Data Warehouse Design

E-Commerce Marketplace using MySQL

Saman Teymouri

February 2025

CONTENTS

ABSTRACT

This study provides the design and implementation of an analytic data warehouse for an online marketplace in order to create a personalized recommendation system. The paper involves initial schema using Chen notation for Entity Relationship Diagram, normalized schema up to third normal form, and SQL-based data analysis. Key objectives of the marketplace analytic system include vendor, customer, and product analysis which provide various insights such as identifying profitable vendors, recognizing customer purchasing behaviors, and extracting product trends. The study also explains query optimization techniques, ACID properties, and CAP theorem applications in the context of the marketplace analytic data warehouse. Finally, the report studies a real-world example of Alibaba's data warehousing to extract details about its functionalities.

1. INTRODUCTION

A data warehouse is a system to collect data from different sources and store them in a centralized location. It is used mostly for analytic purposes to give insight to stakeholders and help them make data-driven decisions. It is also beneficial to provide a recommendation system to add value to the system (Kimball and Ross, 2013). Businesses can use it to create monthly or annual reports, assess performance, and find patterns and trends. Moreover, data warehousing systems help them analyze large amounts of data efficiently to make strategic decisions and gain valuable insights into customers' behavior, sales patterns, and market trends (Inmon, 2005).

A marketplace is a business that connects buyers and sellers to trade a variety of products in different categories. Marketplaces have several challenges to create competitive edge against their rivals. They need to analyze the behavior of customers and vendors to provide a better service for both sides which guarantees their growth. This growth also forces them to utilize more effective solutions to store and analyze large amounts of historical data. Data warehousing systems play a significant role in this area by offering scalability, performance, and reliability in the process of creating data-driven insights from analytic data (Laudon and Traver, 2021).

There are several steps for implementing a data warehouse in the marketplace context effectively. The first step is to identify business requirements and objectives such as monitoring vendors' performance, customers' behavior, and product trends. The next step is data modeling which is related to designing a comprehensive schema to store data obtained from different sources. Afterward, data extraction, transformation, and loading (ETL Process) are required to populate data into the data warehouse. Finally, the whole system should undergo an extensive testing procedure to guarantee accuracy, performance, and scalability (Kimball and Ross, 2013).

2. ANALYTIC DATA WAREHOUSE DESIGN

2.1. Analytic Use Cases

Analytic use cases are designed to show how data analytics can address business problems. They provide exploratory analysis of data and apply analytical methods to achieve measurable results. These results help decision-makers or stakeholders to have better insight into their businesses (Davenport & Harris, 2007). Analytic use cases are typically categorized into descriptive, diagnostic, predictive, or prescriptive analytics (Provost & Fawcett, 2013).

This study suggests three analytic use cases (objectives) to go on with:

1. **Vendor Analysis:**

   The question is how much the marketplace earns from vendor commissions monthly. The results should be shown vendor by vendor and ordered based on the commission amount.

2. **Customer Analysis:**

   This is about customer behavior according to different aspects such as favorite product categories, being a member or guest, or their zip codes. These analysis can give insight to invest in specific categories or optimum places for warehouses to reduce the cost of deliveries.

3. **Product Analysis:**

   The question is what products are the most popular ones. The other issue is that if having more vendors helps the marketplace sell higher number of a product or acts against it. The analysis can also be conducted about seasonal products (number of top-selling products in each quarter) for better inventory management.

2.2. Entity Relationship Diagram

Entity Relationship Diagram (ERD) is a data model that can show important concepts about the real world in a diagram. One of the ways to use this model is Chen notation (Chen, 1976). This study uses Chen notation to depict the relationship between each part of the marketplace (Entity) and their characteristics (Attribute) (Figure 1). For creating this diagram, this paper uses visual-paradigm online tools which provides extensive free tools for making different types of diagrams (Visual Paradigm Online, 2025).

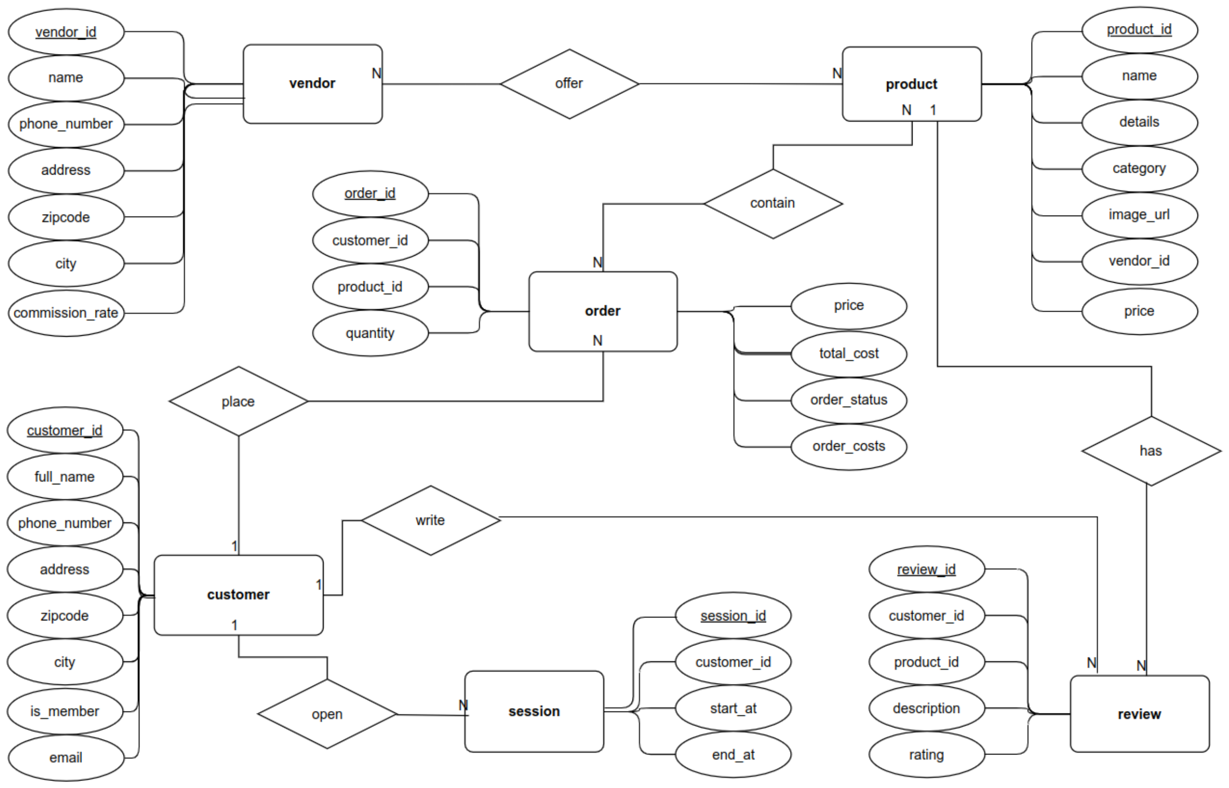


Figure 1: Entity Relationship Diagram (ERD) Using Chen Notation

2.3. **The Schema Normalization**

The normalization process has different levels that start from First Normal Form (1NF) (Codd, 1970). This study continues this process until reaching the Third Normal Form (3NF) (Codd, 1971).

Based on Codd (1970), the First Normal Form schema should satisfy the below conditions in every relation:

- **Atomic Values**: Each attribute must contain atomic (indivisible) values.
- **Unique Attribute Names**: Each attribute must have a unique name.
- **Single Data Type per Attribute**: Each attribute must contain values of the same data type.
- **Order Independence**: The order of tuples and attributes should not affect the meaning of the data.
- **Unique Tuples**: Each tuple must be unique.

According to these conditions, the order_status column in the order entity should be splitted. Each status of every order should be stored in a single row instead of comma-separated values. Respectively, order_cost for each status should be stored in a column of that row.

The other change is about customer full_name. It should be broken down into two attributes, first_name and last_name. The address field in customer and vendor entities should also be converted into street and house_number attributes.

Codd (1971) states following conditions for a schema to be considered as a Second Normal Form:

- **Must be in First Normal Form**
- **No Partial Dependencies**: Every attribute that is not part of any candidate key must be fully dependent on the entire primary key not part of it.

Based on these conditions, in the order entity, order_costs attribute is dependent on order_status, and price and quantity attributes are dependent on a combination of order_id and product_id. These problems lead us to create order_status and order_detail entities.

The other example is the price attribute in the product entity that is dependent on product_id and vendor_id. It forces us to create a vendor_product entity.

Codd (1971) specifies the following rules for the Third Normal Form schema:

- **Must be in Second Normal Form**
- **No Transitive Dependencies**: All attributes should be dependent only on the primary key, not other attributes.

To address 3NF requirements, we need to create city, category, and status entities and put the attributes related to them in their entities.

After normalization, our entities increase from 6 to 12. The results are shown below (Figure 2). MySQL ER Diagram tools are used to create this ERD. As it is visible in Figure 2, the notation is different from Chen notation, and it is more advanced and understandable for larger schemas.

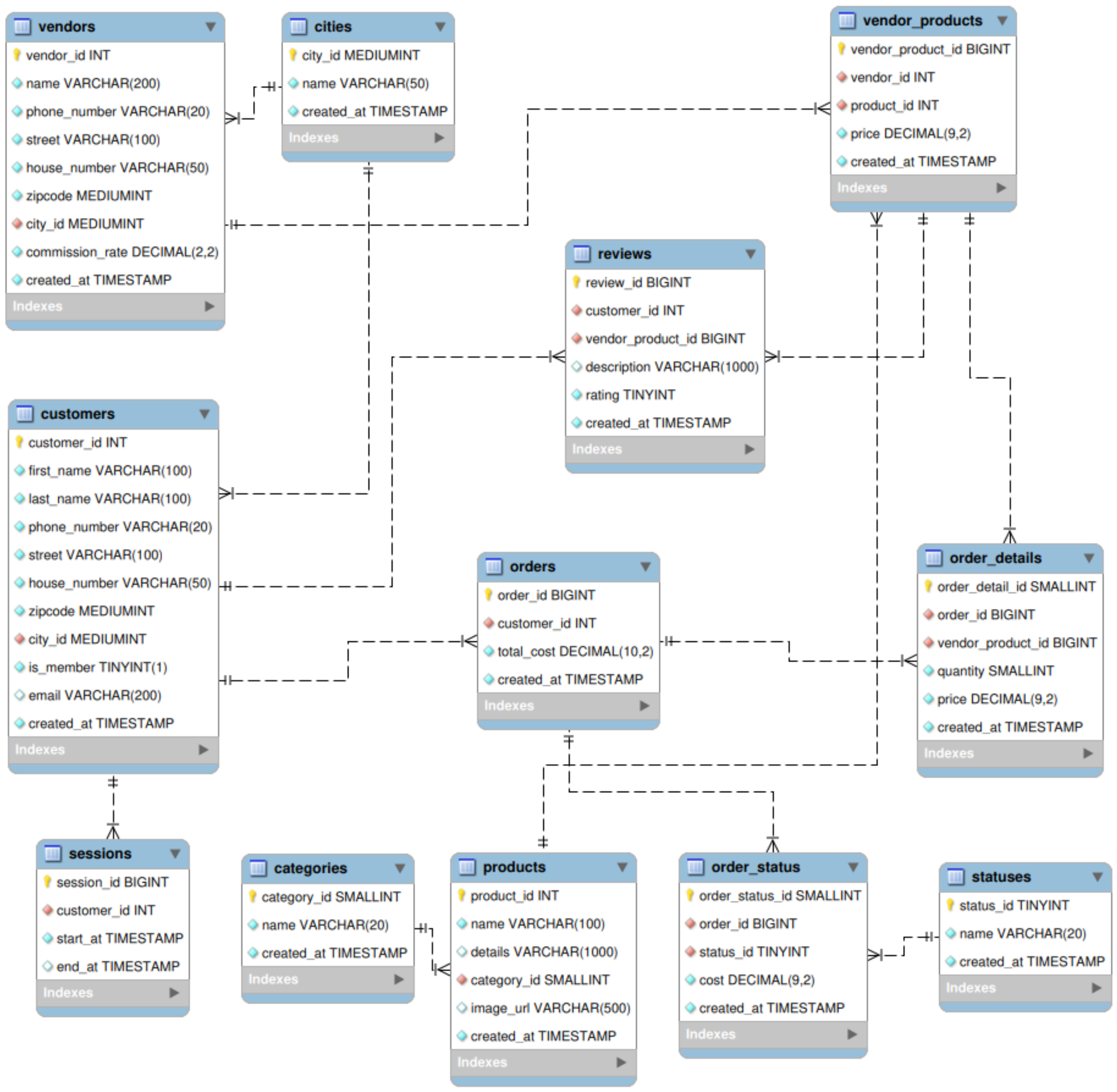


Figure 2: Entity Relationship Diagram (Advanced) After Normalization

2.4. Design Explanation

At this stage, the schema reaches the Third Normal Form which is the enough level in most designs. Now we can discuss each entity (table) to know the reasons behind its design. Before specific analysis of each table, there are some general rules obeyed in all tables. All tables have column created_at which shows the date and time record created. Since it is a data warehouse and updating the records is not necessary most of the time, the design does not use the updated_at column. The other rule is about data types and sizes. All varchar columns have fixed sizes based on the need to reduce storage usage. This fact also exists about numeric columns which have types of tinyint to bigint according to the necessity. This design uses decimal data types for prices or rates to manage the decimal point more effectively.

There are some unified naming rules for all columns. For instance, all column names are lower-case and snake-case for two-word column names. Only primary keys are named by the table name and the other columns do not contain table names in their names. It helps us to use a unique name for all primary keys even if they are used in other tables as foreign keys.

The first tables to discuss are products, vendors, and their relationship vendor_products table. The products table contains a unique column “name”, a “details” column for further descriptions, and a foreign key to the categories table which provides the ability to analyze based on categories. The vendors table has two unique columns “name” and “phone_number”. It also forms the address of the vendor in the “street”, “house_number”, “zipcode”, and “city_id”. This design enables us to analyze vendors based on their cities or neighborhoods. The other important column is “commission_rate” which is used to calculate revenue gained by each vendor. The “zipcode” columns can only contain numbers with exact 5 digits and the “commission_rate” must be between 0 and 1. Based on the project's need to have different vendors for the same products, vendor_products table is created. It has foreign keys from vendors and products to show their relationship as well as a price column to store different prices for each product offered by vendors. The combination of “product_id” and “vendor_id” should be unique (Alternate Key) and “price” cannot be negative.

The other important table to discuss is customers. It contains two unique columns “phone_number” and “email” that can be used to address every customer. The combination of “first_name” and “last_name” is not considered a unique value since it can be duplicate easily. The way of storing addresses is similar to vendors with the same constraints and abilities. The “is_member” column shows that a customer is a member or guest. For guest customers is not necessary to fill the email column.

The next table is order_statuses which plays two roles. First, it helps track every level an order goes through. It enables us to know if an order is completed or not, shipped, or returned. It also has a cost for each status of the order. For example, an order may have cost for shipment or re-attempts after return or in case of not successful delivery. In order_statuses the combination of “order_id” and “status_id” must be unique and “cost” cannot accept negative values.

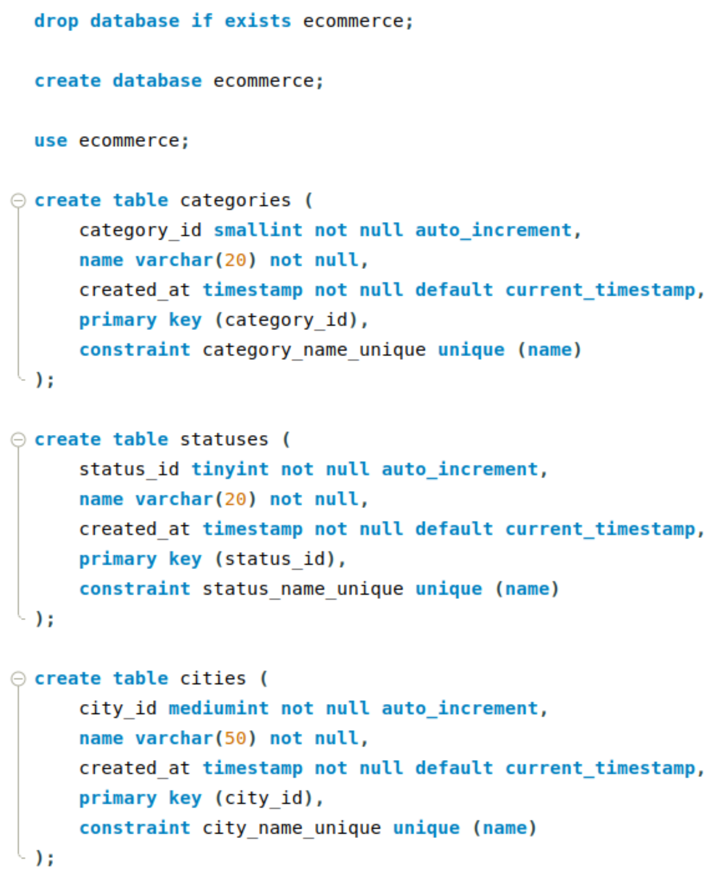
The main tables of the marketplace are orders and order_details. The orders table contains the “customer_id” and the “total_cost” of the order. The “total_cost” is necessary since it can be different from the multiplication of quantity and price in the presence of discounts, special offers, etc. The order_details has a foreign key from the vendor_products table to address the product and its vendor. The quantity is the number of each product in each order and the price shows the price of it at the time of purchase. The price may be changed after purchase by the vendors. In the orders, “total_cost” must be greater than or equal to zero, and “quantity” and “price” must be greater than zero in order_details table.

The other table is reviews which stores the shopping experience of the customers. Each customer can write a review for each product of a vendor and assign a rating to it. The “rating” must be between 1 and 5 and the combination of “customer_id” and “vendor_product_id” must be unique.

The last table that needs explanation is sessions which stores the customer’s starting and ending date time on the website. It helps us calculate the spent time by each customer on the website.

2.5. Database Implementation

After discussing design choices, it is time to implement a database via SQL queries. At first, the database will be deleted if exists, and then a new database named ecommerce is created. The next step is creating some tables such as categories, statuses, and cities (Figure 3).

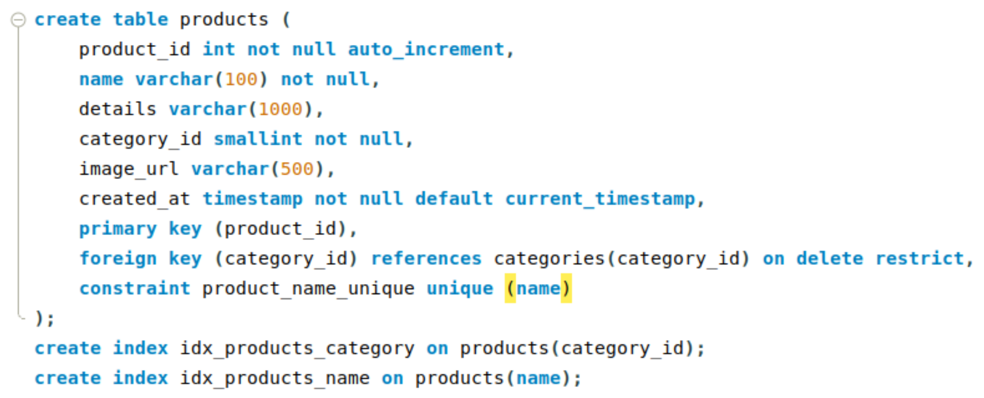


```
drop database if exists ecommerce;

create database ecommerce;

use ecommerce;

create table categories (
    category_id smallint not null auto_increment,
    name varchar(20) not null,
    created_at timestamp not null default current_timestamp,
    primary key (category_id),
    constraint category_name_unique unique (name)
);

create table statuses (
    status_id tinyint not null auto_increment,
    name varchar(20) not null,
    created_at timestamp not null default current_timestamp,
    primary key (status_id),
    constraint status_name_unique unique (name)
);

create table cities (
    city_id mediumint not null auto_increment,
    name varchar(50) not null,
    created_at timestamp not null default current_timestamp,
    primary key (city_id),
    constraint city_name_unique unique (name)
);
```

Figure 3: Create Database and Some Tables

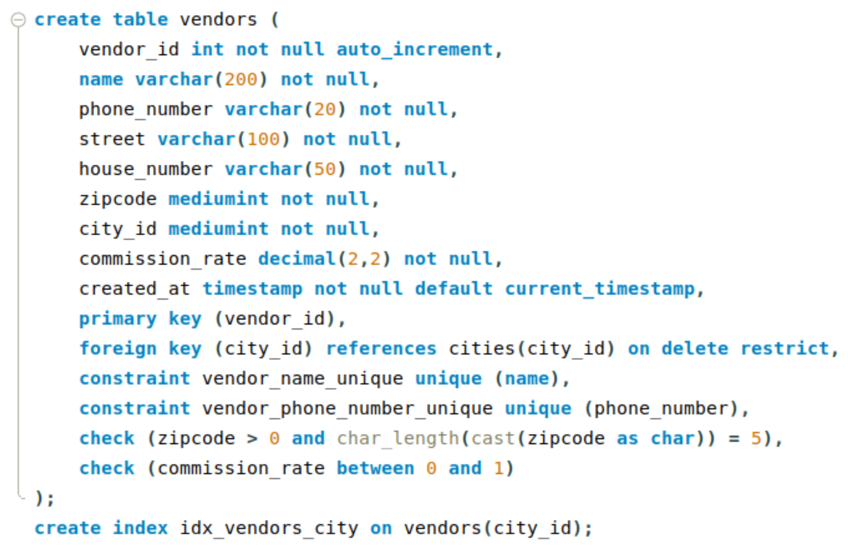
The primary key of all tables are auto_increment to make the inserting process simpler and less faulty. The created_at field has the default value of current_timestamp to be filled in any situation with a valid value.

The next tables to create are products, vendors, and vendor_products (Figure 4-6). All of them have foreign keys with "on delete restrict" which means prevent deletion of the main table record in case of being used as a foreign key in these tables.



```
create table products (
    product_id int not null auto_increment,
    name varchar(100) not null,
    details varchar(1000),
    category_id smallint not null,
    image_url varchar(500),
    created_at timestamp not null default current_timestamp,
    primary key (product_id),
    foreign key (category_id) references categories(category_id) on delete restrict,
    constraint product_name_unique unique (name)
);
create index idx_products_category on products(category_id);
create index idx_products_name on products(name);
```

Figure 4: Create Products Table

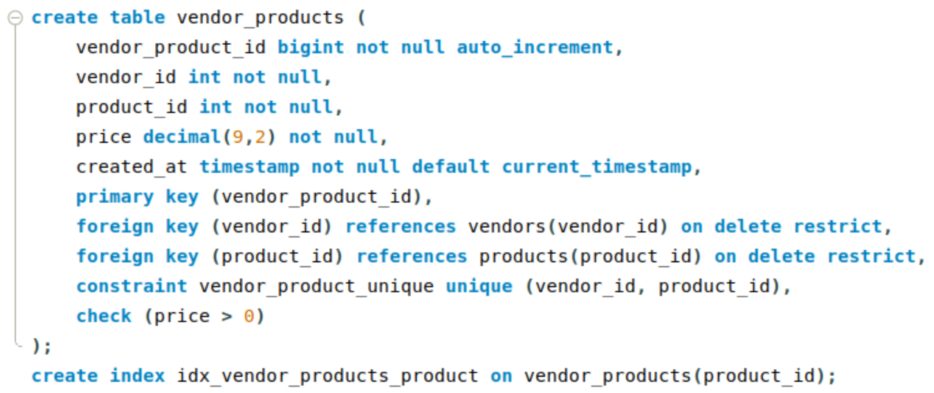


```
create table vendors (
    vendor_id int not null auto_increment,
    name varchar(200) not null,
    phone_number varchar(20) not null,
    street varchar(100) not null,
    house_number varchar(50) not null,
    zipcode mediumint not null,
    city_id mediumint not null,
    commission_rate decimal(2,2) not null,
    created_at timestamp not null default current_timestamp,
    primary key (vendor_id),
    foreign key (city_id) references cities(city_id) on delete restrict,
    constraint vendor_name_unique unique (name),
    constraint vendor_phone_number_unique unique (phone_number),
    check (zipcode > 0 and char_length(cast(zipcode as char)) = 5),
    check (commission_rate between 0 and 1)
);
create index idx_vendors_city on vendors(city_id);
```

Figure 5: Create Vendors Table

These tables also have indexes on some of their fields to make the search operation or query executions more performant. The products table has an index on "category_id" and "name" because these fields are commonly used for filtering products or searching them. The vendors table has an index on "city_id" to make it more efficient to search the vendors of the same city.

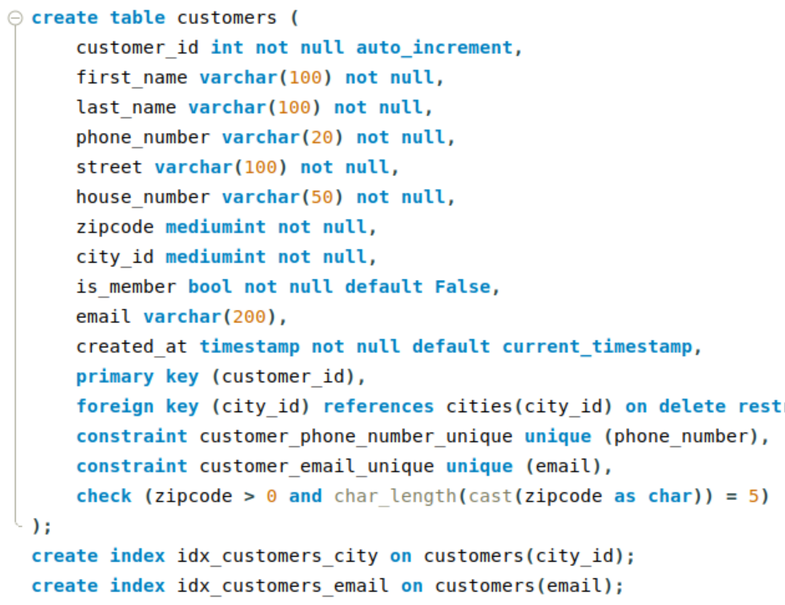
Moreover, vendor_products has an index on the “product_id” to be able to search vendors of a product faster.



```sql
create table vendor_products (
    vendor_product_id bigint not null auto_increment,
    vendor_id int not null,
    product_id int not null,
    price decimal(9,2) not null,
    created_at timestamp not null default current_timestamp,
    primary key (vendor_product_id),
    foreign key (vendor_id) references vendors(vendor_id) on delete restrict,
    foreign key (product_id) references products(product_id) on delete restrict,
    constraint vendor_product_unique unique (vendor_id, product_id),
    check (price > 0)
);
create index idx_vendor_products_product on vendor_products(product_id);
```

Figure 6: Create Vendor_Products Table

The next table is customers (Figure 7) has an index on “city_id” and “email”. The first index is for efficiency in analysis, and the second one is for searching for a customer at login faster.

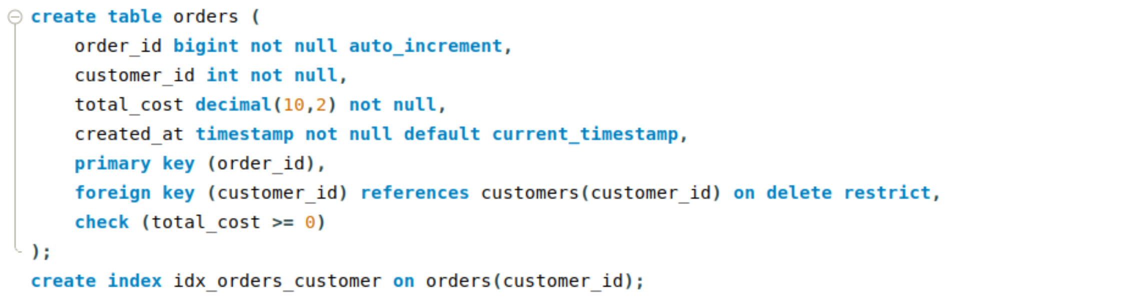


```sql
create table customers (
    customer_id int not null auto_increment,
    first_name varchar(100) not null,
    last_name varchar(100) not null,
    phone_number varchar(20) not null,
    street varchar(100) not null,
    house_number varchar(50) not null,
    zipcode mediumint not null,
    city_id mediumint not null,
    is_member bool not null default False,
    email varchar(200),
    created_at timestamp not null default current_timestamp,
    primary key (customer_id),
    foreign key (city_id) references cities(city_id) on delete restrict,
    constraint customer_phone_number_unique unique (phone_number),
    constraint customer_email_unique unique (email),
    check (zipcode > 0 and char_length(cast(zipcode as char)) = 5)
);
create index idx_customers_city on customers(city_id);
create index idx_customers_email on customers(email);
```

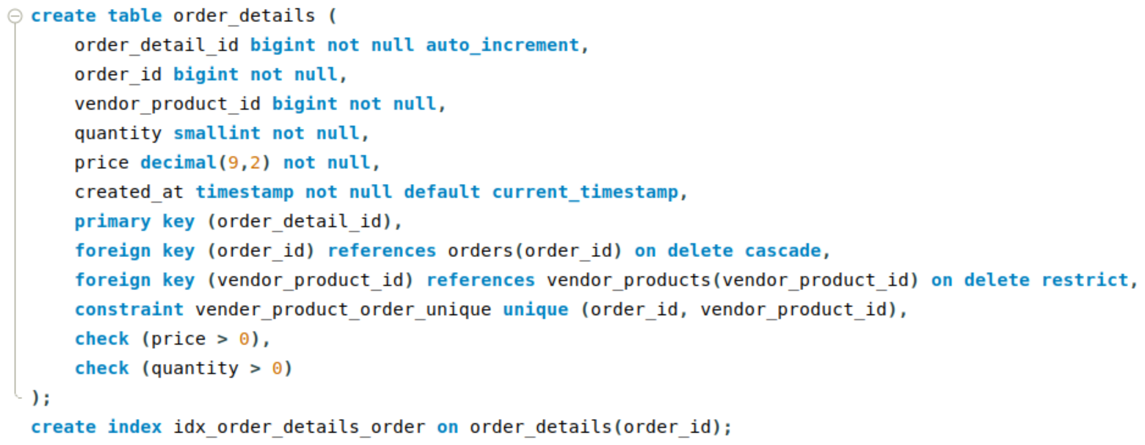
Figure 7: Create Customers Table

The “check” command which is used in some tables is responsible for controlling some situations to go correctly. For example customers table checks if the “zipcode” is greater than zero and has exactly 5 digits.

The most important tables of the marketplace are orders and order_details which stores all data about orders placed by customers (Figure 8). The foreign keys are shown in the SQL script but there is a difference which is “on delete cascade” for “order_id” in the order_details table. It means that if an order is deleted in the orders tables, delete all of its details from order_details tables which assures not to have order details without valid reference. Orders have an index on “customer_id” to find all orders of a customer more efficiently, and order_details has an index on “order_id” to retrieve items of an order faster.



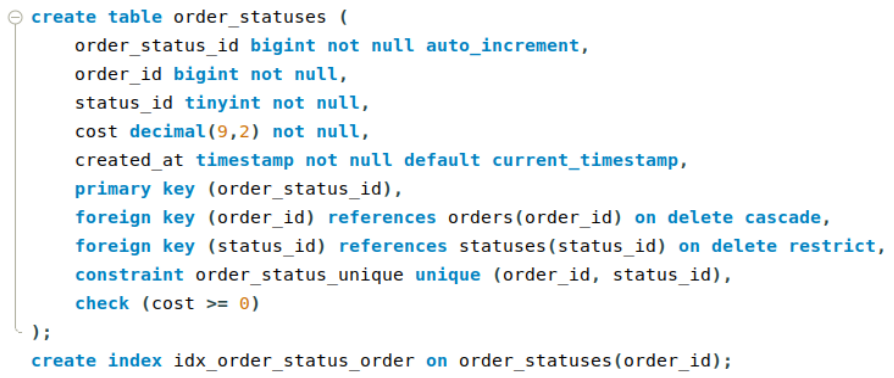
```sql
create table orders (
    order_id bigint not null auto_increment,
    customer_id int not null,
    total_cost decimal(10,2) not null,
    created_at timestamp not null default current_timestamp,
    primary key (order_id),
    foreign key (customer_id) references customers(customer_id) on delete restrict,
    check (total_cost >= 0)
);
create index idx_orders_customer on orders(customer_id);
```



```sql
create table order_details (
    order_detail_id bigint not null auto_increment,
    order_id bigint not null,
    vendor_product_id bigint not null,
    quantity smallint not null,
    price decimal(9,2) not null,
    created_at timestamp not null default current_timestamp,
    primary key (order_detail_id),
    foreign key (order_id) references orders(order_id) on delete cascade,
    foreign key (vendor_product_id) references vendor_products(vendor_product_id) on delete restrict,
    constraint vender_product_order_unique unique (order_id, vendor_product_id),
    check (price > 0),
    check (quantity > 0)
);
create index idx_order_details_order on order_details(order_id);
```

Figure 8: Create Orders and Order_details Tables

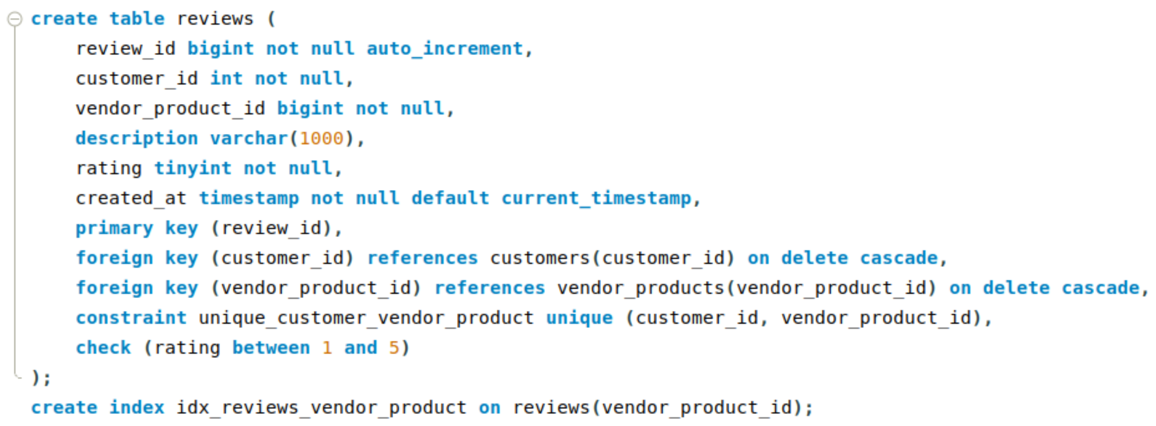
The order_statuses table is the other table that has a foreign key from the orders table with an “on delete cascade” constraint (Figure 9). It also has an index on “order_id” with the same reason as order_details.



```sql
create table order_statuses (
    order_status_id bigint not null auto_increment,
    order_id bigint not null,
    status_id tinyint not null,
    cost decimal(9,2) not null,
    created_at timestamp not null default current_timestamp,
    primary key (order_status_id),
    foreign key (order_id) references orders(order_id) on delete cascade,
    foreign key (status_id) references statuses(status_id) on delete restrict,
    constraint order_status_unique unique (order_id, status_id),
    check (cost >= 0)
);
create index idx_order_status_order on order_statuses(order_id);
```

Figure 9: Create Order_statuses Table

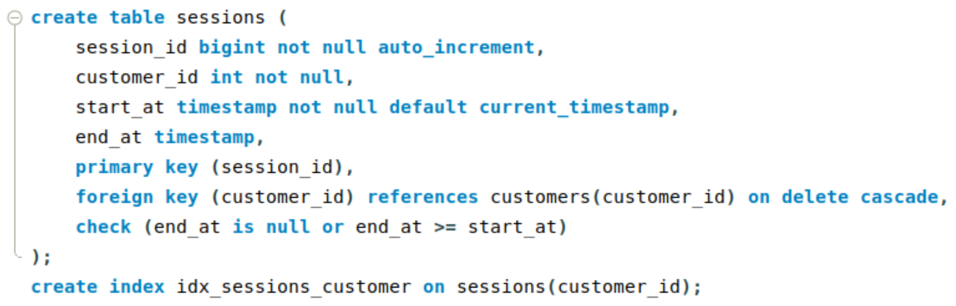
The reviews table has two foreign keys with "on delete cascade" which means if a customer or a product of a vendor is deleted from the database, all of its reviews will be deleted (Figure 10). It has an index on "vendor_product_id" which provides faster loading of the reviews for each product of a vendor.



```sql
create table reviews (
    review_id bigint not null auto_increment,
    customer_id int not null,
    vendor_product_id bigint not null,
    description varchar(1000),
    rating tinyint not null,
    created_at timestamp not null default current_timestamp,
    primary key (review_id),
    foreign key (customer_id) references customers(customer_id) on delete cascade,
    foreign key (vendor_product_id) references vendor_products(vendor_product_id) on delete cascade,
    constraint unique_customer_vendor_product unique (customer_id, vendor_product_id),
    check (rating between 1 and 5)
);
create index idx_reviews_vendor_product on reviews(vendor_product_id);
```

Figure 10: Create Reviews Table

The last table is sessions which has an index on "customer_id" for faster retrieval of the session of each customer.

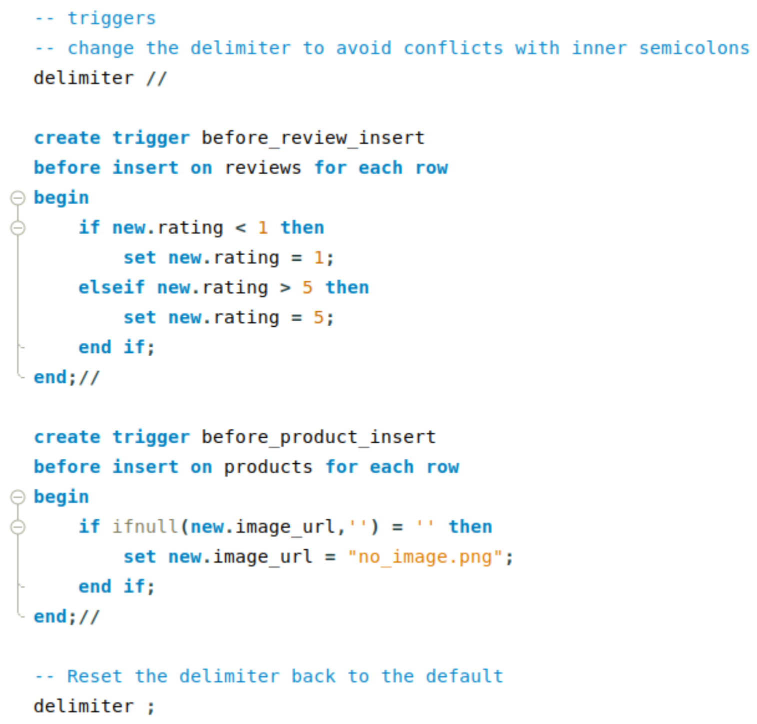


```sql
create table sessions (
    session_id bigint not null auto_increment,
    customer_id int not null,
    start_at timestamp not null default current_timestamp,
    end_at timestamp,
    primary key (session_id),
    foreign key (customer_id) references customers(customer_id) on delete cascade,
    check (end_at is null or end_at >= start_at)
);
create index idx_sessions_customer on sessions(customer_id);
```

Figure 11: Create Session Tables

In many tables, there are fields or combinations of fields that are unique in all records (Alternate Keys). These columns are defined by unique constraints.

The last part of database creation is maintaining triggers. This study gives two examples of triggers (Figure 12). The first one is responsible for passing the valid value for rating. If a rating outside of the range 1 and 5 is passed to the insert command, it changes it into 1 or 5. The second one acts in a way to fill image_url with a specific file name in case of empty or null value is used for it. The delimiter command is used to change the default delimiter from semicolon to // to avoid conflicts with the inner semicolons.



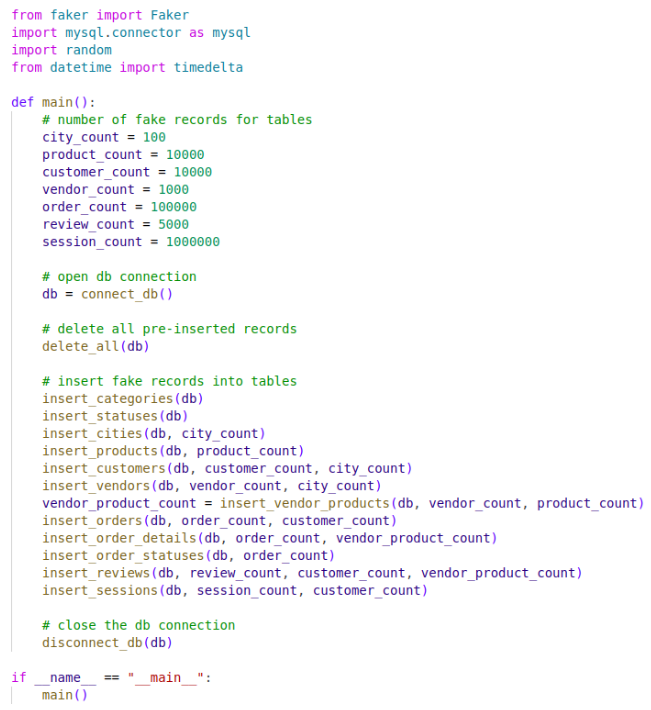
```sql
-- triggers
-- change the delimiter to avoid conflicts with inner semicolons
delimiter //

create trigger before_review_insert
before insert on reviews for each row
begin
    if new.rating < 1 then
        set new.rating = 1;
    elseif new.rating > 5 then
        set new.rating = 5;
    end if;
end;//

create trigger before_product_insert
before insert on products for each row
begin
    if ifnull(new.image_url,'') = '' then
        set new.image_url = "no_image.png";
    end if;
end;//

-- Reset the delimiter back to the default
delimiter ;
```

Figure 12: Create Triggers

3. DATA ANALYSIS WITH SQL

3.1. Populating The Database

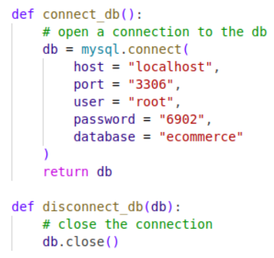
This study provides a comprehensive Python program to populate data in all tables. The program is written in a way to be usable for the desired number of records for each table. It mainly uses faker and random libraries to create realistic data (Figure 13).



```python
from faker import Faker
import mysql.connector as mysql
import random
from datetime import timedelta

def main():
    # number of fake records for tables
    city_count = 100
    product_count = 10000
    customer_count = 10000
    vendor_count = 1000
    order_count = 100000
    review_count = 5000
    session_count = 1000000

    # open db connection
    db = connect_db()

    # delete all pre-inserted records
    delete_all(db)

    # insert fake records into tables
    insert_categories(db)
    insert_statuses(db)
    insert_cities(db, city_count)
    insert_products(db, product_count)
    insert_customers(db, customer_count, city_count)
    insert_vendors(db, vendor_count, city_count)
    vendor_product_count = insert_vendor_products(db, vendor_count, product_count)
    insert_orders(db, order_count, customer_count)
    insert_order_details(db, order_count, vendor_product_count)
    insert_order_statuses(db, order_count)
    insert_reviews(db, review_count, customer_count, vendor_product_count)
    insert_sessions(db, session_count, customer_count)

    # close the db connection
    disconnect_db(db)

if __name__ == "__main__":
    main()
```

Figure 13: Main Function For Data Population

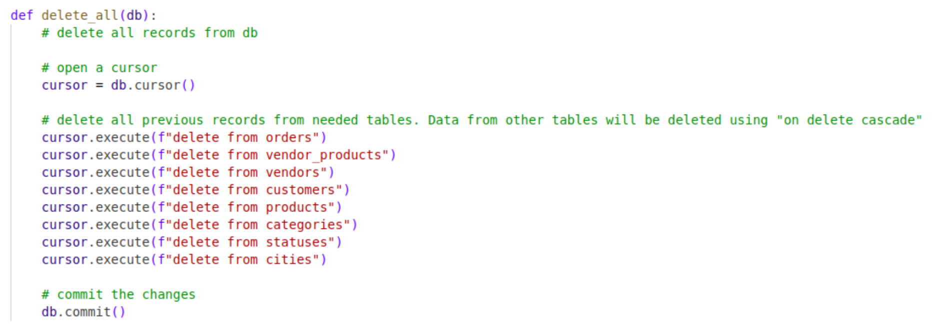
There are two functions to connect and disconnect from the database (Figure 14). They use mysql.connector library for this.



```python
def connect_db():
    # open a connection to the db
    db = mysql.connect(
        host = "localhost",
        port = "3306",
        user = "root",
        password = "6902",
        database = "ecommerce"
    )
    return db

def disconnect_db(db):
    # close the connection
    db.close()
```

Figure 14: Connect and Disconnect Database

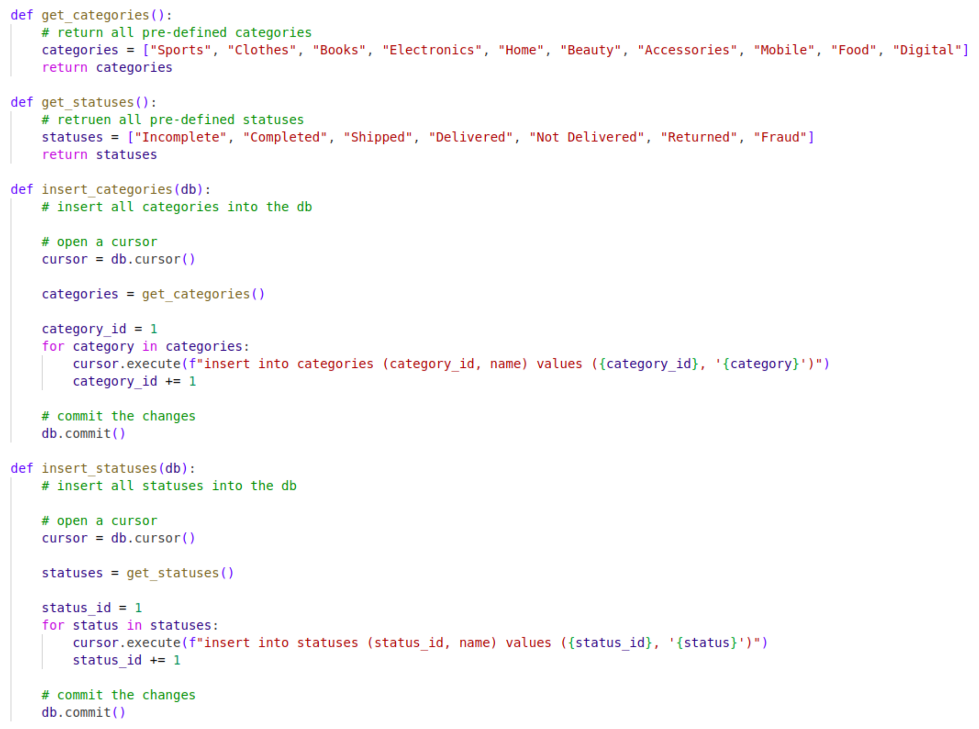
Before inserting data into the database, we need to delete all old data from it (Figure 15). Some tables that are not mentioned in the delete_all function will be deleted using "on delete cascade" property of the foreign key.



```python
def delete_all(db):
    # delete all records from db

    # open a cursor
    cursor = db.cursor()

    # delete all previous records from needed tables. Data from other tables will be deleted using "on delete cascade"
    cursor.execute(f"delete from orders")
    cursor.execute(f"delete from vendor_products")
    cursor.execute(f"delete from vendors")
    cursor.execute(f"delete from customers")
    cursor.execute(f"delete from products")
    cursor.execute(f"delete from categories")
    cursor.execute(f"delete from statuses")
    cursor.execute(f"delete from cities")

    # commit the changes
    db.commit()
```

Figure 15: Delete All Old Data

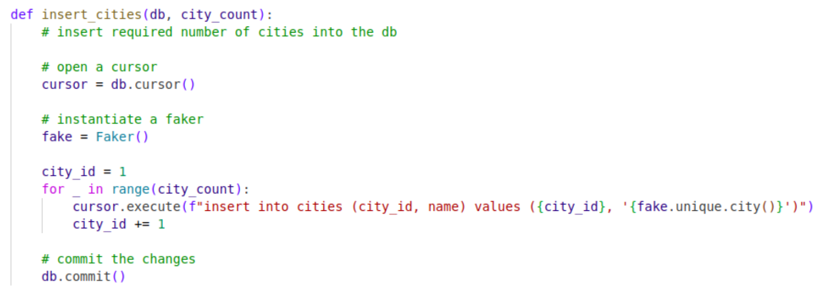
The categories and statuses need to have meaningful and pre-defined values. After that, they are inserted into the database (Figure 16).



```python
def get_categories():
    # return all pre-defined categories
    categories = ["Sports", "Clothes", "Books", "Electronics", "Home", "Beauty", "Accessories", "Mobile", "Food", "Digital"]
    return categories

def get_statuses():
    # retruen all pre-defined statuses
    statuses = ["Incomplete", "Completed", "Shipped", "Delivered", "Not Delivered", "Returned", "Fraud"]
    return statuses

def insert_categories(db):
    # insert all categories into the db

    # open a cursor
    cursor = db.cursor()

    categories = get_categories()

    category_id = 1
    for category in categories:
        cursor.execute(f"insert into categories (category_id, name) values ({category_id}, '{category}')")
        category_id += 1

    # commit the changes
    db.commit()

def insert_statuses(db):
    # insert all statuses into the db

    # open a cursor
    cursor = db.cursor()

    statuses = get_statuses()

    status_id = 1
    for status in statuses:
        cursor.execute(f"insert into statuses (status_id, name) values ({status_id}, '{status}')")
        status_id += 1

    # commit the changes
    db.commit()
```

Figure 16: Categories and Statuses Insertion

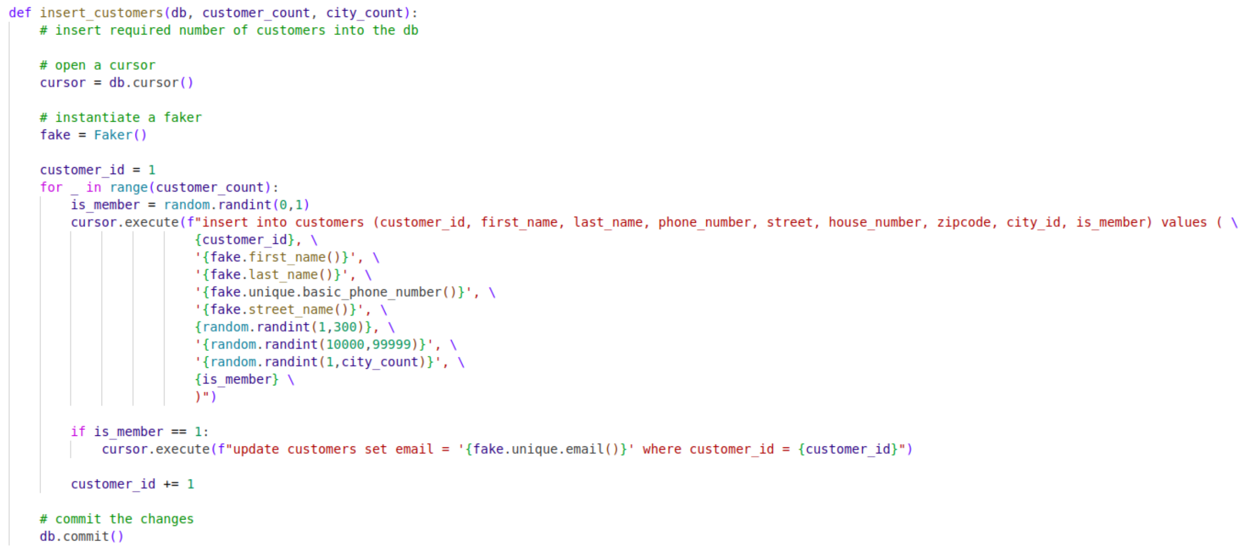
The next table needs to fill is cities. 100 records with realistic city names are populated there (Figure 17).



```python
def insert_cities(db, city_count):
    # insert required number of cities into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    city_id = 1
    for _ in range(city_count):
        cursor.execute(f"insert into cities (city_id, name) values ({city_id}, '{fake.unique.city()}')")
        city_id += 1

    # commit the changes
    db.commit()
```

Figure 17: Cities Insertion

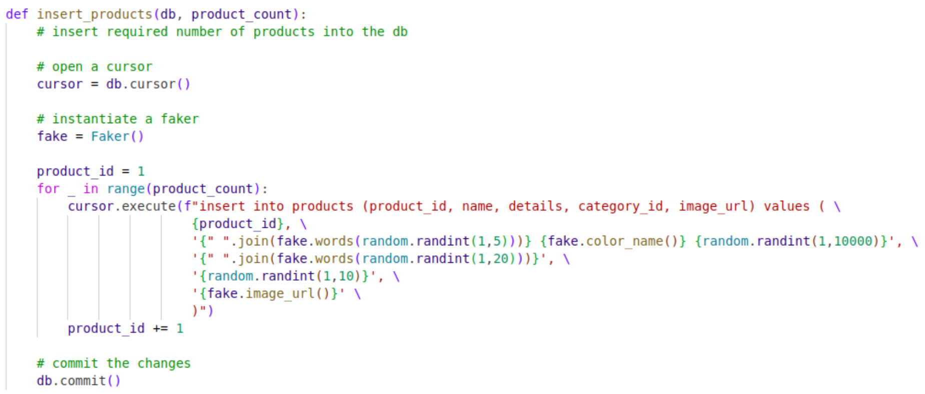
10000 customers are populated in the customers table using the insert_customers function (Figure 18).



```python
def insert_customers(db, customer_count, city_count):
    # insert required number of customers into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    customer_id = 1
    for _ in range(customer_count):
        is_member = random.randint(0,1)
        cursor.execute(f"insert into customers (customer_id, first_name, last_name, phone_number, street, house_number, zipcode, city_id, is_member) values ( \
                                {customer_id}, \
                                '{fake.first_name()}', \
                                '{fake.last_name()}', \
                                '{fake.unique.basic_phone_number()}', \
                                '{fake.street_name()}', \
                                {random.randint(1,300)}, \
                                '{random.randint(10000,99999)}', \
                                '{random.randint(1,city_count)}', \
                                {is_member} \
                                )")

        if is_member == 1:
            cursor.execute(f"update customers set email = '{fake.unique.email()}' where customer_id = {customer_id}")

        customer_id += 1

    # commit the changes
    db.commit()
```

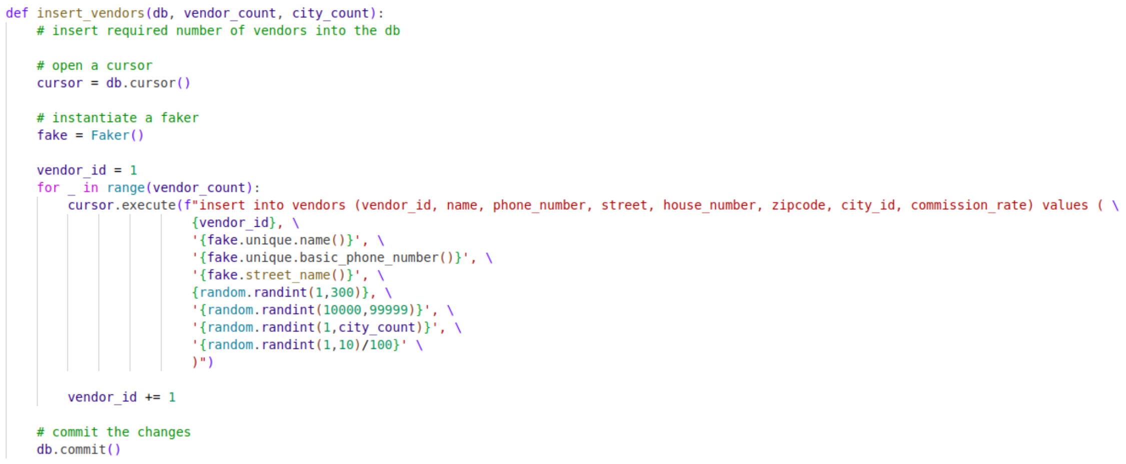
Figure 18: Customers Insertion

The next step is to fill products (Figure 19), vendors (Figure 20), and vendor_products (Figure 21) tables. There are 10000 products and 1000 vendors. The combination of their relationship is formed randomly in the vendor_products table. All these functions use "unique" for single-word names, and specific combinations of words to create unique values for multi-word names.



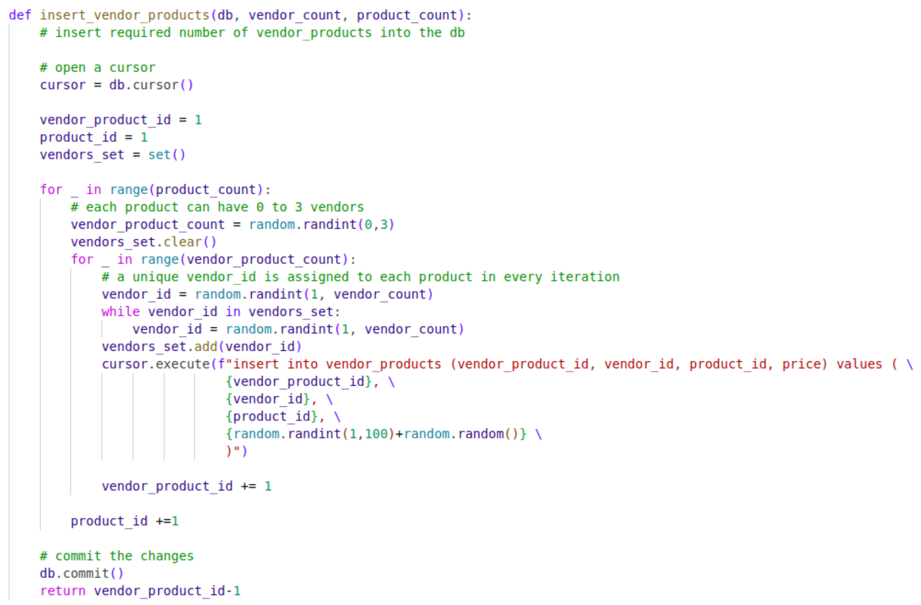
```python
def insert_products(db, product_count):
    # insert required number of products into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    product_id = 1
    for _ in range(product_count):
        cursor.execute(f"insert into products (product_id, name, details, category_id, image_url) values ( \
                        {product_id}, \
                        '{" ".join(fake.words(random.randint(1,5)))} {fake.color_name()} {random.randint(1,10000)}', \
                        '{" ".join(fake.words(random.randint(1,20)))}', \
                        '{random.randint(1,10)}', \
                        '{fake.image_url()}' \
                        )")
        product_id += 1

    # commit the changes
    db.commit()
```

Figure 19: Products Insertion



```python
def insert_vendors(db, vendor_count, city_count):
    # insert required number of vendors into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    vendor_id = 1
    for _ in range(vendor_count):
        cursor.execute(f"insert into vendors (vendor_id, name, phone_number, street, house_number, zipcode, city_id, commission_rate) values ( \
                        {vendor_id}, \
                        '{fake.unique.name()}', \
                        '{fake.unique.basic_phone_number()}', \
                        '{fake.street_name()}', \
                        {random.randint(1,300)}, \
                        '{random.randint(10000,99999)}', \
                        '{random.randint(1,city_count)}', \
                        '{random.randint(1,10)/100}' \
                        )")

        vendor_id += 1

    # commit the changes
    db.commit()
```

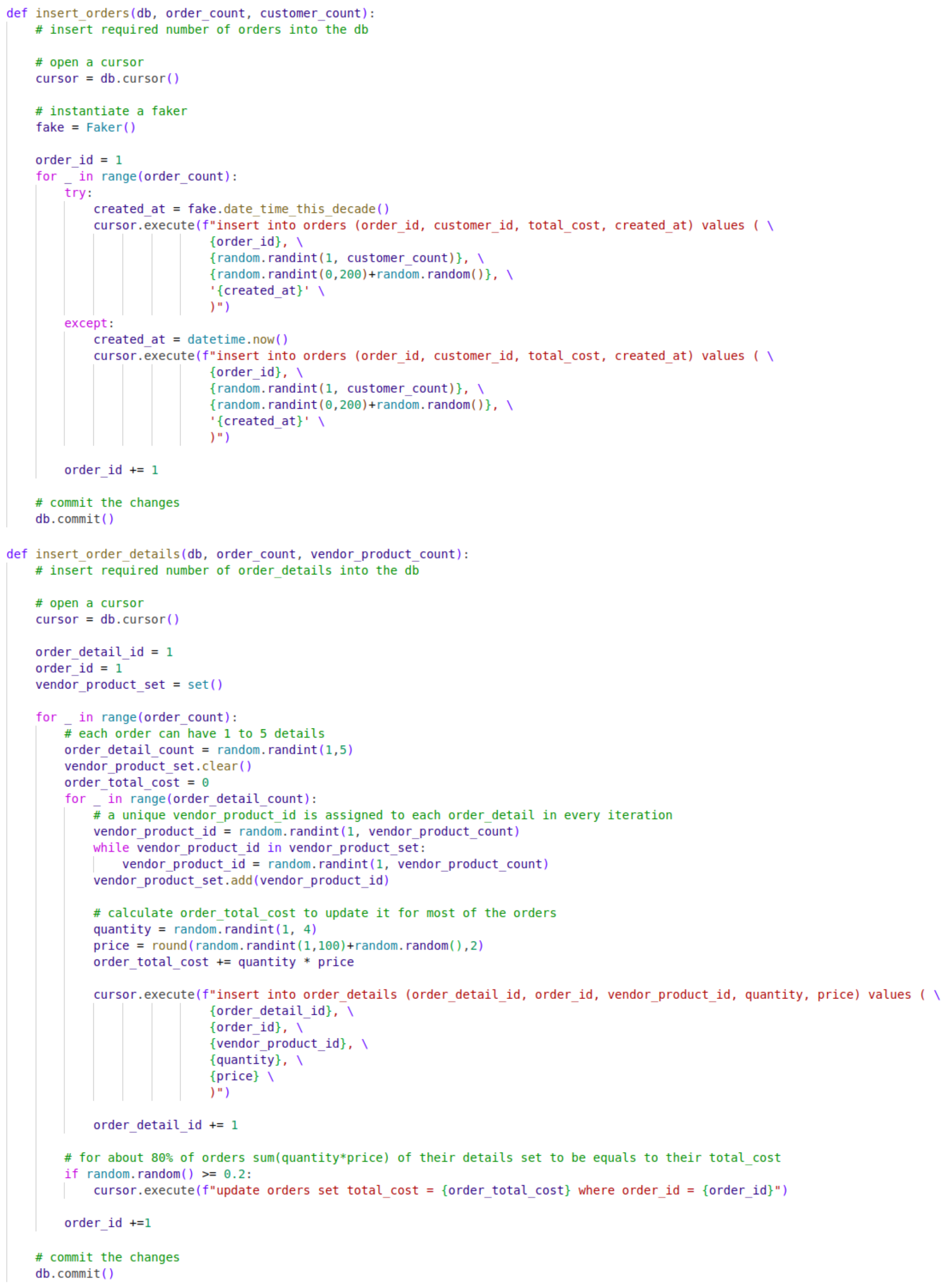
Figure 20: Vendors Insertion



```python
def insert_vendor_products(db, vendor_count, product_count):
    # insert required number of vendor_products into the db

    # open a cursor
    cursor = db.cursor()

    vendor_product_id = 1
    product_id = 1
    vendors_set = set()

    for _ in range(product_count):
        # each product can have 0 to 3 vendors
        vendor_product_count = random.randint(0,3)
        vendors_set.clear()
        for _ in range(vendor_product_count):
            # a unique vendor_id is assigned to each product in every iteration
            vendor_id = random.randint(1, vendor_count)
            while vendor_id in vendors_set:
                vendor_id = random.randint(1, vendor_count)
            vendors_set.add(vendor_id)
            cursor.execute(f"insert into vendor_products (vendor_product_id, vendor_id, product_id, price) values ( \
                            {vendor_product_id}, \
                            {vendor_id}, \
                            {product_id}, \
                            {random.randint(1,100)+random.random()} \
                            )")

            vendor_product_id += 1

        product_id +=1

    # commit the changes
    db.commit()
    return vendor_product_id-1
```

Figure 21: Vendor_Products Insertion

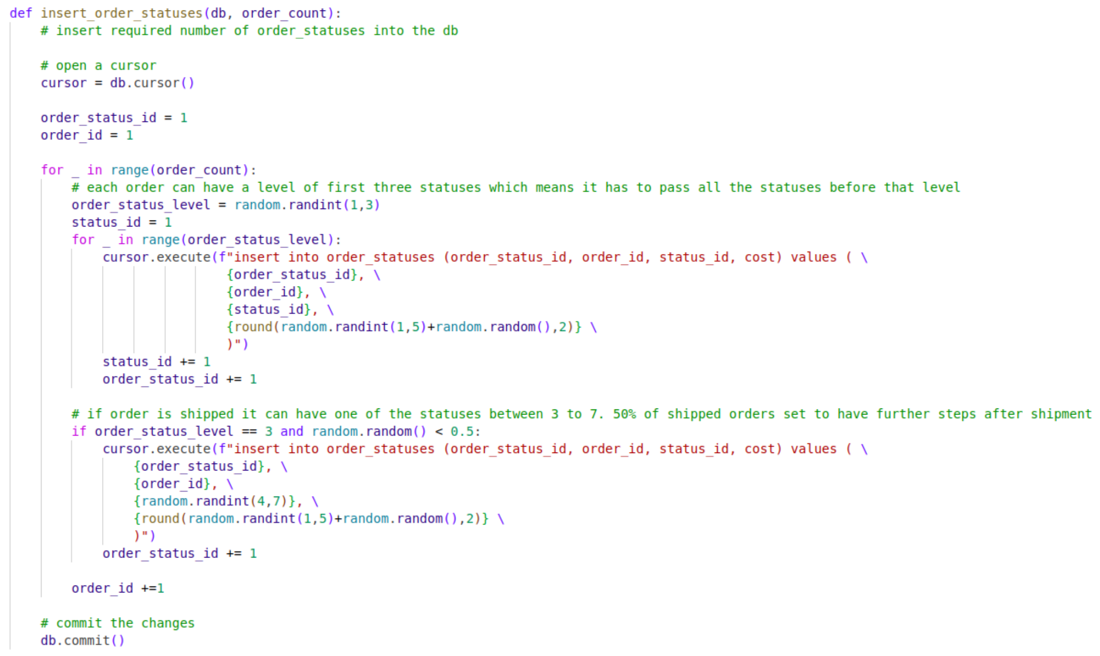
The most important tables of this schema are orders and order_details. There are 100000 orders populated in a way that total_cost of the orders in about 20% of cases is different from the multiplication of quantity and price for their details (Figure 22).



```python
def insert_orders(db, order_count, customer_count):
    # insert required number of orders into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    order_id = 1
    for _ in range(order_count):
        try:
            created_at = fake.date_time_this_decade()
            cursor.execute(f"insert into orders (order_id, customer_id, total_cost, created_at) values ( \
                                {order_id}, \
                                {random.randint(1, customer_count)}, \
                                {random.randint(0,200)+random.random()}, \
                                '{created_at}' \
                                )")
        except:
            created_at = datetime.now()
            cursor.execute(f"insert into orders (order_id, customer_id, total_cost, created_at) values ( \
                                {order_id}, \
                                {random.randint(1, customer_count)}, \
                                {random.randint(0,200)+random.random()}, \
                                '{created_at}' \
                                )")

        order_id += 1

    # commit the changes
    db.commit()

def insert_order_details(db, order_count, vendor_product_count):
    # insert required number of order_details into the db

    # open a cursor
    cursor = db.cursor()

    order_detail_id = 1
    order_id = 1
    vendor_product_set = set()

    for _ in range(order_count):
        # each order can have 1 to 5 details
        order_detail_count = random.randint(1,5)
        vendor_product_set.clear()
        order_total_cost = 0
        for _ in range(order_detail_count):
            # a unique vendor_product_id is assigned to each order_detail in every iteration
            vendor_product_id = random.randint(1, vendor_product_count)
            while vendor_product_id in vendor_product_set:
                vendor_product_id = random.randint(1, vendor_product_count)
            vendor_product_set.add(vendor_product_id)

            # calculate order_total_cost to update it for most of the orders
            quantity = random.randint(1, 4)
            price = round(random.randint(1,100)+random.random(),2)
            order_total_cost += quantity * price

            cursor.execute(f"insert into order_details (order_detail_id, order_id, vendor_product_id, quantity, price) values ( \
                                {order_detail_id}, \
                                {order_id}, \
                                {vendor_product_id}, \
                                {quantity}, \
                                {price} \
                                )")

            order_detail_id += 1

        # for about 80% of orders sum(quantity*price) of their details set to be equals to their total_cost
        if random.random() >= 0.2:
            cursor.execute(f"update orders set total_cost = {order_total_cost} where order_id = {order_id}")

        order_id +=1

    # commit the changes
    db.commit()
```

Figure 22: Orders And Order_Details Insertion

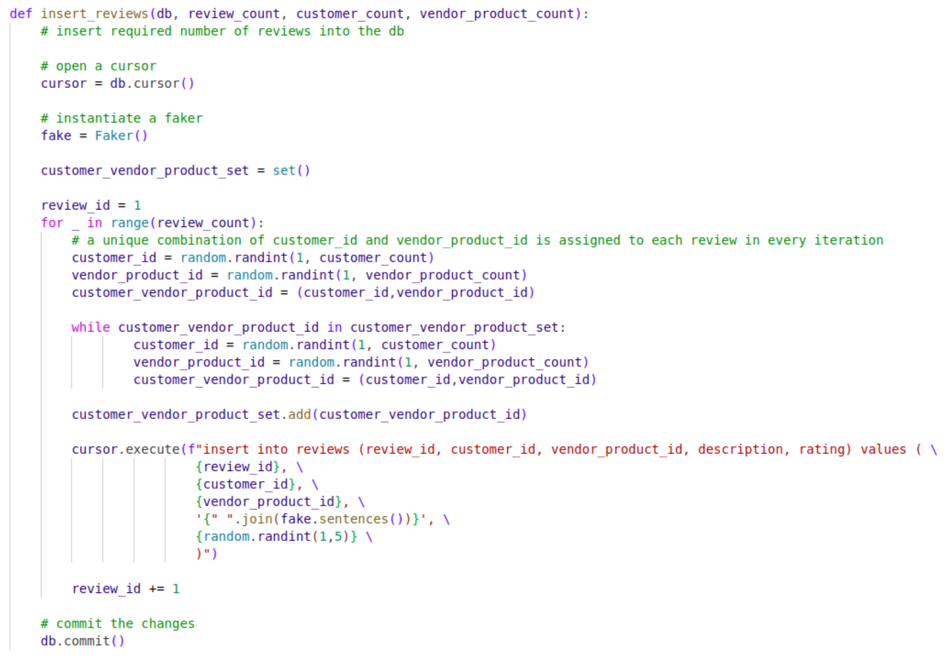
The other table to populate is order_statuses (Figure 23). The insertion is designed to create order_statuses randomly among the first three statuses ("Incomplete", "Completed", "Shipped"). 50% of the orders which reach shipped status will randomly go further with the other four possible statuses ("Delivered", "Not Delivered", "Returned", "Fraud").



```python
def insert_order_statuses(db, order_count):
    # insert required number of order_statuses into the db

    # open a cursor
    cursor = db.cursor()

    order_status_id = 1
    order_id = 1

    for _ in range(order_count):
        # each order can have a level of first three statuses which means it has to pass all the statuses before that level
        order_status_level = random.randint(1,3)
        status_id = 1
        for _ in range(order_status_level):
            cursor.execute(f"insert into order_statuses (order_status_id, order_id, status_id, cost) values ( \
                            {order_status_id}, \
                            {order_id}, \
                            {status_id}, \
                            {round(random.randint(1,5)+random.random(),2)} \
                            )")
            status_id += 1
            order_status_id += 1

        # if order is shipped it can have one of the statuses between 3 to 7. 50% of shipped orders set to have further steps after shipment
        if order_status_level == 3 and random.random() < 0.5:
            cursor.execute(f"insert into order_statuses (order_status_id, order_id, status_id, cost) values ( \
                {order_status_id}, \
                {order_id}, \
                {random.randint(4,7)}, \
                {round(random.randint(1,5)+random.random(),2)} \
                )")
            order_status_id += 1

        order_id +=1

    # commit the changes
    db.commit()
```

Figure 23: Order_Statuses Insertion

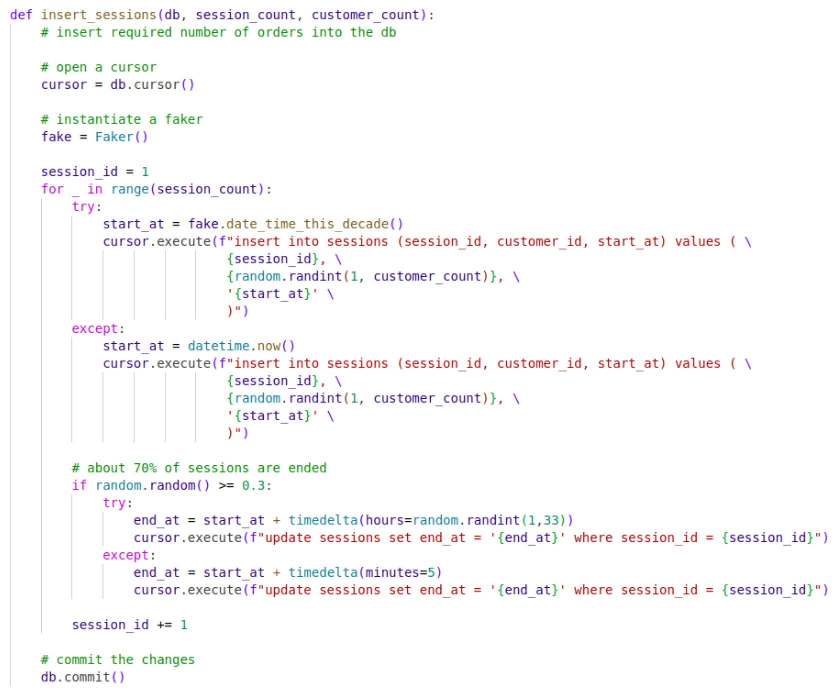
The next table is reviews which is filled with 5000 records from different customers for different products of the vendors (Figure 24).



```python
def insert_reviews(db, review_count, customer_count, vendor_product_count):
    # insert required number of reviews into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    customer_vendor_product_set = set()

    review_id = 1
    for _ in range(review_count):
        # a unique combination of customer_id and vendor_product_id is assigned to each review in every iteration
        customer_id = random.randint(1, customer_count)
        vendor_product_id = random.randint(1, vendor_product_count)
        customer_vendor_product_id = (customer_id,vendor_product_id)

        while customer_vendor_product_id in customer_vendor_product_set:
                customer_id = random.randint(1, customer_count)
                vendor_product_id = random.randint(1, vendor_product_count)
                customer_vendor_product_id = (customer_id,vendor_product_id)

        customer_vendor_product_set.add(customer_vendor_product_id)

        cursor.execute(f"insert into reviews (review_id, customer_id, vendor_product_id, description, rating) values ( \
                        {review_id}, \
                        {customer_id}, \
                        {vendor_product_id}, \
                        '{" ".join(fake.sentences())}', \
                        {random.randint(1,5)} \
                        )")

        review_id += 1

    # commit the changes
    db.commit()
```

Figure 24: Reviews Insertion

The last table is sessions which contains 1000000 sessions (Figure 25). About 70% of the sessions are closed and others remain open yet.



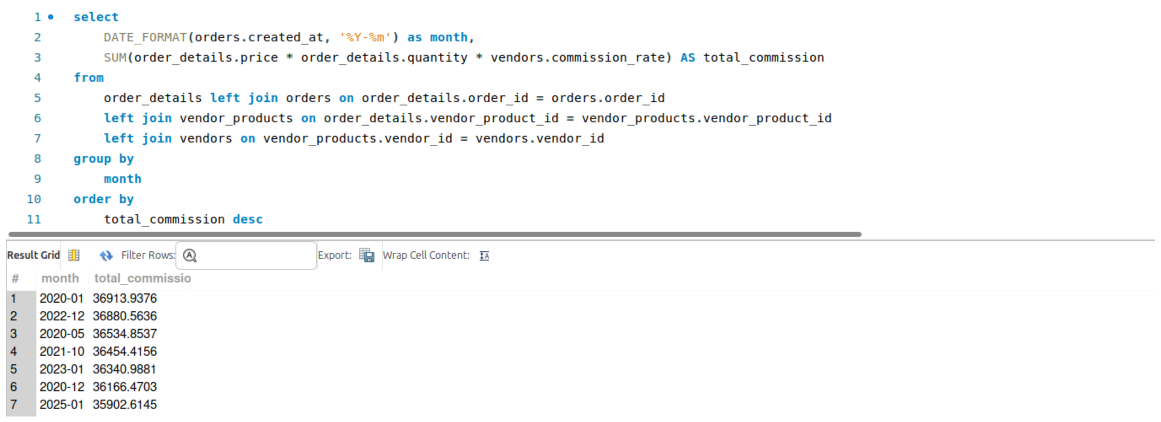
```
def insert_sessions(db, session_count, customer_count):
    # insert required number of orders into the db

    # open a cursor
    cursor = db.cursor()

    # instantiate a faker
    fake = Faker()

    session_id = 1
    for _ in range(session_count):
        try:
            start_at = fake.date_time_this_decade()
            cursor.execute(f"insert into sessions (session_id, customer_id, start_at) values ( \
                            {session_id}, \
                            {random.randint(1, customer_count)}, \
                            '{start_at}' \
                            )")
        except:
            start_at = datetime.now()
            cursor.execute(f"insert into sessions (session_id, customer_id, start_at) values ( \
                            {session_id}, \
                            {random.randint(1, customer_count)}, \
                            '{start_at}' \
                            )")

        # about 70% of sessions are ended
        if random.random() >= 0.3:
            try:
                end_at = start_at + timedelta(hours=random.randint(1,33))
                cursor.execute(f"update sessions set end_at = '{end_at}' where session_id = {session_id}")
            except:
                end_at = start_at + timedelta(minutes=5)
                cursor.execute(f"update sessions set end_at = '{end_at}' where session_id = {session_id}")

        session_id += 1

    # commit the changes
    db.commit()
```

Figure 25: Sessions Insertion

3.2. SQL Queries To Address Objectives

There are some SQL queries in this section to provide insight related to objectives declared in section 2.1. The objectives and their queries are discussed below.

- Vendor Analysis:
  The first query is about monthly revenue based on commissions to know which months have the highest commission revenue among all (Figure 26).

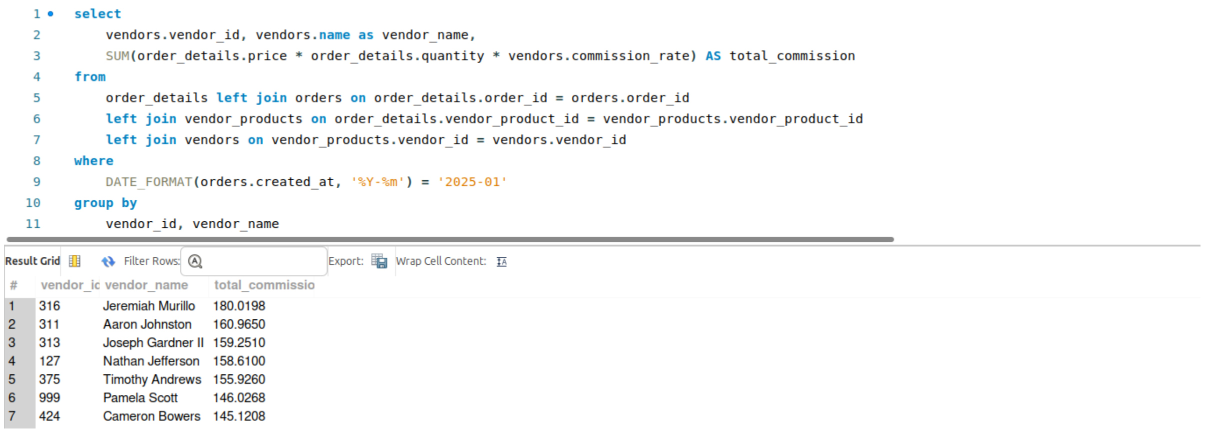


```
select
    DATE_FORMAT(orders.created_at, '%Y-%m') as month,
    SUM(order_details.price * order_details.quantity * vendors.commission_rate) AS total_commission
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join vendors on vendor_products.vendor_id = vendors.vendor_id
group by
    month
order by
    total_commission desc
```

```
#  month    total_commissio
1  2020-01  36913.9376
2  2022-12  36880.5636
3  2020-05  36534.8537
4  2021-10  36454.4156
5  2023-01  36340.9881
6  2020-12  36166.4703
7  2025-01  35902.6145
```

Figure 26: Total Commission Per Month

The next query shows the commission revenue of a specific month for each vendor (Figure 27).



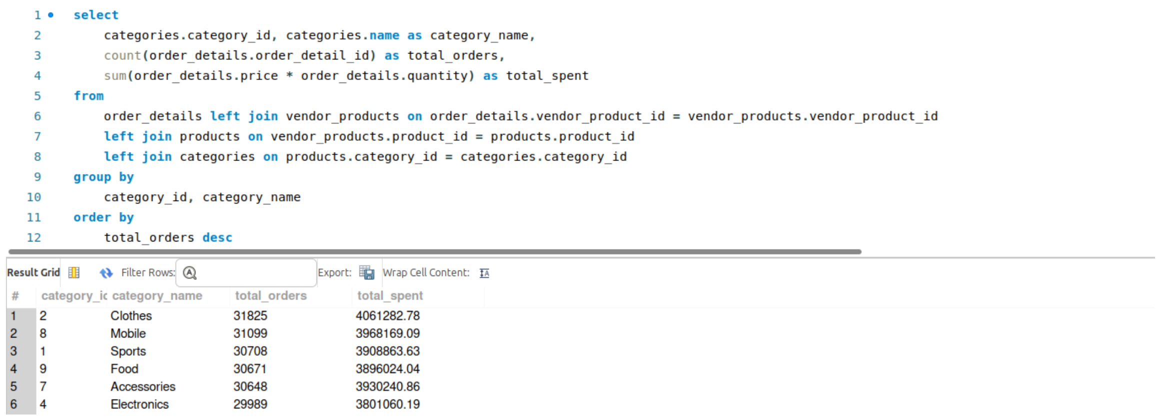
```
select
    vendors.vendor_id, vendors.name as vendor_name,
    SUM(order_details.price * order_details.quantity * vendors.commission_rate) AS total_commission
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join vendors on vendor_products.vendor_id = vendors.vendor_id
where
    DATE_FORMAT(orders.created_at, '%Y-%m') = '2025-01'
group by
    vendor_id, vendor_name
```

```
#  vendor_ic vendor_name       total_commissio
1  316       Jeremiah Murillo  180.0198
2  311       Aaron Johnston    160.9650
3  313       Joseph Gardner II 159.2510
4  127       Nathan Jefferson  158.6100
5  375       Timothy Andrews   155.9260
6  999       Pamela Scott      146.0268
7  424       Cameron Bowers    145.1208
```

Figure 27: Total Commission Per Vendor For A Specific Month

- Customer Analysis:

  In this analysis, the first query states the number of orders and the total spent amount on them based on the product categories. It specifies the most favorite categories (Figure 28).

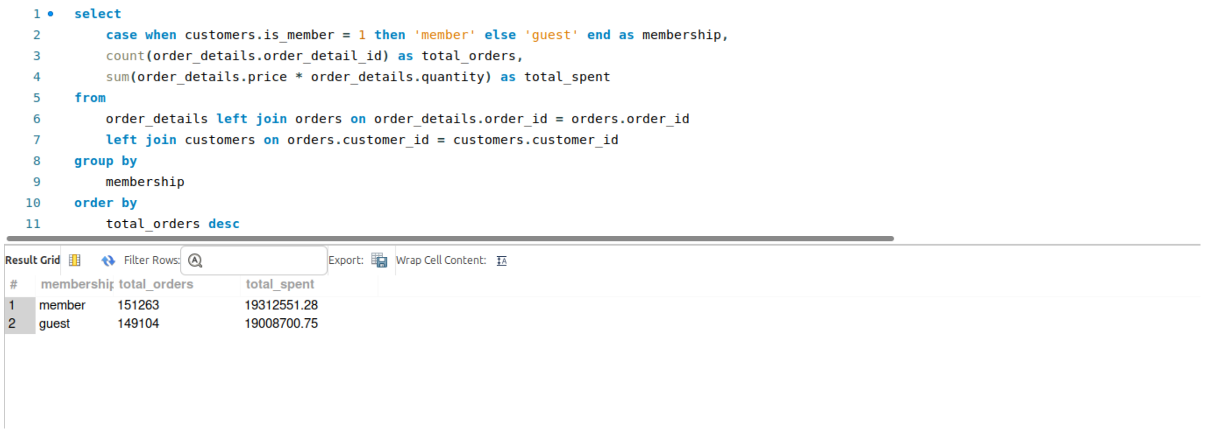


```
select
    categories.category_id, categories.name as category_name,
    count(order_details.order_detail_id) as total_orders,
    sum(order_details.price * order_details.quantity) as total_spent
from
    order_details left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join products on vendor_products.product_id = products.product_id
    left join categories on products.category_id = categories.category_id
group by
    category_id, category_name
order by
    total_orders desc
```

#	category_ic	category_name	total_orders	total_spent
1	2	Clothes	31825	4061282.78
2	8	Mobile	31099	3968169.09
3	1	Sports	30708	3908863.63
4	9	Food	30671	3896024.04
5	7	Accessories	30648	3930240.86
6	4	Electronics	29989	3801060.19

Figure 28: Total Order Per Category

The next one shows if members or guests have more purchases than each other (Figure 29). It depicts that there is no significant difference.



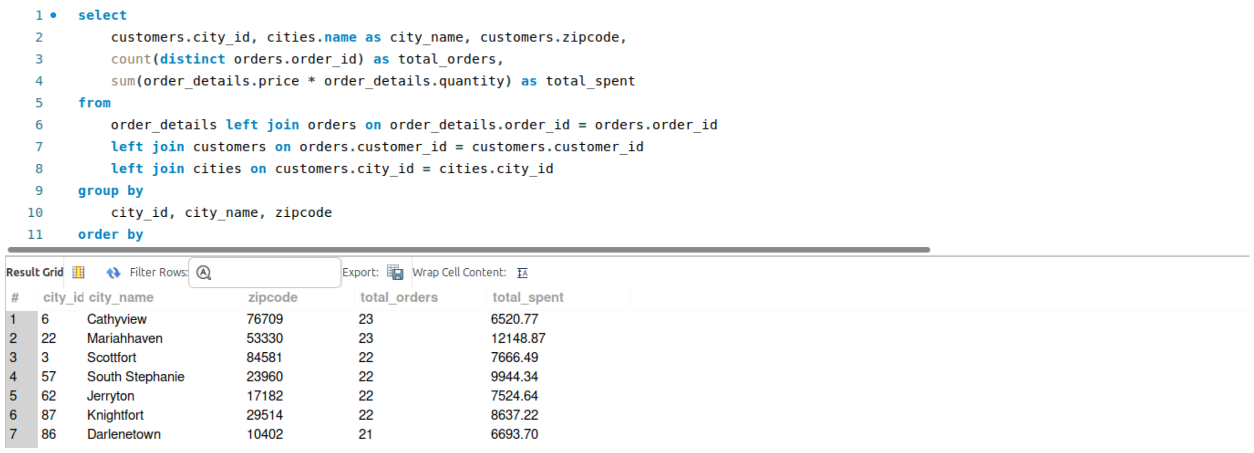
```
select
    case when customers.is_member = 1 then 'member' else 'guest' end as membership,
    count(order_details.order_detail_id) as total_orders,
    sum(order_details.price * order_details.quantity) as total_spent
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join customers on orders.customer_id = customers.customer_id
group by
    membership
order by
    total_orders desc
```

#	membershij	total_orders	total_spent
1	member	151263	19312551.28
2	guest	149104	19008700.75

Figure 29: Total Order Based on Membership

The next two queries show which locations (city and zip code) have the most number of orders and total spent according to customers and vendors respectively (Figure 30-31).

The other query is the intersection of the top 20 cities with the highest number of purchases among customers and the top 20 cities with the highest number of sales among the vendors (Figure 32). These cities can be the bast places to open support offices. Another benefit of these types of queries is identifying the best places for warehouses (e.g. if we analyze zip codes in a city).

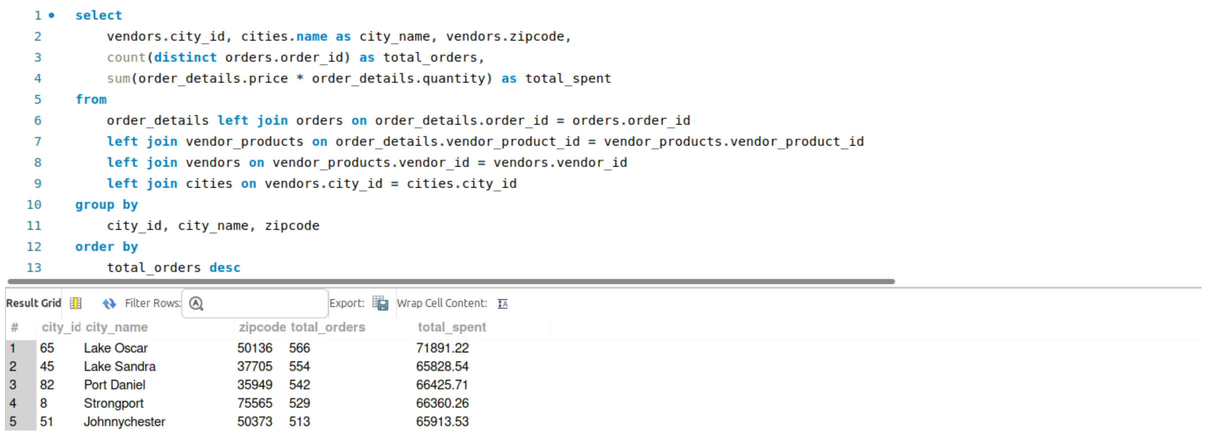


```
select
    customers.city_id, cities.name as city_name, customers.zipcode,
    count(distinct orders.order_id) as total_orders,
    sum(order_details.price * order_details.quantity) as total_spent
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join customers on orders.customer_id = customers.customer_id
    left join cities on customers.city_id = cities.city_id
group by
    city_id, city_name, zipcode
order by
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

#	city_id	city_name	zipcode	total_orders	total_spent
1	6	Cathyview	76709	23	6520.77
2	22	Mariahhaven	53330	23	12148.87
3	3	Scottfort	84581	22	7666.49
4	57	South Stephanie	23960	22	9944.34
5	62	Jerryton	17182	22	7524.64
6	87	Knightfort	29514	22	8637.22
7	86	Darlenetown	10402	21	6693.70

Figure 30: Total Order Per Customer Location

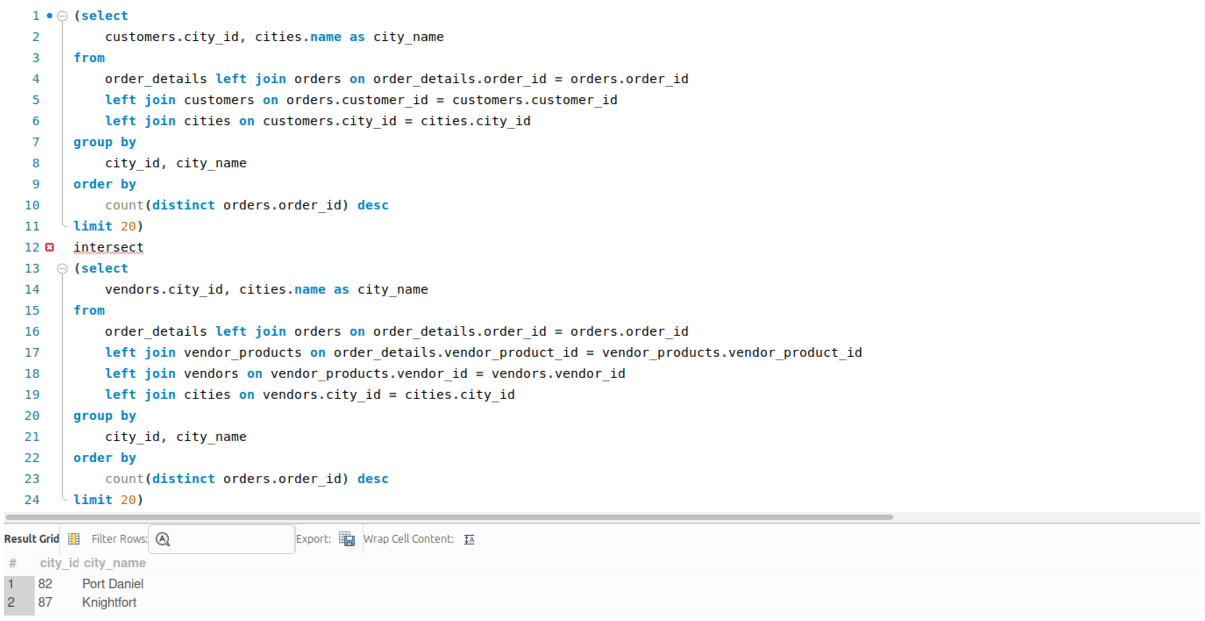


```
select
    vendors.city_id, cities.name as city_name, vendors.zipcode,
    count(distinct orders.order_id) as total_orders,
    sum(order_details.price * order_details.quantity) as total_spent
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join vendors on vendor_products.vendor_id = vendors.vendor_id
    left join cities on vendors.city_id = cities.city_id
group by
    city_id, city_name, zipcode
order by
    total_orders desc
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

#	city_id	city_name	zipcode	total_orders	total_spent
1	65	Lake Oscar	50136	566	71891.22
2	45	Lake Sandra	37705	554	65828.54
3	82	Port Daniel	35949	542	66425.71
4	8	Strongport	75565	529	66360.26
5	51	Johnnychester	50373	513	65913.53

Figure 31: Total Order Per Vendor Location



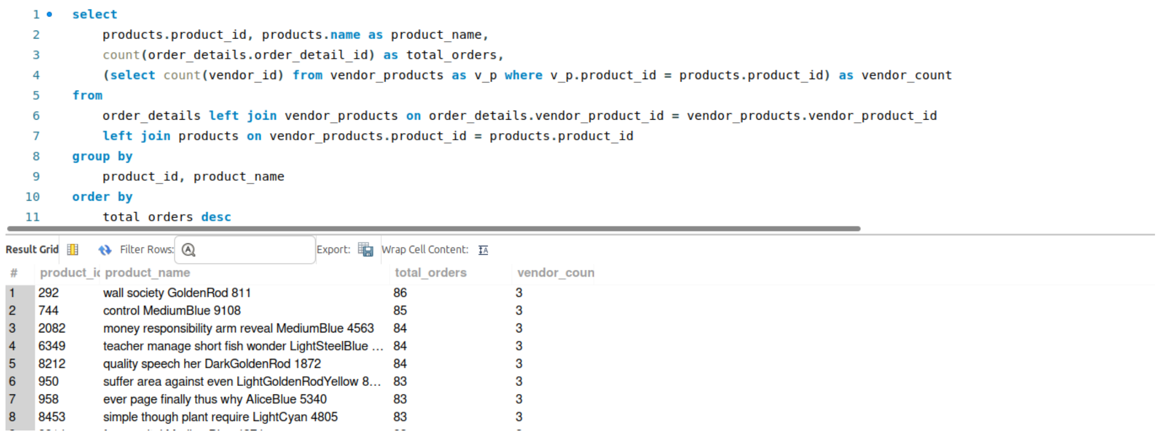
```sql
(select
    customers.city_id, cities.name as city_name
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join customers on orders.customer_id = customers.customer_id
    left join cities on customers.city_id = cities.city_id
group by
    city_id, city_name
order by
    count(distinct orders.order_id) desc
limit 20)
intersect
(select
    vendors.city_id, cities.name as city_name
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join vendors on vendor_products.vendor_id = vendors.vendor_id
    left join cities on vendors.city_id = cities.city_id
group by
    city_id, city_name
order by
    count(distinct orders.order_id) desc
limit 20)
```

```
#   city_id city_name
1   82      Port Daniel
2   87      Knightfort
```

Figure 32: The Favorite Cities Among Customers And Vendors

- Product Analysis:

  The next query analyzes the effect of vendors number for each product on their number of orders. It shows if a product has more vendors it will have a better chance of being purchased (Figure 33).

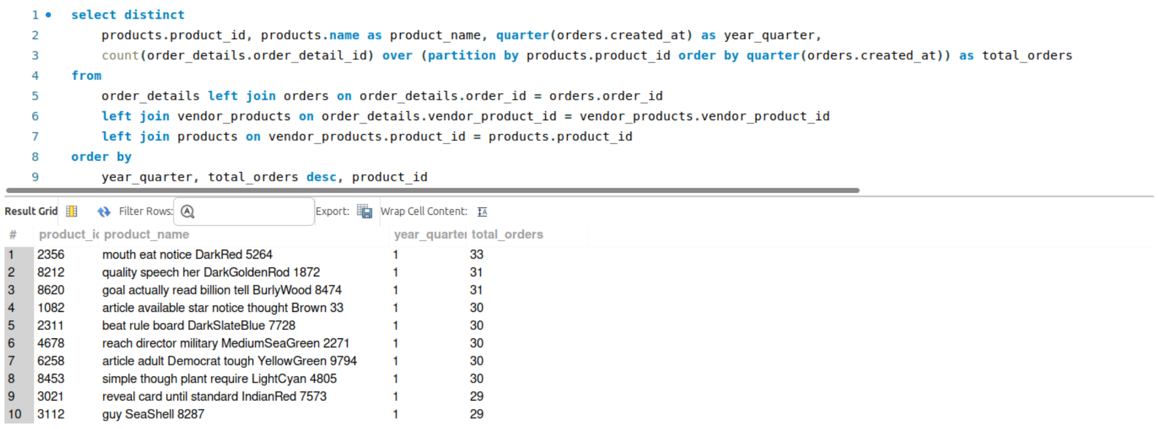


```sql
select
    products.product_id, products.name as product_name,
    count(order_details.order_detail_id) as total_orders,
    (select count(vendor_id) from vendor_products as v_p where v_p.product_id = products.product_id) as vendor_count
from
    order_details left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join products on vendor_products.product_id = products.product_id
group by
    product_id, product_name
order by
    total orders desc
```

```
#  product_ic product_name                                  total_orders  vendor_coun
1  292        wall society GoldenRod 811                    86            3
2  744        control MediumBlue 9108                       85            3
3  2082       money responsibility arm reveal MediumBlue 4563 84          3
4  6349       teacher manage short fish wonder LightSteelBlue ... 84      3
5  8212       quality speech her DarkGoldenRod 1872         84            3
6  950        suffer area against even LightGoldenRodYellow 8... 83       3
7  958        ever page finally thus why AliceBlue 5340     83            3
8  8453       simple though plant require LightCyan 4805    83            3
```

Figure 33: Relation Of Vendors Number and Purchase Chance

The last query shows the number of orders per product in each season (quarter). It uses a window function to extract this information. It can be useful to manage inventory better based on the prediction of purchases in each season of the year (Figure 34).



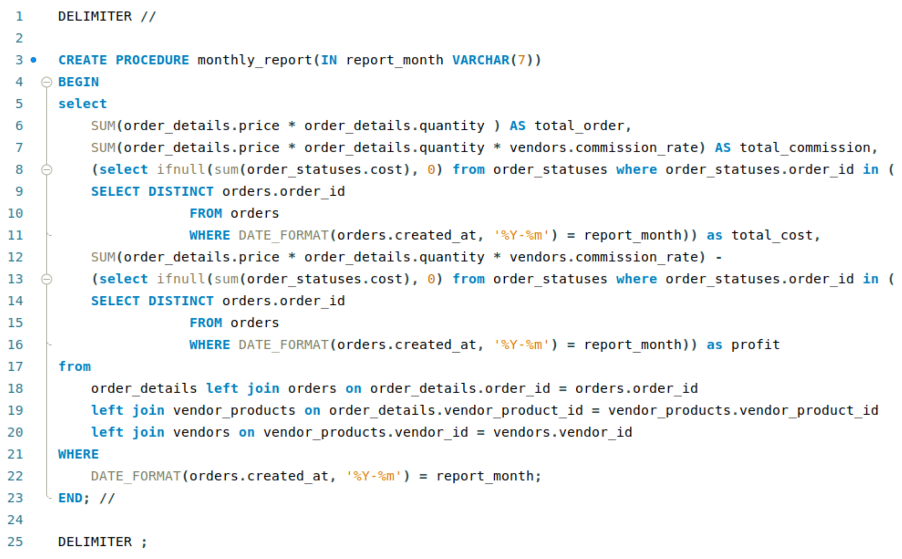
```sql
select distinct
    products.product_id, products.name as product_name, quarter(orders.created_at) as year_quarter,
    count(order_details.order_detail_id) over (partition by products.product_id order by quarter(orders.created_at)) as total_orders
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join products on vendor_products.product_id = products.product_id
order by
    year_quarter, total_orders desc, product_id
```

#	product_i	product_name	year_quarte	total_orders
1	2356	mouth eat notice DarkRed 5264	1	33
2	8212	quality speech her DarkGoldenRod 1872	1	31
3	8620	goal actually read billion tell BurlyWood 8474	1	31
4	1082	article available star notice thought Brown 33	1	30
5	2311	beat rule board DarkSlateBlue 7728	1	30
6	4678	reach director military MediumSeaGreen 2271	1	30
7	6258	article adult Democrat tough YellowGreen 9794	1	30
8	8453	simple though plant require LightCyan 4805	1	30
9	3021	reveal card until standard IndianRed 7573	1	29
10	3112	guy SeaShell 8287	1	29

Figure 34: Total Order Of Products In Each Quarter

3.3. Monthly Report Stored Procedure

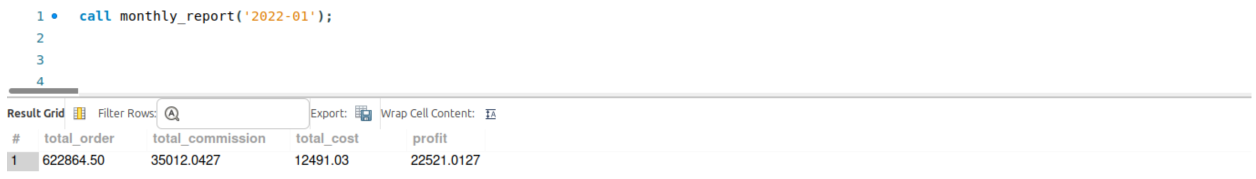
Monthly profit can be gained by calculating the total commission collected by the marketplace from orders of that month. The costs which are specified in the order_statuses table should be subtracted from the commission to reach the profit. The montly_report stored procedure is responsible for this (Figure 35).



```sql
DELIMITER //

CREATE PROCEDURE monthly_report(IN report_month VARCHAR(7))
BEGIN
select
    SUM(order_details.price * order_details.quantity ) AS total_order,
    SUM(order_details.price * order_details.quantity * vendors.commission_rate) AS total_commission,
    (select ifnull(sum(order_statuses.cost), 0) from order_statuses where order_statuses.order_id in (
    SELECT DISTINCT orders.order_id
                FROM orders
                WHERE DATE_FORMAT(orders.created_at, '%Y-%m') = report_month)) as total_cost,
    SUM(order_details.price * order_details.quantity * vendors.commission_rate) -
    (select ifnull(sum(order_statuses.cost), 0) from order_statuses where order_statuses.order_id in (
    SELECT DISTINCT orders.order_id
                FROM orders
                WHERE DATE_FORMAT(orders.created_at, '%Y-%m') = report_month)) as profit
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join vendors on vendor_products.vendor_id = vendors.vendor_id
WHERE
    DATE_FORMAT(orders.created_at, '%Y-%m') = report_month;
END; //

DELIMITER ;
```

Figure 35: Monthly Report Stored Procedure

To see the results for a month (e.g. 2022.01), we need to execute the stored procedure (Figure 36).


```
call monthly_report('2022-01');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

#	total_order	total_commission	total_cost	profit
1	622864.50	35012.0427	12491.03	22521.0127

Figure 36: Execute Monthly Report Stored Procedure

3.4. Proposal To Personalized Recommendation

We need to study customers' and vendors' behavior to offer personalized recommendations. There are some changes that can be applied to some entities.

1. Customers → Monthly Average Spend (New Attribute): It can calculated based on the orders placed by the customer and kept updated.
2. Products → Rating, Popularity (New Attributes): They can be calculated based on the average rating in the reviews and the number of orders containing that product respectively.
3. Vendor_Categories (New Entity): This entity assigns categories to the vendors. Every vendor can have several categories. Vendors can have different ratings in each category which can be calculated based on the average rating of their products of that category in the reviews.
4. Customer_Preferences (New Entity): This entity has attributes such as "customer_id", "category_id", and "relevance_rate". We can populate relevance_rate in this entity based on the number of orders the customers purchased from each product category divided by the total number of their orders. Based on this metric, we can recommend products purchased by a customer to other customers with similar preferences.

After applying changes related to personalized recommendations the Chen ER Diagram will have more attributes (Figure 37). Since the Chen ER Diagram is designed before normalization, all the changes are applied in that way. That is the reason we have attributes for vendor_categories (vendors → category) and customer_preferences (customer → preferences) instead of new entities.

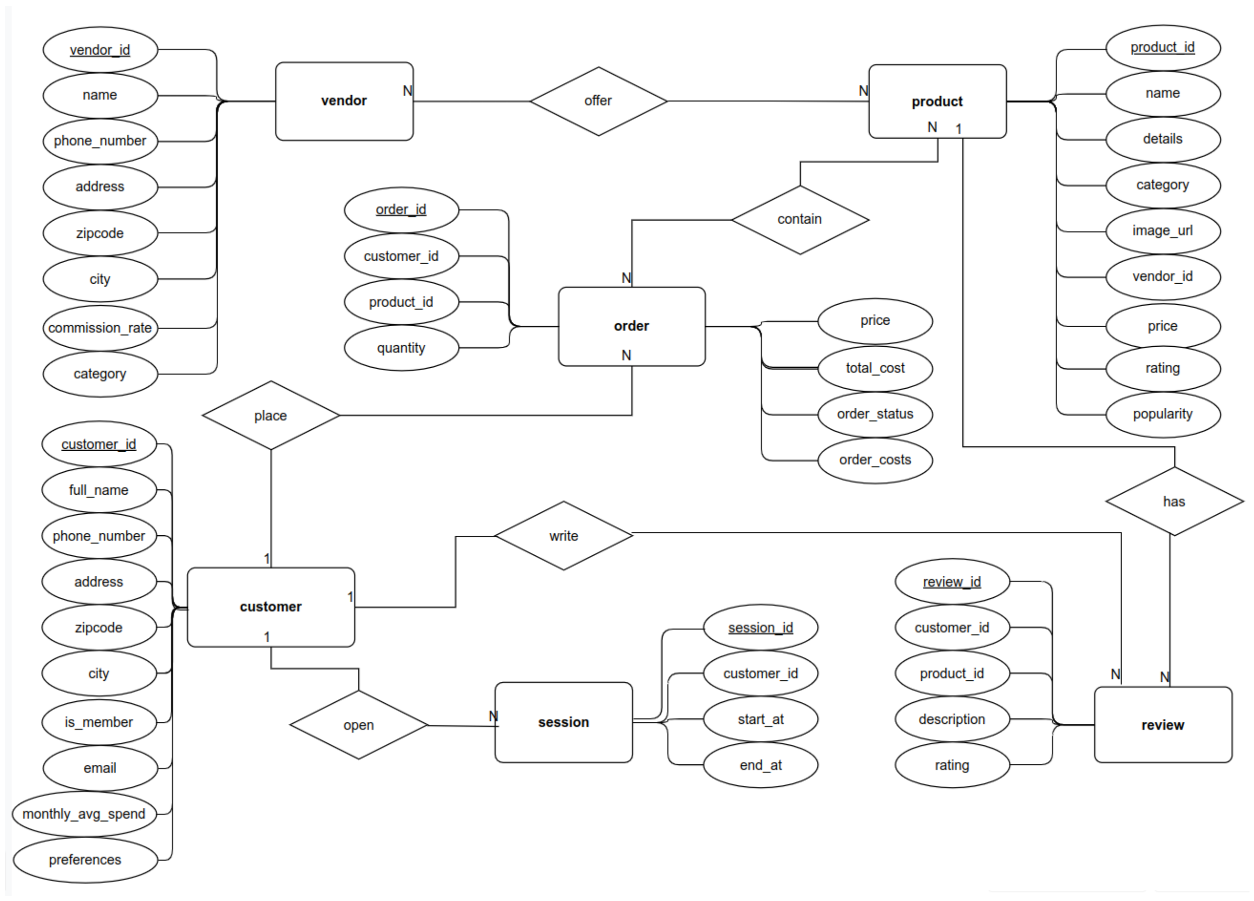


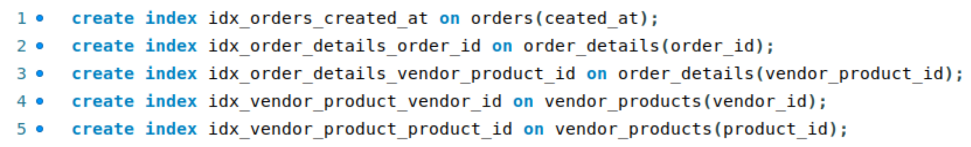
Figure 37: Chen Diagram After Personalized Recommendation Changes

4. THEORETICAL DISCUSSION

4.1. Query Optimization

For this task, we select the total order of products in each quarter (Figure 34). It takes 0.160 seconds to execute for 1000 customers, 1000 products, and 10000 orders. When we enlarge the data volume 10 times (10000 customers, 10000 products, and 100000 orders), it takes 2.287 seconds which means about 14 times slower.

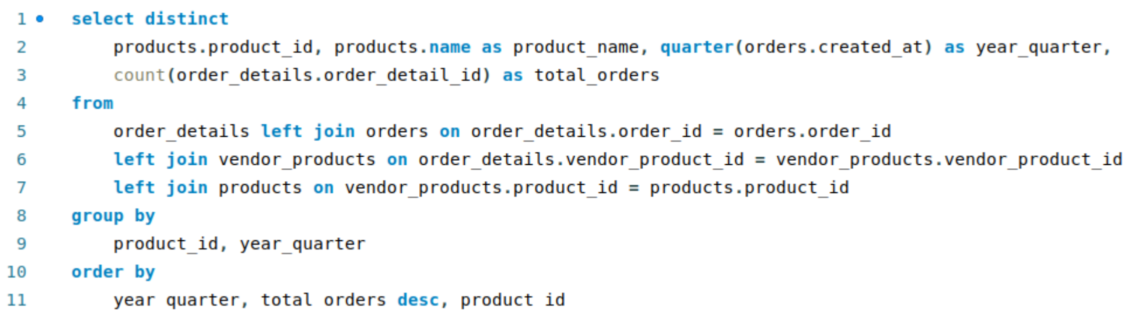
For better performance, we can index some fields participating in the query executions (Figure 38). It reduces the execution time to 2.213 seconds as it is not a very significant change.



```sql
create index idx_orders_created_at on orders(ceated_at);
create index idx_order_details_order_id on order_details(order_id);
create index idx_order_details_vendor_product_id on order_details(vendor_product_id);
create index idx_vendor_product_vendor_id on vendor_products(vendor_id);
create index idx_vendor_product_product_id on vendor_products(product_id);
```

Figure 38: Index Some Fields In The Query

By changing the query stucture, we can reach an execution time of 1.818 seconds (Figure 39). It shows that changing the way of query writing can meaningfully enhance the performance with the same results.



```sql
select distinct
    products.product_id, products.name as product_name, quarter(orders.created_at) as year_quarter,
    count(order_details.order_detail_id) as total_orders
from
    order_details left join orders on order_details.order_id = orders.order_id
    left join vendor_products on order_details.vendor_product_id = vendor_products.vendor_product_id
    left join products on vendor_products.product_id = products.product_id
group by
    product_id, year_quarter
order by
    year_quarter, total_orders desc, product_id
```

Figure 39: Total Order Of Products In Each Quarter (Changed Query)

4.2. ACID Properties

Relational Database management systems (RDBMS) are designed to comply with ACID properties to assure the reliability and correctness of transactions. ACID properties are Atomicity, Consistency, Isolation, and Durability (Härder and Reuter, 1983). This study discusses each one in a simple way and gives examples in the context of the marketplace.

1. **Atomicity:** Atomicity means that in the execution of a transaction either all queries are applied or none of them. There is an example of placing an order by a customer. We need to insert records in orders, order_details, and order_statuses so all the insert queries should be applied or none of them.

2. **Consistency:** Consistency means transitioning from a valid state to another valid state all the time. For instance, in case of deleting an order, all its details and statuses should be deleted with it. We can guarantee this with on delete cascade constraint in foreign key definitions.
3. **Isolation:** Isolation guarantees that the results of a transaction can be seen by others only if the transaction is completed not in the middle of execution. For example, a vendor is creating a new product in their panel. It needs to insert a product in the products table as well as a record in the vendor_produts table for their relationship. If these queries are executed within a transaction, no one can access the new product until the end of the transaction.
4. **Durability:** It means that the effects of transaction execution will remain permanently after committing. For example, if a customer places an order and after committing server crashes, the order will be available after the server restarts.

4.3. CAP Theorem

The CAP theorem states that among Consistency (C), Availability (A), and Partition Tolerance (P) characteristics, only two of them can be guaranteed in a distributed system (Gilbert and Lynch, 2002). In the marketplace data warehouse, these characteristics can be defined as follows.

- **Consistency:** It means that all nodes get the same correct data in the read operation at the same time.

  A success example is when some customers write positive reviews and give high ratings to the products of a vendor, the rating of that vendor can get higher. So they should be more seen in the website by analytic decisions. It should be the same in all regions (distributed nodes) when customers want to buy the same product.

  A failure example is when the website bans an account due to some security issues, the account can still log in from somewhere else according to inconsistency between nodes.

- **Availability:** It guarantees that every request has a response although it does not include the most recent changes.

  A success example is in the high sale periods, e.g. Black Friday. It is important for all services to be available in order to ease the purchase procedure and sell as much as possible. In this situation, it is not important for all services to give the exact same results to the user.
  A failure example is when the number of connections to the servers and the number of placed order increase, customers in some areas face annoying timeout errors.

- **Partition Tolerance:** It means the system keeps on working even in the presence of network issues.

  A success example is a global marketplace that confronts some network issues in Europe and is forced to stop working temporarily but the servers in the US can work well. After the problem is solved, Europe servers can be synchronized with the others.
  A failure example can be about the global system shutdown for the marketplace website due to network issues in one of its servers.

To conclude, this study suggests AP (Availability and Partition Tolerance) for this data warehouse because it is the source of data for analysis and is not operational so data can be inconsistent sometimes.

4.4. **A Real E-Commerce Company**

The real-world e-commerce company which is chosen for this study is Alibaba. It is one of the largest e-commerce companies in the world. Its data warehousing platform is MaxCompute. The platform is highly scalable, distributed, and cloud-native developed by Alibaba Cloud (Alibaba Cloud, 2025). It also integrates with other Alibaba Cloud services like DataWorks for ETL workflows and Quick BI for business intelligence visualization (Shi and Wang, 2020).

This platform is chosen for its scalability, high throughput, and compatibility with structured, semi-structured, and unstructured data. It supports distributed processing as well as integration with tools like Spark and Flink. Other platforms like the Hadoop ecosystem used initially but were replaced when Alibaba Data grew exponentially.

One of the use cases of this system is a real-time personalized recommendation. MaxCompute analyzes data from different sources such as purchase history and customer behavior to provide customized recommendations for each customer. It increases not only customer satisfaction but also the total purchase amount. It can also increase the conversion rate in corporations with some first-purchase offers.

Role-Based Access Control can be a suitable solution for access management of the company. It ensures that employees only have access to the data necessary for their roles, such as marketing, supply chain, or finance.

5. **CONCLUSION**

This study successfully designed and developed an analytic data warehouse for a global online marketplace. The design starts with an ER Diagram in Chen notation and ends in a normalized schema in a third normal form to guarantee a good structure. Data warehouse schema utilizes primary keys, foreign keys with on-delete constraints, unique constraints, checks and triggers, and indexes to provide consistency and performance. Database is populated with realistic data, and analysis with SQL queries gives a lot of insights about it. They offer information about revenue from vendors, customers' behavior, and product trends in different quarters. Moreover, the study states useful information about query optimizations. It reveals that in some cases the way of writing a query can be more effective on performance than applying indexes. The personalized recommendation section depicts analytic data warehouses need some more entities and attributes rather than normal ones to offer more effective and performant recommendations. Furthermore, The study on ACID properties and CAP theorem gives examples of success and failure in the marketplace context. In the end, a real-world scenario of Alibaba's data warehousing system is discussed and shows why Alibaba has chosen MaxCompute as its data warehouse platform.

BIBLIOGRAPHY

Alibaba Cloud, 2025. *MaxCompute: Big data platform*. [online] Available at: https://www.alibabacloud.com/ [Accessed 1 February 2025].

Chen, P.P., 1976. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1), pp.9–36.

Codd, E.F., 1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), pp.377–387.

Codd, E.F., 1971. Further normalization of the data base relational model. In: R. Rustin, ed. *Courant Computer Science Symposia Series*, Vol. 6: Data Base Systems. Englewood Cliffs, NJ: Prentice-Hall, pp.33–64.

Davenport, T. H., & Harris, J. G. (2007). *Competing on Analytics: The New Science of Winning*. Harvard Business Review Press.

Gilbert, S. and Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), pp.51–59.

Härder, T. and Reuter, A., 1983. *Principles of transaction-oriented database recovery*. ACM Computing Surveys (CSUR), 15(4), pp.287–317

Inmon, W.H. (2005) *Building the Data Warehouse.* 4th edn. Indianapolis: Wiley.

Kimball, R. and Ross, M. (2013) *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling.* 3rd edn. Hoboken, NJ: Wiley.

Laudon, K.C. and Traver, C.G. (2021) *E-commerce 2021: Business, Technology, Society.* 16th edn. London: Pearson.

Provost, F., & Fawcett, T. (2013). *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking*. O'Reilly Media.

Shi, Y. and Wang, X., 2020. *Big data technologies at Alibaba*. Springer.

Visual Paradigm Online (2025) *Visual Paradigm Online*. Available at: https://online.visual-paradigm.com/diagrams/ (Accessed: 20 January 2025).

TABLE OF FIGURES