Data Management Group_31 Report

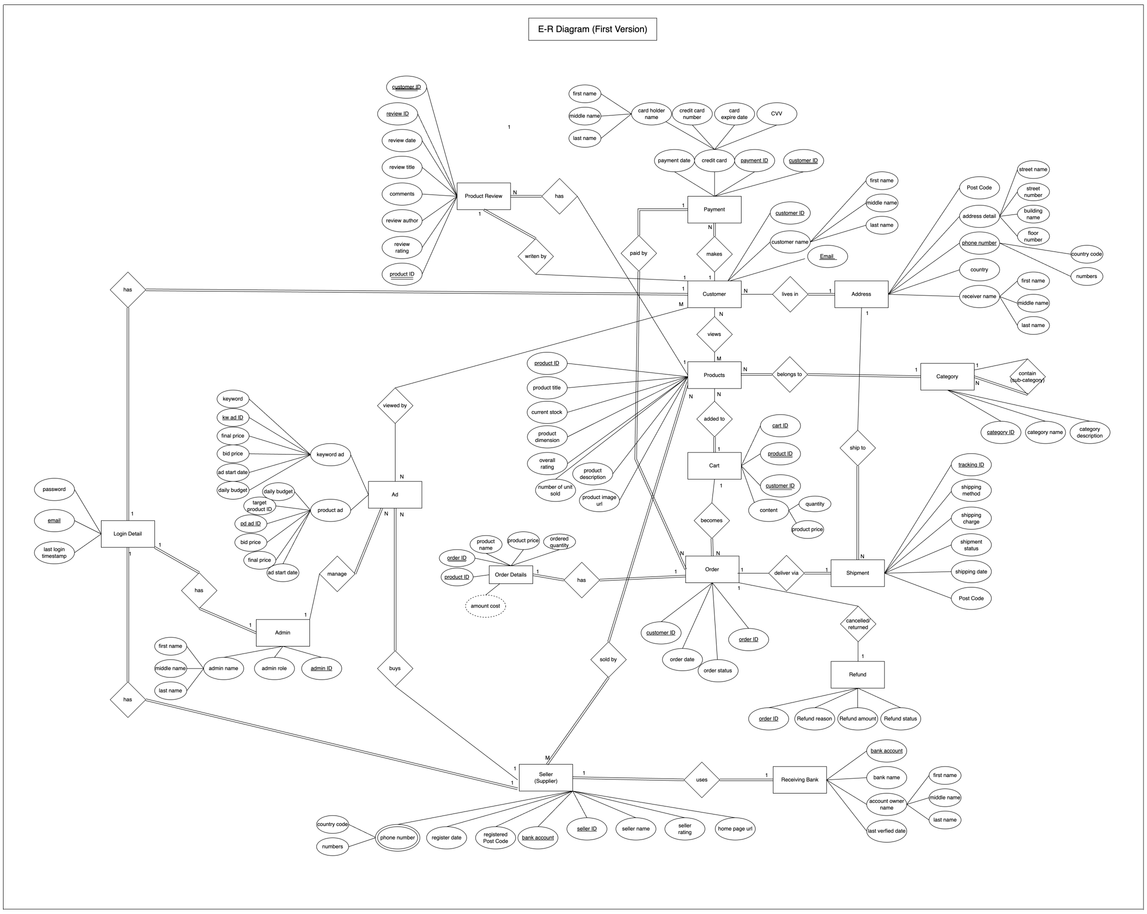
Table of contents

1 Part1: Database Design and Implementation

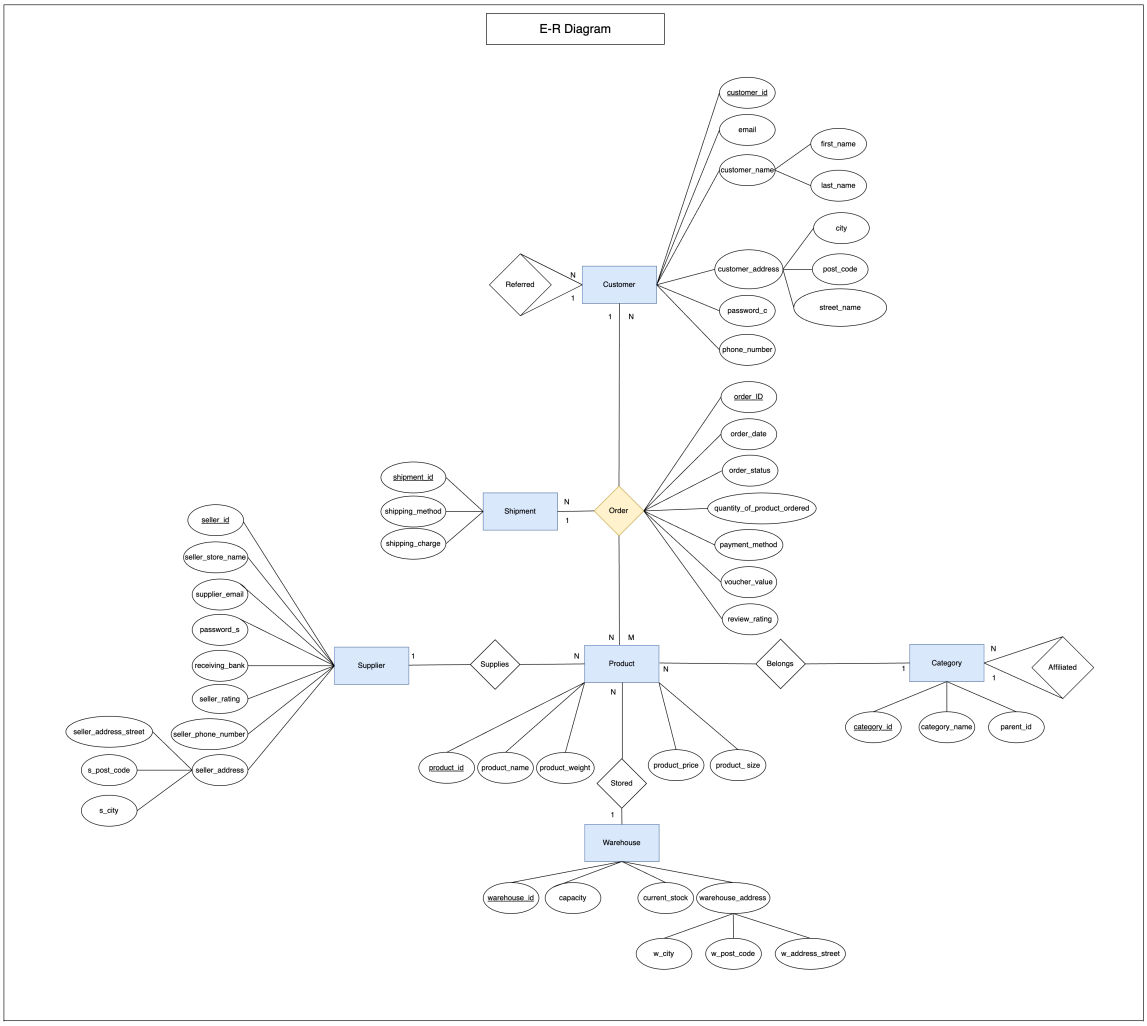
1.1 Task 1.1: E-R Diagram Design

The first step in building our e-commerce database was to design the E-R diagram. Creating the ER diagram was an iterative process. Our first version, shown in Figure 1, was too complex involving more than ten entities with multiple attributes and relationships. It also included participation constraints which we then removed for simplicity. We modified the E-R diagram multiple times removing loops as they created unnecessary relationships, reducing the number of entities and relationships for simplification and to account for mistakes that hindered implementation. Moreover, we removed or replaced several attributes during the data generation phase as we faced difficulties in generating them. Our final diagram is shown in Figure 2.

1.1.0.0.1 Figure 1: Our Original E-R diagram



1.1.0.0.2 Figure 2: Our Final E-R diagram



Our final diagram contains 6 main entities and their attributes. Seven 1 to N relationships, one M to N relationship between Product and Customer and 2 self referencing relationships(referred and belongs). The key attributes (unique) of each entity are underlined.

1.1.1 Assumption Made for Cardinality

We made several assumptions about cardinality of the relationships shown below together with the relationship sets:

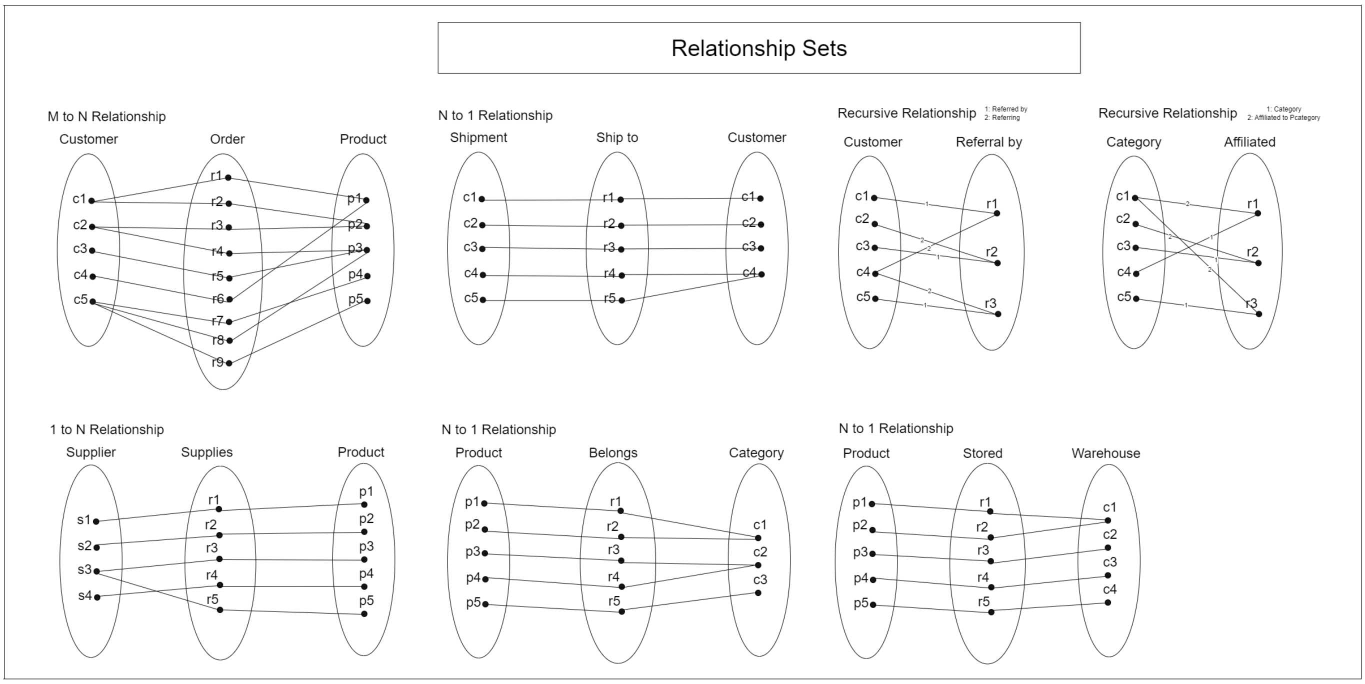
1. Each customer can be referred by only one customer at most.
2. Each customer can refer more than one customer.
3. Emails used for login cannot be repeated between customers and suppliers.
4. Passwords can be the same for different users (customers or suppliers).
5. Suppliers can sell multiple products, but one product can only be sold by one supplier.

6. Each order is made by one customer.
7. Each shipment can only contain one order.
8. All products within one order will be shipped together in one shipment (i.e. one shipment can ship more than one product).
9. One customer can place multiple orders that contain multiple products.
10. Each product can only use one voucher; each order can apply multiple vouchers.
11. Each product must belong to only one sub-category.
12. Each sub-category must be categorized in one parent-category.
13. One warehouse can contain multiple products; one product can only be stored in one warehouse.
14. Each customer can order multiple products and each product can be bought by multiple customers.

1.1.2 Relationship Sets

Below are shown the relationship sets used in the E-R diagram:

1.1.2.0.1 Figure 3: Relationship Sets



1.2 Task 1.2: SQL Database Schema Creation

1.2.1 Logical Schema

Following the conceptual modelling we translated the E-R diagram to the logical schema converting each entity and each many to many relationship to a separate table including the primary and foreign keys. Category, Customer, Supplier, Warehouse, Product and Shipment were all entities and were converted to tables. Order was a many to many relationship between Product and Customer hence became a table as well.

Product is an entity with product_id as primary key. Product also has attribute of product_name, product_weight, product_price, product_size, supplier_id(foreign key), category_id(foreign key), and warehouse_Id(foreign key) since Product has relationship with Supplier, Category, and Warehouse(1:N).

Shipment is an entity with shipment_id as primary key. Shipment also has attribute of shipping_method and shipping_charge.

Category is an entity with category_id as primary key. Category also has attribute of category_name and parent_id.

Customer is an entity with customer_id as primary key. Customer also has attribute of email, city, post_code, street_name, first_name, last_name, password_c, phone_number, and referral_by.

Supplier is an entity with seller_id as primary key. Supplier also has attribute of seller_store_name, supplier_email, password_s, receiving_bank, seller_rating, seller_phone_number, seller_address_street, s_post_code, and s_city.

Warehouse is an entity with warehouse_id as primary key. Warehouse also has attribute of w_city, w_post_code, w_address_street, capacity, and current_stock.

Order table is the table from the M to N relationship of Customer and Product entity. In result, the Order table will have composite primary key consists of order_id, customer_id, and product_id. In addition, Order table also has attribute of quantity_of_product_ordered, order_status, payment_method, order_date, voucher_value, and review_rating, as well product_id, customer_id, shipment_id as foreign key since Order is ternary relationship between Product, Customer, and Shipment.

Logical Schema

Product (product_id, product_name, product_weight, product_price, product_size, supplier_id, category_id, warehouse_id)

Shipment (shipment_id, shipping_method, shipping_charge)

Category (category_id, category_name, parent_id)

Customer (customer_id, email, city, post_code, street_name, first_name, last_name, password_c, phone_number, referral_by)

Supplier (seller_id, seller_store_name, supplier_email, password_s, receiving_bank, seller_rating, seller_phone_number, seller_address_street, s_post_code, s_city)

Warehouse (warehouse_id, w_city, w_post_code, w_address_street, capacity, current_stock)

Order(order_id, product_id, customer_id, shipment_id, quantity_of_product_ordered, order_status, payment_method, order_date, voucher_value, review_rating)

Since we planned to generate data ourselves, we had ensured that the schema was normalized up to at least 3NF and every record was atomic to prevent data redundancy and ensure data integrity.

1.2.2 SQL Database Creation

In the process of setting up the database for our clothing website, it was essential to establish the necessary tables to organize and manage the data effectively. With the logical schema as the blueprint we used SQL DDL to create the database.

Before creating the tables we imported the necessary packages and established a connection.

```
#install.packages("readr")
#install.packages("RSQLite")
#install.packages("dplyr")
#install.packages("chron")
#install.packages("ggplot2")
library(readr)
library(RSQLite)
library(dplyr)
library(chron)
library(ggplot2)
```

```
my_connection <- RSQLite::dbConnect(RSQLite::SQLite(),"e-commerce.db")
```

Category table

We started by creating a Category table to store information about different product categories.This table serves as the fundamental component of our database schema, providing a structured framework for organizing and categorizing our product inventory. The Category table includes fields to store unique identifiers for each category, along with their respective names and any hierarchical relationships, such as parent categories.

```sql
--Check if the table exists and drops it if it does
--Ensure a clean slate for creating the table
DROP TABLE IF EXISTS Category;
```

```sql
CREATE TABLE IF NOT EXISTS Category(
  category_id VARCHAR(20) PRIMARY KEY NOT NULL,
  category_name VARCHAR (20) NOT NULL,
  parent_id VARCHAR(20)
  );
```

Customer table

As we continue to build our database infrastructure, another crucial aspect is managing customer information effectively. The Customer table serves as a central repository for storing essential details about our customers, enabling us to personalize their experience and facilitate seamless interactions with our platform. By this, Customer table is a structured framework to capture key attributes of each customer, including their unique identifier, contact information, address details, and authentication credentials.

```sql
DROP TABLE IF EXISTS Customer;
```

```sql
CREATE TABLE IF NOT EXISTS Customer(
  customer_id VARCHAR(50) PRIMARY KEY NOT NULL,
  email VARCHAR (100) NOT NULL,
  first_name VARCHAR (100) NOT NULL,
  last_name VARCHAR (100) NOT NULL,
  street_name VARCHAR (100) NOT NULL,
  post_code VARCHAR(64) NOT NULL,
  city VARCHAR (100) NOT NULL,
  password_c VARCHAR (10) NOT NULL,
  phone_number INT (11) NOT NULL,
  referral_by VARCHAR(50)
  );
```

Supplier table

The Supplier table aims to centralize essential details about each supplier, including their unique identifier, contact information, banking details, and performance metrics. This table plays a pivotal role in supplier management, enabling to track supplier ratings, monitor transaction activities, and streamline procurement processes.

```
DROP TABLE IF EXISTS Supplier;
```

```
CREATE TABLE IF NOT EXISTS Supplier (
    seller_id VARCHAR(50) PRIMARY KEY NOT NULL,
    seller_store_name VARCHAR(100) NOT NULL,
    supplier_email VARCHAR(255) NOT NULL,
    password_s VARCHAR(255) NOT NULL,
    receiving_bank VARCHAR(50) NOT NULL,
    seller_rating INT,
    seller_phone_number VARCHAR(20) NOT NULL,
    seller_address_street VARCHAR(255) NOT NULL,
    s_post_code VARCHAR(50) NOT NULL,
    s_city VARCHAR(50) NOT NULL
    );
```

Warehouse table

Next , the Warehouse table provides a foundation for organizing and monitoring warehouse facilities across various locations. It helps establish a comprehensive repository for storing key details about each warehouse, such as unique identifiers, capacity metrics, current stock levels, and address information. This table serves as a critical asset for inventory management, enabling the company to track inventory levels, optimize storage space utilization, and streamline logistics operations.

```
DROP TABLE IF EXISTS Warehouse;
```

```
CREATE TABLE IF NOT EXISTS Warehouse (
    warehouse_id VARCHAR(50) PRIMARY KEY NOT NULL,
    capacity INT NOT NULL,
    current_stock INT NOT NULL,
    w_city VARCHAR(50) NOT NULL,
    w_post_code VARCHAR(50) NOT NULL,
    w_address_street VARCHAR(255) NOT NULL
    );
```

Product table

Product table serves as the backbone of the company's inventory management system, providing an organized framework for cataloging and tracking details about each product in the company's inventory. This table supports effective decision-making processes related to pricing, restocking, and product assortment. Additionally, the inclusion of foreign key constraints ensures data integrity and enforces relationships with the Supplier, Category, and Warehouse tables, fostering a cohesive database architecture.

```
DROP TABLE IF EXISTS Product;
```

```
CREATE TABLE IF NOT EXISTS Product (
  product_id INT PRIMARY KEY NOT NULL,
  product_name VARCHAR(50) NOT NULL,
  category_id VARCHAR(20) NOT NULL,
  warehouse_id VARCHAR(50),
  seller_id VARCHAR(50) NOT NULL,
  product_weight FLOAT NOT NULL,
  product_price FLOAT NOT NULL,
  product_size VARCHAR(20) NOT NULL,
  FOREIGN KEY (seller_id) REFERENCES Supplier(seller_id)
  FOREIGN KEY (category_id) REFERENCES Category(category_id),
  FOREIGN KEY (warehouse_id) REFERENCES Warehouse(warehouse_id)
  );
```

Shipment table

Moving forward, the focus shifts to the creation of the shipment table. This table complements our inventory management efforts by facilitating the tracking and management of order shipments and delivery processes. It also enables us to monitor shipment statuses, track delivery timelines, and calculate shipping costs accurately. The primary key constraint ensures the uniqueness of each shipment record, while also facilitating efficient retrieval and manipulation of shipment data within our database.

```
DROP TABLE IF EXISTS Shipment;
```

```
CREATE TABLE IF NOT EXISTS Shipment (
    shipment_id VARCHAR(50) PRIMARY KEY NOT NULL,
    shipping_method VARCHAR(50) NOT NULL,
    shipping_charge FLOAT NOT NULL
    );
```

Orders table

Continuing our database development efforts for the clothing website, the orders table serves as a foundation element of our order management system, offering a robust framework for capturing and managing essential details about each customer order. This helps create a centralized repository for recording comprehensive order information, including unique order identifiers, order dates, order statuses, quantities of products ordered, payment methods, voucher values, review ratings, and associated shipment and customer details. This table facilitates organized order processing, enabling to efficiently track order statuses, manage inventory levels, and analyze customer purchasing behaviors.

```
DROP TABLE IF EXISTS Orders;
```

```
CREATE TABLE IF NOT EXISTS Orders (
    order_id VARCHAR(50) NOT NULL,
    order_date DATE NOT NULL,
    order_status VARCHAR(50) NOT NULL,
    quantity_of_product_ordered INT NOT NULL,
    payment_method VARCHAR(50) NOT NULL,
    voucher_value INT NOT NULL,
    review_rating INT,
    shipment_id VARCHAR(50) NOT NULL,
    product_id VARCHAR(50) NOT NULL,
    customer_id VARCHAR(50) NOT NULL,
    PRIMARY KEY (order_id, customer_id, product_id),

    FOREIGN KEY (shipment_id) REFERENCES Shipment(shipment_id),
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id)
    );
```

2 Part2: Data Generation and Management

2.1 Task 2.1: Synthetic Data Generation

We employed python 'faker' package, combined with tools such as ChatGPT, to generate synthetic data. For example, we can ask ChatGPT give us a list for postcode and city names:

```
postcode_city_data = {
    "AB10": "Aberdeen",
    "AB22": "Aberdeen",
    "EH1": "Edinburgh",
    "EH8": "Edinburgh",
    "G1": "Glasgow",
    "G2": "Glasgow",
    "KA1": "Kilmarnock",
    "KA22": "Ardrossan",
    "IV1": "Inverness",
```

```
    "IV2": "Inverness",
    "KY1": "Kirkcaldy",
    "KY7": "Glenrothes",
    "DG1": "Dumfries",
    "DG6": "Castle Douglas",
    "PA1": "Paisley",
    "PA19": "Gourock",
    "DD1": "Dundee",
    "DD10": "Montrose",
    "ML1": "Motherwell",
    "ML12": "Biggar"
}
```

Then, we used ‘faker’ package in python to generate customer data as follows:

```
def customer_data(num_customers, postcode_city_data, filename):
    fake = Faker()
    customer_id_set = set()
    with open(filename, 'w', newline='') as file:
        writer = csv.writer(file, quoting=csv.QUOTE_NONNUMERIC)
        writer.writerow(['customer_id', 'email', 'first_name', 'last_name',
        'street_name', 'post_code', 'city', 'password_c', 'phone_number',
        'referral_by'])

        # Generate list of customer IDs
        while len(customer_id_set) < num_customers:
            customer_id_set.add(fake.random_int(min=10001, max=50000))

        # Convert set to list for easy popping
        customer_ids = list(customer_id_set)
        random.shuffle(customer_ids)  # Shuffle the list of customer IDs

        for _ in range(num_customers):
            post_code, city = random.choice(list(postcode_city_data.items()))
            street_name = fake.street_address()
            # Get a customer ID and remove it from the list
            customer_id = customer_ids.pop()
            first_name = fake.first_name()
            last_name = fake.last_name()
            # Create email using first name and last name
            email = f"{first_name.lower()}.{last_name.lower()}@gmail.com"
            password_c = fake.password()
            phone_number = '7' + str(fake.random_number(digits=9))
```

```
        if customer_ids:
            referral_by = random.choice(customer_ids)
        else:
            referral_by = None

        writer.writerow([
            customer_id,
            email,
            first_name,
            last_name,
            street_name,
            post_code,
            city,
            password_c,
            phone_number,
            referral_by
        ])
return list(customer_id_set)
```

2.1.1 Assumptions Made for Data Generation Background

1. This is a fashion company that sells mostly clothes and accessories.
2. Seller rating is an integer between 1 and 5.
3. Product rating is an integer between 1 and 5.
4. All IDs are unique series of numerical digits (only integer values).
5. The available sizes for all products are between XS to XL.
6. The price supplied by the supplier is the price the product is sold at (before a voucher is applied).
7. All prices shown are in pounds.
8. Product weight is in grams.
9. A customer can leave different reviews for different products in the same order.
10. A customer can purchase at most 8 different products in one order.
11. Product rating can only be given and shown after the order is done.
12. If the customer apply for return, then all products in that order will be returned together, and the review rating for the product will not be shown.
13. There are 5 order statuses: processing, paid, shipping, done, and return.
14. Orders cannot be cancelled but can be returned.
15. There are 3 shipping methods: one-day, three-days, and seven-day.
16. Shipping charge is based on which shipping methods the customer chose.
17. There are 5 types of payment methods: Apple Pay, Mastercard, Visa, Google Pay, Paypal. A customer can only pay via one payment method for each order.
18. The vouchers are price discounts in pounds.
19. Each product can only use one voucher; each order can apply to multiple vouchers.
20. 'voucher_value' equals zero means that no discount is applied to that ordered product.

21. All suppliers must provide a store name, receiving bank account, phone number, and address information.
22. If there is no rating left for the supplier, the 'seller_rating' column will be blank (NA).
23. Suppliers must provide product price, weight, and size information for every product they sell.

2.2 Task 2.2: Data Import and Quality Assurance

Before reading and writing data into the database, we checked the uniqueness of primary key for all of the files:

```
# primary key check for category data
all_files <- list.files("data_upload/Category_dataset/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Category_dataset/",variable)
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ",nrow(unique(this_file_contents[,1]))==number_of_rows))
}
```

```
[1] "Checking for: category_data_new.csv"
[1] " is TRUE"
[1] "Checking for: category_data.csv"
[1] " is TRUE"
```

```
# primary key check for customer data
all_files <- list.files("data_upload/Customer_dataset/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Customer_dataset/",variable)
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ",nrow(unique(this_file_contents[,1]))==number_of_rows))
}
```

```
[1] "Checking for: customer_data_new.csv"
[1] " is TRUE"
```

```
[1] "Checking for: customer_data.csv"
[1] " is TRUE"
```

```r
# primary key check for warehouse data
all_files <- list.files("data_upload/Warehouse_dataset/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Warehouse_dataset/",variable)
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ",nrow(unique(this_file_contents[,1]))==number_of_rows))
}
```

```
[1] "Checking for: warehouse_data_new.csv"
[1] " is TRUE"
[1] "Checking for: warehouse_data.csv"
[1] " is TRUE"
```

```r
# primary key check for supplier data
all_files <- list.files("data_upload/Supplier_dataset/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Supplier_dataset/",variable)
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ",nrow(unique(this_file_contents[,1]))==number_of_rows))
}
```

```
[1] "Checking for: supplier_data_new.csv"
[1] " is TRUE"
[1] "Checking for: supplier_data.csv"
[1] " is TRUE"
```

```r
# primary key check for product data
all_files <- list.files("data_upload/Product_dataset/")

for (variable in all_files) {
  this_filepath <- paste0("data_upload/Product_dataset/",variable)
```

```r
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ", nrow(unique(this_file_contents[,1])) == number_of_rows))
}
```

```
[1] "Checking for: product_data_new.csv"
[1] " is TRUE"
[1] "Checking for: product_data.csv"
[1] " is TRUE"
```

```r
# primary key check for shipment data
all_files <- list.files("data_upload/Shipment_dataset/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Shipment_dataset/",variable)
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ",nrow(unique(this_file_contents[,1]))==number_of_rows))
}
```

```
[1] "Checking for: shipment_data_new.csv"
[1] " is TRUE"
[1] "Checking for: shipment_data.csv"
[1] " is TRUE"
```

```r
# primary key check for order data
all_files <- list.files("data_upload/Orders_dataset/")

for (variable in all_files) {
  this_filepath <- paste0("data_upload/Orders_dataset/",variable)
  this_file_contents <- readr::read_csv(this_filepath)
  number_of_rows <- nrow(this_file_contents)

  print(paste0("Checking for: ",variable))

  print(paste0(" is ", nrow(unique(this_file_contents[,1])) == number_of_rows))
}
```

```
[1] "Checking for: order_data_new.csv"
[1] " is FALSE"
[1] "Checking for: order_data.csv"
[1] " is FALSE"
```

Except for order data, the uniqueness of the primary keys in other data are ensured. Since the primary key for Orders table is a composite of three columns, we will check the primary key for this table after appending data into the database.

We then read the .csv files from the data_upload folder and write them into the database by using **append = TRUE** function so that these data will not destroy the data type we set before; making sure that columns such as ids are character instead of number, so that when writing them to the database, they will not show in decimal format.

```
# set a function to list all csv files in the assigned path
list_csv_files <- function(folder_path) {
  files <- list.files(path = folder_path, pattern = "\\.csv$",
                      full.names = TRUE)
  return(files)
}

# create a folder and table name mapping for later use
folder_table_mapping <- list(
  "Customer_dataset" = "Customer",
  "Supplier_dataset" = "Supplier",
  "Category_dataset" = "Category",
  "Product_dataset" = "Product",
  "Orders_dataset" = "Orders",
  "Warehouse_dataset" = "Warehouse",
  "Shipment_dataset" = "Shipment"
)

# make sure some columns are in the data type we want before writing data
convert_column_types <- function(data, column_types) {
  for (col_name in names(column_types)) {
    if (col_name %in% names(data)) {
      col_type <- column_types[[col_name]]
      if (col_type == "character") {
        data[[col_name]] <- as.character(data[[col_name]])
      } else if (col_type == "date") {
        data[[col_name]] <- as.Date(data[[col_name]], format = "%Y/%m/%d")
        data[[col_name]] <- as.character(data[[col_name]])
      }
```

```r
    }
  }
  return(data)
}

# Data type mapping for each table's columns
column_types_mapping <- list(
  "Category" = c("category_id" = "character", "parent_id" = "character"),
  "Customer" = c("customer_id" = "character", "referral_by" = "character"),
  "Supplier" = c("seller_id" = "character"),
  "Warehouse" = c("warehouse_id" = "character"),
  "Product" = c("product_id" = "character", "seller_id" = "character",
              "warehouse_id" = "character", "category_id" = "character"),
  "Shipment" = c("shipment_id" = "character"),
  "Orders" = c("order_id" = "character", "customer_id" = "character",
             "product_id" = "character", "shipment_id" = "character",
             "order_date" = "date")
)

# Path to the main folder containing sub-folders
main_folder <- "data_upload"

# Process each sub-folder (table)
for (folder_name in names(folder_table_mapping)) {
  folder_path <- file.path(main_folder, folder_name)
  if (dir.exists(folder_path)) {
    cat("Processing folder:", folder_name, "\n")
    # List CSV files in the sub-folder
    csv_files <- list_csv_files(folder_path)

    # Get the corresponding table name from the mapping
    table_name <- folder_table_mapping[[folder_name]]

    # Append data from CSV files to the corresponding table
    for (csv_file in csv_files) {
      cat("Appending data from:", csv_file, "\n")
      tryCatch({
        # Read CSV file
        file_contents <- readr::read_csv(csv_file)

        # Convert column data types
        file_contents <- convert_column_types(file_contents,
                                             column_types_mapping[[table_name]])
```

```r
        # Append data to the table in SQLite
        RSQLite::dbWriteTable(my_connection, table_name, file_contents,
                              append = TRUE)
        cat("Data appended to table:", table_name, "\n")
      }, error = function(e) {
        cat("Error appending data:", csv_file, "\n")
        cat("Error message:", e$message, "\n")
      })
    }
  } else {
    cat("Folder does not exist:", folder_path, "\n")
  }
}
```

```
Processing folder: Customer_dataset
Appending data from: data_upload/Customer_dataset/customer_data_new.csv
Data appended to table: Customer
Appending data from: data_upload/Customer_dataset/customer_data.csv
Data appended to table: Customer
Processing folder: Supplier_dataset
Appending data from: data_upload/Supplier_dataset/supplier_data_new.csv
Data appended to table: Supplier
Appending data from: data_upload/Supplier_dataset/supplier_data.csv
Data appended to table: Supplier
Processing folder: Category_dataset
Appending data from: data_upload/Category_dataset/category_data_new.csv
Data appended to table: Category
Appending data from: data_upload/Category_dataset/category_data.csv
Data appended to table: Category
Processing folder: Product_dataset
Appending data from: data_upload/Product_dataset/product_data_new.csv
Data appended to table: Product
Appending data from: data_upload/Product_dataset/product_data.csv
Data appended to table: Product
Processing folder: Orders_dataset
Appending data from: data_upload/Orders_dataset/order_data_new.csv
Data appended to table: Orders
Appending data from: data_upload/Orders_dataset/order_data.csv
Data appended to table: Orders
Processing folder: Warehouse_dataset
Appending data from: data_upload/Warehouse_dataset/warehouse_data_new.csv
Data appended to table: Warehouse
```

```
Appending data from: data_upload/Warehouse_dataset/warehouse_data.csv
Data appended to table: Warehouse
Processing folder: Shipment_dataset
Appending data from: data_upload/Shipment_dataset/shipment_data_new.csv
Data appended to table: Shipment
Appending data from: data_upload/Shipment_dataset/shipment_data.csv
Data appended to table: Shipment
```

```r
# List tables to confirm data appending
tables <- RSQLite::dbListTables(my_connection)
print(tables)
```

```
[1] "Category"  "Customer"  "Orders"    "Product"   "Shipment"  "Supplier"
[7] "Warehouse"
```

As can be seen here, 7 tables were created successfully with assigned table names.

Use **PRAGMA table_info()** to verify the primary key, column names, data type, and NOT NULL setting of each table we created again.

```sql
PRAGMA table_info(Customer);
```

Table 1: Displaying records 1 - 10

cid	name	type	notnull	dflt_value	pk
0	customer_id	VARCHAR(50)	1	NA	1
1	email	VARCHAR (100)	1	NA	0
2	first_name	VARCHAR (100)	1	NA	0
3	last_name	VARCHAR (100)	1	NA	0
4	street_name	VARCHAR (100)	1	NA	0
5	post_code	VARCHAR(64)	1	NA	0
6	city	VARCHAR (100)	1	NA	0
7	password_c	VARCHAR (10)	1	NA	0
8	phone_number	INT (11)	1	NA	0
9	referral_by	VARCHAR(50)	0	NA	0

In Customer table, ‘customer_id’ is the only primary key; all columns except for ‘referral_by’ should be NOTNULL. Column names and data types match what we set when creating the table.

```
PRAGMA table_info(Category);
```

Table 2: 3 records

cid	name	type	notnull	dflt_value	pk
0	category_id	VARCHAR(20)	1	NA	1
1	category_name	VARCHAR (20)	1	NA	0
2	parent_id	VARCHAR(20)	0	NA	0

In Category table, 'category_id' is the only primary key; 'category_id' and 'category_name' follow the NOTNULL rule. Column names and data types match what we set when creating the table.

```
PRAGMA table_info(Supplier);
```

Table 3: Displaying records 1 - 10

cid	name	type	notnull	dflt_value	pk
0	seller_id	VARCHAR(50)	1	NA	1
1	seller_store_name	VARCHAR(100)	1	NA	0
2	supplier_email	VARCHAR(255)	1	NA	0
3	password_s	VARCHAR(255)	1	NA	0
4	receiving_bank	VARCHAR(50)	1	NA	0
5	seller_rating	INT	0	NA	0
6	seller_phone_number	VARCHAR(20)	1	NA	0
7	seller_address_street	VARCHAR(255)	1	NA	0
8	s_post_code	VARCHAR(50)	1	NA	0
9	s_city	VARCHAR(50)	1	NA	0

In Supplier table, 'seller_id' is the only primary key; except for 'seller_rating', all columns follow the NOTNULL rule. Column names and data types match what we set when creating the table.

```
PRAGMA table_info(Warehouse);
```

Table 4: 6 records

cid	name	type	notnull	dflt_value	pk
0	warehouse_id	VARCHAR(50)	1	NA	1
1	capacity	INT	1	NA	0

cid	name	type	notnull	dflt_value	pk
2	current_stock	INT	1	NA	0
3	w_city	VARCHAR(50)	1	NA	0
4	w_post_code	VARCHAR(50)	1	NA	0
5	w_address_street	VARCHAR(255)	1	NA	0

In Warehouse table, 'warehouse_id' is the only primary key; all columns follow the NOTNULL rule. Column names and data types match what we set when creating the table.

```
PRAGMA table_info(Product);
```

Table 5: 8 records

cid	name	type	notnull	dflt_value	pk
0	product_id	INT	1	NA	1
1	product_name	VARCHAR(50)	1	NA	0
2	category_id	VARCHAR(20)	1	NA	0
3	warehouse_id	VARCHAR(50)	0	NA	0
4	seller_id	VARCHAR(50)	1	NA	0
5	product_weight	FLOAT	1	NA	0
6	product_price	FLOAT	1	NA	0
7	product_size	VARCHAR(20)	1	NA	0

In Product table, 'product_id' is the only primary key; all columns except for 'warehouse_id' follow the NOTNULL rule. Column names and data types match what we set when creating the table.

```
PRAGMA table_info(Shipment);
```

Table 6: 3 records

cid	name	type	notnull	dflt_value	pk
0	shipment_id	VARCHAR(50)	1	NA	1
1	shipping_method	VARCHAR(50)	1	NA	0
2	shipping_charge	FLOAT	1	NA	0

In Shipment table, 'shipment_id' is the only primary key; all columns follow the NOTNULL rule. Column names and data types match what we set when creating the table.

```
PRAGMA table_info(Orders);
```

Table 7: Displaying records 1 - 10

cid	name	type	notnull	dflt_value	pk
0	order_id	VARCHAR(50)	1	NA	1
1	order_date	DATE	1	NA	0
2	order_status	VARCHAR(50)	1	NA	0
3	quantity_of_product_ordered	INT	1	NA	0
4	payment_method	VARCHAR(50)	1	NA	0
5	voucher_value	INT	1	NA	0
6	review_rating	INT	0	NA	0
7	shipment_id	VARCHAR(50)	1	NA	0
8	product_id	VARCHAR(50)	1	NA	3
9	customer_id	VARCHAR(50)	1	NA	2

Since the primary of Orders table are a composite of order_id, customer_id, and product_id, so in the 'pk' column it marks these three columns from 1 to 3 and leaves others as 0. All columns except for 'review_rating' follow the NOTNULL rule. Column names and data types match what we set when creating the table.

In the end of this section, we read these tables into data frame in R for the following analysis and visualization.

```
Customer <- dbGetQuery(my_connection, "SELECT * FROM Customer")
Supplier <- dbGetQuery(my_connection, "SELECT * FROM Supplier")
Warehouse <- dbGetQuery(my_connection, "SELECT * FROM Warehouse")
Product <- dbGetQuery(my_connection, "SELECT * FROM Product")
Orders <- dbGetQuery(my_connection, "SELECT * FROM Orders")
Shipment <- dbGetQuery(my_connection, "SELECT * FROM Shipment")
Category <- dbGetQuery(my_connection, "SELECT * FROM Category")
```

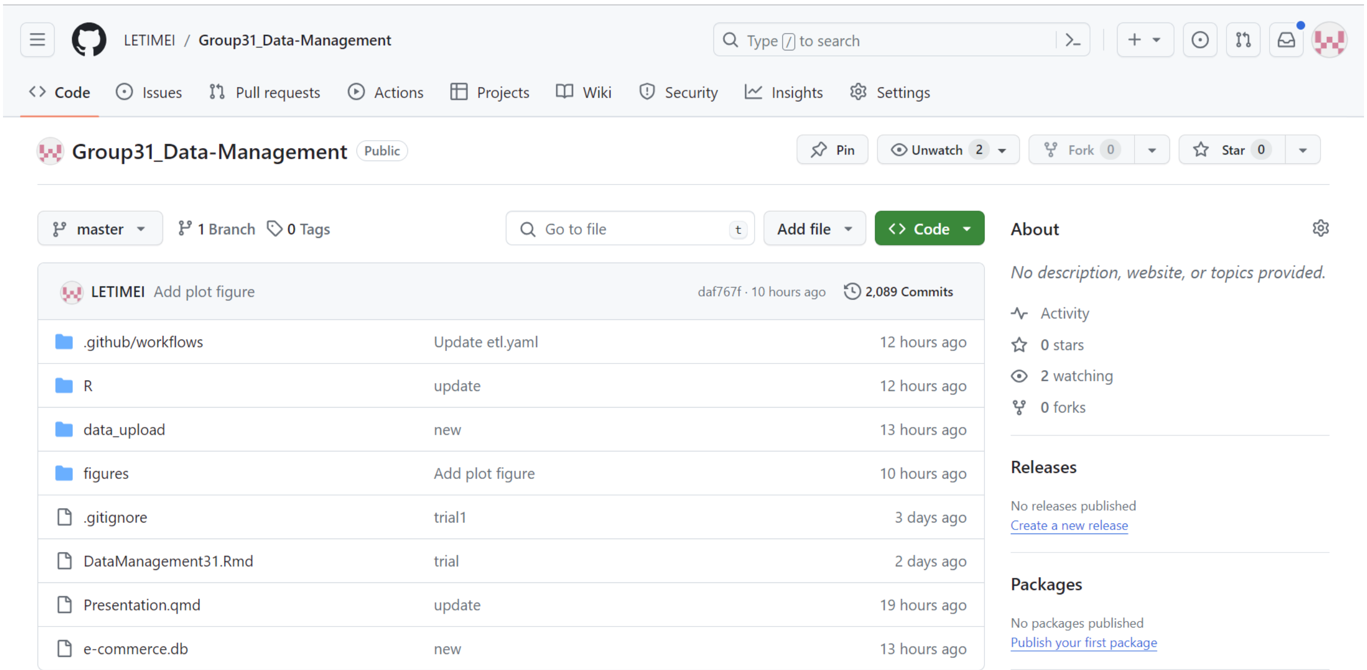
3 Part3: Data Pipeline Generation

In this section, we focus on setting up a data pipeline for efficient management and version control of our project using GitHub. The link to the team's GitHub work space is:

https://github.com/LETIMEI/Group31_Data-Management

3.1 Task 3.1: GitHub repository and Workflow Setup

The objective here is to utilize a GitHub repository to manage our project. We connected our file on Posit Cloud with the GitHub work space, and used ‘push’ and ‘pull’ to control the version and synchronize necessary files and script to run in the workflow.



3.2 Task 3.2: GitHub Action for Continuous Integration

By setting up workflows triggered by specific events like pushes or pull requests, we can automate data validation, database updates, and execute basic data analysis tasks seamlessly within our development environment.

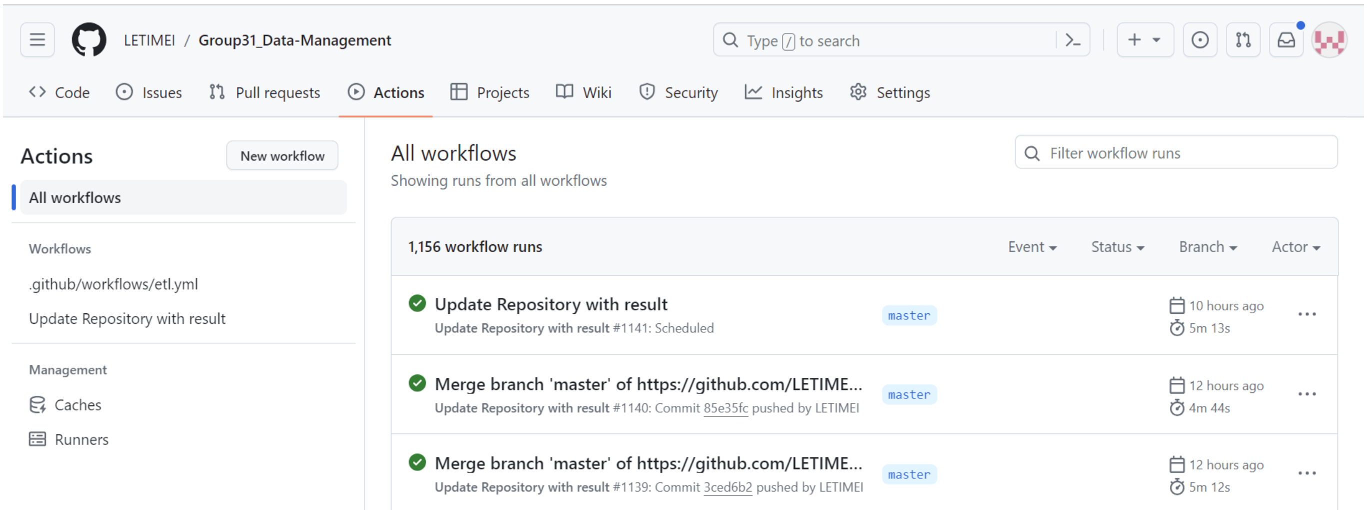
Subsequently, we established our workflow as outlined below. We specified the interval and some conditions for the R script reruns, identified required packages, defined the script to execute, designated the file path for saved figures, and specified the token name for reference:

```
name: Update Repository with Result

on:
  schedule:
    - cron: '0 */12 * * *' # Run every 12 hours
  push:
    branches: [ master ]
```

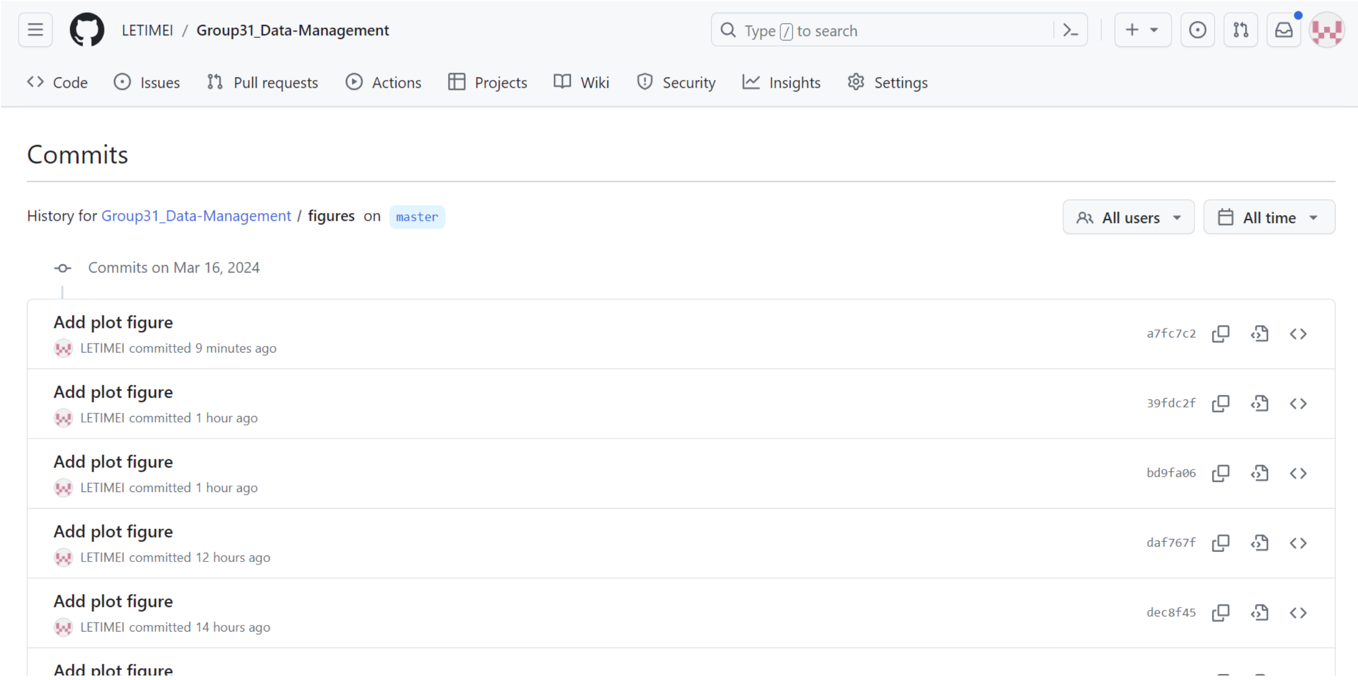
```
    paths:
     - '.github/workflows/**' # Run whenever the workflow code update
     - 'R/**'  # Run whenever the R script update
     - 'data_upload/**'#Run whenever new data uploaded to the data_upload folder

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
        if: steps.cache.outputs.cache-hit != 'true'
        run: |
          Rscript -e 'install.packages(c("readr","ggplot2","RSQLite",
          "dplyr","chron","png"))'
      - name: Execute R script
        run: |
          Rscript R/DataManagement31.R
      - name: Add files
        run: |
          git config --local --unset-all "http.https://github.com/.extraheader"
          git config --global user.email "meimelody1129@gmail.com"
          git config --global user.name "LETIMEI"
          git add --all figures/
      - name: Commit files
        run: |
          git commit -m "Add plot figure"
      - name: Pull changes
        run: |
          git stash save "temp changes"
          git pull --no-rebase origin master
```

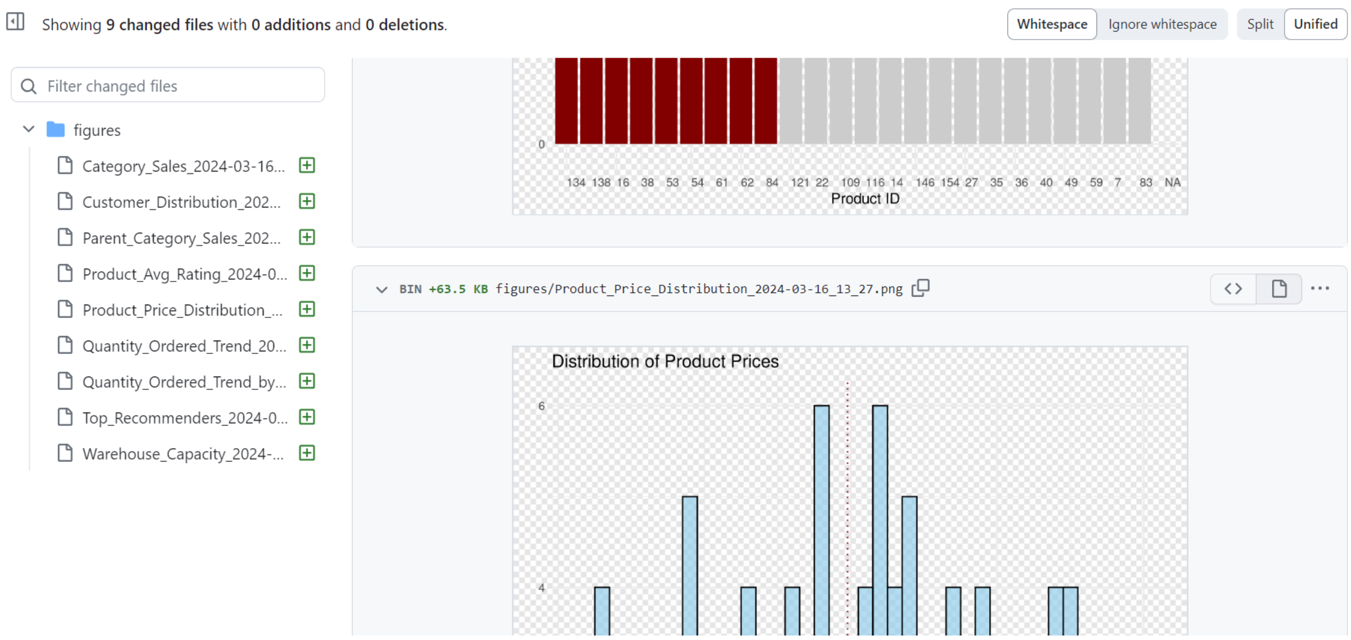
```
        git stash pop
    - name: Push changes
      uses: ad-m/github-push-action@v0.6.0
      with:
          github_token: ${{ secrets.MY_TOKEN }}
          branch: master
```

The workflow action would look like the image below, updating whenever we push an updated script, edit the workflow code, upload new data to the data_upload folder, or for every 12 hours.



Every time it reruns, new plots will be created automatically with updated data (if any) and with a time stamp at the end of the plot name as follows:


LETIMEI / Group31_Data-Management
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Commits
History for Group31_Data-Management / figures on master
All users
All time
Commits on Mar 16, 2024
Add plot figure
LETIMEI committed 9 minutes ago
a7fc7c2
Add plot figure
LETIMEI committed 1 hour ago
39fdc2f
Add plot figure
LETIMEI committed 1 hour ago
bd9fa06
Add plot figure
LETIMEI committed 12 hours ago
daf767f
Add plot figure
LETIMEI committed 14 hours ago
dec8f45


Showing 9 changed files with 0 additions and 0 deletions.
Whitespace
Ignore whitespace
Split
Unified
Filter changed files
figures
Category_Sales_2024-03-16...
Customer_Distribution_202...
Parent_Category_Sales_202...
Product_Avg_Rating_2024-0...
Product_Price_Distribution_...
Quantity_Ordered_Trend_20...
Quantity_Ordered_Trend_by...
Top_Recommenders_2024-0...
Warehouse_Capacity_2024-...
Product ID
BIN +63.5 KB figures/Product_Price_Distribution_2024-03-16_13_27.png
Distribution of Product Prices

4 Part4: Data Analysis

4.1 Task 4.1: Advanced Data Analysis with SQL, R, and ggplot

1. Identify Customers with Most Orders

The below customers showcased a substantial level of engagement with the platform,based on the number of orders they have placed. These insights offers valuable opportunities for targeted marketing efforts and customer retention strategies. Recognizing and rewarding these customers with loyalty programs or special promotions can encourage continued patronage and foster a strong customer-brand relationship.

```
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(*) AS number_of_orders
FROM
    Orders o
JOIN
    Customer c ON o.customer_id = c.customer_id
GROUP BY
    c.customer_id
ORDER BY
    number_of_orders DESC
LIMIT 10;
```

Table 8: Displaying records 1 - 10

customer_id	first_name	last_name	number_of_orders
55041	Eric	Harris	21
61179	Norma	Rodriguez	17
51051	James	Donovan	15
29315	Robert	Nelson	14
25270	Lori	Nelson	14
95869	Jennifer	Tapia	13
22621	Sarah	Martin	13
96525	Peter	Trujillo	12
49127	Luke	Harris	12
29717	David	Collins	12

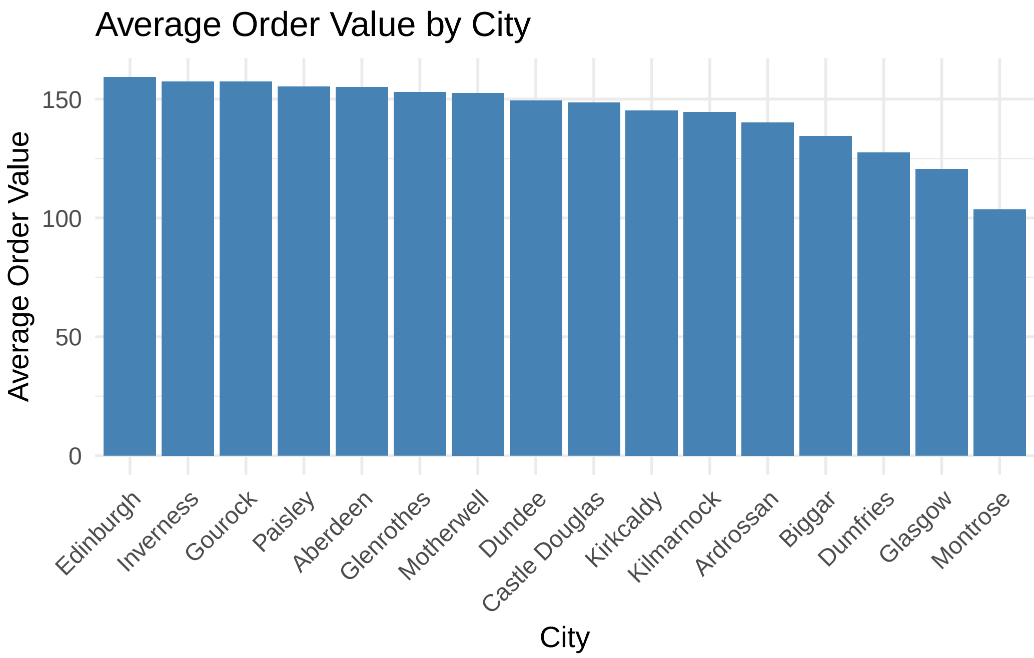
2. Calculate Average Order Value by City

The average order value is calculated for each city, offering insights into regional sales performance, and is calculated by aggregating orders by city and calculating the average value, incorporating product prices, discounts, and shipping charges. Tailoring marketing strategies to regions with lower average order values or reinforcing successful strategies in high-performing areas can optimize sales and customer reach.

```
(top_city <- RSQLite::dbGetQuery(my_connection,"
SELECT
    c.city,
    COUNT(*) AS number_of_orders,
    AVG(o.quantity_of_product_ordered *
    (p.product_price - o.voucher_value) + s.shipping_charge)
    AS avg_order_value
FROM
    Orders o
JOIN
    Shipment s ON o.shipment_id = s.shipment_id
JOIN
    Customer c ON o.customer_id = c.customer_id
JOIN
    Product p ON o.product_id = p.product_id
GROUP BY
    c.city;
"))
```

```
             city number_of_orders avg_order_value
1        Aberdeen               14        154.9429
2      Ardrossan               32        140.1531
3          Biggar               19        134.4053
4  Castle Douglas               14        148.4643
5        Dumfries               23        127.5348
6          Dundee               24        149.3500
7       Edinburgh               22        159.2318
8         Glasgow               61        120.4852
9      Glenrothes               24        152.9333
10        Gourock               14        157.2500
11      Inverness               53        157.4547
12     Kilmarnock               30        144.5800
13      Kirkcaldy                8        145.2500
14       Montrose               30        103.5933
15     Motherwell               22        152.5864
16        Paisley               40        155.2450
```

```r
# Reorder the levels of the city factor based on avg_order_value
top_city$city <- factor(top_city$city,
                        levels = top_city$city[order(-top_city$avg_order_value)])

# Plotting the data with reordered levels
ggplot(top_city, aes(x = city, y = avg_order_value)) +
  geom_bar(stat = "identity", fill = "steelblue") +
  labs(x = "City", y = "Average Order Value",
       title = "Average Order Value by City") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



3. Product Sales Rank

Analyzing product sales performance is critical for businesses to understand product popularity and sales volume. This information helps in identifying top-selling products, evaluating demand trends, and further optimizing decisions regarding product promotions and pricing strategies.

```sql
SELECT
    p.product_id,
    p.product_name,
    COUNT(*) AS number_of_order,
    SUM(o.quantity_of_product_ordered) AS quantity_sold
FROM
```

```
    Orders o
JOIN
    Product p ON o.product_id = p.product_id
GROUP BY
    p.product_id, p.product_name
ORDER BY
    quantity_sold DESC;
```

Table 9: Displaying records 1 - 10

product_id	product_name	number_of_order	quantity_sold
37	Luxury Silk Coat Nexus Collection	10	57
36	Eco-Friendly Polyester Midi dress Black Harmony Series	8	54
56	Elegant Scarf Orange Vitality Series	9	52
58	Minimalist Suede Striped dress Maroon Harmony Series	7	48
134	Minimalist Mesh Denim-Tote-Bag Red Phoenix Series	7	47
33	Modern Wood Leggings Pink Odyssey Series	5	44
52	Stylish Velvet Long skirt Pink Zenith Series	5	44
25	Sporty Stainless Steel Swim dress Teal Vitality Series	6	43
38	Rugged Denim Cargo pants Infinite Horizon Series	7	42
39	Bohemian Wool T-shirt Pink Elysium Line	6	41

4. Total Sold Units by Sub-Category

The below table and diagram provide information regarding the total units of products sold for each sub-category, colored by their respective parent categories. Analyzing sales performance by category helps businesses understand product category popularity and sales volume. This information helps businesses understand hot product categories and those with weak performance; businesses can thus tailor-made different marketing campaign for categories according to this information.

```
# calculate the number of units sold in each (sub)category
# and save the value as top_categ
(top_categ <- RSQLite::dbGetQuery(my_connection,"SELECT
    pc.category_id AS parent_category_id,
    pc.category_name AS parent_category_name,
    c.category_id,
    c.category_name,
    COUNT(o.quantity_of_product_ordered) AS total_sold_unit
FROM
    Orders o
JOIN
    Product p ON o.product_id = p.product_id
```

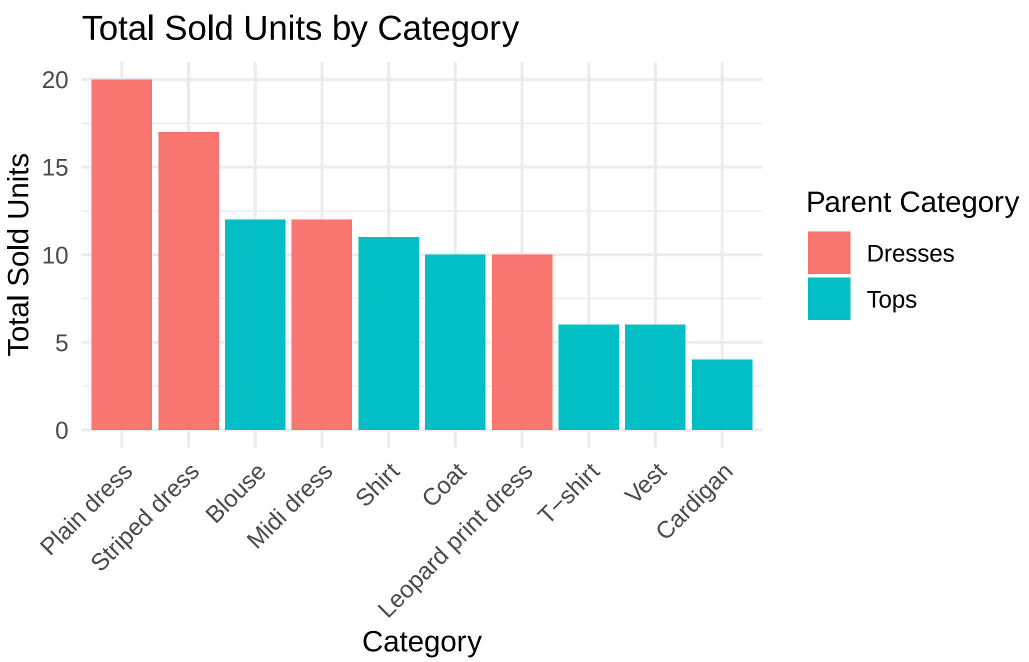
```
JOIN
    Category c ON p.category_id = c.category_id
JOIN
    Category pc ON c.parent_id = pc.category_id
GROUP BY
    pc.category_id, pc.category_name, c.category_id, c.category_name
ORDER BY
    pc.category_id, total_sold_unit DESC
LIMIT 10;
"))
```

```
   parent_category_id parent_category_name category_id       category_name
1                   1                 Tops           8              Blouse
2                   1                 Tops           3               Shirt
3                   1                 Tops           7                Coat
4                   1                 Tops           2             T-shirt
5                   1                 Tops           9                Vest
6                   1                 Tops          11            Cardigan
7                  12              Dresses          15         Plain dress
8                  12              Dresses          17       Striped dress
9                  12              Dresses          21          Midi dress
10                 12              Dresses          13 Leopard print dress
   total_sold_unit
1               12
2               11
3               10
4                6
5                6
6                4
7               20
8               17
9               12
10              10
```

```
# visualize the total sold unit by (sub)category
# and color them with their corresponding parent categories
top_categ_summary <- top_categ %>%
  group_by(category_name, parent_category_name) %>%
  summarise(total_sold_unit = sum(total_sold_unit)) %>%
  #Arrange in descending order based on total_sold_unit
  arrange(desc(total_sold_unit))
```

```
`summarise()` has grouped output by 'category_name'. You can override using the
`.groups` argument.
```

```
# Reorder category_name based on total_sold_unit
top_categ_summary$category_name <- factor(top_categ_summary$category_name,
levels = top_categ_summary$category_name[order(-top_categ_summary$total_sold_unit)])

# Plotting the reordered data
ggplot(top_categ_summary, aes(x = category_name, y = total_sold_unit,
                              fill = parent_category_name)) +
  geom_bar(stat = "identity") +
  labs(x = "Category", y = "Total Sold Units",
       title = "Total Sold Units by Category") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_fill_discrete(name = "Parent Category")
```



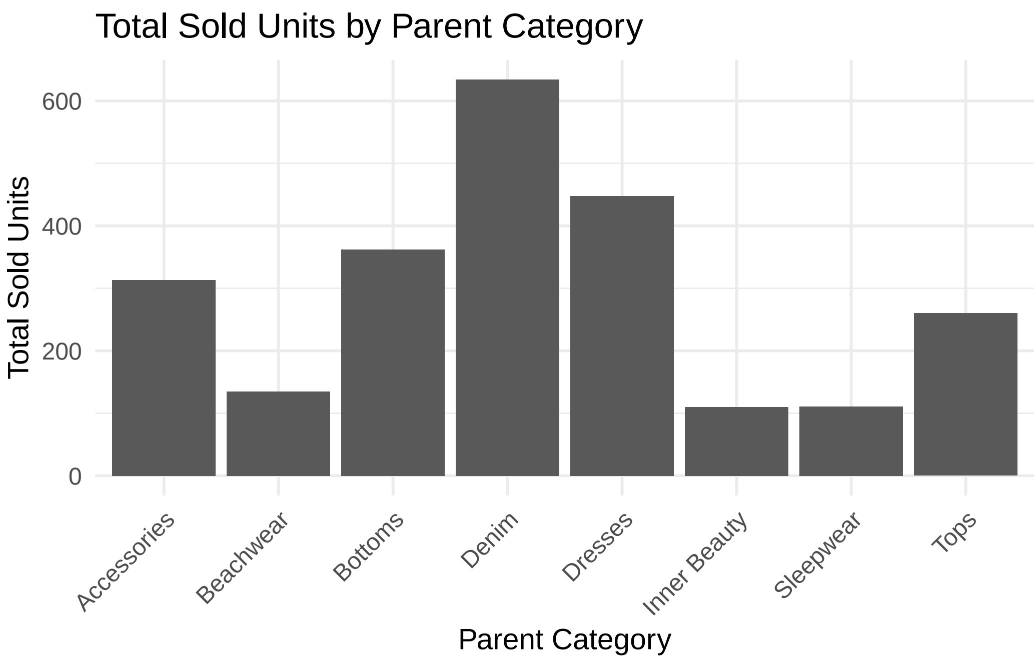
5. Total Sold Units by Parent Category

This diagram further summarizes the total units of products sold for each parent category. Analyzing sales performance at the parent category level provides businesses with information into overall product category performance.

```r
# calculate the sold units for each parent category
# and save the value as top_parent_categ
(top_parent_categ <- RSQLite::dbGetQuery(my_connection,"
SELECT
    pc.category_id AS parent_category_id,
    pc.category_name AS parent_category_name,
    SUM(o.quantity_of_product_ordered) AS total_sold_unit
FROM
    Orders o
JOIN
    Product p ON o.product_id = p.product_id
JOIN
    Category c ON p.category_id = c.category_id
JOIN
    Category pc ON c.parent_id = pc.category_id
GROUP BY
    pc.category_id, pc.category_name
ORDER BY
    total_sold_unit DESC;
"))
```

```
  parent_category_id parent_category_name total_sold_unit
1                 53                Denim             634
2                 12              Dresses             448
3                 22              Bottoms             362
4                 43          Accessories             313
5                  1                 Tops             260
6                 34            Beachwear             135
7                 30            Sleepwear             111
8                 39         Inner Beauty             110
```

```r
# visualize the total sold units by parent category with ggplot bar chart
ggplot(top_parent_categ, aes(x = parent_category_name, y = total_sold_unit
                             )) +
  geom_bar(stat = "identity") +
  labs(x = "Parent Category", y = "Total Sold Units",
       title = "Total Sold Units by Parent Category") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_fill_discrete(name = "Parent Category")
```



6. Top Recommenders

The below table and diagram show the top customers who have referred the highest number of new customers. Understanding and acknowledging top referrers is of integral importance for businesses to encourage and reward loyal customers in the future, foster word-of-mouth marketing, and drive customer acquisition through referral programs.

```
# save the top 20 recommenders in top_recommender
(top_recommender <- RSQLite::dbGetQuery(my_connection,"SELECT
    c1.customer_id AS customer_id,
    CONCAT(c1.first_name, ' ', c1.last_name) AS customer_name,
    COUNT(c2.referral_by) AS referred_number
FROM
    Customer c1
LEFT JOIN
    Customer c2 ON c1.customer_id = c2.referral_by
GROUP BY
    c1.customer_id, c1.first_name, c1.last_name
ORDER BY
    referred_number DESC
LIMIT 20;
"))
```

```
   customer_id    customer_name referred_number
```

```
1        10699       Paige Love           9
2        29206      Jared Porter          8
3        42812    Felicia Rivera          7
4        33995       Jesus Hill           6
5        64370      Paul Daniel           6
6        14385  Lindsey Russell           5
7        58756     Kyle Cabrera           5
8        59043    Michael Glover          5
9        22772    Gregory Morris          4
10       70602 James Strickland           4
11       79997     Charles Jones          4
12       92048      Mary Wright           4
13       16478      Brandi West           3
14       25199     Connie Sawyer          3
15       25270      Lori Nelson           3
16       28178 Natalie Hamilton           3
17       40980     Tracy Harris           3
18       43715  Victoria Thomas           3
19       58368    Jeffrey Carson          3
20       64488    Candice Holmes          3
```

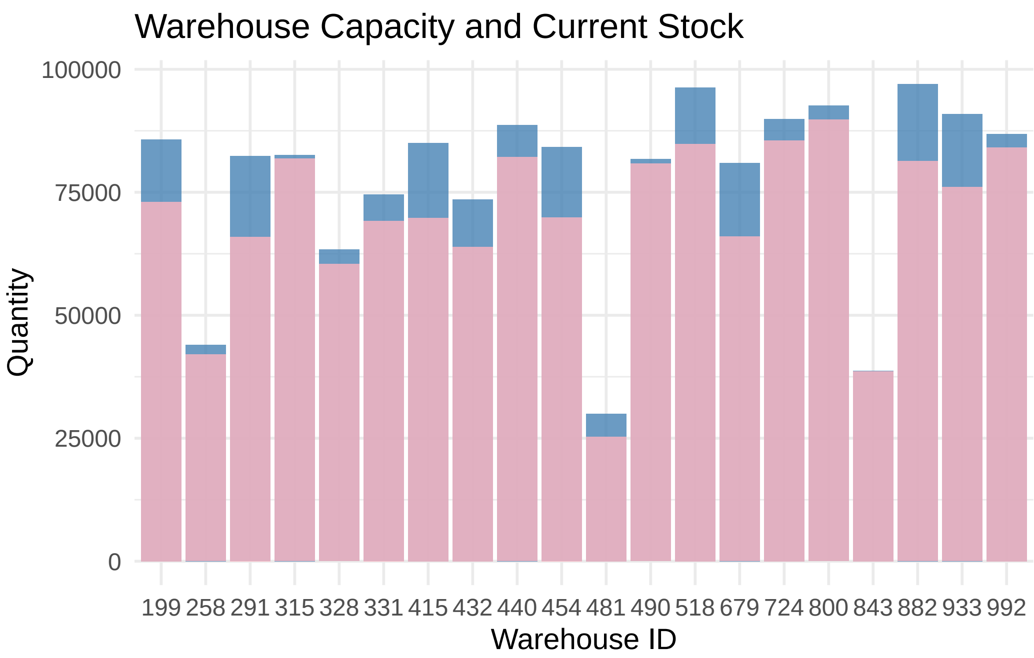
```r
# plot the top 20 recommenders by ggplot
ggplot(top_recommender, aes(x = customer_name, y = referred_number)) +
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(x = "Customer Name", y = "Number of Referrals",
       title = "Top 20 Recommenders") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  coord_flip()
```

7. Warehouse Capacity and Current Stock

The bar plot represents the comparison between warehouse capacity (in blue) and current stock quantities (in pink) for each warehouse; we can easily compare the stock level with the diagram. We filtered only those with current stock more than 80% of the capacity to focus more on those with higher possibility to experience overstocking. Understanding the capacity versus actual stock levels is vital for inventory management and logistics planning. It helps businesses ensure optimal stock levels, and avoid stock-outs or overstocking situations.

```
# filter only those with current stock more than 80% of the capacity
filtered_warehouse <- Warehouse %>%
  filter(current_stock > capacity / 1.25)

# Plotting with filtered data to see the stock level for these warehouses
ggplot(filtered_warehouse, aes(x = warehouse_id)) +
  geom_bar(aes(y = capacity), stat = "identity",
           fill = "steelblue", alpha = 0.8) +
  geom_bar(aes(y = current_stock), stat = "identity",
           fill = "lightpink", alpha = 0.8) +
  labs(title = "Warehouse Capacity and Current Stock",
       x = "Warehouse ID", y = "Quantity") +
  theme_minimal() +
  theme(legend.position = "top") +
  scale_fill_manual(values = c("steelblue", "lightpink"),
                    labels = c("Capacity", "Current Stock"),
```

```
                        name = "Legend") +
  guides(fill = guide_legend(title = "Legend"))
```



8. Product Review Rating Rank

By computing the average rating for each product, the query revealed the top five items with the highest rating. The information about the average rating for each product helps identify highly rated products, which can be used for product recommendations, and increase customer satisfaction.

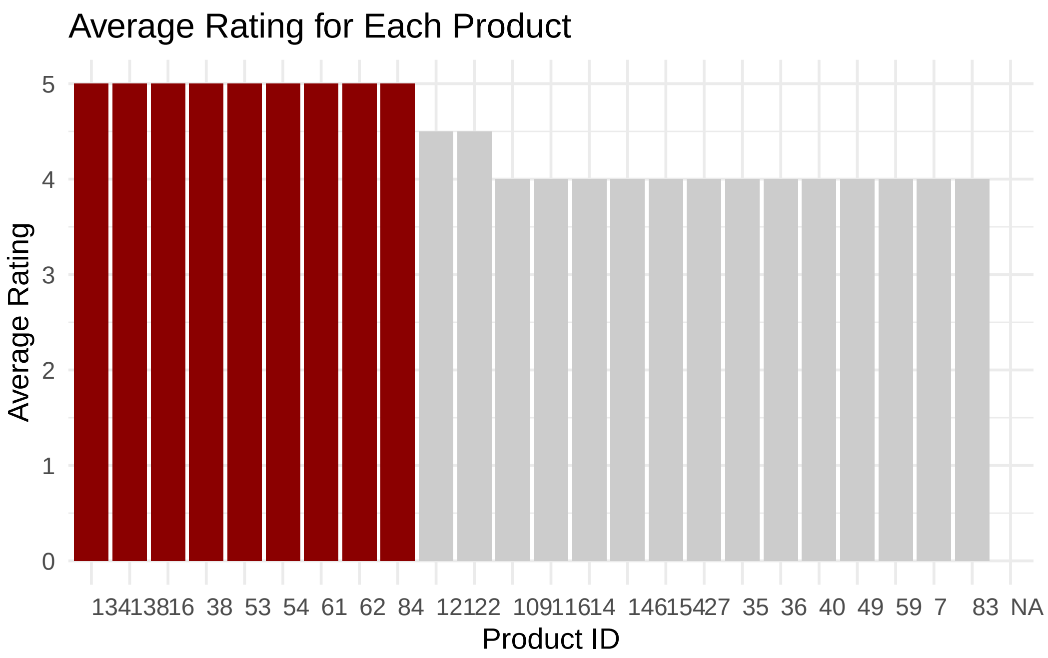
```
# check the data type for 'review_rating' before analyzing
class(Orders$review_rating)
```

```
[1] "character"
```

```
# make sure the data type is numeric so that we can calculate the average value
Orders$review_rating <- as.numeric(Orders$review_rating)

# calculate average rating for each product
(product_ratings <- Orders %>%
  group_by(product_id) %>%
  summarise(avg_rating = mean(review_rating, na.rm = TRUE)) %>%
  arrange(desc(avg_rating)))
```

```
# A tibble: 159 x 2
   product_id avg_rating
   <chr>           <dbl>
 1 134               5
 2 138               5
 3 16                5
 4 38                5
 5 53                5
 6 54                5
 7 61                5
 8 62                5
 9 84                5
10 121               4.5
# i 149 more rows
```

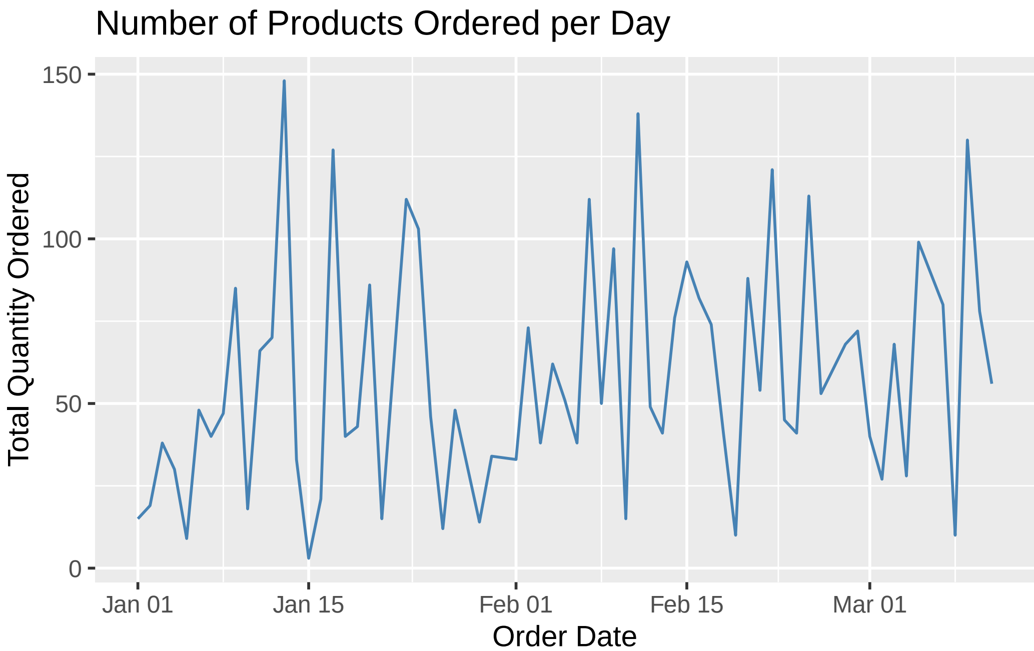
```r
# specify that we only want to show those with an average rating higher than 4
product_ratings <- product_ratings[product_ratings$avg_rating >= 4,]

product_ratings <- product_ratings[order(-product_ratings$avg_rating),]

# define those with average rating=5 as top products
top_products <- product_ratings[product_ratings$avg_rating == 5,]

# visualize the rating by ggplot
ggplot(product_ratings, aes(x = reorder(product_id, -avg_rating),
                            y = avg_rating,
                    fill = factor(product_id %in% top_products$product_id))) +
  geom_bar(stat = "identity") +
  labs(x = "Product ID", y = "Average Rating",
       title = "Average Rating for Each Product") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 00, hjust = 0)) +
  scale_fill_manual(values = c("grey80", "darkred"), guide = FALSE)
```



9. Number of Products Ordered per Day

Tracking the number of products ordered per day helps in identifying peak sales periods, analyzing demand fluctuations, and making decisions related to marketing campaigns. Additionally, it provides insights into customer behavior patterns and can improve forecasting future sales volumes to ensure adequate stock levels. This information can help businesses prepare future promotion on high or low orders period.

```
# ensure that data type are correct before calculating and viualizing them
Orders$order_date <- as.Date(Orders$order_date)
Orders$quantity_of_product_ordered<-as.numeric(Orders$quantity_of_product_ordered)

# calculate the number of products ordered every day
agg_data <- Orders %>%
  group_by(order_date) %>%
  summarise(total_quantity = sum(quantity_of_product_ordered))

# plot the result using ggplot
ggplot(agg_data, aes(x = order_date, y = total_quantity)) +
  geom_line(stat = "identity", color = "steelblue") +
  labs(x = "Order Date", y = "Total Quantity Ordered",
       title = "Number of Products Ordered per Day")
```



10. Units Sold by Parent Category Across Time

Analyzing units sold by each parent category across time helps in identifying top-performing parent categories, tracking sales and market trends over time, and optimizing marketing strategies for different parent categories.

```
# Ensure that data type are the same before joining tables together
Product$product_id <- as.character(Product$product_id)
Orders$product_id <- as.character(Orders$product_id)
Product$category_id <- as.character(Product$category_id)
Category$category_id <- as.character(Category$category_id)
Category$parent_id <- as.character(Category$parent_id)

# use self join for Category table
Category <- Category %>%
  left_join(Category, by = c("parent_id" = "category_id"),
            suffix = c("", "_parent"))

# create the parent_name column based on the join result
Category <- Category %>%
  mutate(parent_name = ifelse(is.na(parent_id), NA, category_name_parent)) %>%
  select(category_id, category_name, parent_id, parent_name)

# calculate unit sold by each parent category across time
sales_data <- Orders %>%
```

```
  inner_join(Product, by = "product_id") %>%
  inner_join(Category, by = "category_id") %>%
  group_by(order_date, parent_id, parent_name) %>%
  summarise(units_sold = sum(quantity_of_product_ordered))
```

```
`summarise()` has grouped output by 'order_date', 'parent_id'. You can override
using the `.groups` argument.
```

```
# filter some parent categories we want to focus
filtered_sales_data <- sales_data %>%
  filter(parent_name %in% c("Denim", "Dresses", "Tops"))

# Plotting the filtered data
ggplot(filtered_sales_data, aes(x = order_date, y = units_sold,
                                color = parent_name)) +
  geom_line() +
  labs(x = "Order Date", y = "Units Sold",
       title = "Units Sold by Parent Category Across Time") +
  scale_color_discrete(name = "Parent Category")
```