

# Data Management Group 29

2024-03-19

## Table of contents

<b>Objective</b>	<b>2</b>
<b>Assumptions</b>	<b>2</b>
<b>Part 1: Database Design and Implementation</b>	<b>3</b>
1.1: Conceptual Design : E-R Diagram . . . . .	3
1.2: Relationship Set . . . . .	4
1.3: 3NF Tables . . . . .	7
1.4: Logical Design . . . . .	9
1.5: SQL Database Schema Creation . . . . .	9
<b>Part 2: Data Generation and Management</b>	<b>12</b>
2.1: Synthetic Data Generation . . . . .	12
2.2: Data Import and Quality Assurance . . . . .	17
2.2.1: Import data . . . . .	17
2.2.2: Data Validation . . . . .	18
<b>Part 3 : Data Pipeline Generation</b>	<b>40</b>
<b>Part 4: Data Analysis and Visualisation</b>	<b>42</b>
4.1: The Key Metrics Performance Analysis (Revenue, Order Completion Day) . . .	43
4.2: Customer Analysis . . . . .	44
4.3: Payment Method . . . . .	46
4.4: Category Analysis . . . . .	48
4.5: Product Analysis . . . . .	50
4.6: Customer Review Analysis . . . . .	52
4.7: Supplier Analysis . . . . .	55
<b>Appendix</b>	<b>58</b>

## Objective

The project aims to optimise the revenue of fashion e-commerce based on the sales performance and customer behaviours. This entails creating detailed customer profiles, such as preferences, purchasing patterns, and interactions with the company's products or services. Through a comprehensive analysis of these customer portraits, the project seeks to uncover valuable insights that can inform strategic decisions and initiatives within the company.

The database design of the project is intricately aligned with its revenue optimisation objective. It comprises tables capturing vital data points such as customer profiles, sales transactions, product information, and interactions. This structured approach enables efficient storage, retrieval, and analysis of data, facilitating informed decision-making processes. Furthermore, the database design adheres to principles of normalisation, ensuring data integrity and minimizing redundancy, thereby bolstering the reliability and scalability of the system.

## Assumptions

1. The business operates solely within the United Kingdom, catering to domestic customers and suppliers.
2. Product prices are determined to ensure a 30% profit margin, with individual product prices ranging from £20 to £150.
3. Product stock refers to the inventory recorded at the end of each month, starting with January.
4. All products are one-size-fits-all.
5. Free shipping is provided for all orders, and there are no associated freight costs, regardless of the number of products ordered.
6. Order quantities range from a minimum of 1 to a maximum of 5 units per product.
7. Undelivered orders are not rated; ratings are only provided by customers upon receiving their products.
8. Payment statuses primarily consist of completed payments (95%), with the remaining 5% being declined payments.
9. There are seven types of discounts available, ranging from 0% to 30%, with each product being assigned one discount type.
10. Each order will be shipped in a single package, regardless of the number of products ordered.

# Part 1: Database Design and Implementation

## 1.1: Conceptual Design : E-R Diagram

Initially, our database comprises over 10 entities, which can make it challenging to understand the relationships between entities. This complexity often leads to difficulties in designing and maintaining the database. Moreover, with such complexity, there is a potential risk of data redundancy and inconsistency, posing challenges to maintaining data integrity. To mitigate these issues, we have chosen to model the order process as a relationship which captures the relationship between customer, product and shipment without introducing an intermediary entity which simplifies the database schema by reducing the number of entities. As a result, our diagram now comprises six major entities, simplifying the structure and improving the clarity of relationships between elements in our database.

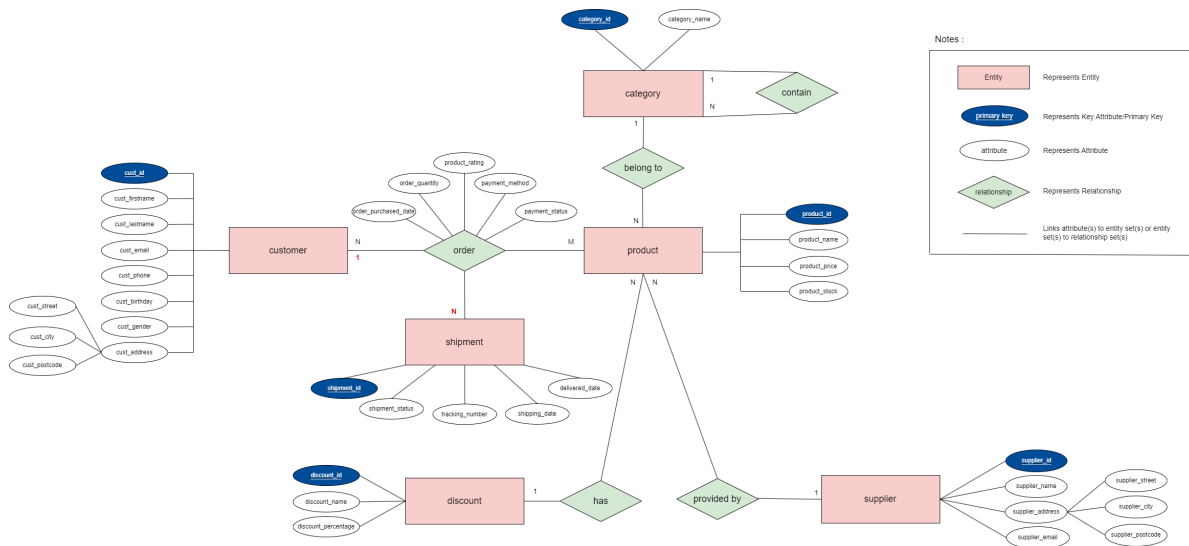


Figure 1: ER Diagram

[ER Diagram Link](#)

### Entities and Attributes

#### 1. Customer

The “Customer” entity stores mandatory information of individuals who make purchases through the online store. Each customer is assigned with `customer_id` which serves as the primary key. Other attributes include `cust_firstname`, `cust_lastname`, `cust_email`, `cust_phone`, `cust_birthday`, `cust_gender`, `cust_address` which is a composite attribute consisting of `cust_street`, `cust_city`, and `cost_postcode`.

## 2. Product

The “Product” entity represents individual clothing items available for sale all the product details are mandatory field. (prices, stock, name, etc.) Each product has a `product_id` as the primary key. Other attributes are `product_name`, `product_price`, and `product_stock`.

## 3. Category

The “Category” entity acts as a hierarchical structure (application of parent category) that organizes products into different groups. The attributes are `category_id` (primary key) and `category_name`.

**4. Supplier** The “Supplier” entity represents external businesses that provide the online clothing store with the products it sells. With `supplier_id` as a unique identifier for each supplier (primary key). Other attributes include `supplier_name`, `supplier_email`, and `supplier_address` which consists of `supplier_street`, `supplier_city`, and `supplier_postcode`.

**5. Shipment** The “Shipment” entity stores the information of shipping records for each order, from the store to the customer. Each shipment is uniquely identified by a `shipment_id` (primary key). A unique tracking number will be generated only if the `shipment_status` is success (Delivered, in transit and packaging). Other attributes include `shipping_date`, representing the date when the shipment was initiated, and `delivered_date`, indicating the date when the shipment was delivered.

**6. Discount** The Discount entity represents the various promotional offers and discounts available for each product. Discounts are seasonal and each discount is uniquely identified by a `discount_id` (primary key). Other attributes include `discount_name` and `discount_percentage`.

## 1.2: Relationship Set

### 1. Product to Category (N:1)

A category consists of many products but a product can only belong to 1 category.

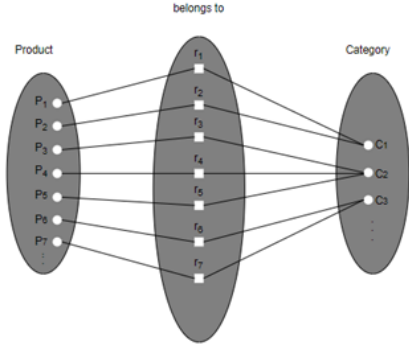


Figure 2: Product-Category Relationship

## 2. Product to Discount (N:1)

A product has exactly one discount but a discount is applicable to many products.

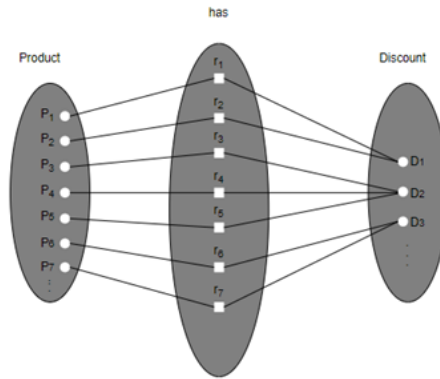


Figure 3: Product-Discount Relationship

## 3. Product to Supplier (N:1)

A product provides by exactly one supplier but a supplier can provide many products.

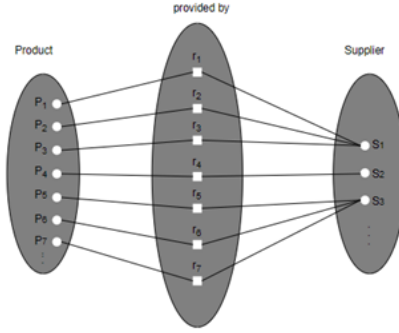


Figure 4: Product-Supplier Relationship

#### 4. Customer to Product (M:N)

A customer can purchase multiple products, and each product can be purchased by multiple customers.

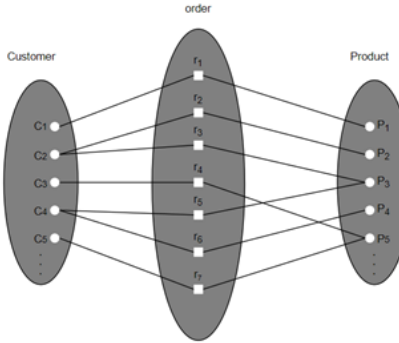


Figure 5: Customer-Product Relationship

In many-to-many relationship, it's possible for the relationship itself to have attributes. In this case, order has several attributes, including `order_purchased_date`, `order_quantity`, `product_rating`, `payment_method`, and `payment_status` (complete/decline). Moreover, this many-to-many relationship resulted in the creation of an intermediate or junction table (order table)

#### 5. Customer to Shipment (1:N)

Each customer can be associated with multiple shipments, while each shipment is linked to only one customer.

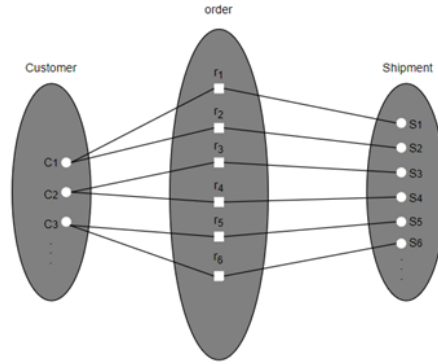


Figure 6: Customer-Shipment Relationship

## 6. Category (Self Referencing)

In a self-referencing category relationship, each category may include a reference to its parent category.

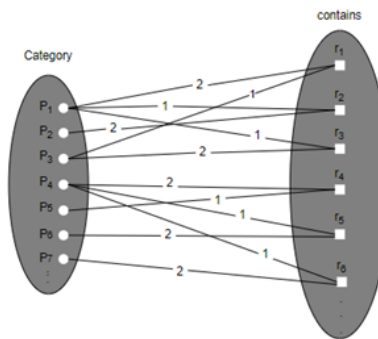


Figure 7: Self-referencing Category

## 1.3: 3NF Tables

The process of achieving the Third Normal Form (3NF) involves eliminating transitive dependencies in addition to adhering to the rules of the 1NF and 2NF.

We ensure that each attribute (column) in the table contains atomic values, meaning that it holds only a single piece of data. If an attribute contains multiple values or is a composite of multiple sub-attributes such as address, it should be decomposed into separate attributes. We also ensure that each row in the table represents a unique entity or occurrence, and there are no duplicate rows.

In 2NF, we ensure that each attribute is fully functionally dependent on the primary key. In the order table, if we have product\_name in addition to product\_id, it can cause redundancy because product\_name is functionally dependent on product\_id. To normalize to 2NF, we remove product\_name from the order table and create a separate product table.

In 3NF, we eliminate transitive dependencies. If supplier\_city is dependent on supplier\_id, and supplier\_id is dependent on product\_id in the product table, then supplier\_city is transitively dependent on product\_id. To normalize to 3NF, we create a separate supplier table.

The figure below visualise tables as a base that we use for data generation for fashion e-commerce.

customer table									
cust_id	cust_first_name	cust_last_name	cust_email	cust_phone	cust_birthday	cust_gender	cust_street	cust_city	cust_postcode
CU998797	Heall	Tointon	htointon0@etsy.com	+44 9078 588208	1991-12-06	Male	5267 Del Mar Trail	Swansea	UW1 3QV
CU019006	Sheffy	Rosbrough	srosbrough1@nature.com	+44 0294 542839	1965-07-02	Male	6 Welch Terrace	Peterborough	W18 9CE
CU344900	Giselbert	Wiburn	gwiburn2@discovery.com	+44 8573 560729	1974-08-17	Male	8 Corscot Point	Edinburgh	DM4 4ES
CU399496	Chancey	Enns	cenns3@illinois.edu	+44 8516 693169	1974-03-21	Male	68972 Lawn Pass	Norwich	BZ4 2RO

product table						
product_id	category_id	supplier_id	discount_id	product_name	product_price	product_stock
P369132	C10040	S73798	D65025	Men's Shadow 5000 Trainer	102	45
P373155	C10040	S36956	D65010	Men's Classic Leather Loafer	131	89
P719403	C12985	S70866	D65020	Women's Cropped Twill Zip	67	50
P846623	C12985	S85417	D65015	Women's Belted Double-fa	150	10

supplier table					
supplier_id	supplier_name	supplier_street	supplier_city	supplier_postcode	supplier_email
S36588	H&M	63 Vahlen Hill	Leeds	BC7 2UF	H&M@gmail.com
S27903	COS	1925 Kenwood Junction	Manchester	NM4 7XK	COS@gmail.com
S28491	Arket	0289 Alpine Drive	Edinburgh	ZH5 9LD	Arket@gmail.com
S86620	Zara	7755 Canary Circle	Leeds	AI3 7PN	Zara@gmail.com

category table		
category_id	category_name	p_category_id
C12985	Jacket & Coats	PC61662
C39924	Dresses	PC61662
C78292	Trousers	PC61662
C72554	Jeans	PC61662

order table								
order_id	customer_id	product_id	shipping_id	order_purchased_date	order_quantity	product_rating	payment_method	payment_status
G857239	CU748577	P131431	SH14008	2024-01-19	3	3	online banking	complete
G707816	CU315854	P263514	SH13428	2024-01-02	5		paypal	declined
G752808	CU438019	P802161	SH91315	2024-01-24	1	1	credit/debit card	complete
G230551	CU967520	P574811	SH85973	2024-01-04	4	3	gift card	complete

discount table		
discount_id	discount_name	discount_percentage
D65000	no_discount	0%
D65005	5% off	5%
D65010	10% off	10%
D65015	15% off	15%

shipment table				
shipping_id	tracking_number	shipping_date	delivered_date	shipment_status
SH14008	TN969180001	2024-01-23	2024-01-31	Delivered
SH13428				No Shipment
SH91315	TN226702721	2024-01-27	2024-02-04	Delivered
SH85973	TN189105038	2024-01-11	2024-01-20	Delivered

Note : Green shows Primary Key, Orange shows Foreign Key, and Grey shows attributes



## 1.4: Logical Design

After transferring the relational schema to logical we have the following tables (underline denote primary key, double underline denote foreign key) :

- **customer**(cust\_id, cust\_firstname, cust\_lastname, cust\_email, cust\_phone, cust\_birthday, cust\_gender, cust\_street, cust\_city, cust\_postcode)
- **product**(product\_id, category\_id, supplier\_id, discount\_id, product\_name, product\_price, product\_stock)
- **category**(category\_id, p\_category\_id, category\_name)
- **discount**(disc\_id, discount\_name, discount\_percentage)
- **supplier**(supplier\_id, supplier\_name, supplier\_street, supplier\_city, supplier\_postcode, supplier\_email)
- **shipment**(shipping\_id, tracking\_number, shipping\_date, delivered\_date, shipment\_status)
- **order**(order\_id, customer\_id, product\_id, shipping\_id, order\_purchased\_date, order\_quantity, product\_rating, payment\_method, payment\_status)

The uniqueness of a record in the order table is determined by the combination of **order\_id**, **customer\_id** and **product\_id**.

## 1.5: SQL Database Schema Creation

The following SQL code presents the schema design for a comprehensive database tailored to support a fashion e-commerce platform.

We've enforced NOT NULL constraints to maintain data integrity, preventing essential attributes from containing null values. UNIQUE constraints ensure attributes like email, phone, and tracking numbers are unique across all records in the table. Indexes have been incorporated to expedite data retrieval and enhance query performance, streamlining operations in our e-commerce database. By utilising VARCHAR for textual data, INT for integers, DECIMAL for precise numeric values, and DATE for dates, we ensure accurate representation of stored information.

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

```
DROP TABLE IF EXISTS 'customer';
```

```
CREATE TABLE 'customer'(  
  'cust_id' VARCHAR(10) PRIMARY KEY,  
  'cust_firstname' VARCHAR(255) NOT NULL,  
  'cust_lastname' VARCHAR(255),
```

```
'cust_email' VARCHAR(255) UNIQUE NOT NULL,  
'cust_phone' VARCHAR(20) UNIQUE NOT NULL,  
'cust_birthday' DATE NOT NULL,  
'cust_gender' VARCHAR(10),  
'cust_street' VARCHAR(255) NOT NULL,  
'cust_city' VARCHAR(255) NOT NULL,  
'cust_postcode' VARCHAR(20) NOT NULL  
);
```

```
CREATE INDEX idx_customer_location ON customer(cust_city, cust_postcode);
```

```
DROP TABLE IF EXISTS 'product';
```

```
CREATE TABLE 'product'(  
  'product_id' VARCHAR(10) PRIMARY KEY,  
  'category_id' VARCHAR(10) NOT NULL,  
  'supplier_id' VARCHAR(10) NOT NULL,  
  'discount_id' VARCHAR(10) NOT NULL,  
  'product_name' VARCHAR(255) NOT NULL,  
  'product_price' DECIMAL(10,2) NOT NULL,  
  'product_stock' INT NOT NULL,  
  FOREIGN KEY ('category_id') REFERENCES category('category_id'),  
  FOREIGN KEY ('discount_id') REFERENCES discount('discount_id')  
);
```

```
CREATE INDEX idx_product_name ON product(product_name);
```

```
CREATE INDEX idx_product_stock ON product(product_stock);
```

```
DROP TABLE IF EXISTS 'category';
```

```
CREATE TABLE 'category' (  
  'category_id' VARCHAR(10) PRIMARY KEY,  
  'category_name' VARCHAR(255) NOT NULL,  
  'p_category_id' VARCHAR(10) NOT NULL  
);
```

```
CREATE INDEX idx_category_p_category_id ON category(p_category_id);
```

```
DROP TABLE IF EXISTS 'supplier';
```

```
CREATE TABLE 'supplier'(  
    'supplier_id' VARCHAR(10) PRIMARY KEY,  
    'supplier_name' VARCHAR(255) NOT NULL,  
    'supplier_street' VARCHAR(255) NOT NULL,  
    'supplier_city' VARCHAR(255) NOT NULL,  
    'supplier_postcode' VARCHAR(20) NOT NULL,  
    'supplier_email' VARCHAR(255) NOT NULL  
);
```

```
CREATE INDEX idx_supplier_email ON supplier(supplier_email);
```

```
DROP TABLE IF EXISTS 'order';
```

```
CREATE TABLE 'order'(  
    'order_id' VARCHAR(10) NOT NULL,  
    'customer_id' VARCHAR(10) NOT NULL,  
    'product_id' VARCHAR(10) NOT NULL,  
    'shipping_id' VARCHAR(10) NOT NULL,  
    'order_purchased_date' DATE NOT NULL,  
    'order_quantity' INT NOT NULL,  
    'product_rating' INT,  
    'payment_method' VARCHAR(20) NOT NULL,  
    'payment_status' VARCHAR(10) NOT NULL,  
    FOREIGN KEY ('customer_id') REFERENCES customer('cust_id'),  
    FOREIGN KEY ('product_id') REFERENCES product('product_id'),  
    FOREIGN KEY ('shipping_id') REFERENCES shipment('shipping_id')  
);
```

```
CREATE INDEX idx_order_customer ON 'order'(customer_id);
```

```
CREATE INDEX idx_order_product ON 'order'(product_id);
```

```
DROP TABLE IF EXISTS 'shipment';
```

```
CREATE TABLE 'shipment'(  
    'shipping_id' VARCHAR(10) PRIMARY KEY,  
    'tracking_number' INT UNIQUE,  
    'shipping_date' DATE,
```

```
'delivered_date' DATE,  
'shipment_status' VARCHAR(20)  
);
```

```
CREATE INDEX idx_shipment_dates ON shipment(shipping_date, delivered_date);
```

```
DROP TABLE IF EXISTS 'discount';
```

```
CREATE TABLE 'discount' (  
  'discount_id' VARCHAR(10) PRIMARY KEY,  
  'discount_name' VARCHAR(200) NOT NULL,  
  'discount_percentage' INT NOT NULL  
);
```

## Part 2: Data Generation and Management

### 2.1: Synthetic Data Generation

Following the decision on the database objectives and the completion of the ER diagram, which now includes seven tables due to a many-to-many relationship, data population became the focus. We used Mockaroo for tables without foreign keys (category, discount, supplier, and customer), customizing city options to match UK locales and ensuring category data alignment with the ER diagram. Supplier email addresses were scripted in R Studio to resemble actual structures, and limited entries in the discount table were manually crafted. Synthetic data generation was then applied to tables with foreign keys (products, orders, and shipments). Mockaroo initially generated product data, with adjustments made to product names in Excel for uniqueness. Orders and shipments, more complex, were generated using R, simulating monthly sales data and realistic payment scenarios. Constraints were imposed on order quantities, and shipment data ensured logical sequencing and defined statuses reflecting order fulfillment stages. Finally, constraints in the order table incorporated product ratings based on shipment statuses, emphasizing rating upon product delivery by customers. This comprehensive methodology ensures that our database accurately reflects real-world business scenarios while also upholding data integrity and consistency throughout all operations.

The code utilised to generate synthetic data will be presented below.

*Note : Category\_table, Product\_table, Customer\_table and Supplier\_table are first generated by Mockaroo. Discount\_table is handcrafted. Supplier Table - Change the supplier email for the supplier data generated by Mockaroo*

```

# Read the CSV file
data <- read.csv("supplier.csv")
data$supplier_email <- NULL

# Extract company names from the Excel file (assuming the company name column is named "Company Name")
supplier_name <- data$supplier_name

# Generate email addresses
supplier_email <- paste0(supplier_name, "@gmail.com")

# Create a data frame with company names and email addresses
full_data <- cbind(data, supplier_email)

# Define the file path to your desktop on macOS
desktop_path <- file.path(Sys.getenv("HOME"), "Desktop")

# Print the desktop path
print(desktop_path)

# Write the data to an Excel file on the desktop
write.xlsx(full_data, file.path("data_upload", "Supplier_table.xlsx"), rowNames = FALSE)

```

Customer Table - Change the customer email for the customer data generated by Mockaroo

```

# Read the CSV file
data <- read.csv("customer.csv")
data$cust_email <- NULL

# Extract customer first names from the Excel file
cust_fname <- data$cust_first_name
cust_lname <- data$cust_last_name

# Generate email addresses
cust_email <- tolower(paste0(cust_fname, ".", cust_lname, "@gmail.com"))

# Create a data frame with company names and email addresses
full_data <- cbind(data, cust_email)

# Write the data to an Excel file on the desktop
write.xlsx(full_data, file.path("data_upload", "Customer_table.xlsx"), rowNames = FALSE)

```

Generate Order Table

```

# Define function to generate random order ID
generate_order_id <- function() {
  paste0("G", sample(100000:999999, 1))
}

# Load product and customer data
product_data <- read.csv("product.csv")
customer_data <- read.csv("customer600.csv")

# Generate sample data
set.seed(123)
n <- 1000

# Generate order_id
order_ids <- replicate(n, generate_order_id())

# Generate unique order_id with some repeats
unique_order_ids <- sample(order_ids, 1000, replace = TRUE)

# Repeat order_id to match total rows
order_ids <- rep(unique_order_ids, each = n / 1000)

# Generate order_purchased_date for each unique order_id
order_purchased_dates <- sample(seq(as.Date("2024-01-01"), as.Date("2024-01-31"), by = "day"),
                                n, replace = TRUE)

# Repeat order_purchased_date to match total rows
order_purchased_dates <- rep(order_purchased_dates, each = n / 1000)

# Sample customer_id from customer_data for each unique order_id
customer_ids <- sample(customer_data$cust_id, 1000, replace = TRUE)

# Repeat customer_id to match total rows
customer_ids <- rep(customer_ids, each = n / 1000)

# Sample product_id, order_quantity, and product_rating for each row
sampled_product_ids <- sample(product_data$product_id, n, replace = TRUE)
order_quantity <- sample(1:5, n, replace = TRUE)
product_rating <- sample(1:5, n, replace = TRUE)

# Generate payment_method and payment_status
payment_methods <- sample(c("paypal", "credit/debit card", "online banking", "gift card"), n, replace = TRUE)
payment_status <- sample(c("declined", "complete"), n, replace = TRUE, prob = c(0.05, 0.95))

```

```

# Create order data frame
order_data <- data.frame(
  order_id = order_ids,
  customer_id = customer_ids,
  product_id = sampled_product_ids,
  order_purchased_date = order_purchased_dates,
  order_quantity = order_quantity,
  product_rating = product_rating,
  payment_method = payment_methods,
  payment_status = payment_status
)

# Apply the rule
order_data <- order_data %>%
  group_by(order_id) %>%
  mutate(
    customer_id = first(customer_id),
    order_purchased_date = first(order_purchased_date),
    payment_method = first(payment_method),
    payment_status = first(payment_status)
  ) %>%
  ungroup()

# Generate unique shipping_ids for each unique order_id
unique_shipping_ids <- paste0("SH", sample(10000:99999, length(unique(order_data$order_id))),

# Match shipping_ids to order_ids
shipping_mapping <- data.frame(order_id = unique(order_data$order_id), shipping_id = unique_
order_data <- left_join(order_data, shipping_mapping, by = "order_id")

# Define the file path to your desktop on macOS
desktop_path <- file.path(Sys.getenv("HOME"), "Desktop")

# Write the data to an Excel file on the desktop
write.xlsx(order_data, file.path(desktop_path, "order_data.xlsx"), rowNames = FALSE)

# Modify the product rating after the shipping table has been done
# Load the shipping data
shipping_data <- read.xlsx("shipping_data.xlsx")

# Generate random values for product rating only when shipment status is 'Delivered'

```

```

order_data <- order_data %>%
  mutate(
    product_rating = ifelse(shipping_id %in% shipping_data$shipping_id[shipping_data$shipment_status == "delivered"],
                           sample(1:5, sum(shipping_id %in% shipping_data$shipping_id[shipping_data$shipment_status == "delivered"])),
                           NA)
  )

# Write the modified data to an Excel file on the desktop
write.xlsx(full_data, file.path("data_upload", "Order_table.xlsx"), rowNames = FALSE)

```

### Generate Shipping Table

```

# Function to generate random tracking number
generate_tracking_number <- function() {
  paste0("TN", sample(100000000:999999999, 1))
}

# Generate sample data
set.seed(123)

# Define minimum range between dates (in days)
min_range <- 2
max_range <- 10

# Extract unique orders while keeping order_purchased_date and payment_status displayed
unique_orders <- order_data %>%
  distinct(shipping_id, .keep_all = TRUE)

# Calculate the length of n based on the number of unique shipping_ids
n <- nrow(unique_orders)

# Sample from unique orders to create shipping_ids, purchased_dates, and payment_status
shipping_ids <- unique_orders$shipping_id
purchased_date <- unique_orders$order_purchased_date
payment_status <- unique_orders$payment_status

# Initialize vectors for shipping_date, delivered_date, tracking_number, and shipment_status
shipping_date <- purchased_date + sample(min_range:max_range, n, replace = TRUE)
delivered_date <- shipping_date + sample(min_range:max_range, n, replace = TRUE)

# Constraints 1: Introduce NA values for shipping_date, and if shipping_date is NA, set delivery_status to 'incomplete'
complete_indices <- which(payment_status == 'complete')

```



```

na_indices_shipping <- sample(complete_indices, round(length(complete_indices) * 0.1))
shipping_date[na_indices_shipping] <- NA
delivered_date[na_indices_shipping] <- NA

# Constraints 2: Introduce NA values for delivered_date for some observations where shipping
non_na_shipping_indices <- which(!is.na(shipping_date))
na_indices_delivered <- sample(non_na_shipping_indices, round(length(non_na_shipping_indices)
delivered_date[na_indices_delivered] <- NA

# Constraints 3: For observations with 'declined' payment status, set tracking_number, shipping_date, delivered_date
declined_indices <- which(payment_status == 'declined')
tracking_numbers <- rep(NA, n)
shipping_date[declined_indices] <- NA
delivered_date[declined_indices] <- NA

# Generate tracking numbers for observations with 'complete' payment status
tracking_numbers[complete_indices] <- sapply(1:length(complete_indices), function(x) generate_tracking_number())

# Create shipment_status based on conditions
shipment_status <- ifelse(payment_status == 'declined', 'No Shipment',
                          ifelse(is.na(shipping_date), 'Packaging',
                                ifelse(is.na(delivered_date), 'In Transit', 'Delivered')))

# Create data frame
shipping_data <- data.frame(
  shipping_id = shipping_ids,
  tracking_number = tracking_numbers,
  purchased_date = purchased_date,
  shipping_date = shipping_date,
  delivered_date = delivered_date,
  shipment_status = shipment_status
)

# Write the data to an Excel file on the desktop
write.xlsx(shipping_data, file.path("data_upload", "Shipping_table.xlsx"), rowNames = FALSE)

```

## 2.2: Data Import and Quality Assurance

### 2.2.1: Import data

Loading using `read_excel` and `dbWriteTable()`

```
customer <- readxl::read_excel("data_upload/Customer/Customer_table.xlsx")
category <- readxl::read_excel("data_upload/Category/Category_table.xlsx")
discount <- readxl::read_excel("data_upload/Discount/Discount_table.xlsx")
order <- readxl::read_excel("data_upload/Order/Order_table.xlsx")
product <- readxl::read_excel("data_upload/Product/Product_table.xlsx")
shipment <- readxl::read_excel("data_upload/Shipment/Shipment_table.xlsx")
supplier <- readxl::read_excel("data_upload/Supplier/Supplier_table.xlsx")
```

```
customer$cust_birthday <- as.character(customer$cust_birthday)
order$order_purchased_date <- as.character(order$order_purchased_date)
shipment$shipping_date <- as.character(shipment$shipping_date)
shipment$delivered_date <- as.character(shipment$delivered_date)
```

Write them to the database

```
RSQLite::dbWriteTable(my_connection,"customer",customer, append = TRUE)
RSQLite::dbWriteTable(my_connection,"category",category,append=TRUE)
RSQLite::dbWriteTable(my_connection,"discount",discount,append=TRUE)
RSQLite::dbWriteTable(my_connection,"product",product,append=TRUE)
RSQLite::dbWriteTable(my_connection,"order",order,append=TRUE)
RSQLite::dbWriteTable(my_connection,"shipment",shipment,append=TRUE)
RSQLite::dbWriteTable(my_connection,"supplier",supplier,append=TRUE)
```

## 2.2.2: Data Validation

### Primary Key Check

The provided code snippet serves to verify the integrity of primary keys in various data tables stored within specific folders. Initially, it iterates through each file in the “Customer,” “Category,” “Discount,” “Order,” “Product,” “Shipment,” and “Supplier” folders, performing a primary key check. Each folder contains two files, with the new file presumably containing new data compared to the existing file.

Following this primary key check, the script then processes Excel files in the same folders. It converts the contents of these Excel files into appropriate data types based on predefined mappings. Subsequently, it appends the data from these Excel files into corresponding SQLite tables, ensuring consistency with the existing data and schema.

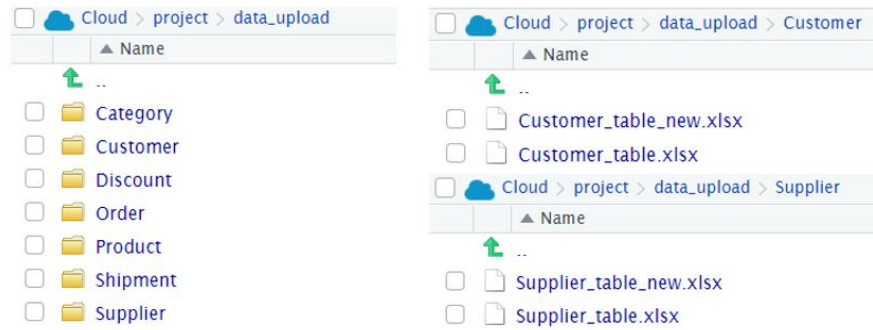


Figure 8: The “data\_upload” folder contains two files each

```
# primary key check for customer data
all_files <- list.files("data_upload/Customer/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Customer/",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_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Customer_table.xlsx"
[1] " is TRUE"
```

```
# primary key check for category data
all_files <- list.files("data_upload/Category/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Category/",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_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Category_table.xlsx"
[1] " is TRUE"
```

```
# primary key check for discount data
all_files <- list.files("data_upload/Discount/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Discount/",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: Discount_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Discount_table.xlsx"
[1] " is TRUE"
```

```
# primary key check for order data
all_files <- list.files("data_upload/Order/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Order/",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_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Order_table.xlsx"
[1] " is TRUE"
```

```
# primary key check for product data
all_files <- list.files("data_upload/Product/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Product/",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: Product_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Product_table.xlsx"
[1] " is TRUE"
```

```
# primary key check for shipment data
all_files <- list.files("data_upload/Shipment/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Shipment/",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_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Shipment_table.xlsx"
[1] " is TRUE"
```

```
# primary key check for supplier data
all_files <- list.files("data_upload/Supplier/")
for (variable in all_files) {
  this_filepath <- paste0("data_upload/Supplier/",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_table_new.xlsx"
[1] " is TRUE"
[1] "Checking for: Supplier_table.xlsx"
[1] " is TRUE"

# Function to list Excel files in a folder
list_excel_files <- function(folder_path) {
  files <- list.files(path = folder_path, pattern = "\\..xlsx$", full.names = TRUE)
  return(files)
}

folder_table_mapping <- list(
  "Customer" = "customer",
  "Supplier" = "supplier",
  "Category" = "category",
  "Product" = "product",
  "Order" = "order",
  "Discount" = "discount",
  "Shipment" = "shipment"
)

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]])
      }
    }
  }
  return(data)
}

# Data type mapping for each table's columns

```

```

column_types_mapping <- list(
  "Category" = c("category_id" = "character", "p_category_id" = "character"),
  "Customer" = c("cust_id" = "character", "cust_birthday" = "character"),
  "Supplier" = c("supplier_id" = "character"),
  "Discount" = c("discount_id" = "character"),
  "Product" = c("product_id" = "character", "supplier_id" = "character",
                "category_id" = "character", "discount_id" = "character"),
  "Shipment" = c("shipping_id" = "character"),
  "Order" = c("order_id" = "character", "customer_id" = "character",
              "product_id" = "character", "shipping_id" = "character")
)

# Path to the main folder containing subfolders (e.g., data_upload)
main_folder <- "data_upload"

# Process each subfolder (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 Excel files in the subfolder
    excel_files <- list_excel_files(folder_path)

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

    # Append data from Excel files to the corresponding table
    for (excel_file in excel_files) {
      cat("Appending data from:", excel_file, "\n")
      tryCatch({
        # Read Excel file
        file_contents <- readxl::read_excel(excel_file)

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

        # 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:", excel_file, "\n")
        #cat("Error message:", e$message, "\n")
      })
    }
  }
}

```

```

    })
  }
} else {
  cat("Folder does not exist:", folder_path, "\n")
}
}

```

```

Processing folder: Customer
Appending data from: data_upload/Customer/Customer_table_new.xlsx
Data appended to table: customer
Appending data from: data_upload/Customer/Customer_table.xlsx
Processing folder: Supplier
Appending data from: data_upload/Supplier/Supplier_table_new.xlsx
Data appended to table: supplier
Appending data from: data_upload/Supplier/Supplier_table.xlsx
Processing folder: Category
Appending data from: data_upload/Category/Category_table_new.xlsx
Data appended to table: category
Appending data from: data_upload/Category/Category_table.xlsx
Processing folder: Product
Appending data from: data_upload/Product/Product_table_new.xlsx
Data appended to table: product
Appending data from: data_upload/Product/Product_table.xlsx
Processing folder: Order
Appending data from: data_upload/Order/Order_table_new.xlsx
Data appended to table: order
Appending data from: data_upload/Order/Order_table.xlsx
Data appended to table: order
Processing folder: Discount
Appending data from: data_upload/Discount/Discount_table_new.xlsx
Data appended to table: discount
Appending data from: data_upload/Discount/Discount_table.xlsx
Processing folder: Shipment
Appending data from: data_upload/Shipment/Shipment_table_new.xlsx
Data appended to table: shipment
Appending data from: data_upload/Shipment/Shipment_table.xlsx

```

```

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

```

```

[1] "category" "customer" "discount" "order"      "product"  "shipment" "supplier"

```



```
customer <- dbGetQuery(my_connection, "SELECT * FROM customer")
supplier <- dbGetQuery(my_connection, "SELECT * FROM supplier")
discount <- dbGetQuery(my_connection, "SELECT * FROM discount")
product <- dbGetQuery(my_connection, "SELECT * FROM product")
order <- dbGetQuery(my_connection, "SELECT * FROM 'order'")
shipment <- dbGetQuery(my_connection, "SELECT * FROM shipment")
category <- dbGetQuery(my_connection, "SELECT * FROM category")
```

## Format Validation

Key validation steps include:

1. **Data Integrity Checks:** Ensuring that data adheres to predefined rules and constraints, such as primary and foreign key relationships, to maintain database integrity.
2. **Data Format Validation:** Verifying the format of critical data fields, such as email addresses, phone numbers, dates of birth, and postal codes, to ensure they meet specified standards.
3. **Duplicate Record Detection:** Identifying and removing duplicate records to prevent data redundancy and ensure data accuracy.
4. **Range and Constraint Validation:** Validating numerical values, such as product prices, stock levels, discount percentages, and order quantities, to ensure they fall within acceptable ranges and adhere to defined constraints.
5. **Consistency Checks:** Ensuring consistency in data across related tables, such as maintaining consistent parent category IDs in the Category Table or ensuring consistent customer and order details in the Order Table.
6. **Reference Integrity Verification:** Verifying that foreign key relationships are correctly linked to corresponding primary keys, maintaining the referential integrity of the database.

Overall, the validation process aims to identify and rectify any discrepancies, errors, or inconsistencies in the data, thereby ensuring the reliability and usability of the e-commerce database for effective business operations.

```
library(RSQLite)
library(DBI)
library(lubridate)

# Connect to database
connection <- dbConnect(RSQLite::SQLite(), dbname= 'project_dm.db')
```

```

# Function to check email format
check_email_format <- function(email) {
  valid.email <- grepl("[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.com$", email)
  return(valid.email)
}

# Function to check phone number format
check_phone_format <- function(phone) {
  valid.phone <- grepl("\\+\\d{2} \\d{4} \\d{6}$", phone)
  return(valid.phone)
}

#Function to check date
check_date_format <- function(date){
  valid.date <- grepl("^\\d{4}-\\d{2}-\\d{2}$", date)
  return(valid.date)
}

#Function to check postcode
check_postcode_format <- function(post){
  valid.postcode <- grepl("[A-Z]{1,2}\\d{1,2} \\d[A-Z]{2}$",post)
  return(valid.postcode)
}

#Function to check age
check_age <- function(age){
  current_year <- year(Sys.Date())
  birth_years <- year(age)
  ages <- current_year - birth_years
  valid_age <- ages >= 18 & ages <= 60
  return(valid_age)
}

#Function to check price
price_range_check <- function(price) {
  valid_price <- price >= 20 & price <= 150
  return(valid_price)
}

#Function to check stock
stock_check <- function(stock){
  valid_stock <- stock >=0

```

```

    return(valid_stock)
}

## Customer Table
# Email Validation
query_emails.customer <- "SELECT cust_email FROM customer"

# Execute the query
email.customer <- dbGetQuery(connection, query_emails.customer)
# Apply the function to check email format
email_validity.customer <- sapply(email.customer, check_email_format)

# Print their validity
cust.email.validity <- data.frame(Email = email.customer, Valid = email_validity.customer)

invalid.email <- subset(cust.email.validity, cust_email.1 == FALSE)
print(paste("Number of invalid customer emails:",nrow(invalid.email)))

```

```
[1] "Number of invalid customer emails: 262"
```

```

# Phone number validation
query_phone.customer <- "SELECT cust_phone FROM customer"

phone.customer <- dbGetQuery(connection, query_phone.customer)

phone_validity.customer <- sapply(phone.customer, check_phone_format)

cust.phone.validity <- data.frame(Phone = phone.customer, Valid = phone_validity.customer)

invalid.phone <- subset(cust.phone.validity, cust_phone.1 == FALSE)
print(paste("Number of invalid customer phone numbers:",nrow(invalid.phone)))

```

```
[1] "Number of invalid customer phone numbers: 0"
```

```

#Date validation
query_date.customer <- "SELECT cust_birthday FROM customer"

date.customer <- dbGetQuery(connection, query_date.customer)

```

```

date_validity.customer <- sapply(date.customer, check_date_format)

cust.date.validity <- data.frame(Date=date.customer, Valid=date_validity.customer)

invalid.date <- subset(cust.date.validity, cust_birthday.1==FALSE)
print(paste("Number of invalid customer birthdays:",nrow(invalid.date)))

```

```
[1] "Number of invalid customer birthdays: 10"
```

```

#Postcode validation
query_postcode.customer <- "SELECT cust_postcode FROM customer"

postcode.customer <- dbGetQuery(connection, query_postcode.customer)

postcode_validity.customer <- sapply(postcode.customer, check_postcode_format)

cust.postcode.validity <- data.frame(postcode=postcode.customer, Valid=postcode_validity.cust

invalid.post <- subset(cust.postcode.validity, cust_postcode.1==FALSE)
print(paste("Number of invalid customer postcodes:",nrow(invalid.post)))

```

```
[1] "Number of invalid customer postcodes: 0"
```

```

#Age Validation

query_age.customer <- "SELECT cust_birthday FROM customer"

age.customer <- dbGetQuery(connection, query_age.customer)

age.customer$cust_birthday <- as.Date(age.customer$cust_birthday)

age.customer$age <- year(Sys.Date()) - year(age.customer$cust_birthday)

# Define thresholds for invalid ages
invalid_age_min <- 18
invalid_age_max <- 100

# Count customers under 18 and over 100
n_under_18 <- sum(age.customer$age < invalid_age_min)
n_over_100 <- sum(age.customer$age > invalid_age_max)

```

```
# Print the results
cat("Number of customers under 18:", n_under_18, "\n")
```

Number of customers under 18: NA

```
cat("Number of customers over 100:", n_over_100, "\n")
```

Number of customers over 100: NA

```
#Duplicates
check_duplicates.customer <- "
  SELECT *
  FROM customer
  WHERE cust_id IN (
    SELECT cust_id
    FROM customer
    GROUP BY cust_id
    HAVING COUNT(*) > 1
  )
  OR cust_email IN (
    SELECT cust_email
    FROM customer
    GROUP BY cust_email
    HAVING COUNT(*) > 1
  )
  OR cust_phone IN (
    SELECT cust_phone
    FROM customer
    GROUP BY cust_phone
    HAVING COUNT(*) > 1
  )
"

# Execute the query
duplicates.customer <- dbGetQuery(connection, check_duplicates.customer)

print(paste("Number of duplicates in customer table:", nrow(duplicates.customer)))
```

[1] "Number of duplicates in customer table: 0"

```

#Supplier Table
#Email
query_emails.supplier <- "SELECT supplier_email FROM supplier"

email.supplier <- dbGetQuery(connection, query_emails.supplier)

email_validity.supplier <- sapply(email.supplier, check_email_format)

supplier.email.validity <- data.frame(Email = email.supplier, Valid = email_validity.supplier)

invalid.email.supplier <- subset(supplier.email.validity, supplier_email.1==FALSE)
print(paste("Number of invalid supplier emails:",nrow(invalid.email.supplier)))

```

```
[1] "Number of invalid supplier emails: 0"
```

```

#Duplicates
check_duplicates.supplier <- "
    SELECT *
    FROM supplier
    WHERE supplier_id IN (
        SELECT supplier_id
        FROM supplier
        GROUP BY supplier_id
        HAVING COUNT(*) > 1
    )
    OR supplier_email IN (
        SELECT supplier_email
        FROM supplier
        GROUP BY supplier_email
        HAVING COUNT(*) > 1
    )
"

# Execute the query
duplicates.supplier <- dbGetQuery(connection, check_duplicates.supplier)
print(paste("Number of duplicates in supplier :",nrow(duplicates.supplier)))

```

```
[1] "Number of duplicates in supplier : 2"
```

```

#Product Table
#Price Validation
query_price.product <- "SELECT product_price FROM product"

price.product <- dbGetQuery(connection, query_price.product)

price_validity.product <- sapply(price.product, price_range_check)

product.price.validity <- data.frame(price = price.product, Valid = price_validity.product)

invalid.price.product <- subset(product.price.validity, product_price.1==FALSE)
print(paste("Number of invalid product prices:",nrow(invalid.price.product)))

```

```
[1] "Number of invalid product prices: 2"
```

```

#Stock validation
query_stock.product <- "SELECT product_stock FROM product"

stock.product <- dbGetQuery(connection, query_stock.product)

stock_validity.product <- sapply(price.product, stock_check)

product.stock.validity <- data.frame(stock = stock.product, Valid = stock_validity.product)

invalid.stock.product <- subset(product.stock.validity, product_price==FALSE)
print(paste("Number of invalid product stocks:",nrow(invalid.stock.product)))

```

```
[1] "Number of invalid product stocks: 0"
```

```

#Duplicates
check_duplicates.product <- "
  SELECT *
  FROM product
  WHERE product_id IN (
    SELECT product_id
    FROM product
    GROUP BY product_id
    HAVING COUNT(*) > 1
  )
"

```

```
# Execute the query
duplicates.product <- dbGetQuery(connection, check_duplicates.product)
print(paste("Number of duplicates in product :",nrow(duplicates.product)))
```

```
[1] "Number of duplicates in product : 0"
```

```
#Reference Integrity
query.categoryid <- "
    UPDATE product
    SET category_id = CASE
                                WHEN category_id NOT IN (SELECT category_id FROM category) THEN 1
                                ELSE category_id
                            END;"
query.supplierid <- "
    UPDATE product
    SET supplier_id = CASE
                                WHEN supplier_id NOT IN (SELECT supplier_id FROM supplier) THEN 1
                                ELSE supplier_id
                            END;"
query.discountid <- "
    UPDATE product
    SET discount_id = CASE
                                WHEN discount_id NOT IN (SELECT discount_id FROM discount) THEN 1
                                ELSE discount_id
                            END;"
dbExecute(connection, query.categoryid)
```

```
[1] 100
```

```
dbExecute(connection, query.supplierid)
```

```
[1] 100
```

```
dbExecute(connection, query.discountid)
```

```
[1] 100
```



```
## Shipment Table
# Date format validation
query_shippingdate.shipment <- "SELECT shipping_date FROM shipment"

shippingdate.shipment <- dbGetQuery(connection, query_shippingdate.shipment)

shippingdate_validity.shipment <- sapply(shippingdate.shipment, check_date_format)

shipment.shipmentdate.validity <- data.frame(shippingdate = shippingdate.shipment, Valid = sapply(shippingdate.shipment, check_date_format))

invalid_shippingdate <- subset(shipment.shipmentdate.validity, shipping_date.1==FALSE)
print(paste("Number of invalid shipping date:", nrow(invalid_shippingdate)))
```

```
[1] "Number of invalid shipping date: 93"
```

```
query_deliverydate.shipment <- "SELECT delivered_date FROM shipment"

deliverydate.shipment <- dbGetQuery(connection, query_deliverydate.shipment)

deliverydate_validity.shipment <- sapply(deliverydate.shipment, check_date_format)

shipment.deliverydate.validity <- data.frame(deliverydate = deliverydate.shipment, Valid = sapply(deliverydate.shipment, check_date_format))

invalid_deliverydate <- subset(shipment.deliverydate.validity, delivered_date.1==FALSE)
print(paste("Number of invalid delivery dates:", nrow(invalid_deliverydate)))
```

```
[1] "Number of invalid delivery dates: 147"
```

```
#Logical date check
check_shipping_delivered_dates <-
  shipping_dates <- as.Date(shippingdate.shipment$shipping_date)
  delivered_dates <- as.Date(deliverydate.shipment$delivered_date)

is_before <- shipping_dates < delivered_dates
print(paste("Number of invalid shipment:", nrow(is_before)))
```

```
[1] "Number of invalid shipment: "
```

```
## Discount Table
# Query to identify entries with duplicate discount names or duplicate discount percentages
query.discount <- "
    SELECT discount_id, discount_name, discount_percentage, COUNT(*) AS num_duplicates
    FROM discount
    GROUP BY discount_name, discount_percentage
    HAVING COUNT(*) > 1
"
duplicate_entries <- dbGetQuery(connection, query.discount)
print(paste("Number of duplicate discount names or percentages:", nrow(duplicate_entries)))
```

```
[1] "Number of duplicate discount names or percentages: 0"
```

```
# Loop over duplicate entries and remove them
for (i in 1:nrow(duplicate_entries)) {
  discount_id <- duplicate_entries[i, "discount_id"]
  discount_name <- duplicate_entries[i, "discount_name"]
  discount_percentage <- duplicate_entries[i, "discount_percentage"]

  # Delete entries with duplicate discount names or duplicate discount percentages
  query.discount <- "DELETE FROM discount
    WHERE (discount_name = ? AND discount_id <> ?)
    OR (discount_percentage = ? AND discount_id <> ?)"
  dbExecute(connection, query.discount, params = list(discount_name, discount_id, discount_p
})
```

```
# Query to identify entries with discount percentages not within the range of 0 and 1
query.discount2 <- "
    SELECT discount_id, discount_name, discount_percentage
    FROM discount
    WHERE discount_percentage < 0 OR discount_percentage > 1
"
out_of_range_entries <- dbGetQuery(connection, query.discount2)
print(paste("Number of out of range discount percentages:", nrow(out_of_range_entries)))
```

```
[1] "Number of out of range discount percentages: 0"
```

```
# Loop over out-of-range entries and remove them
for (i in 1:nrow(out_of_range_entries)) {
  discount_id <- out_of_range_entries[i, "discount_id"]
```

```

# Delete entries with discount percentages not within the range of 0 and 1
query.discount2 <- "DELETE FROM discount WHERE discount_id = ?"
dbExecute(connection, query.discount2, params = list(discount_id))
}

# Query to identify entries with duplicate parent category IDs and category names
query.discount3 <- "
  SELECT p_category_id, category_name, MIN(category_id) AS keep_category_id, COUNT(*) AS n
  FROM category
  GROUP BY p_category_id, category_name
  HAVING COUNT(*) > 1
"
duplicate_entries <- dbGetQuery(connection, query.discount3)
print(paste("Number of duplicate parent category IDs and category names :", nrow(duplicate_entries)))

```

```
[1] "Number of duplicate parent category IDs and category names : 0"
```

```

# Loop over duplicate entries and remove all but one of the duplicates
for (i in 1:nrow(duplicate_entries)) {
  p_category_id <- duplicate_entries[i, "p_category_id"]
  category_name <- duplicate_entries[i, "category_name"]
  keep_category_id <- duplicate_entries[i, "keep_category_id"]

  # Delete duplicate entries except for the one to keep
  query.discount3 <- "DELETE FROM category WHERE p_category_id = ? AND category_name = ? AND category_id != ?"
  dbExecute(connection, query.discount3, params = list(p_category_id, category_name, keep_category_id))
}

# SQL statement to update the p_category_id column based on the constraint
query.discount4 <- "
  UPDATE category
  SET p_category_id = CASE
    WHEN category_id = p_category_id THEN NULL
    ELSE p_category_id
  END;
"
# Execute the SQL statement
dbExecute(connection, query.discount4)

```

[1] 26

```
## Category Table

# SQL statement to update the p_category_id column based on the constraint
query.category <- "
    UPDATE category
    SET p_category_id = CASE
                                WHEN p_category_id NOT IN (SELECT category_id FROM category) THEN
                                ELSE p_category_id
                            END;
"

# Execute the SQL statement
dbExecute(connection, query.category)
```

[1] 26

```
## Order Table

#Reference Integrity
query.customerid<- "
    UPDATE 'order'
    SET customer_id = CASE
                                WHEN customer_id NOT IN (SELECT customer_id FROM customer) THEN
                                ELSE customer_id
                            END;"

query.productid <- "
    UPDATE 'order'
    SET product_id = CASE
                                WHEN product_id NOT IN (SELECT product_id FROM product) THEN NULL
                                ELSE product_id
                            END;"

query.shippingid <- "
    UPDATE 'order'
    SET shipping_id = CASE
                                WHEN shipping_id NOT IN (SELECT shipping_id FROM shipment) THEN
                                ELSE shipping_id
                            END;"

dbExecute(connection, query.customerid)
```

```
[1] 2000
```

```
dbExecute(connection, query.productid)
```

```
[1] 2000
```

```
dbExecute(connection, query.shippingid)
```

```
[1] 2000
```

```
# Quantity Validation
query.order.quantity <- "SELECT order_quantity FROM 'order'"
order.quantity <- dbGetQuery(connection, query.order.quantity)

quantity.validate <- order.quantity>=1 & order.quantity <=5
invalid.quantity <- subset(quantity.validate, order.quantity==FALSE)
print(paste("Number of invalid quantitiy in order:",nrow(invalid.quantity)))
```

```
[1] "Number of invalid quantitiy in order: 0"
```

```
#To ensure the foreign key (customer, product and shipping) in the order table is linked with
# SQL statement to delete observations
query.order1 <- "
    DELETE FROM 'order'
    WHERE customer_id NOT IN (SELECT cust_id FROM customer)
    OR product_id NOT IN (SELECT product_id FROM product)
    OR shipping_id NOT IN (SELECT shipping_id FROM shipment);
"

# Execute the SQL statement
dbExecute(connection, query.order1)
```

```
[1] 0
```

```
#To ensure for the same order ID, the customer, shipment, payment and purchased date will be
# SQL statement to delete observations that violate the constraint
query.order2 <- "
    DELETE FROM 'order'
    WHERE (order_id, customer_id, shipping_id, payment_method, order_purchased_date, payment.
```

```

        NOT IN (
            SELECT order_id, customer_id, shipping_id, payment_method, order_purchased_date
            FROM 'order'
            GROUP BY order_id
        );
    "

# Execute the SQL statement
dbExecute(connection, query.order2)

```

```
[1] 1000
```

```

#To ensure there would be no repeated product entries for the same order
# Query to identify unique combinations of order_id and product_id with duplicates
query.order3 <- "
    SELECT order_id, product_id
    FROM 'order'
    GROUP BY order_id, product_id
    HAVING COUNT(*) > 1
"

duplicate_combinations <- dbGetQuery(connection, query.order3)

# Loop over duplicate combinations and selectively delete duplicate observations
for (i in 1:nrow(duplicate_combinations)) {
    order_id <- duplicate_combinations[i, "order_id"]
    product_id <- duplicate_combinations[i, "product_id"]

    # Query to select all duplicate observations for the current combination
    query.order3 <- "
        SELECT rowid
        FROM 'order'
        WHERE order_id = ? AND product_id = ?
        ORDER BY rowid
    "

    duplicate_obs <- dbGetQuery(connection, query.order3, params = list(order_id, product_id))

    # Keep the first observation and delete the rest
    if (nrow(duplicate_obs) > 1) {
        rows_to_keep <- duplicate_obs$rowid[1]
        rows_to_delete <- duplicate_obs$rowid[-1]
    }
}

```

```

# Delete duplicate observations
for (rowid in rows_to_delete) {
  query.order3 <- "DELETE FROM 'order' WHERE rowid = ?"
  dbExecute(connection, query.order3, params = list(rowid))
}
}

# Ensure the range of the product rating
# SQL statement to update product ratings
query.order4 <- "
  UPDATE 'order'
  SET product_rating = CASE
    WHEN product_rating < 1 THEN 1
    WHEN product_rating > 5 THEN 5
    ELSE product_rating
  END;
"

# Execute the SQL statement
dbExecute(connection, query.order4)

```

[1] 995

```

#Ensure consistent order payment method
query.order5 <- "
  UPDATE 'order'
  SET payment_method = CASE
    WHEN payment_method IN ('online banking', 'paypal', 'credit/debit card', 'gift card')
    ELSE 'others'
  END;
"

dbExecute(connection, query.order5)

```

[1] 995

```

#Ensure consistent payment status
query.order6 <- "
UPDATE 'order'

```

```

SET payment_status = CASE
  WHEN payment_status IN ('complete', 'declined') THEN payment_status
  ELSE payment_status IS NULL
END;
"
dbExecute(connection, query.order6)

```

[1] 995

## Part 3 : Data Pipeline Generation

For GitHub workflow automation:

- R packages to be installed are added to pipeline
- We have automated all R scripts which includes Analysis and Validation of the project to run when there is a change in code.
- Automated R scripts when new data is loaded into tables.
- When new data is added and the potential graphs that need to be saved with new data are automated using the git add and commit commands in the pipeline.
- Push changes are added using the secret token in actions.

We were able to successfully automate the required actions in the Git workflow. Here is our Github page : [Github\\_Group29](#)

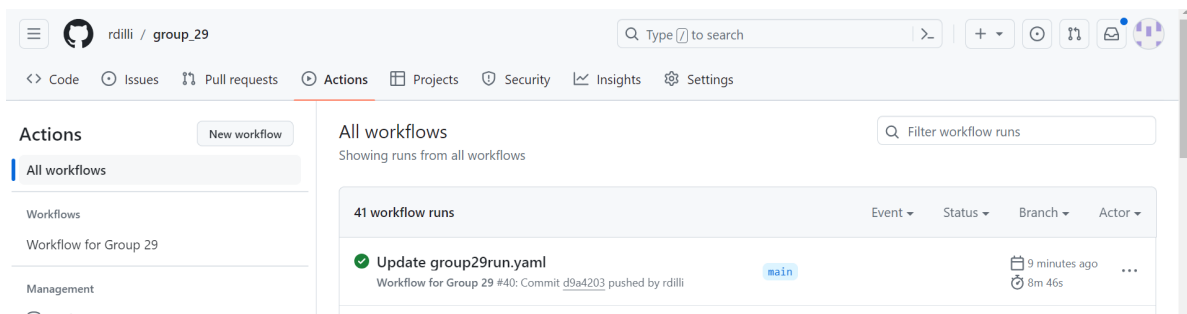


Figure 9: Github Page



group\_29 Public

Pin Unwatch 1 Fork 0 Star 0

main 1 Branch 0 Tags

Go to file Add file Code

rdilli Merge branch 'main' of https://github.com/rdilli/group\_29 into main 3d616a5 · 1 minute ago 55 Commits

.github/workflows	Update group29run.yaml	12 minutes ago
Picture	adding Pictures	1 minute ago
R	adding R script to git	15 minutes ago
data_upload	adding final changes to code	7 hours ago
graphs	Adding DM group 29 analysis graphs	3 minutes ago
.gitignore	Initial commit	5 days ago
README.md	Initial commit	5 days ago
project_dm.db	adding final db	5 hours ago

About

DM Group Assignment Group 29

Readme Activity 0 stars 1 watching 0 forks

Releases

No releases published [Create a new release](#)

Packages

No packages published [Publish your first package](#)

Figure 10: Workflow Runs

```
name: Workflow for Group 29

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

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

```

    if: steps.cache.outputs.cache-hit != 'true'
    run: |
        Rscript -e 'install.packages(c("ggplot2","RSQLite","stringr","readxl","dplyr","readr"))'
- name: Execute R script
  run: |
    Rscript R/Data_Analysis.R
- name: Execute R script
  run: |
    Rscript R/Data_Validation.R
- name: Add files
  run: |
    git config --global user.email "ramyad6222@gmail.com"
    git config --global user.name "rdilli"
    git add --all graphs/
- name: Commit files
  run: |
    git commit -m "Adding DM group 29 analysis graphs"
- name: Push changes
  uses: ad-m/github-push-action@v0.6.0
  with:
    github_token: ${{ secrets.GROUP29_TOKEN }}
    branch: main

```

### Challenges:

While adding jobs to workflow, we have faced the following challenges:

- We encountered difficulties with package installations in our pipeline due to instances where packages were added but not utilized in the R script, leading to conflicts. To address this, we conducted a thorough review of the pipeline configuration and scripts to ensure that only necessary packages are being installed.
- Additionally, we faced workflow failures during push changes because the permissions settings for read/write were missed when adding tokens in Actions. To rectify this issue, we implemented a process to double-check the permissions settings whenever generating or updating tokens for workflows.

## **Part 4: Data Analysis and Visualisation**

A comprehensive analysis covering several critical aspects including key metric performance, customer portfolio analysis, utilisation of payment methods, popularity of product categories,

factors contribute to sales and evaluation of the performance of our suppliers has been conducted.

#### 4.1: The Key Metrics Performance Analysis (Revenue, Order Completion Day)

This analysis focuses on the company's daily revenue dynamics, aiming to identify patterns like cyclical trends or fluctuations in this time series data. To provide management with concise insights into the company's performance.

```
# Connect to the SQLite database
conn <- dbConnect(RSQLite::SQLite(), dbname = 'project_dm.db')

# Execute the SQL query
query <- "
    SELECT o.order_purchased_date,
           SUM(o.order_quantity) AS total_order_quantity,
           SUM(o.order_quantity * p.product_price * (1 - d.discount_percentage)) AS total_sales,
           SUM(o.order_quantity * ((p.product_price * (1 - d.discount_percentage)) - p.product_price)) AS total_discount
    FROM \"order\" o
    JOIN product p ON o.product_id = p.product_id
    LEFT JOIN discount d ON p.discount_id = d.discount_id
    WHERE o.payment_status = 'complete'
    GROUP BY o.order_purchased_date
    ORDER BY o.order_purchased_date ASC;
"
daily_revenue <- dbGetQuery(conn, query)

# Close the database connection
dbDisconnect(conn)

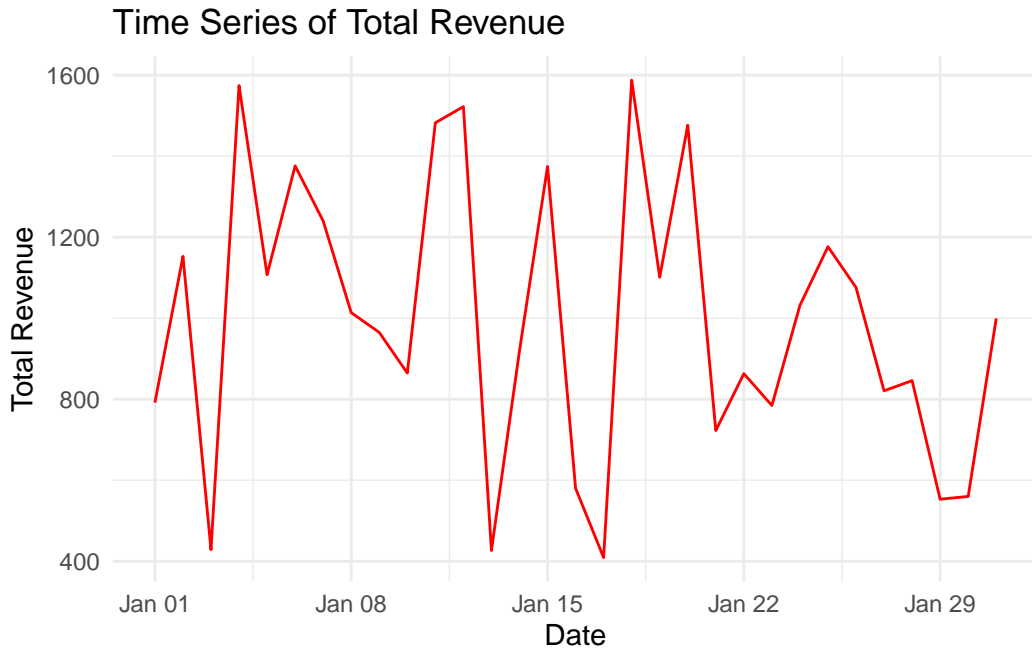
# Convert the order_purchased_date column to Date format
daily_revenue$order_purchased_date <- as.Date(daily_revenue$order_purchased_date)

# Create time series plot for total revenue
g1 <- ggplot(daily_revenue, aes(x = order_purchased_date, y = total_revenue)) +
  geom_line(color = "red") +
  labs(title = "Time Series of Total Revenue",
       x = "Date",
       y = "Total Revenue") +
  theme_minimal()

print(g1)
```

Table 1: Average order completion day

Order_Competion_days
12.3



```
ggsave(plot = g1, filename = "graphs/revenue_trend.jpeg", width = 10, height=5,dpi=300)
```

This section calculates the average order fulfilment time by subtracting the order placement date from the delivery date. Analysing this metric helps assess our efficiency in order processing, offering insights into operational capacity for strategic planning and implementation.

```
# Execute the SQL query
completion_day <-RSQLite::dbGetQuery(my_connection,"SELECT AVG(julianday(s.delivered_date) -
FROM 'order'o
JOIN shipment s ON o.shipping_id = s.shipping_id")

completion_day %>%
  kbl(digits = 1, caption = "Average order completion day") %>%
  kable_styling()
```

## 4.2: Customer Analysis

This section offers insights into spending patterns on our products that highlights the varying preferences for clothing among different age demographics. By understanding which age groups

exhibit higher spending tendencies for each gender, we gain valuable insights for targeted marketing efforts to maximize engagement with our current customer base. Furthermore, this analysis presents an opportunity for market expansion by identifying the needs and preferences of age groups with lower spending tendencies.

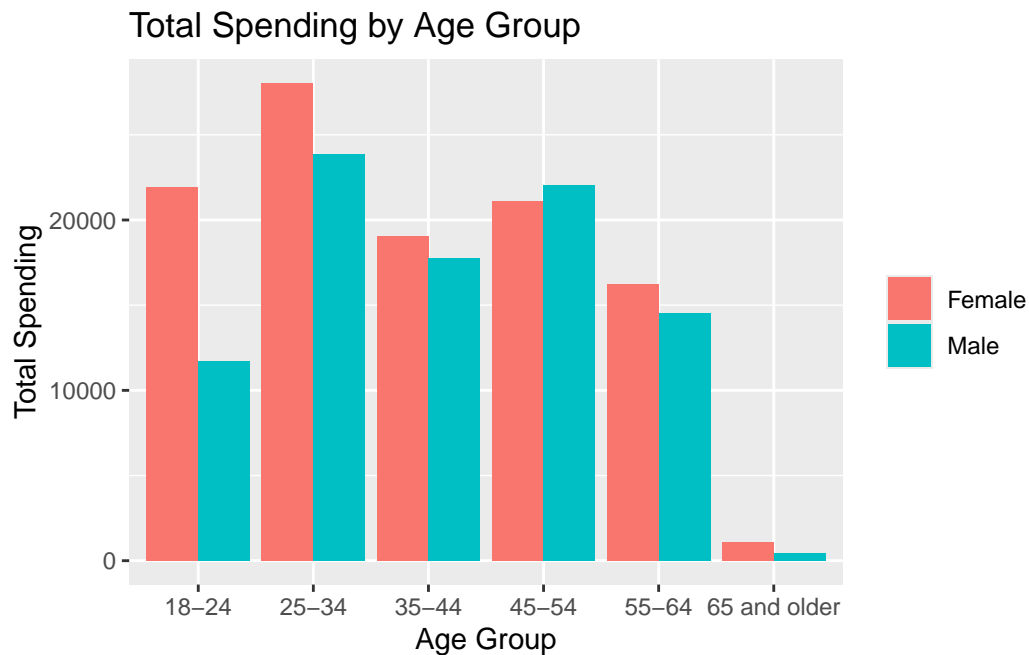
```
# Connect to the SQLite database
conn <- dbConnect(RSQLite::SQLite(), dbname = 'project_dm.db')

# Execute the SQL query
query <- "
    SELECT c.cust_gender,
           CASE
             WHEN strftime('%Y', 'now') - strftime('%Y', c.cust_birthday) BETWEEN 18 AND 24 THEN
             WHEN strftime('%Y', 'now') - strftime('%Y', c.cust_birthday) BETWEEN 25 AND 34 THEN
             WHEN strftime('%Y', 'now') - strftime('%Y', c.cust_birthday) BETWEEN 35 AND 44 THEN
             WHEN strftime('%Y', 'now') - strftime('%Y', c.cust_birthday) BETWEEN 45 AND 54 THEN
             WHEN strftime('%Y', 'now') - strftime('%Y', c.cust_birthday) BETWEEN 55 AND 64 THEN
             ELSE '65 and older'
           END AS age_group,
           COUNT(DISTINCT c.cust_id) AS customer_count,
           SUM(o.order_quantity * p.product_price * (1 - d.discount_percentage)) AS total_spending
    FROM \"order\" o
    JOIN product p ON o.product_id = p.product_id
    LEFT JOIN discount d ON p.discount_id = d.discount_id
    JOIN customer c ON o.customer_id = c.cust_id
    WHERE o.payment_status = 'complete'
    GROUP BY c.cust_gender, age_group
    ORDER BY c.cust_gender, MIN(strftime('%Y', 'now') - strftime('%Y', c.cust_birthday));
"
customer_portfolio <- dbGetQuery(conn, query)

# Close the database connection
dbDisconnect(conn)

g2<- ggplot(data=customer_portfolio, aes(x=age_group, y=total_spending, fill=cust_gender)) +
  geom_bar(stat="identity", position=position_dodge()) +
  labs(title = "Total Spending by Age Group",
       x = "Age Group",
       y = "Total Spending") +
  theme(legend.title = element_blank())

print(g2)
```



```
ggsave(plot = g2, filename = "graphs/Total_Spending_by_Age_Group.jpeg", width = 10, height=8)
```

### 4.3: Payment Method

This section presents an analysis of the most utilized payment methods by our customers. Analyzing the most frequently used payment methods for completed transactions as well as for declined transactions provides valuable information for optimizing our payment processing systems, and addressing any potential issues that may arise with specific payment methods.

```
# Connect to the SQLite database
conn <- dbConnect(RSQLite::SQLite(), dbname = 'project_dm.db')

# Execute the SQL query
query <- "
    SELECT o.payment_status,
           o.payment_method,
           COUNT(*) AS number_of_used
    FROM \"order\" o
    GROUP BY o.payment_status, o.payment_method
    ORDER BY number_of_used DESC;
"
payment_method <- dbGetQuery(conn, query)
```

```

# Close the database connection
dbDisconnect(conn)

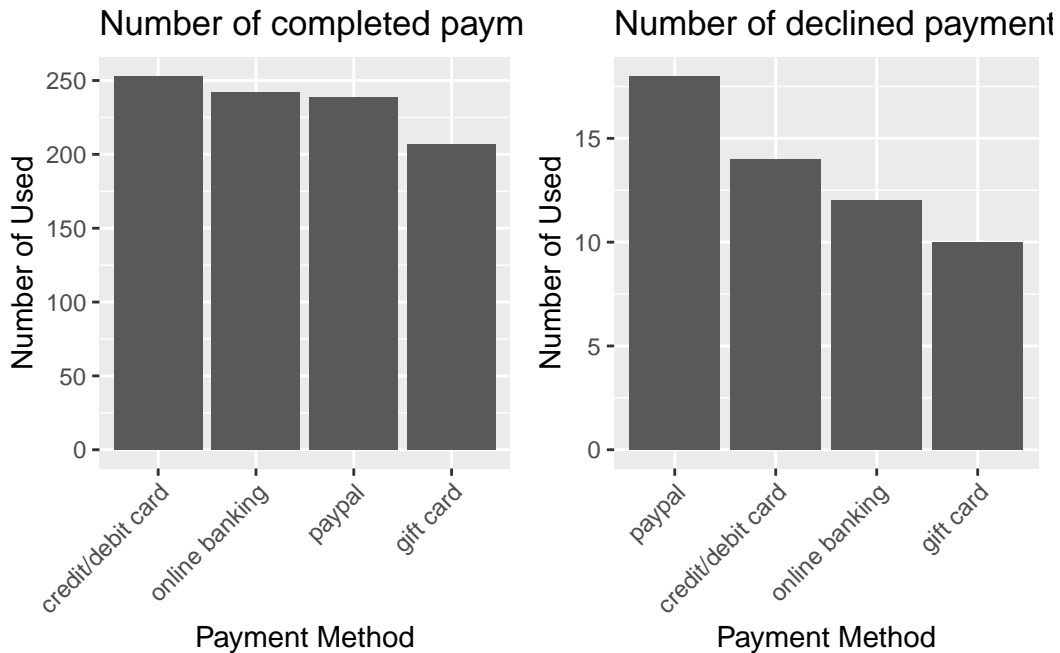
# Separate the data into men and women
complete_payment <- payment_method[payment_method$payment_status == "complete", ]
declined_payment <- payment_method[payment_method$payment_status == "declined", ]

# Create separate plots for men and women
plot_complete <- ggplot(complete_payment, aes(x = factor(payment_method, levels = payment_me
  geom_bar(stat = "identity", show.legend = FALSE) +
  labs(title = "Number of completed payment",
        x = "Payment Method",
        y = "Number of Used") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1),
        legend.title = element_blank())

plot_declined <- ggplot(declined_payment, aes(x = factor(payment_method, levels = payment_me
  geom_bar(stat = "identity", show.legend = FALSE) +
  labs(title = "Number of declined payment",
        x = "Payment Method",
        y = "Number of Used") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1),
        legend.title = element_blank())

# Arrange the plots using grid.arrange
g3 <- grid.arrange(plot_complete, plot_declined, nrow = 1)

```



```
ggsave(plot = g3, filename = "graphs/payment_method.jpeg", width = 10, height=8,dpi=300)
```

#### 4.4: Category Analysis

By analysing the revenue rankings of categories within women's and men's clothing, we aim to identify priority areas for strategic focus. This entails determining which categories within women's clothing and which subcategories within men's clothing exhibit the highest revenue. This data-driven approach allowing us to allocate resources effectively and prioritise efforts towards optimising sales performance in targeted segments.

```
# Connect to the SQLite database
conn <- dbConnect(RSQLite::SQLite(), dbname = 'project_dm.db')

# Execute the SQL query
query <- "
  SELECT
    parent.category_name AS parent_category,
    sub.category_name AS subcategory,
    SUM(o.order_quantity) AS total_order_quantity,
    SUM(o.order_quantity * p.product_price * (1 - d.discount_percentage)) AS total_sales,
    SUM(o.order_quantity * ((p.product_price * (1 - d.discount_percentage)) - p.product_price)) AS total_discount
  FROM
```



```

        'order' o
    JOIN
        product p ON o.product_id = p.product_id
    JOIN
        category sub ON p.category_id = sub.category_id
    LEFT JOIN
        category parent ON sub.p_category_id = parent.category_id
    LEFT JOIN
        discount d ON p.discount_id = d.discount_id
    WHERE o.payment_status = 'complete'
    GROUP BY
        parent.category_name, sub.category_name
    ORDER BY
        total_order_quantity DESC
"
category_sales <- dbGetQuery(conn, query)

# Close the database connection
dbDisconnect(conn)

# Separate the data into men and women
men_data <- category_sales[category_sales$parent_category == "Men", ]
women_data <- category_sales[category_sales$parent_category == "Women", ]

# Order the data frames by total sales in descending order
men_data <- men_data[order(-men_data$total_revenue), ]
women_data <- women_data[order(-women_data$total_revenue), ]

# Create separate plots for men and women
plot_men <- ggplot(men_data, aes(x = factor(subcategory, levels = subcategory[order(-total_r
    geom_bar(stat = "identity", show.legend = FALSE) +
    labs(title = "Total Revenue by Men",
        x = "Subcategory",
        y = "Total Revenue") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1),
        legend.title = element_blank())

plot_women <- ggplot(women_data, aes(x = factor(subcategory, levels = subcategory[order(-tot
    geom_bar(stat = "identity", show.legend = FALSE) +
    labs(title = "Total Revenue by Women",
        x = "Subcategory",
        y = "Total Revenue") +

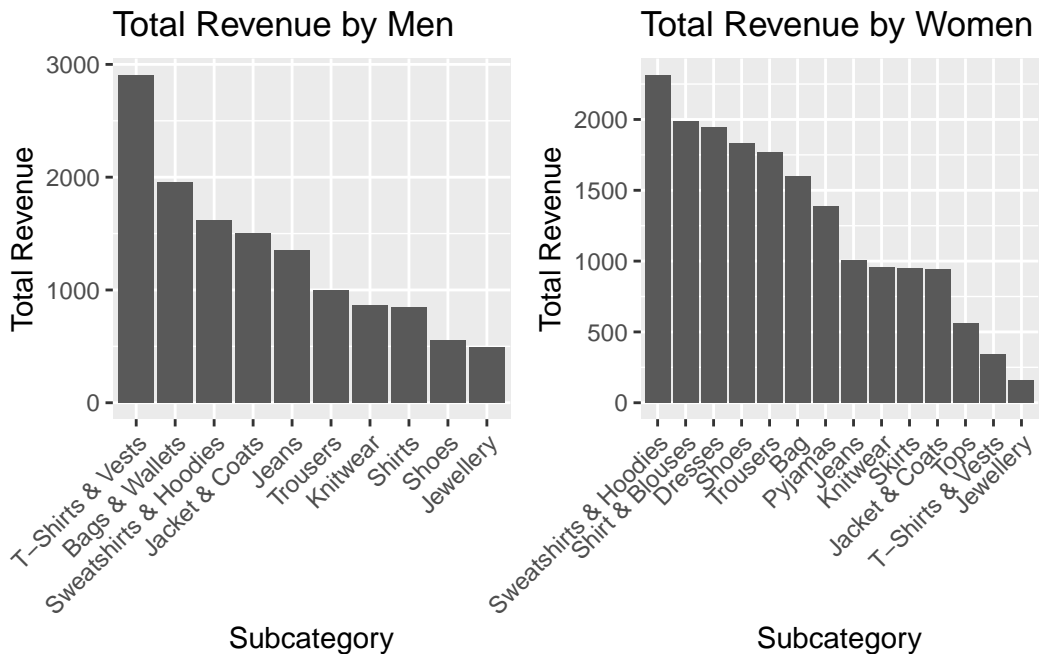
```

```

theme(axis.text.x = element_text(angle = 45, hjust = 1),
      legend.title = element_blank())

# Arrange the plots using grid.arrange
g4 <- grid.arrange(plot_men, plot_women, nrow = 1)

```



```

ggsave(plot = g4, filename = "graphs/revenue_by_category.jpeg", width = 10, height=8,dpi=300)

```

## 4.5: Product Analysis

To provide more specific insights in terms of customer preferences, the sales performance of top 5 products has been plotted to make more accurate sales forecasts and develop strategic plans.

```

# Execute the SQL query
p_revenue <-RSQLite::dbGetQuery(my_connection,"SELECT p.product_name,
      SUM(o.order_quantity) AS total_order_quantity,
      SUM(o.order_quantity * ((p.product_price * (1 - d.discount_percentage))) - p.product_p
FROM 'order' o
JOIN product p ON o.product_id = p.product_id
LEFT JOIN discount d ON p.discount_id = d.discount_id
WHERE o.payment_status = 'complete'

```

Table 2: Best Selling Products

product_name	total_order_quantity	total_revenue
Men's Half-Zip POLO Shirt	61	1516.3
Women's Heavyweight Panelled Hoodie	41	1208.0
Women's Pleated Satin Trousers	44	1089.0
Women's Stylish Leather Ankle Boots	51	1079.7
Women's Chic Leather Tote Bag	48	1078.6

```

GROUP BY p.product_name
ORDER BY total_revenue DESC
LIMIT 5;")

# Use a table to present the result
p_revenue %>%
  kbl(digits = 1, caption = "Best Selling Products") %>%
  kable_styling()

```

To set up an appropriate pricing strategy (i.e. the amount of discount), we are interested to know whether the unit price is doing effect on quantity sold.

```

# Execute the SQL query
price_quantity <- RSQLite::dbGetQuery(my_connection, "SELECT p.product_name,
                                                    p.product_price * (1 - d.discount_percentage) AS
                                                    unit_price,
                                                    SUM(o.order_quantity) AS quantity
                                                    FROM `order` o
                                                    JOIN product p ON o.product_id = p.product_id
                                                    JOIN discount d ON p.discount_id = d.discount_id
                                                    GROUP BY p.product_name
                                                    ORDER BY quantity DESC")

# Build the regression model
m.quantity.by.price <- lm(quantity~unit_price, data=price_quantity)

# Plot the model
g5 <- ggplot(price_quantity, aes(x = unit_price, y = quantity)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Linear Regression: Quantity vs Unit Price",
       x = "Unit Price",

```

```
y = "Quantity")
print(g5)
```



```
ggsave(plot = g5, filename = "graphs/unitprice_to_quantity.jpeg", width = 10, height=8,dpi=300)
```

#### 4.6: Customer Review Analysis

We are interested to know the impact of customer perception, analysing the effect of product rating on unit sold provides valuable insights into customer preferences and brand perception. The result can be leverage to enhance our product quality.

```
# Execute the SQL query
rate_quantity <-RSQLite::dbGetQuery(my_connection,"SELECT p.product_name,
                                         AVG(o.product_rating) AS avg_rating,
                                         SUM(o.order_quantity) AS quantity
                                         FROM 'order' o
                                         JOIN product p ON o.product_id = p.product_id
                                         GROUP BY p.product_name
                                         ORDER BY quantity DESC")

# Build the regression model
```

```

m.quantity.by.rate <- lm(quantity~avg_rating, data=rate_quantity)

# Plot the model
g6 <- ggplot(rate_quantity, aes(x = avg_rating, y = quantity)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Linear Regression: Quantity vs Avg Rating",
       x = "Avg Rating",
       y = "Quantity")

print(g6)

```



```

ggsave(plot = g6, filename = "graphs/rating_to_quantity.jpeg", width = 10, height=8,dpi=300)

```

The analysis of the effect of order completion duration to product rating targeting to propose strategies that enhance customer experience. We are interested to know how customer satisfaction correlates with the efficiency of order fulfilment.

```

# Execute the SQL query
rate_day <-RSQLite::dbGetQuery(my_connection,"SELECT AVG(o.product_rating) AS rating,
                                     AVG(julianday(s.delivered_date) -
                                     julianday(o.order_purchased_date)) AS completion_day

```

```

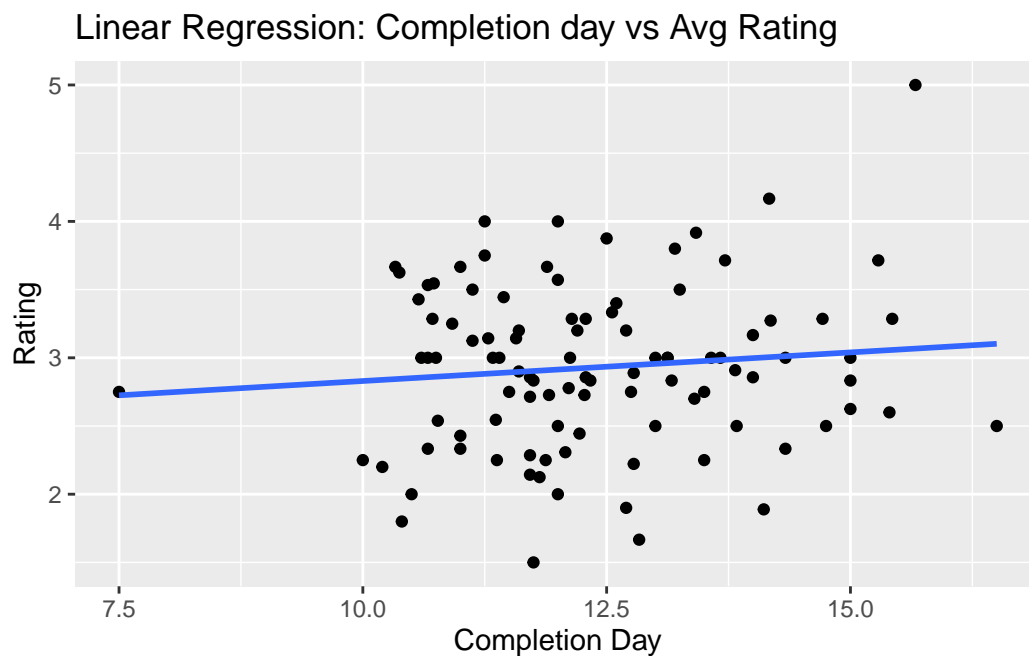
FROM 'order' o
JOIN shipment s ON o.shipping_id = s.shipping_id
JOIN product p ON o.product_id = p.product_id
GROUP BY p.product_name")

# Build the regression model
m.rate.by.day <- lm(completion_day~rating, data=rate_day)

# Plot the model
g7 <- ggplot(rate_day, aes(x = completion_day, y = rating)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Linear Regression: Completion day vs Avg Rating",
       x = "Completion Day",
       y = "Rating")

print(g7)

```



```

ggsave(plot = g7, filename = "graphs/day_to_rate.jpeg", width = 10, height=8,dpi=300)

```

## 4.7: Supplier Analysis

The following section evaluate the performance for our supplier in terms of 3 aspects: average rating, total order quantity and total revenue of the products supplied.

Higher average product rating suggests that customers are generally pleased with the quality of products offered by the suppliers which may enjoy a competitive advantage in attracting customers. Suppliers with larger total order quantities sold indicates products that are in high demand among customers providing general insights into market trends and customer preferences. Suppliers with higher total revenue demonstrate higher sales volumes and generate more revenue for our company. It helps us to identify opportunities for growth and improvement.

```
# Connect to the SQLite database
conn <- dbConnect(RSQLite::SQLite(), dbname = 'project_dm.db')

# Execute the SQL query
query <- "
    SELECT s.supplier_name,
           SUM(o.order_quantity) AS total_order_quantity,
           SUM(o.order_quantity * ((p.product_price * (1 - d.discount_percentage)) - p.product_price)) AS total_revenue,
           COUNT(o.product_rating) AS total_reviews,
           AVG(o.product_rating) AS average_rating
FROM \"order\" o
JOIN product p ON o.product_id = p.product_id
LEFT JOIN discount d ON p.discount_id = d.discount_id
JOIN supplier s ON p.supplier_id = s.supplier_id
WHERE o.payment_status = 'complete'
GROUP BY s.supplier_name
ORDER BY total_order_quantity DESC;
"
supplier_performance <- dbGetQuery(conn, query)

# Close the database connection
dbDisconnect(conn)

# Top 5 suppliers based on rating
top_rating <- supplier_performance[order(-supplier_performance$average_rating), ][1:5, ]

# Top 5 suppliers based on total order quantity
top_order_quantity <- supplier_performance[order(-supplier_performance$total_order_quantity), ][1:5, ]

# Top 5 suppliers based on total sales
top_revenue <- supplier_performance[order(-supplier_performance$total_revenue), ][1:5, ]
```

```

# Create plots
plot_top_rating <- ggplot(top_rating, aes(x = reorder(supplier_name, -average_rating), y = a
  geom_bar(stat = "identity", fill = "salmon") +
  labs(title = "Top 5 Rating",
        x = "Supplier",
        y = "Average Rating") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

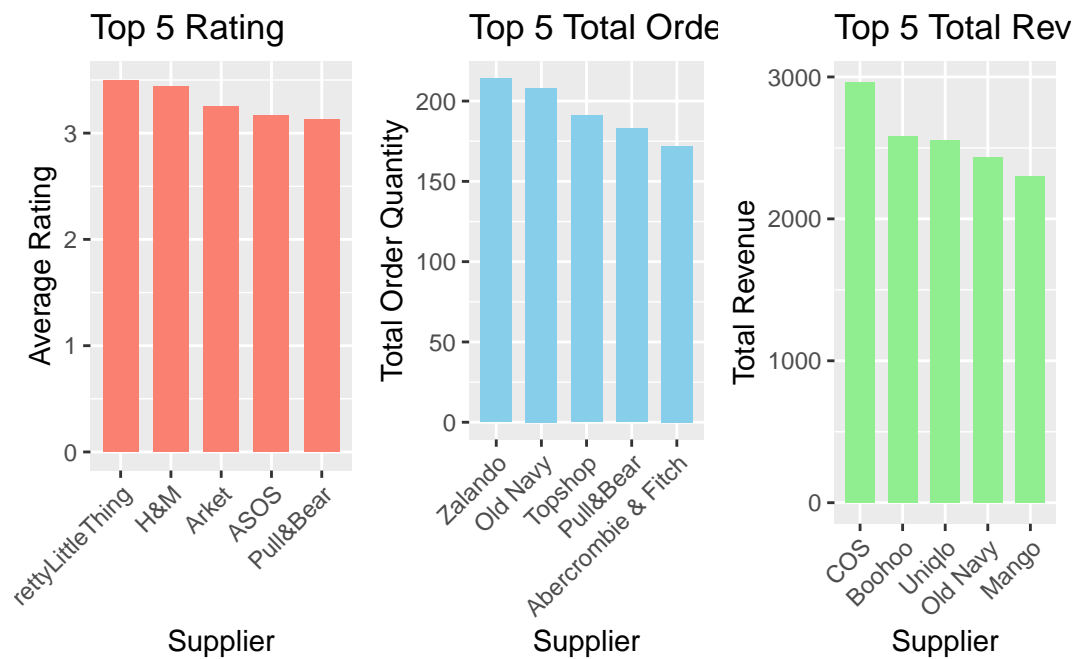
plot_top_order_quantity <- ggplot(top_order_quantity, aes(x = reorder(supplier_name, -total_
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(title = "Top 5 Total Order Quantity",
        x = "Supplier",
        y = "Total Order Quantity") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

plot_top_revenue <- ggplot(top_revenue, aes(x = reorder(supplier_name, -total_revenue), y = t
  geom_bar(stat = "identity", fill = "lightgreen") +
  labs(title = "Top 5 Total Revenue",
        x = "Supplier",
        y = "Total Revenue") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

# Plot the graphs
g8 <- grid.arrange(plot_top_rating, plot_top_order_quantity, plot_top_revenue, nrow = 1)

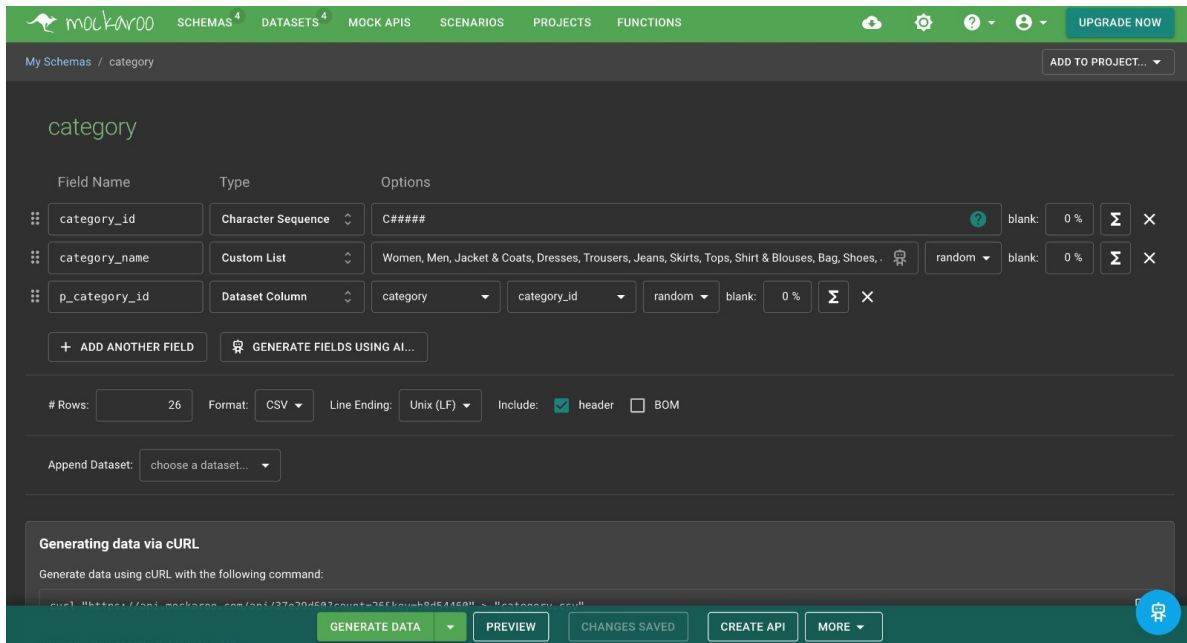
```





```
ggsave(plot = g8, filename = "graphs/supplier.jpeg", width = 10, height=6,dpi=300)
```

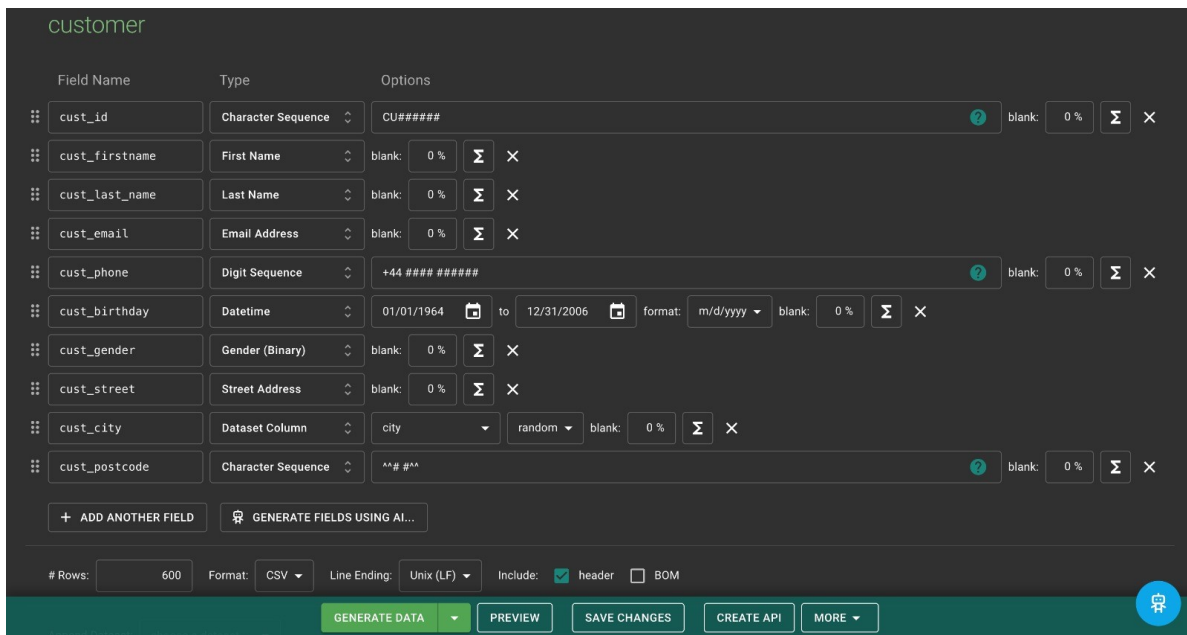
## Appendix



The image shows the Mockaroo 'category' schema configuration interface. The top navigation bar includes 'SCHEMAS', 'DATASETS', 'MOCK APIS', 'SCENARIOS', 'PROJECTS', and 'FUNCTIONS'. The main header displays 'My Schemas / category' and an 'ADD TO PROJECT...' button. The schema name 'category' is shown in green. Below this is a table with columns 'Field Name', 'Type', and 'Options'. The table contains three fields: 'category\_id' (Character Sequence, C####), 'category\_name' (Custom List, Women, Men, Jacket & Coats, Dresses, Trousers, Jeans, Skirts, Tops, Shirt & Blouses, Bag, Shoes, .), and 'p\_category\_id' (Dataset Column, category, category\_id, random). Each field has a 'blank' percentage (0%), a 'Σ' icon, and an 'X' icon. Below the table are buttons for '+ ADD ANOTHER FIELD' and 'GENERATE FIELDS USING AI...'. The bottom section includes '# Rows: 26', 'Format: CSV', 'Line Ending: Unix (LF)', 'Include: ☒ header ☐ BOM', and 'Append Dataset: choose a dataset...'. A section titled 'Generating data via cURL' shows a cURL command. The bottom bar contains 'GENERATE DATA', 'PREVIEW', 'CHANGES SAVED', 'CREATE API', and 'MORE' buttons, along with a blue circular icon.

Field Name	Type	Options
category_id	Character Sequence	C####
category_name	Custom List	Women, Men, Jacket & Coats, Dresses, Trousers, Jeans, Skirts, Tops, Shirt & Blouses, Bag, Shoes, .
p_category_id	Dataset Column	category, category_id, random

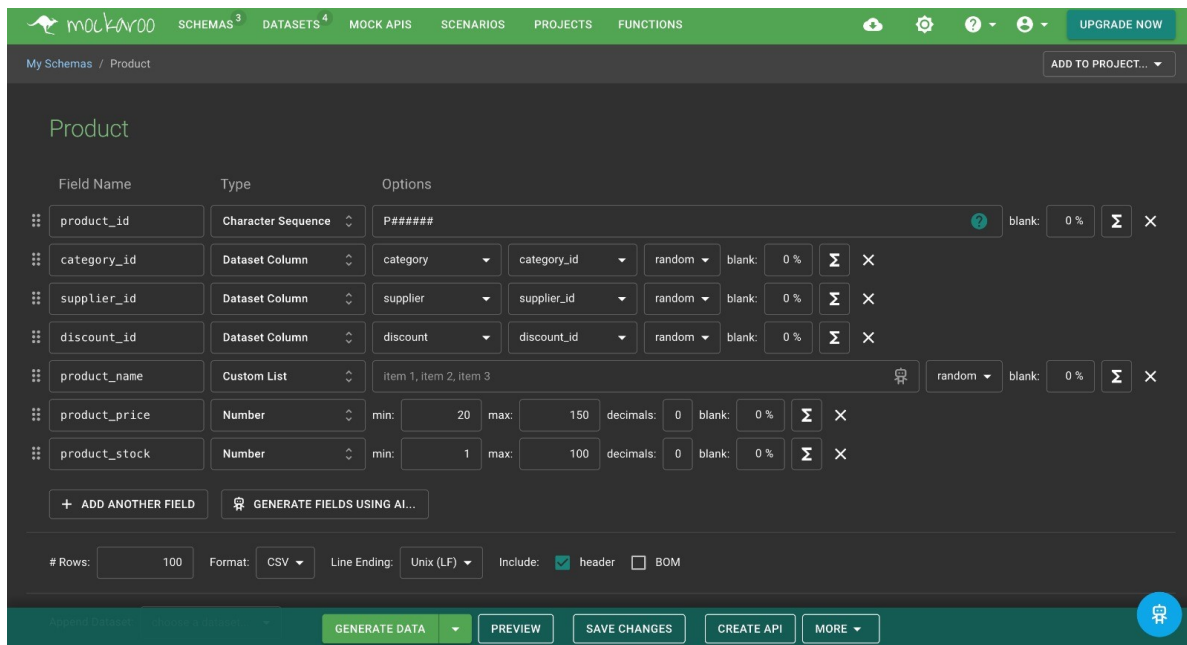
Figure 11: Mockaroo - Category



The image shows the Mockaroo 'customer' schema configuration interface. The top navigation bar includes 'SCHEMAS', 'DATASETS', 'MOCK APIS', 'SCENARIOS', 'PROJECTS', and 'FUNCTIONS'. The main header displays 'My Schemas / customer' and an 'ADD TO PROJECT...' button. The schema name 'customer' is shown in green. Below this is a table with columns 'Field Name', 'Type', and 'Options'. The table contains ten fields: 'cust\_id' (Character Sequence, CU#####), 'cust\_firstname' (First Name), 'cust\_lastname' (Last Name), 'cust\_email' (Email Address), 'cust\_phone' (Digit Sequence, +44 #####), 'cust\_birthday' (Datetime, 01/01/1964 to 12/31/2006, format: m/d/yyyy), 'cust\_gender' (Gender (Binary)), 'cust\_street' (Street Address), 'cust\_city' (Dataset Column, city, random), and 'cust\_postcode' (Character Sequence, ### ##). Each field has a 'blank' percentage (0%), a 'Σ' icon, and an 'X' icon. Below the table are buttons for '+ ADD ANOTHER FIELD' and 'GENERATE FIELDS USING AI...'. The bottom section includes '# Rows: 600', 'Format: CSV', 'Line Ending: Unix (LF)', 'Include: ☒ header ☐ BOM', and 'Append Dataset: choose a dataset...'. The bottom bar contains 'GENERATE DATA', 'PREVIEW', 'SAVE CHANGES', 'CREATE API', and 'MORE' buttons, along with a blue circular icon.

Field Name	Type	Options
cust_id	Character Sequence	CU#####
cust_firstname	First Name	
cust_lastname	Last Name	
cust_email	Email Address	
cust_phone	Digit Sequence	+44 #####
cust_birthday	Datetime	01/01/1964 to 12/31/2006, format: m/d/yyyy
cust_gender	Gender (Binary)	
cust_street	Street Address	
cust_city	Dataset Column	city, random
cust_postcode	Character Sequence	### ##

Figure 12: Mockaroo - Customer

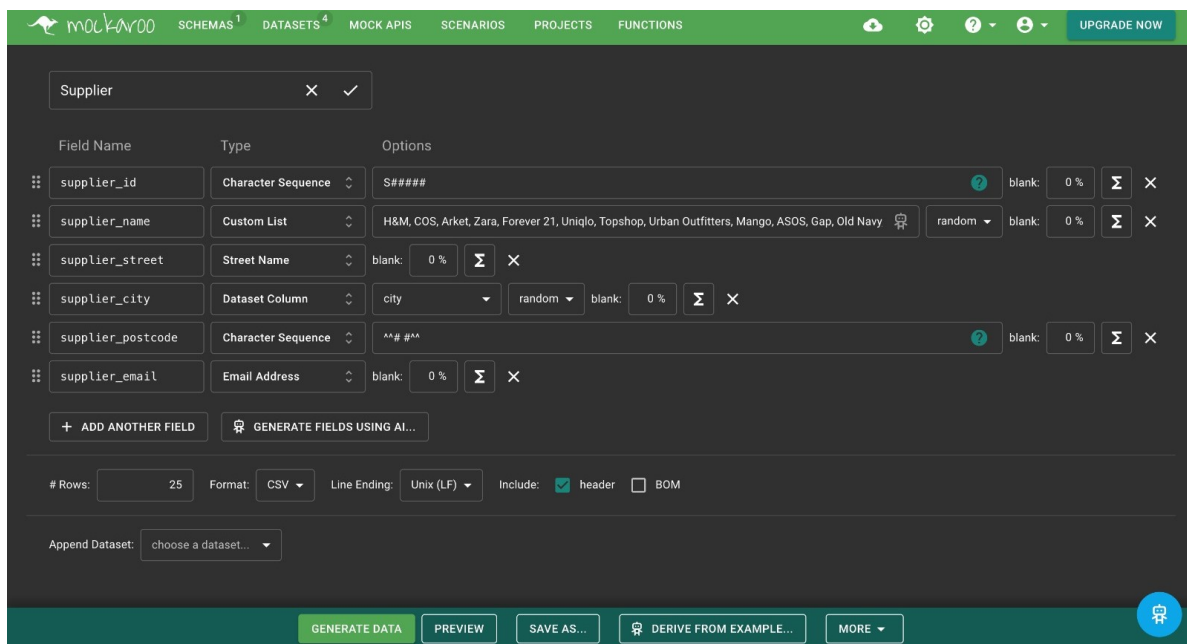


The image shows the Mockaroo interface for configuring a 'Product' schema. The top navigation bar includes 'SCHEMAS', 'DATASETS', 'MOCK APIS', 'SCENARIOS', 'PROJECTS', and 'FUNCTIONS'. The main header shows 'My Schemas / Product' and an 'ADD TO PROJECT...' button. The schema configuration table is as follows:

Field Name	Type	Options
product_id	Character Sequence	P#####
category_id	Dataset Column	category, category_id, random
supplier_id	Dataset Column	supplier, supplier_id, random
discount_id	Dataset Column	discount, discount_id, random
product_name	Custom List	item 1, item 2, item 3
product_price	Number	min: 20, max: 150, decimals: 0
product_stock	Number	min: 1, max: 100, decimals: 0

Below the table, there are buttons for '+ ADD ANOTHER FIELD' and 'GENERATE FIELDS USING AI...'. At the bottom, there are settings for '# Rows: 100', 'Format: CSV', 'Line Ending: Unix (LF)', 'Include: header', and 'BOM'. The bottom bar contains 'GENERATE DATA', 'PREVIEW', 'SAVE CHANGES', 'CREATE API', and 'MORE' buttons.

Figure 13: Mockaroo - Product



The image shows the Mockaroo interface for configuring a 'Supplier' schema. The top navigation bar is the same as in Figure 13. The main header shows 'My Schemas / Supplier' and an 'ADD TO PROJECT...' button. The schema configuration table is as follows:

Field Name	Type	Options
supplier_id	Character Sequence	S#####
supplier_name	Custom List	H&M, COS, Arket, Zara, Forever 21, Uniqlo, Topshop, Urban Outfitters, Mango, ASOS, Gap, Old Navy
supplier_street	Street Name	blank: 0%
supplier_city	Dataset Column	city, random
supplier_postcode	Character Sequence	## ###
supplier_email	Email Address	blank: 0%

Below the table, there are buttons for '+ ADD ANOTHER FIELD' and 'GENERATE FIELDS USING AI...'. At the bottom, there are settings for '# Rows: 25', 'Format: CSV', 'Line Ending: Unix (LF)', 'Include: header', and 'BOM'. The bottom bar contains 'GENERATE DATA', 'PREVIEW', 'SAVE AS...', 'DERIVE FROM EXAMPLE...', and 'MORE' buttons.

Figure 14: Mockaroo - Supplier