Data Management Project

Table of contents

Introduction

This project aims to replicate a real-world e-commerce data environment by covering the entire data management process from database design to data analysis and reporting. The report involves simulating the e-commerce of a university in the United States. SQLite is used to manage the database, GitHub Actions for automation, and Quarto with R for data analysis and reporting.

1 Database Design and Implementation

1.1 E-R Diagram Design

In this section, we explored a data model using an Entity Relationship Diagram (ERD) (Figure 1) to describe the entities and their connections within a fictional e-commerce environment. The ERD plays a vital role in the database design process, providing a clear and concise overview of data requirements and relationships as a conceptual design. Designing an ERD is an essential preliminary step before the database's logical design is implemented.

Various entities such as users, sellers, product, product category, delivery, and shippers, are displayed in an ER diagram. Within the system, each entity represents a unique concept, and the accompanying characteristics provide precise data points to be recorded. For example, the User entity contains details such as User_ID, First_Name, Last_Name, Email, Password, Mobile_Number, Membership_Status, City, Country, State, Type, and PostCode, indicating that it can store information related to users. Ausers' addresses. A payment entity includes attributes such as Payment_ID, Payment_Datetime, Payment_Method, and Payment_Amount that indicate details related to user payments and transactions.In addition, the diagram describes entities such as Shipper, Delivery, Order Details, Seller, Product, and Review along with their respective attributes. For example, a Product entity includes attributes such as Product_ID, Product_Weight, Product_Description, Product_Price, and Published_Date, etc. that indicate details about the products available in the system. Overall, an ER diagram provides a comprehensive overview of the entities and their attributes within a system and offers insight into the potential relationships and functions of various components. This structured representation lays the foundation for the design and implementation of a database system to support the envisioned e-commerce platform, facilitating efficient data management and a seamless user experience.or logistics platform, facilitating efficient data management and a seamless user experience.The page also lists entities and their corresponding characteristics, including Shipper, Delivery, Order Details, Seller, Product, and Review. For example, Product_ID, Product_Weight, Product_Description, Product_Price, and Published_Date are properties of the Product entity that provide information about the items available in the system.An ER diagram provides a thorough picture of the entities and their characteristics within the system and provides insight into the possible interactions and functionality of the various parts. In order to support the intended e-commerce or logistics platform, a database system must be designed and implemented. This will enable efficient data management and a smooth user experience.

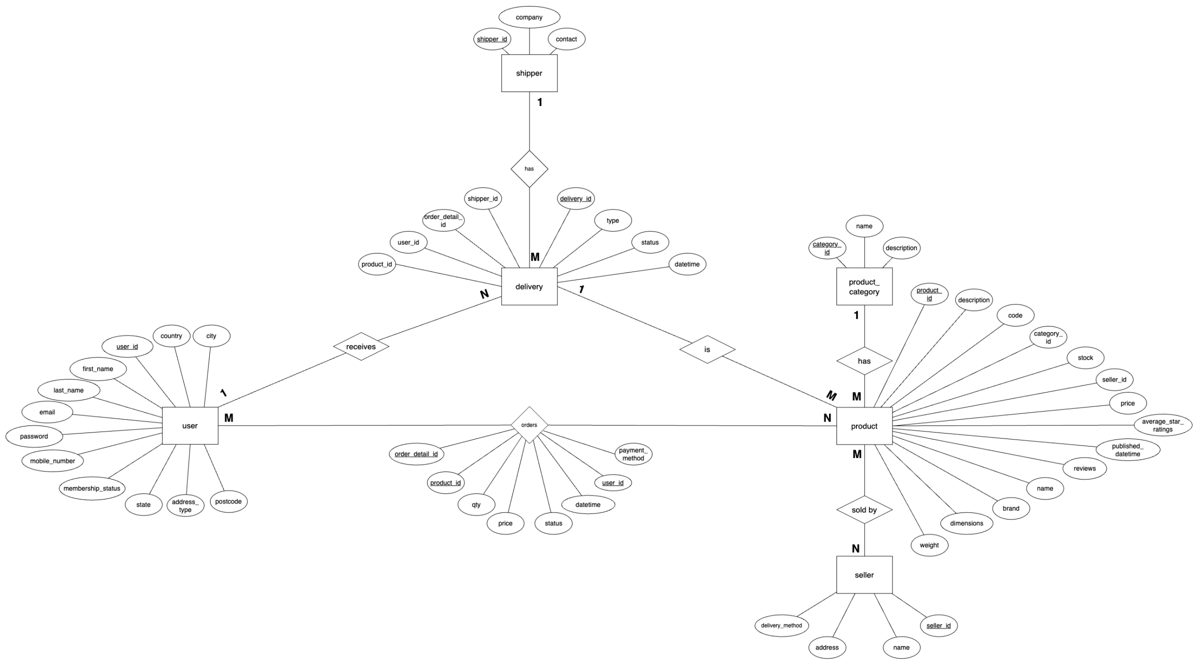


Figure 1: ER Diagram

1.2 SQL Database Schema Creation

After creating an ERD for an e-commerce business, the next step is to convert it into a logical design, which involves organising the attributes of entities into a structured database, such as relational database tables. The ultimate goal is to create well-structured tables that accurately represent the business environment and can store data about entities in a non-repetitive manner. Foreign keys also should be included in the tables to support all relationships among entities (Gillenson, 2012).

To convert an ERD into a relational table, each entity should be converted into a table, and each many-to-many relationship or associative entity should also be converted into a table. However, following specific rules during the conversion is crucial to ensure that foreign keys are placed in their proper locations in the tables (Gillenson, 2012). Therefore, based on the designed ERD (Figure 1), seven tables are being converted, including user, product, product category, order details, seller, delivery, and shipper tables.

The process of conversion involves changing the attributes of an entity into columns of a table, as demonstrated in the example of the User entity (Figure 2), where the attributes such as user__ID, first__name, last__name, email, password, contact, member__status, etc. were converted into columns (Figure 3).

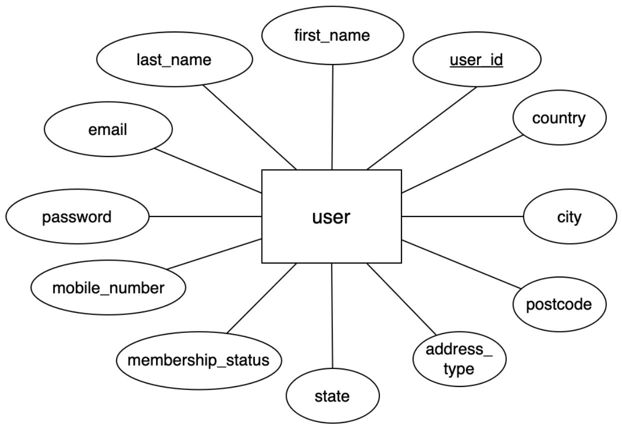
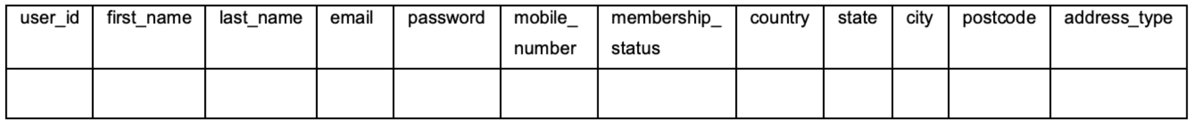


Figure 2: User Entity in ERD

user



user_id	first_name	last_name	email	password	mobile_ number	membership_ status	country	state	city	postcode	address_type

Figure 3: The Converted User Table

This task (1.2) solely focuses on creating tables based on the ERD using SQLite with this query:

```r
# Clearing all existing variables from the workspace
rm(list=ls())

# Loading necessary libraries
library(readr)
library(RSQLite)
library(dplyr)

# Establishing a connection to the SQLite database
my_connection <- dbConnect(SQLite(), "database/database.db")

# Creating the 'user' table
```

```
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS user(
  user_id INT,
  user_first_name VARCHAR(30) NOT NULL,
  user_last_name VARCHAR(30) NOT NULL,
  user_email VARCHAR(100) NOT NULL,
  user_password VARCHAR(20) NOT NULL,
  user_mobile_number VARCHAR(20) NOT NULL,
  user_membership_status VARCHAR(20) NOT NULL,
  address_city VARCHAR(30) NOT NULL,
  address_country VARCHAR(30) NOT NULL,
  address_state VARCHAR(20) NOT NULL,
  address_postcode VARCHAR(20) NOT NULL,
  address_type VARCHAR(20) NOT NULL,
  PRIMARY KEY (user_id)
)")

# Creating the 'product' table
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS product(
  product_id INT,
  product_description VARCHAR(100) NOT NULL UNIQUE,
  product_code VARCHAR(20) NOT NULL UNIQUE,
  category_id INT,
  product_stock INT NOT NULL,
  product_price INT NOT NULL,
  product_name VARCHAR(30) NOT NULL,
  product_brand VARCHAR(30),
  product_weight INT NOT NULL,
  product_dimensions INT NOT NULL,
  product_published_datetime DATE NOT NULL,
  product_average_star_ratings DOUBLE(3,2),
  product_reviews VARCHAR(50),
  seller_id INT,
  PRIMARY KEY (product_id),
  FOREIGN KEY (category_id) REFERENCES product_category(category_id),
  FOREIGN KEY (seller_id) REFERENCES seller(seller_id)
)")

# Creating the 'product_category' table
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS product_category(
  category_id INT,
  category_name VARCHAR(30) NOT NULL,
  category_description VARCHAR(100) NOT NULL,
  PRIMARY KEY (category_id)
)")

# Creating the 'seller' table
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS seller(
  seller_id INT,
  seller_name VARCHAR(20) NOT NULL,
  seller_address VARCHAR(50) NOT NULL,
  seller_delivery_method VARCHAR(20) NOT NULL,
  PRIMARY KEY (seller_id)
)")

# Creating the 'order_details' table
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS order_details(
  order_detail_id INT,
  product_id INT,
  order_qty INT NOT NULL,
  order_price INT NOT NULL,
  order_status VARCHAR(30) NOT NULL,
  order_datetime DATETIME,
  payment_method VARCHAR(20) NOT NULL,
  user_id INT,
  PRIMARY KEY (order_detail_id, product_id, user_id),
  FOREIGN KEY (product_id) REFERENCES product(product_id),
  FOREIGN KEY (user_id) REFERENCES user(user_id)
)")

# Creating the 'delivery' table
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS delivery(
  delivery_id INT,
  delivery_type VARCHAR(20) NOT NULL,
  delivery_status VARCHAR(20) NOT NULL,
  delivery_datetime DATETIME,
  product_id INT,
  user_id INT,
  order_detail_id INT,
  shipper_id INT,
  PRIMARY KEY (delivery_id),
  FOREIGN KEY (product_id) REFERENCES address(product_id),
  FOREIGN KEY (user_id) REFERENCES user(user_id),
  FOREIGN KEY (order_detail_id) REFERENCES order_details(order_detail_id),
  FOREIGN KEY (shipper_id) REFERENCES shipper(shipper_id)
```

```
)")

# Creating the 'shipper' table
dbExecute(my_connection, "CREATE TABLE IF NOT EXISTS shipper(
  shipper_id INT,
  shipper_company VARCHAR(20) NOT NULL,
  shipper_contact VARCHAR(20) NOT NULL,
  PRIMARY KEY (shipper_id)
)")
```

In this case, the user_ID attribute was assigned as the primary key of this table as it is underlined in the ERD and serves as the index of the User entity. We also made a rule that every column here cannot be empty, we set it as NOT NULL, since the details here are needed and a user must only have one address. In addition, we also defined the data type of each column in this process. This process must be repeated to create other tables. After that, we move on to the process of generating synthetic data for the table's content.

2 Data Generation and Management

2.1 Synthetic Data Generation

The e-commerce data represents data for 2023 and assumes that the current date is 01/01/2024.

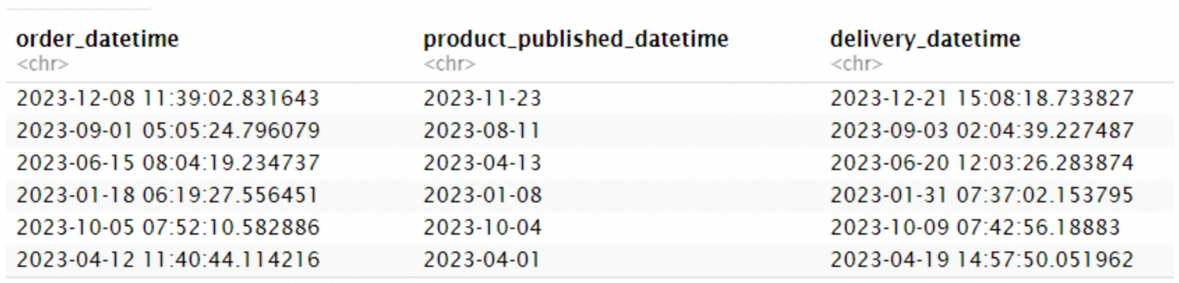
To manufacture the synthetic data, we decided to use a combination of R and Mockaroo, with most of the coding being done by R. If orders were made before the 20th of December they would have been delivered already in the year.

During the synthetic generation, we initially made one large dataset, containing repeating values, (e.g. multiple users bought multiple products). To reflect the normalization process, we then made different tables and ensured uniqueness for the primary keys. Making one large dataset helped give a general overview of the data and we believed this would be better for the analysis stage.

We researched the attributes ensuring we created realistic information. For example, we incorporated weightings of 70% and 30% for the Prime to Not Prime ratio as it roughly resembled the real-life statistics of Amazon USA users. However, we added our twists such as the fact that the prime subscription in our e-commerce would lead to a 5% discount. Given that the background that we created for this company was that it was created by students, we decided to limit the number of categories in the data, as this would be more realistic for a small-scale company.

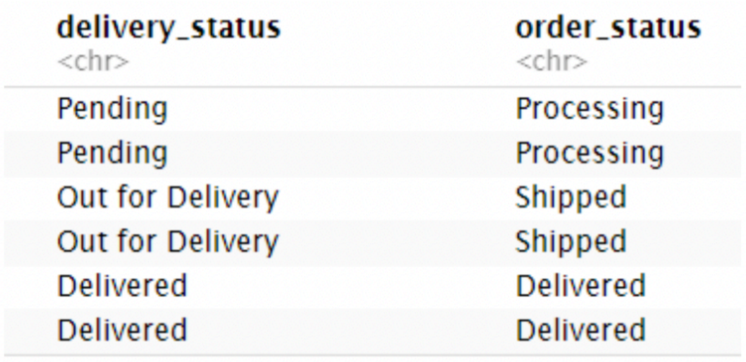
Creating the synthetic data was a highly iterative process, and as discussions progressed constant changes were made. During the forming of a large dataset first, it was crucial to make sure that all logic requirements were met and that there was sense within the data. For example, the dates that we had in our dataset had to be in a temporal order (published date < order date < delivery date). We incorporated logic to update the order status and delivery status based on the temporal attributes and made sure the statuses corresponded to each other correctly. We also ensured that shipping information was associated only with the relevant delivery methods.

By dividing the data into tables and ensuring proper relationships and dependencies we aimed to structure the data efficiently for a relational database.



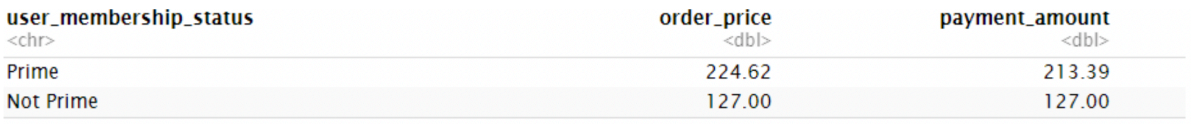
order_datetime <chr>	product_published_datetime <chr>	delivery_datetime <chr>
2023-12-08 11:39:02.831643	2023-11-23	2023-12-21 15:08:18.733827
2023-09-01 05:05:24.796079	2023-08-11	2023-09-03 02:04:39.227487
2023-06-15 08:04:19.234737	2023-04-13	2023-06-20 12:03:26.283874
2023-01-18 06:19:27.556451	2023-01-08	2023-01-31 07:37:02.153795
2023-10-05 07:52:10.582886	2023-10-04	2023-10-09 07:42:56.18883
2023-04-12 11:40:44.114216	2023-04-01	2023-04-19 14:57:50.051962

Figure 4: Ensuring Temporal Order



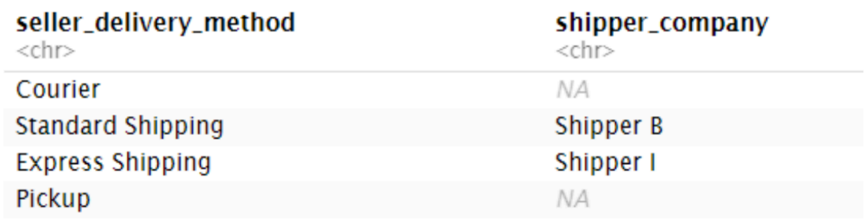
delivery_status <chr>	order_status <chr>
Pending	Processing
Pending	Processing
Out for Delivery	Shipped
Out for Delivery	Shipped
Delivered	Delivered
Delivered	Delivered

Figure 5: Appropriate Logic for Statuses



user_membership_status <chr>	order_price <dbl>	payment_amount <dbl>
Prime	224.62	213.39
Not Prime	127.00	127.00

Figure 6: 5pp Discount (Prime Members)



seller_delivery_method <chr>	shipper_company <chr>
Courier	NA
Standard Shipping	Shipper B
Express Shipping	Shipper I
Pickup	NA

Figure 7: Shipper Information only for Correct Methods

```r
##Part 2.1
library(dplyr)
library(stringi)
#Creation of Large Dataset
#First we endeavoured to create a large dataset which contained all the attributes and values
# Function to generate synthetic data for other columns with logic requirements
generate_synthetic_data <- function(n) {
  # Generate product_published_datetime
  product_published_datetime <- as.POSIXct(sample(seq(as.POSIXct('2023-01-01'), as.POSIXct('2
  # Generate order_datetime based on product_published_datetime
  order_datetime <- product_published_datetime + runif(n, 0, 30*24*60*60) # Assuming order pl
  # Generate payment_datetime based on order_datetime
  payment_datetime <- order_datetime + runif(n, 0, 3*24*60*60) # Assuming payment made within
  # Generate delivery_datetime based on payment_datetime
  delivery_datetime <- payment_datetime + runif(n, 0, 14*24*60*60) # Assuming delivery within
  # Adjust delivery_datetime to ensure the difference between order_datetime and delivery_dat
  delivery_datetime <- pmin(delivery_datetime, order_datetime + 10*24*60*60)
  # Randomize delivery status
  delivery_status <- sample(c("Pending", "Out for Delivery", "Delivered"), n, replace = TRUE)
  # Update delivery_datetime to NA for orders not marked as "Delivered"
  delivery_datetime[delivery_status != "Delivered"] <- NA
  delivery_status[delivery_status == "Out for Delivery" & as.POSIXct(delivery_datetime) == as
  # Randomize order status
  order_status <- sample(c("Pending", "Processing", "Shipped", "Delivered"), n, replace = TRU
  # Update order_status and delivery_status for deliveries before December 25, 2023
  order_status[delivery_datetime < as.POSIXct("2023-12-20")] <- "Delivered"
  delivery_status[delivery_datetime < as.POSIXct("2023-12-20")] <- "Delivered"
  payment_status <- rep("paid", n)
  # Generate a list of product brands with more options
  product_brands <- c("Brand A", "Brand B", "Brand C", "Brand D", "Brand E", "Brand F", "Bran
  # Generate other columns as before
  delivery_id <- sample(1000:9999, n, replace = FALSE)  # Random 4-digit values
  delivery_type <- sample(c("Standard", "Express", "Same-Day", "Pickup"), n, replace = TRUE)
  order_detail_id <- sample(1000:9999, n, replace = FALSE)  # Random 4-digit values
  order_qty <- sample(1:10, n, replace = TRUE)
  product_price <- round(runif(n, 10, 500), 2)  # Round to two decimal places
  order_price <- round(order_qty * product_price, 2)  # Round to two decimal places
  payment_amount <- order_price  # Payment amount equals order price except for when a member
  payment_id <- sample(1000:9999, n, replace = FALSE)  # Random 4-digit values
  payment_method <- sample(c("Credit Card", "Debit Card", "PayPal"), n, replace = TRUE)
  product_average_star_ratings <- runif(n, 1, 5)
  product_brand <- sample(product_brands, n, replace = TRUE)  # Updated to include more produ
  product_code <- sample(1000:9999, n, replace = TRUE)  # Random 4-digit values
  product_description <- stri_rand_strings(n, 10, pattern = "[A-Za-z0-9]")
  product_dimensions <- paste0(sample(1:10, n, replace = TRUE), "x", sample(1:10, n, replace
  product_id <- sample(1000:9999, n, replace = TRUE)  # Random 4-digit values
  product_name <- replicate(n, paste0(sample(LETTERS, 5, replace = TRUE), collapse = ""))  #
  product_reviews <- sample(1:100, n, replace = TRUE)
  product_stock <- sample(1:1000, n, replace = TRUE)
  product_weight <- round(runif(n, 0.1, 50), 2)  # Round to two decimal places
  seller_address <- replicate(n, paste0("Seller Address ", sample(1:100, 1)))
  seller_delivery_method <- sample(c("Standard Shipping", "Express Shipping", "Pickup", "Cou
  seller_id <- sample(1000:9999, n, replace = TRUE)  # Random 4-digit values
  seller_name <- replicate(n, paste0("Seller ", sample(LETTERS, 3, replace = TRUE), collapse
  shipper_company <- sample(c("Shipper A", "Shipper B", "Shipper C", "Shipper D", "Shipper E
  shipper_contact <- ifelse(is.na(shipper_company), NA, sample(1000000000:9999999999, n, rep
  shipper_id <- sample(1000:9999, n, replace = TRUE)  # Random 4-digit values
# Inside your generate_synthetic_data function:
# All deliveries made before December 10th have been delivered
# and ensuring the logic sets order_status and delivery_status to valid values
for (i in 1:n) {
  if (order_datetime[i] < as.POSIXct("2023-12-10")) {
    order_status[i] <- "Delivered"
    delivery_status[i] <- "Delivered"
    delivery_datetime[i] <- order_datetime[i] + runif(1, 1*24*60*60, 14*24*60*60)  # Deliver
  } else {
    # Ensure these statuses are not set to NA or any unintended value
    order_status[i] <- sample(c("Pending", "Processing", "Shipped"), 1)
    delivery_status[i] <- ifelse(order_status[i] == "Shipped", "Out for Delivery", "Pending"
  }
}
# Ensure delivery_status is set to "Delivered" when order_status is "Delivered"
delivery_status[order_status == "Delivered"] <- "Delivered"
  # Combine all the generated data into a single data frame
  synthetic_data <- data.frame(
    delivery_datetime,
    delivery_id,
    delivery_status,
    delivery_type,
    order_datetime,
    order_detail_id,
    order_price,
    order_qty,
    order_status,
    payment_status,
    payment_amount,
    payment_datetime = payment_datetime,
    payment_id,
    payment_method,
    product_average_star_ratings,
    product_brand,
    product_code,
    product_description,
    product_dimensions,
    product_id,
    product_name,
    product_price,
    product_published_datetime,
    product_reviews,
    product_stock,
    product_weight,
    seller_address,
    seller_delivery_method,
    seller_id,
    seller_name,
    shipper_company,
    shipper_contact,
    shipper_id
  )
  return(synthetic_data)
}
# Generate synthetic data for 1400 records
synthetic_data <- generate_synthetic_data(1400)
#Mockaroo dataset
#The Mockaroo data that we read in, contained information about the user as well as category
#Reading in Mockaroo data
mockaroo <- read.csv("mockaroo_data (2).csv")
```

```r
# Change the column names in the mockaroo dataframe to make all the columns consistent
colnames(mockaroo) <- gsub("\\.", "_", colnames(mockaroo))
# Changing address_id to be 6 digits
mockaroo$address_id <- ifelse(nchar(as.character(mockaroo$address_id)) == 6,
                              mockaroo$address_id,
                              sample(100000:999999, nrow(mockaroo), replace = TRUE))
# Changing user_id to be 6 digits
mockaroo$user_id <- ifelse(nchar(as.character(mockaroo$user_id)) == 6,
                              mockaroo$user_id,
                              sample(100000:999999, nrow(mockaroo), replace = TRUE))
# Print the updated column names
print(colnames(mockaroo))
# Check for duplicate user_id values
duplicate_ids <- mockaroo$user_id[duplicated(mockaroo$user_id)]
# If duplicates exist, modify them
if (length(duplicate_ids) > 0) {
  # For simplicity, you could append a suffix or generate a new ID
  # Here's a simple approach to append a suffix to make them unique
  for (id in duplicate_ids) {
    # Find the rows with the duplicated ID
    duplicate_rows <- which(mockaroo$user_id == id)

    # Append a suffix to make each duplicated ID unique
    for (i in seq_along(duplicate_rows)) {
      mockaroo$user_id[duplicate_rows[i]] <- paste(id, i, sep = "_")
    }
  }
}
# Verify that all user_id values are now unique
sum(duplicated(mockaroo$user_id))  # This should return 0 if all IDs are unique
#Category Tidying
#We decided to reduce the number of categories of this artificial e-commerce company later o
#Adding Appropriate Descriptions
# Define the category names
category_names <- c(
  "Electronics", "Clothing", "Shoes & Jewelry", "Toys & Games",
  "Books", "Grocery & Gourmet Food")
# Define the category descriptions
category_descriptions <- c(
  "This category includes products like smartphones, laptops, cameras, televisions, headphon
  "Apparel for men, women, and children are available in this category.",
  "A variety of shoes, jewelry including rings, necklaces, bracelets, and earrings are avail
  "A variety of toys, games, puzzles, and other recreational products for children of all age
  "This category encompasses a vast collection of books including fiction, non-fiction, textb
  "Food and beverage products ranging from pantry staples to gourmet and specialty items are
# Update the category_description column based on category_name
mockaroo <- mutate(mockaroo,
                   category_description = case_when(
                     category_name == "Electronics" ~ category_descriptions[1],
                     category_name == "Clothing" ~ category_descriptions[2],
                     category_name == "Shoes & Jewelry" ~ category_descriptions[3],
                     category_name == "Toys & Games" ~ category_descriptions[4],
                     category_name == "Books" ~ category_descriptions[5],
                     category_name == "Grocery & Gourmet Food" ~ category_descriptions[6],
                     TRUE ~ NA_character_
                   ))
#Incorporating logic with Category Information
# First, ensure that each unique category name and description pair has a unique ID
category_mapping <- mockaroo %>%
  distinct(category_name, category_description) %>%
  mutate(category_id = row_number())  # Assign a new unique ID
# Join the original mockaroo dataframe with the category_mapping to update category_ids
mockaroo <- mockaroo %>%
  select(-category_id) %>%
  left_join(category_mapping, by = c("category_name", "category_description"))
# Print the modified data frame
print(mockaroo)
#Duplicating user_id
#We created repeating user_id values and made sure that all user information and address info
# Define the maximum number of repeats for each user
max_repeats <- 3
# Create a dataframe to store the repeated user information
repeated_users <- mockaroo %>%
  group_by(user_id) %>%
  slice(rep(1:n(), each = sample(1:max_repeats, length(user_id), replace = TRUE)))
# Print the first few rows to verify the structure
print(head(repeated_users))
# Now, let's ensure that 'user_id' correctly corresponds to the correct user information
# Group by 'user_id' and sample the values for user information
repeated_users <- repeated_users %>%
  group_by(user_id) %>%
  mutate(
    user_first_name = sample(user_first_name, 1),
    user_last_name = sample(user_last_name, 1),
    user_email = sample(user_email, 1),
    user_membership_status = sample(user_membership_status, 1)
  )
# Now, let's ensure that 'user_ID' correctly corresponds to the correct address information
# Group by 'user_ID' and sample the values for address information
repeated_users <- repeated_users %>%
  group_by(user_id) %>%
  mutate(
    address_id = sample(address_id, 1),
    address_city = sample(address_city, 1),
    address_country = sample(address_country, 1),
    address_postcode = sample(address_postcode, 1),
    address_state = sample(address_state, 1),
    address_type = sample(address_type, 1)
  )
# Print the updated dataframe
print(repeated_users)
# Check for duplicated user_ID values
duplicated_user_ids <- repeated_users[duplicated(repeated_users$user_id), "user_id"]
# Print duplicated user IDs
print(duplicated_user_ids)
# Count the number of duplicated user_ID values
num_repeated_values <- sum(duplicated(repeated_users$user_id))
# Print the number of repeated values
print(num_repeated_values)
# Ensure it has exactly 1400 rows
# If the dataset has more than 1400 rows, truncate it
if (nrow(repeated_users) > 1400) {
  repeated_users <- repeated_users[1:1400, ]
}
# If the dataset has fewer than 2000 rows, duplicate rows to reach 2000
while (nrow(repeated_users) < 1400) {
  repeated_users <- rbind(repeated_users, repeated_users[1:(1400 - nrow(mockaroo)), ])
}
# Limit the dataframe to 2000 rows
mockaroo_final <- head(repeated_users, 1400)
# Print the limited dataframe
print(mockaroo_final)
# Combine the limited dataframe with the synthetic data
combined_data <- cbind(mockaroo_final, synthetic_data)
# Write the combined dataset to a new CSV file
write.csv(combined_data, "combined_mockaroo_with_synthetic.csv", row.names = FALSE)
```

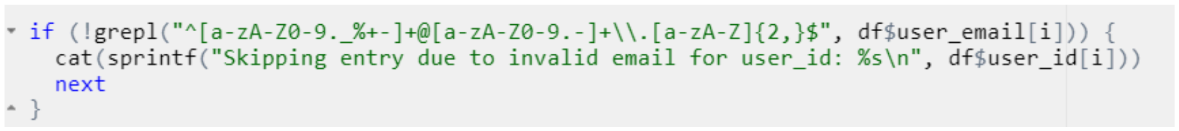
```r
# Check if the CSV file is created
file.exists("combined_mockaroo_with_synthetic.csv")
#Incorporating Some More logic and Data Tidying
# Read the combined data from the CSV file
combined_data <- read.csv("combined_mockaroo_with_synthetic.csv")

combined_data <- combined_data %>%
  mutate(payment_amount = if_else(user_membership_status == "Prime",
                                  round(payment_amount * 0.95, 2),
                                  payment_amount))
#Ensuring that we only have shipping information, when the seller delivery method is express
# Create a vector of shipper companies, IDs, and contacts from A to J
shippers <- data.frame(
  shipper_company = paste0("Shipper ", LETTERS[1:10]),
  shipper_id = sample(1000:9999, 10, replace = FALSE),  # Random 4-digit values for shipper I
  shipper_contact = sample(1000000000:9999999999, 10, replace = FALSE)  # Random 10-digit val
)
# Identify rows where the seller delivery method is Express Shipping or Standard Shipping
express_standard_rows <- combined_data$seller_delivery_method %in% c("Express Shipping", "Sta
# Assign shipper information only to rows with Express Shipping or Standard Shipping
combined_data$shipper_company[express_standard_rows] <- sample(shippers$shipper_company, sum
combined_data$shipper_id[express_standard_rows] <- sample(shippers$shipper_id, sum(express_st
combined_data$shipper_contact[express_standard_rows] <- sample(shippers$shipper_contact, sum
# For other rows, set shipper information to NA
combined_data$shipper_company[!express_standard_rows] <- NA
combined_data$shipper_id[!express_standard_rows] <- NA
combined_data$shipper_contact[!express_standard_rows] <- NA
# Ensuring values are rounded to 2 decimal places
combined_data$product_average_star_ratings <- round(combined_data$product_average_star_rating
# Group the data by product_code and sample one value for each column
combined_data <- combined_data %>%
  group_by(product_code) %>%
  mutate(
    product_brand = sample(product_brand, 1),
    product_dimensions = sample(product_dimensions, 1),
    product_average_star_ratings = sample(product_average_star_ratings, 1),
    product_name = sample(product_name, 1),
    product_price = sample(product_price, 1),
    product_published_datetime = sample(product_published_datetime, 1),
    product_reviews = sample(product_reviews, 1)
  ) %>%
  ungroup()
```

```
#Note that we kept product_ID unique
# Ensure order_status is Processing if delivery_status is Pending
combined_data$order_status <- ifelse(combined_data$delivery_status == "Pending", "Processing"
# Ensure order_status is Shipped if delivery_status is Out for Delivery
combined_data$order_status <- ifelse(combined_data$delivery_status == "Out for Delivery", "Sh
# Ensure order_status is Pending if payment_status is Pending
combined_data$order_status <- ifelse(combined_data$payment_status == "pending", "Pending", co
# Since the payment is made, the order is considered at least "Processing"
combined_data$order_status <- ifelse(combined_data$order_status %in% c("Pending", "Processing
# Assuming payment_datetime should be after order_datetime but before or equal to the delive
combined_data$payment_datetime <- ifelse(is.na(combined_data$payment_datetime) | combined_dat
                                          combined_data$order_datetime + runif(nrow(combined_d
                                          combined_data$payment_datetime)
combined_data$payment_datetime <- pmin(combined_data$payment_datetime, combined_data$delivery
# Saving combined_data as CSV
write.csv(combined_data, "final_ecommerce.csv", row.names = FALSE)
#Making Multiple Tables
#Having formed the large dataset, we are now going to create separate tables for each entity
# 1. Users Table
user_table <- combined_data %>%
  distinct(user_id, user_email, user_mobile_number, .keep_all = TRUE) %>%
  select(user_id, user_first_name, user_last_name, user_email, user_password, user_mobile_num
#write.csv(user_table, "users_table.csv", row.names = FALSE)
# 2. Products Table
products_table <- combined_data %>%
  distinct(product_id, .keep_all = TRUE) %>%
  select(product_id, product_description, product_code, category_id, product_stock, product_p
#write.csv(products_table, "products_table.csv", row.names = FALSE)
# 3. Product Categories Table
product_categories_table <- combined_data %>%
  distinct(category_id, .keep_all = TRUE) %>%
  select(category_id, category_name, category_description)
#write.csv(product_categories_table, "product_categories_table.csv", row.names = FALSE)
# 4. Sellers Table
sellers_table <- combined_data %>%
  distinct(seller_id, .keep_all = TRUE) %>%
  select(seller_id, seller_name, seller_address, seller_delivery_method)  # Assuming product_
#write.csv(sellers_table, "sellers_table.csv", row.names = FALSE)
# 5. Order_details Table
if (nrow(combined_data) != nrow(distinct(combined_data, order_detail_id))) {
  stop("Duplicate order_detail_id found.")
}
```

```
order_details_table <- combined_data %>%
  select(order_detail_id, product_id, order_qty, order_price, order_status, order_datetime, u
#write.csv(order_details_table, "order_details_table.csv", row.names = FALSE)
# 6. Payments Table
payments_table <- combined_data %>%
  distinct(payment_id, .keep_all = TRUE) %>%
  select(payment_id, payment_datetime, payment_method, payment_amount, user_id, order_detail_
#write.csv(payments_table, "payments_table.csv", row.names = FALSE)
# 7. Deliveries Table
deliveries_table <- combined_data %>%
  distinct(delivery_id, .keep_all = TRUE) %>%
  select(delivery_id, delivery_type, delivery_status, delivery_datetime, address_id, user_id
#write.csv(deliveries_table, "deliveries_table.csv", row.names = FALSE)
# 8. Shippers Table
shippers_table <- combined_data %>%
  distinct(shipper_id, .keep_all = TRUE) %>%
  select(shipper_id, shipper_company, shipper_contact)
#write.csv(shippers_table, "shippers_table.csv", row.names = FALSE)
```

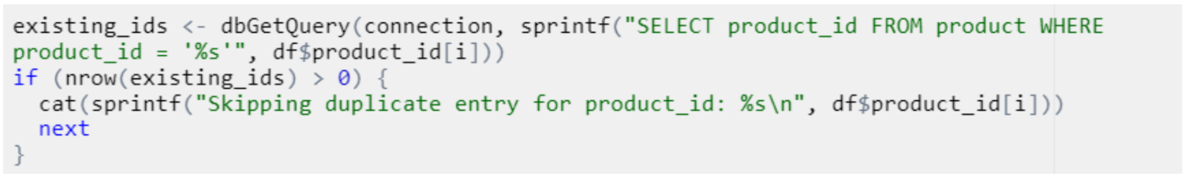
2.2 Data Import and Quality Assurance

The next stage involved populating the SQL database with the generated data by ingestion. It involved validating email formats, filtering out missing values and ensuring uniqueness when needed (primary keys). We also performed referential integrity checks (validated the existence of referenced attributes before inserting the data). Each data ingestion function also included error handling using "tryCatch" to manage any issues that occur during the insertion process and to provide feedback on the success or failure of each operation.



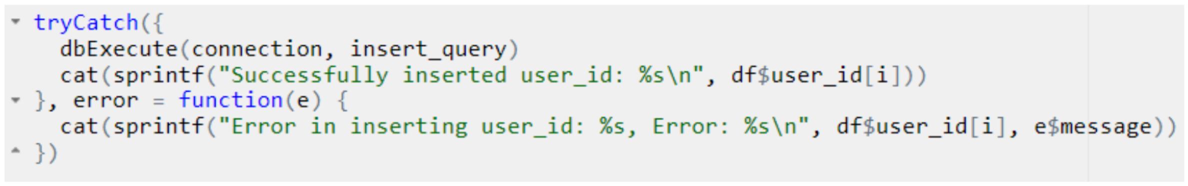
```
if (!grepl("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$", df$user_email[i])) {
  cat(sprintf("Skipping entry due to invalid email for user_id: %s\n", df$user_id[i]))
  next
}
```

Figure 8: Email Validity



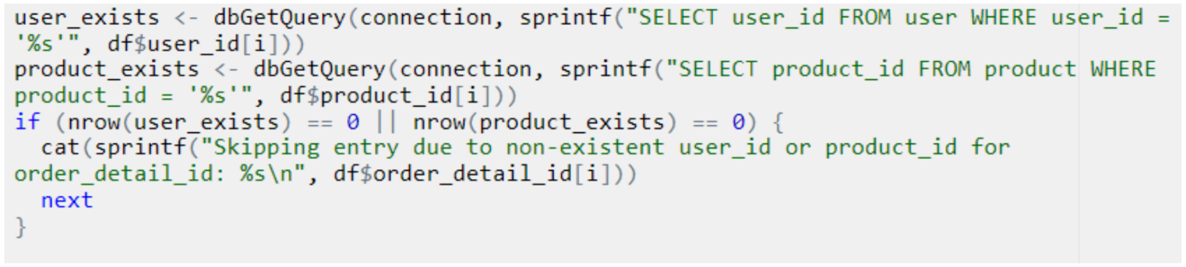
```r
existing_ids <- dbGetQuery(connection, sprintf("SELECT product_id FROM product WHERE
product_id = '%s'", df$product_id[i]))
if (nrow(existing_ids) > 0) {
  cat(sprintf("Skipping duplicate entry for product_id: %s\n", df$product_id[i]))
  next
}
```

Figure 9: Duplication Checks



```r
tryCatch({
    dbExecute(connection, insert_query)
    cat(sprintf("Successfully inserted user_id: %s\n", df$user_id[i]))
}, error = function(e) {
    cat(sprintf("Error in inserting user_id: %s, Error: %s\n", df$user_id[i], e$message))
})
```

Figure 10: tryCatch for feedback



```r
user_exists <- dbGetQuery(connection, sprintf("SELECT user_id FROM user WHERE user_id =
'%s'", df$user_id[i]))
product_exists <- dbGetQuery(connection, sprintf("SELECT product_id FROM product WHERE
product_id = '%s'", df$product_id[i]))
if (nrow(user_exists) == 0 || nrow(product_exists) == 0) {
  cat(sprintf("Skipping entry due to non-existent user_id or product_id for
order_detail_id: %s\n", df$order_detail_id[i]))
  next
}
```

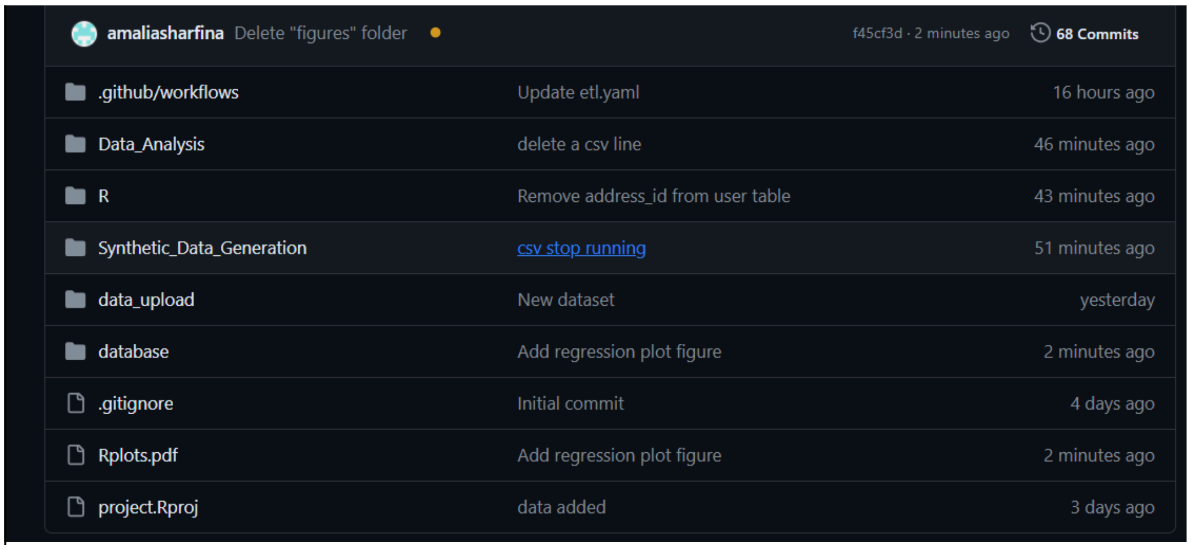
Figure 11: Referential Integrity

```r
##Part 2.2

#Ingesting, Data Validation and Referential Integrity Checks

library(readr)
library(RSQLite)
library(dplyr)

# Ingest User Data Function
ingest_user_data <- function(df, connection) {
  required_columns <- c("user_id", "user_email", "user_password", "user_mobile_number", "addi
                        "address_city", "address_country", "address_state", "address_postcode", "

  # Filter out rows with NA in any of the required columns
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]
  for (i in 1:nrow(df)) {
    # Check for duplicate user_id
    existing_users <- dbGetQuery(connection, sprintf("SELECT user_id FROM user WHERE user_id
    if (nrow(existing_users) > 0) {
      cat(sprintf("Skipping duplicate entry for user_id: %s\n", df$user_id[i]))
      next
    }

    # Check for valid email
    if (!grepl("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$", df$user_email[i])) {
      cat(sprintf("Skipping entry due to invalid email for user_id: %s\n", df$user_id[i]))
      next
    }

    # Prepare the INSERT statement including user and address information
    insert_query <- sprintf("INSERT INTO user (user_id, user_first_name, user_last_name, use
                            df$user_id[i], df$user_first_name[i], df$user_last_name[i], df$u
                            df$user_password[i], df$user_mobile_number[i], df$user_membershi
                            df$address_id[i], df$address_city[i], df$address_country[i], df$
                            df$address_postcode[i], df$address_type[i])

    # Execute the INSERT statement and handle any errors
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted user_id: %s\n", df$user_id[i]))
    }, error = function(e) {
      cat(sprintf("Error in inserting user_id: %s, Error: %s\n", df$user_id[i], e$message))
    })
  }
}

# Ingest Product Data Function
ingest_product_data <- function(df, connection) {
  required_columns <- c("product_id", "product_name", "product_price", "product_description"
                        "product_stock", "product_code", "product_brand", "product_weight",
                        "product_published_datetime", "product_average_star_ratings", "produ

  # Filter out rows with NA in any of the required columns
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  for(i in 1:nrow(df)) {
    # Check for duplicate product_id
    existing_ids <- dbGetQuery(connection, sprintf("SELECT product_id FROM product WHERE proc
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for product_id: %s\n", df$product_id[i]))
      next
    }

    # Check for valid price
    if(!is.numeric(df$product_price[i]) || df$product_price[i] <= 0) {
      cat(sprintf("Skipping entry due to invalid price for product_id: %s\n", df$product_id[i
      next
    }

    # Prepare the INSERT statement with all required columns
    insert_query <- sprintf("INSERT INTO product (product_id, product_name, product_descripti
                             category_id, product_stock, product_code, product_brand, product
                             product_dimensions, product_published_datetime, product_average_
                             product_reviews) VALUES ('%s', '%s', '%s', %f, '%s', %d, '%s',
                            df$product_id[i], df$product_name[i], df$product_description[i],
                            df$category_id[i], df$product_stock[i], df$product_code[i], df$pr
                            df$product_weight[i], df$product_dimensions[i], df$product_publis
                            df$product_average_star_ratings[i], df$product_reviews[i])

    # Execute the INSERT statement and handle any errors
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted row: %d\n", i))
    }, error = function(e) {
      cat(sprintf("Error in inserting row: %d, Error: %s\n", i, e$message))
    })
  }
}

# Function to ingest product_category data
ingest_product_category_data <- function(df, connection) {
  required_columns <- c("category_id", "category_name", "category_description")
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  for(i in 1:nrow(df)) {
    # Check for duplicate category_id
    existing_ids <- dbGetQuery(connection, sprintf("SELECT category_id FROM product_category
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for category_id: %s\n", df$category_id[i]))
      next
    }

    # Insert validated data into the database
    insert_query <- sprintf("INSERT INTO product_category (category_id, category_name, catego
                            df$category_id[i], df$category_name[i], df$category_description[i
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted category_id: %s\n", df$category_id[i]))
    }, error = function(e) {
      cat(sprintf("Error in inserting category_id: %s, Error: %s\n", df$category_id[i], e$mes
    })
  }
}

# Function to ingest order_details data
ingest_order_details_data <- function(df, connection) {
  required_columns <- c("order_detail_id", "product_id", "order_qty", "order_price",
                        "order_status", "order_datetime", "user_id", "category_id")
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  for(i in 1:nrow(df)) {
    # Check for duplicate order_detail_id
    existing_ids <- dbGetQuery(connection, sprintf("SELECT order_detail_id FROM order_details
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for order_detail_id: %s\n", df$order_detail_id[i]
      next
    }

    # Ensure referenced user_id, product_id, and category_id exist
    user_exists <- dbGetQuery(connection, sprintf("SELECT user_id FROM user WHERE user_id =
    product_exists <- dbGetQuery(connection, sprintf("SELECT product_id FROM product WHERE pr
    category_exists <- dbGetQuery(connection, sprintf("SELECT category_id FROM product_catego

    if(nrow(user_exists) == 0 || nrow(product_exists) == 0 || nrow(category_exists) == 0) {
      cat(sprintf("Skipping entry due to non-existent user_id, product_id, or category_id fo
      next
    }

    # Insert validated data into the database
    insert_query <- sprintf("INSERT INTO order_details (order_detail_id, product_id, order_q
                            df$order_detail_id[i], df$product_id[i], df$order_qty[i], df$orde
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted order_detail_id: %s\n", df$order_detail_id[i]))
    }, error = function(e) {
      cat(sprintf("Error in inserting order_detail_id: %s, Error: %s\n", df$order_detail_id[i
    })
  }
}

# Function to ingest delivery data
ingest_delivery_data <- function(df, connection) {
  required_columns <- c("delivery_id", "delivery_type", "delivery_status", "user_id", "order_
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  for(i in 1:nrow(df)) {
    # Check for duplicate delivery_id
    existing_ids <- dbGetQuery(connection, sprintf("SELECT delivery_id FROM delivery WHERE de
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for delivery_id: %s\n", df$delivery_id[i]))
      next
    }

    # Ensure referenced user_id, and order_detail_id exist
    user_exists <- dbGetQuery(connection, sprintf("SELECT user_id FROM user WHERE user_id =
    order_exists <- dbGetQuery(connection, sprintf("SELECT order_detail_id FROM order_details
    if(nrow(user_exists) == 0 || nrow(order_exists) == 0 || nrow(address_exists) == 0) {
      cat(sprintf("Skipping entry due to non-existent user_id, order_detail_id for delivery_i
      next
    }

    # Insert validated data into the database
    insert_query <- sprintf("INSERT INTO delivery (delivery_id, delivery_type, delivery_statu
                            df$delivery_id[i], df$delivery_type[i], df$delivery_status[i], di
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted delivery_id: %s\n", df$delivery_id[i]))
    }, error = function(e) {
      cat(sprintf("Error in inserting delivery_id: %s, Error: %s\n", df$delivery_id[i], e$mes
    })
  }
}
```

```r
# Function to ingest seller data
ingest_seller_data <- function(df, connection) {
  required_columns <- c("seller_id", "seller_name", "seller_address", "seller_delivery_metho
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  for(i in 1:nrow(df)) {
    # Check for duplicate seller_id
    existing_ids <- dbGetQuery(connection, sprintf("SELECT seller_id FROM seller WHERE selle
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for seller_id: %s\n", df$seller_id[i]))
      next
    }

    # Insert validated data into the database
    insert_query <- sprintf("INSERT INTO seller (seller_id, seller_name, seller_address, sel
                            df$seller_id[i], df$seller_name[i], df$seller_address[i], df$sel
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted seller_id: %s\n", df$seller_id[i]))
    }, error = function(e) {
      cat(sprintf("Error in inserting seller_id: %s, Error: %s\n", df$seller_id[i], e$messag
    })
  }
}

# Function to ingest shipper data
ingest_shipper_data <- function(df, connection) {
  required_columns <- c("shipper_id", "shipper_company", "shipper_contact")
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  for(i in 1:nrow(df)) {
    # Check for duplicate shipper_id
    existing_ids <- dbGetQuery(connection, sprintf("SELECT shipper_id FROM shipper WHERE shi
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for shipper_id: %s\n", df$shipper_id[i]))
      next
    }

    # Insert validated data into the database
    insert_query <- sprintf("INSERT INTO shipper (shipper_id, shipper_company, shipper_conta
                            df$shipper_id[i], df$shipper_company[i], df$shipper_contact[i])
    tryCatch({
      dbExecute(connection, insert_query)
      cat(sprintf("Successfully inserted shipper_id: %s\n", df$shipper_id[i]))
    }, error = function(e) {
      cat(sprintf("Error in inserting shipper_id: %s, Error: %s\n", df$shipper_id[i], e$messa
    })
  }
}

# Load the data from CSV files or data frames
user_df <- read_csv("Synthetic_Data_Generation/users_table.csv")
product_df <- read_csv("Synthetic_Data_Generation/products_table.csv")
product_category_df <- read_csv("Synthetic_Data_Generation/product_categories_table.csv")
order_details_df <- read_csv("Synthetic_Data_Generation/order_details_table.csv")
delivery_df <- read_csv("Synthetic_Data_Generation/deliveries_table.csv")
seller_df <- read_csv("Synthetic_Data_Generation/sellers_table.csv")
shipper_df <- read_csv("Synthetic_Data_Generation/shippers_table.csv")

# Establish a connection to the SQLite database
my_connection <- dbConnect(RSQLite::SQLite(), "database/database.db")

# Ingest the data into the database
ingest_user_data(user_df, my_connection)
ingest_product_category_data(product_category_df, my_connection)
ingest_product_data(product_df, my_connection)
ingest_order_details_data(order_details_df, my_connection)
ingest_delivery_data(delivery_df, my_connection)
ingest_seller_data(seller_df, my_connection)
ingest_shipper_data(shipper_df, my_connection)

# Close the database connection
dbDisconnect(my_connection)
```

3 Data Pipeline Generation

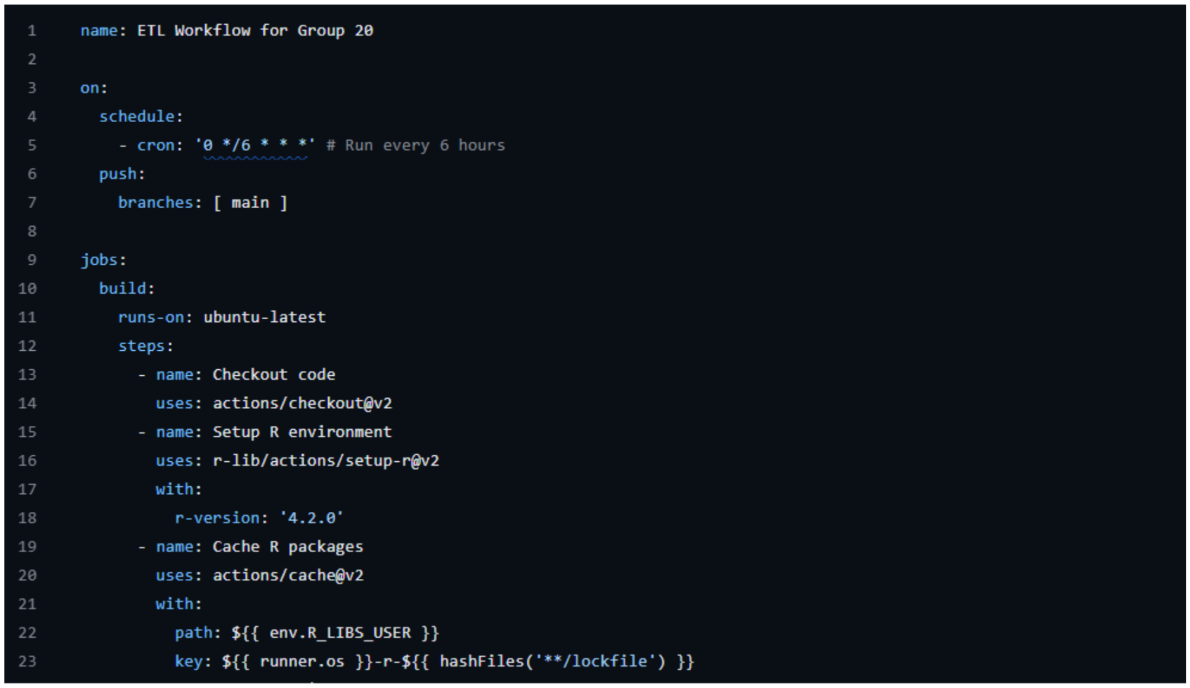
The GitHub repository "data-management-group-20" has been configured with a comprehensive workflow setup leveraging GitHub Actions for continuous integration. The YAML file, titled "ETL Workflow for Group 20," defines the workflow triggered by pushes to the main branch.



amaliasharfina Delete "figures" folder	f45cf3d · 2 minutes ago	68 Commits
.github/workflows	Update etl.yaml	16 hours ago
Data_Analysis	delete a csv line	46 minutes ago
R	Remove address_id from user table	43 minutes ago
Synthetic_Data_Generation	csv stop running	51 minutes ago
data_upload	New dataset	yesterday
database	Add regression plot figure	2 minutes ago
.gitignore	Initial commit	4 days ago
Rplots.pdf	Add regression plot figure	2 minutes ago
project.Rproj	data added	3 days ago

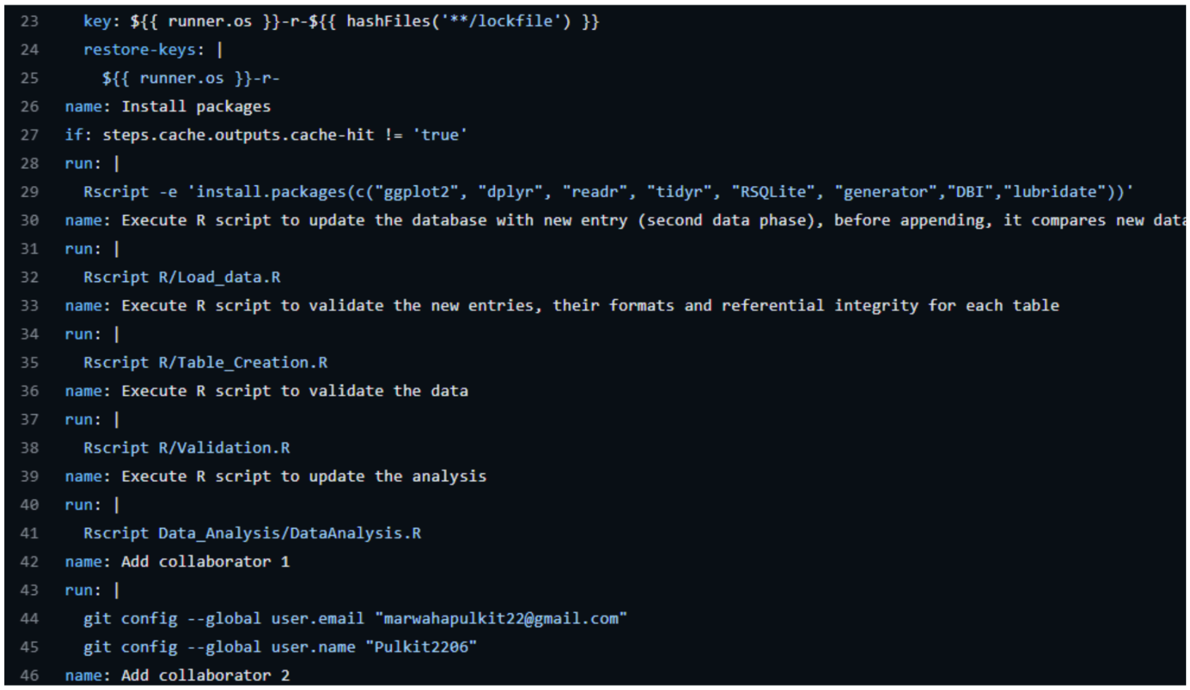
Figure 12: Github Folder

The workflow runs on the latest Ubuntu environment and consists of multiple jobs orchestrated to execute various tasks. These tasks include checking out the code, setting up the R environment with the specified version, caching R packages to optimise workflow speed, and installing necessary packages using Rscript.



```yaml
name: ETL Workflow for Group 20

on:
  schedule:
    - cron: '0 */6 * * *' # Run every 6 hours
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Setup R environment
        uses: r-lib/actions/setup-r@v2
        with:
          r-version: '4.2.0'
      - name: Cache R packages
        uses: actions/cache@v2
        with:
          path: ${{ env.R_LIBS_USER }}
          key: ${{ runner.os }}-r-${{ hashFiles('**/lockfile') }}
```

Figure 13: workflow for initial setup and installing of packages



```
  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("ggplot2", "dplyr", "readr", "tidyr", "RSQLite", "generator","DBI","lubridate"))'
name: Execute R script to update the database with new entry (second data phase), before appending, it compares new dat
run: |
  Rscript R/Load_data.R
name: Execute R script to validate the new entries, their formats and referential integrity for each table
run: |
  Rscript R/Table_Creation.R
name: Execute R script to validate the data
run: |
  Rscript R/Validation.R
name: Execute R script to update the analysis
run: |
  Rscript Data_Analysis/DataAnalysis.R
name: Add collaborator 1
run: |
  git config --global user.email "marwahapulkit22@gmail.com"
  git config --global user.name "Pulkit2206"
name: Add collaborator 2
```

Figure 14: workflow for initial setup and installing of packages

Subsequently, the workflow executes R scripts responsible for updating the database with new entries, validating the new entries' formats and referential integrity for each table, and updating the analysis considering new data arrivals. Moreover, it adds collaborators to the repository and commits and pushes changes to the main branch, ensuring the repository's integrity and up-to-date status.

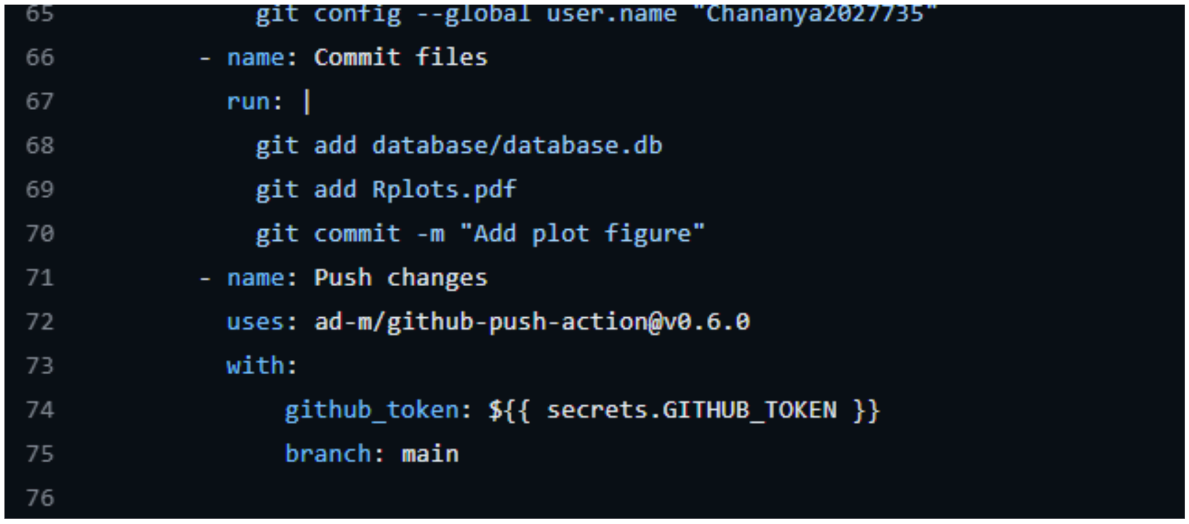

```
          git config --global user.name "Chananya2027735"
      - name: Commit files
        run: |
          git add database/database.db
          git add Rplots.pdf
          git commit -m "Add plot figure"
      - name: Push changes
        uses: ad-m/github-push-action@v0.6.0
        with:
            github_token: ${{ secrets.GITHUB_TOKEN }}
            branch: main

```

Figure 15: Pushing final changes and adding a pdf of plots

We used below command in PositCloud console to connect PositCloud with our Github repository:

Install.packages ("devtools")

Install.packages ("Rcpp", dependencies = TRUE)

usethis::use_git_config (user.name="Username",user.email="User Email Id")

Overall, the setup ensures smooth execution of data-related tasks, automating processes and maintaining consistency within the collaborative environment. By utilising GitHub Actions, the repository streamlines continuous integration practices, facilitating efficient data pipeline generation and management.

4 Data Analysis of User Portrail

Based on the order data from 2023, we will focus on analyzing data related to user behavior, portraying user profiles in terms of their product preference, loyalty and regional distribution, in order to gain a more comprehensive understanding of their purchasing behavior.

4.1 Data Analysis Process

4.1.1 Data Preparation

First, we created three new tables using the inner join function, extracted all the variables we needed from the normalized tables, and completed the variable type transformation task.

```r
library(rmarkdown)
library(dplyr)
library(tidyr)
library(ggplot2)
library(lubridate)
library(quarto)

# read the file
users_table <- read.csv("Synthetic_Data_Generation/users_table.csv")
products_table <- read.csv("Synthetic_Data_Generation/products_table.csv")
product_categories_table <- read.csv("Synthetic_Data_Generation/product_categories_table.csv
order_details_table <- read.csv("Synthetic_Data_Generation/order_details_table.csv")

# create new table for category analysis with inner join function
data_ctg <- inner_join(order_details_table, product_categories_table, by = "category_id") %>
  mutate(order_id = paste(order_detail_id, product_id, user_id, sep = "")) %>%
  select(order_id, order_datetime, category_name, order_qty, product_id)
# Extract year, month, and day components into separate columns and change to factors
data_ctg <- data_ctg %>%
  mutate(order_datetime = ymd_hms(order_datetime),
         order_year = as.factor(year(order_datetime)),
         order_month = as.factor(month(order_datetime)),
         order_day = as.factor(day(order_datetime)),
         order_monthf = factor(month.abb[order_month], levels = month.abb),
         order_id = as.factor(order_id),
         category_name = as.factor(category_name))
# create new table for orders analysis with inner join function
data_orders <- inner_join(order_details_table, users_table, by = "user_id") %>%
  mutate(order_id = paste(order_detail_id, product_id, user_id, sep = "")) %>%
  select(order_id, user_id, order_datetime, order_price, order_qty, product_id, user_members
# Extract year, month, and day components into separate columns and change to factors
data_orders <- data_orders %>%
  mutate(order_datetime = ymd_hms(order_datetime),
         order_year = as.factor(year(order_datetime)),
         order_monthf = factor(month.abb[month(order_datetime)], levels = month.abb),
         order_day = as.factor(day(order_datetime)),
         order_id = as.factor(order_id))
# calculate the user number by month
total_unique_users <- data_orders %>%
  distinct(user_id, .keep_all = TRUE) %>%
  group_by(order_monthf) %>%
  summarise(total_unique_users = n_distinct(user_id))
# calculate the user with 'prime' by month
prime_unique_users <- data_orders %>%
  filter(user_membership_status == 'Prime') %>%
  distinct(user_id, .keep_all = TRUE) %>%
  group_by(order_monthf) %>%
  summarise(prime_unique_users = n_distinct(user_id))
# calculate the user status by month
unique_users_by_status <- data_orders %>%
  distinct(user_id, .keep_all = TRUE) %>%
  group_by(order_monthf, user_membership_status) %>%
  summarise(user_count = n_distinct(user_id))
# calculate the order number by month by membership
data_member <- data_orders %>%
  group_by(user_membership_status,order_id,order_year,order_monthf) %>%
  summarise(total_order = n(), total_price = sum(order_price))
# calculate the order number by month
total_orders_by_month <- data_member %>%
  group_by(order_monthf) %>%
  summarise(total_orders = n())
# create new table for locations analysis with inner join function
data_locations <- inner_join(order_details_table, users_table, by = "user_id") %>%
  mutate(order_id = paste(order_detail_id, product_id, user_id, sep = "")) %>%
  select(order_id, address_state, user_id, order_datetime, order_price, order_qty, product_i
# Extract year, month, and day components into separate columns and change to factors
data_locations <- data_locations %>%
  mutate(order_datetime = ymd_hms(order_datetime),
         order_year = as.factor(year(order_datetime)),
         order_monthf = factor(month.abb[month(order_datetime)], levels = month.abb),
         order_day = as.factor(day(order_datetime)),
         order_id = as.factor(order_id))
```

4.1.2 Plot Coding

Next, we applied the ggplot function to graph the analysis.

```r
# 1. Total sales of products by category in 2023.
data_ctg_year <- data_ctg %>%
  group_by(category_name) %>%
  summarise(total_order_qty = sum(order_qty)) %>%
  ungroup() %>%
  mutate(category_name = factor(category_name, levels = rev(unique(category_name))))
p.total.sales <- ggplot(data_ctg_year) +
  geom_col(aes(y = reorder(category_name, total_order_qty), x = total_order_qty), fill = "sky
  geom_text(aes(y = reorder(category_name, total_order_qty), x = total_order_qty, label = to
  ylab("Category Name") +
  xlab("Total Sales Quantity") +
  theme_minimal()  +
  ggtitle("Sales by Category")

# 2. Distribution of sales of products by category by months in 2023.
data_ctg_month <- data_ctg %>%
  group_by(order_monthf,category_name) %>%
  summarise(total_order_qty = sum(order_qty))  %>%
  arrange(desc(total_order_qty))
color_palette <- c("brown","orange", 'lightgreen', "lightblue", "lightgrey", "darkgrey")
p.total.sales.month <- ggplot(data_ctg_month) +
  geom_bar(aes(x = total_order_qty, y = order_monthf, fill = category_name), stat = "identit
  scale_fill_manual(values = color_palette) +
  ylab("Month") +
  xlab("Total Sales Quantity") +
  coord_flip() +
  theme_minimal() +
  ggtitle("Monthly Sales by Category")

# 3.Total customers by month in 2023.
color_palette <- c("lightblue","orange")
p.total.customers <- ggplot() +
  geom_bar(data = unique_users_by_status, aes(x = order_monthf, y = user_count, fill = user_
  scale_fill_manual(values = color_palette) +
  geom_text(data = total_unique_users, aes(x = order_monthf, y = total_unique_users, label =
  geom_line(data = total_unique_users, aes(x = order_monthf, y = total_unique_users, group =
  geom_line(data = prime_unique_users, aes(x = order_monthf, y = prime_unique_users, group =
  xlab("Order Month") +
  ylab("Unique User Count") +
  theme_minimal() +
  scale_fill_manual(values = color_palette) +
  scale_color_manual(values = c("Total Users" = 'grey', "Prime Users" = "red")) +
  ggtitle("User Distribution by Membership Status by Month") +
  guides(fill = guide_legend(title = "Membership Status"), linetype = guide_legend(title = "(

# 4. The order frequency and price of customers by month.
p.order.qty <- ggplot(data_member) +
  geom_bar(aes(x = order_monthf, fill = user_membership_status)) +
  labs(x = "Month", y = "Number of Orders", fill = "Membership Status") +
  scale_fill_manual(values = color_palette) +
  theme_minimal() +
  ggtitle("Orders Count by Membership Status by Month") +
  geom_line(data = total_orders_by_month, aes(x = order_monthf, y = total_orders, group = 1,
  geom_text(data = total_orders_by_month, aes(x = order_monthf, y = total_orders, label = to
  scale_linetype_manual(values = c("Total Orders" = "solid")) +
  guides(fill = guide_legend(title = "Membership Status"), linetype = guide_legend(title = "(
data_member_avg <- data_member %>%
  group_by(user_membership_status,order_year,order_monthf) %>%
  summarise(avg_price = mean(total_price))
p.order.price <- ggplot(data_member_avg) +
  geom_point(aes(x = order_monthf, y = avg_price, col = user_membership_status)) +
  geom_line(aes(x = order_monthf, y = avg_price, group = user_membership_status, col = user_r
  labs(x = "Month", y = "Average Total Price", col = "Membership Status") +
  theme_minimal() +
  ggtitle("Orders Price by Membership Status by Month")

# 5. Regional Distribution of Orders
data_state <- data_locations %>%
  group_by(user_id,address_state) %>%
  summarise(total_order_qty = sum(order_qty), total_price = sum(order_price))
data_state$user_id <- as.factor(data_state$user_id)
data_state_count <- data_state %>% group_by(address_state) %>% count(address_state) %>% arran
data_state_count$address_state <- as.factor(data_state_count$address_state)
p.region <- ggplot(data_state_count) +
  geom_bar(aes(x = reorder(address_state, n), y = n), stat = "identity") +
  geom_text(aes(x = reorder(address_state, n), y = n, label = n), vjust = 0.5, hjust = -0.1,
  labs(x = "State", y = "The number of customers") +
  coord_flip() +
  theme_minimal() +
  ggtitle("TOP 10 Deliveried States")
```

4.2 Analytics Results

4.2.1 Product Preference

First, we analyzed user order data to understand the sales quantity of each category as well as the monthly trend.

After summarizing the number of sales of each products in the order we get the annual sales figures for each category, we sold a total of 7,684 products in 2023. Books, Clothing and Electronics were the top selling categories with sales of 3295, 1544 and 1330 respectively, accounting for 43%, 20% and 17% of annual product sales, indicating that students have the greatest demand for learning materials, school-created apparel peripherals are popular, and teaching-supported electronics have steady sales. However, Toys & Game's sales of about 250 suggests that school-branded peripheral products are not attractive enough to students, which may be due to insufficient product design, high pricing, or diversion from offline stores, and the specific reasons require further research.

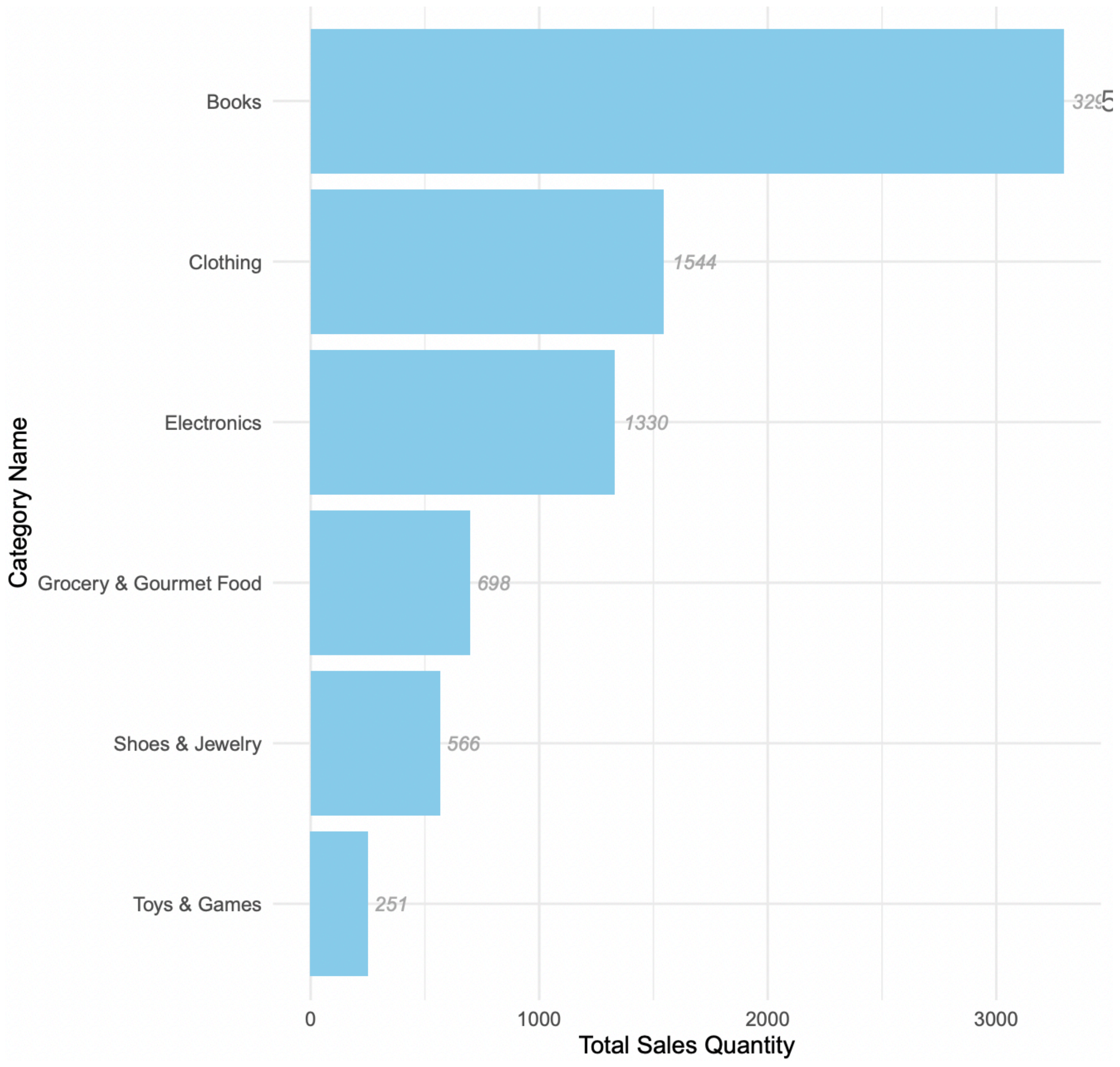


Figure 16: Sales by Category

On the monthly data, sales were significantly lower during the Christmas holiday period (December and January). By category, students have the highest demand for clothing in the spring and fall, and demand for food decreases during summer vacation.

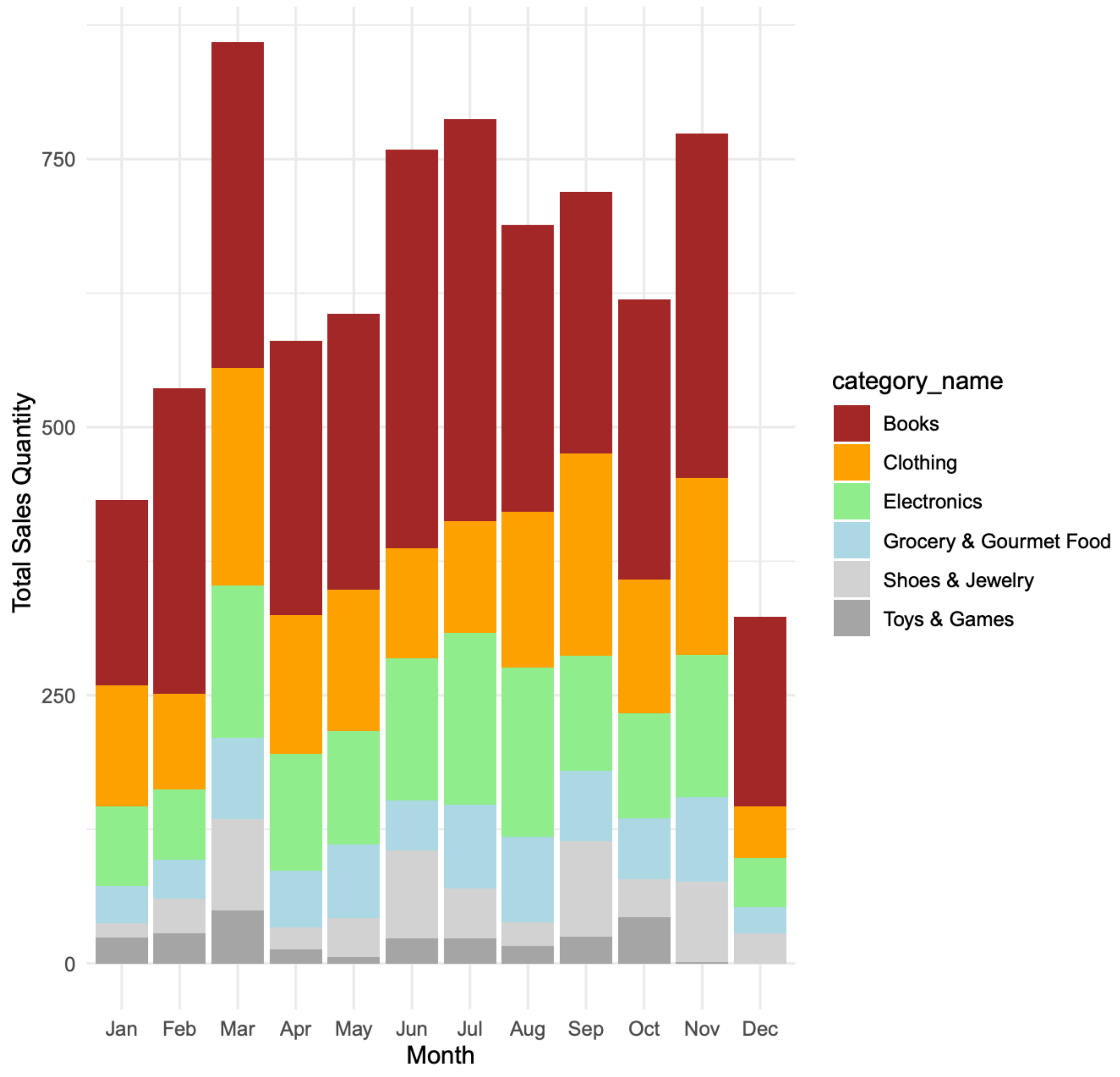


Figure 17: Monthly Sales by Category

4.2.2 Member loyalty

The University of USA Shopping Platform has implemented a fee-based membership system, with members enjoying benefits such as priority delivery as well as cashback on points. We analyzed the difference in purchasing capacity between member and non-member groups to see if membership is effective in increasing user loyalty and spending.

From the monthly distribution chart of the number of members of users who have made purchases, we can see that most of the users have opened a membership, which shows that the membership is attractive enough to our users.

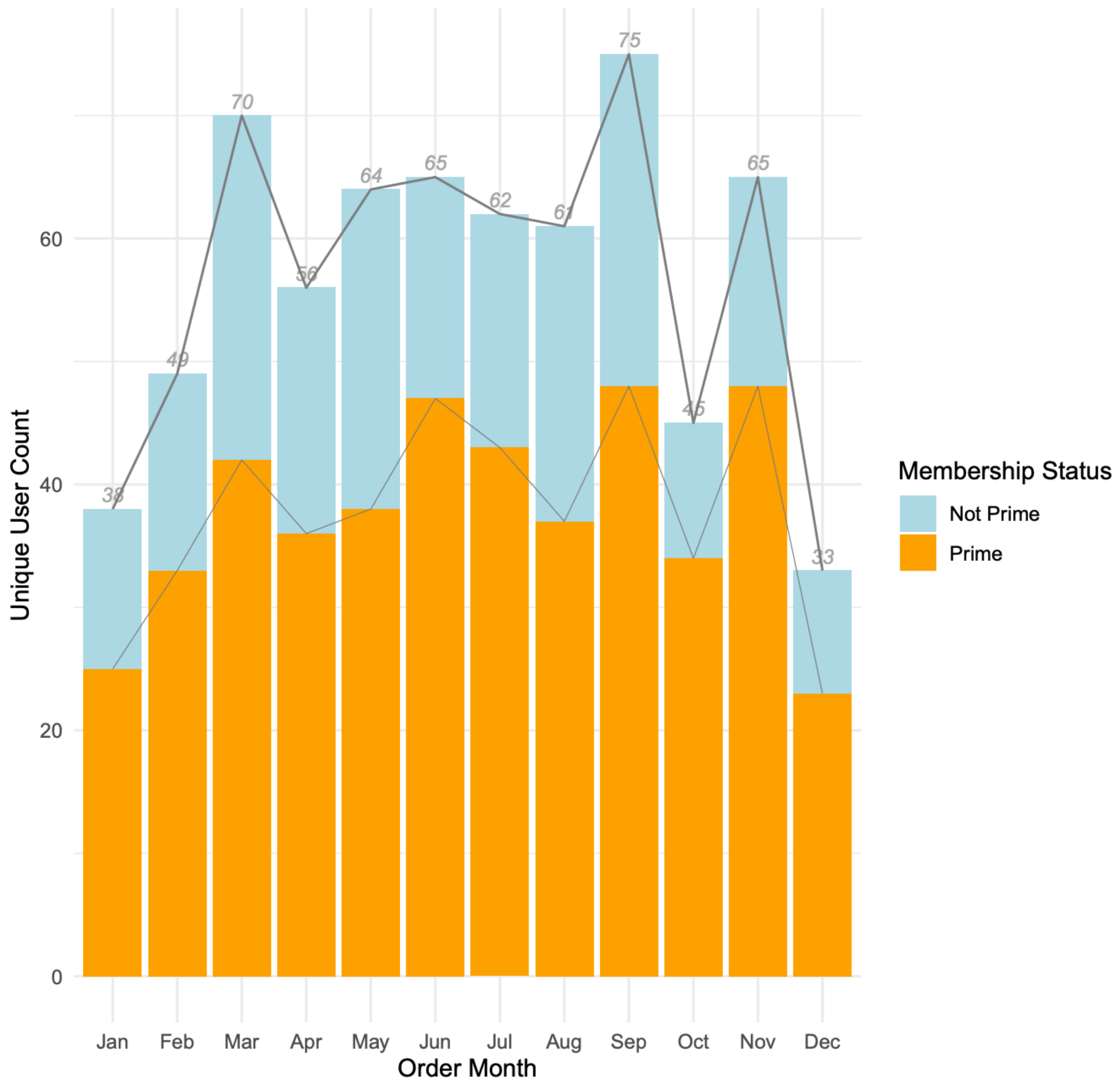


Figure 18: User Distribution by Membership Status by Month

Therefore, we further analyzed the differences in the number of purchases and order prices of member users. It is obvious that member users contribute most of the order quantity, and orders placed by members is much higher than that of non-members, which also indicates that member users will use the school shopping platfrom more frequently. However, the difference between members and non-members in the orders prices was more unusual in Jauary, with non-members purchasing substantially more during this month, due to the large alumni socials held at the university, which attracted a large number of alumni to return to campus and purchase souvenirs.

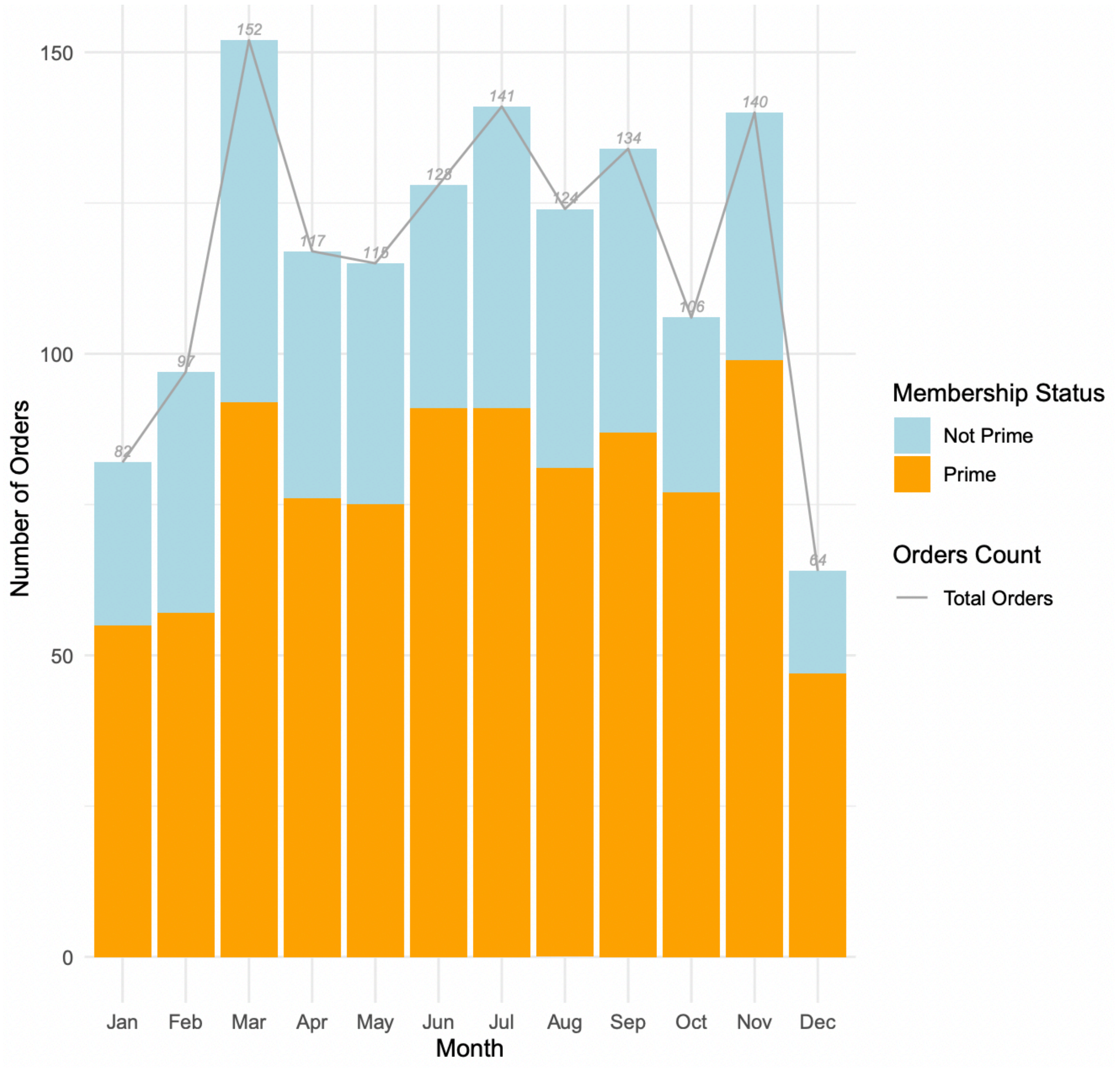


Figure 19: Orders Count by Membership Status by Month

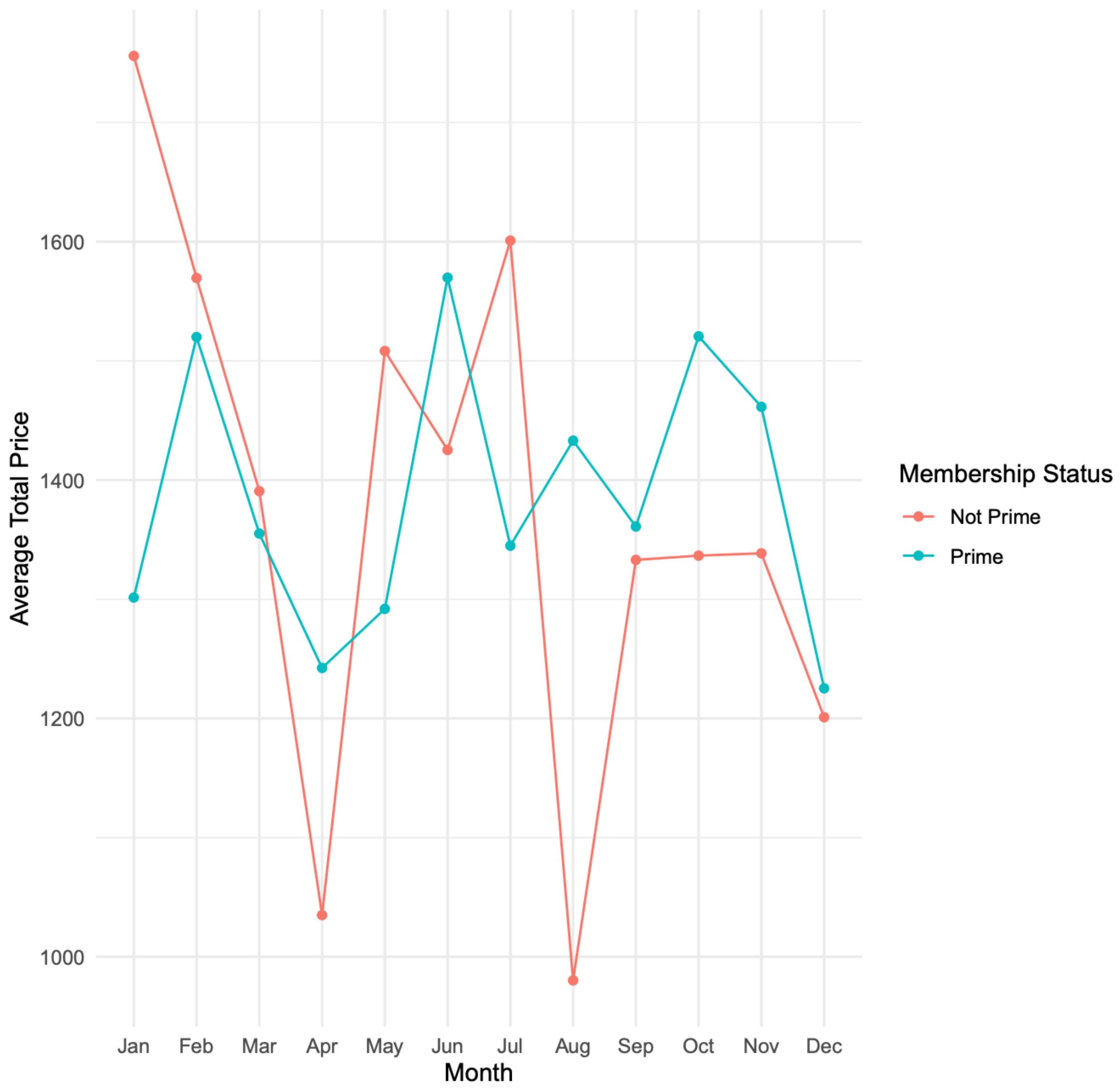


Figure 20: Orders Price by Membership Status by Month

4.2.3 Regional Distribution

Express delivery efficiency is an important element of the user's shopping experience, so we plan to enhance express delivery services in popular delivery areas. Through the chart, we can see that the top ten states in the order delivery quantity ranking Texas has the highest order quantity, which indicates that the school neighborhood is still the most important delivery area, we can increase the offline store pickup service and so that students can get the products faster. Meanwhile, California and Florida, as large neighboring states, also have higher delivery needs, which means we should prioritize improving delivery efficiency in these states.

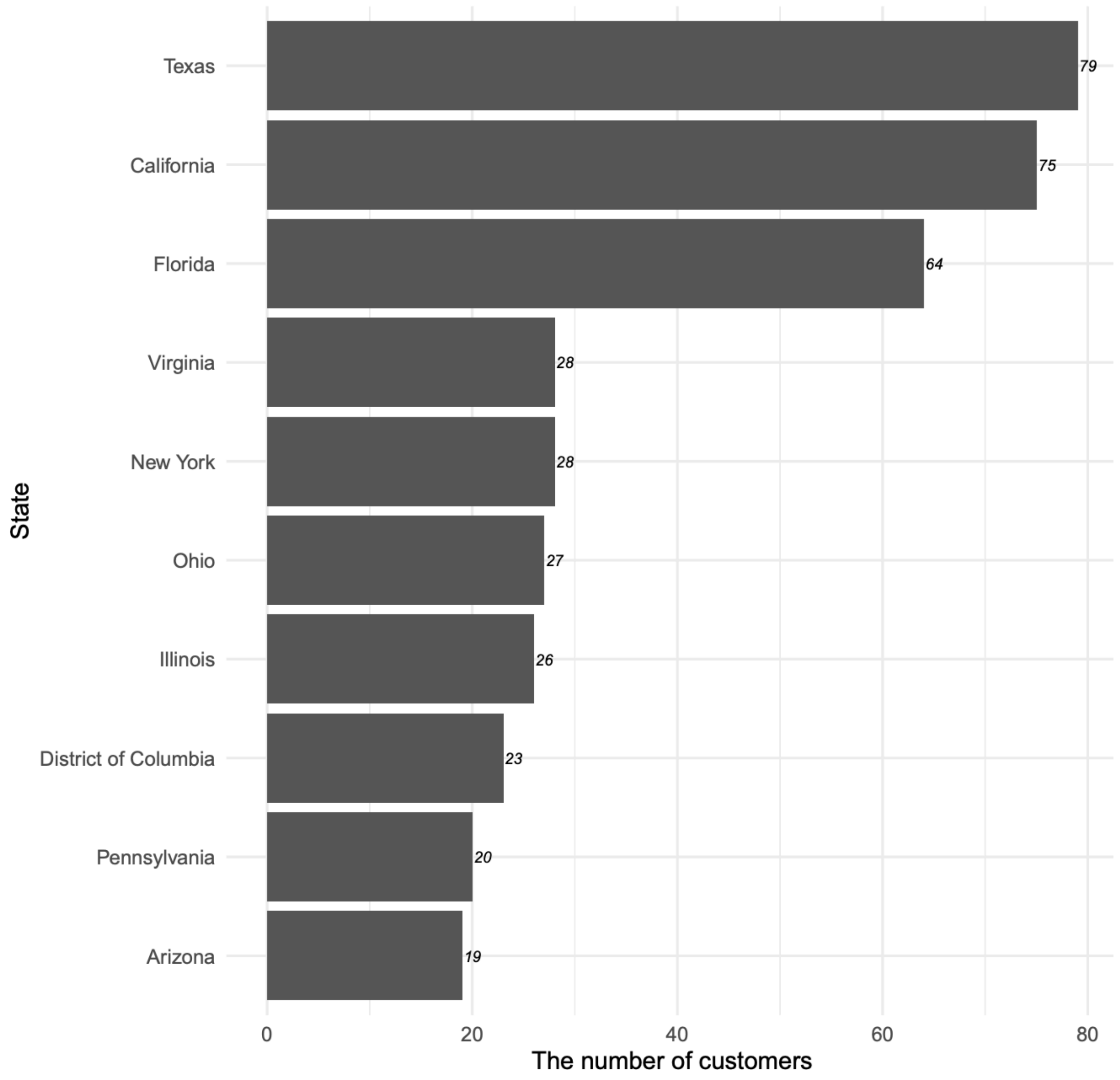


Figure 21: TOP 10 Deliveried States

Conclusion