WARWICK BUSINESS SCHOOL
THE UNIVERSITY OF WARWICK

Student Numbers: Please list numbers of all group members	**5587248,5550544,2289197,5587165,5576953**
Module Code:	**IB9HP0**
Module Title:	**Data Management**
Submission Deadline:	**12:00 PM (UK Time) Monday 20th March 2024**
Date Submitted:	**19th March 2024**
Word Count:	
Number of Pages:	**72**
Question Attempted: *(question number/title, or description of assignment)*	**5**
Have you used Artificial Intelligence (AI) in any part of this assignment?	**Section 2 and 3**

Academic Integrity Declaration
We're part of an academic community at Warwick. Whether studying, teaching, or researching, we're all taking part in an expert conversation which must meet standards of academic integrity. When we all meet these standards, we can take pride in our own academic achievements, as individuals and as an academic community.

Academic integrity means committing to honesty in academic work, giving credit where we've used others' ideas and being proud of our own achievements.

In submitting my work, I confirm that:

- I have read the guidance on academic integrity provided in the Student Handbook and understand the University regulations in relation to Academic Integrity. I am aware of the potential consequences of Academic Misconduct.
- I declare that this work is being submitted on behalf of my group and is all our own, , except where I have stated otherwise.
- No substantial part(s) of the work submitted here has also been submitted by me in other credit bearing assessments courses of study (other than in certain cases of a resubmission of a piece of work), and I acknowledge that if this has been done this may lead to an appropriate sanction.
- Where a generative Artificial Intelligence such as ChatGPT has been used I confirm I have abided by both the University guidance and specific requirements as set out in the Student Handbook and the Assessment brief. I have clearly acknowledged the use of any generative Artificial Intelligence in my submission, my reasoning for using it and which generative AI (or AIs) I have used. Except where indicated the work is otherwise entirely my own.
- I understand that should this piece of work raise concerns requiring investigation in relation to any of points above, it is possible that other work I have submitted for assessment will be checked, even if marks (provisional or confirmed) have been published.
- Where a proof-reader, paid or unpaid was used, I confirm that the proof-reader was made aware of and has complied with the University's proofreading policy.

Upon electronic submission of your assessment, you will be required to agree to the statements above

Group 28 IB9HP0

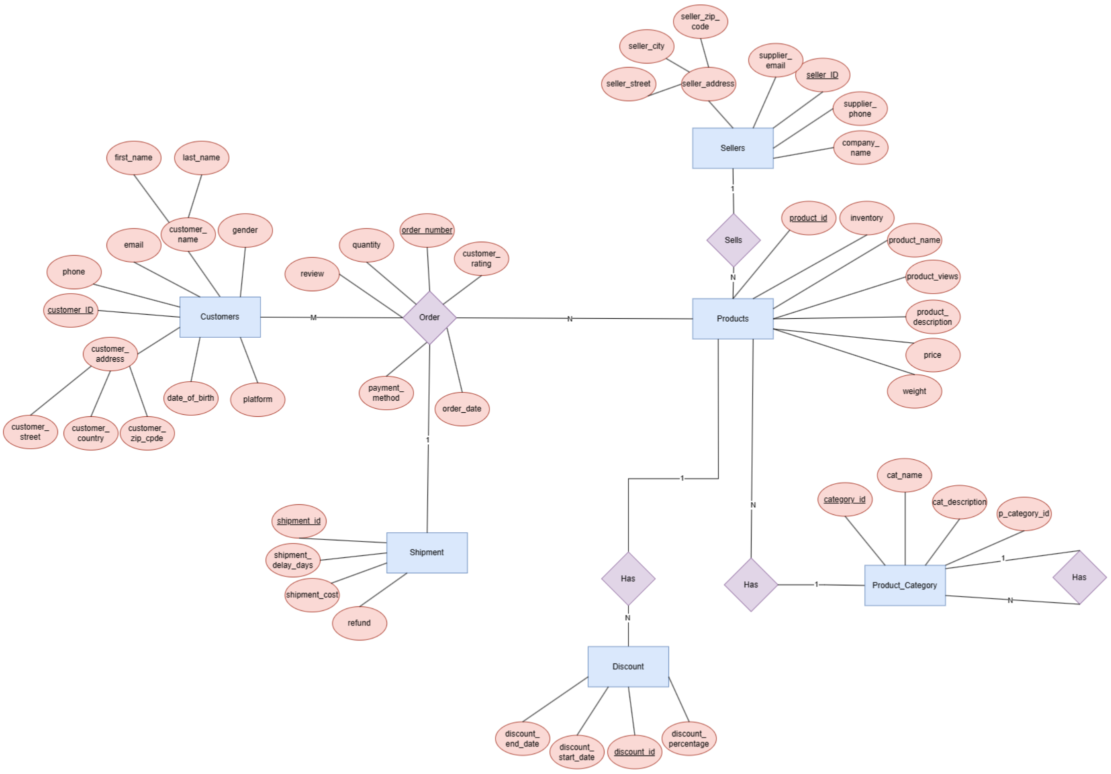
Introduction

In today's constantly expanding digital commerce landscape, having a deep understanding of data is essential for making well-informed decisions and strategizing for business success. This report offers an examination of an e-commerce data environment, from the initial creation of a database to the sophisticated analysis of its contents. Beginning with the E-R diagram as a foundation, it serves as the basis for constructing the entire SQL schema. For generating data, Python was utilized to produce artificial information. Subsequently, a data pipeline was established using GitHub to ensure smooth version control and transparent workflow management - keeping our dataset up-to-date and aligned with business expansion needs. Lastly, leveraging R along with tools such as dplyr and ggplot2 enabled us to conduct advanced analysis on the dataset revealing valuable trends and insights.

The link to our team's repository: [https://github.com/Isaacdexi/dm28]

1. Database Design and Implementation

1.1 Entity Relationship Diagram



1. *Customers Entity*: This represents the customers who use the platform. Attributes include personal information such as first name, last name, email, phone number, and customer ID. Customers also have an associated address that includes street, country, and ZIP code, as well as additional information like gender, date of birth, and platform (the platform through which the customer was acquired).

2. *Order Relationship*: It is linked to the Customers and Products entity. This shows that customers can place orders on products. Each order has an order number, date, payment method, quantity of items ordered, customer rating, and a review of the purchase.

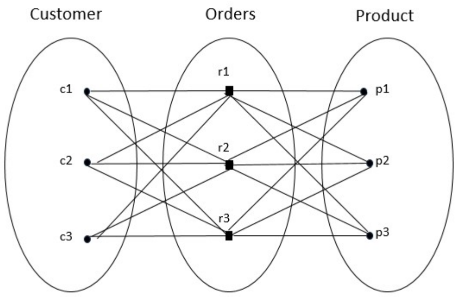
3. *Shipment Entity*: Connected to the Order entity. This represents the shipping details of an order, including a shipment ID, delay in days, cost, and whether a refund was issued.

4. *Products Entity*: This represents the items that are for sale on the platform. Product attributes include a unique product ID, name, description, the number of views, price, and weight. Products are also linked to categories and have an inventory level.

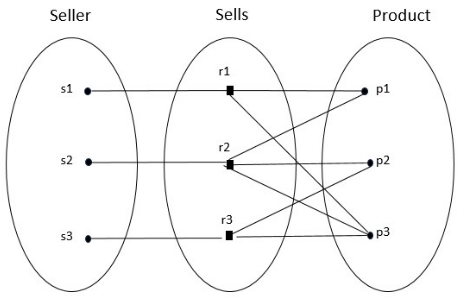
5. *Product_Category Entity*: A junction table that associates products with categories, indicating that products can belong to one or more categories. Categories have a unique ID, name, and description.
6. *Sellers Entity*: Represents the sellers or suppliers on the platform. This includes a seller ID, address details, email, phone, and company name..
7. *Discount Entity*: Connected to the Products entity. This represents discounts that can be applied to products, with attributes like discount ID, start and end date, and the percentage of the discount.

Each rectangle represents an entity, which is a table in the database, while the ovals represent the attributes or fields within those tables. Diamonds represent relationships between entities, and the lines connecting the entities indicate how they are related. The 'M' and 'N' notation specifies the nature of the relationship

Relationship Sets



Some instances in a "Orders" relationship set of M:N relationship, between customer and products.



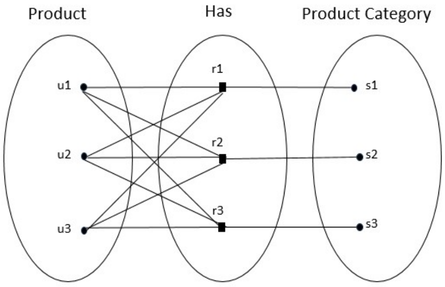
Some instances in a "Sells" relationship set of 1:N, between seller and product.

Customers, Orders, Products

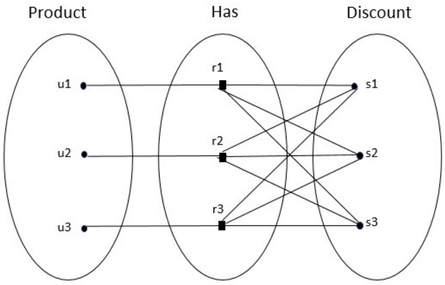
In a many-to-many association involving customers, orders, and products, each customer has the ability to make multiple orders while each order can encompass multiple products. This arrangement provides flexibility in recording the interactions between customers and the items they purchase via orders. It also facilitates monitoring of customer preferences, buying history, and product popularity.

Sellers, Sells, Products

A one-to-many relationship exists among sellers, sales activities (sells), and products where each seller can vend numerous products; however, each product is vended by only one seller. Sellers are connected to the products they sell through their unique identifiers with the products being linked to sellers via the identifier belonging to the respective seller.



Some instances in a "Has" relationship set of N:1 relationship, between product and product category.



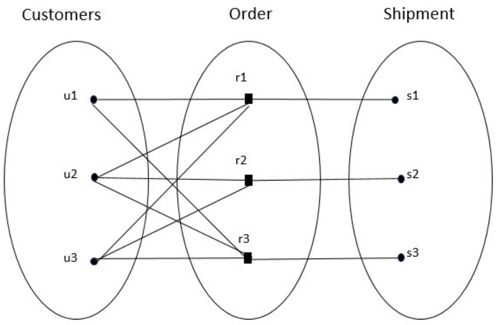
Some instances in a "Has" relationship set of 1:N relationship, between product and discount.

Products, Has, Product Category

In a many-to-one association between products and product categories, several items can be part of the same product category, while each item is associated with only one category. This structure helps organize items into specific categories, making it easier to classify products.

Product, Has, Discount

When it comes to the connection between products and discounts, there is a one-to-many correlation where each product can have multiple related discounts. However, each discount is only applicable to a single product. This arrangement aids in managing the various discounts offered within a product catalog.



Some instances in a "Order" relationship set of M:1 relationship, between customers and shipment.

Customer, Order, Shipment

In a one-to-many connection linking customers, orders, and shipments, a single customer can have multiple associated orders while individual shipments can fulfill various orders. This arrangement facilitates the effective supervision and monitoring of order processing for each customer.

Challenges:

The complexities primarily revolve around excessive entities at the outset, difficulties in normalization, ambiguity in identifying entities and relationships, scope creep, data integrity concerns, user requirements changes, lack of expertise, and conflicting requirements. These multifaceted challenges impede progress, leading to prolonged revisions and delays in schema and data generation. Effective communication, collaboration, and a thorough understanding of database design principles are essential to address these obstacles and ensure the creation of an accurate and efficient Entity-Relationship diagram.

1.2 SQL Database Schema Creation

1.2.1 Logical Schema

Customer (customer_id(primary key), first_name, last_name, gender, date_of_birth, email, phone, customer_street, customer_country, customer_zip_code, platform)

Product (product_id(primary key), product_name, product_description, price, weight, inventory, category_id (foreign key), seller_id (foreign key), product_views)

Category (category_id(primary key), p_category_id, cat_name, cat_description)

Discount (discount_id(primary key), discount_percentage, discount_start_date, discount_end_date, product_id (foreign key))

Order (order_number(primary key), payment_method, order_date, quantity, review, customer_id (foreign key), product_id (primary key), shipment_id (foreign key), customer_rating)

Shipment (shipment_id(primary key), shipment_delay_days, shipment_cost, order_number (foreign key), refund)

Sellers (seller_id(primary key), company_name, supplier_phone, supplier_email, seller_street, seller_country, seller_zip_code)

1.2.2 Physical Schema

```
sqlite3 Ecommerce.db
```

```
# Establish a connection to the SQLite database
database <- dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')
```

The implementation of a robust physical schema essentially converts the conceptual design of the database into a tangible structure, comprising tables, columns, and constraints. It deals with the detailed conversion of logical entities, attributes, and relationships into concrete database tables, indices, and storage mechanisms. Accurate mapping of logical entities and attributes to their physical counterparts ensures efficient storage, retrieval, and integrity of data. This forms the backbone of the platform's data management infrastructure, enabling seamless operations and facilitating informed decision-making in the competitive landscape of online commerce

Customer Table

```
# If Customer table exist, drop it
if(dbExistsTable(database, "Customer")){
  dbExecute(database, "DROP TABLE Customer")
}

dbExecute(database, "CREATE TABLE 'Customer' (
    'customer_id' TEXT PRIMARY KEY,
    'first_name' VARCHAR(50) NOT NULL,
    'last_name' VARCHAR(50) NOT NULL,
    'gender' VARCHAR(50) NOT NULL,
    'date_of_birth' DATETIME NOT NULL,
    'email' VARCHAR(50) NOT NULL,
    'phone' VARCHAR(20) NOT NULL,
    'customer_street' VARCHAR(50) NOT NULL,
    'customer_country' VARCHAR(50) NOT NULL,
    'customer_zip_code' VARCHAR(10) NOT NULL,
    'platform'TEXT NOT NULL)")
```

The Customer entity, representing registered users, is translated into a database table named “Customer”. Each attribute, such as customer_id, first_name, last_name, etc., corresponds to a column within this table. Data types are assigned to each column based on their nature (e.g., VARCHAR for textual data, DATETIME for date of birth).

Product Table

```
# If Product table exist, drop it
if(dbExistsTable(database, "Product")){
  dbExecute(database, "DROP TABLE Product")
}

dbExecute(database,"CREATE TABLE  'Product' (
    'product_id' TEXT PRIMARY KEY,
    'product_name' VARCHAR(255) NOT NULL,
    'price' DECIMAL(10,2) NOT NULL,
    'product_description' TEXT NOT NULL,
    'inventory' INT NOT NULL,
    'weight' DECIMAL(10,2) NOT NULL,
    'category_id' TEXT NOT NULL,
    'seller_id' TEXT NOT NULL,
    'product_views' INT NOT NULL,
    FOREIGN KEY ('category_id') REFERENCES Category ('category_id'),
    FOREIGN KEY ('seller_id') REFERENCES Sellers ('seller_id'))")
```

The Product entity, representing items available for sale, is mapped to the “Product” table in the database. Attributes like product_id, product_name, etc., are represented as columns within this table. Foreign key constraints are implemented for attributes like category_id and seller_id, ensuring referential integrity with the Category and Seller tables.

Category Table

```
# If Category table exist, drop it
if(dbExistsTable(database, "Category")){
  dbExecute(database, "DROP TABLE Category")
}
dbExecute(database,"CREATE TABLE 'Category' (
    'category_id' TEXT PRIMARY KEY,
    'p_category_id' TEXT,
    'cat_name' VARCHAR(255) NOT NULL,
    'cat_description' TEXT NOT NULL)")
```

The Category entity, defining product categories, is instantiated as the “Category” table. Attributes like category_id, cat_name, etc., are mapped to corresponding columns in this table.

Discount Table

```
# If Discount table exist, drop it
if(dbExistsTable(database, "Discount")){
  dbExecute(database, "DROP TABLE Discount")
}
dbExecute(database,"CREATE TABLE  'Discount' (
    'discount_id' INT PRIMARY KEY,
    'discount_percentage' DECIMAL(10,2) NOT NULL,
    'discount_start_date' DATETIME NOT NULL,
    'discount_end_date' DATETIME NOT NULL,
    'product_id' INT NOT NULL,
    FOREIGN KEY ('product_id') REFERENCES Product ('product_id')
)")
```

The Discount entity, representing discounts applicable to products, is materialized as the “Discount” table. Attributes like discount_id, discount_percentage, etc., are mapped to columns within this table. A foreign key constraint is implemented for the product_id attribute, ensuring integrity with the Product table.

Order Table

```
# If Order table exists, drop it
if (dbExistsTable(database, "Order")) {
  dbExecute(database, "DROP TABLE 'Order'")
}
dbExecute(database,"CREATE TABLE 'Order' (
    'order_number' TEXT NOT NULL,
    'payment_method' TEXT NOT NULL ,
    'order_date' DATETIME NOT NULL ,
    'quantity' INTEGER NOT NULL ,
    'review' TEXT,
    'customer_id' TEXT NOT NULL ,
    'product_id' TEXT NOT NULL ,
    'shipment_id' TEXT NOT NULL ,
    'customer_rating' INT NOT NULL,
    PRIMARY KEY ('order_number', 'product_id'),
    FOREIGN KEY ('product_id') REFERENCES Product ('product_id'),
    FOREIGN KEY ('customer_id') REFERENCES Customer ('customer_id'),
    FOREIGN KEY ('shipment_id') REFERENCES Shipment ('shipment_id')
)")
```

The Order entity, capturing customer orders, is realized as the "Order" table. Each attribute, such as order_number, payment_method, etc., is represented as a column in this table.

Foreign key constraints are established for attributes like customer_id, product_id, and shipment_id, maintaining referential integrity with the Customer, Product, and Shipment tables, respectively.

Shipment Table

```
# If shipment table exist, drop it
if(dbExistsTable(database, "Shipment")){
  dbExecute(database, "DROP TABLE 'Shipment'")
}

dbExecute(database,"CREATE TABLE 'Shipment' (
  'shipment_id' TEXT PRIMARY KEY,
  'shipment_delay_days' INT NOT NULL,
  'shipment_cost' DECIMAL(10,2) NOT NULL,
  'order_number' TEXT NOT NULL,
  'refund' TEXT NOT NULL,
  FOREIGN KEY ('order_number') REFERENCES `Order` ('order_number')
)")
```

The Shipment entity, representing shipment details, is translated into the "Shipment" table. Attributes like shipment_id, shipment_delay_days, etc., are mapped to columns in this table. A foreign key constraint is implemented for the order_number attribute, ensuring integrity with the Order table.

Seller Table

```
# If Sellers table exist, drop it
if(dbExistsTable(database, "Sellers")){
  dbExecute(database, "DROP TABLE 'Sellers'")
}

dbExecute(database,"CREATE TABLE 'Sellers' (
    'seller_Id' TEXT PRIMARY KEY,
    'company_name' VARCHAR(100) NOT NULL ,
    'supplier_phone' VARCHAR(20) NOT NULL,
    'supplier_email' VARCHAR(100) NOT NULL UNIQUE,
    'seller_Street' VARCHAR(255) NOT NULL,
    'seller_country' VARCHAR(255) NOT NULL,
    'seller_zip_code' VARCHAR(10) NOT NULL)")
```

The Sellers entity, representing platform vendors, is instantiated as the “Sellers” table. Attributes like seller_Id, company_name, etc., are represented as columns within this table.

Challenges:

While designing and implementing the schema for our e-commerce platform database, we encountered several challenges that required careful consideration and problem-solving.

Complex Relationships:

- Managing the many links between entities was one of the main difficulties. For instance, managing many-to-many relationships was necessary to build relationships between customers, orders, and items.

- It was difficult to conclude what can be an Entity, Attribute, or associative relationship. For instance, Order could be an Entity, Attribute as well as associative relationship depending on the scenario. However, we concluded that order is an associative relationship because it acts as a bridge between the customer and the product entities. In a typical e-commerce scenario, a customer places an order for one or more products. This establishes a many-to-many relationship between customers and products since a single customer can place multiple orders, and each order can contain multiple products.

Normalization: Achieving normalization while avoiding excessive data redundancy was another challenge. We had to brainstorm about various attributes aligned to different entities. For instance, we made two entities Product and Product Category to ensure the minimization of data redundancy.

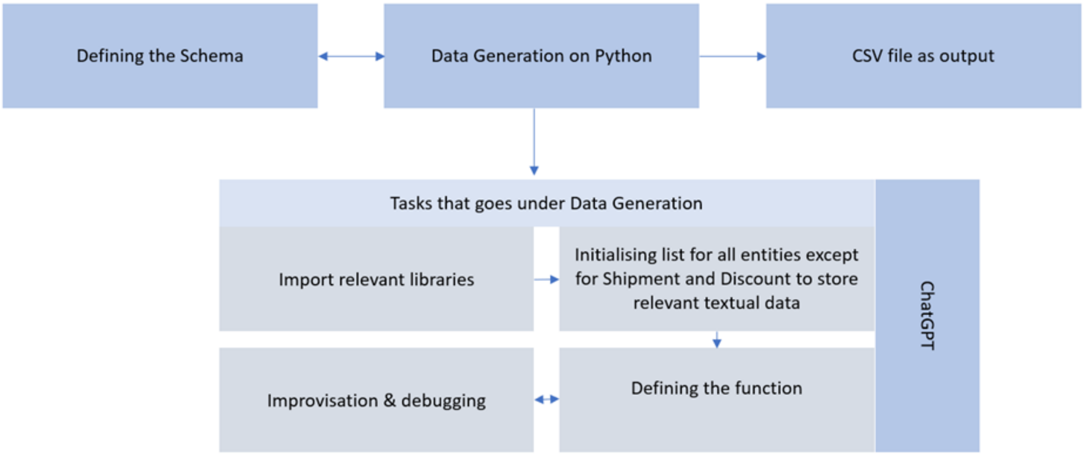
Performance Optimization: Optimizing database performance, especially for complex queries was an important task. We ensured that the query was dynamic and ran efficiently in a loop without any halt. For instance, in our query, we added a drop at the start of every creation query, and instead of the ‘overwrite’ function, we used ‘append’ to maintain the data consistency of the database.

Data Integrity: Data Integrity is important because it measures the accuracy, consistency, and reliability of data stored in a database. We faced challenges while deciding on the primary key of Order Relationship because we were particularly concerned about the need to uniquely identify each order and its associated products. We needed a key that would uniquely identify each order while also because an order could consist of multiple products. After careful consideration, we concluded that a single attribute alone, such as ‘order_number’, may not be sufficient to uniquely identify orders, especially in scenarios where multiple orders with the same number could exist if placed by different customers or at different times. To address this challenge, we opted for a composite primary key consisting of ‘order_number’ and ‘product_id’.

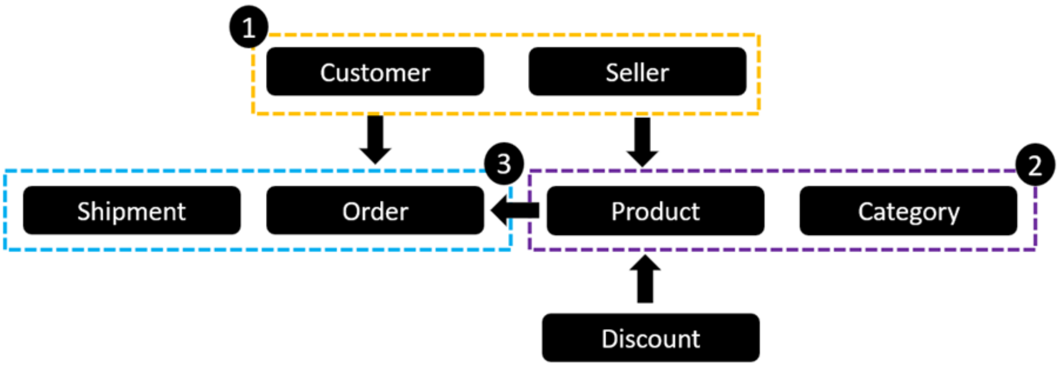
2. Data Generation and Management

2.1 Synthetic Data Generation

During this stage, the team employed Python V3.8.1 and Spyder IDE 5.4.3 to create synthetic data in a CSV file using ChatGPT. The task was aided by the Faker and random libraries to produce a highly authentic eCommerce dataset. Additionally, ChatGPT was used for generating textual data lists, function enhancements, and debugging assistance. The code can be seen in Appendix 2


Defining the Schema
Data Generation on Python
CSV file as output
Tasks that goes under Data Generation
Import relevant libraries
Initialising list for all entities except for Shipment and Discount to store relevant textual data
Improvisation & debugging
Defining the function
ChatGPT

The above diagram depicts the comprehensive approach used to generate data, involving an ongoing cycle between schema design and data generation. This iterative process ensures the creation of highly realistic datasets that accurately reflect an eCommerce database. A specific list is needed to store various textual data, such as reviews, which will be randomized based on purchase orders. Meanwhile, details like product descriptions will be linked to their respective products. ChatGPT supports this process by automating the arduous task of generating large amounts of textual information. Additionally, it aids in organizing codes coherently and enables real-time improvisation, thereby enhancing code interpretability and streamlining data generation efficiency.

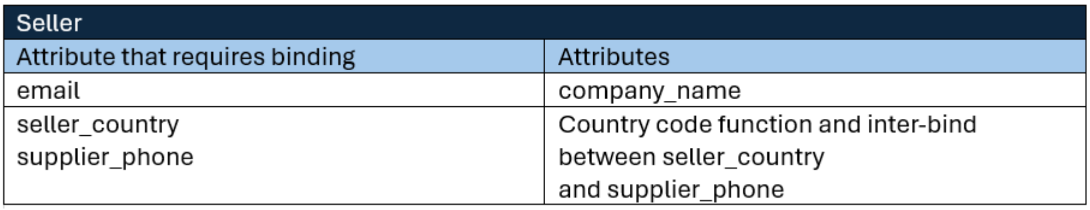


Sequencing Logic between Entities

The logic of sequencing as illustrated can be segmented into 3 tranches, in which, dependencies is considered. The first would be to generate customer and seller information, so category can be created followed by products. Order information is dependent on product and customer, so it makes logical sense to generate the table after the first and second tranches. Shipment is dependent on order, which can be created as the last table alongside discount. It is important to generate this systematically, as certain attributes such as customer_id would need to be first created as it is included in the order table.

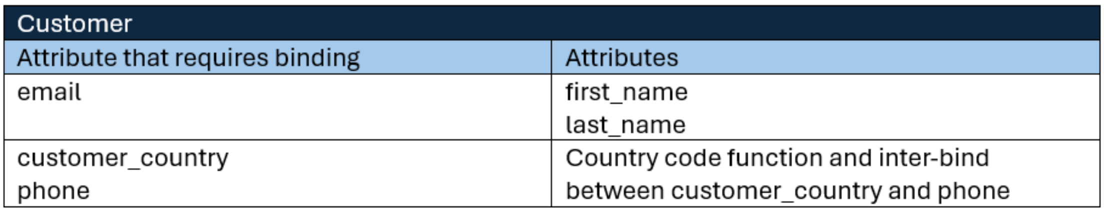
Attribute Data Generation Logic

Faker library is able to do at least 70% of the data generation work, which leaves the remaining to the use of ChatGPT, random library and logic binding of variables (focus of this section). All tables have such operations except for product and discount.



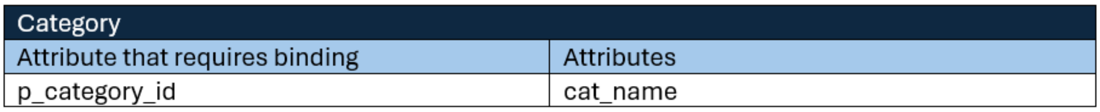
Seller	
Attribute that requires binding	Attributes
email	company_name
seller_country supplier_phone	Country code function and inter-bind between seller_country and supplier_phone

Since Faker is unable to match email domais, data generated for email variables would be based on a [random point-of-contact's name + '@' + extraction of company first name + '-' + '.com', to reflect true email domains in real business setting. As for country, it extracts from another function and list of country code, then based on that, phone number randomise 10 digits attached with the “(+country code)”.



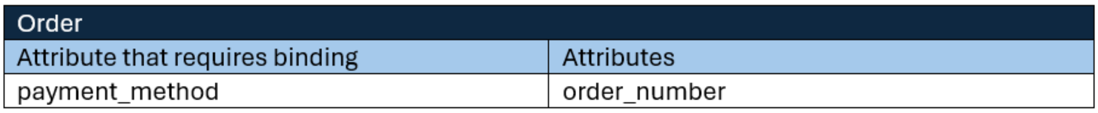
Customer	
Attribute that requires binding	Attributes
email	first_name last_name
customer_country phone	Country code function and inter-bind between customer_country and phone

Since Faker is unable to match email names, data generated for email variables would be based on extracting [first_name + ‘_’ + last_name + ‘@gmail.com’. As for country, it extracts from another function and list of country code, then based on that, phone number randomise 10 digits attached with the “(+country code)”.



Category	
Attribute that requires binding	Attributes
p_category_id	cat_name

Parent category id is based on the mapping of category names since, it is subsumed to a main category, for instance, ‘Sportswear’ is under ‘Sports and Outdoors’.

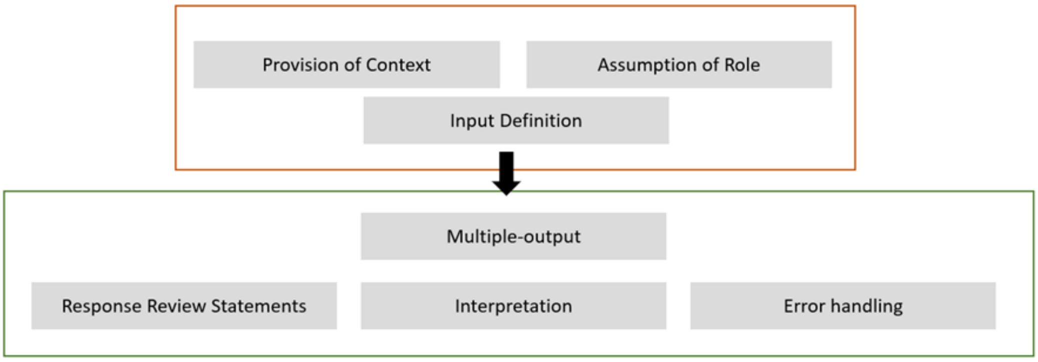


Order	
Attribute that requires binding	Attributes
payment_method	order_number

As a single order number can include multiple products, it is essential to bind the payment method to ensure that it makes logical sense that one order can only have the same payment method. Also, to reflect the M:N relationship between customers and products, the order table would have records that reflect purchases with multiple products.

Overall, certain simplifications are also made, such as a single customer only has one delivery address, an order date would be the same as shipment data, where it is immediately dispatched. Also, the number of records should be tallied in the shipment table, where it references the latest order number in the order table for the creation of shipping_id.

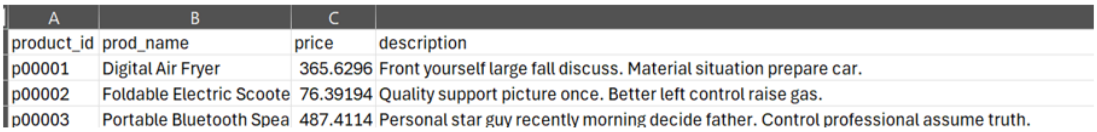
Prompt Strategy



The team defined the ChatGPT prompt strategy into two main fronts, the contextual phase, and the output management stage. To ensure precise output, providing contextual information is crucial, for instance: “This is a database project, which requires the generation of datasets that reflect the eCommerce database environment”. Following which, specifying the role is important for accuracy too, “I am a data engineer working on this project”. Input definition would constitute the granular details and parameters, “The dataset should be in a CSV file, using the Faker library in python, I would like X1, X2, X3, X3....Xn variables, such that X1 should take on the format of a unique identifier, “c00001”, and so on sequentially. In total, there should be 500 rows of data.”

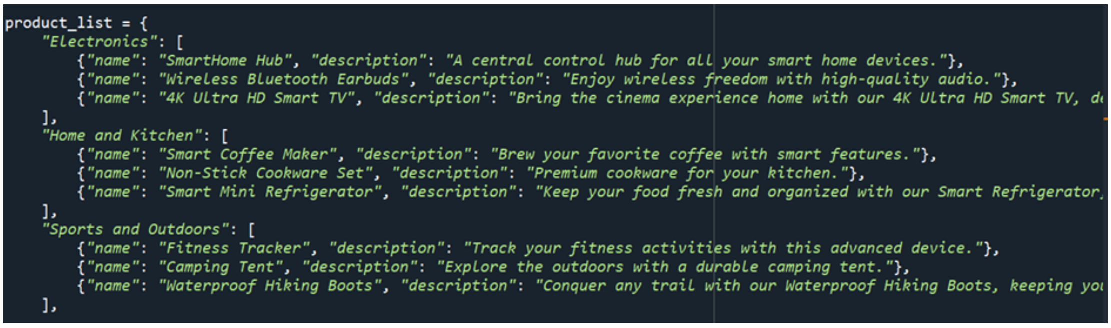
Generating multiple outputs also provide us more flexibility in selecting the most applicable codes to the project, and combined with interpretation prompts, “Please explain the codes”, we can conduct testing of the data generation on our local environment more intuitively. We can create review statements prompts for improvisation, “X2 variable is not supposed to be an integer, please make amendments”, and if errors occur, simply pasting the codes and prompting it to fix the errors would ensure that our data generation is executed smoothly.

Challenges:



A	B	C	
product_id	prod_name	price	description
p00001	Digital Air Fryer	365.6296	Front yourself large fall discuss. Material situation prepare car.
p00002	Foldable Electric Scoote	76.39194	Quality support picture once. Better left control raise gas.
p00003	Portable Bluetooth Spea	487.4114	Personal star guy recently morning decide father. Control professional assume truth.

Although Faker library is considerably powerful in creating realistic datasets, there are certain limitations, particularly, the inability to match records correctly. This is especially striking when generating review and product descriptions, and as shown above, the description is totally not relevant to the product at all.



```
product_list = {
    "Electronics": [
        {"name": "SmartHome Hub", "description": "A central control hub for all your smart home devices."},
        {"name": "Wireless Bluetooth Earbuds", "description": "Enjoy wireless freedom with high-quality audio."},
        {"name": "4K Ultra HD Smart TV", "description": "Bring the cinema experience home with our 4K Ultra HD Smart TV, d
    ],
    "Home and Kitchen": [
        {"name": "Smart Coffee Maker", "description": "Brew your favorite coffee with smart features."},
        {"name": "Non-Stick Cookware Set", "description": "Premium cookware for your kitchen."},
        {"name": "Smart Mini Refrigerator", "description": "Keep your food fresh and organized with our Smart Refrigerator,
    ],
    "Sports and Outdoors": [
        {"name": "Fitness Tracker", "description": "Track your fitness activities with this advanced device."},
        {"name": "Camping Tent", "description": "Explore the outdoors with a durable camping tent."},
        {"name": "Waterproof Hiking Boots", "description": "Conquer any trail with our Waterproof Hiking Boots, keeping yo
    ],
```

The team overcame the challenge through the use of ChatGPT, to create a list of products and relevant descriptions generated by ChatGPT, then store and integrate it in the Python codes with Faker. This requires multiple rounds of as well as multiple trial and error before arriving at a satisfactory solution.

2.2 Data Import and Quality Assurance

2.2.1 Data Loading

```r
setwd("/cloud/project")
Customer <- readr::read_csv("customer.csv")
Category <- readr::read_csv("category.csv")
Sellers <- readr::read_csv("seller.csv")
Product <- readr::read_csv("product.csv")
Discount <- readr::read_csv("discount.csv")
Shipment <- readr::read_csv("shipment.csv")
Order <- readr::read_csv("order.csv")
```

2.2.2 Data Validation

Customer Data Validation

```r
## Validation for customer data

# Function to check if a datetime is in the desired format ("%Y-%m-%d %H:%M:%S")
is_datetime_format <- function(datetime_string) {
  tryCatch({
    as.POSIXlt(datetime_string, format = "%Y-%m-%d %H:%M:%S")
    TRUE
  }, error = function(e) {
    FALSE
  })
}

# Check if dates are in the desired format, if not, convert them
for (i in 1:nrow(Customer)) {
  if (!is_datetime_format(Customer$date_of_birth[i])) {
    Customer$date_of_birth[i] <- as.POSIXct(Customer$date_of_birth[i], format = "%Y-%m-%d %H
  }
}


# Perform the rest of the validation checks
missing_values <- apply(is.na(Customer), 2, sum)

# Check unique customer IDs
if (length(unique(Customer$customer_id)) != nrow(Customer)) {
  print("Customer ID is not unique.")
}

# Check data types for first_name and last_name
if (!all(sapply(Customer$first_name, is.character)) || !all(sapply(Customer$last_name, is.cha
  print("First name and last name should be character.")
}

# Check valid gender values
valid_genders <- c("Male", "Female", "Other")
if (any(!Customer$gender %in% valid_genders)) {
  print("Gender should be Male, Female, or Other.")
}

# Check email format
if (any(!grepl("^\\S+@\\S+\\.\\S+$", Customer$email))) {
  print("Invalid email format")
}

# Regular expressions for phone number formats of Belgium, China, France, United Kingdom, Un
phone_regex <- "^\\(\\+\\d+\\)\\d{10}$"

# Check phone number format for specific countries
invalid_phone_indices <- which(!grepl(phone_regex, Customer$phone))
if (length(invalid_phone_indices) > 0) {
  print("Invalid phone numbers:")
  print(Customer[invalid_phone_indices, ])
}

# Regular expressions for zip code formats of Belgium, China, France, United Kingdom, United
zip_regex <- c(
  "^[0-9]{4}$",  # Belgium
  "^[0-9]{6}$",  # China
  "^[0-9]{5}$",  # France
  "^[A-Z]{2}[0-9]{1,2}[A-Z]? [0-9][A-Z]{2}$",  # United Kingdom
  "^[0-9]{5}-[0-9]{4}$"    # United States
)

# Check zip code format for specific countries
invalid_zip_indices <- which(!grepl(paste(zip_regex, collapse = "|"), Customer$customer_zip_
if (length(invalid_zip_indices) > 0) {
  print("Invalid zip codes:")
  print(Customer[invalid_zip_indices, ])
}

# Check platform values
valid_platforms <- c("Referral", "Instagram", "Facebook", "Others")
if (any(!Customer$platform %in% valid_platforms)) {
  print("Invalid platform values.")
}

# If no errors are found, print a message indicating that the data is valid
if (!any(is.na(missing_values)) &&
    length(unique(Customer$customer_id)) == nrow(Customer) &&
    all(sapply(Customer$first_name, is.character)) &&
    all(sapply(Customer$last_name, is.character)) &&
    all(Customer$gender %in% valid_genders) &&
    all(grepl("^\\S+@\\S+\\.\\S+$", Customer$email)) &&
    length(invalid_phone_indices) == 0 &&
    length(invalid_zip_indices) == 0 &&
    all(Customer$platform %in% valid_platforms)) {
  print("Customer Data is valid. Loading data into the database...")
  RSQLite::dbWriteTable(database, "Customer", Customer, append = TRUE)
  # Load the data into the database
} else {
  print("Data is not valid. Please correct the errors.")
}
```

In the data validation process for the “Customer” dataset, It starts by making sure that the birthdates of customers are in the correct format, so they can be properly analyzed. Then, it checks for common issues like duplicate customer IDs, ensuring that each customer is unique in the dataset. It also verifies that names are stored as text, and gender is recorded as either “Male”, “Female”, or “Other”. Email addresses are checked to ensure they follow a valid format. The script also examines phone numbers and zip codes, making sure they match the expected patterns for different countries. If everything checks out, it gives the green light to load the data into the database; otherwise, it flags any errors that need attention.

Discount Data Validation

```r
# Function to check if date is in the desired format
is_datetime_format <- function(x) {
  tryCatch({
    as.POSIXlt(x, format = "%Y-%m-%d %H:%M:%S")
    TRUE
  }, error = function(e) {
    FALSE
  })
}

# Convert discount_start_date and discount_end_date to desired format if not already in that
if (!all(sapply(Discount$discount_start_date, is_datetime_format))) {
  Discount$discount_start_date <- as.POSIXlt(Discount$discount_start_date, format = "%Y-%m-%
}

if (!all(sapply(Discount$discount_end_date, is_datetime_format))) {
  Discount$discount_end_date <- as.POSIXlt(Discount$discount_end_date, format = "%Y-%m-%d %H
}

# Check for missing values in Discount dataframe
na_disc <- apply(is.na(Discount), 2, sum)

# Validate discount_percentage, discount_start_date, and discount_end_date data types
valid_decimal <- function(x) {
  !is.na(as.numeric(x))
}

valid_datetime <- function(x) {
  !is.na(as.POSIXlt(x))
}

# Check discount percentage range (assuming it's between 0 and 100)
```

```r
if (any(Discount$discount_percentage < 0 | Discount$discount_percentage > 100) ||
    !all(sapply(Discount$discount_percentage, valid_decimal))) {
  print("Invalid discount percentage.")
}

# Check discount dates
if (any(Discount$discount_start_date >= Discount$discount_end_date) ||
    !all(sapply(Discount$discount_start_date, valid_datetime)) ||
    !all(sapply(Discount$discount_end_date, valid_datetime))) {
  print("Discount start date should be before the end date.")
}

# Check if discount_id is unique
if (any(duplicated(Discount$discount_id))) {
  print("Duplicate discount IDs found.")
}

# Check if product_id exists in Product table
if (any(!Discount$product_id %in% Product$product_id)) {
  print("Invalid product IDs. Some product IDs do not exist in the Product table.")
}

# If no errors are found, print a message indicating that the data is valid
if (!any(is.na(na_disc)) &&
    all(Discount$discount_percentage >= 0 & Discount$discount_percentage <= 100) &&
    all(Discount$discount_start_date < Discount$discount_end_date) &&
    !any(duplicated(Discount$discount_id)) &&
    all(Discount$product_id %in% Product$product_id) &&
    all(sapply(Discount$discount_percentage, valid_decimal)) &&
    all(sapply(Discount$discount_start_date, valid_datetime)) &&
    all(sapply(Discount$discount_end_date, valid_datetime))) {
  print("Discount data is valid. Loading data into the database...")
  RSQLite::dbWriteTable(database, "Discount", Discount, append = TRUE)
  # Load the data into the database
} else {
  print("Data is not valid. Please correct the errors.")
}
```

The validation process for the “Discount” dataset begins with checking if the format of discount start and end dates conforms to the specified datetime format (“%Y-%m-%d %H:%M:%S”). If needed, it converts them accordingly. Subsequently, it detects any missing values in the discount dataset. Then, it verifies the data types of discount percentage, start date, and end date.

Furthermore, it ensures that the discount percentage is within the expected range (0 to 100) and validates that the start date precedes the end date. The process also includes confirming unique discount IDs and validating product IDs referenced in the discount table against those existing in a separate product table. If all checks pass successfully, then script affirms validity of data for loading into database; otherwise prompts correction before proceeding further.

Order Data Validation

```r
na_order <- apply(is.na(Order), 2, sum)

# Check quantity (assuming it should be a positive integer)
if (any(Order$quantity <= 0)) {
  print("Invalid quantity.")
}

# Check customer rating (assuming it should be between 1 and 5)
if (any(Order$customer_rating < 1 | Order$customer_rating > 5)) {
  print("Invalid customer rating.")
}

# Check if product_id exists in Product table
if (any(!Order$product_id %in% Product$product_id)) {
  print("Invalid product IDs. Some product IDs do not exist in the Product table.")
}

# Check if customer_id exists in Customer table
if (any(!Order$customer_id %in% Customer$customer_id)) {
  print("Invalid customer IDs. Some customer IDs do not exist in the Customer table.")
}

# Check if shipment_id exists in Shipment table
if (any(!Order$shipment_id %in% Shipment$shipment_id)) {
  print("Invalid shipment IDs. Some shipment IDs do not exist in the Shipment table.")
}

# Check uniqueness based on primary key (order_number, customer_id, product_id)
if (any(duplicated(Order[c("order_number", "customer_id", "product_id")]))) {
  print("Duplicate records found based on order_number, customer_id, and product_id.")
}

# Check order date format and range
if (any(!is_datetime_format(Order$order_date))) {
  # Convert order date to the desired format if not already
  Order$order_date <- as.POSIXct(Order$order_date, format = "%Y-%m-%d %H:%M:%S", tz = "UTC")
}

# If no errors are found, print a message indicating that the data is valid
if (!any(is.na(na_order)) &&
    all(Order$quantity > 0) &&
    all(Order$customer_rating >= 1 & Order$customer_rating <= 5)&&
    all(Order$product_id %in% Product$product_id) &&
    all(Order$customer_id %in% Customer$customer_id) &&
    all(Order$shipment_id %in% Shipment$shipment_id) &&
    !any(duplicated(Order[c("order_number", "customer_id", "product_id")])) &&
    all(is_datetime_format(Order$order_date))) {
  print("Order data is valid. Loading data into the database...")
  RSQLite::dbWriteTable(database, "Order", Order, append = TRUE)
  # Load the data into the database
} else {
  print("Order data is not valid. Please correct the errors.")
}
```

The validation procedure for the “Order” dataset begins by checking for missing values in the order dataset. It then ensures that the quantity of items ordered is a positive integer and that customer ratings fall within the range of 1 to 5. Additionally, it verifies if the product IDs referenced in the order table exist in the product table, and similarly for customer IDs in the customer table and shipment IDs in the shipment table. The script also checks for uniqueness based on the primary key consisting of order number, customer ID, and product ID. If all validation checks pass without errors, the script confirms the validity of the order data and suggests loading it into the database; otherwise, it prompts to correct the errors before proceeding.

Product Category Data Validation

```r
na_prod_cat <- apply(is.na(Category), 2, sum)

# Ensure "category_id" values are unique
if (length(unique(Category$category_id)) != nrow(Category)) {
  print("category_id values are not unique.")
}

# Check length of "cat_name"
if (any(nchar(Category$cat_name) > 255)) {
  print("cat_name exceeds 255 characters.")
}
```

```r

# Check data type of each column
if (!all(sapply(Category$category_id, is.character)) ||
    !all(sapply(Category$cat_name, is.character)) ||
    !all(sapply(Category$cat_description, is.character))) {
  print("Invalid data type for one or more columns.")
}

# If no errors are found, print a message indicating that the data is valid
if (!any(is.na(na_prod_cat)) &&
    length(unique(Category$category_id)) == nrow(Category) &&
    !any(nchar(Category$cat_name) > 255) &&
    all(sapply(Category$category_id, is.character)) &&
    all(sapply(Category$cat_name, is.character)) &&
    all(sapply(Category$cat_description, is.character))) {
  print("product_category data is valid. Loading data into the database...")
  RSQLite::dbWriteTable(database, "Category", Category, append = TRUE)
  # Load the data into the database
} else {
  print("product_category data is not valid. Please correct the errors.")
}
```

First, it checks if there are any missing values in the dataset. Then, it ensures that each category has a unique identifier. Next, it examines the length of category names to make sure they're not too long, as this could cause issues with storing the data. After that, it verifies the data type of each column to ensure they are all in the expected format. If everything checks out, it prints a message confirming the validity of the product category data and suggests loading it into the database. However, if any issues are detected during the validation process, it prompts the user to correct the errors before proceeding with data loading.

Products Data Validation

```r
# Function to check if a value is decimal
valid_decimal <- function(x) {
  !is.na(as.numeric(x))
}

# Function to check if a value is an integer
valid_integer <- function(x) {
  !is.na(as.integer(x))
}

na_Product <- apply(is.na(Product), 2, sum)
```

```r

# Ensure "product_id" values are unique
if (length(unique(Product$product_id)) != nrow(Product)) {
  print("product_id values are not unique.")
}

# Check length of "product_name"
if (any(nchar(Product$product_name) > 255)) {
  print("product_name exceeds 255 characters.")
}

if (any(!Product$category_id %in% Category$category_id)) {
  print("Invalid category IDs. Some category IDs do not exist in the product_category table."
}

if (any(!Product$seller_id %in% Sellers$seller_id)) {
  print("Invalid seller IDs. Some seller IDs do not exist in the Sellers table.")
}

# Check if inventory and product views are integers
if (any(!sapply(Product$inventory, valid_integer)) || any(!sapply(Product$product_views, val
  print("Inventory and product views should be integers.")
}

# Check if price and weight are decimal
if (any(!sapply(Product$price, valid_decimal)) || any(!sapply(Product$weight, valid_decimal)
  print("Price and weight should be decimal values.")
}

# If no errors are found, print a message indicating that the data is valid
if (!any(is.na(na_Product)) &&
    length(unique(Product$product_id)) == nrow(Product) &&
    !any(nchar(Product$product_name) > 255) &&
    all(Product$category_id %in% Category$category_id) &&
    all(Product$seller_id %in% Sellers$seller_id) &&
    all(sapply(Product$inventory, valid_integer)) &&
    all(sapply(Product$product_views, valid_integer)) &&
    all(sapply(Product$price, valid_decimal)) &&
    all(sapply(Product$weight, valid_decimal))) {
  print("Product data is valid. Loading data into the database...")
  RSQLite::dbWriteTable(database, "Product", Product, append = TRUE)
  # Load the data into the database
} else {
  print("Product data is not valid. Please correct the errors.")
}
```

This script begins by defining two functions to check if a value is a decimal or an integer. Then, it checks for missing values in the product dataset. It ensures that each product has a unique identifier and checks the length of product names to ensure they don't exceed 255 characters. Additionally, it verifies that the category IDs referenced in the product table exist in the product category table and that the seller IDs exist in the sellers table. Furthermore, it checks if inventory and product views are integers and if price and weight are decimal values. If all validation checks pass without errors, the script confirms the validity of the product data and suggests loading it into the database. Otherwise, it prompts to correct the errors before proceeding with data loading.

Seller Data Validation

```
library(stringr)
na_sellers <- apply(is.na(Sellers), 2, sum)

# Ensure "seller_Id" values are unique
if (length(unique(Sellers$seller_id)) != nrow(Sellers)) {
  print("seller_Id values are not unique.")
}

# Check length of "company_name"
if (any(nchar(Sellers$company_name) > 100)) {
  print("company_name exceeds 100 characters.")
}

# Check email format
invalid_emails <- which(!str_detect(Sellers$supplier_email, "\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9
if (length(invalid_emails) > 0) {
  print("Invalid email addresses:")
  print(Sellers[invalid_emails, ])
}

# If no errors are found, print a message indicating that the data is valid
if (!any(is.na(na_sellers)) &&
    length(unique(Sellers$seller_id)) == nrow(Sellers) &&
    !any(nchar(Sellers$company_name) > 100) &&
    length(invalid_emails) == 0) {
  print("Sellers data is valid. Loading data into the database...")
```

```r
  RSQLite::dbWriteTable(database, "Sellers", Sellers, append = TRUE)
  # Load the data into the database
} else {
  print("Sellers data is not valid. Please correct the errors.")
}
```

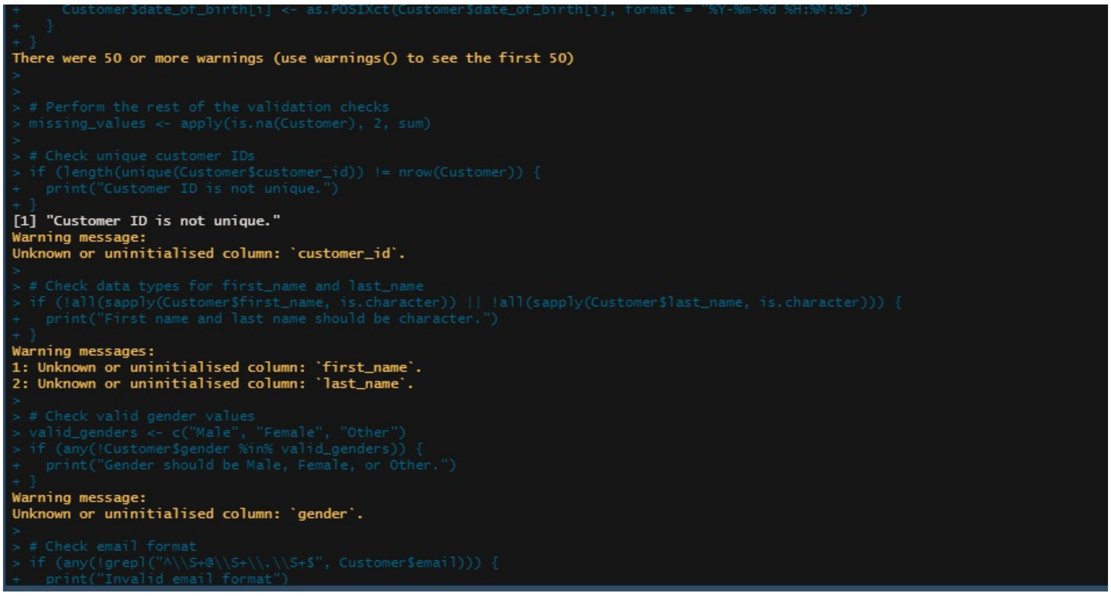
Initially, missing values are identified within the dataset. Then, the uniqueness of “seller_Id” values is confirmed to ensure each identifier is distinct. Moreover, the length of “company_name” is examined to ensure it does not exceed 100 characters. Additionally, email addresses are validated for correct formatting, with any invalid entries flagged for review. If no errors are detected and all validation criteria are met, the “Sellers” data is deemed valid and ready for database loading. However, should any discrepancies arise, corrective measures must be taken before proceeding with data integration to maintain data integrity.

Shipment Data Validation

```r
na_shipment <- sapply(Shipment, function(x) sum(is.na(x)))

# Ensure "shipment_id" values are unique
if (length(unique(Shipment$shipment_id)) != nrow(Shipment)) {
  print("shipment_id values are not unique.")
}

# Validate "refund" column
valid_refunds <- c("Yes", "No")
if (!all(Shipment$refund %in% valid_refunds)) {
  print("Invalid values in the 'refund' column.")
}

# Validate "shipment_delay_days" and "shipment_cost" columns
if (any(Shipment$shipment_delay_days <= 0) || any(Shipment$shipment_cost <= 0)) {
  print("shipment_delay_days and shipment_cost should be positive numbers.")
}

# Ensure that "shipment_delay_days" is an integer
if (any(!as.integer(Shipment$shipment_delay_days) == Shipment$shipment_delay_days)) {
  print("shipment_delay_days should be integers.")
}

# Ensure that all "order_number" values exist in the "Order" table
order_numbers <- unique(Shipment$order_number)
if (!all(order_numbers %in% Order$order_number)) {
  print("Some order numbers do not exist in the 'Order' table.")
}

# If no errors are found, print a message indicating that the data is valid
if (all(na_shipment == 0) &&
    length(unique(Shipment$shipment_id)) == nrow(Shipment) &&
    all(Shipment$refund %in% valid_refunds) &&
    all(Shipment$shipment_delay_days > 0) &&
    all(Shipment$shipment_cost > 0) &&
    all(as.integer(Shipment$shipment_delay_days) == Shipment$shipment_delay_days) &&
    all(order_numbers %in% Order$order_number)) {
  print("Shipment data is valid. Loading data to database ...")
  RSQLite::dbWriteTable(database, "Shipment",Shipment, append = TRUE)
  # Load the data into the database
} else {
  print("Shipment data is not valid. Please correct the errors.")
}
```

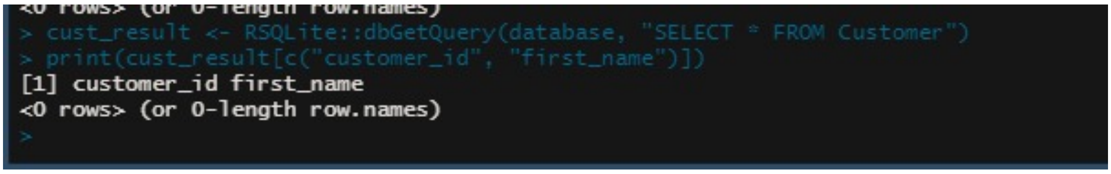
The script is designed to check if our shipment dataset is in good shape. It first counts how many missing values we have in each column, then ensures that every shipment has a unique ID. Next, it looks at the “refund” column to make sure it only has “Yes” or “No” values. It also checks that our shipment delay days and costs are positive numbers, with the delay days being whole numbers. Lastly, it confirms that all the order numbers mentioned in the shipment dataset actually exist in our orders. If everything checks out, it gives the green light to load the shipment data into our database. But if there are issues, it asks us to fix them before moving forward with the data loading process.

Validation Test



```
+     Customer$date_of_birth[i] <- as.POSIXct(Customer$date_of_birth[i], format = "%Y-%m-%d %H:%M:%S")
+   }
+ }
There were 50 or more warnings (use warnings() to see the first 50)
>
>
> # Perform the rest of the validation checks
> missing_values <- apply(is.na(Customer), 2, sum)
>
> # Check unique customer IDs
> if (length(unique(Customer$customer_id)) != nrow(Customer)) {
+   print("Customer ID is not unique.")
+ }
[1] "Customer ID is not unique."
Warning message:
Unknown or uninitialised column: `customer_id`.
>
> # Check data types for first_name and last_name
> if (!all(sapply(Customer$first_name, is.character)) || !all(sapply(Customer$last_name, is.character))) {
+   print("First name and last name should be character.")
+ }
Warning messages:
1: Unknown or uninitialised column: `first_name`.
2: Unknown or uninitialised column: `last_name`.
>
> # Check valid gender values
> valid_genders <- c("Male", "Female", "Other")
> if (any(!Customer$gender %in% valid_genders)) {
+   print("Gender should be Male, Female, or Other.")
+ }
Warning message:
Unknown or uninitialised column: `gender`.
>
> # Check email format
> if (any(!grepl("^\\S+@\\S+\\.\\S+$", Customer$email))) {
+   print("Invalid email format")
```

From the validation results, it is evident that attempts were made to input duplicate values, and the algorithm was able to identify this.



```
<0 rows> (or 0-length row.names)
> cust_result <- RSQLite::dbGetQuery(database, "SELECT * FROM Customer")
> print(cust_result[c("customer_id", "first_name")])
[1] customer_id first_name
<0 rows> (or 0-length row.names)
>
```

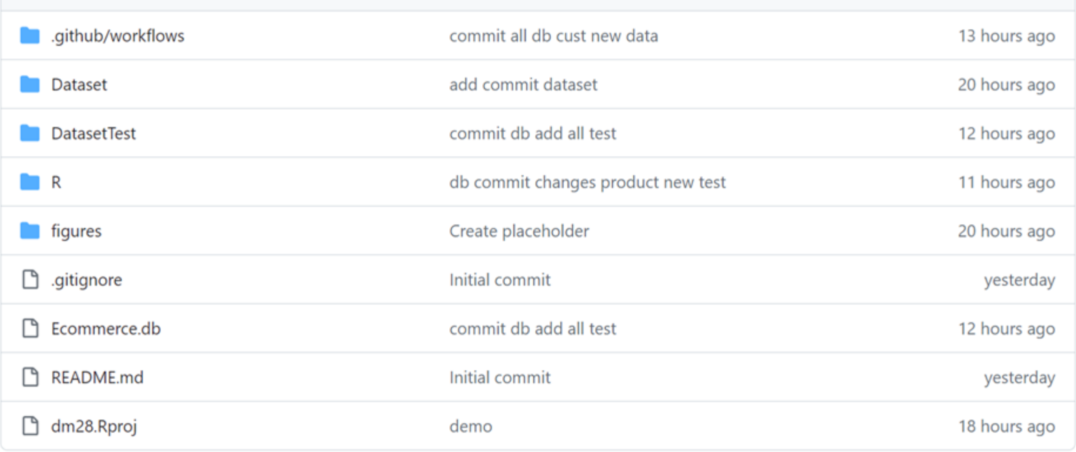
Afterward, we attempted to verify if the data had been stored in the database. Since it was not stored, the code correctly prevents invalid data from being sent to the database.

Challenges:

One of the primary challenges in crafting data validation rules lies in achieving the delicate balance between specificity and generality. These rules must be finely tuned to detect errors effectively without overly restricting legitimate data variations, demanding a profound grasp of the business context to identify critical integrity constraints while accommodating genuine data fluctuations. Moreover, determining how to address invalid data introduces further complexity, requiring careful consideration of factors like data significance, downstream implications, and regulatory compliance. Furthermore, ensuring that validation rules remain dynamic and adaptable to evolving business needs and data trends adds another layer of intricacy, underscoring the importance of ongoing collaboration among teams and iterative refinement of schema and data generation processes.

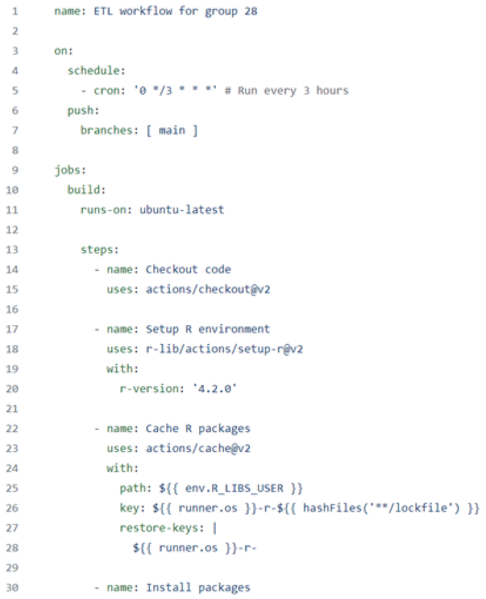
3. Data Pipeline Generation

3.1 GitHub Repository and Workflow Setup

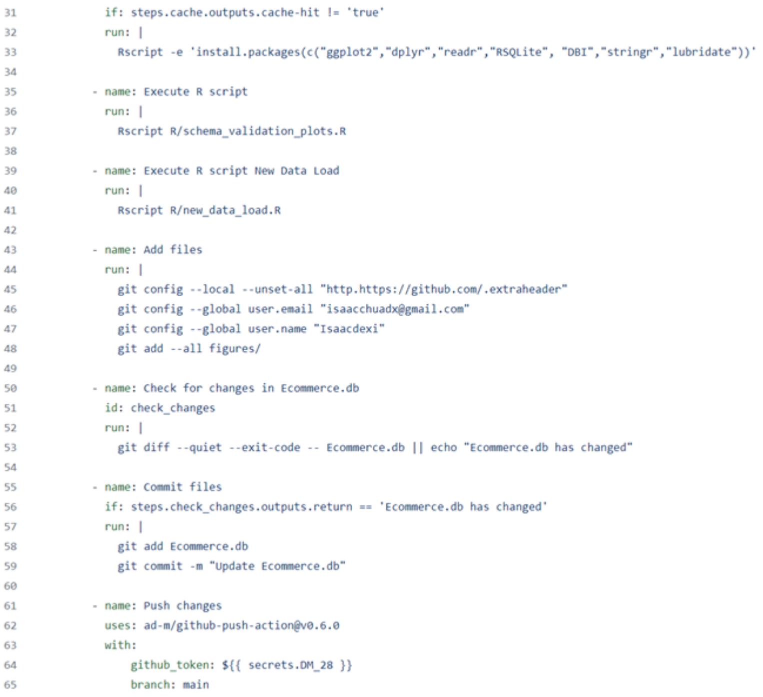


.github/workflows	commit all db cust new data	13 hours ago
Dataset	add commit dataset	20 hours ago
DatasetTest	commit db add all test	12 hours ago
R	db commit changes product new test	11 hours ago
figures	Create placeholder	20 hours ago
.gitignore	Initial commit	yesterday
Ecommerce.db	commit db add all test	12 hours ago
README.md	Initial commit	yesterday
dm28.Rproj	demo	18 hours ago

For version control and data pipeline management purposes, the team created a Github repository, which consists of multiple folders, Dataset, stores our initial database CSV, and DatasetTest, for testing if there are new updates made to the existing database. In R, the team stores 2 Rscripts, one for the schema, validation and analysis, while another one for functions of loading new datasets.



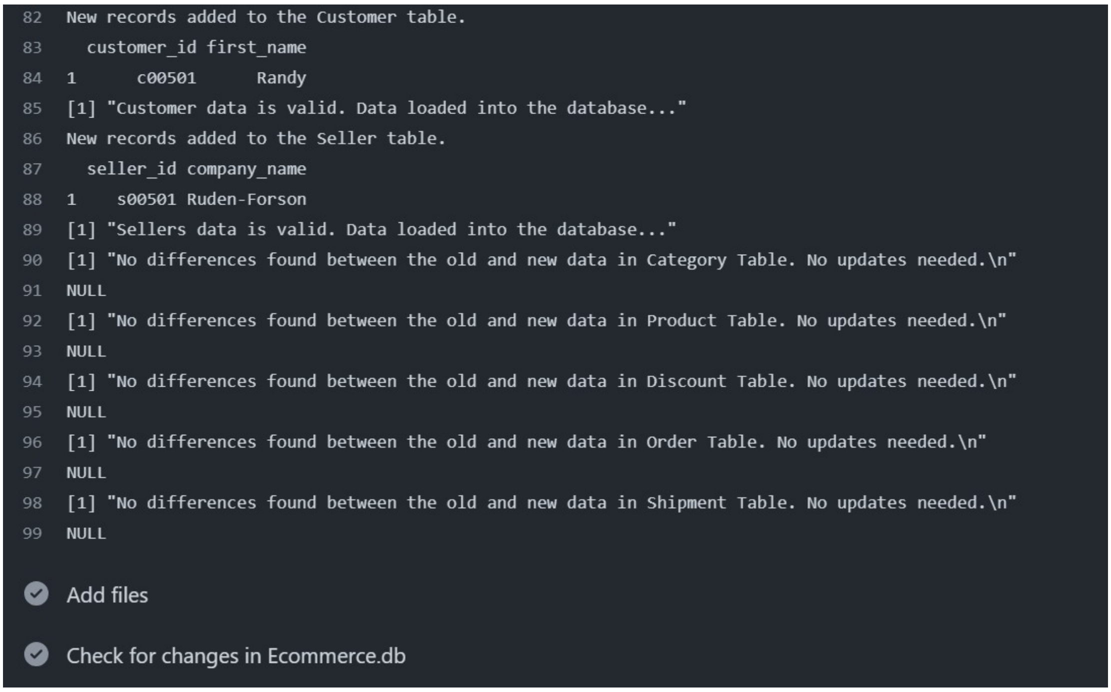
```
name: ETL workflow for group 28

on:
  schedule:
    - cron: '0 */3 * * *' # Run every 3 hours
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Setup R environment
        uses: r-lib/actions/setup-r@v2
        with:
          r-version: '4.2.0'

      - name: Cache R packages
        uses: actions/cache@v2
        with:
          path: ${{ env.R_LIBS_USER }}
          key: ${{ runner.os }}-r-${{ hashFiles('**/lockfile') }}
          restore-keys: |
            ${{ runner.os }}-r-

      - name: Install packages
```



```yaml
        if: steps.cache.outputs.cache-hit != 'true'
        run: |
          Rscript -e 'install.packages(c("ggplot2","dplyr","readr","RSQLite", "DBI","stringr","lubridate"))'

      - name: Execute R script
        run: |
          Rscript R/schema_validation_plots.R

      - name: Execute R script New Data Load
        run: |
          Rscript R/new_data_load.R

      - name: Add files
        run: |
          git config --local --unset-all "http.https://github.com/.extraheader"
          git config --global user.email "isaacchuadx@gmail.com"
          git config --global user.name "Isaacdexi"
          git add --all figures/

      - name: Check for changes in Ecommerce.db
        id: check_changes
        run: |
          git diff --quiet --exit-code -- Ecommerce.db || echo "Ecommerce.db has changed"

      - name: Commit files
        if: steps.check_changes.outputs.return == 'Ecommerce.db has changed'
        run: |
          git add Ecommerce.db
          git commit -m "Update Ecommerce.db"

      - name: Push changes
        uses: ad-m/github-push-action@v0.6.0
        with:
            github_token: ${{ secrets.DM_28 }}
            branch: main
```

Workflow file, “etl.yaml”, is also created, and this provides the main anchor to automate the process of updating the database. It can be seen that the workflow is set by the team to execute the Rscripts, any related files, Ecommerce.db, and pushed accordingly if there are any changes made.

3.2 Github Actions for Continuous Integration

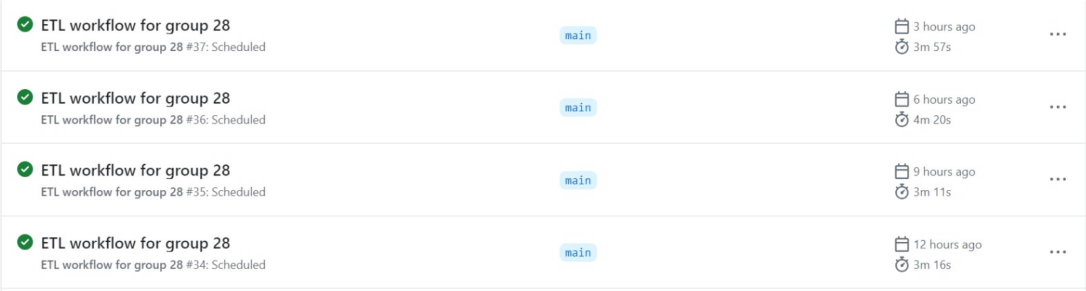


```
New records added to the Customer table.
  customer_id first_name
1      c00501      Randy
[1] "Customer data is valid. Data loaded into the database..."
New records added to the Seller table.
  seller_id company_name
1    s00501 Ruden-Forson
[1] "Sellers data is valid. Data loaded into the database..."
[1] "No differences found between the old and new data in Category Table. No updates needed.\n"
NULL
[1] "No differences found between the old and new data in Product Table. No updates needed.\n"
NULL
[1] "No differences found between the old and new data in Discount Table. No updates needed.\n"
NULL
[1] "No differences found between the old and new data in Order Table. No updates needed.\n"
NULL
[1] "No differences found between the old and new data in Shipment Table. No updates needed.\n"
NULL
```

✓ Add files

✓ Check for changes in Ecommerce.db

Testing our new datasets involves assuming daily uploads of csv files into the environment. We have developed multiple test datasets, modifying customer and supplier tables while keeping the rest unchanged. The outcomes above exhibit the addition of a new customer (c00501) and seller (s00501). This automated database update confirms the successful functioning of our data pipeline.



✓ ETL workflow for group 28
ETL workflow for group 28 #37: Scheduled — main — 3 hours ago — 3m 57s

✓ ETL workflow for group 28
ETL workflow for group 28 #36: Scheduled — main — 6 hours ago — 4m 20s

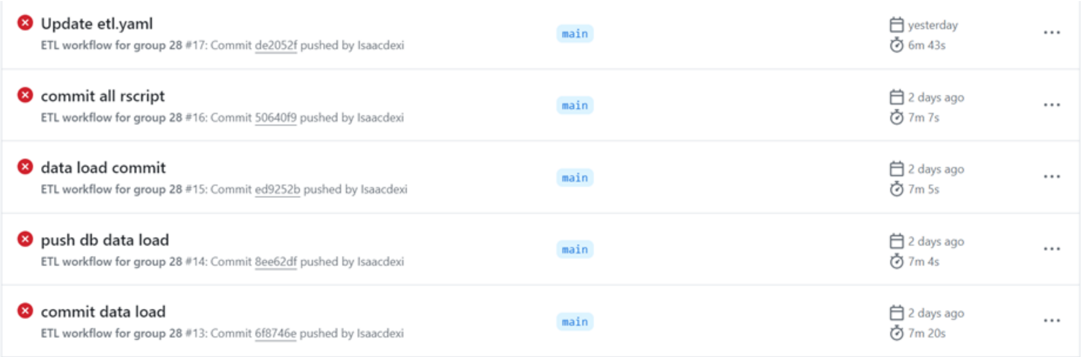
✓ ETL workflow for group 28
ETL workflow for group 28 #35: Scheduled — main — 9 hours ago — 3m 11s

✓ ETL workflow for group 28
ETL workflow for group 28 #34: Scheduled — main — 12 hours ago — 3m 16s

As shown, there are continuous and seamless integration of our workflow in intervals of 3 hours. This ensure that our overall database ecosystem stays updated automatically, whenever there are changes make to the datasets or scripts.

As for the testing of our new datasets, we make the assumption that there would be daily uploads of csv pushed into environment. We created various test datasets, with changes made to customer and supplier tables, while no changes are made to the rest. We also included data validation of the new dataset to ensure that in the event, if there are any errors, the database will not load the new data accordingly (refer to Appendix 1 for Rscript Codes). The results are reflected above, reflecting new customer c00501 and seller, s00501. Therefore, the database is automatically updated, indicating the success of our data pipeline.

Challenges:



Update etl.yaml
ETL workflow for group 28 #17: Commit de2052f pushed by Isaacdexi
main
yesterday
6m 43s

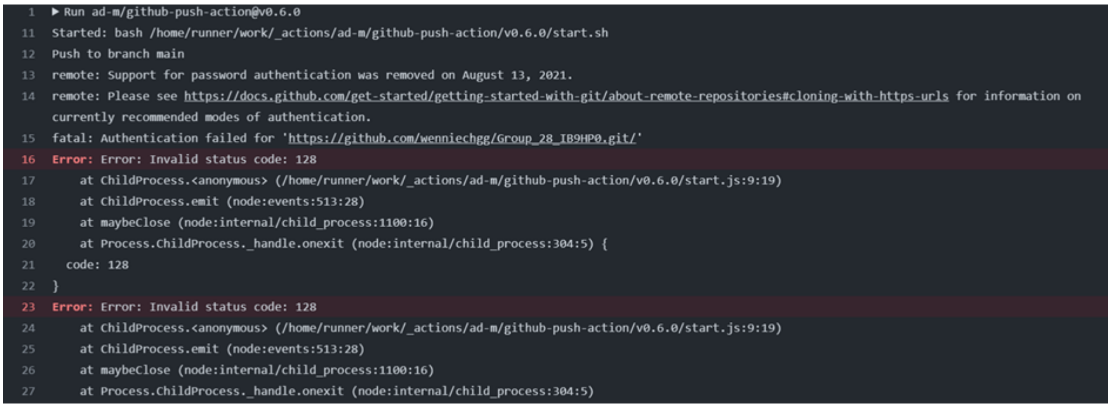
commit all rscript
ETL workflow for group 28 #16: Commit 50640f9 pushed by Isaacdexi
main
2 days ago
7m 7s

data load commit
ETL workflow for group 28 #15: Commit ed9252b pushed by Isaacdexi
main
2 days ago
7m 5s

push db data load
ETL workflow for group 28 #14: Commit 8ee62df pushed by Isaacdexi
main
2 days ago
7m 4s

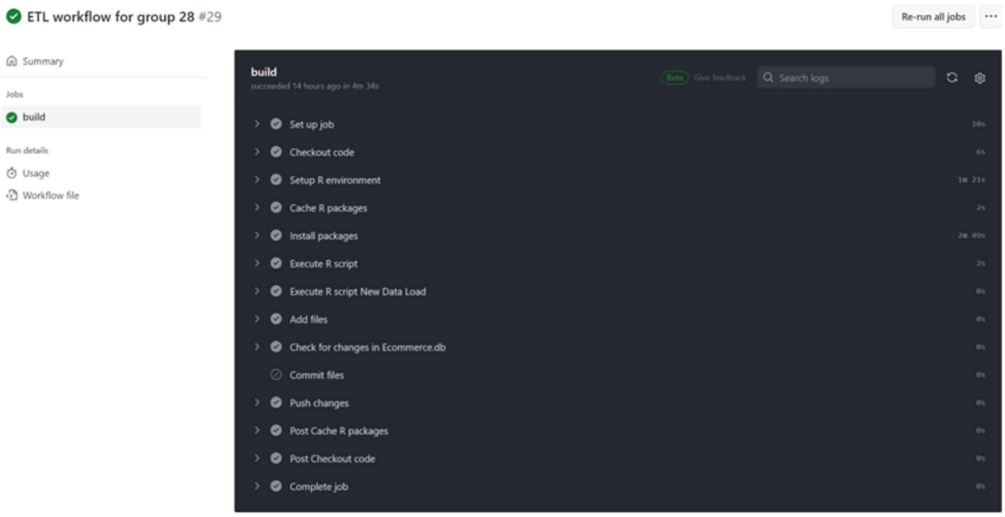
commit data load
ETL workflow for group 28 #13: Commit 6f8746e pushed by Isaacdexi
main
2 days ago
7m 20s

Certainly, the team faces insurmountable challenges in the Github phase of the project. There were multiple errors in the initial stage, and it is due to a variety of reasons such as failure to locate file directory, wrong libraries, syntax errors and authentication failure.



```
▶ Run ad-m/github-push-action@v0.6.0
Started: bash /home/runner/work/_actions/ad-m/github-push-action/v0.6.0/start.sh
Push to branch main
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on
currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/wenniechgg/Group_28_IB9HP0.git/'
Error: Error: Invalid status code: 128
    at ChildProcess.<anonymous> (/home/runner/work/_actions/ad-m/github-push-action/v0.6.0/start.js:9:19)
    at ChildProcess.emit (node:events:513:28)
    at maybeClose (node:internal/child_process:1100:16)
    at Process.ChildProcess._handle.onexit (node:internal/child_process:304:5) {
  code: 128
}
Error: Error: Invalid status code: 128
    at ChildProcess.<anonymous> (/home/runner/work/_actions/ad-m/github-push-action/v0.6.0/start.js:9:19)
    at ChildProcess.emit (node:events:513:28)
    at maybeClose (node:internal/child_process:1100:16)
    at Process.ChildProcess._handle.onexit (node:internal/child_process:304:5)
```

The team overcome these barriers by taking a step-by-step approach to resolve the different erros in the different stages, of which, the most problematic issue was authentication as shown above. The team tried setting up a new repository, and the challenge persisted, until we decide to push the files and change the repository token in our local desktop instead of Posit Cloud. As a result, the workflow was successfully run without any errors and commit any changes accordingly, as shown below.



ETL workflow for group 28 #29

build
succeeded 14 hours ago in 4m 34s

```
Set up job                                10s
Checkout code                             6s
Setup R environment                       1m 21s
Cache R packages                          2s
Install packages                          2m 49s
Execute R script                          2s
Execute R script New Data Load            0s
Add files                                 0s
Check for changes in Ecommerce.db         0s
Commit files                              0s
Push changes                              0s
Post Cache R packages                     0s
Post Checkout code                        0s
Complete job                              0s
```

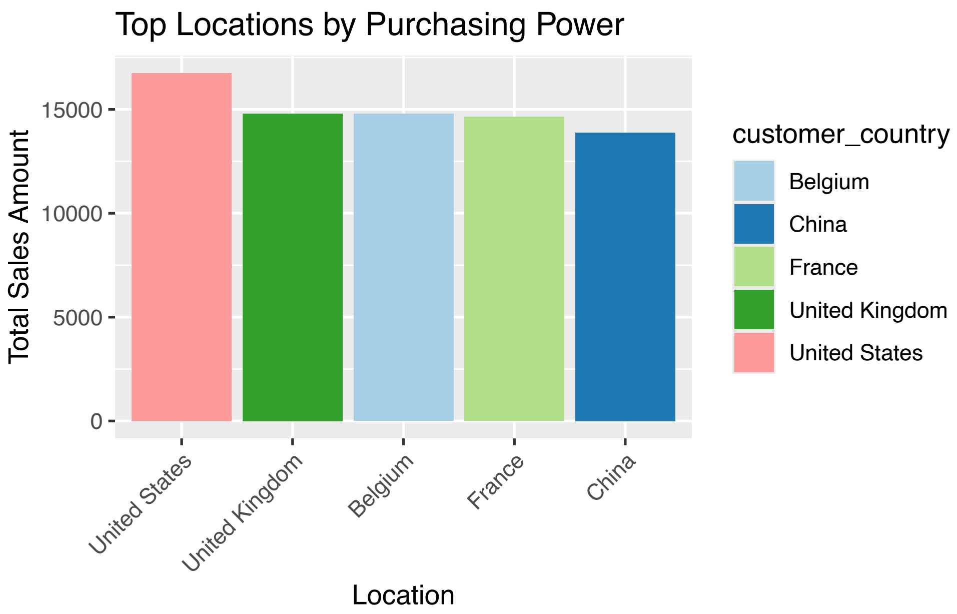
4. Data Analysis and Reporting with Quarto in R

4.1 Advanced Data Analysis in R

1. Top Locations by Purchasing Power

```r
# Query Order and Customer tables, joining them on customer_id
order_customer <- dbGetQuery(database, "
  SELECT O.order_number, O.customer_id, O.product_id, O.quantity, C.customer_country
  FROM `Order` AS O
  JOIN Customer AS C ON O.customer_id = C.customer_id
")

# Join Product table to get product_price
order_customer_product <- dbGetQuery(database, "
  SELECT OC.*, P.price
  FROM (SELECT O.order_number, O.customer_id, O.product_id, O.quantity, C.customer_country
        FROM `Order` AS O
        JOIN Customer AS C ON O.customer_id = C.customer_id) AS OC
  JOIN Product AS P ON OC.product_id = P.product_id
")

# Calculate total sales amount by multiplying quantity and price for each order
order_customer_product <- mutate(order_customer_product, total_sales = quantity * price)

```

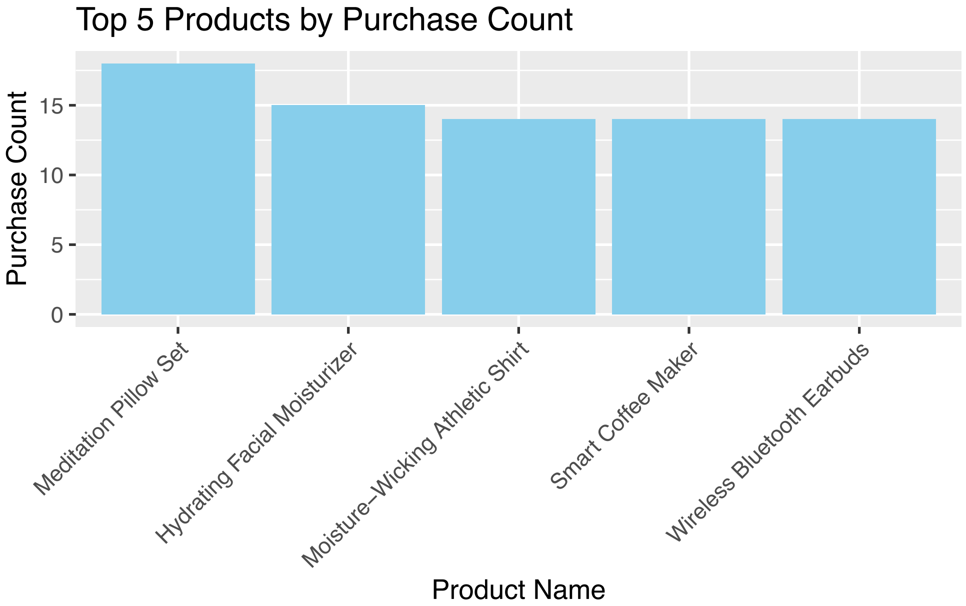
```r
# Group by country and sum the total sales amount
country_sales <- order_customer_product %>%
  group_by(customer_country) %>%
  summarize(total_sales = sum(total_sales))

# Sort the countries by total sales amount in descending order
country_sales <- arrange(country_sales, desc(total_sales))

# Visualize top locations by purchasing power (total sales amount)
ggplot(country_sales[1:5, ], aes(x = reorder(customer_country, -total_sales), y = total_sale
  geom_bar(stat = "identity") +
  labs(x = "Location", y = "Total Sales Amount", title = "Top Locations by Purchasing Power"
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_fill_brewer(palette = "Paired")
```



The analysis begins with identifying the top selling locations by purchasing power, revealing which countries generate the highest total sales. This insight is crucial for understanding market dynamics and targeting regions with higher spending capabilities. The total sales is calculated by joining the ‘Order’ and ‘Customer’ tables and incorporating product prices for each country, identifying the markets with significant economic impact on the e-commerce operations. The figure shows that China leads in purchasing power, followed by France and Belgium, suggesting a strategic opportunity to further tailor marketing and expansion efforts in these countries.

2. Top 5 Products by Number of Purchases

```
# Query Order table and join with Product table to get product names
top_products <- dbGetQuery(database, "
  SELECT P.product_name, COUNT(*) AS purchase_count
  FROM `Order` AS O
  JOIN Product AS P ON O.product_id = P.product_id
  GROUP BY P.product_name
  ORDER BY purchase_count DESC
  LIMIT 5
")


# Visualize the top 5 products by purchase count
ggplot(top_products, aes(x = reorder(product_name, -purchase_count), y = purchase_count)) +
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(x = "Product Name", y = "Purchase Count", title = "Top 5 Products by Purchase Count")
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

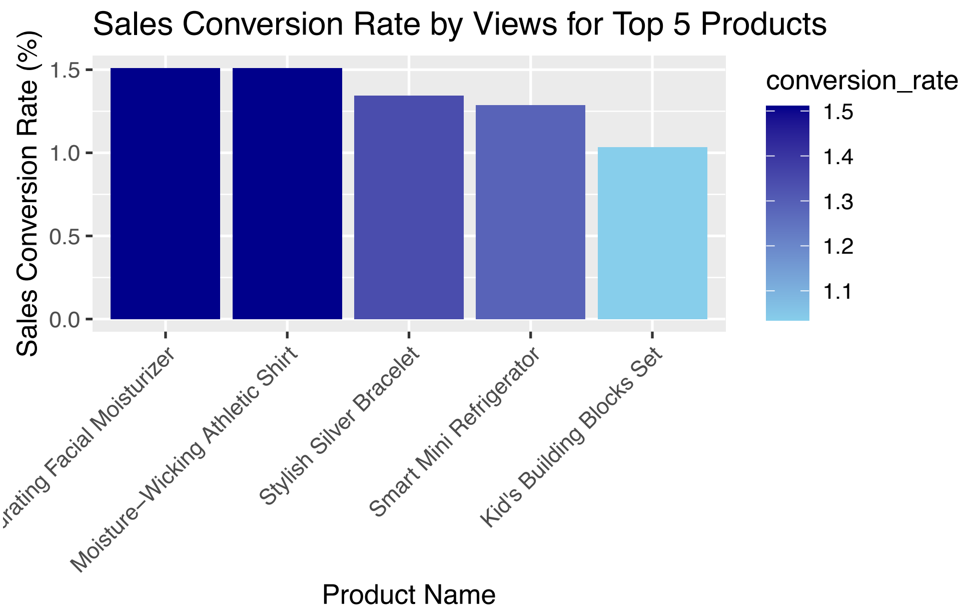


Utilise the sales volume of each product, the product popularity figure shows the top 5 products: Meditation Pillow Set, Hydrating Facial Moisturizer, Moisture-Wicking Athletic Shirt, Smart Coffee Maker, Wireless Bluetooth Earbuds, offering a clear view of consumer preferences. This analysis is invaluable for inventory management and marketing, highlighting the need to focus on these popular products.

Furthermore, we calculate the sales conversion rate for the top 5 products, which shows the direct correlation between product views and purchase behavior.

3. Top 5 Products by Conversion Rate

```
# Query top 5 products by views
top_products <- dbGetQuery(database, "
  SELECT product_id, product_name, product_views
  FROM Product
  ORDER BY product_views DESC
  LIMIT 5
")

# Query total number of purchases for each of the top 5 products
product_purchases <- dbGetQuery(database, "
  SELECT O.product_id, COUNT(*) AS purchases
  FROM `Order` AS O
  WHERE O.product_id IN (SELECT product_id FROM Product ORDER BY product_views DESC LIMIT 5)
  GROUP BY O.product_id
")

# Join product views and purchases
product_conversion <- left_join(top_products, product_purchases, by = "product_id")

# Calculate sales conversion rate (purchases / views) and handle NA values
product_conversion <- mutate(product_conversion, conversion_rate = ifelse(is.na(purchases) |

# Remove NA values
product_conversion <- na.omit(product_conversion)

# Visualize the sales conversion rates for top 5 products
ggplot(product_conversion, aes(x = reorder(product_name, -conversion_rate), y = conversion_ra
  geom_bar(stat = "identity") +
  labs(x = "Product Name", y = "Sales Conversion Rate (%)", title = "Sales Conversion Rate by
  scale_fill_gradient(low = "skyblue", high = "darkblue") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



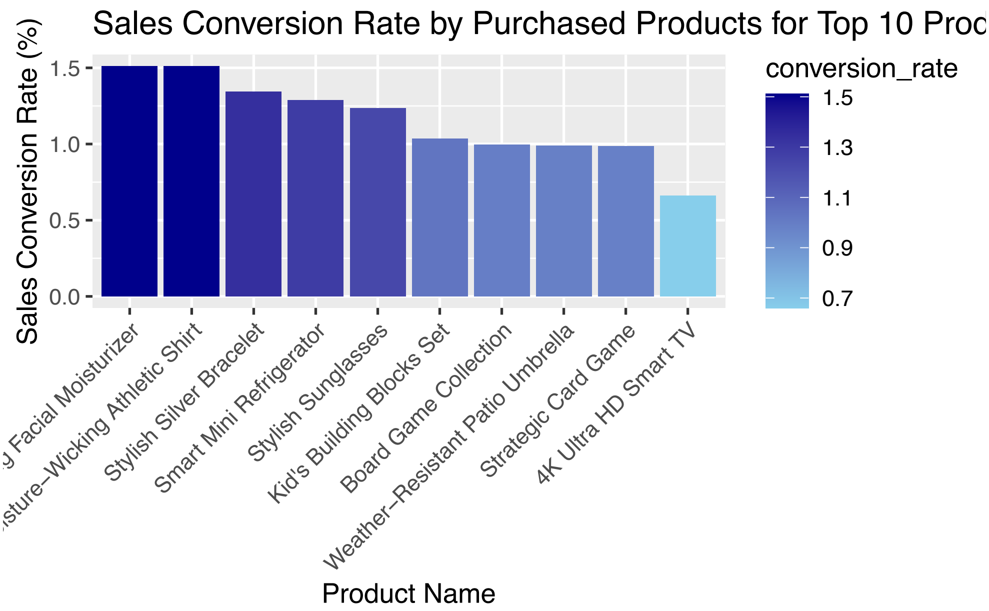
The figure shows that the sales conversion rate for the 'Moisture-Wicking Athletic Shirt' is notably high, suggesting that this product has an effective market fit and presentation. Emphasizing such products in promotional materials could enhance conversion rates.

Then, an extended analysis on the top 10 products offered deeper insights into how visibility influences sales.

4. Sales Conversion Rate by Purchased Products for Top 10 Products

```
# Query top 10 products by views
top_products <- dbGetQuery(database, "
  SELECT product_id, product_name, product_views
  FROM Product
  ORDER BY product_views DESC
  LIMIT 10
")

# Query total number of purchases for each of the top 10 products
product_purchases <- dbGetQuery(database, "
  SELECT O.product_id, COUNT(*) AS purchases
  FROM `Order` AS O
  WHERE O.product_id IN (SELECT product_id FROM Product ORDER BY product_views DESC LIMIT 10)
  GROUP BY O.product_id
")

```

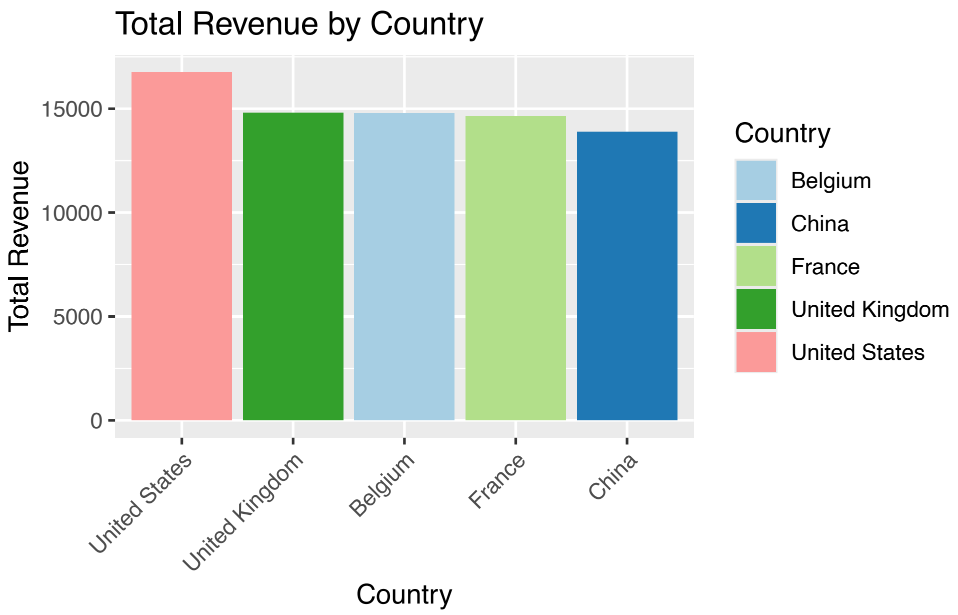
```
# Join product views and purchases
product_conversion <- left_join(top_products, product_purchases, by = "product_id")

# Calculate sales conversion rate (purchases / views) and handle NA values
product_conversion <- mutate(product_conversion, conversion_rate = ifelse(is.na(purchases) |

# Remove NA values
product_conversion <- na.omit(product_conversion)

# Visualize the sales conversion rates for top 10 products
ggplot(product_conversion, aes(x = reorder(product_name, -conversion_rate), y = conversion_ra
  geom_bar(stat = "identity") +
  labs(x = "Product Name", y = "Sales Conversion Rate (%)", title = "Sales Conversion Rate by
  scale_fill_gradient(low = "skyblue", high = "darkblue") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



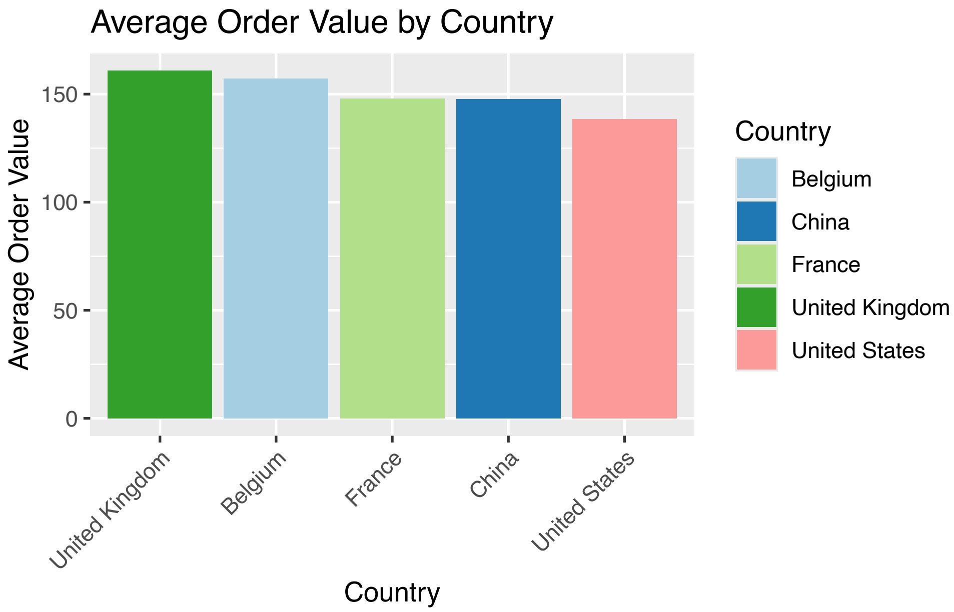
The figure shows that across the top 10 products, the ‘Jing Outdoor Lounge Chair’ has the highest conversion rate, indicating strong consumer interest and the potential for outdoor furniture as a category for business growth.

5. Average orders and revenue by Country

```
# Query the total revenue and count of orders by country
country_orders_revenue <- dbGetQuery(database, "
  SELECT C.customer_country,
         COUNT(O.order_number) AS order_count,
         SUM(O.quantity * P.price) AS total_revenue
  FROM `Order` AS O
  INNER JOIN Customer AS C ON O.customer_id = C.customer_id
  INNER JOIN Product AS P ON O.product_id = P.product_id
  GROUP BY C.customer_country
")

# Calculate the average order value for each country
country_orders_revenue <- mutate(country_orders_revenue, average_order_value = total_revenue

# Visualize the average order value and total revenue by countries
ggplot(country_orders_revenue, aes(x = reorder(customer_country, -total_revenue), y = total_:
  geom_bar(stat = "identity") +
  labs(x = "Country", y = "Total Revenue", title = "Total Revenue by Country", fill = "Count:
  scale_fill_brewer(palette = "Paired") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```
ggplot(country_orders_revenue, aes(x = reorder(customer_country, -average_order_value), y = a
  geom_bar(stat = "identity") +
  labs(x = "Country", y = "Average Order Value", title = "Average Order Value by Country", fi
  scale_fill_brewer(palette = "Paired") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

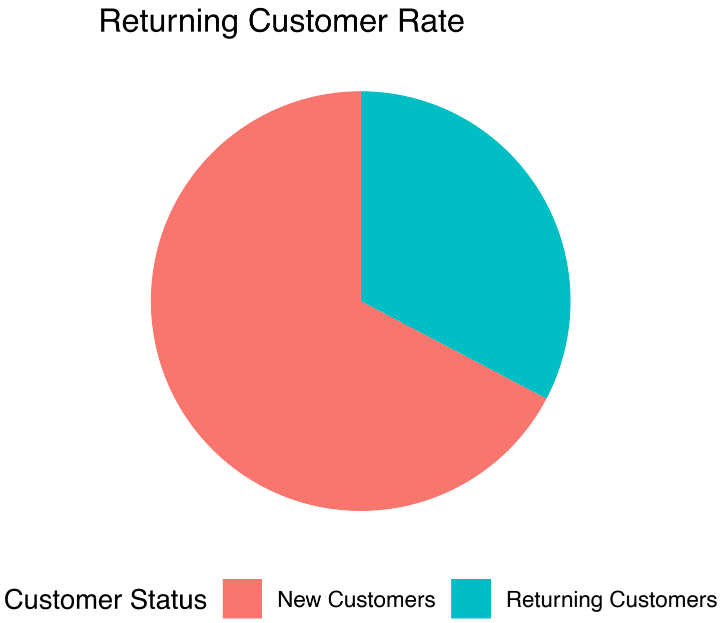


It can be seen in these two figures that China dominates total revenue by country but doesn't lead in average order value, where France takes the lead. Tailoring strategies to increase the average order value in high-revenue countries could balance revenue streams.

6. Rate of returning customers

```
# Query the number of orders for each customer
customer_order_count <- dbGetQuery(database, "
  SELECT customer_id, COUNT(DISTINCT order_number) AS order_count
  FROM `Order`
  GROUP BY customer_id
")

# Calculate the number of returning customers
returning_customers <- sum(customer_order_count$order_count > 1)

# Calculate the total number of customers
```

```r
total_customers <- nrow(customer_order_count)

# Calculate the returning customer rate
returning_customer_rate <- (returning_customers / total_customers) * 100

# Create a data frame for visualization
data <- data.frame(
  Customer_Status = c("Returning Customers", "New Customers"),
  Count = c(returning_customers, total_customers - returning_customers)
)

# Visualize the returning customer rate using a pie chart
ggplot(data, aes(x = "", y = Count, fill = Customer_Status)) +
  geom_bar(stat = "identity", width = 1) +
  coord_polar(theta = "y") +
  labs(fill = "Customer Status", title = "Returning Customer Rate") +
  theme_void() +
  theme(legend.position = "bottom")
```

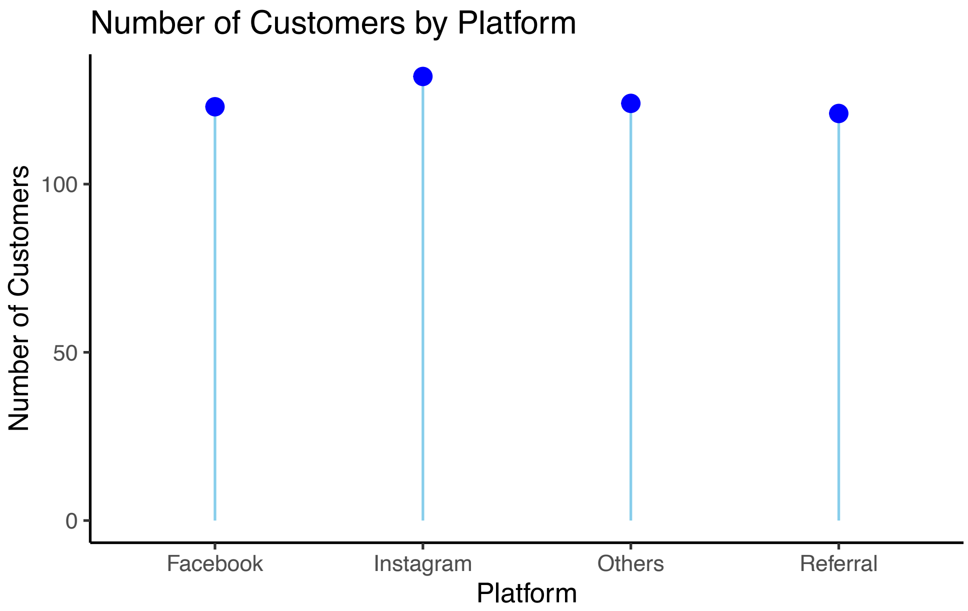


In order to understand customer loyalty, the rate of returning customers was calculated, revealing the proportion of the customer base that makes repeated purchases. This metric is essential for evaluating the success of retention strategies and the overall satisfaction of the customer base. A high rate of returning customers is indicative of a healthy e-commerce ecosystem with strong customer loyalty.

The significant segment of returning customers in this figure implies a strong base of customer loyalty. Retention strategies should continue to be prioritized to maintain this valuable consumer segment.

7. Demographics and Platform Analysis

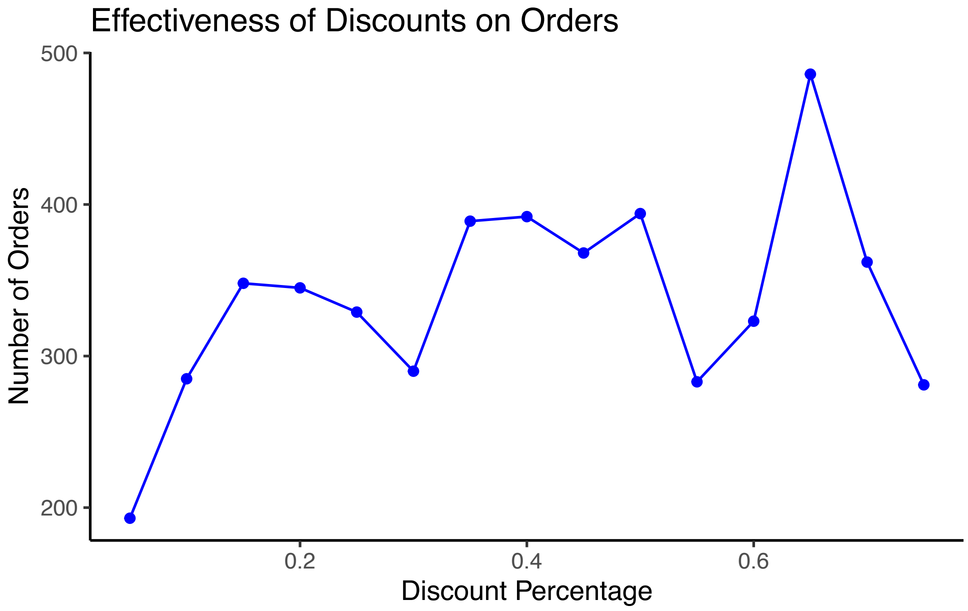
```
# Query the platforms used by customers
customer_platforms <- dbGetQuery(database, "
  SELECT platform, COUNT(*) AS customer_count
  FROM Customer
  WHERE platform IS NOT NULL
  GROUP BY platform
")

# Visualize the analysis
ggplot(customer_platforms, aes(x = platform, y = customer_count)) +
  geom_segment(aes(xend = platform, yend = 0), color = "skyblue") +  # Lollipop stems
  geom_point(color = "blue", size = 3) +  # Lollipop heads
  labs(x = "Platform", y = "Number of Customers", title = "Number of Customers by Platform")
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  theme_classic()
```



The number of customers by platform reveals the most popular channels among the customer base, guiding efforts to optimise presence across platforms. This analysis aids in understanding where to focus digital marketing efforts for maximum engagement and conversion. Shown in this figure, the distribution of customers across four different platforms is relatively even, with 'Facebook' having a slight lead. This suggests an opportunity to optimise and diversify platform-specific marketing strategies for better engagement.

8. The effectiveness of discounts on product sales.

```
# Query discount effectiveness
discount_analysis <- dbGetQuery(database, "
  SELECT D.discount_percentage, COUNT(*) AS order_count
  FROM Discount AS D
  INNER JOIN `Order` AS O ON D.product_id = O.product_id
  GROUP BY D.discount_percentage
")

# Visualize the results
ggplot(discount_analysis, aes(x = discount_percentage, y = order_count)) +
  geom_line(group=1, color = "blue") +
  geom_point(color = "blue") +
  labs(x = "Discount Percentage", y = "Number of Orders", title = "Effectiveness of Discounts
  theme_classic()
```

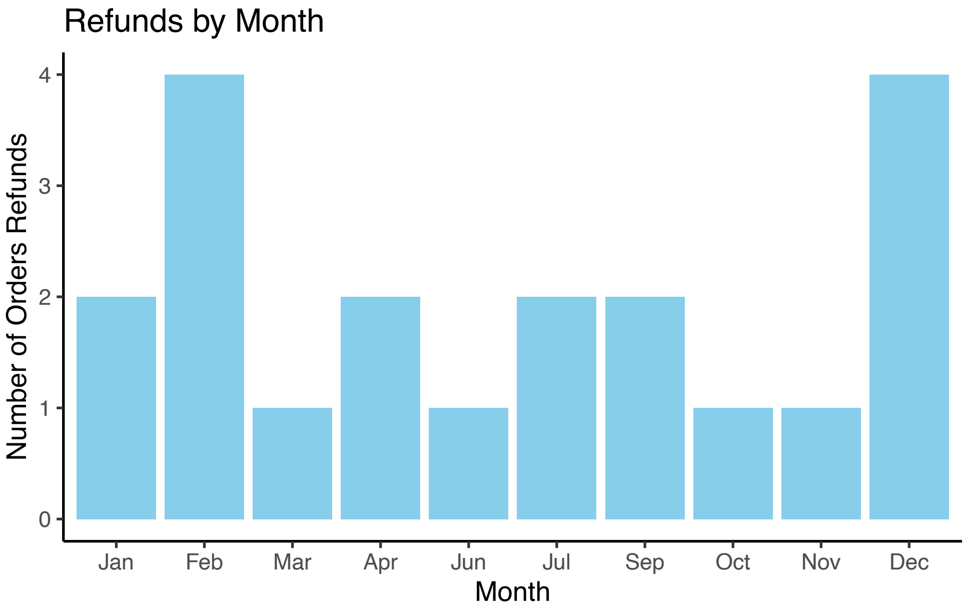


Lastly, the effectiveness of discounts on each product sales is analysed, uncovering how different discount percentages influence the number of orders. This analysis provides a deep understanding of pricing strategies' impact on sales volume, essential for optimising pricing for increased sales and customer acquisition.

Discount effectiveness seems to peak at a moderate discount percentage, with diminishing returns as the discount increases. This suggests an optimal discount rate can be found that balances attractiveness to customers with profitability.

9. Number of orders refunds by month

```r
# Query refunded orders with order dates and extract month directly in SQL
refund_by_month <- dbGetQuery(database, "
  SELECT strftime('%Y-%m', O.order_date) AS month,
         COUNT(*) AS refund_count
  FROM `Order` AS O
  INNER JOIN Shipment AS S ON O.order_number = S.order_number
  WHERE S.refund = 'Yes'
  GROUP BY month
")

# Convert month to Date type
refund_by_month$month <- ymd(paste(refund_by_month$month, "-01", sep = "-"))

# Convert month labels to abbreviated month names
refund_by_month$month <- format(refund_by_month$month, "%b")

# Set the order of months
months_order <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov
refund_by_month$month <- factor(refund_by_month$month, levels = months_order)

# Visualize the number of refunds by month
ggplot(refund_by_month, aes(x = month, y = refund_count)) +
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(x = "Month", y = "Number of Orders Refunds", title = "Refunds by Month") +
  theme_classic()
```



The bar graph displays monthly order refunds, highlighting noticeable trends. February and December exhibit the highest refund counts at four, while March, June, October, and November have the lowest, with only one refund each.

Challenges:

Choosing the right data: Given the large number of data encompassing customer behavior, order, and product details, it becomes a challenging endeavor to distill information that is not only aligned with our analytical objectives but is also precise and readily comprehensible.

Incorrect Join Operations: Using joins to link different tables is quite tricky. It's crucial for us to determine which key to use for the connection. For example, if the 'JOIN' conditions between Order and Shipment tables are specified incorrectly, it could lead to an inaccurate dataset, such as joining on the wrong key or missing the correct join logic.

Visualization Complexity: Designing clear and informative visualizations to present the data effectively can be challenging. Selecting appropriate visualization techniques, formatting charts and graphs, and ensuring visual clarity while conveying complex information require careful attention to detail.

5. Conclusion

In summary, this report delves deeply into the intricacies of managing and analyzing e-commerce data. It starts by laying the groundwork with an Entity-Relationship Diagram (ERD), providing a clear blueprint of how various elements like customers, orders, products, and discounts are interrelated within the database structure. Through meticulous validation processes, the integrity and quality of data are ensured before integration into the database.

Utilizing tools like Python for data generation and R for advanced analysis, the report uncovers valuable insights into customer behavior, product popularity, revenue distribution by country, and the impact of discounts on sales. These insights are crucial for strategic decision-making in areas such as inventory management, marketing optimization, and customer retention strategies. However, the report also acknowledges the challenges inherent in managing e-commerce data, such as database complexity, uncertainty in identifying relationship between elements and lack of expertise issues. Despite these challenges, the meticulous validation and insightful analysis provided in this report offer a solid foundation for informed decision-making and strategic planning in the ever-evolving landscape of digital commerce.

6. Appendix

Appendix 1

```
## Customer

compare_and_update_database_cust <- function(old_csv, new_csv, table_name, primary_key) {
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print("No differences found between the old and new data in Customer Table. No updates n
    return(NULL)
  }
  # Function to check if a datetime is in the desired format ("%Y-%m-%d %H:%M:%S")
  is_datetime_format <- function(datetime_string) {
    tryCatch({
      as.POSIXlt(datetime_string, format = "%Y-%m-%d %H:%M:%S")
      TRUE
    }, error = function(e) {
      FALSE
    })
  }

```

```
# Check if dates are in the desired format, if not, convert them
for (i in 1:nrow(Customer)) {
  if (!is_datetime_format(Customer$date_of_birth[i])) {
    Customer$date_of_birth[i] <- as.POSIXct(Customer$date_of_birth[i], format = "%Y-%m-%d
  }
}

# Data validation for new customer data
validate_customer_data <- function(data) {
  # Check unique customer IDs
  if (length(unique(data$customer_id)) != nrow(data)) {
    stop("Customer ID is not unique.")
  }

  # Check data types for first_name and last_name
  if (!all(sapply(data$first_name, is.character)) || !all(sapply(data$last_name, is.charac
    stop("First name and last name should be character.")
  }

  # Check valid gender values
  valid_genders <- c("Male", "Female", "Other")
  if (any(!data$gender %in% valid_genders)) {
    stop("Gender should be Male, Female, or Other.")
  }

  # Check email format
  if (any(!grepl("^\\S+@\\S+\\.\\S+$", data$email))) {
    stop("Invalid email format")
  }

  # Regular expressions for phone number formats of Belgium, China, France, United Kingdom
  phone_regex <- "^\\(\\+\\d+\\)\\d{10}$"

  # Check phone number format for specific countries
  invalid_phone_indices <- which(!grepl(phone_regex, data$phone))
  if (length(invalid_phone_indices) > 0) {
    stop("Invalid phone numbers.")
  }

  # Regular expressions for zip code formats of Belgium, China, France, United Kingdom, Un
  zip_regex <- c(
    "^[0-9]{4}$",  # Belgium
```

```r
    "^[0-9]{6}$",  # China
    "^[0-9]{5}$",  # France
    "^[A-Z]{2}[0-9]{1,2}[A-Z]? [0-9][A-Z]{2}$",  # United Kingdom
    "^[0-9]{5}-[0-9]{4}$"    # United States
  )

  # Check zip code format for specific countries
  invalid_zip_indices <- which(!grepl(paste(zip_regex, collapse = "|"), data$customer_zip_
  if (length(invalid_zip_indices) > 0) {
    stop("Invalid zip codes.")
  }

  # Check platform values
  valid_platforms <- c("Referral", "Instagram", "Facebook", "Others")
  if (any(!data$platform %in% valid_platforms)) {
    stop("Invalid platform values.")
  }

  return(TRUE)
}

# Validate new customer data
if (!validate_customer_data(added_rows)) {
  return(NULL)
}

# Create a database connection
database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

# Drop existing Customer table if it exists
if (dbExistsTable(database, table_name)) {
  dbExecute(database, paste0("DROP TABLE ", table_name))
}

# Write the new data to the database with append
RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE)
cat("New records added to the Customer table.\n")

# Print information about the last added record
if (nrow(added_rows) > 0) {
  cust_result <- RSQLite::dbGetQuery(database, "SELECT * FROM Customer ORDER BY ROWID DESC
  print(cust_result[c("customer_id", "first_name")])
```

```
    print("Customer data is valid. Data loaded into the database...")
  }

  # Close database connection
  dbDisconnect(database)
}

# Usage example
compare_and_update_database_cust("Dataset/customer.csv", "DatasetTest/customer_data_test_new_

## Seller

compare_and_update_database_seller <- function(old_csv, new_csv, table_name, primary_key) {
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print("No differences found between the old and new data in Seller Table. No updates nee
    return(NULL)
  }

  # Create a database connection
  database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

  # Drop existing Seller table if it exists
  if (dbExistsTable(database, table_name)) {
    dbExecute(database, paste0("DROP TABLE ", table_name))
  }

  # Write the new data to the database with append
  RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE)
  cat("New records added to the Seller table.\n")

  # Print information about the last added record
  if (nrow(added_rows) > 0) {
    seller_result <- RSQLite::dbGetQuery(database, paste0("SELECT * FROM ", table_name, " OR
    print(seller_result[c("seller_id", "company_name")])
  }

  ## Validation of Seller Data
  library(stringr)
  na_sellers <- apply(is.na(added_rows), 2, sum)

  # Ensure "seller_Id" values are unique
  if (length(unique(added_rows$seller_id)) != nrow(added_rows)) {
    print("seller_Id values are not unique.")
  }

  # Check length of "company_name"
  if (any(nchar(added_rows$company_name) > 100)) {
    print("company_name exceeds 100 characters.")
  }

  # Check email format
  invalid_emails <- which(!str_detect(added_rows$supplier_email, "\\b[A-Za-z0-9._%+-]+@[A-Za-
  if (length(invalid_emails) > 0) {
    print("Invalid email addresses:")
    print(added_rows[invalid_emails, ])
  }

  # If no errors are found, print a message indicating that the data is valid
  if (!any(is.na(na_sellers)) &&
      length(unique(added_rows$seller_id)) == nrow(added_rows) &&
      !any(nchar(added_rows$company_name) > 100) &&
      length(invalid_emails) == 0) {
    print("Sellers data is valid. Data loaded into the database...")
    RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE
    # Load the data into the database
  } else {
    print("Sellers data is not valid. Please correct the errors.")
  }

  # Close database connection
  RSQLite::dbDisconnect(database)
}

# Usage example
compare_and_update_database_seller("Dataset/seller.csv", "DatasetTest/seller_data_test_new_r

## Category
```

```r

compare_and_update_database_category <- function(old_csv, new_csv, table_name, primary_key)
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print(paste("No differences found between the old and new data in", table_name, "Table.
    return(NULL)
  }

  # Validation for category data
  na_prod_cat <- apply(is.na(added_rows), 2, sum)

  # Ensure "category_id" values are unique
  if (length(unique(added_rows$category_id)) != nrow(added_rows)) {
    print("category_id values are not unique.")
  }

  # Check length of "cat_name"
  if (any(nchar(added_rows$cat_name) > 255)) {
    print("cat_name exceeds 255 characters.")
  }

  # Check data type of each column
  if (!all(sapply(added_rows$category_id, is.character)) ||
      !all(sapply(added_rows$cat_name, is.character)) ||
      !all(sapply(added_rows$cat_description, is.character))) {
    print("Invalid data type for one or more columns.")
  }

  # If any errors are found, do not proceed with writing to the database
  if (any(is.na(na_prod_cat)) ||
      length(unique(added_rows$category_id)) != nrow(added_rows) ||
      any(nchar(added_rows$cat_name) > 255) ||
      !all(sapply(added_rows$category_id, is.character)) ||
      !all(sapply(added_rows$cat_name, is.character)) ||
      !all(sapply(added_rows$cat_description, is.character))) {
    print(paste(table_name, "data is not valid. Please correct the errors."))
    return(NULL)
  }

  # Create a database connection
  database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

  # Drop existing Category table if it exists
  if (dbExistsTable(database, table_name)) {
    dbExecute(database, paste0("DROP TABLE ", table_name))
  }

  # Write the new data to the database with append
  RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE)
  cat(paste("New records added to the", table_name, "table.\n"))

  # Print information about the last added record
  if (nrow(added_rows) > 0) {
    category_result <- RSQLite::dbGetQuery(database, paste0("SELECT * FROM ", table_name, " (
    print(category_result[c("category_id", "cat_name")])
    print("Category data is valid. Loaded data into the database...")
  }

  # Close database connection
  dbDisconnect(database)
}

compare_and_update_database_category("Dataset/category.csv", "DatasetTest/category_data_no_ne

## Product

compare_and_update_database_prod <- function(old_csv, new_csv, table_name, primary_key) {
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print("No differences found between the old and new data in Product Table. No updates nee
    return(NULL)
  }
```

```

# Function to validate product data
validate_product_data <- function(data) {
  # Function to check if a value is decimal
  valid_decimal <- function(x) {
    !is.na(as.numeric(x))
  }

  # Function to check if a value is an integer
  valid_integer <- function(x) {
    !is.na(as.integer(x))
  }

  # Check for missing values
  na_values <- apply(is.na(data), 2, sum)

  # Ensure "product_id" values are unique
  if (length(unique(data$product_id)) != nrow(data)) {
    stop("product_id values are not unique.")
  }

  # Check length of "product_name"
  if (any(nchar(data$product_name) > 255)) {
    stop("product_name exceeds 255 characters.")
  }

  # Check if inventory and product views are integers
  if (any(!sapply(data$inventory, valid_integer)) || any(!sapply(data$product_views, valid_
    stop("Inventory and product views should be integers.")
  }

  # Check if price and weight are decimal
  if (any(!sapply(data$price, valid_decimal)) || any(!sapply(data$weight, valid_decimal)))
    stop("Price and weight should be decimal values.")
  }

  # If no errors are found, return TRUE
  return(TRUE)
}

# Validate new product data
if (!validate_product_data(added_rows)) {
```

```r
    return(NULL)
  }

  # Create a database connection
  database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

  # Drop existing Product table if it exists
  if (dbExistsTable(database, table_name)) {
    dbExecute(database, paste0("DROP TABLE ", table_name))
  }

  # Write the new data to the database with append
  RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE)
  cat("New records added to the Product table.\n")

  # Print information about the last added record
  if (nrow(added_rows) > 0) {
    prod_result <- RSQLite::dbGetQuery(database, "SELECT * FROM Product ORDER BY ROWID DESC L
    print(prod_result[c("product_id", "product_name")])
    print("Product data is valid. Data loaded into the database...")
  }

  # Close database connection
  dbDisconnect(database)
}
# Usage example
compare_and_update_database_prod("Dataset/product.csv", "DatasetTest/product_data_test_no_new

## Discount

compare_and_update_database_discount <- function(old_csv, new_csv, table_name, primary_key) {
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print("No differences found between the old and new data in Discount Table. No updates ne
    return(NULL)
  }
```

```r

# Validation for discount data
## Function to check if date is in the desired format
is_datetime_format <- function(x) {
  tryCatch({
    as.POSIXlt(x, format = "%Y-%m-%d %H:%M:%S")
    TRUE
  }, error = function(e) {
    FALSE
  })
}

## Convert discount_start_date and discount_end_date to desired format if not already in t
if (!all(sapply(added_rows$discount_start_date, is_datetime_format))) {
  added_rows$discount_start_date <- as.POSIXlt(added_rows$discount_start_date, format = "%
}

if (!all(sapply(added_rows$discount_end_date, is_datetime_format))) {
  added_rows$discount_end_date <- as.POSIXlt(added_rows$discount_end_date, format = "%Y-%m
}

## Check for missing values in Discount dataframe
na_disc <- apply(is.na(added_rows), 2, sum)

## Validate discount_percentage, discount_start_date, and discount_end_date data types
valid_decimal <- function(x) {
  !is.na(as.numeric(x))
}

valid_datetime <- function(x) {
  !is.na(as.POSIXlt(x))
}

## Check discount percentage range (assuming it's between 0 and 100)
if (any(added_rows$discount_percentage < 0 | added_rows$discount_percentage > 100) ||
    !all(sapply(added_rows$discount_percentage, valid_decimal))) {
  print("Invalid discount percentage.")
}

## Check discount dates
if (any(added_rows$discount_start_date >= added_rows$discount_end_date) ||
    !all(sapply(added_rows$discount_start_date, valid_datetime)) ||
      !all(sapply(added_rows$discount_end_date, valid_datetime))) {
    print("Discount start date should be before the end date.")
  }

  ## Check if discount_id is unique
  if (any(duplicated(added_rows$discount_id))) {
    print("Duplicate discount IDs found.")
  }

  ## Check if product_id exists in Product table
  if (any(!added_rows$product_id %in% Product$product_id)) {
    print("Invalid product IDs. Some product IDs do not exist in the Product table.")
  }

  ## If no errors are found, print a message indicating that the data is valid
  if (!any(is.na(na_disc)) &&
      all(added_rows$discount_percentage >= 0 & added_rows$discount_percentage <= 100) &&
      all(added_rows$discount_start_date < added_rows$discount_end_date) &&
      !any(duplicated(added_rows$discount_id)) &&
      all(added_rows$product_id %in% Product$product_id) &&
      all(sapply(added_rows$discount_percentage, valid_decimal)) &&
      all(sapply(added_rows$discount_start_date, valid_datetime)) &&
      all(sapply(added_rows$discount_end_date, valid_datetime))) {
    print("Discount data is valid. Loaded data into the database...")

    database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

    # Drop existing Discount table if it exists
    if (dbExistsTable(database, table_name)) {
      dbExecute(database, paste0("DROP TABLE ", table_name))
    }

    RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE
    cat("New records added to the Discount table.\n")

    discount_result <- RSQLite::dbGetQuery(database, paste0("SELECT * FROM ", table_name, " 
    print(discount_result[c("discount_id", "discount_percentage")])
    dbDisconnect(database)
  } else {
    print("Data is not valid. Please correct the errors.")
  }
}
```

```r

# Usage example
compare_and_update_database_discount("Dataset/discount.csv", "DatasetTest/discount_data_no_n

## Order

compare_and_update_database_order <- function(old_csv, new_csv, table_name, primary_key) {
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print("No differences found between the old and new data in Order Table. No updates need
    return(NULL)
  }

  # Validation for order data
  na_order <- apply(is.na(added_rows), 2, sum)

  # Check quantity (assuming it should be a positive integer)
  if (any(added_rows$quantity <= 0)) {
    print("Invalid quantity.")
  }

  # Check customer rating (assuming it should be between 1 and 5)
  if (any(added_rows$customer_rating < 1 | added_rows$customer_rating > 5)) {
    print("Invalid customer rating.")
  }

  # Check if product_id exists in Product table
  if (any(!added_rows$product_id %in% Product$product_id)) {
    print("Invalid product IDs. Some product IDs do not exist in the Product table.")
  }

  # Check if customer_id exists in Customer table
  if (any(!added_rows$customer_id %in% Customer$customer_id)) {
    print("Invalid customer IDs. Some customer IDs do not exist in the Customer table.")
  }

```

```r
# Check if shipment_id exists in Shipment table
if (any(!added_rows$shipment_id %in% Shipment$shipment_id)) {
  print("Invalid shipment IDs. Some shipment IDs do not exist in the Shipment table.")
}

# Check uniqueness based on primary key (order_number, customer_id, product_id)
if (any(duplicated(added_rows[c("order_number", "customer_id", "product_id")]))) {
  print("Duplicate records found based on order_number, customer_id, and product_id.")
}

# Check order date format and range
if (any(!is_datetime_format(Order$order_date))) {
  # Convert order date to the desired format if not already
  Order$order_date <- as.POSIXct(Order$order_date, format = "%Y-%m-%d %H:%M:%S", tz = "UTC
}

# If any errors are found, do not proceed with writing to the database
if (any(is.na(na_order)) ||
    any(added_rows$quantity <= 0) ||
    any(added_rows$customer_rating < 1 | added_rows$customer_rating > 5) ||
    any(!added_rows$product_id %in% Product$product_id) ||
    any(!added_rows$customer_id %in% Customer$customer_id) ||
    any(!added_rows$shipment_id %in% Shipment$shipment_id) ||
    any(duplicated(added_rows[c("order_number", "customer_id", "product_id")])) &&
    all(is_datetime_format(Order$order_date))) {
  print("Order data is not valid. Please correct the errors.")
  return(NULL)
}

# Create a database connection
database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

# Drop existing Order table if it exists
if (dbExistsTable(con, table_name)) {
  dbExecute(con, paste0("DROP TABLE ", table_name))
}

# Write the new data to the database with append
RSQLite::dbWriteTable(con, table_name, added_rows, append = TRUE, row.names = FALSE)
cat("New records added to the Order table.\n")

# Print information about the last added record
```

```
  if (nrow(added_rows) > 0) {
    order_result <- RSQLite::dbGetQuery(database, paste0("SELECT * FROM ", table_name, " ORD
    print(order_result[c("order_number", "product_id")])
    print("Order data is valid. Loaded data into the database...")
  }

  # Close database connection
  dbDisconnect(database)
}

# Usage example
compare_and_update_database_order("Dataset/order.csv", "DatasetTest/order_data_test_no_new.c


## Shipment Data

compare_and_update_database_shipment <- function(old_csv, new_csv, table_name, primary_key) 
  # Load old and new data
  old_data <- read.csv(old_csv)
  new_data <- read.csv(new_csv)

  # Check for differences
  added_rows <- anti_join(new_data, old_data, by = primary_key)

  if (nrow(added_rows) == 0) {
    print(paste("No differences found between the old and new data in", table_name, "Table. 
    return(NULL)
  }

  # Validation for Shipment Data
  na_shipment <- sapply(added_rows, function(x) sum(is.na(x)))

  # Ensure "shipment_id" values are unique
  if (length(unique(added_rows$shipment_id)) != nrow(added_rows)) {
    print("shipment_id values are not unique.")
  }

  # Validate "refund" column
  valid_refunds <- c("Yes", "No")
  if (!all(added_rows$refund %in% valid_refunds)) {
    print("Invalid values in the 'refund' column.")
  }
```

```

# Validate "shipment_delay_days" and "shipment_cost" columns
if (any(added_rows$shipment_delay_days <= 0) || any(added_rows$shipment_cost <= 0)) {
  print("shipment_delay_days and shipment_cost should be positive numbers.")
}

# Ensure that "shipment_delay_days" is an integer
if (any(!as.integer(added_rows$shipment_delay_days) == added_rows$shipment_delay_days)) {
  print("shipment_delay_days should be integers.")
}

# Ensure that all "order_number" values exist in the "Order" table
order_numbers <- unique(added_rows$order_number)
if (!all(order_numbers %in% Order$order_number)) {
  print("Some order numbers do not exist in the 'Order' table.")
}

# If any errors are found, do not proceed with writing to the database
if (any(na_shipment != 0) ||
    length(unique(added_rows$shipment_id)) != nrow(added_rows) ||
    !all(added_rows$refund %in% valid_refunds) ||
    any(added_rows$shipment_delay_days <= 0) || any(added_rows$shipment_cost <= 0) ||
    any(!as.integer(added_rows$shipment_delay_days) == added_rows$shipment_delay_days) ||
    !all(order_numbers %in% Order$order_number)) {
  print(paste(table_name, "data is not valid. Please correct the errors."))
  return(NULL)
}

# Create a database connection
database <- RSQLite::dbConnect(RSQLite::SQLite(), dbname = 'Ecommerce.db')

# Drop existing Shipment table if it exists
if (dbExistsTable(database, table_name)) {
  dbExecute(database, paste0("DROP TABLE ", table_name))
  cat(paste("Existing", table_name, "table dropped.\n"))
}

# Write the new data to the database with append
RSQLite::dbWriteTable(database, table_name, added_rows, append = TRUE, row.names = FALSE)
cat(paste("New records added to the", table_name, "table.\n"))

# Print information about the last added record
```

```
  if (nrow(added_rows) > 0) {
    shipment_result <- RSQLite::dbGetQuery(database, paste0("SELECT * FROM ", table_name, " (
    print(shipment_result[c("shipment_id", "order_number")])
  }

  # Close database connection
  dbDisconnect(database)
}

# Usage example
compare_and_update_database_shipment("Dataset/shipment.csv", "DatasetTest/shipment_data_no_n
```

Appendix 2

```
knitr::opts_chunk$set(python.reticulate = FALSE)
# install faker library
# pip install Faker

# csv library
import csv

# random library
import random

# datetime
from datetime import datetime, timedelta

# inititalise faker generator
from faker import Faker
fake = Faker()


# %% customer
country_codes = ['+44','+1', '+32', '+33', '+86'] #usa/canada belgium france china

# Function to get city information based on country code
def get_country_info_cust(country_code):
    if country_code == '+44':
        return 'United Kingdom'
    elif country_code == '+1':
        return 'United States'
    elif country_code == '+32':
        return 'Belgium'
    elif country_code == '+33':
        return 'France'
    elif country_code == '+86':
        return 'China'
    else:
        return 'Unknown'

def customer(filename, num_customers=500):
    fieldnames = ['customer_id', 'first_name', 'last_name', 'gender', 'date_of_birth', 'emai

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Generate and write fake data to the CSV file
        for customer_id_num in range(1, num_customers + 1):
            customer_id = f'c{customer_id_num:05}'  # Format customer_id as 'c' followed by
            gender = fake.random_element(elements=('Male', 'Female', 'Other'))
            first_name = fake.first_name_male() if gender == 'Male' else fake.first_name_fema
            last_name = fake.last_name()
            email = f"{first_name.lower()}_{last_name.lower()}@gmail.com"
            country_code = fake.random_element(elements=country_codes)
            phone = f"({country_code}){fake.random_number(digits=10, fix_len=True)}"
            customer_street = fake.street_address()
            customer_country = get_country_info_cust(country_code)
            customer_zip_code = fake.zipcode()
            date_of_birth = fake.date_of_birth(minimum_age=35, maximum_age=60).strftime('%d/
            platform = fake.random_element(elements=('Facebook', 'Instagram', 'Referral', 'O

            csvwriter.writerow({
                'customer_id': customer_id,
                'first_name': first_name,
                'last_name': last_name,
                'gender': gender,
                'date_of_birth': date_of_birth,
                'email': email,
                'phone': phone,
                'customer_street': customer_street,
                'customer_country': customer_country,
                'customer_zip_code': customer_zip_code,
                'platform': platform
            })

# Call the function to generate and save fake customer data to a CSV file
customer('customer.csv', num_customers=500)

# %% seller entity


country_codes = ['+44', '+1', '+32', '+33', '+86']  # United Kingdom, USA/Canada, Belgium, F

# Function to get city information based on country code
def get_country_info_seller(country_code):
    if country_code == '+44':
        return 'United Kingdom'
    elif country_code == '+1':
        return 'United States'  # You can add more cities for USA/Canada
    elif country_code == '+32':
        return 'Belgium'
    elif country_code == '+33':
        return 'France'
    elif country_code == '+86':
        return 'China'
    else:
        return 'Unknown'

# Function to generate fake data and save it to a CSV file
def seller(filename, num_records=500):
    fieldnames = ['seller_id', 'company_name', 'supplier_phone', 'supplier_email', 'seller_s

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Generate and write fake data to the CSV file
        for seller_id_num in range(1, num_records + 1):
            seller_id = f's{seller_id_num:05}'
            company_name = fake.company().replace(',', '-High')
            country_code = random.choice(country_codes)
            supplier_phone = f"({country_code}){fake.random_number(digits=10, fix_len=True)}"
            supplier_email = f"{fake.first_name().lower()}@{company_name.split()[0].lower()}
            seller_street = fake.street_address()
            seller_country = get_country_info_seller(country_code)
            seller_zip_code = fake.zipcode()

            csvwriter.writerow({
                'seller_id': seller_id,
                'company_name': company_name,
                'supplier_phone': supplier_phone,
                'supplier_email': supplier_email,
                'seller_street': seller_street,
                'seller_country': seller_country,
                'seller_zip_code': seller_zip_code
            })

# Call the function to generate and save fake data to a CSV file (500 records)
seller('seller.csv', num_records=500)


# %% product category

category_descriptions = {
    "Electronics": "Explore the latest in cutting-edge technology with our electronic gadgets
    "Home and Kitchen": "Enhance your living spaces with our stylish and functional home and
    "Sports and Outdoors": "Gear up for outdoor adventures with our high-quality sports and
    "Clothing and Accessories": "Stay on trend with our fashionable clothing and accessories
    "Beauty and Personal Care": "Discover a world of beauty and personal care products to enh
    "Health and Wellness": "Prioritize your well-being with our selection of health and welln
    "Toys and Games": "Entertain and educate with our fun and exciting toys and games for all
    "Automotive": "Keep your vehicle running smoothly with our automotive parts and accessori
    "Books and Literature": "Immerse yourself in captivating stories and knowledge with our
    "Garden and Outdoor": "Create a lush and inviting outdoor space with our gardening and ou
    "Sportswear": "Elevate your active lifestyle with our stylish and high-performance sports
    "Jewelry": "Adorn yourself with exquisite jewelry that complements your unique style.",
    "Skincare": "Indulge in luxurious skincare products designed to nourish and rejuvenate yo
    "Health Supplements": "Boost your health and vitality with our premium selection of healt
    "Board Games": "Gather friends and family for memorable game nights with our exciting boa
    "Car Engine Products": "Enhance the performance and longevity of your vehicle with our hi
    "Gardening Tools": "Transform your garden into a vibrant oasis with our premium Gardening
```

```python
}


# Function to generate categories and save them to a CSV file
def category(filename, num_categories=17):
    fieldnames = ['category_id','p_category_id', 'cat_name', 'cat_description']

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Mapping of cat_name to p_category_id
        p_category_mapping = {
            "Sportswear": "pc01",
            "Jewelry": "pc02",
            "Skincare": "pc03",
            "Health Supplements": "pc04",
            "Board Games": "pc05",
            "Car Engine Products": "pc06",
            "Gardening Tools": "pc07"
        }

        # Generate and write category data to the CSV file
        for category_id_num, (cat_name, cat_description) in enumerate(category_descriptions.
            category_id = f'c{category_id_num:02}'  # Format category_id as 'c' followed by

            # Get the corresponding p_category_id based on cat_name
            p_category_id = p_category_mapping.get(cat_name, None)

            # Set p_category_id to 'NULL' if not found in the mapping
            p_category_id = 'NULL' if p_category_id is None else p_category_id

            csvwriter.writerow({
                'category_id': category_id,
                'p_category_id': p_category_id,
                'cat_name': cat_name,
                'cat_description': cat_description
            })

# Call the function to generate and save category data to a CSV file
```

```
category('category.csv', num_categories=17)


# %% products

product_list = {
    "Electronics": [
        {"name": "SmartHome Hub", "description": "A central control hub for all your smart h
        {"name": "Wireless Bluetooth Earbuds", "description": "Enjoy wireless freedom with h
        {"name": "4K Ultra HD Smart TV", "description": "Bring the cinema experience home wi
    ],
    "Home and Kitchen": [
        {"name": "Smart Coffee Maker", "description": "Brew your favorite coffee with smart 
        {"name": "Non-Stick Cookware Set", "description": "Premium cookware for your kitchen
        {"name": "Smart Mini Refrigerator", "description": "Keep your food fresh and organiz
    ],
    "Sports and Outdoors": [
        {"name": "Fitness Tracker", "description": "Track your fitness activities with this 
        {"name": "Camping Tent", "description": "Explore the outdoors with a durable camping
        {"name": "Waterproof Hiking Boots", "description": "Conquer any trail with our Water
    ],

    "Clothing and Accessories": [
        {"name": "Classic Leather Jacket", "description": "Make a statement with our Classic
        {"name": "Stylish Sunglasses", "description": "Shield your eyes in style with our St
        {"name": "Cozy Knit Sweater", "description": "Embrace warmth and comfort with our Co
    ],
    "Beauty and Personal Care": [
        {"name": "Luxury Concealer", "description": "Indulge in the ultimate concealer exper
        {"name": "Professional Hair Dryer", "description": "Achieve salon-quality results at
        {"name": "Relaxing Aromatherapy Candle", "description": "Unwind and de-stress with o
    ],
    "Health and Wellness": [
        {"name": "Fitness Tracker Watch", "description": "Take charge of your health journey
        {"name": "Organic Superfood Blend", "description": "Boost your nutrition with our Or
        {"name": "Meditation Pillow Set", "description": "Find tranquility and peace with ou
    ],
    "Toys and Games": [
        {"name": "Interactive Robot Toy", "description": "Spark creativity and play with our
        {"name": "Board Game Collection", "description": "Gather friends and family for game
        {"name": "Kid's Building Blocks Set", "description": "Foster imagination and creativ
    ],
    "Automotive": [
        {"name": "Car Care Kit", "description": "Keep your vehicle in top condition with our
        {"name": "Portable Tire Inflator", "description": "Stay prepared on the road with ou
        {"name": "HD Dash Cam", "description": "Capture every moment on the road with our HD
    ],
    "Books and Literature": [
        {"name": "Bestselling Mystery Novel", "description": "Dive into a gripping mystery w
        {"name": "Personal Development Guide", "description": "Embark on a journey of self-d
        {"name": "Illustrated Children's Book", "description": "Spark imagination and joy wi
    ],
    "Garden and Outdoor": [
        {"name": "Solar-Powered Garden Lights", "description": "Illuminate your garden with
        {"name": "Folding Outdoor Lounge Chair", "description": "Relax in style with our Fol
        {"name": "Weather-Resistant Patio Umbrella", "description": "Enjoy outdoor living to
    ],
    "Sportswear": [
        {"name": "High-Performance Running Shoes", "description": "Achieve your fitness goal
        {"name": "Moisture-Wicking Athletic Shirt", "description": "Stay cool and dry during
        {"name": "Compression Fit Leggings", "description": "Enhance your performance with o
    ],
   "Jewelry": [
        {"name": "Elegant Diamond Necklace", "description": "Adorn yourself with our Elegant
        {"name": "Stylish Silver Bracelet", "description": "Complete your look with our Styl
        {"name": "Classic Gold Hoop Earrings", "description": "Make a statement with our Cla
    ],
   "Skincare": [
        {"name": "Hydrating Facial Moisturizer", "description": "Nourish and hydrate your sk
        {"name": "Gentle Cleansing Foam", "description": "Achieve a clean and refreshed compl
        {"name": "Anti-Aging Serum", "description": "Turn back the clock with our Anti-Aging
    ],
   "Health Supplements": [
        {"name": "Multivitamin Capsules", "description": "Support your overall health with o
        {"name": "Omega-3 Fish Oil Softgels", "description": "Boost heart health with our Om
        {"name": "Immune System Booster Tablets", "description": "Enhance your immune system
   ],
   "Board Games": [
        {"name": "Strategic Card Game", "description": "Engage in thrilling battles of strat
        {"name": "Classic Chess Set", "description": "Exercise your mind with our Classic Ch
        {"name": "Family-Friendly Board Game", "description": "Create lasting memories with
   ],
    "Car Engine Products": [
        {"name": "Performance Gear Oil", "description": "Enhance your car's performance with
        {"name": "Engine Degreaser Spray", "description": "Keep your engine clean and runnin
        {"name": "Heavy-Duty Transmission Fluid", "description": "Ensure optimal transmissio
    ],
    "Gardening Tools": [
        {"name": "Premium Garden Pruner", "description": "Achieve precision in your gardenin
        {"name": "Durable Garden Trowel", "description": "Dig, plant, and transplant with ea
        {"name": "Extendable Telescopic Lopper", "description": "Reach high branches and tri
    ]


}
def product(filename, num_products=51):
    fieldnames = ['product_id', 'product_name', 'price', 'product_description', 'inventory',

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Generate and write product data to the CSV file
        product_id_num = 1
        product_names_used = set()

        # List to store 10 random seller_ids for duplication
        random_seller_ids = random.sample(range(1, 501), 10)

        for category, products in product_list.items():
            category_id = f'c{(product_id_num - 1) // 3 + 1:02}'  # Calculate category_id ba

            for product_data in products:
                product_name = product_data["name"]
                product_description = product_data["description"]

                # Ensure no duplicates in product names
                while product_name in product_names_used:
                    product_name = fake.word() + ' ' + fake.word()

                product_names_used.add(product_name)

                product_id = f'p{product_id_num:03}'  # Format product_id as 'p' followed by
                price = round(random.uniform(1, 150), 1)  # Use round to ensure two decimal
                inventory = fake.random_int(min=1, max=100)  # Random inventory between 1 an
                weight = round(random.uniform(1.00, 10.00), 2)  # Use random.uniform for wei
                seller_id = f's{int((product_id_num - 1) / 2) + 1:05}' if product_id_num <=
                product_views = fake.random_int(min=500, max=1000)  # Random product views b

                # Assign category_id based on the pattern
                csvwriter.writerow({
                    'product_id': product_id,
                    'product_name': product_name,
                    'price': price,
                    'product_description': product_description,
                    'inventory': inventory,
                    'weight': weight,
                    'category_id': category_id,
                    'seller_id': seller_id,
                    'product_views': product_views
                })

                product_id_num += 1

# Call the function to generate and save product data to a CSV file
product('product.csv', num_products=51)


# %% discount
def discount(filename, num_discounts=500):
    fieldnames = ['discount_id', 'discount_percentage', 'discount_start_date', 'discount_end

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Generate and write discount data to the CSV file
        for discount_id_num in range(1, num_discounts + 1):
            discount_id = f'd{discount_id_num:05}'  # Format discount_id as 'd' followed by
            discount_percentage = min(round(fake.random.randint(1, 15) * 5 / 100, 2), 0.8)
            start_date = fake.date_between(start_date='-1y', end_date='now')  # Start date w
            end_date = start_date + timedelta(days=30)  # End date is 1 month after the star

            # Assign product_id with some random duplication for the first 50 to 100 product
```

```python
            if 50 <= discount_id_num <= 100:
                product_id = f'p{random.randint(20, 30):03}'
            else:
                product_id = f'p{random.randint(1, 30):03}'

            csvwriter.writerow({
                'discount_id': discount_id,
                'discount_percentage': discount_percentage,
                'discount_start_date': start_date.strftime('%Y-%m-%d'),
                'discount_end_date': end_date.strftime('%Y-%m-%d'),
                'product_id': product_id
            })

# Call the function to generate and save discount data to a CSV file
discount('discount.csv', num_discounts=500)

# %% order

general_reviews = [
    "Great product! Very satisfied with my purchase.",
    "Highly recommend this item. Excellent quality.",
    "Good value for money. Happy with my choice.",
    "Exactly as described. No complaints here.",
    "Impressed with the functionality. Works well.",
    "Smooth transaction. Quick delivery and good packaging.",
    "Nice product. Met my expectations.",
    "Easy to use and durable. Very pleased.",
    "Would buy again. Reliable and efficient.",
    "Overall, a positive experience with this product."
]

# Function to generate order data
def generate_order_data(filename, num_orders=500):
    fieldnames = ['order_number', 'payment_method', 'order_date', 'quantity', 'review', 'cus

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Initialize variables to keep track of the last order_number and payment_method
        last_order_number = None
        last_payment_method = None

        # Generate unique product IDs for the first 50 orders
        unique_product_ids = set(f'p{i:03}' for i in range(1, 52))

        # Generate and write order data to the CSV file
        for order_number_num in range(1, num_orders + 1):
            # Generate order_number and customer_id based on the pattern for the first 100 o
            if order_number_num <= 100:
                order_number = f'on{(order_number_num - 1) // 2 + 1:05}'
                customer_id = f'c{(order_number_num - 1) // 2 + 1:05}'

                if unique_product_ids:
                    # Use unique product IDs for the first 50 orders
                    product_id = unique_product_ids.pop()
                else:
                    # If the set is empty, generate random product IDs
                    product_id = f'p{random.randint(1, 51):03}'
            else:
                order_number = f'on{order_number_num - 50:05}' # Start from previous order nu
                customer_id = f'c{random.randint(1, 500):05}'

                # Generate random product IDs for orders after the first 50 orders
                product_id = f'p{random.randint(1, 51):03}'

            quantity = fake.random_int(min=1, max=3)  # Random quantity between 1 and 3
            review = random.choice(general_reviews) if fake.boolean(chance_of_getting_true=6

            # Order_date is the same as order_number
            order_date = fake.date_between(start_date='-1y', end_date='now').strftime('%Y-%m-

            # Payment method takes reference from order_number for cash transactions
            if last_order_number != order_number:
                last_order_number = order_number
                last_payment_method = fake.random_element(elements=('Credit Card', 'PayPal',

            payment_method = last_payment_method

            # Generate shipment_id_num based on order_number
            shipment_id = f'sh{order_number[2:]}'

            # Generate customer_rating
            customer_rating = random.randint(1, 5)

            csvwriter.writerow({
                'order_number': order_number,
                'payment_method': payment_method,
                'order_date': order_date,
                'quantity': quantity,
                'review': review,
                'customer_id': customer_id,
                'product_id': product_id,
                'shipment_id': shipment_id,
                'customer_rating': customer_rating
            })

# Call the function to generate and save order data to a CSV file
generate_order_data('order.csv', num_orders=500)


# %% shipment
def shipment(filename, num_shipments=450):
    fieldnames = ['shipment_id', 'shipment_delay_days', 'shipment_cost', 'order_number', 're

    with open(filename, 'w', newline='') as csvfile:
        csvwriter = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the header
        csvwriter.writeheader()

        # Generate and write shipment data to the CSV file
        for shipment_id_num in range(1, num_shipments + 1):
            shipment_id = f'sh{shipment_id_num:05}'  # Format shipment_id as 'sh' followed b

            # Reference order_number from the existing order data
            order_number = f'on{shipment_id_num:05}'  # Assuming order_number follows the sa

            shipment_delay_days = fake.random_int(min=1, max=3)
            shipment_cost = round(random.uniform(1, 4), 1)

            # 5% chance of 'Yes', 95% chance of 'No'
            refund = 'Yes' if fake.random_int(min=1, max=100) <= 5 else 'No'

            csvwriter.writerow({
                'shipment_id': shipment_id,
                'shipment_delay_days': shipment_delay_days,
                'shipment_cost': shipment_cost,
                'order_number': order_number,
                'refund': refund
            })

# Call the function to generate and save shipment data to a CSV file
shipment('shipment.csv', num_shipments=450)
```