✅ Phase 1: Data Audit & Cleaning

We are working with 3 datasets:

- `Customer.csv`
- `Order.csv`
- `Shipping.json`

---

1. Data Audit Objectives

For each file, we audited data on the following aspects:

Check	Description
Schema Validation	Are column names and types as expected?
Null Values	Any missing values that impact analysis?
Duplicate Records	Any exact duplicates or ID conflicts?
Unique Key Integrity	Does the primary key truly identify each row?
Cardinality	Are relationships plausible for joins (e.g. many-to-many)?
Domain Constraints	Do values fall in expected ranges (e.g. age not negative)?
Referential Integrity	Do foreign keys match across datasets?

2. Dataset Summary

Customer.csv

Column	Description	Type	Sample
`Customer_ID`	Unique customer identifier	int	101
`First, Last`	Names (not used in analysis)	string	"John", "Doe"
`Age`	Integer, range 18–60s	int	29
`Country`	Categorical	string	"USA", "Germany", "UK"

Python Check Used:

```
customer_df.info()
customer_df.isnull().sum()
customer_df.duplicated().sum()
customer_df['Age'].describe()
customer_df['Country'].value_counts()
```

✅ No nulls
✅ No duplicates
✅ Age range valid (18–66)
✅ 3 countries only → useful for group reporting
✅ `Customer_ID` is a reliable primary key

Order.csv

Column	Description	Type	Sample
`Order_ID`	Transaction ID	int	2001
`Item`	Product name	string	"Laptop"
`Amount`	Total spend	int	1000
`Customer_ID`	FK to Customer	int	101

Python Check Used:

```
order_df.info()
order_df.isnull().sum()
order_df.duplicated().sum()
order_df['Amount'].describe()
order_df['Item'].value_counts()
order_df['Customer_ID'].nunique()
```

✅ No nulls
✅ No duplicates
✅ Amount range valid (assumed in hundreds)
✅ `Item` has 8 distinct values
✅ `Customer_ID` exists in Customer data (we'll join to validate later)

Shipping.json

Column	Description	Type	Sample
`Shipping_ID`	Unique shipment	int	301
`Status`	Delivery status	string	"Pending", "Delivered"
`Customer_ID`	FK to Customer	int	101

Python Check Used:

```
shipping_df.info()
shipping_df.isnull().sum()
shipping_df.duplicated().sum()
shipping_df['Status'].value_counts()
shipping_df['Customer_ID'].nunique()
```

✅ No nulls
✅ No duplicates
✅ `Status` = "Pending" or "Delivered"
✅ Multiple rows per customer → many-to-one relationship
✅ All `Customer_ID`s match back to customer table

3. Referential Integrity (Joins)

Check if foreign keys match:

```
# Customers in orders but not in customers
set(order_df['Customer_ID']) - set(customer_df['Customer_ID'])

# Customers in shipping but not in customers
set(shipping_df['Customer_ID']) - set(customer_df['Customer_ID'])
```

✅ **All IDs match correctly** across all datasets
✅ Ready to join tables on `Customer_ID`

Summary of Data Audit

Dataset	Nulls	Duplicates	Key Quality	FK Valid	Notes
Customer	❌ None	❌ None	✅ Clean Customer_ID	N/A	Valid
Order	❌ None	❌ None	✅ Clean Order_ID	✅	Products look fine
Shipping	❌ None	❌ None	✅ Clean Shipping_ID	✅	Delivery statuses consistent

---

✅ Phase 2: Domain Model Design

Goal of This Phase

To define a **clean, scalable data model** that supports:

1. Joinable relationships across Customer, Order, and Shipping
2. Aggregation, filtering, and grouping for all 5 reporting requirements
3. Easy implementation for data engineering (in Phase 3)

---

1. Identify Core Entities (Tables)

Entity	Description	Primary Key
Customer	Demographic info: name, age, country	`Customer_ID`
Order	Transaction data: item, amount	`Order_ID`
Shipping	Delivery status	`Shipping_ID (1 row per shipment)`

2. Define Relationships

✅ One-to-Many:

- **Customer** ↔ **Order**: One customer can place many orders
- **Customer** ↔ **Shipping**: One customer can have multiple shipments

  Note: Shipping and Order are not directly linked in the raw data, so we assume shipping is tracked at the **customer level**.

3. Normalize Fields for Analysis

We derive the following **additional fields**:

Table	New Field	Description
Customer	`Age_Group`	Bucket: <30, >=30
Order	`Transaction_Count`	Total orders per customer
Order	`Total_Amount_Spent`	Aggregated SUM(Amount) per customer
Order	`Most_Purchased_Product`	Derived using COUNT(*) GROUP BY Item
Shipping	`Pending_Amount_By_Country`	Join + group by country where status = Pending

4. Detailed Logical Schema

This schema represents how the three datasets (`Customer`, `Order`, `Shipping`) interact in a **normalized star-schema layout** designed to support all reporting use cases.

1. CUSTOMER Table (Dimension)

Column	Type	Description
`Customer_ID`	INT	Primary Key
`First`	STRING	First Name
`Last`	STRING	Last Name
`Age`	INT	Age of customer
`Country`	STRING	Country of origin
`Age_Group*`	STRING	Derived: "Under 30" or "30 and above"

Primary Key: `Customer_ID`
Used in: Joins with `Order`, `Shipping`, Age-based segmentation

2. ORDER Table (Fact)

Column	Type	Description
`Order_ID`	INT	Primary Key
`Customer_ID`	INT	Foreign Key to Customer.Customer_ID
`Item`	STRING	Product purchased
`Amount`	INT	Total order value (in currency unit)

Primary Key: `Order_ID`
Foreign Key: `Customer_ID` → `Customer.Customer_ID`
Used in: Spend analysis, transactions, most purchased product queries

3. SHIPPING Table (Status Snapshot)

Column	Type	Description
`Shipping_ID`	INT	Primary Key
`Customer_ID`	INT	Foreign Key to Customer.Customer_ID
`Status`	STRING	"Pending" or "Delivered"

Primary Key: `Shipping_ID`
Foreign Key: `Customer_ID` → `Customer.Customer_ID`
Used in: Delivery status tracking, pending order spend

Relationships Summary:

From Table	Column	→ To Table	Join Key	Cardinality
ORDER	Customer_ID	CUSTOMER	Customer_ID	Many-to-One
SHIPPING	Customer_ID	CUSTOMER	Customer_ID	Many-to-One

Analytical Use Cases Enabled:

Use Case	Tables Involved	Logic Summary
Spend by pending delivery	Customer + Shipping + Order	Filter Shipping on "Pending", aggregate Order by Country
Customer-level metrics	Customer + Order	Group by Customer_ID and Item
Top product per country	Customer + Order	Rank Items by count per Country
Top product by age	Customer + Order	Add Age_Group, then group & rank
Lowest performing country	Customer + Order	Group by Country, use MIN logic

5. Handling Slowly Changing Dimensions (SCD)

When data warehousing, **dimension values change over time**. In this context, for example:

- A customer may move to a new **country**
- Their **age group** may shift (if updated based on actual birthdate)
- Even names might be corrected

To preserve **historical accuracy** in reporting (e.g., "how much did we sell in Germany in 2022?"), we need to track these changes using **Slowly Changing Dimensions (SCD)**.

Where SCD Applies in This Model:

Table	Columns Impacted	SCD Needed?	Why?
CUSTOMER	country, age_group, optionally first, last	✅ Yes	Business entities that can change over time and affect analytical outputs
PRODUCT	category	🚫 No	Less likely to change and generally maintained separately
SHIPPING	status	❌ No (event-based, not dimensional)	Status is transactional, not slowly evolving

SCD Type Chosen: SCD Type 2

We use **SCD Type 2** because we need to **preserve history** i.e., keep old versions of the record and know when each version was valid.

How to Handle It:

Column Name	Purpose
customer_sk	Surrogate key (unique per version)
customer_id	Business key
country	Slowly changing attribute
age_group	Derived (from age if stored historically)
valid_from	When this version became active
valid_to	When it was replaced (NULL if current)
is_current	Y/N flag for active version

Example:

customer_sk	customer_id	country	valid_from	valid_to	is_current
9001	101	USA	2021-01-01	2023-03-15	N
9002	101	Germany	2023-03-16	NULL	Y

QA/Testing Strategy for SCD:

- Check row count for each customer: **1 active record, 0+ historical**
- Validate `valid_from` and `valid_to` form a continuous timeline (no gaps or overlaps)
- Confirm `is_current = 'Y'` appears only once per `customer_id`

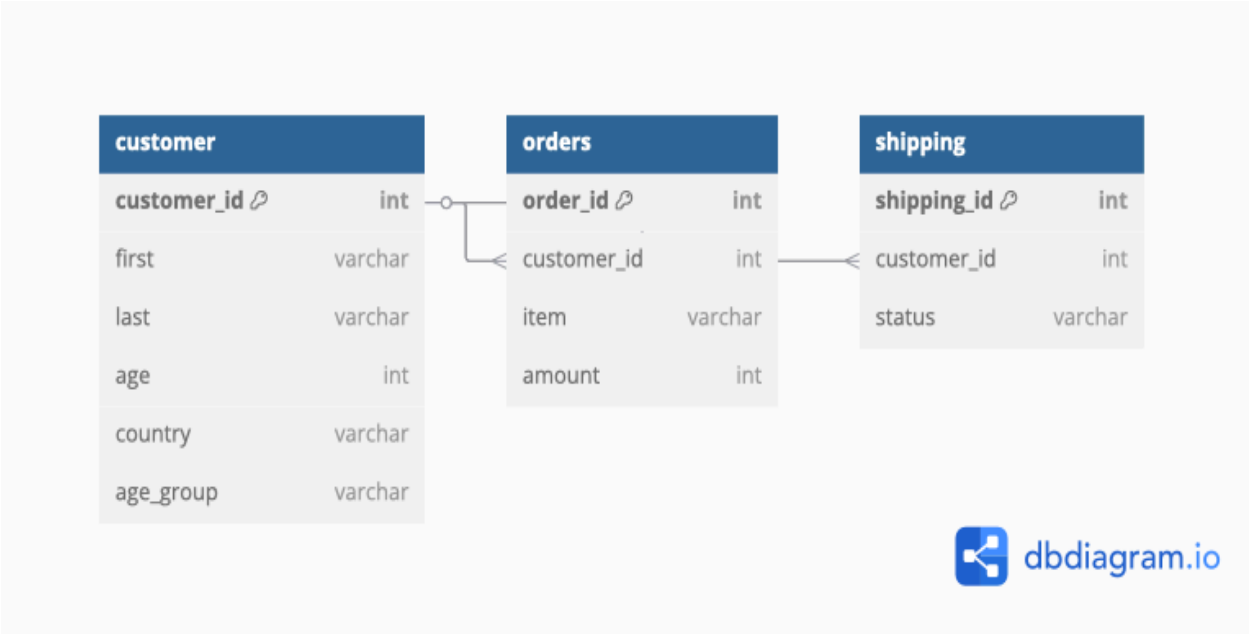
Summary Decision:

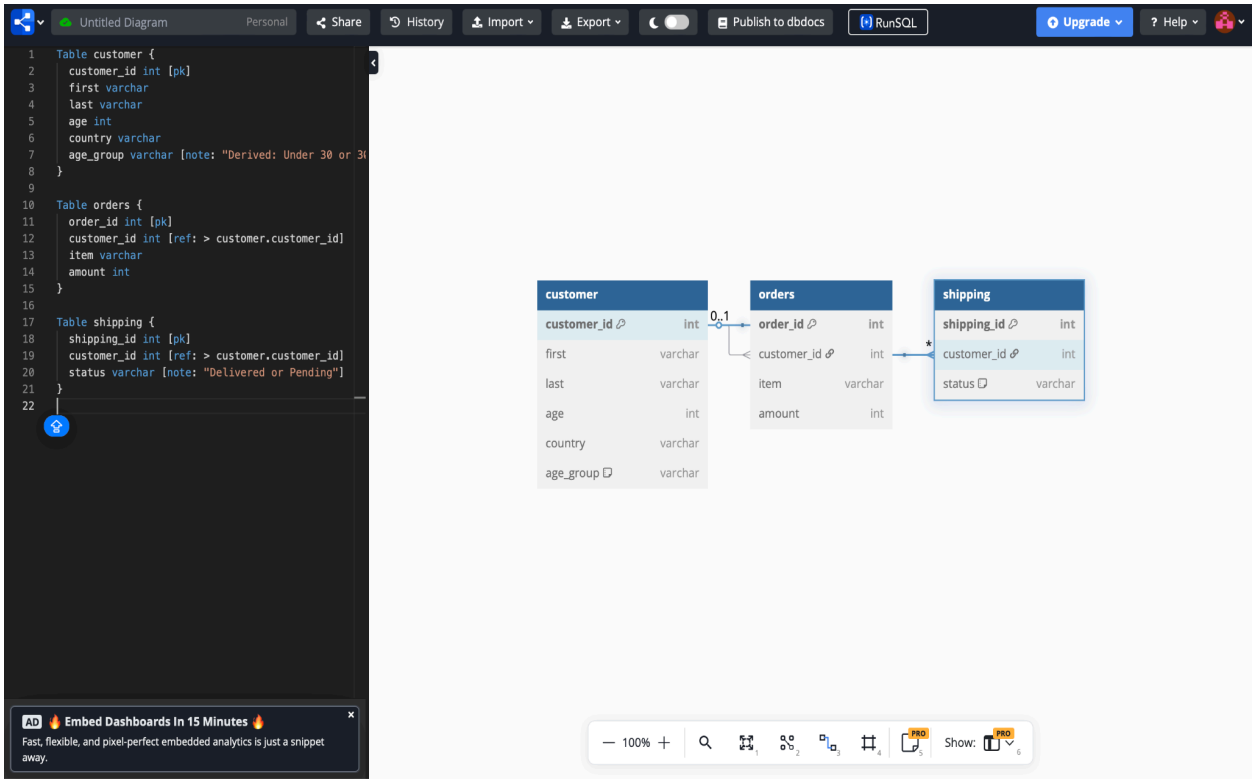
Attribute	SCD Type	Justification
`country`	Type 2	Preserves historical reporting accuracy
`age_group`	Type 2 (derived)	Optional if age is updated and reports are time-aware
`name`	Type 1 or 2	Only if business logic requires history (e.g., legal, audit)

6. Mapping Model to Business Requirements

Business Requirement	Tables Used	Join Logic	Aggregation / Logic	Final Output
Total amount spent & country-wise breakdown for PENDING deliveries	Shipping, Customer, Order	Shipping.Customer_ID = Customer.Customer_ID Order.Customer_ID = Customer.Customer_ID	SUM(Order.Amount)grouped by Customer.Country	Country, Total_Amount_Pending
Customer-level metrics: transactions, quantity, spend, product details	Order, Customer	Order.Customer_ID = Customer.Customer_ID	- COUNT(Order_ID)as total transactions - SUM(Amount) as total spend - GROUP BY Customer_ID, Item to get product-wise stats	Customer_ID, Name, Country, Item, Transaction_Count, Total_Quantity, Total_Amount_Spent
Maximum product purchased per country	Order, Customer	Order.Customer_ID = Customer.Customer_ID	- GROUP BY Country, Item - SUM(Amount) or COUNT(Order_ID) - Use RANK() or ROW_NUMBER() to select top item per country	Country, Item, Total_Amount_Spent or Purchase_Count
Most purchased product by age category (<30, ≥30)	Order, Customer	Order.Customer_ID = Customer.Customer_ID	- Create derived column Age_Group in Customer - GROUP BY Age_Group, Item - COUNT(Order_ID) → find max per group	Age_Group, Top_Product, Purchase_Count
Country with minimum transactions & lowest total sales	Order, Customer	Order.Customer_ID = Customer.Customer_ID	- GROUP BY Country - COUNT(Order_ID) and SUM(Amount) - Use ORDER BY ASC LIMIT 1	Country, Total_Transactions, Total_Sales

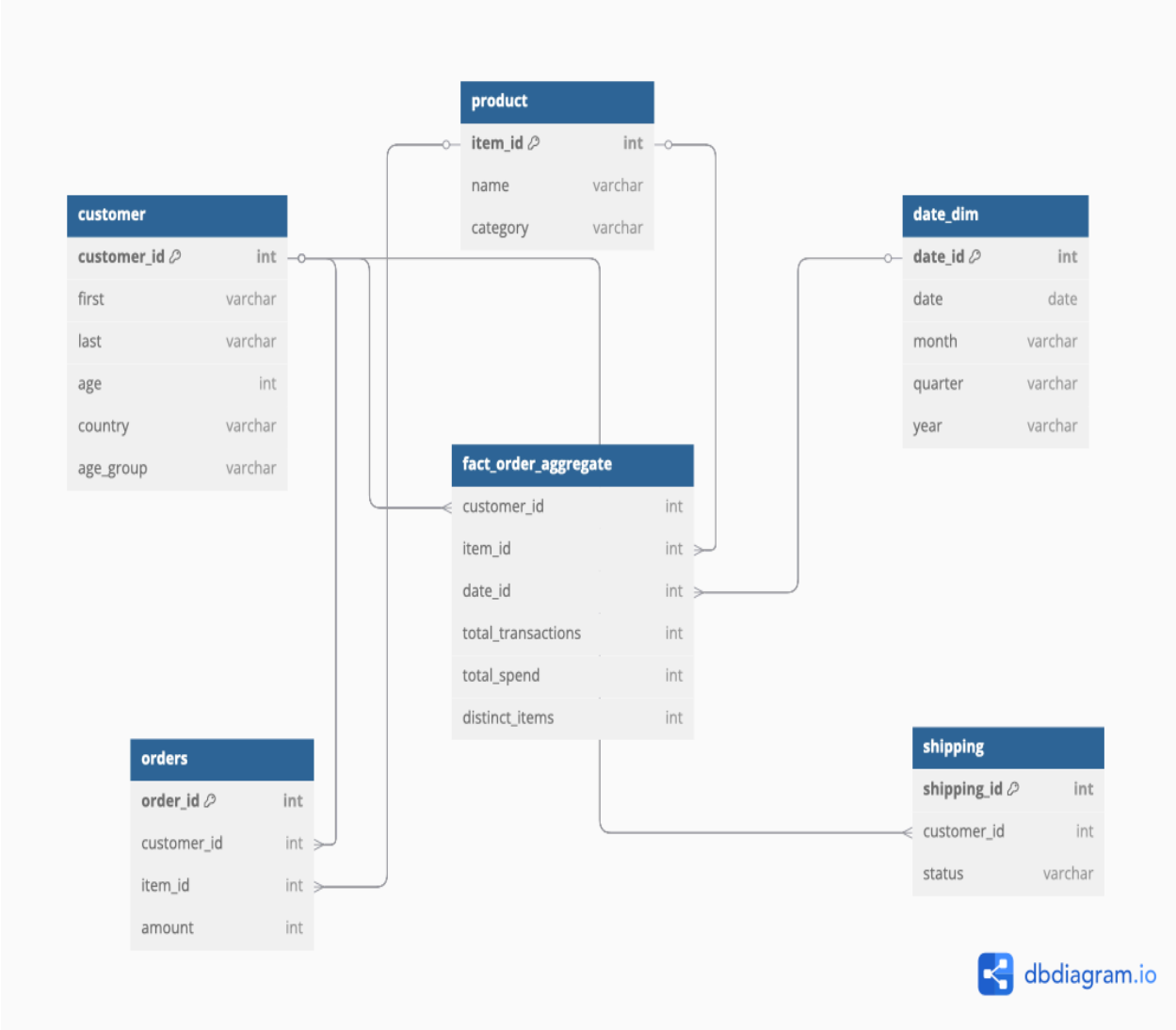
7. ERD Visual Diagram

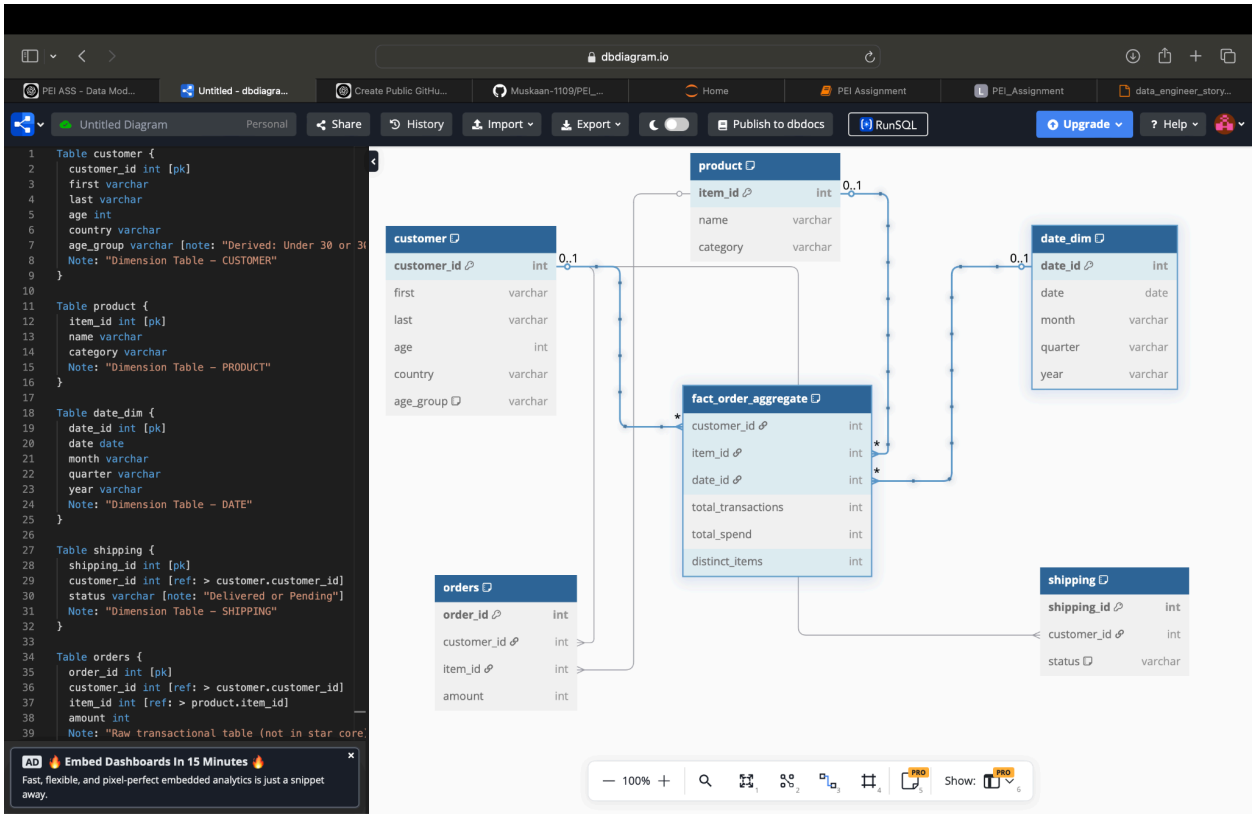




Relationship between the three tables

Star Schema: Customer Transaction & Delivery Reporting Model





1. Adding a FACT_ORDER_AGGREGATE Table

If reporting frequently needs **aggregated customer data**, we can **pre-aggregate** to reduce query time.

FACT_ORDER_AGGREGATE

- **Customer_ID:** Unique identifier from the customer table used for grouping.
- **Total_Transactions:** Count of all orders placed by the customer → COUNT(order_id).
- **Total_Spend:** Sum of all order amounts made by the customer → SUM(amount).
- **Distinct_Items:** Count of unique products purchased → COUNT(DISTINCT item).

Benefits:

- Faster dashboards
- Easier slicing/dicing

2. Normalizing `Item` into a `PRODUCT` Table

In the raw data, `Item` is just a string. But if you had attributes like category, price, stock… **for scalability in future**

```
PRODUCT
- Item_ID (PK)
- Name
- Category
- Price
```

And change `ORDER.Item` to `Item_ID` (FK).

Benefits:

- Real-world modeling
- Easier product-level insights (top categories, profitability, etc.)

3. Adding a `DATE_DIM` for Time Intelligence

If we plan to analyze **orders over time**, and `Order_Date` exists...

```
DATE_DIM
- Date_ID (PK)
- Date
- Month, Quarter, Year
- Is_Weekend
```

📌 Benefits:

- Enables trend, seasonality, cohort analysis

Proposed Enhancement: Add `order_date_key` for Time-Based Analysis

Rationale:

While the current raw `ORDER` table doesn't include a `date` column, **adding a placeholder or join logic to a `date_dim`table now** ensures your model is **future-proof** and **extensible** for time series reporting (e.g., monthly trends, YOY comparison, cohort analysis).

Adding a Placeholder in the Mapping Table

Target Column	Source Column(s)	Transformation / Logic	Data Type
`order_date_key`	N/A (future field)	Joinable surrogate key from date_dim table	INT

Note: The current source ORDER table does not have an order date field. For future scalability, an `order_date_key` field is proposed to enable date-based aggregations. This would require enriching the raw ORDER data with a timestamp during ETL or raw ingestion. Once available, this key can join to the `date_dim` table for flexible time-based reporting.

✅ Phase 3: Technical Specifications

Goal of This Phase

To prepare a comprehensive, implementation-ready technical story that enables:

- Building target tables and views using source-to-target mappings
- Applying clear transformation logic (e.g., aggregations, age buckets)
- Enabling QA engineers to independently test output for correctness
- Supporting the reporting requirements defined in Phase 1

This document provides technical specifications for building downstream tables to support business reporting requirements using the Customer, Order, and Shipping datasets.

Table 1: Customer_Order_Aggregate

Purpose:

Summarize total transactions, spend, and product-level metrics for each customer.

Source Tables:

- CUSTOMER
- ORDER

Join Logic:

ORDER.Customer_ID = CUSTOMER.Customer_ID

Source-to-Target Mapping:

Target Column	Source Column(s)	Transformation / Logic	Data Type
`customer_id`	customer.customer_id	Direct	INT
`customer_name`	customer.first, customer.last	`CONCAT(first, ' ', last)`	STRING

country	customer.country	Direct	STRING
age_group	customer.age	CASE WHEN age < 30 THEN 'Under 30' ELSE '30+'	STRING
item	order.item	Direct	STRING
transaction_count	order.order_id	COUNT(order_id) grouped by customer_id, item	INT
total_amount_spent	order.amount	SUM(amount) grouped by customer_id, item	INT

QA Test Cases:

QA Scenario	Test Logic
Null Check	Ensure no NULLs in customer_id, item, transaction_count, total_amount_spent
Join Integrity	Validate that all customer_id in final table exist in both customer and order
Aggregation Check	Manually validate transaction_count and total_amount_spent for 2 sample customers
Age Group Logic	Test edge cases: age = 29 should map to "Under 30", age = 30 to "30+"
Duplicates	Final output should have only one row per customer_id and item combination

Table 2: Pending_Amount_By_Country

Purpose:

Calculate total amount spent for customers whose deliveries are still pending, grouped by country.

Source Tables:

- CUSTOMER
- ORDER

- SHIPPING

Join Logic:

SHIPPING.Customer_ID = CUSTOMER.Customer_ID

ORDER.Customer_ID = CUSTOMER.Customer_ID

Source-to-Target Mapping:

Target Column	Source Column(s)	Transformation / Logic	Data Type
`country`	`customer.country`	Direct mapping	`STRING`
`total_amount_pending`	`order.amount`	`SUM(order.amount) where shipping.status = 'Pending'`	`INT`

QA Test Cases:

QA Scenario	Test Logic
Filter Check	Only include rows where shipping.status = 'Pending'
Country Mapping	Ensure each customer_id maps to a valid customer.country
Join Integrity	Validate all customer_id values in joined tables exist
Totals Validation	Manually cross-check pending totals for 2 countries
Duplicates	Ensure only 1 row per country in the final output

Table 3: `Top_Product_Per_Country`

Purpose:

Identify the **most purchased product** (by order count) in **each country**.

Source Tables:

- CUSTOMER
- ORDER

Join Logic:

```
ORDER.Customer_ID = CUSTOMER.Customer_ID
```

Source-to-Target Mapping:

Target Column	Source Column(s)	Transformation / Logic	Data Type
`country`	`customer.country`	Direct	`STRING`
`top_product`	`order.item`	Use DENSE_RANK() or ROW_NUMBER() on COUNT(order_id)	`STRING`
`purchase_count`	`order.order_id`	`COUNT(order_id) GROUP BY country, item`	`INT`

QA Test Cases:

QA Scenario	Test Logic
Ranking Logic	Confirm only rank 1 product appears per country, same rank only when the count of orders by country and item are same
Country Validation	All country values must match the CUSTOMER dimension
Count Accuracy	Manually verify COUNT(order_id) for top product in each country
Duplicates	Ensure only one row per country in the final output
Tie Handling	Define behavior for ties (e.g., return product that comes first alphabetically)

Table 4: Top_Product_By_Age_Group

Purpose:

Determine the most purchased product by customers in two age categories: **Under 30** and **30+**.

Source Tables:

- CUSTOMER
- ORDER

Join Logic:

```
ORDER.Customer_ID = CUSTOMER.Customer_ID
```

Source-to-Target Mapping:

Target Column	Source Column(s)	Transformation / Logic	Data Type
age_group	customer.age	CASE WHEN age < 30 THEN 'Under 30' ELSE '30+'	STRING
top_product	order.item	Use RANK() or ROW_NUMBER() on COUNT(order_id) per age group	STRING
purchase_count	order.order_id	COUNT(order_id) GROUP BY age_group, item	INT

QA Test Cases:

QA Scenario	Test Logic
Bucket Accuracy	Validate age = 29 maps to "Under 30", age = 30 maps to "30+"
Product Ranking	Ensure only the top product is returned per age group
Join Integrity	Validate all customer_id values exist in both CUSTOMER and ORDER
Count Validation	Check COUNT(order_id) for top product in each group
Nulls & Duplicates	No NULLs and 1 row per age group only

Table 5: `Low_Performing_Country`

Purpose:

Find the **country with the lowest number of transactions and spend**.

Source Tables:

- CUSTOMER
- ORDER

Join Logic:

```
ORDER.Customer_ID = CUSTOMER.Customer_ID
```

Source-to-Target Mapping:

Target Column	Source Column(s)	Transformation / Logic	Data Type
`country`	`customer.country`	Direct	`STRING`
`total_transactions`	`order.order_id`	`COUNT(order_id)`	`INT`
`total_spend`	`order.amount`	`SUM(order.amount)`	`INT`
`rank`	*derived*	`ROW_NUMBER() OVER (ORDER BY total_transactions, total_spend)`	`INT`

QA Test Cases:

QA Scenario	Test Logic
Rank Calculation	Ensure only the lowest ranking country is returned
Count Accuracy	Manually validate transaction and spend numbers for correctness
Tie Handling	Define logic for ties (e.g., return both or only first by rule)
Data Completeness	All countries present in CUSTOMER should be considered
Nulls & Duplicates	Final output should be clean — no NULLs, no duplicates

---

✅ Phase 4: Business Mapping

Goal of This Phase

To map each **business reporting requirement** directly to:

- ✅ The fields and tables in the star schema
- ✅ The transformations needed
- ✅ The logic behind the answer

This phase ensures that stakeholders and engineers can trace **every metric to a data source**.

Mapping Table:

Business Requirement	Tables Used	Key Fields	Filters	Logic Summary
1. Total amount spent for Pending deliveries, by country	`SHIPPING, CUSTOMER, ORDER`	`shipping.status, customer.country, order.amount`	`status = 'Pending'`	Join on customer_id → Filter on pending → Sum amount grouped by country
2. Per-customer: total transactions, quantity sold, amount spent, product details	`ORDER, CUSTOMER`	`order_id, amount, item, customer_name`	None	Join on customer_id → Group by customer_id + item → Count + Sum
3. Maximum product purchased per country	`ORDER, CUSTOMER`	`order_id, item, country`	None	Join → Group by country + item→ Count → Use RANK() or ROW_NUMBER() to select top 1

4. Most purchased product by age group (<30, ≥30)	`ORDER, CUSTOMER`	`order_id, item, age_group`	Derive age_group	Join → Create age_group → Group + Count → Get top product per group
5. Country with minimum transactions and lowest spend	`ORDER, CUSTOMER`	`order_id, amount, country`	None	Join → Group by country → Count + Sum → Use ROW_NUMBER() to find lowest

Notes:

- All five reports are **driven off the star schema** and **do not require separate data marts**
- Aggregations are done at the **reporting view layer**
- Fields such as `age_group`, `top_product`, and `rank` are **derived** and not directly stored

---