The provided dataset represents an e-commerce system stored in a single flat table containing 39 attributes. The data includes information related to customers, orders, products, sellers, payments, and shipping. Because all entities are combined in one table, redundancy and dependency issues may exist. This document analyzes the dataset structure and evaluates its normalization level.

Identify all entities in the dataset.

The dataset combines multiple logical business entities into one table, which leads to redundancy.

- **Customer** – person who buy or place order
- **Order** -- the order itself
- **Product** -- product info that placed in order
- **Seller.** – seller info sells the products
- **Payment**. – including all payment info (WEAK ENTITIY)

  → Its fully depend on order – composite primary key (order_id , payment_sequential)

It cannot exist without an Order

- **Order_Item**

  → Since we have multi products and in order we may place multi product so we had an many to many Relationship between order and product

Primary Key of Order_Items = (order_id, order_item_id)

Because:

- order_item_id = 1 can exist in many orders
- but (order_id + order_item_id) together are unique

- **Shipping** -- Shipping entity details linked to Order

- The likely primary key is (order_id, order_item_id).
- Because each row represents one product inside one order.
- order_id alone is not unique.
- order_item_id alone is not unique.

Detect redundant attributes. → same information is repeated in multiple rows

Because the primary key of the flat table is assumed to be:

(order_id, order_item_id),

each row represents one product within a specific order.

As a result, attributes that depend only on order_id, customer_id, product_id, or seller_id are repeated for every order item in the same order.

This creates redundancy.

Why Redundancy Exists ? - Primary key of flat table:**(order_id, order_item_id)**

Each row = one product inside one order.

So anything that does NOT change per order item will repeat.

1. Customer Attributes

The following attributes depend only on customer_id:

- customer_city
- customer_state
- customer_zip_code_prefix
- customer_unique_id

If an order contains multiple products, these customer attributes are repeated in every row of that order. Therefore, they are redundant in the flat table.

2. Product Attributes

The following attributes depend only on product_id:

- product_category_name
- product_weight_g
- product_length_cm
- product_height_cm
- product_width_cm
- product_name_lenght
- product_description_lenght
- product_photos_qty

If the same product appears in many orders, its descriptive information is stored repeatedly. This leads to redundancy.

3. Seller Attributes

The following attributes depend only on seller_id:

- seller_city
- seller_state
- seller_zip_code_prefix

If a seller sells multiple products across different orders, these values are duplicated many times in the table.

4. Derived Time Attributes

The following attributes are derived from order_purchase_timestamp:

- day_of_purchase
- month_of_purchase
- year_of_purchase
- month/year_of_purchase

Since they can be calculated from the timestamp, storing them separately introduces unnecessary redundancy.

Why Redundancy Is Dangerous in Database Design

- Data inconsistency
- Integrity problems
- Maintenance complexity

Redundancy is dangerous because it causes **data anomalies** and **inconsistency**.

When the same information is stored multiple times in different rows, any modification must be applied to all repeated records. If one record is updated and others are not, the database becomes inconsistent.

There are three main problems:

1.Update Anomaly

If customer information (like customer_city) is stored in many rows and the customer changes city, all rows must be updated.
If one row is not updated, incorrect data remains in the database.

2.insert Anomaly

We cannot insert a new customer unless an order exists, because customer data is tied to order rows.
This limits flexibility.

3.Delete Anomaly

If we delete the last order of a customer, we may accidentally delete all customer information from the system.

Identify functional dependencies Rule: A → B If A is known, B is determined.

Start From Primary Key
Assume flat table PK: **(order_id, order_item_id)**
So we automatically know:
(order_id, order_item_id) → product_id
(order_id, order_item_id) → price
(order_id, order_item_id) → freight_value

customer_id → customer_city, customer_state, customer_zip_code_prefix

product_id → product_category_name, product_weight_g, product_length_cm, product_height_cm, product_width_cm seller_id → seller_city, seller_state, seller_zip_code_prefix

order_id → customer_id, order_status, order_purchase_timestamp

(order_id, order_item_id) → product_id, freight_value

(order_id, payment_sequential) → payment_type, payment_installments, payment_value

Why the Dataset Violates 2NF

The dataset violates Second Normal Form (2NF) because the primary key of the table is composite:

(order_id, order_item_id).

For a table to be in 2NF, all non-key attributes must depend on the entire composite primary key.

- not on part of the composite key instead of the whole key

However, in this dataset:

- customer_id depends only on order_id.
- order_status depends only on order_id.
- seller attributes depend only on seller_id.
- product descriptive attributes depend only on product_id.

These attributes do not depend on the full composite key (order_id, order_item_id), but only on part of it or on another non-key attribute.

Therefore, the table contains partial dependencies and violates 2NF.

Why the Dataset Violates 3NF

The dataset violates Third Normal Form (3NF) because it contains transitive dependencies.

A table is in 3NF if no non-key attribute depends on another non-key attribute.

In this dataset:

- customer_city and customer_state depend on customer_id.
- product_category_name and product_weight_g depend on product_id.
- seller_city and seller_state depend on seller_id.
- day_of_purchase and month_of_purchase depend on order_purchase_timestamp.

Since customer_id, product_id, seller_id, and order_purchase_timestamp are not part of the primary key, these create transitive dependencies.

The dependency structure is:

Primary Key → Non-key Attribute → Another Non-key Attribute

Therefore, the table violates 3NF.

Create Tables

```sql
CREATE TABLE customer (
   customer_id VARCHAR(50) PRIMARY KEY,
   customer_unique_id VARCHAR(50) NOT NULL UNIQUE,
   customer_zip_code_prefix VARCHAR(10) NOT NULL,
   customer_city VARCHAR(100) NOT NULL,
   customer_state VARCHAR(50) NOT NULL
);


CREATE TABLE orders (
   order_id VARCHAR(32) PRIMARY KEY,
   customer_id VARCHAR(32) NOT NULL,
   order_status VARCHAR(32) NOT NULL,
   order_purchase_timestamp TIMESTAMP NOT NULL,
   order_approved_at TIMESTAMP NULL,
   order_delivered_carrier_date TIMESTAMP NULL,
   order_delivered_customer_date TIMESTAMP NULL,
   order_estimated_delivery_date TIMESTAMP NULL,

   FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
```

```
);

CREATE TABLE order_item (
    order_id VARCHAR(32) NOT NULL,
    order_item_id INT NOT NULL,
    product_id VARCHAR(32) NOT NULL,
    seller_id VARCHAR(32) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    freight_value DECIMAL(10,2) NOT NULL,

    PRIMARY KEY (order_id, order_item_id),

    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id),
    FOREIGN KEY (seller_id) REFERENCES seller(seller_id)

);
CREATE TABLE product (
product_id VARCHAR(32) PRIMARY KEY,
product_category_name VARCHAR(100) NOT NULL,
product_name_length INT NOT NULL,
product_description_length INT NOT NULL,
product_photos_qty INT NOT NULL,
product_weight_g INT NOT NULL,
product_length_cm DECIMAL(10,2) NOT NULL,
product_height_cm DECIMAL(10,2) NOT NULL,
product_width_cm DECIMAL(10,2) NOT NULL
);

CREATE TABLE seller (
seller_id VARCHAR(32) PRIMARY KEY,
seller_zip_code_prefix VARCHAR(10) NOT NULL,
seller_city VARCHAR(100) NOT NULL,
seller_state VARCHAR(50) NOT NULL
);

CREATE TABLE payment (
    order_id VARCHAR(32) NOT NULL,
    payment_sequential INT NOT NULL,
    payment_type VARCHAR(50) NOT NULL,
    payment_installments INT NOT NULL,
    payment_value DECIMAL(10,2) NOT NULL,

    PRIMARY KEY (order_id, payment_sequential),

    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

CREATE TABLE shipping (
    order_id VARCHAR(32) PRIMARY KEY,
    shipping_limit_date TIMESTAMP NOT NULL,
    order_delivered_carrier_date TIMESTAMP NULL,
    order_delivered_customer_date TIMESTAMP NULL,
    order_estimated_delivery_date TIMESTAMP NOT NULL,

    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

NOTE : The price is stored in the order_item table to preserve the historical transaction value at the time of purchase. Since product prices may change over time, storing the price only in the product table would cause inconsistencies in past orders. Therefore, the price must be stored at the order item level to ensure data integrity and accurate financial records.

---

schema is correct for normalization.

But production-ready schema includes:

- Explicit cascade rules
- Indexes for heavy joins
- Data validation constraints

1. **ON DELETE / ON UPDATE**

Right now your foreign keys look like:

```
FOREIGN KEY (order_id) REFERENCES orders(order_id)
```

But what happens if:

An order is deleted?

Should:

- Its order_items be deleted automatically?
- Its payment records be deleted?
- Its shipping record be deleted?

Right now, behavior depends on default database settings.

In a real production schema, we define it explicitly.

Example improvement for order_item:

```
FOREIGN KEY (order_id)
REFERENCES orders(order_id)
ON DELETE CASCADE
ON UPDATE CASCADE
```

Why?

Because:

If an order is deleted → its items must disappear too.

Otherwise you create orphan records.

Same applies to:

- payment
- shipping

2. **Indexing for Performance**

Primary keys are automatically indexed.

But what about foreign keys?

For example:

- order_item.product_id

- order_item.seller_id
- payment.order_id

If you frequently JOIN these tables,
you should add indexes.

3. **CHECK Constraints (Data Integrity)**

Example:

```
CHECK (price >= 0)
CHECK (freight_value >= 0)
CHECK (payment_installments > 0)
```

Prevents invalid data from entering database.

4. **ENUM or Constraint for order_status**

Instead of:

```
order_status VARCHAR(32)
```

You could restrict it:

```
CHECK (order_status IN ('delivered', 'shipped', 'canceled', 'processing'))
```

Real-World Design Considerations

Although the schema was normalized to Third Normal Form (3NF), practical database design in real financial systems requires additional considerations beyond normalization.

1. Deletion Policies

In a purely logical model, dependent entities such as order_item and payment can use ON DELETE CASCADE, since they cannot exist without their parent order.

However, in real financial systems:

- Orders represent legal and accounting records.
- Payments are financial transactions.

- Historical data must be preserved for auditing and compliance.

Therefore, instead of deleting orders and related records, production systems typically:

- Use **soft deletion** (e.g., is_active flag).
- Use ON DELETE RESTRICT to prevent accidental data loss.
- Archive historical records instead of removing them.

---

2. Financial Data Integrity

Payment records must remain accurate and immutable.
Even if a customer account is deleted or deactivated, past financial transactions should remain stored in the database.

This ensures:

- Legal compliance
- Accurate reporting
- Audit traceability
- Data consistency

---

3. Schema vs Production Reality

While normalization ensures elimination of redundancy and proper dependency structure, real-world database systems must also consider:

- Performance optimization (indexes)
- Data validation constraints
- Security
- Audit logging
- Transaction management

Therefore, a production-ready schema extends beyond pure normalization theory.

Business Logic

If a customer account is deleted:

Should we delete:

- All past orders?
- All payment history?
- All financial records?

In real companies (Amazon, banks, etc.):

- Orders are financial and legal records.
- They must be preserved for accounting and auditing.

If you delete a customer and cascade delete all orders:

You lose:

- Revenue history
- Tax records
- Legal evidence
- Payment data

That is very dangerous.

in real-world systems:

do NOT cascade delete orders when deleting a customer.

Instead, options are:

Option 1: Soft Delete

Add a column in customer:

is_active BOOLEAN DEFAULT TRUE

Instead of deleting the row, mark it inactive.

Option 2: Restrict Deletion

Use:

ON DELETE RESTRICT

This prevents deleting a customer if they have orders.

Option 3: Set NULL (rarely used)

ON DELETE SET NULL

But this breaks historical traceability in most systems.

Data Loading

Rule Before Writing Any INSERT

must load tables in the correct order.
Because of FOREIGN KEY constraints.
cannot insert into orders before inserting into customer.

Correct Loading Order

1. customer
2. product
3. seller
4. orders
5. shipping
6. order_item
7. payment

If you change the order → FK errors.

Step 1 – Create `raw_table` This is called a **staging table**.

---

raw_table SQL Server

```
CREATE TABLE raw_table (
    order_id VARCHAR(32),
    order_item_id INT,
    customer_id VARCHAR(32),
    customer_unique_id VARCHAR(32),
    customer_zip_code_prefix VARCHAR(10),
    customer_city VARCHAR(100),
    customer_state VARCHAR(50),

    product_id VARCHAR(32),
    product_category_name VARCHAR(100),
    product_name_length INT,
    product_description_length INT,
    product_photos_qty INT,
    product_weight_g INT,
    product_length_cm DECIMAL(10,2),
    product_height_cm DECIMAL(10,2),
    product_width_cm DECIMAL(10,2),

    seller_id VARCHAR(32),
    seller_city VARCHAR(100),
    seller_state VARCHAR(50),
    seller_zip_code_prefix VARCHAR(10),

    payment_type VARCHAR(50),
    payment_sequential INT,
    payment_installments INT,
    price DECIMAL(10,2),
    freight_value DECIMAL(10,2),
    payment_value DECIMAL(10,2),

    shipping_limit_date DATETIME2,
    order_purchase_timestamp DATETIME2,
    order_approved_at DATETIME2,
    order_delivered_carrier_date DATETIME2,
    order_delivered_customer_date DATETIME2,
    order_estimated_delivery_date DATETIME2,

    day_of_purchase INT,
    month_of_purchase INT,
    year_of_purchase INT,
    month_year_of_purchase VARCHAR(10),

    order_status VARCHAR(50),
    order_unique_id VARCHAR(32)
);
```

Step 2 – Load CSV in SQL Server

```
BULK INSERT raw_table
FROM 'C:\path\to\your\file.csv'
WITH (
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '\n',
    TABLOCK
);
```

load **customer** from `raw_table`:

```
INSERT INTO customer (
    customer_id,
    customer_unique_id,
    customer_zip_code_prefix,
    customer_city,
    customer_state
)
SELECT DISTINCT
    customer_id,
    customer_unique_id,
    customer_zip_code_prefix,
    customer_city,
    customer_state
FROM raw_table
WHERE customer_id IS NOT NULL;
```

Why This Is Correct

`SELECT DISTINCT`

Because the flat table contains many repeated customer rows (one per order item).
DISTINCT removes duplicates.

Column Order Matches Target Table

Very important in SQL Server.
The selected columns must match the insert column list.

`WHERE customer_id IS NOT NULL`

Prevents inserting invalid records and avoids PK violations

Handling Duplicates During Data Loading

When loading data from the raw flat table into normalized tables, the `SELECT DISTINCT` statement is used to remove duplicate rows caused by denormalization.

However, it is important to understand how `DISTINCT` works.

`DISTINCT` removes rows only when **all selected columns are identical**.

For example:

product_id	product_weight_g
P100	500
P100	500

Using:

```
SELECT DISTINCT product_id, product_weight_g
FROM raw_table;
```

Returns:

product_id	product_weight_g
P100	500

Because both rows are identical.

However, if the dataset contains inconsistent data:

product_id	product_weight_g
P100	500
P100	520

The same query will return:

product_id	product_weight_g
P100	500
P100	520

Because the rows are not identical.

This means that `DISTINCT` does not guarantee one row per `product_id`.
It guarantees uniqueness based on the combination of selected columns.

Importance in ETL

If inconsistent values exist for the same entity identifier, inserting into a normalized table with a primary key constraint may result in errors.

In production systems, such inconsistencies require:

- Data cleaning
- Aggregation (like GROUP BY)
- Business validation rules (CHECK)

ETL Assumptions and Data Quality Considerations

1. ETL Assumptions

During the data loading process from the denormalized raw table into the normalized 3NF schema, the following assumptions were made:

- The raw dataset contains consistent entity data.
- Each `product_id` corresponds to a single set of product attributes.
- Each `seller_id` corresponds to a single seller location.
- Each `customer_id` corresponds to consistent customer information.
- No conflicting attribute values exist for the same primary identifier.

Because of these assumptions, `SELECT DISTINCT` was used to remove duplicate rows created by denormalization.

---

2. Data Quality Risks

In real-world datasets, inconsistencies may occur. For example:

- The same `product_id` may appear with different `product_weight_g` values.
- The same `customer_id` may appear with different city or state values.
- Payment values may not match the sum of order items.

If such inconsistencies exist, `SELECT DISTINCT` would not guarantee one row per entity. Instead, it could result in multiple rows for the same identifier, potentially causing primary key violations.

3.Production-Level Handling

In enterprise systems, additional ETL logic would be required, such as:

- Data validation rules
- Aggregation using `GROUP BY`
- Data cleansing procedures
- Logging of rejected records
- Conflict resolution policies

In this project, the dataset is assumed to be clean and internally consistent; therefore, normalization and data loading were performed using `SELECT DISTINCT` without additional transformation.

Error 2627 (UNIQUE KEY violation)

- during loading data rom raw_table to customer table
- `SELECT DISTINCT` is not enough when attributes differ.

- The error happened while inserting into `customer`.

There are multiple rows with the same:

```
customer_id
```

But with different:

- customer_city
- customer_state
- zip_code
- or customer_unique_id

Violation of UNIQUE KEY constraint 'UQ__customer__...'
Duplicate key value is (004288347e5e88a27ded2bb23747066c)

have multiple rows where:

```
customer_unique_id = 004288347e5e88a27ded2bb23747066c
```

But they have **different customer_id values**.

One real customer (unique_id) is linked to multiple customer_id values.

Confirm the Issuen by Run this diagnostic query:

```
SELECT
    customer_unique_id,
    COUNT(DISTINCT customer_id) AS id_count
FROM raw_table
GROUP BY customer_unique_id
HAVING COUNT(DISTINCT customer_id) > 1;
-- 2700 row Dublicated customer_unique_id
```

Same person places multiple orders → different `customer_id` each time.

So your assumption that:

```
customer_unique_id UNIQUE
```

That constraint is too strict.

the data model is:

- customer_unique_id → real person
- customer_id → order-specific identifier

Relationship:

- One customer_unique_id → many customer_id

That is 1-to-many.

One person
→ can place multiple orders
→ each order may generate a new `customer_id`

So your earlier UNIQUE constraint on `customer_unique_id` was incorrect.

Customer Identifier Clarification

During data loading, it was discovered that `customer_unique_id` is not unique in the dataset. Approximately 2700 cases exist where a single `customer_unique_id` is associated with multiple `customer_id` values.

This indicates that:

- `customer_unique_id` represents the real customer.

- `customer_id` represents an order-specific identifier.

Therefore, the UNIQUE constraint on `customer_unique_id` was removed to correctly reflect the 1-to-many relationship between customers and their orders.

Data Quality Validation and Diagnostic Process

Before loading data from the denormalized `raw_table` into the normalized 3NF schema, a series of diagnostic queries were executed to validate entity consistency and prevent constraint violations.

Because the raw table contains repeated records due to denormalization, it was necessary to verify that repeated entities did not contain conflicting attribute values.

---

1. Customer Validation

A diagnostic query was executed to detect whether a single `customer_unique_id` was associated with multiple `customer_id` values.

The results showed approximately 2700 cases where one `customer_unique_id` corresponded to multiple `customer_id` values.

This indicates that:

- `customer_unique_id` represents the real person.
- `customer_id` represents an order-level identifier.

Therefore, the UNIQUE constraint on `customer_unique_id` was removed to correctly model the 1-to-many relationship.

---

2. Product Validation

A conflict detection query was executed to verify that each `product_id` had consistent descriptive attributes (category, weight, dimensions).

The diagnostic query returned no conflicting records.

This confirms that the dataset maintains consistent product information across all repeated rows.

Therefore, `SELECT DISTINCT` was safely used to load the `product` table.

3. Seller Validation

A similar diagnostic query was executed for `seller_id` to detect variations in city, state, or zip code.

The query returned no conflicting records.

This confirms that seller location data is internally consistent in the dataset.

---

4. Payment Validation

Because payment data is financial in nature and more sensitive to inconsistency, a diagnostic query was executed to detect conflicting values for each `(order_id, payment_sequential)` pair.

The validation checked:

- payment_type
- payment_installments
- payment_value

The diagnostic query returned no conflicting records.

This result is expected in well-designed transactional systems, where payment values are not duplicated with inconsistent amounts for the same order and payment sequence.

Therefore, the `payment` table was safely populated using `SELECT DISTINCT`.

ETL Conclusion

The diagnostic phase confirmed that:

- No attribute conflicts exist for `product`, `seller`, or `payment`.
- The only structural modeling issue involved `customer_unique_id`, which was resolved by adjusting the schema design.
- The dataset is internally consistent and suitable for normalization into 3NF.

This validation step ensures data integrity before enforcing primary and foreign key constraints in the normalized schema.

This will show all cases where: Same unique customer → multiple customer_id values.

Detect Conflicting Customer Records by Running this diagnostic query:

```
SELECT
  customer_id,
  COUNT(DISTINCT customer_city) AS city_count,
  COUNT(DISTINCT customer_state) AS state_count,
  COUNT(DISTINCT customer_zip_code_prefix) AS zip_count,
  COUNT(DISTINCT customer_unique_id) AS unique_id_count
FROM raw_table
GROUP BY customer_id
HAVING
  COUNT(DISTINCT customer_city) > 1
  OR COUNT(DISTINCT customer_state) > 1
  OR COUNT(DISTINCT customer_zip_code_prefix) > 1
  OR COUNT(DISTINCT customer_unique_id) > 1;
```

If there is any duplicate

- ➔ most recent order information (using MAX(order_purchase_timestamp)
- ➔ But we didn't detect any after running diagnostic query

Insert for Customer

Instead of DISTINCT, use GROUP BY + Aggregation: