Report

Group 14

Table of contents

Masters Programmes: Group Assignment Cover Sheet 1
Part 1: Introduction 3
Part 2: Database Design and Implementation 4
2.1 E-R Diagram Design 4
2.2 SQL Database Schema Creation 6
Part 3: Data Generation and Validation 10
3.1 Data Generation 10
3.2. Data Import and Quality Assurance 12
Data Validation 12
3.3 Data Import 19
Part 4: Data Pipeline Generation 22
4.1 GitHub Repository and Workflow Setup 22
4.2 Github Actions for Continuous Integration 23
4.3 Navigating Challenges 23
Part 5: Data Analysis and Reporting with Quarto in R 23
5.1 Advance Data Analysis in R 23
Part 6: Conclusion 30

Masters Programmes: Group Assignment Cover Sheet

Student Numbers: Please list numbers of all group members	**5528636,** **5503558,** **5586034** **5576106,** **5562516,** **5521398**
Module Code:	**IB9HP0**

Module Title:	**Data Management**
Submission Deadline:	**20/03/2024**
Date Submitted:	**20/03/2024**
Word Count:	**1995**
Number of Pages:	**30**
Question Attempted: *(question number/title, or description of assignment)*	**40% Group Assignment**
Have you used Artificial Intelligence (AI) in any part of this assignment?	**Yes**

Part 1: Introduction

Our goal was to prepare an e-commerce database, starting with crafting an Entity-Relationship diagram to blueprint our structure. We transformed the diagram into entities using Data Definition Language, laying the groundwork for our database. Synthetic data was then generated and populated into these entities using the Extract, Transform, Load methodology. This involved extracting data from our synthetic datasets, transforming them to fit our schema, and loading it into the respective entities. Our journey also included deep dives into data analysis, uncovering insights that could shape e-commerce strategies.

Part 2: Database Design and Implementation

2.1 E-R Diagram Design

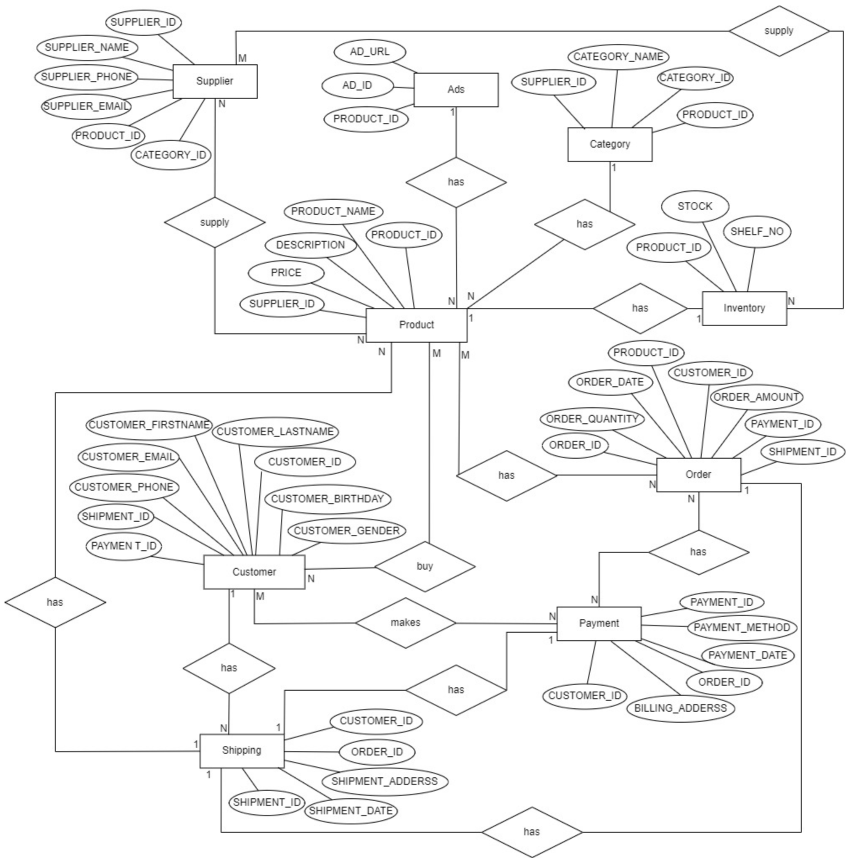


Figure 1: **Initial E-R Diagram**

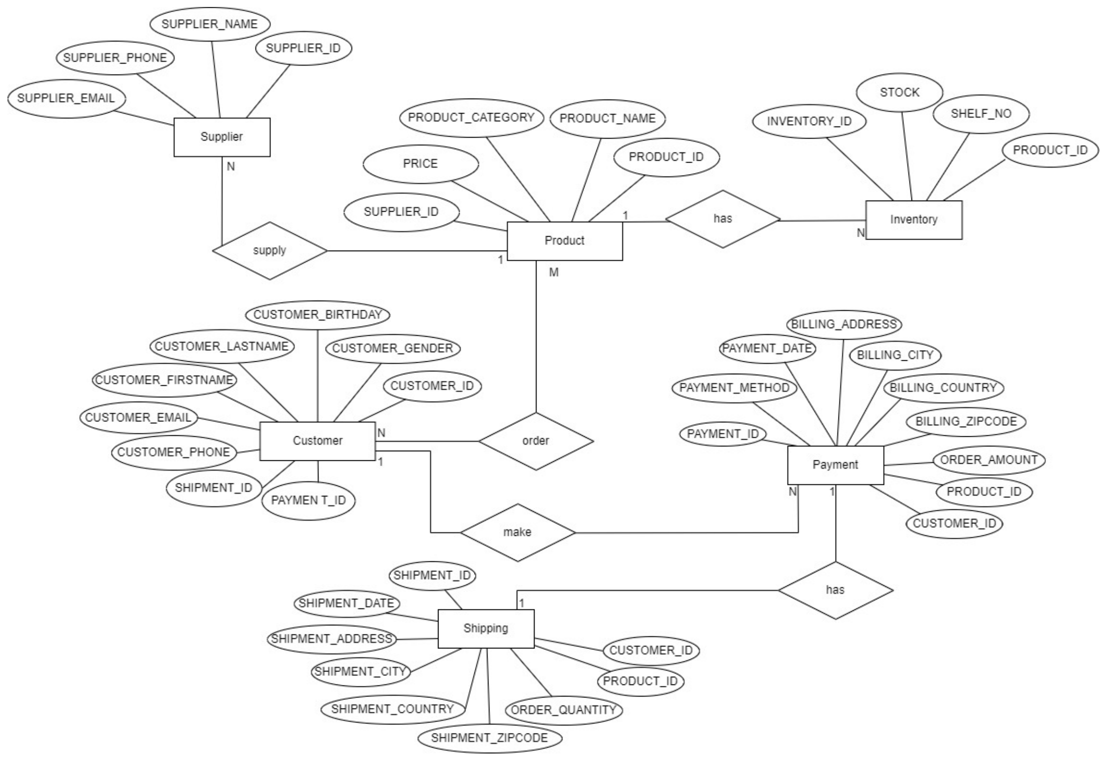


Figure 2: **Final E-R Diagram**

The initial Entity-Relationship (E-R) diagram contained nine entities, interconnected by eleven relationships (Figure 1).

However, deeper examination revealed shortcomings in the initial E-R diagram. When converting it into the schema, the database was incompatible with the third normal form (3NF). Hence, a series of adjustments were made, such as adding and removing entities.

To improve structural integrity, we eliminated the Order, Category, and Ads entities. Order was transformed into a relationship between Customer and Product whereas Category entity was transformed to an attribute under the Product entity to make the synthetic data more realistic. Ads were removed to simplify data generation.

Additionally, several looping relationships were detected. For example, as seen in Figure 1, we included a looping relationship between Supplier, Product and Inventory. This led to unnecessary duplicates. Therefore, we eliminated the direct relationship between Supplier and Inventory. PRODUCT_ID was then used as a linkage. Similarly, a loop between Customers, Payment, and Shipping was resolved.

Finally, non-atomic attributes like SHIPMENT_ADDRESS and DESCRIPTION were addressed. To enhance granularity, more attributes for city, country, and zip-code were added. Moreover, DESCRIPTION was removed due to redundancy with PRODUCT_CATEGORY.

PRODUCT_ID was also removed from the Supplier entity as the SUPPLIER_ID was already existing in the Product entity. This enabled us to use Supplier as a starting point for the schema creation. Lastly, INVENTORY_ID was created since it was missing from the Inventory entity.

Following these adjustments, a final E-R diagram was designed (Figure 2) to accommodate a database complying with 3NF. The final diagram comprises six core entities interconnected by five relationships.

2.2 SQL Database Schema Creation

In the SQL database schema creation phase, deficiencies in normalization were handled by removing unnecessary entities, redundant relationships, and non-atomic attributes. Consequently, the final schema consists of six tables, each with specified data types for attributes and columns (Figure 4). For example, attributes like SUPPLIER_NAME were designed as text, PRICE as numeric, and CUSTOMER_BIRTHDAY as date. This helps in organizing and managing the data while maintaining consistency and integrity.

The assumptions used for the final E-R diagram and Schema:

1. A single product may have multiple suppliers.

2. Products can be stored on multiple shelves depending on quantity.

3. Customers can order multiple products, and products can be bought by several customers.

4. A customer can make multiple payments, each associated with one customer.

5. Each shipment is associated with only one payment.

All relationships between entities are illustrated in Figure 3 and 4.

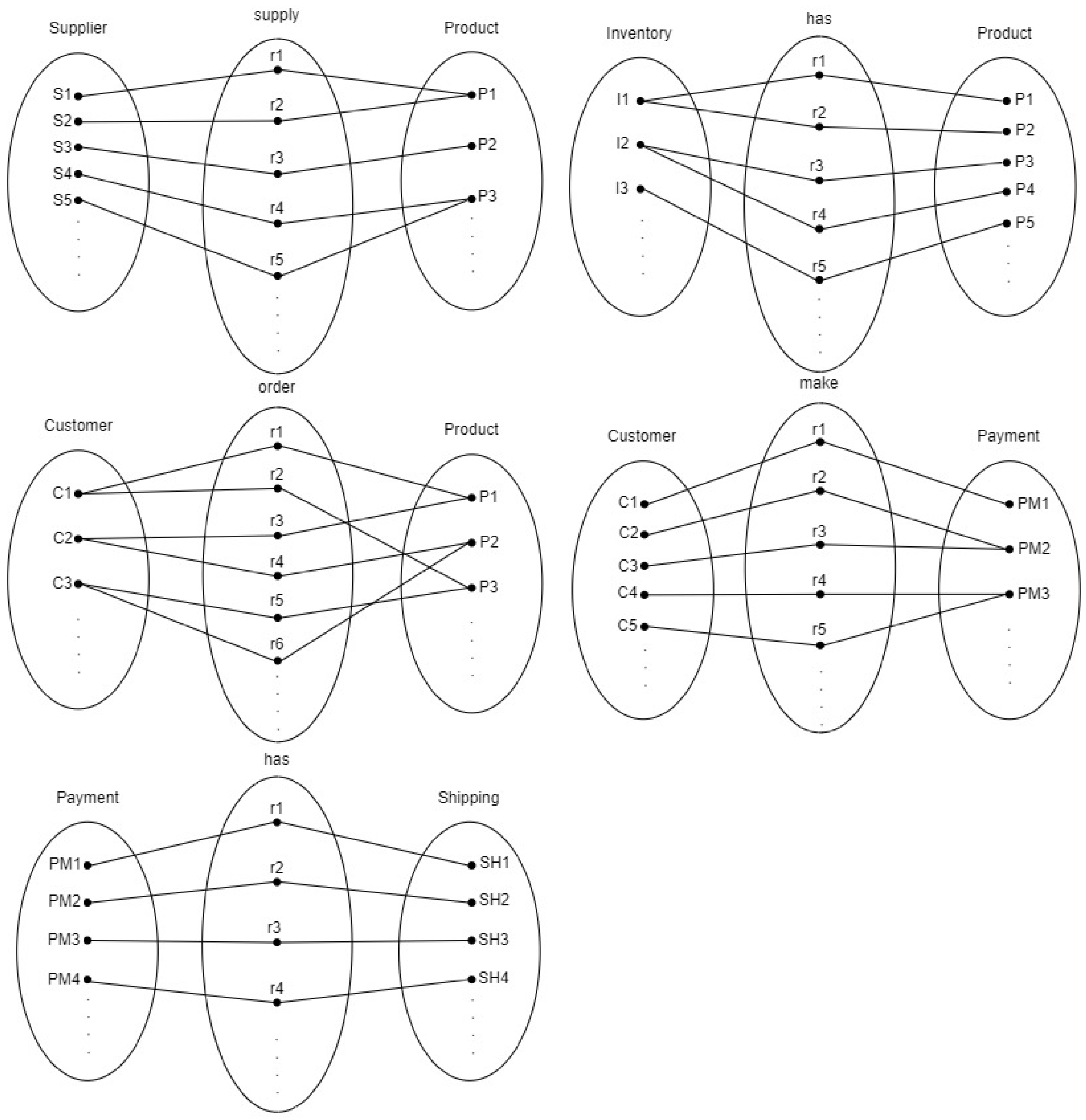
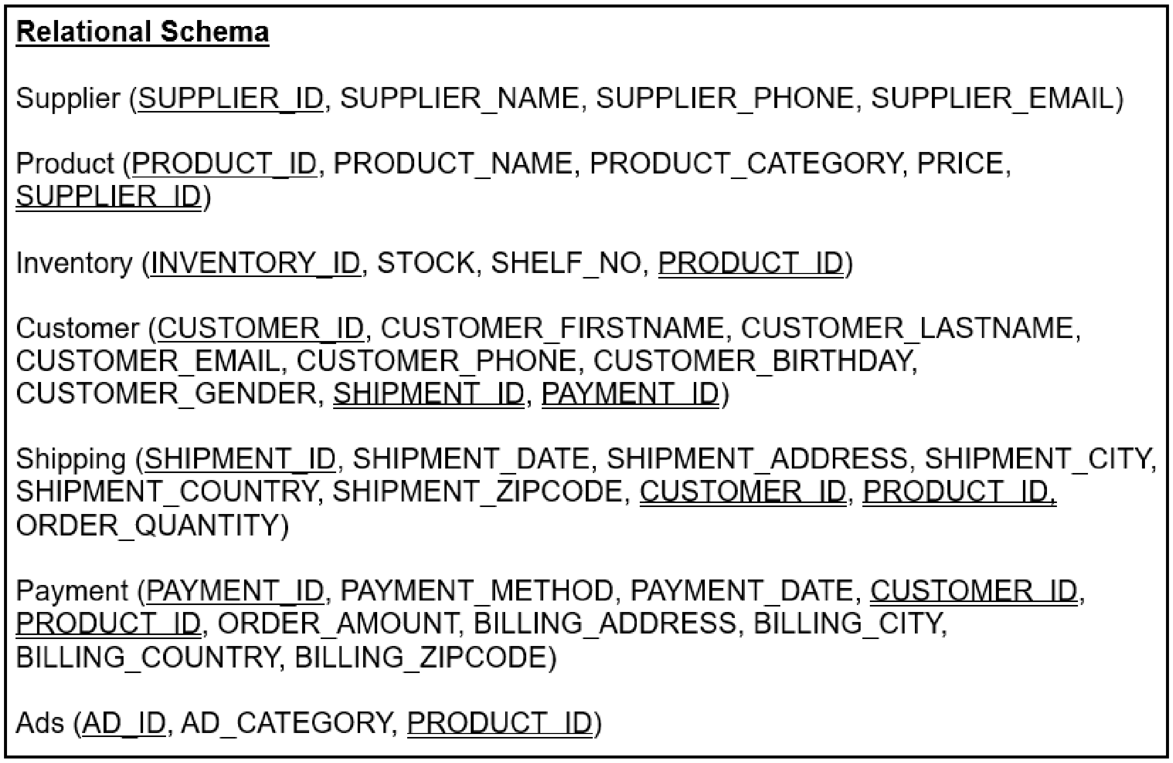


Figure 3: **Relationship Set**


Relational Schema

Supplier (SUPPLIER_ID, SUPPLIER_NAME, SUPPLIER_PHONE, SUPPLIER_EMAIL)

Product (PRODUCT_ID, PRODUCT_NAME, PRODUCT_CATEGORY, PRICE, SUPPLIER_ID)

Inventory (INVENTORY_ID, STOCK, SHELF_NO, PRODUCT_ID)

Customer (CUSTOMER_ID, CUSTOMER_FIRSTNAME, CUSTOMER_LASTNAME, CUSTOMER_EMAIL, CUSTOMER_PHONE, CUSTOMER_BIRTHDAY, CUSTOMER_GENDER, SHIPMENT_ID, PAYMENT_ID)

Shipping (SHIPMENT_ID, SHIPMENT_DATE, SHIPMENT_ADDRESS, SHIPMENT_CITY, SHIPMENT_COUNTRY, SHIPMENT_ZIPCODE, CUSTOMER_ID, PRODUCT_ID, ORDER_QUANTITY)

Payment (PAYMENT_ID, PAYMENT_METHOD, PAYMENT_DATE, CUSTOMER_ID, PRODUCT_ID, ORDER_AMOUNT, BILLING_ADDRESS, BILLING_CITY, BILLING_COUNTRY, BILLING_ZIPCODE)

Ads (AD_ID, AD_CATEGORY, PRODUCT_ID)

Figure 4: **Relational Schema**

The following step included translating the E-R diagram into a functional SQL database schema, where tables with appropriate datatypes, constraints and keys were created. The SQL code can be seen below.

```
library(dplyr)
library(lubridate)
library(ggplot2)
library(readr)
library(RSQLite)
library(DBI)
library(readxl)
library(gridExtra)
```

```
db <- dbConnect(RSQLite::SQLite(), dbname = "e_commerce_database.db")
```

```
sql_commands <- c(
  "CREATE TABLE IF NOT EXISTS Supplier (
    SUPPLIER_ID VARCHAR(255) PRIMARY KEY NOT NULL,
    SUPPLIER_NAME VARCHAR(255) NOT NULL,
    SUPPLIER_PHONE VARCHAR(255),
    SUPPLIER_EMAIL VARCHAR(255)
  );",

  "CREATE TABLE IF NOT EXISTS Product (
    PRODUCT_ID VARCHAR(255) PRIMARY KEY NOT NULL,
    PRODUCT_NAME VARCHAR(255) NOT NULL,
    PRODUCT_CATEGORY VARCHAR(255),
    PRICE FLOAT NOT NULL,
    SUPPLIER_ID VARCHAR(255) NOT NULL,
    FOREIGN KEY(SUPPLIER_ID) REFERENCES Supplier(SUPPLIER_ID)
  );",

  "CREATE TABLE IF NOT EXISTS Inventory (
    INVENTORY_ID VARCHAR(255) PRIMARY KEY NOT NULL,
    STOCK INTEGER NOT NULL,
    SHELF_NO VARCHAR(255),
    PRODUCT_ID VARCHAR(255) NOT NULL,
    FOREIGN KEY(PRODUCT_ID) REFERENCES Product(PRODUCT_ID)
  );",

  "CREATE TABLE IF NOT EXISTS Customer (
    CUSTOMER_ID VARCHAR(255) PRIMARY KEY NOT NULL,
    CUSTOMER_FIRSTNAME VARCHAR(255) NOT NULL,
    CUSTOMER_LASTNAME VARCHAR(255) NOT NULL,
    CUSTOMER_EMAIL VARCHAR(255),
    CUSTOMER_PHONE VARCHAR(255),
    CUSTOMER_BIRTHDAY DATE,
    CUSTOMER_GENDER VARCHAR(50),
    SHIPMENT_ID VARCHAR(255),
    PAYMENT_ID VARCHAR(255)
  );",

  "CREATE TABLE IF NOT EXISTS Shipment (
    SHIPMENT_ID VARCHAR(255) PRIMARY KEY NOT NULL,
    SHIPMENT_DATE DATE NOT NULL,
    SHIPMENT_ADDRESS VARCHAR(255) NOT NULL,
    SHIPMENT_CITY VARCHAR(255) NOT NULL,
    SHIPMENT_ZIPCODE VARCHAR(255) NOT NULL,
    SHIPMENT_COUNTRY VARCHAR(255) NOT NULL,
    CUSTOMER_ID VARCHAR(255) NOT NULL,
    PRODUCT_ID VARCHAR(255) NOT NULL,
    FOREIGN KEY(CUSTOMER_ID) REFERENCES Customer(CUSTOMER_ID),
    FOREIGN KEY(PRODUCT_ID) REFERENCES Product(PRODUCT_ID)
  );",

  "CREATE TABLE IF NOT EXISTS Payment (
    PAYMENT_ID VARCHAR(255) PRIMARY KEY NOT NULL,
    PAYMENT_METHOD VARCHAR(255) NOT NULL,
    ORDER_AMOUNT FLOAT NOT NULL,
    PAYMENT_DATE DATE NOT NULL,
    BILLING_ADDRESS VARCHAR(255) NOT NULL,
    BILLING_CITY VARCHAR(255) NOT NULL,
    BILLING_ZIPCODE VARCHAR(255) NOT NULL,
    BILLING_COUNTRY VARCHAR(255) NOT NULL,
    CUSTOMER_ID VARCHAR(255) NOT NULL,
    PRODUCT_ID VARCHAR(255) NOT NULL,
    FOREIGN KEY(CUSTOMER_ID) REFERENCES Customer(CUSTOMER_ID),
    FOREIGN KEY(PRODUCT_ID) REFERENCES Product(PRODUCT_ID)
  );"
)

# Execute each SQL command to create the tables
for(sql_command in sql_commands) {
  dbExecute(db, sql_command)
}
```

Part 3: Data Generation and Validation

3.1 Data Generation

Generating Datasets: We initiated the project by creating datasets from scratch, by leveraging AI platforms and advanced language models such as ChatGPT. A decision was made to generate six distinct datasets tailored to specific entities and additional auxiliary data. Each dataset was chosen to meet the requirements of our database schema and adhere to normalization forms.

Product, Category and Supplier Data: To populate the datasets, we used ChatGPT to generate realistic observations. We generated unique product names paired with relevant categories while supplier information was obtained to match product categories. This ensures the relevance within the dataset. The prompt used for ChatGPT can be seen below.

```
"I want you to act as an expert in database management and create a
comprehensive table for an e-commerce database. The table should include
150 unique products spread across 15 categories, with relevant suppliers
listed for each product. Ensure consistency and relevance between the
fields in a real-world sense"
```

```
[1] "I want you to act as an expert in database management and create a \ncomprehensive tabl
```

Customer, Shipping and Payment Information: Customer, shipping, and payment data were generated using a combination of tools such as Mockaroo and Python scripting. When handling customer data, we generated realistic names by dividing them into first and last names. Python scripting was used to merge these names and append random email domains such as Yahoo and Google to enhance authenticity. Finally, payment and shipping information were generated to include relevant details like payment addresses. The prompts used are seen below.

```
"I want you to act like an expert in database management, and generate
dates ranging from the 1980s to 2015 randomly using Python. Please
provide me with a code snippet showcasing how to achieve this task
efficiently"
```

```
[1] "I want you to act like an expert in database management, and generate \ndates ranging f
```

```
"I want you to act as a database management expert and provide me
with a code snippet to merge first and last names, adding a random
email domain (Google or Yahoo) to create a new field called email
address"
```

```
[1] "I want you to act as a database management expert and provide me \nwith a code snippet
```

Additional Data: Additional auxiliary data, including fields like date of birth and UK-based phone numbers, were handled, either by enriching them or omitting them if deemed irrelevant to the dataset's objective.

Key Placement and Cross-Referencing: The strategic placement of primary and foreign keys was a critical aspect in our database design. We ensured that all keys were appropriately positioned to establish cohesive relationships between tables. Cross-referencing between entities, particularly among customers, payments, and shipping, was implemented to maintain data integrity and facilitate seamless data retrieval.

3.2. Data Import and Quality Assurance

Following the generation of the datasets, the subsequent step involved loading and populating our tables, using the ETL process. Each dataset is stored in a dataframe for the respective entities.

```r
suppliers_data <- read.csv("supplier_ecommerce.csv")
products_data <- read.csv("products_ecommerce.csv")
inventories_data <- read.csv("inventory_ecommerce.csv")
customers_data <- read.csv("customers_ecommerce.csv")
shipments_data <- read.csv("shipment_ecommerce.csv")
payments_data <- read.csv("payments_ecommerce.csv")
```

Data Validation

In our data validation process, we assessed attributes across our datasets to ensure data quality and consistency. Initially, we verified primary key uniqueness. Subsequently, we scrutinized attributes such as names to ensure they're in character format and don't exceed 25 characters. Phone numbers were checked for precisely 9 digits, while email addresses adhere to standard formatting. Shipping details were restricted to specific UK regions. Gender is categorized as male, female, or other. Dates were validated in the dd/mm/yyyy format, and payment methods are aligned with the five options provided by our store.

```r
#Data Validation

#Customer check

##customer id
unique_customer_id <- nrow(customers_data) == length(unique(customers_data$CUSTOMER_ID))
customers_data <- customers_data[unique_customer_id, ]

validate_customer_id <- function(customer_id) {
  !is.na(customer_id) && substr(customer_id, 1, 1) == "C" && nchar(customer_id) == 6 && grepl
}

# Apply the validation function to the CUSTOMER_ID column
valid_customer_id <- sapply(customers_data$CUSTOMER_ID, validate_customer_id)
customers_data <- customers_data[valid_customer_id, ]

# Check which entries fail validation
invalid_entries <- customers_data[!valid_customer_id, "CUSTOMER_ID"]
```

```r
## Phone Number - Numeric, Length and Uniqueness
validate_phone_number <- function(phone_number) {
  all(grepl("^[0-9]{9}$", phone_number) & !duplicated(phone_number))
}

customers_data$CUSTOMER_PHONE <- as.integer(customers_data$CUSTOMER_PHONE)

# Apply the validation function to the CUSTOMER_PHONE column
valid_phone_number <- sapply(customers_data$CUSTOMER_PHONE, validate_phone_number)
invalid_entries <- customers_data[!valid_phone_number, "CUSTOMER_PHONE"]
customers_data <- customers_data[valid_phone_number, ]

##email
### Define domain list
validate_email <- function(email) {
  domains <- c("gmail.com", "outlook.com", "yahoo.com", "hotmail.com", "icloud.com")
  all(grepl("@", email) & grepl(paste(domains, collapse="|"), email))
}

## First Name - Characters and Max Length
validate_firstname <- function(firstname) {
  !is.na(firstname) && all(grepl("^[[:alpha:]]+$", firstname)) && nchar(firstname) <= 25
}

### Apply the validation function to the CUSTOMER_FIRSTNAME column
valid_firstname <- sapply(customers_data$CUSTOMER_FIRSTNAME, validate_firstname)

### Keep only the rows with valid first names
customers_data <- customers_data[valid_firstname, ]

## Last Name
validate_lastname <- function(lastname) {
  !is.na(lastname) && all(grepl("^[-'[:alpha:][:space:]]+$", lastname)) && nchar(lastname) <
}

### Apply the validation function to the CUSTOMER_LASTNAME column
valid_lastname <- sapply(customers_data$CUSTOMER_LASTNAME, validate_lastname)

###check for invalid entries
invalid_entries <- customers_data[!valid_lastname, "CUSTOMER_LASTNAME"]
invalid_entries
```

```
[1] "Frenzel;"
```

```r
### Keep only the rows with valid last names
customers_data <- customers_data[valid_lastname, ]

##Gender check

###function for check
validate_gender <- function(gender) {
  !is.na(gender) && gender %in% c("Male", "Female", "Other")
}

###filtering invalid data
valid_gender <- sapply(customers_data$CUSTOMER_GENDER, validate_gender)
customers_data <- customers_data[valid_gender, ]

## Birthday
validate_date <- function(date) {
  !is.na(date) && !is.na(as.Date(date, format = "%d/%m/%Y", tryFormats = c("%d/%m/%Y")))
}

#Products check

##Product ID
unique_product_id <- nrow(products_data) == length(unique(products_data$PRODUCT_ID))

validate_product_id <- function(product_id) {
  !is.na(product_id) && substr(product_id, 1, 1) == "P" && nchar(product_id) == 4 && grepl("
}

valid_product_id <- sapply(products_data$PRODUCT_ID, validate_product_id)
products_data <- products_data[valid_product_id, ]

##PRICE
products_data$PRICE <- as.integer(products_data$PRICE)
validate_price <- function(price) {
  !is.na(price) && grepl("^[0-9]{1,10}$", price)
}

### Apply the validation function to the PRICE column
valid_price <- sapply(products_data$PRICE, validate_price)
```

```r
### Check which entries fail validation
invalid_entries <- products_data[!valid_price, "PRICE"]
invalid_entries
```

```
integer(0)
```

```r
products_data <- products_data[valid_price, ]

##Product Category
validate_category <- function(category) {
  !is.na(category) && all(grepl("^[-'[:alpha:]&[:space:]]+$", category)) && nchar(category)
}

### Apply the validation function to the PRODUCT_CATEGORY column
valid_category <- sapply(products_data$PRODUCT_CATEGORY, validate_category)

###check for invalid entries
invalid_entries <- products_data[!valid_category, "PRODUCT_CATEGORY"]

products_data <- products_data[valid_category, ]

##Product name
validate_product_name <- function(product_name) {
  !is.na(product_name) && all(grepl("^[-'[:alnum:]&[:space:],.()\"\\\\]+$", product_name)) &&
}
### Apply the validation function to the PRODUCT_NAME column
valid_product_name <- sapply(products_data$PRODUCT_NAME, validate_product_name)

###check for invalid entries
invalid_entries <- products_data[!valid_product_name, "PRODUCT_NAME"]
invalid_entries
```

```
character(0)
```

```r
products_data <- products_data[valid_product_name, ]

##Supplier data
unique_supplier_id <- nrow(suppliers_data) == length(unique(suppliers_data$SUPPLIER_ID))

validate_supplier_id <- function(supplier_id) {
  !is.na(supplier_id) && substr(supplier_id, 1, 1) == "S" && nchar(supplier_id) == 4 && grepl
}

valid_supplier_id <- sapply(suppliers_data$SUPPLIER_ID, validate_supplier_id)
suppliers_data <- suppliers_data[valid_supplier_id, ]

##Supplier phone
valid_sphone_number <- sapply(suppliers_data$SUPPLIER_PHONE, validate_phone_number)
invalid_entries <- suppliers_data[!valid_sphone_number, "SUPPLIER_PHONE"]
suppliers_data <- suppliers_data[valid_sphone_number, ]

##Supplier email

# Apply the validation function to the SUPPLIER_EMAIL column
valid_semail <- sapply(suppliers_data$SUPPLIER_EMAIL, validate_email)
invalid_entries <- suppliers_data[!valid_semail, "SUPPLIER_PHONE"]
suppliers_data <- suppliers_data[valid_semail, ]

##Supplier name
validate_supplier_name <- function(name) {
  !is.na(name) && all(grepl("^[-'[:alpha:]&[:space:],.]+$", name))
}

### Apply the validation function to the SUPPLIER_NAME column
valid_supplier_name <- sapply(suppliers_data$SUPPLIER_NAME, validate_supplier_name)

### Check which entries fail validation
invalid_entries <- suppliers_data[!valid_supplier_name, "SUPPLIER_NAME"]
suppliers_data <- suppliers_data[valid_supplier_name, ]

#Inventory

##inventory id
unique_inventory_id <- nrow(inventories_data) == length(unique(inventories_data$INVENTORY_ID

validate_inventory_id <- function(inventory_id) {
  !is.na(inventory_id) && substr(inventory_id, 1, 3) == "INV" && nchar(inventory_id) <= 7 &&
}

valid_inventory_id <- sapply(inventories_data$INVENTORY_ID, validate_inventory_id)
invalid_entries <- inventories_data[!valid_inventory_id, "INVENTORY_ID"]
inventories_data <- inventories_data[valid_inventory_id, ]
```

```
##stock
valid_stock <- sapply(inventories_data$STOCK, validate_price)
invalid_entries <- inventories_data[!valid_stock, "STOCK"]
inventories_data <- inventories_data[valid_stock, ]

##shelf no.
validate_shelf_no <- function(shelf_no) {
  !is.na(shelf_no) && nchar(shelf_no) == 2 &&
    grepl("^[A-Z][1-9]$", shelf_no)
}

### Apply the validation function to the SHELF_NO column
valid_shelf_no <- sapply(inventories_data$SHELF_NO, validate_shelf_no)

### Check which entries fail validation
invalid_entries <- inventories_data[!valid_shelf_no, "SHELF_NO"]
inventories_data <- inventories_data[valid_shelf_no, ]

#Shipments

##ID
unique_shipment_id <- nrow(shipments_data) == length(unique(shipments_data$SHIPMENT_ID))

validate_shipment_id <- function(shipment_id) {
  !is.na(shipment_id) && substr(shipment_id, 1, 1) == "E" && nchar(shipment_id) <= 10 && grep
}

valid_shipment_id <- sapply(shipments_data$SHIPMENT_ID, validate_shipment_id)
invalid_entries <- shipments_data[!valid_shipment_id, "SHIPMENT_ID"]
shipments_data <- shipments_data[valid_shipment_id, ]

## shipment date
valid_shipment_date <- sapply(shipments_data$SHIPMENT_DATE, validate_date)

# Check which entries fail validation
invalid_entries <- shipments_data[!valid_shipment_date, "SHIPMENT_DATE"]
shipments_data <- shipments_data[valid_shipment_date, ]

## shipping city
unique(shipments_data$SHIPMENT_ZIPCODE)
```

```
 [1] "CV35" "NE46" "BS14" "BD23" "RH5"  "S33"  "CT15" "DN36" "WC1B" "BT66"
[11] "LE15" "NN4"  "NG22" "TF6"  "WC2H" "AB55" "DL10" "SN13" "NG34" "SY4"
[21] "LN6"  "BS37" "L33"  "RG20" "LS6"  "AB56" "BS41" "WF9"  "B40"  "OX12"
[31] "NR34" "N3"   "DT10" "M14"  "M34"  "CH48" "G4"   "ST20" "NR29" "GL54"
[41] "DL8"  "NN11" "PH43" "W1F"  "SN1"  "EC3M" "EH9"  "CB4"  "SW19" "S8"
[51] "LE14" "S1"   "PR1"  "EH52" "SG4"  "LE16" "CT16" "L74"  "BT2"  "LS9"
[61] "SW1E" "B12"  "BH21" "KW10" "AB39" "GU32" "EC1V" "OX7"  "DN21" "DN22"
[71] "IV1"  "BD7"
```

```r
valid_uk_cities <- c("London", "Birmingham", "Manchester", "Glasgow", "Edinburgh", "Liverpool

valid_uk_zipcodes <- c("CV35", "NE46", "BS14", "BD23", "RH5", "S33", "CT15", "DN36", "WC1B",

### Function to validate shipment city and zipcode
validate_shipment_city <- function(city) {
  return(city %in% valid_uk_cities)
}

### Function to validate shipment zipcode
validate_shipment_zipcode <- function(zipcode) {
  return(zipcode %in% valid_uk_zipcodes)
}

### Check the validity of each shipment city and zipcode
valid_city <- sapply(shipments_data$SHIPMENT_CITY, validate_shipment_city)
valid_zipcode <- sapply(shipments_data$SHIPMENT_ZIPCODE, validate_shipment_zipcode)

# Filter out invalid entries
invalid_entries <- shipments_data[!valid_city | !valid_zipcode, ]

## Shipment country
validate_shipment_country <- function(country) {
  return(country %in% c("UK", "United Kingdom"))
}

# Check the validity of each shipment country
valid_country <- sapply(shipments_data$BILLING_COUNTRY, validate_shipment_country)

# Filter out invalid entries
invalid_entries <- shipments_data[!valid_country, ]

#Payments
```

```r
##ID
unique_payment_id <- nrow(payments_data) == length(unique(payments_data$PAYMENT_ID))

validate_payment_id <- function(payment_id) {
  !is.na(payment_id) && substr(payment_id, 1, 1) == "P" && nchar(payment_id) <= 10 && grepl(
}

valid_payment_id <- sapply(payments_data$PAYMENT_ID, validate_payment_id)
invalid_entries <- payments_data[!valid_payment_id, "PAYMENT_ID"]
payments_data <- payments_data[valid_payment_id, ]

## Payment Method
validate_payment_method <- function(payment_method) {
  payment_methods <- c("Credit card", "Klarna", "Apple Pay", "PayPal", "Debit card")
  !is.na(payment_method) && payment_method %in% payment_methods
}

### Apply the validation function to the PAYMENT_METHOD column
valid_payment_method <- sapply(payments_data$PAYMENT_METHOD, validate_payment_method)

### Check which entries fail validation
invalid_entries <- payments_data[!valid_payment_method, "PAYMENT_METHOD"]
payments_data <- payments_data[valid_payment_method, ]

## Order Amount
valid_amount <- sapply(payments_data$ORDER_AMOUNT, validate_price)
invalid_entries <- payments_data[!valid_amount, "ORDER_AMOUNT"]
payments_data <- payments_data[valid_amount, ]

##Payment Date
valid_payment_date <- sapply(payments_data$PAYMENT_DATE, validate_date)

# Check which entries fail validation
invalid_entries <- payments_data[!valid_payment_date, "PAYMENT_DATE"]
payments_data <- payments_data[valid_payment_date, ]
```

3.3 Data Import

Post-validation, the dataframes undergo updates in the corresponding tables via SQL operations. Our process assumes that new data inserted into the tables is presented in dataframes bearing the same filenames as those used in the initial dataframe creation. We developed a function to scrutinize duplicates, primarily focusing on primary keys. If a record already exists, the function abstains from adding redundant entries to the respective table. Only unique entries are appended, ensuring data integrity.

```r
#uploading data in the table.
data_exists <- function(db, table_name, criteria) {
  query <- paste0("SELECT COUNT(*) FROM ", table_name, " WHERE ", criteria, ";")
  result <- dbGetQuery(db, query)
  return(result[[1]] > 0)
}

# Function to insert data into a database table for each entity
insert_data <- function(db, table_name, data_frame, unique_column) {
  for (i in 1:nrow(data_frame)) {
    # Extract row data
    row_data <- data_frame[i, ]

    # Extract unique value for validation
    unique_value <- row_data[[unique_column]]

    # Check if data already exists
    if (data_exists(db, table_name, paste0(unique_column, " = '", unique_value, "'"))) {
      print(paste("Data with", unique_column, unique_value, "already exists in", table_name,
    } else {
      # Insert data into the table
      dbWriteTable(db, table_name, row_data, append = TRUE)
    }
  }
}

#insert_data(db, "Supplier", suppliers_data, "SUPPLIER_ID")
#insert_data(db, "Product", products_data, "PRODUCT_ID")
#insert_data(db, "Inventory", inventories_data, "INVENTORY_ID")
#insert_data(db, "Customer", customers_data, "CUSTOMER_ID")
#insert_data(db, "Shipment", shipments_data, "SHIPMENT_ID")
#insert_data(db, "Payment", payments_data, "PAYMENT_ID")
```

Once the entities were populated, rigorous checks were undertaken to validate compliance with normalization principles, specifically 2NF and 3NF.

Further validation checks are shown below.

```
# Check for duplicate Customer records
duplicate_customers_query <- "
SELECT CUSTOMER_EMAIL, COUNT(*)
FROM Customer
GROUP BY CUSTOMER_EMAIL
HAVING COUNT(*) > 1;"
duplicate_customers <- dbGetQuery(db, duplicate_customers_query)
# Print the results
print("Duplicate Customers:")
```

```
[1] "Duplicate Customers:"
```

```
print(duplicate_customers)
```

```
[1] CUSTOMER_EMAIL COUNT(*)
<0 rows> (or 0-length row.names)
```

```
# Check for Products without a Category
products_without_category_query <- "
SELECT PRODUCT_ID, PRODUCT_NAME
FROM Product
WHERE PRODUCT_CATEGORY IS NULL OR PRODUCT_CATEGORY = '';"
products_without_category <- dbGetQuery(db, products_without_category_query)
print("Products without a Category:")
```

```
[1] "Products without a Category:"
```

```
print(products_without_category)
```

```
[1] PRODUCT_ID   PRODUCT_NAME
<0 rows> (or 0-length row.names)
```

```
# Check for Suppliers without contact information (both phone and email missing)
suppliers_without_contact_query <- "
SELECT SUPPLIER_ID, SUPPLIER_NAME
FROM Supplier
WHERE SUPPLIER_PHONE IS NULL AND SUPPLIER_EMAIL IS NULL;"
suppliers_without_contact <- dbGetQuery(db, suppliers_without_contact_query)
print("Suppliers without Contact Information:")
```

```
[1] "Suppliers without Contact Information:"
```

```
print(suppliers_without_contact)
```

```
[1] SUPPLIER_ID   SUPPLIER_NAME
<0 rows> (or 0-length row.names)
```

```
# Check Inventory for negative stock values
negative_stock_query <- "
SELECT PRODUCT_ID, STOCK
FROM Inventory
WHERE STOCK < 0;"
negative_stock <- dbGetQuery(db, negative_stock_query)
print("Inventory Items with Negative Stock:")
```

```
[1] "Inventory Items with Negative Stock:"
```

```
print(negative_stock)
```

```
[1] PRODUCT_ID STOCK
<0 rows> (or 0-length row.names)
```

Part 4: Data Pipeline Generation

4.1 GitHub Repository and Workflow Setup

linked: https://github.com/AdnanAndar98/MSBA_DM_Group_14

The first task was to set up a GitHub repository, a critical component for managing and version-controlling our project. The objective was straightforward yet foundational: create a centralized repository to house our database files, scripts, and other necessary documents. This was crucial for ensuring that all team members had access to the latest versions of our work and could contribute effectively.

We initiated the project by creating the repository and organizing it to include all relevant material. The repository served as a storage space and as a platform for collaboration. Through Git's version control capabilities, every change was carefully tracked and documented, ensuring a transparent evolution of our project.

4.2 Github Actions for Continuous Integration

Automating data validation and database updates using GitHub Action aimed to streamline our workflow, minimize manual errors, and ensure that our database system was consistently updated with data.

We approached this by setting up workflows triggered by specific events, such as push or pull requests. These automated sequences were designed to perform several critical functions:

- **Data Validation**: Automatically executing scripts to validate data integrity upon new commits or pullrequests, ensuring consistency.
- **Database Updates**: Detecting changes related to database files or scripts, then automatically updating it with new data to keep our system current.
- **Data Analysis**: Automatically running data analysis scripts to assess and derive insights, enhancing our understanding of the eCommerce environment.

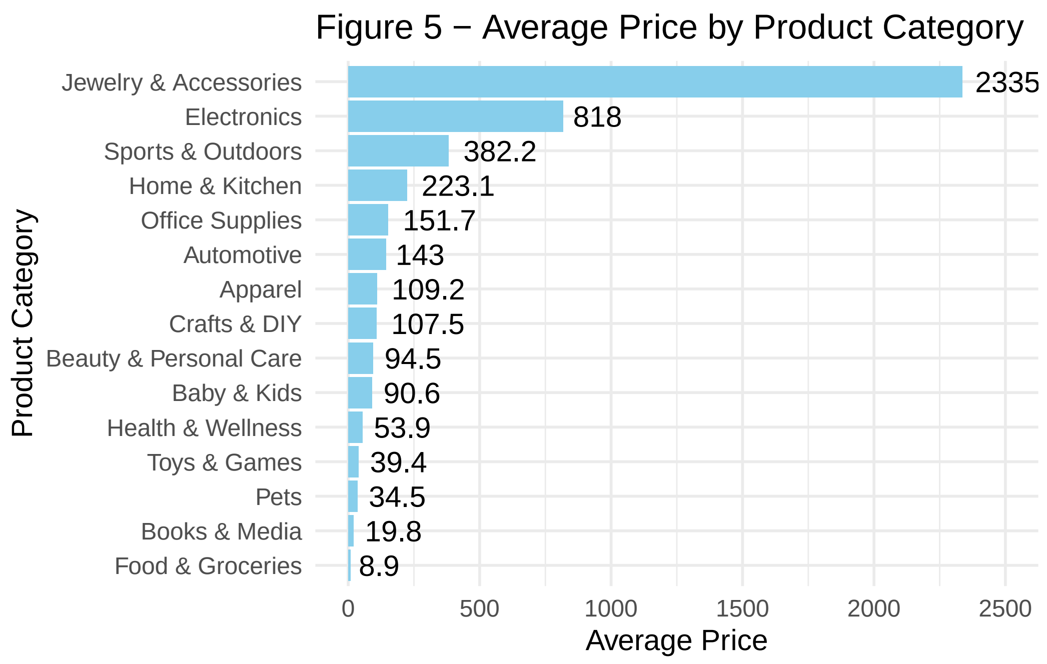
4.3 Navigating Challenges

Setting up GitHub Actions presented a steep learning curve, requiring deep documentation and collaborative problem-solving. Additionally, automating R scripts for database management and analysis necessitated planning and testing for seamless workflow integration. Managing a project with six members demanded clear communication and an organized approach to prevent overlaps and conflicts.

Part 5: Data Analysis and Reporting with Quarto in R

5.1 Advance Data Analysis in R

5.1.1. Average Price for each Product Category



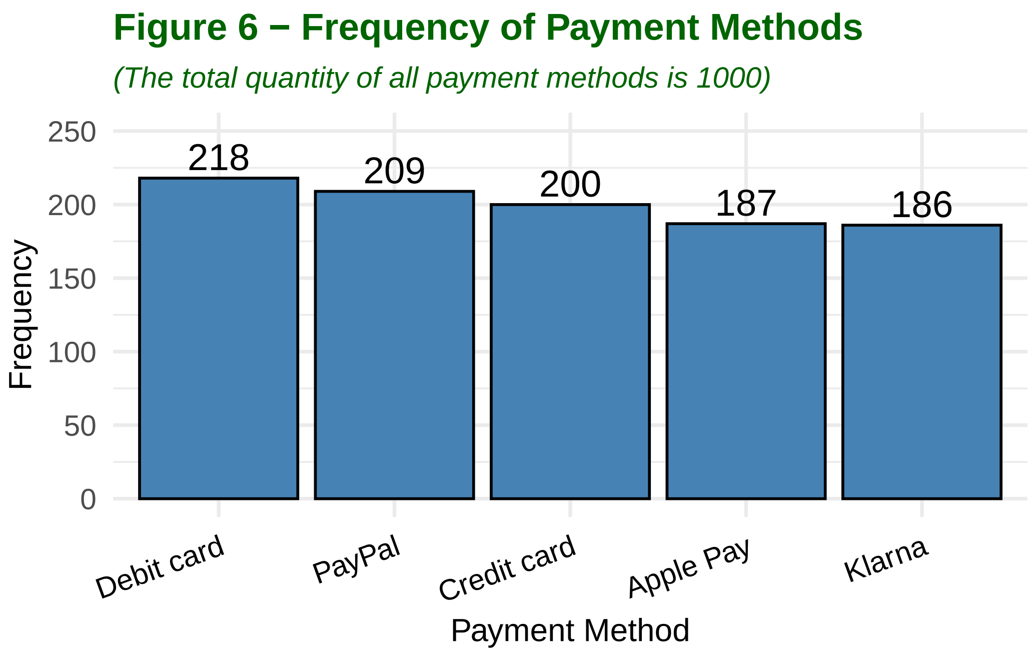
```
List of 1
 $ axis.text.x:List of 11
  ..$ family       : NULL
  ..$ face         : NULL
  ..$ colour       : NULL
  ..$ size         : NULL
  ..$ hjust        : num 1
  ..$ vjust        : NULL
  ..$ angle        : num 45
  ..$ lineheight   : NULL
  ..$ margin       : NULL
  ..$ debug        : NULL
  ..$ inherit.blank: logi FALSE
  ..- attr(*, "class")= chr [1:2] "element_text" "element"
 - attr(*, "class")= chr [1:2] "theme" "gg"
 - attr(*, "complete")= logi FALSE
 - attr(*, "validate")= logi TRUE
```

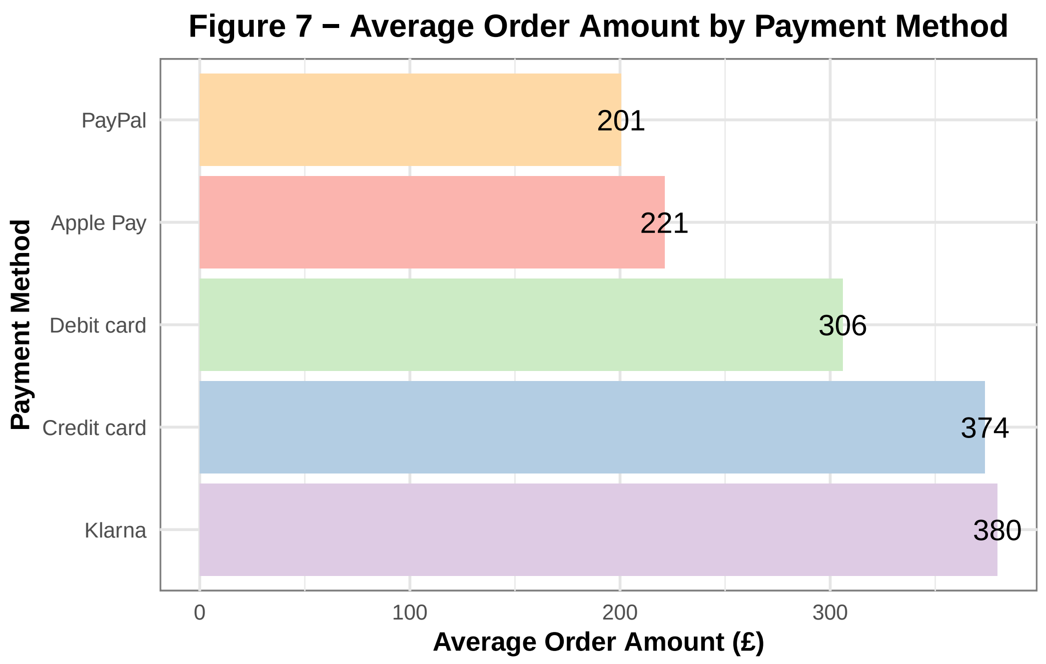
As seen in Figure 5, the product category with the highest average price is Jewelery and Accessories, closely followed by Electronics. Understanding these patterns serves as a powerful tool for strategic decision-making.

For example, when considering inventory management, it can help us allocate resources more efficiently, by allocating resources towards high-demand categories and minimizing stockouts.

Furthermore, adjusting prices according to demand trends is essential in unlocking revenue opportunities. Implementing dynamic pricing and/or value-based pricing strategies will allow our platform to capture the customers' willingness to pay premium prices, hence, maximizing profit.

5.1.2. Payment Methods





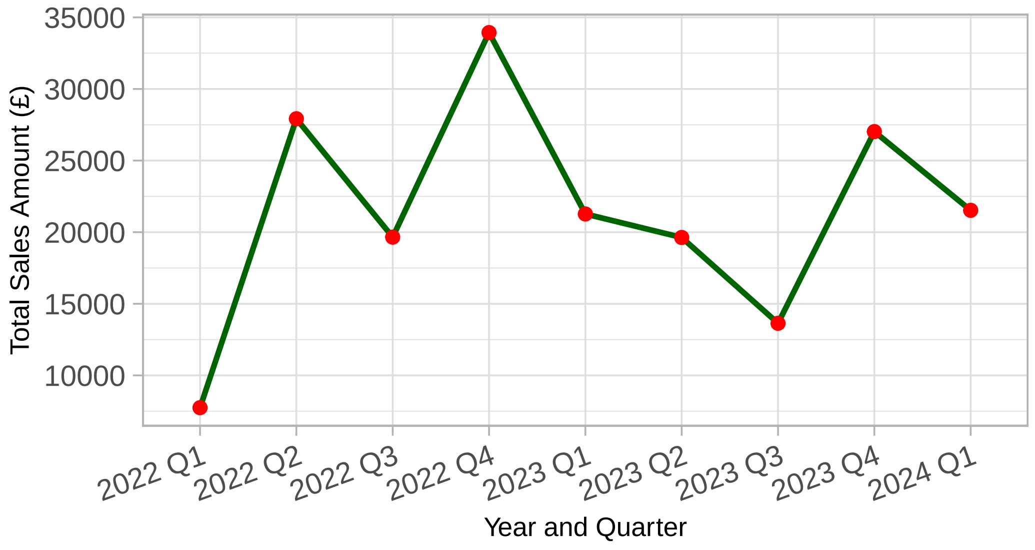
The payment analysis reveals that customers most commonly use debit cards and PayPal. However, the data shows a balanced distribution across different payment methods.

Figure 7 shows that customers use Klarna and credit cards for higher-value purchases. This implies that there's an inclination towards installment-based payment solutions or credit facilities for larger purchases. This insight can help us improve the overall customer experience.

5.1.3. Quarterly Sales Amount Trend

Figure 8 − Quarterly Sales Amount Trend

Total sales amount across different quarters



The fluctuating sales trends indicate a degree of market volatility and/or shifts in consumer behavior. The initial increase in sales during Q2 2022 suggests potential factors at play, such as seasonal trends and marketing initiatives. Subsequent declines in sales during 2023 Q1 to Q2 reflect challenges such as economic downturns because of COVID-19.

Nonetheless, the recovery in sales during 2023 Q4 indicates the effectiveness of strategic interventions or market adjustments to stimulate demand and regain momentum. Overall, these fluctuations underscore the dynamic nature of the market and the need to remain responsive to changing conditions.

5.1.4. Top 10 Cities with Highest Order Amount

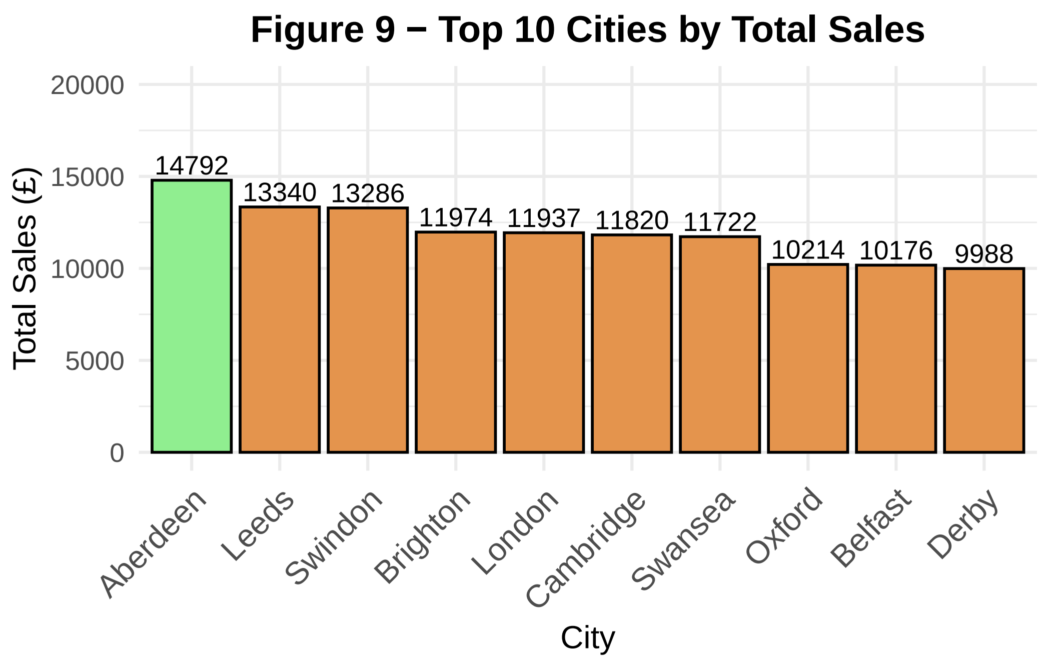


Figure 9 illustrates the top 10 cities by total sales. Aberdeen contributes significantly to sales revenue, nearly reaching £15,000. Aberdeen's robust economy is driven by industries such as oil and gas which could lead to higher purchasing power, thus, more spending.

Leeds, Swindon, Brighton, and London also emerge as noteworthy contributors. Leeds's economy shows strengths in finance and manufacturing while Swindon's retail sector could be the reason for high sales. Finally, London's inclusion among the top contributors is unsurprising given its status as a global financial and commercial hub.

The above analysis provides valuable insights that can be used to optimize operations. For example, localized strategies can be implemented to increase customer engagement.

5.1.5. Customer Segmentation Across Generational Cohorts

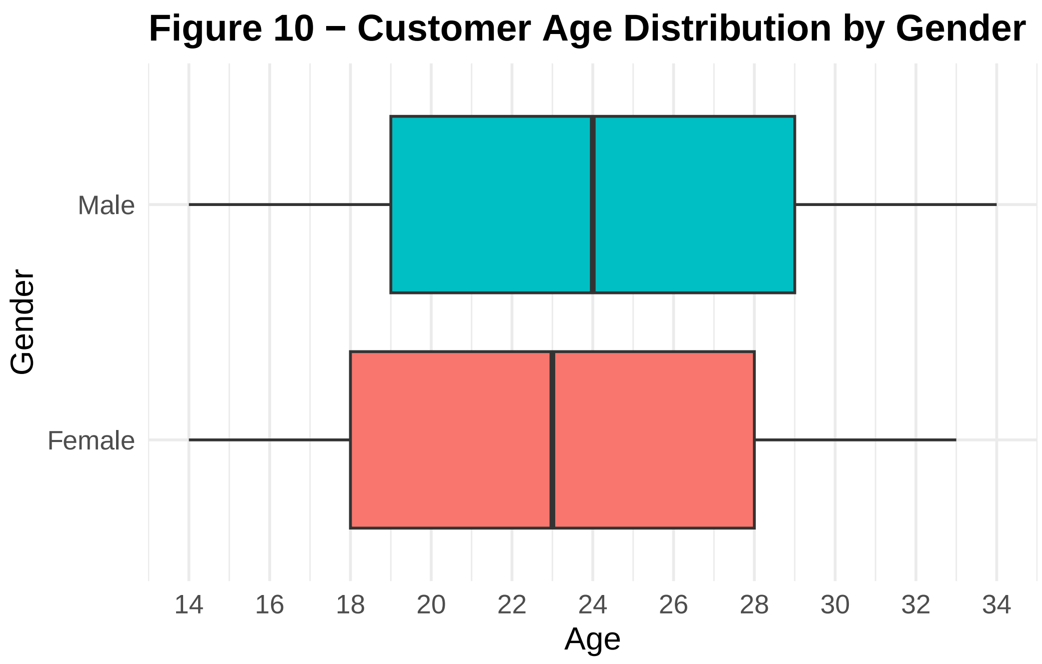


Figure 10 compares the distribution of customer ages across different genders. The boxplot shows that the mean age for females is 23, whereas for males is 24. The similarity in ages between genders suggests that we effectively target individuals in their late teens and twenties.

This insight presents an opportunity for the company to refine its marketing strategies by tailoring advertisements and commercials to resonate more effectively with the specific audience segment.

5.1.6 Total Stock by Product Category

Figure 11 – Total Stock by Product Category

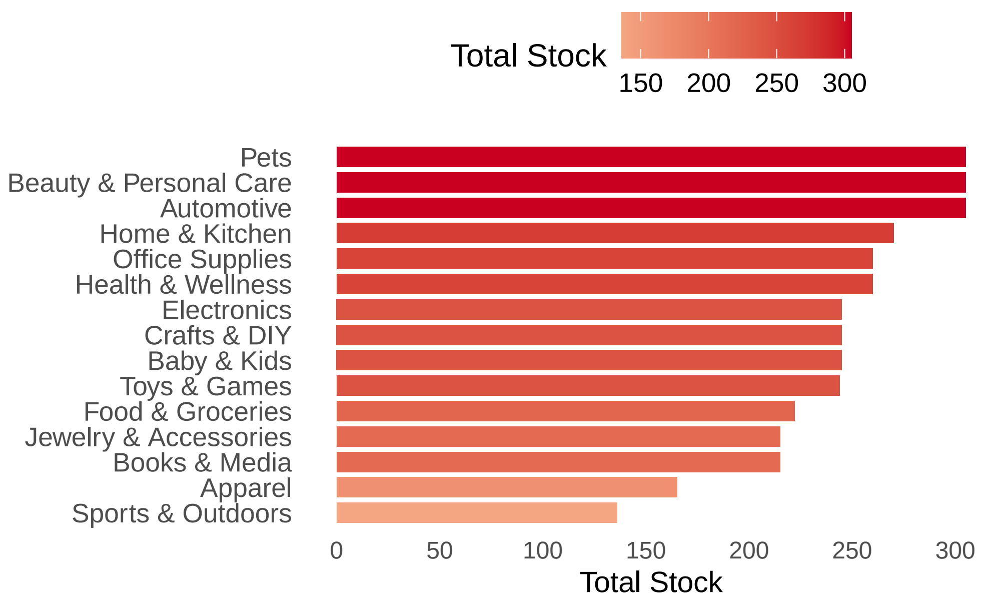


Figure 11 represents the total stock quantities across different product categories. Pets, beauty & personal care, and automotive products prove to have the highest stock availability. This suggests that the eCommerce prioritized maintaining inventory levels for these categories, potentially due to their popularity among customers.

On the other hand, sports & outdoors, as well as apparel exhibit the lowest stocks, being almost half compared to the top categories. The limited availability suggests the need to monitor stock levels to ensure customer demand is always met. Insufficient stock can cause disruptions and stockouts which are undesirable in all businesses.

Part 6: Conclusion

To conclude, this report outlines the comprehensive process of database management. The journey involved various stages including relational database and schema design, synthetic data generation, and automation processes. These were achieved using different tools such as SQL and GitHub.

Lastly, the eCommerce data were analysed to reflect the real-world scenarios about customer behavior, trend sales, payment preferences and inventory management. These insights are crucial in guiding strategic decision-making.