Approach

1. Use the **Fact_Transation** table to access all transactions.
2. Join with:
   Dim_Product to get product brand.
   Dim_Order_Status to identify order outcomes (e.g., Returned, Cancelled).
3. For each brand:
   Count total orders.
   Count how many were either *Returned* or *Cancelled*.
4. Compute return rate:
5. *Return Rate (%) = (Returned or Cancelled OrdersTotal Orders)×100*

Pyspark code

```
fact_joined = Fact_Transation.alias("f") \
    .join(Dim_Product.alias("p"), col("f.product_id") == col("p.product_id"),
"inner") \
    .join(Dim_order_status.alias("s"), col("f.order_status_id") ==
col("s.order_status_id"), "inner") \
    .select(
        col("p.product_brand").alias("product_brand"),
        col("s.order_status").alias("order_status")
    )
#fact_joined.show()
from pyspark.sql.functions import count, when,round

brand_return_rate = fact_joined.groupBy("product_brand").agg(
    count("*").alias("total_orders"),
    count(when(col("order_status").isin("Returned", "Cancelled"),
True)).alias("returned_or_cancelled")
).withColumn(
    "return_rate",
    round((col("returned_or_cancelled") / col("total_orders") * 100),2)
)
# brand_return_rate.show()

brand_return_rate =
brand_return_rate.select("product_brand","return_rate").orderBy("return_rate")


brand_return_rate.show()
```

SQL Code:

```
%sql
with Brand_return as (
  SELECT
    p.product_brand AS product_brand,
    COUNT(*) AS total_orders,
```

```
   COUNT(CASE WHEN s.order_status IN ('Returned', 'Cancelled') THEN 1 END) AS
returned_or_cancelled,
   ROUND(
      COUNT(CASE WHEN s.order_status IN ('Returned', 'Cancelled') THEN 1 END) *
100.0 / COUNT(*),
      2
   ) AS return_rate
FROM dbx_internalproc.sliver_layer.Fact_Transation f
JOIN dbx_internalproc.sliver_layer.Dim_Product p ON f.product_id = p.product_id
JOIN dbx_internalproc.sliver_layer.Dim_Order_Status s ON f.order_status_id =
s.order_status_id
GROUP BY p.product_brand
ORDER BY return_rate
)
SELECT product_brand,return_rate from Brand_return;
```

Output

	product_brand	return_rate
1	IKEA	21.36
2	Bed Bath & Beyond	21.41
3	Zara	21.42
4	Adidas	21.45
5	Samsung	21.50
6	HarperCollins	21.54
7	Nestle	21.55
8	Home Depot	21.60
9	Apple	21.63
10	PenguinBooks	21.65
11	Nike	21.65
12	Amazon	21.78
13	Coca-Cola	21.87
14	Random House	21.97

KPI :4 Average Product Rating by Category

Description

This KPI calculates the **average customer rating** for products in each **product category**. It gives insight into **customer satisfaction** with different types of products and helps identify which categories are most appreciated by users.

Approach

1. Use the **Fact_Transation** table to get all transactions.
2. Join with:
    a. **Dim_Feedback** to get rating values.
    b. **Dim_Product** to access product category.
3. Group by product_category.
4. Compute the **average rating**, rounded to 2 decimal places.
5. Order by average rating to highlight top-rated categories.

Pyspark code

```
fact_joined = Fact_Transation.alias("T") \
    .join(Dim_Product.alias("p"), col("T.product_id") == col("p.product_id"),
"inner") \
    .join(Dim_Feedback.alias("f"), col("T.Feedback_id") == col("f.Feedback_id"),
"inner") \
    .select(
        col("p.product_category").alias("product_category"),
        col("f.rating").alias("rating")
    )
#fact_joined.show()
Average_Product_Rating =
fact_joined.groupBy("product_category").agg(round(avg("rating"),
2).alias("avg_rating")).orderBy("avg_rating")
Average_Product_Rating.show()
```

SQL code

```
%sql
SELECT
   p.product_category AS product_category,
   ROUND(AVG(fb.rating), 2) AS avg_rating
FROM dbx_internalproc.sliver_layer.Fact_Transation f
JOIN dbx_internalproc.sliver_layer.Dim_Feedback fb ON f.Feedback_id = fb.Feedback_id
JOIN dbx_internalproc.sliver_layer.Dim_Product p ON f.product_id = p.product_id
GROUP BY p.product_category
ORDER BY avg_rating;
```

Output

	product_category	avg_rating
1	Grocery	2.99
2	Electronics	2.99
3	Clothing	2.99
4	Home Decor	3
5	Books	3.02

KPI: 5 Payment Method Preference

Description

This KPI identifies the **most commonly used payment methods** by customers. It helps businesses understand which options are preferred, enabling optimization of the checkout process and improving customer satisfaction.

Approach

1. Use the **Fact_Transation** table to get all transactions.
2. Join with the **Dim_Payment_Method** table to get readable payment method names.
3. Count how often each payment method was used.
4. Sort the results by usage frequency in descending order.

Pyspark code

```
fact_joined = Fact_Transation.alias("T") \
    .join(Dim_payment_method.alias("p"), col("T.payment_method_Id") ==
col("p.payment_method_Id"), "inner") \
    .select(
        col("p.Payment_Method").alias("Payment_Method"),
    )
# fact_joined.show()
Payment_Method_Preference = fact_joined.groupBy("payment_method") \
                            .agg(count("*").alias("frequency")) \
                            .orderBy(col("frequency"))

Payment_Method_Preference.show()
```

SQL Code

```
%sql
```

```
select  p.payment_method, count(*) as frequency
from dbx_internalproc.sliver_layer.Fact_Transation T
join dbx_internalproc.sliver_layer.dim_payment_method P on p.payment_method_Id =
T.payment_method_Id group by p.payment_method order by frequency
```

Output

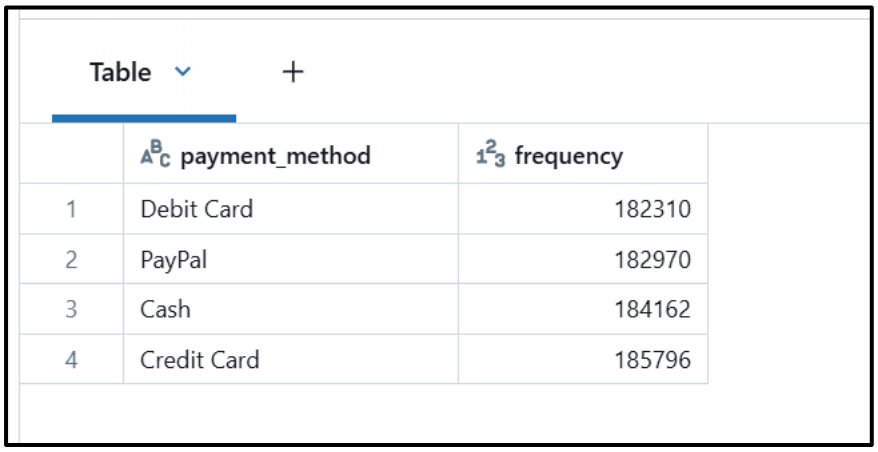


Table +

	payment_method	frequency
1	Debit Card	182310
2	PayPal	182970
3	Cash	184162
4	Credit Card	185796

KPI 6: Most Product Purchases by Day, Weekday, Month, and Year **Description**

This KPI identifies **when customers are most active in purchasing**, by analyzing transaction volumes across **various time frames**. It helps businesses:

1. Identify **peak purchase days**.
2. Understand **weekday behavior** (e.g., weekend vs. weekday).
3. Track **monthly trends** for seasonal insights.
4. Analyze **year-over-year** performance for long-term growth.

Approach

For each time frame, we join the Fact_Transation table with the Dim_Date table to access detailed date information. We then group and count transactions accordingly.

Pyspark code:

```
# Join Fact_Transation with Dim_Date
fact_with_date = Fact_Transation.alias("f").join(
    Dim_Date.alias("d"),
    col("f.date_Id") == col("d.date_Id"),
    "inner"
)

# DAY-WISE FREQUENCY
daywise = fact_with_date.groupBy("d.date") \
```

```
    .count() \
    .withColumnRenamed("count", "purchase_count") \
    .orderBy(col("purchase_count").desc())

# WEEKDAY FREQUENCY
weekday = fact_with_date.groupBy("d.weekday") \
    .count() \
    .withColumnRenamed("count", "purchase_count") \
    .orderBy(col("purchase_count").desc())

# MONTHLY FREQUENCY
monthly = fact_with_date.groupBy("d.month") \
    .count() \
    .withColumnRenamed("count", "purchase_count") \
    .orderBy("d.month")

# YEARLY FREQUENCY
yearly = fact_with_date.groupBy("d.year") \
    .count() \
    .withColumnRenamed("count", "purchase_count") \
    .orderBy("d.year")

# Show all
daywise.show()
weekday.show()
monthly.show()
yearly.show()
```

Sql Code:

```
%sql
-- Daywise
SELECT d.date, COUNT(*) AS purchase_count
FROM dbx_internalproc.sliver_layer.Fact_Transation f
JOIN dbx_internalproc.sliver_layer.Dim_Date d ON f.date_Id = d.date_Id
GROUP BY d.date
ORDER BY purchase_count DESC;
```

	date	purchase_count
1	2025-02-19	1222
2	2024-12-17	1215
3	2024-09-14	1205
4	2025-01-04	1204
5	2024-09-03	1195
6	2024-07-11	1173
7	2024-09-13	1166
8	2024-09-26	1166
9	2024-12-23	1164
10	2024-09-02	1162
11	2025-01-05	1160

```
%sql
-- Weekday
SELECT d.weekday, COUNT(*) AS purchase_count
FROM dbx_internalproc.sliver_layer.Fact_Transation f
JOIN dbx_internalproc.sliver_layer.Dim_Date d ON f.date_Id = d.date_Id
GROUP BY d.weekday
ORDER BY purchase_count DESC;
```

	weekday	purchase_count
1	Thursday	53147
2	Wednesday	52982
3	Sunday	52853
4	Monday	52500
5	Saturday	52310
6	Friday	52295
7	Tuesday	51532

```
%sql

-- Monthly
SELECT d.month, COUNT(*) AS purchase_count
FROM dbx_internalproc.sliver_layer.Fact_Transation f
JOIN dbx_internalproc.sliver_layer.Dim_Date d ON f.date_Id = d.date_Id
GROUP BY d.month
ORDER BY d.month;
```

	month	purchase_count
1	1	31740
2	2	28336
3	3	30906
4	4	29941
5	5	31355
6	6	29225
7	7	30917
8	8	31459
9	9	31538
10	10	30127
11	11	30420

```
%sql

-- Yearly
SELECT d.year, COUNT(*) AS purchase_count
FROM dbx_internalproc.sliver_layer.Fact_Transation f
JOIN dbx_internalproc.sliver_layer.Dim_Date d ON f.date_Id = d.date_Id
GROUP BY d.year
ORDER BY d.year;
```

	year	purchase_count
1	2024	254389
2	2025	113230

KPI 7: Q: Average Delivery Time by Shipping Method (in days)

Description

This KPI calculates the **average delivery time** for orders based on their **shipping method**, measured from the **order shipped date** to the **delivery date**. It provides insights into the efficiency of each shipping method over time and helps identify logistics bottlenecks.

Approach

1. Filter and extract records where order status is either **"Shipped"** or **"Delivered"**.

2. Join shipped and delivered records using customer ID, product ID, and shipping method ID.
3. Calculate the **number of days between shipped and delivered dates** using DATEDIFF.
4. Join with the **Dim_Date** and **Dim_Shipping_Method** tables to get descriptive time and shipping details.
5. Group the data by **year**, **month**, and **shipping method** to calculate the **average delivery duration**

Pyspark code

```
from pyspark.sql.functions import col, datediff, round, avg


# SHIPPED records
shipped_df = Fact_Transation.alias("T") \
    .join(Dim_order_status.alias("o"), col("T.order_status_Id") ==
col("o.order_status_Id")) \
    .filter(col("o.order_status") == "Shipped") \
    .select(
        col("T.customer_id"),
        col("T.product_Id"),
        col("T.shipping_method_Id"),
        col("T.date_Id").alias("shipped_date_id")
    )


# DELIVERED records
delivered_df = Fact_Transation.alias("T") \
    .join(Dim_order_status.alias("o"), col("T.order_status_Id") ==
col("o.order_status_Id")) \
    .filter(col("o.order_status") == "Delivered") \
    .select(
        col("customer_id"),
        col("product_Id"),
        col("shipping_method_Id"),
        col("T.date_Id").alias("delivered_date_id")
    )
delivered_df.show()


status_df = shipped_df.join(
    delivered_df,
    on=["customer_id", "product_Id", "shipping_method_Id"],
    how="inner"
)
#status_df.show()
status_df = status_df.alias("s") \
    .join(Dim_Date.alias("sd"), col("s.shipped_date_id") == col("sd.date_id")) \
    .join(Dim_Date.alias("dd"), col("s.delivered_date_id") == col("dd.date_id")) \
```

```
    .join(Dim_shipping_method.alias("sm"), col("s.shipping_method_Id") ==
col("sm.shipping_method_Id")) \
    .select(
        col("sm.shipping_method"),
        col("sd.date").alias("shipped_date"),
        col("dd.date").alias("delivered_date"),
        col("dd.year").alias("year"),
        col("dd.month").alias("month"),
        datediff(col("delivered_date"), col("shipped_date")).alias("delivery_days")
    ).filter(col("delivery_days") >=0)
#status_df.show()

Average_Delivery_Time_by_Shipping_Method = status_df.groupBy("year", "month",
"shipping_method") \
    .agg(round(avg("delivery_days"), 2).alias("Avg_delivery_time")) \
    .orderBy("year", "month", "shipping_method")

Average_Delivery_Time_by_Shipping_Method.show()
```

Sql Code

```
%sql
WITH shipped_df AS (
   SELECT
      T.customer_id,
      T.product_Id,
      T.shipping_method_Id,
      T.date_Id AS shipped_date_id
   FROM dbx_internalproc.sliver_layer.Fact_Transation T
   JOIN dbx_internalproc.sliver_layer.Dim_order_status O
      ON T.order_status_Id = O.order_status_Id
   WHERE O.order_status = "Shipped"
),
delivered_df AS (
   SELECT
      T.customer_id,
      T.product_Id,
      T.shipping_method_Id,
      T.date_Id AS delivered_date_id
   FROM dbx_internalproc.sliver_layer.Fact_Transation T
   JOIN dbx_internalproc.sliver_layer.Dim_order_status O
      ON T.order_status_Id = O.order_status_Id
   WHERE O.order_status = "Delivered"
```

```
),
status_df AS (
    SELECT
        s.customer_id,
        s.product_Id,
        s.shipping_method_Id,
        s.shipped_date_id,
        d.delivered_date_id
    FROM shipped_df s
    JOIN delivered_df d
        ON s.customer_id = d.customer_id
        AND s.product_Id = d.product_Id
        AND s.shipping_method_Id = d.shipping_method_Id
),
status_df_2 AS (
    SELECT
        sm.shipping_method,
        sd_date.date AS shipped_date,
        dd_date.date AS delivered_date,
        dd_date.year AS year,
        dd_date.month AS month,
        DATEDIFF(dd_date.date, sd_date.date) AS delivery_days
    FROM status_df sd
    JOIN dbx_internalproc.sliver_layer.Dim_Date sd_date
        ON sd.shipped_date_id = sd_date.date_id
    JOIN dbx_internalproc.sliver_layer.Dim_Date dd_date
        ON sd.delivered_date_id = dd_date.date_id
    JOIN dbx_internalproc.sliver_layer.Dim_shipping_method sm
        ON sd.shipping_method_Id = sm.shipping_method_Id
    WHERE DATEDIFF(dd_date.date, sd_date.date) >= 0
)
SELECT
    shipping_method,
    year,
    month,
    ROUND(AVG(delivery_days), 2) AS Avg_delivery_time
FROM status_df_2
GROUP BY shipping_method, year, month
ORDER BY year, month, shipping_method;
```

Output

	shipping_method	year	month	Avg_delivery_time
1	Express	2024	4	1.95
2	Same-day	2024	4	0.03
3	Standard	2024	4	1.73
4	Express	2024	5	2.03
5	Same-day	2024	5	0.07
6	Standard	2024	5	2.03
7	Express	2024	6	2.31
8	Same-day	2024	6	0.29
9	Standard	2024	6	2.24
10	Express	2024	7	2.53
11	Same-day	2024	7	0.96
12	Standard	2024	7	2.76
13	Express	2024	8	3.28

KPI 8: Repeat Product Purchases by Customers

Description

This KPI identifies **customers who repeatedly purchased the same product** over time. It helps in understanding **customer loyalty** and **product stickiness**, and can support upselling strategies or personalized marketing efforts.

Approach

1. Join the transaction fact table with the customer and product dimension tables to get **customer names** and **product details**.
2. Group by **customer name** and **product name/type** to count how many times each product was purchased.
3. Use HAVING to **filter for customers who purchased the same product more than once**.

Pyspark code

```
from pyspark.sql.functions import countDistinct

# Join to get required columns
joined_df = Fact_Transation.alias("T") \
    .join(Dim_customer.alias("c"), col("T.customer_id") == col("c.customer_id")) \
    .join(Dim_Product.alias("p"), col("T.product_Id") == col("p.product_Id")) \
    .select(
```

```
        col("c.name").alias("customer_name"),
        col("p.product_type"),
        col("p.product_Id")
    )

customers_bought_same_product = joined_df.groupBy("customer_name", "product_type") \
    .agg(countDistinct("product_Id").alias("total_purchases"))\
    .filter(countDistinct("product_Id") > 1)

customers_bought_same_product.show()
```

Sql code

```
%sql

SELECT
  c.name,
  p.product_type,
  COUNT(DISTINCT p.product_Id) AS distinct_products_bought
FROM dbx_internalproc.sliver_layer.Fact_Transation T
JOIN dbx_internalproc.sliver_layer.Dim_customer c ON T.customer_id = c.customer_id
JOIN dbx_internalproc.sliver_layer.Dim_Product p ON T.product_Id = p.product_Id
GROUP BY c.name, p.product_type HAVING COUNT(DISTINCT p.product_Id) > 1
```

Output

	name	product_type	distinct_products_bought
1	Stacy Mosley	Pendant lights	2
2	Matthew Mills	Chocolate cookies	2
3	Henry Stevenson	Sneakers	2
4	Zachary Diaz	Sneakers	2
5	Shannon Davis	Polo shirt	2
6	Beth Johnson	Pendant lights	2
7	Diana Bowers	Decorative pillows	2
8	Joseph Gutierrez	Polo shirt	2
9	Dr. Kenneth Trevino	Polo shirt	2
10	Lori Smith	Sneakers	2
11	Hannah Maxwell	Sneakers	2
12	Brittany Ward	Chocolate cookies	2

KPI 9: Revenue Risk Due to Returns

Description

This KPI calculates the **financial impact of returned or canceled orders** across countries and states. It helps identify locations where high return rates are contributing significantly to **revenue loss**, allowing the business to **mitigate risk** and optimize operations.

Approach

1. Join transaction data with location, product, and order status dimensions to get geographic and order details.
2. Calculate:
    a. **Total Revenue** for successful orders.
    b. **Revenue Lost** from returned or canceled orders.
3. Join both results and compute **revenue risk percentage** as:

revenue_risk_percentage=(revenue_lost/total_revenue)×100

Pyspark code:

```
joined_df = Fact_Transation.alias("T") \
    .join(Dim_Product.alias("p"), col("T.product_Id") == col("p.product_Id")) \
    .join(Dim_order_status.alias("o"), col("T.order_status_Id") ==
col("o.order_status_Id")) \
    .join(Dim_Location_Address.alias("A"), col("T.Location_Id") ==
col("A.Location_Id")) \
    .join(Dim_location_country.alias("C"), col("A.Country_Id") ==
col("C.Country_Id")) \
    .join(Dim_location_state.alias("S"), col("A.State_Id") == col("S.State_Id")) \
    .select(
        col("o.order_status").alias("order_status"),
        col("C.Country_name").alias("Country_name"),
        col("S.State_name").alias("State_name"),
        col("p.amount").alias("amount")
    ) \
    .filter((col("p.active_flag") == "Y"))
# joined_df.show()



total_revenue_df = joined_df.filter(col("order_status") == "Order") \
    .groupBy("Country_name", "State_name") \
    .agg(sum("amount").cast(LongType()).alias("total_revenue"))

returns_df = joined_df.filter((col("order_status") == "Returned")
|(col("order_status") == "Cancelled")) \
    .groupBy("Country_name", "State_name") \
    .agg(sum("amount").cast(LongType()).alias("revenue_lost"))
# returns_df.show()
revenue_risk_df = total_revenue_df.join(
```

```
    returns_df,
    on=["Country_name", "State_name"],
    how="left"
).fillna(0, subset=["revenue_lost"])

revenue_risk_df = revenue_risk_df.withColumn(
    "revenue_risk_percentage",
    round((col("revenue_lost") / col("total_revenue")) * 100, 2)
)

revenue_risk_df.select(
    "Country_name",
    "State_name",
    "total_revenue",
    "revenue_lost",
    "revenue_risk_percentage"
).show()
```

Sql code:

```
%sql
WITH joined_data AS (
   SELECT
      o.order_status,
      C.Country_name,
      S.State_name,
      p.amount
   FROM dbx_internalproc.sliver_layer.Fact_Transation T
   JOIN dbx_internalproc.sliver_layer.Dim_Product p ON T.product_Id = p.product_Id
   JOIN dbx_internalproc.sliver_layer.Dim_order_status o ON T.order_status_Id =
o.order_status_Id
   JOIN dbx_internalproc.sliver_layer.Dim_Location_Address A ON T.Location_Id =
A.Location_Id
   JOIN dbx_internalproc.sliver_layer.Dim_location_country C ON A.Country_Id =
C.Country_Id
   JOIN dbx_internalproc.sliver_layer.Dim_location_state S ON A.State_Id = S.State_Id
   WHERE p.active_flag = 'Y'
),

total_revenue AS (
   SELECT
      Country_name,
      State_name,
```

```
        CAST(SUM(amount) AS BIGINT) AS total_revenue
    FROM joined_data
    WHERE order_status = 'Order'
    GROUP BY Country_name, State_name
),

returns AS (
    SELECT
        Country_name,
        State_name,
        CAST(SUM(amount) AS BIGINT) AS revenue_lost
    FROM joined_data
    WHERE order_status IN ('Returned', 'Cancelled')
    GROUP BY Country_name, State_name
)

SELECT
    tr.Country_name,
    tr.State_name,
    tr.total_revenue,
    r.revenue_lost,
    ROUND(r.revenue_lost* 100.0 / tr.total_revenue, 2) AS revenue_risk_percentage
FROM total_revenue tr
LEFT JOIN returns r
ON tr.Country_name = r.Country_name AND tr.State_name = r.State_name;
```

Output

	Country_name	State_name	total_revenue	revenue_lost	revenue_risk_percentage
1	UK	Northern Ireland	172786881	106089372	61.40
2	Germany	Bavaria	171528694	105294694	61.39
3	Germany	Hesse	176151791	107628737	61.10
4	UK	Scotland	173947057	105188596	60.47
5	Germany	Berlin	168148684	103170478	61.36
6	UK	England	175866237	105258705	59.85
7	UK	Wales	167760479	102017384	60.81
8	Germany	Hamburg	173583273	106286181	61.23

KPI 10: Most Product Purchase Frequency by Timeslot

Description

This KPI analyzes customer **purchase behavior across different times of day**. It helps identify peak purchase windows to optimize marketing, notifications, inventory restocking, and server loads.

Timeslot Breakdown

- **12 AM - 6 AM**
- **6 AM - 12 PM**
- **12 PM - 6 PM**
- **6 PM - 12 AM**

Approach

1. Extract **hour** from the time field in Dim_Date.
2. Group orders into **4 timeslots** based on the extracted hour.
3. Count the number of orders per timeslot.

Pyspark code:

```
fact_df = Fact_Transation.alias("T") \
    .join(Dim_Date.alias("D"), col("T.date_Id") == col("D.date_id")) \
    .join(Dim_order_status.alias("O"), col("T.order_status_Id") ==
col("O.order_status_Id")) \
    .filter(col("O.order_status") == "Order") \
    .select(
        col("T.product_Id"),
        col("time")
    )



fact_df = fact_df.withColumn("hour", substring_index(col("time"), ":",
1).cast("int"))

fact_df = fact_df.withColumn(
    "timeslot",
    when((col("hour") >= 0) & (col("hour") < 6), "12 AM - 6 AM")
    .when((col("hour") >= 6) & (col("hour") < 12), "6 AM - 12 PM")
    .when((col("hour") >= 12) & (col("hour") < 18), "12 PM - 6 PM")
    .otherwise("6 PM - 12 AM")
)

timeslot_kpi = fact_df.groupBy("timeslot") \
    .agg(count("*").alias("frequency")) \
    .orderBy("frequency", ascending=False)
```

```
timeslot_kpi.show()
```

Sql Code

```
%sql
WITH ordered_fact AS (
   SELECT
      T.product_Id,
      D.time,
      CAST(SPLIT(D.time, ':')[0] AS INT) AS hour
   FROM dbx_internalproc.sliver_layer.Fact_Transation T
   JOIN dbx_internalproc.sliver_layer.Dim_Date D
      ON T.date_Id = D.date_id
   JOIN dbx_internalproc.sliver_layer.Dim_order_status O
      ON T.order_status_Id = O.order_status_Id
   WHERE O.order_status = 'Order'
),
timeslot_fact AS (
   SELECT
      product_Id,
      hour,
      CASE
         WHEN hour >= 0 AND hour < 6 THEN '12 AM - 6 AM'
         WHEN hour >= 6 AND hour < 12 THEN '6 AM - 12 PM'
         WHEN hour >= 12 AND hour < 18 THEN '12 PM - 6 PM'
         ELSE '6 PM - 12 AM'
      END AS timeslot
   FROM ordered_fact
)
SELECT
   timeslot,
   COUNT(*) AS frequency
FROM timeslot_fact
GROUP BY timeslot
ORDER BY frequency DESC;
```

Output

Table

	timeslot	frequency
1	6 AM - 12 PM	32722
2	12 PM - 6 PM	32710
3	12 AM - 6 AM	32673
4	6 PM - 12 AM	32290

KPI 11: Product Type Revenue by Gender & Age Group

Description

This KPI provides a **demographic breakdown** of revenue generated by each **product type**, segmented by:

- **Gender**
- **Age Group**

This helps identify which segments contribute most to revenue and how product appeal varies across different audiences.

Approach

1. Join transaction data with customer, product, and feedback details.
2. Calculate age from birth date and classify into age groups.
3. Group by product_type, gender, and age_group.
4. Calculate:
    a. Total revenue: sum of product amounts.
    b. Average rating: from customer feedback.

Pyspark code:

```
current_year = datetime.now().year
joined_df = Fact_Transation.alias("T") \
    .join(Dim_customer.alias("c"), col("T.customer_id") == col("c.customer_id")) \
    .join(Dim_Product.alias("p"), col("T.product_Id") == col("p.product_Id")) \
    .join(Dim_Feedback.alias("f"), col("T.Feedback_id") == col("f.Feedback_id")) \
    .select(
        col("c.gender").alias("gender"),
        col("c.birth_date").alias("birth_date"),
        col("p.product_type").alias("product_type"),
        col("f.rating"),
        col("p.amount").alias("amount")
```

```
    )\
    .filter(col("p.active_flag") == "Y")
# joined_df.show()



# Convert birth_date string to date type first
joined_df = joined_df.withColumn("birth_date", to_date(col("birth_date"),
"dd/MM/yyyy"))
# joined_df.show()

# calculate age
joined_df = joined_df.withColumn("age", (current_year -
year(col("birth_date")))).cast(IntegerType()))
# joined_df.show()
joined_df = joined_df.withColumn(
    "age_group",
    when((col("age") >= 18) & (col("age") <= 25), "18-25")
    .when((col("age") >= 26) & (col("age") <= 35), "26-35")
    .when((col("age") >= 36) & (col("age") <= 45), "36-45")
    .when((col("age") >= 46) & (col("age") <= 60), "46-60")
    .when((col("age") >= 61) & (col("age") <= 75), "61-75")
    .when((col("age") >= 76) & (col("age") <= 90), "76-90")
    .otherwise("90+")
)
product_type_by_Revenue_and_gender = joined_df.groupBy("product_type", "gender",
"age_group") \
    .agg(
        sum("amount").cast(LongType()).alias("Total_Revenue"),
        round(avg("rating"),2).alias("Avg_Rating")
    ).orderBy("product_type", "gender", "age_group")
product_type_by_Revenue_and_gender .show()
```

Sql Code

```
%sql
WITH joined_df AS (
  SELECT
    c.gender,
    TO_DATE(c.birth_date, 'dd/MM/yyyy') AS birth_date,
    CAST(YEAR(CURRENT_DATE) - YEAR(TO_DATE(c.birth_date, 'dd/MM/yyyy'))
AS INT) AS age,
    p.product_type,
    f.rating,
    p.amount
  FROM dbx_internalproc.sliver_layer.fact_transation T
  JOIN dbx_internalproc.sliver_layer.Dim_customer c ON T.customer_id = c.customer_id
```

```
   JOIN dbx_internalproc.sliver_layer.Dim_Product p ON T.product_Id = p.product_Id
   JOIN dbx_internalproc.sliver_layer.Dim_Feedback f ON T.Feedback_id =
f.Feedback_id
   WHERE p.active_flag = 'Y'
),
grouped_df AS (
   SELECT *,
      CASE
         WHEN age BETWEEN 18 AND 25 THEN '18-25'
         WHEN age BETWEEN 26 AND 35 THEN '26-35'
         WHEN age BETWEEN 36 AND 45 THEN '36-45'
         WHEN age BETWEEN 46 AND 60 THEN '46-60'
         WHEN age BETWEEN 61 AND 75 THEN '61-75'
         WHEN age BETWEEN 76 AND 90 THEN '76-90'
         ELSE '90+'
      END AS age_group
   FROM joined_df
)
SELECT
   product_type,
   gender,
   age_group,
   CAST(SUM(amount) AS BIGINT) AS Total_Revenue,
   ROUND(AVG(rating), 2) AS Avg_Rating
FROM grouped_df
GROUP BY product_type, gender, age_group
ORDER BY product_type, gender, age_group;
```

Output

	product_type	gender	age_group	Total_Revenue	Avg_Rating
1	Amazon Fire Tablet	Female	18-25	34714036	2.96
2	Amazon Fire Tablet	Female	26-35	45755780	3.07
3	Amazon Fire Tablet	Female	36-45	41335532	2.95
4	Amazon Fire Tablet	Female	90+	13376132	3.01
5	Amazon Fire Tablet	Male	18-25	35699272	3.01
6	Amazon Fire Tablet	Male	26-35	43057476	2.97
7	Amazon Fire Tablet	Male	36-45	42791196	2.91
8	Amazon Fire Tablet	Male	90+	11920468	2.95
9	Amazon Fire Tablet	male	18-25	71008	4.27
10	Amazon Fire Tablet	male	26-35	221900	3.21
11	Amazon Fire Tablet	male	36-45	195272	2.8
12	Amazon Fire Tablet	male	90+	26628	2.4
13	Chocolate cookies	Female	18-25	45379264	2.99

KPI 12: Repeat Purchase Score by Product

Description

This KPI identifies products that have a **high repeat purchase rate**, which helps in understanding:

- **Customer loyalty** to specific products.
- **Top products** likely to be purchased again.
- Marketing opportunities for repeat-driven categories.

Approach

1. **Step 1:** Count how many times each customer bought each product.
2. **Step 2:** For products with repeat buyers, calculate:
    a. How many extra purchases occurred beyond the first.
    b. How many customers repeated.
3. **Step 3:** Get the total number of purchases per product.
4. **Step 4:** Calculate percentage of repeat purchases and list top repeat customers.

Pyspark code:

```
from pyspark.sql.functions import count, sum, when, col, round, countDistinct


#Get customer-level purchase counts per product
customer_product_freq = Fact_Transation.alias("T") \
    .join(Dim_Product.alias("p"), col("T.product_Id") == col("p.product_Id")) \
    .join(Dim_customer.alias("c"), col("T.customer_id") == col("c.customer_id")) \
    .groupBy(
```

```
        col("p.product_id"),
        col("p.product_type"),
        col("p.product_category"),
        col("c.customer_id")
    ).agg(
        count("*").alias("purchase_count")
    )

#Aggregate repeat and total counts in one step
aggregated_product_stats = customer_product_freq.groupBy(
    "product_id", "product_type", "product_category"
).agg(
    sum(when(col("purchase_count") > 1, col("purchase_count") -
1).otherwise(0)).alias("Repeat_Purchase_Count"),
    count("*").alias("Unique_Customers"),
    countDistinct(when(col("purchase_count") > 1,
col("customer_id"))).alias("Top_Repeat_Customers")
)

#Get total transactions per product
total_purchases = Fact_Transation.alias("T") \
    .join(Dim_Product.alias("p"), col("T.product_Id") == col("p.product_Id")) \
    .groupBy("p.product_id") \
    .agg(count("*").alias("Total_Purchase_Count"))

#Join and calculate final KPI
final_df = aggregated_product_stats.alias("a") \
    .join(total_purchases.alias("t"), col("a.product_id") == col("t.product_id"),
"left") \
    .select(
        col("a.product_type"),
        col("a.product_category"),
        col("a.Repeat_Purchase_Count"),
        col("t.Total_Purchase_Count"),
        round(
            (col("a.Repeat_Purchase_Count") * 100.0 /
when(col("t.Total_Purchase_Count") == 0,
1).otherwise(col("t.Total_Purchase_Count"))),
            2
        ).alias("Repeat_Purchase_Percentage"),
        col("a.Top_Repeat_Customers")
    ).orderBy(col("Repeat_Purchase_Percentage").desc())
final_df.show()
```

Sql code

```
%sql
WITH customer_product_freq AS (
```

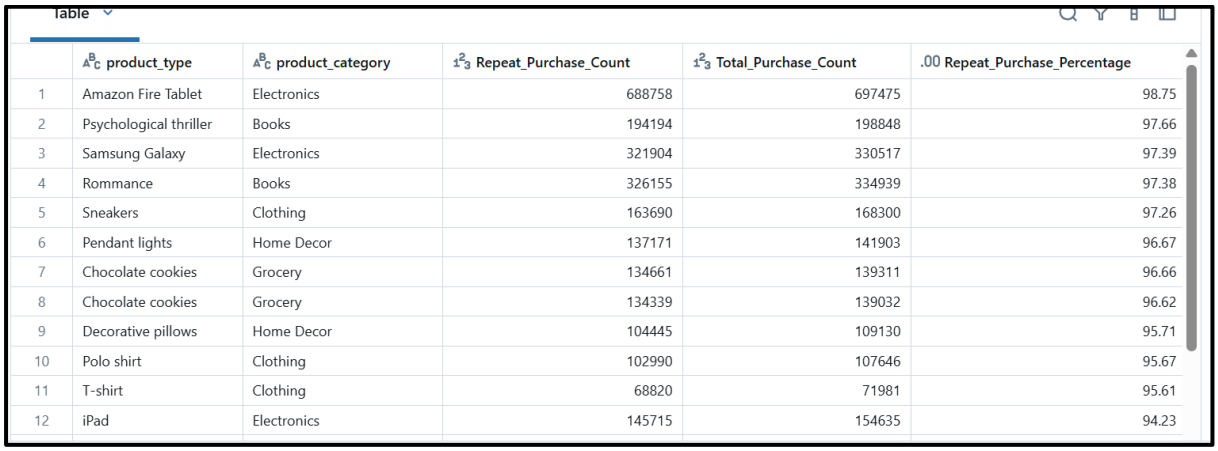
```
    SELECT
        p.product_id,
        p.product_type,
        p.product_category,
        c.customer_id,
        COUNT(*) AS purchase_count
    FROM dbx_internalproc.sliver_layer.Fact_Transation T
    JOIN dbx_internalproc.sliver_layer.Dim_Product p ON T.product_Id =
p.product_Id
    JOIN dbx_internalproc.sliver_layer.dim_customer c ON T.customer_id =
c.customer_id
    GROUP BY p.product_id, p.product_type, p.product_category, c.customer_id
),
aggregated_product_stats AS (
    SELECT
        product_id,
        product_type,
        product_category,
        SUM(CASE WHEN purchase_count > 1 THEN purchase_count - 1 ELSE 0
END) AS Repeat_Purchase_Count,
        COUNT(*) AS Unique_Customers,
        COUNT(DISTINCT CASE WHEN purchase_count > 1 THEN customer_id
END) AS Top_Repeat_Customers
    FROM customer_product_freq
    GROUP BY product_id, product_type, product_category
),

-- Get total transactions per product
total_purchases AS (
    SELECT
        p.product_id,
        COUNT(*) AS Total_Purchase_Count
    FROM dbx_internalproc.sliver_layer.Fact_Transation T
    JOIN dbx_internalproc.sliver_layer.Dim_Product p ON T.product_Id =
p.product_Id
    GROUP BY p.product_id
)

SELECT
    a.product_type,
    a.product_category,
```

```
    a.Repeat_Purchase_Count,
    t.Total_Purchase_Count AS Total_Purchase_Count,
    ROUND(a.Repeat_Purchase_Count* 100.0 / t.Total_Purchase_Count,2) AS
Repeat_Purchase_Percentage,
    a.Top_Repeat_Customers AS Top_Repeat_Customers
FROM aggregated_product_stats a
LEFT JOIN total_purchases t
    ON a.product_id = t.product_id
ORDER BY Repeat_Purchase_Percentage DESC;
```

Output



Table

	product_type	product_category	Repeat_Purchase_Count	Total_Purchase_Count	Repeat_Purchase_Percentage
1	Amazon Fire Tablet	Electronics	688758	697475	98.75
2	Psychological thriller	Books	194194	198848	97.66
3	Samsung Galaxy	Electronics	321904	330517	97.39
4	Rommance	Books	326155	334939	97.38
5	Sneakers	Clothing	163690	168300	97.26
6	Pendant lights	Home Decor	137171	141903	96.67
7	Chocolate cookies	Grocery	134661	139311	96.66
8	Chocolate cookies	Grocery	134339	139032	96.62
9	Decorative pillows	Home Decor	104445	109130	95.71
10	Polo shirt	Clothing	102990	107646	95.67
11	T-shirt	Clothing	68820	71981	95.61
12	iPad	Electronics	145715	154635	94.23

KPI 13: **Brand Loyalty Score**

Description

This KPI evaluates **customer loyalty toward specific brands** by tracking:

- **Frequency** of repeat purchases by customer per brand.
- **Last purchase date** to understand recent engagement.
- **Average rating** given by customers for that brand.

It helps **identify brand loyalists**—customers who consistently purchase from the same brand.

Approach

1. Join the transaction, product, feedback, and date tables.
2. Group by customer and brand.
3. Use COUNT(*) to calculate repeat purchases per brand.
4. Track the **most recent purchase date** for freshness.

5. Average customer rating to assess brand sentiment.

Pysaprk code

```
from pyspark.sql.functions import max
# Join fact table with product and feedback
joined_df = Fact_Transation.alias("T") \
    .join(Dim_Product.alias("p"), col("T.product_Id") == col("p.product_Id")) \
    .join(Dim_Feedback.alias("f"), col("T.Feedback_id") == col("f.Feedback_id")) \
    .join(Dim_Date.alias("D"), col("T.date_Id") == col("D.date_id")) \
    .select(
        col("T.customer_id"),
        col("p.product_brand").alias("brand"),
        col("D.date").alias("purchase_date"),
        col("f.rating")
    ).filter(col("f.rating").isNotNull())
# joined_df.show()
#Group by customer and brand
brand_loyalty_df = joined_df.groupBy("customer_id", "brand") \
    .agg(
        count("*").alias("Repeated_Purchase_Count"),
        max("purchase_date").alias("Last_Purchase_Date"),
        round(avg("rating"), 2).alias("Avg_Rating_Give")
    ) \
    .orderBy("brand", "Last_Purchase_Date", "Avg_Rating_Give")

brand_loyalty_df.show()
```

Sql Code

```
%sql
select
T.customer_id,
p.product_brand,
max(d.date) as Last_Purchase_Date,
round(avg(f.rating), 2) as Avg_Rating_Give
from dbx_internalproc.sliver_layer.Fact_Transation T
   JOIN dbx_internalproc.sliver_layer.Dim_Product p ON T.product_Id = p.product_Id
   JOIN dbx_internalproc.sliver_layer.Dim_Feedback f ON T.Feedback_id =
f.Feedback_id
   JOIN dbx_internalproc.sliver_layer.dim_date d ON T.date_Id = d.date_Id
   where f.rating IS NOT NULL
group by customer_id,product_brand
order by product_brand,Last_Purchase_Date,Avg_Rating_Give;
```

Output

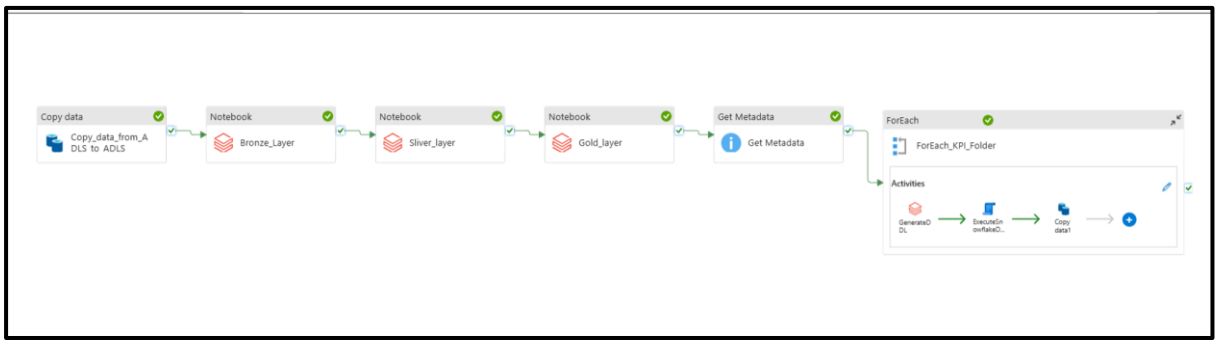
Table

	customer_id	product_brand	Last_Purchase_Date	Avg_Rating_Give
1	666de367439d4875a2db54	Adidas	2024-04-20	2
2	19b420294582440ca073b6	Adidas	2024-04-20	5
3	1f6ba7f2ae0b4569a14e67	Adidas	2024-04-21	1.2
4	46296d9f35df40348d4406	Adidas	2024-04-21	1.5
5	0dfe9ff954a847e8909f33	Adidas	2024-04-21	2.5
6	1f4f8eb62cb04822b2f3ab	Adidas	2024-04-21	2.6
7	a441958a3067451ea2a234	Adidas	2024-04-21	2.7
8	dc163a6b41a645faab93d3	Adidas	2024-04-21	3
9	194a9ff3017740089fe8fc	Adidas	2024-04-21	4
10	c3b46356a7f24ba9969dc8	Adidas	2024-04-22	1.5
11	8508c00f16084481ba0b58	Adidas	2024-04-22	3.5
12	995073eac60f45ac967b59	Adidas	2024-04-22	3.5

All KPI outputs (from KPI 1 to KPI 13) have been stored as **Delta tables** in an **external location on Azure Data Lake Storage Gen2 (ADLS Gen2)**. This setup ensures that the data is **secure**, **easily accessible**, and ready for **further analysis or reporting**.

Pipeline Approach: 1

Without snowflake connector or without JDBC (11-12 min)



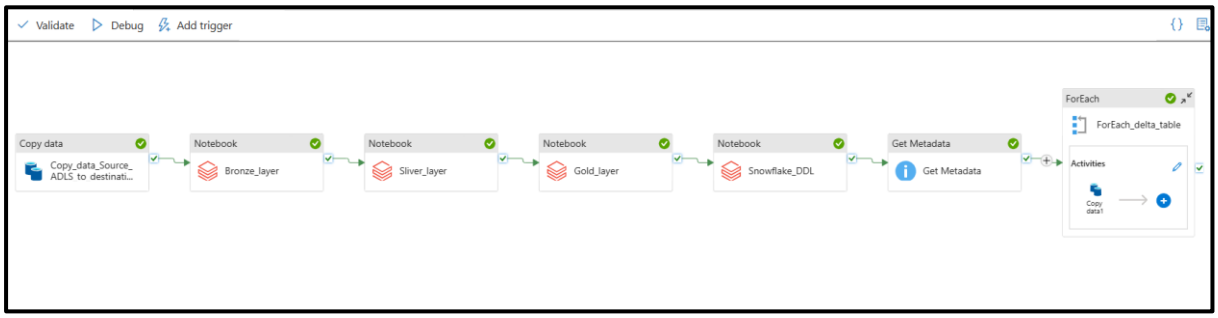
The complete data processing pipeline is designed to support **incremental data loads** and completes execution within **10–11 minutes**. Here's a breakdown of the workflow:

1. **Copy Data**: Initial raw data is copied from **ADLS Gen2 source** to the processing area in **ADLS Gen2**.
2. **Bronze Layer Notebook**: Performs minimal transformations such as data cleaning and type casting.
3. **Silver Layer Notebook**: Applies business logic and joins to enrich the data.
4. **Gold Layer Notebook**: Computes all 13 KPIs and stores the results as **Delta tables** in external folders on ADLS Gen2.
5. **Get Metadata Activity**: Dynamically retrieves schema details (DDL) of each Delta table in the gold layer.

6. **ForEach Activity**:
    a. **Generate DDL**: Converts the retrieved Delta schema into Snowflake-compatible DDL.
    b. **Execute Script**: Runs the DDL on the connected **Snowflake trial account** to create the corresponding target tables.
    c. **Copy Data**: Loads the KPI data from ADLS Gen2 (Delta format) into the mapped Snowflake tables.

This entire process is **automated**, **scalable**, and ensures that the latest KPI insights are consistently reflected in Snowflake for further reporting or analysis.

Pipeline Approach: 2



This pipeline automates the end-to-end flow from raw data ingestion to Snowflake loading and is optimized for **incremental data processing**. The pipeline takes around 9–**10 minutes** to complete.

1. **Copy Data**: Ingests raw data from **ADLS Gen2 source** to the staging location in ADLS.
2. **Bronze Layer Notebook**: Applies basic transformations such as type casting and data cleansing.
3. **Silver Layer Notebook**: Enhances data with business logic, filtering, and joins.
4. **Gold Layer Notebook**: Computes and stores the final KPIs as **Delta tables** in the Gold layer.
5. **Snowflake_DDL Notebook**:
    a. Scans the Delta table folders in the Gold layer.
    b. Uses the **Snowflake connector** to check if the corresponding Snowflake table already exists.
    c. If the table does **not exist**, generates and executes the Snowflake-compatible DDL for table creation.
    d. If the table **already exists**, skips DDL creation to avoid duplication.
6. **Get Metadata Activity**: Extracts schema information from the Delta tables.

7. **ForEach Activity**:
    a. **Copy Data**: Loads data from the Gold layer (Delta format in ADLS Gen2) into the matched Snowflake tables.

GitHub Link:

https://github.com/TulsiThakkar-kenexai/DBX_inetrnalProject