Table of contents

1 Database Design and Implementation

1.1 E-R Diagram Design

1.1.1 E-R Diagram

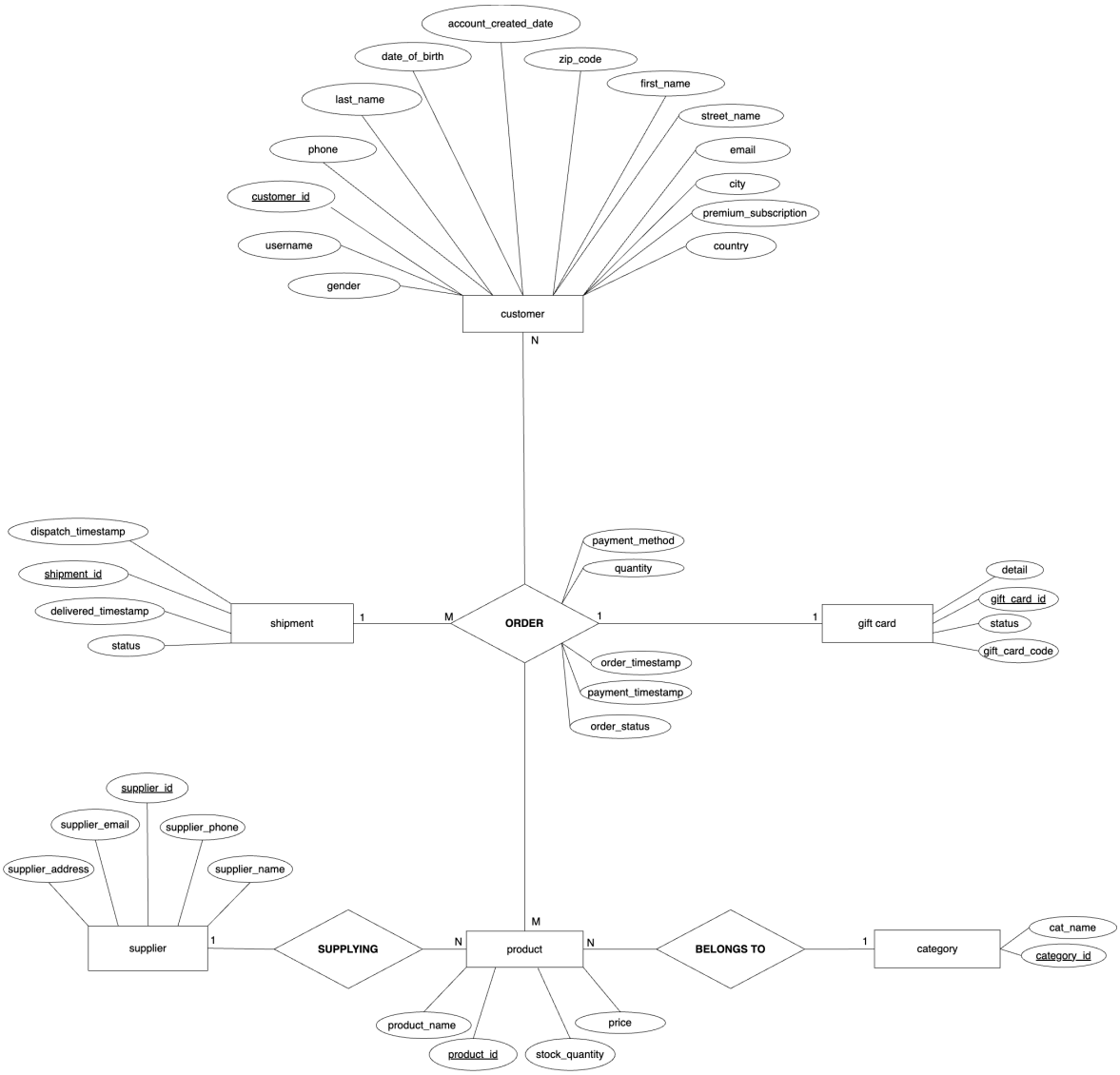


Figure 1: E-R Diagram

1.1.2 Logical Shema and Relationship Sets

Customers (customer_id,first_name,last_name,username,gender,date_of_birth, email, phone, street_name, city, country,zip_code, account_created_date, premium_subscription)

Product_category(category_id, cat_name)

Suppliers(supplier_id, supplier_name, supplier_address, supplier_phone, supplier_email)

Products(product_id, product_name, price, stock_quantity, category_id, supplier_id)

Gift_card(gift_card_id, gift_card_code,detail,status)

Orders(order_id, customer_id, product_id, gift_card_id ,payment_method,quantity,order_timestamp,payment_timestamp,order_status, shipment_id)

Shipment(shipment_id,dispatch_timestamp,delivered_timestamp,status)

Figure 2: Logical Schema

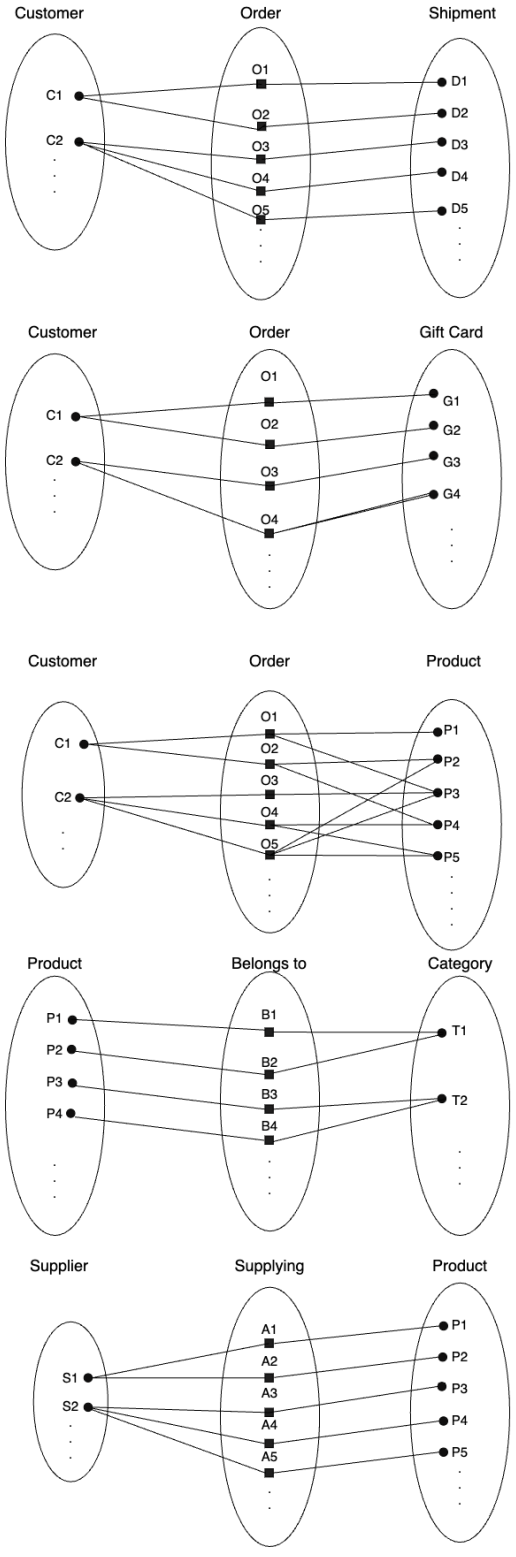


Figure 3: Relationship Sets

1.1.3 Cardinalities:

- A customer can have multiple orders (**1:N** relationship between **CUSTOMERS** and **ORDERS**).
- We assume that a customer can order only one type of product.
- A product belongs to only one category, but a category can have multiple products (**1:N** relationship between **PRODUCT_CATERGORY** and **PRODUCTS**).
- A supplier can supply many products (**1:N** relationship between **SUPPLIERS** and **PRODUCTS**).
- An order results in one shipment but one shipment can contain multiple orders(**1:M** relationship between **ORDERS** and **SHIPMENTS**).
- A gift card can be associated with only one order (**1:1** relationship between **GIFT_CARDS** and **ORDERS**).

1.1.4 Assumptions:

- Every Order must have a Customer, but a Customer does not necessarily need to have an Order.
- If a customer buys several things on the same day, and if all those things are coming from the same place, they'll be packed together and sent off with one tracking number.
- Our e-commerce business operates directly with suppliers, and we do not have any storage facilities for inventory.
- Every customer, product category, supplier, product, gift card, order, and shipment is uniquely identified by their respective ID fields (**customer_id, category_id, supplier_id, product_id, gift_card_id, order_id, shipment_id**).
- Orders reference **CUSTOMERS, PRODUCTS, SHIPMENT,** and **GIFT_CARD** through their respective ID fields, establishing a connection to existing records in those tables.
- Products reference **PRODUCT_CATEGORY** and **SUPPLIERS** through **category_id** and **supplier_id**, ensuring that each product is linked to existing categories and suppliers.
- **Mandatory Information**:
  - Customers must have a **customer_id, first_name, and date_of_birth**.
  - Products must have a **product_id, stock_quantity, category_id, and supplier_id**.
  - Orders must have an **order_id and order_status**.

  - Shipments must have a **shipment_id and status**.

- **Nullable Fields**: Some fields are optional, such as last_name for customers, which suggests that not all information is required to create a record in the database.
- **Data Type Restrictions**: Email and phone fields for customers and supplier_email for suppliers are unique, implying that no two records can have the same value for these fields.
- **premium_subscription** in **CUSTOMERS** is an integer, which is indicated using a **boolean** value (0 or 1)
- The price in **PRODUCTS** is of type **REAL**, allowing for decimal values.
- **Gift Cards**: Gift cards are considered an entity but might not be required for an order, as the gift_card_id in the ORDERS table can be null.
- **Shipment Process**: The **SHIPMENT** table's **dispatch_timestamp** and **delivered_timestamp** suggest tracking the timeline of a shipment but they're not set as NOT NULL, so there might be cases where a shipment is created in the system before an actual dispatch time is known.
- **Payment and Order Timing**: Orders have both an **order_timestamp** and a **payment_timestamp**, which may not always be the same—this allows tracking the time the order was made and when the payment was processed.
- **Stock Management**: stock_quantity in PRODUCTS suggests the system tracks inventory levels, but there is no direct link to orders for decrementing stock, which implies this might be managed by a separate process or system.
- **Data Consistency**: The use of foreign keys enforces data consistency, ensuring that records in linked tables must exist before they can be referenced in an association.

1.2 SQL Database Schema Creation

This code is like organizing a digital warehouse for an e-commerce business. It sets up different sections in a database for storing information about customers, product categories, suppliers, products, gift cards, orders, and shipments. Each section is designed to keep specific types of information, ensuring that everything from customer details to order and shipment records is neatly organized and interconnected. The code ensures that each item, whether a customer or a product, is unique and correctly linked to related information, like linking a product to its supplier. It's like setting up shelves and labeling them in a warehouse to ensure everything is easy to find and in the right place.

```r
#connect to the SQLite database
my_connection <- RSQLite::dbConnect(RSQLite::SQLite(),
                                    "../database/ecommerce_database_v1.db")


dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS CUSTOMERS
                (
                    customer_id VARCHAR(255) NOT NULL PRIMARY KEY,
                    first_name VARCHAR(255) NOT NULL,
                    last_name VARCHAR(255),
                    username VARCHAR(255),
                    gender TEXT,
                    date_of_birth DATE NOT NULL,
                    email VARCHAR(255) UNIQUE,
                    phone VARCHAR(20) UNIQUE,
                    street_name VARCHAR(255),
                    city VARCHAR(255),
                    country VARCHAR(255),
                    zip_code VARCHAR(20),
                    account_created_date TIMESTAMP,
                    premium_subscription INTEGER
                );"
          )

dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS PRODUCT_CATEGORY
                (
                    category_id VARCHAR(255) NOT NULL PRIMARY KEY,
                    cat_name VARCHAR(255)
                );"
          )

dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS SUPPLIERS
                (
                    supplier_id VARCHAR(255) NOT NULL PRIMARY KEY,
                    supplier_name VARCHAR(255),
                    supplier_address VARCHAR(500),
                    supplier_phone VARCHAR(20),
                    supplier_email VARCHAR(255) UNIQUE
                );"
```

```
          )

dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS PRODUCTS
                (
                    product_id VARCHAR(255) NOT NULL PRIMARY KEY,
                    product_name VARCHAR(255),
                    price REAL,
                    stock_quantity INTEGER NOT NULL,
                    category_id VARCHAR(255) NOT NULL,
                    supplier_id VARCHAR(255) NOT NULL,
                    FOREIGN KEY(category_id) REFERENCES
                          PRODUCT_CATEGORY(category_id),
                    FOREIGN KEY(supplier_id) REFERENCES SUPPLIERS(supplier_id)
                );"
          )

dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS GIFT_CARD
                (
                gift_card_id VARCHAR(50) NOT NULL PRIMARY KEY,
                gift_card_code VARCHAR(50),
                detail INTEGER,
                status VARCHAR(50)
                );"
          )
dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS ORDERS
                (
                    order_id VARCHAR(255) NOT NULL PRIMARY KEY,
                    customer_id VARCHAR(255),
                    product_id VARCHAR(255),
                    gift_card_id VARCHAR(255),
                    payment_method TEXT,
                    quantity INTEGER,
                    order_timestamp TIMESTAMP,
                    payment_timestamp TIMESTAMP,
                    order_status VARCHAR(50) NOT NULL,
                    shipment_id VARCHAR(255),
                    FOREIGN KEY(customer_id) REFERENCES CUSTOMERS(customer_id),
                    FOREIGN KEY(product_id) REFERENCES PRODUCTS(product_id),
                    FOREIGN KEY(shipment_id) REFERENCES SHIPMENT(shipment_id),
```

```
                    FOREIGN KEY(gift_card_id) REFERENCES GIFT_CARD(gift_card_id)
                );"
            )
dbExecute(my_connection,
                "CREATE TABLE IF NOT EXISTS SHIPMENT
                (
                shipment_id VARCHAR(255) NOT NULL PRIMARY KEY,
                dispatch_timestamp DATETIME,
                delivered_timestamp DATETIME,
                status VARCHAR(50) NOT NULL
                );"
            )

#Check if the tables are created

dbGetQuery(my_connection,
    sprintf("SELECT name FROM sqlite_master WHERE type='table';")
  )
```

2 Data Generation and Management

2.1 Synthetic Data Generation

This code simulates part of an e-commerce operation by creating a set of mock orders based on existing data for customers, products, suppliers, and gift cards. It goes through these steps:

1. Finding Data: It locates and organizes data files for different categories like customers and products.
2. Loading Data: The first file from each category is loaded to form datasets for analysis.
3. Sampling: A subset of products and customers is randomly selected to make the data more manageable.
4. Order Creation: It generates fake orders, pairing products with customers and applying gift cards as needed, while recording details like payment method and order status.
5. Linking Suppliers: Each order is linked to the corresponding supplier based on the product information.
6. Shipment Grouping: Unique shipment IDs are created for orders, grouping them logically for shipping.
7. Data Cleaning: The orders are refined for realism, like removing shipment details from canceled orders.
8. Gift Card Update: The status of gift cards used in the orders is updated to 'USED'.

Essentially, this script creates a realistic snapshot of transactions for analysis, reflecting various aspects of e-commerce activities.

```
## Find all files matching the pattern
customer_files <- list.files(path = "../datasets"
                             ,pattern = "CUSTOMERS.*\\.csv$",full.names = TRUE )
category_files <- list.files(path = "../datasets"
                             ,pattern = "CATEGORY.*\\.csv$",full.names = TRUE )
gift_card_files <- list.files(path = "../datasets"
                              ,pattern = "GIFT_CARDS.*\\.csv$",full.names = TRUE )
suppliers_files <- list.files(path = "../datasets"
                              ,pattern = "SUPPLIERS.*\\.csv$",full.names = TRUE )
products_files <- list.files(path = "../datasets"
                             ,pattern = "PRODUCTS.*\\.csv$",full.names = TRUE )

customers_df <- readr::read_csv(customer_files[1])
gift_card_df <- readr::read_csv(gift_card_files[1])
suppliers_df <- readr::read_csv(suppliers_files[1])
category_df <- readr::read_csv(category_files[1])
products_df <- readr::read_csv(products_files[1])



#Sample Customers

sample_size <- floor(0.2 * nrow(products_df))
sampled_product_ids <- sample(products_df$product_id
                              , size = sample_size, replace = FALSE)
sampled_products_df <- products_df[products_df$product_id
                                   %in% sampled_product_ids, ]


#Sample Products

sample_size <- floor(0.2 * nrow(customers_df))
sampled_customer_ids <- sample(customers_df$customer_id
                               , size = sample_size, replace = FALSE)
sampled_customers_df <- customers_df[customers_df$customer_id
                                     %in% sampled_customer_ids, ]


generate_orders_data <- function(n = 1000) {
  set.seed(123)
```

```r
  orders_df <- tibble(
    order_id = sprintf("%s-%04d", "ORD", 1:n),
    customer_id = sample(sampled_customers_df$customer_id, n, replace = TRUE),
    product_id = sample(sampled_products_df$product_id, n, replace = TRUE),
    gift_card_id = sample(c(NA, gift_card_df$gift_card_id), n, replace = TRUE)
    , # Assuming gift cards are used as discounts
    payment_method = sample(c("Credit Card", "Debit Card"
                              , "PayPal", "Gift Card"), n, replace = TRUE),
    quantity = sample(1:5, n, replace = TRUE),
    order_timestamp = sample(seq(as.POSIXct('2024/02/01')
                                 , as.POSIXct('2024/02/29'), by="day")
                             , n, replace = TRUE),
    payment_timestamp = order_timestamp + hours(sample(1:72, n, replace = TRUE)),
    order_status = sample(c("Processing", "Shipped", "Delivered"
                            , "Cancelled","Pending Payment","Out for Delivery")
                          , n, replace = TRUE),
  )

  # Augment the orders data frame with supplier_id using left_join
  orders_df <- orders_df %>%
    left_join(sampled_products_df %>% select(product_id, supplier_id)
              , by = "product_id") %>%
    select(order_id, customer_id, product_id, gift_card_id, payment_method
           , quantity, order_timestamp, payment_timestamp
           , order_status, supplier_id)

  return(orders_df)
}

# Generate orders data
orders_df <- generate_orders_data(n = 1000)

generate_shipment_ids <- function(df) {
  # Create a unique identifier for each group
  df <- df %>%
    mutate(date_only = as.Date(order_timestamp)) %>%
    group_by(customer_id, supplier_id, date_only) %>%
    mutate(shipment_group_id = cur_group_id()) %>%
    ungroup() %>%
    mutate(shipment_id = sprintf("SHIP%05d", shipment_group_id)) %>%
    select(-shipment_group_id, -date_only) # Clean up the extra columns
```

```
  df
}

# Apply the function to your data frame
orders_df <- generate_shipment_ids(orders_df)

# Optional: Adjusting for logical consistency (e.g.,
# cancelled orders should not have a shipment_id)
  orders_df <- orders_df %>%
    mutate(shipment_id = if_else(order_status %in%
                                   c("Cancelled","Pending Payment")
                                 , NA_character_,as.character(shipment_id)),
           payment_method = if_else(order_status == "Pending Payment"
                                    ,NA_character_,payment_method),
           gift_card_id = if_else(payment_method == "Gift Card",
                                  gift_card_id,NA_character_)) %>%
    mutate(supplier_id = NULL)

used_gift_cards <- unique(na.omit(orders_df$gift_card_id))
gift_card_df$status[gift_card_df$gift_card_id %in% used_gift_cards] <- 'USED'
```

This code transforms order data into shipment information by doing the following:

1. It sets a dispatch date for each order to either the day the order was made or the next day.
2. It then assigns a delivery date to each order, ensuring it's 2 to 14 days after the dispatch date.
3. Orders are given a status based on their current phase, like "Ready for Dispatch" if they're being processed, or "In Transit" if they've been shipped.
4. The code cleans up the data by keeping only shipment-related details, removing duplicates and any incomplete records.
5. Lastly, it updates the dispatch and delivery dates based on the shipment status, for example, clearing the delivery date for orders "In Transit."

The updated shipment details are saved, providing a clear snapshot of when orders are dispatched, expected delivery times, and their current status.

```
#Shipment Table

  shipment_df <- orders_df %>%
    mutate(
      # Dispatch date could be the same as the order date or a day after
```

```
    dispatch_timestamp = order_timestamp + days(sample(0:1, n()
                                                       , replace = TRUE)),

    # Delivered date should be after the dispatch date;
    #here I assume delivery takes between 2 to 5 days
    delivered_timestamp = dispatch_timestamp + days(sample(2:14, n()
                                                           , replace = TRUE)),

    # Randomly assign a delivery status
    status = if_else(order_status == "Processing","Ready for Dispatch"
                     ,if_else(order_status == "Shipped","In Transit"
                    ,if_else(order_status == "Out for Delivery",order_status
                   ,if_else(order_status == "Delivered",order_status,"NA"))))
  ) %>%
  # Select only the relevant columns for the shipment table
  select(shipment_id, dispatch_timestamp, delivered_timestamp, status) %>%
  # Remove duplicate rows to ensure unique shipments
  distinct()

shipment_df <- na.omit(shipment_df)


shipment_df <- shipment_df %>%
  mutate(
    # Assign NA to dispatch_timestamp if status is 'Ready for Dispatch'
    dispatch_timestamp = if_else(status == "Ready for Dispatch"
                                 , NA_Date_, dispatch_timestamp),
    delivered_timestamp = if_else(status == "Ready for Dispatch"
                                  , NA_Date_, delivered_timestamp),

    # 'In Transit' status should have a dispatch date but no delivery date
    dispatch_timestamp = if_else(status == "In Transit"
                      , Sys.Date() - days(sample(1:5, 1)), dispatch_timestamp),
    delivered_timestamp = if_else(status == "In Transit"
                      , NA_Date_, delivered_timestamp),

    # 'In Transit' status should have a dispatch date but no delivery date
    dispatch_timestamp = if_else(status == "Out for Delivery"
                    , Sys.Date() - days(sample(1:5, 1)), dispatch_timestamp),
    delivered_timestamp = if_else(status == "Out for Delivery"
                    , NA_Date_, delivered_timestamp),
```

```
      # If status is 'Delivered', both dates should be in the past,
      #with delivered after dispatched
      dispatch_timestamp = if_else(status == "Delivered" &
                                     is.na(dispatch_timestamp)
                     , Sys.Date() - days(sample(6:10, 1)), dispatch_timestamp),
      delivered_timestamp = if_else(status == "Delivered"
              , dispatch_timestamp + days(sample(1:5, 1)), delivered_timestamp)
    )

write_csv(orders_df,"../datasets/ORDERS.csv")

write_csv(shipment_df,"../datasets/SHIPMENTS.csv")
```

2.2 Data Import and Quality Assurance

This code is like a checklist for a data table, ensuring all the needed items (columns) are there. If anything's missing, it alerts you; otherwise, it confirms everything is in order.

```
check_column_match <- function(df, expected_cols) {
  # Check if all expected columns are present in the dataframe
  missing_cols <- setdiff(expected_cols, names(df))

  if (length(missing_cols) > 0) {
    stop(sprintf("The dataframe is missing these columns expected
                                 by the SQL table: %s",
                 paste(missing_cols, collapse = ", ")))
  }

  return(TRUE)
}
```

This code is like a checklist for a data table, ensuring all the needed items (columns) are there. If anything's missing, it alerts you; otherwise, it confirms everything is in order.

2.3 Data Import and Quality Assurance

CUSTOMERS

This code is like a gatekeeper for customer data entering a digital database:

1. Connection: It links up with the database to start processing customer information.

2. Checklist: It verifies that each piece of incoming data has all the required fields, like name, contact info, and other personal details.
3. Validation: It ensures emails and genders are in the right format and that essential data isn't missing.
4. Updating: It adds new customer details to the database, avoiding duplicates and ensuring data is accurately recorded.
5. Wrap-Up: After processing, it closes the connection to the database to secure the data.

This is done for each batch of customer data, keeping the database current and correct.

```
ingest_customer_data <- function(df) {

  my_connection <- RSQLite::dbConnect(RSQLite::SQLite()
                                      , "../database/ecommerce_database_v1.db")
  # Data validation

  expected_cols <- c("customer_id", "first_name", "last_name", "username",
                     "gender", "date_of_birth", "email", "phone", "street_name",
                     "city", "country", "zip_code", "account_created_date",
                     "premium_subscription")

  if (!check_column_match(df, expected_cols)) return(FALSE)

  #email check
  valid_email <- grepl("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"
                       , df$email)
  df <- df[valid_email, ]

  #gender check
  valid_genders <- c("Male", "Female", "Other")
  df <- df[df$gender %in% valid_genders, ]


  # Data type checks (adjust according to your data frame)
  df$date_of_birth <- as.Date(df$date_of_birth,format = "%d/%m/%y")
  df$account_created_date <- as.Date(df$account_created_date,format = "%d/%m/%y")
  df$premium_subscription <- as.integer(df$premium_subscription)

  # Check for null values in NOT NULL columns
  required_columns <- c("customer_id", "first_name", "date_of_birth")
  df <- df[!rowSums(is.na(df[required_columns])) > 0, ]

  # Insert validated data into the database
```

```
  for(i in 1:nrow(df)){

      #Check for duplicate records based on the primary key
   existing_ids <- dbGetQuery(my_connection,
  sprintf("SELECT customer_id FROM CUSTOMERS WHERE customer_id = '%s'"
          ,   df$customer_id[i]))
    if(nrow(existing_ids) > 0) {
      cat(sprintf("Skipping duplicate entry for customer_id: %s\n"
                  , df$customer_id[i]))
      next
    }

    insert_query <- sprintf("INSERT INTO CUSTOMERS (customer_id, first_name
    , last_name, username, gender, date_of_birth, email, phone, street_name
    , city, country, zip_code, account_created_date, premium_subscription)
    VALUES ('%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s'
    , '%s', '%s', %d)",
    df$customer_id[i], df$first_name[i], df$last_name[i], df$username[i]
    , df$gender[i], df$date_of_birth[i],df$email[i], df$phone[i]
    , df$street_name[i], df$city[i], df$country[i], df$zip_code[i]
    , df$account_created_date[i], df$premium_subscription[i])
    tryCatch({
    dbExecute(my_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))
    })
      }

      # Close the database connection
    dbDisconnect(my_connection)
}

for(file in customer_files) {
  df <- readr::read_csv(file)
  ingest_customer_data(df)

}
```

```
                customer_id first_name last_name    username gender
1 01HQZS38KRC38NFNQR9QF1MTBZ       Poul  Jellings pjellingsdv   Male
  date_of_birth                        email        phone           street_name
```

```
1    1992-12-11 pjellingsdv@reverbnation.com 277-129-0314 3 Stone Corner Street
      city        country zip_code account_created_date premium_subscription
1 Aberdeen United Kingdom     AB39           2023-04-01                     0
```

PRODUCT_CATEGORY

This code is about adding new product categories to an e-commerce database. Here's a simplified breakdown:

1. Connecting to Database: It starts by connecting to the e-commerce database to prepare for adding new information.
2. Checking the List: The code expects each product category data to have two specific pieces of information: a unique category ID and the category name. It checks to make sure this data is present before proceeding.
3. Ensuring Completeness: It makes sure that none of the required details (category ID and name) are missing for any of the categories.
4. Adding Categories: For each category, it first checks if the category ID already exists in the database to avoid duplicates. If the category is new, it adds the category ID and name into the database.
5. Handling Issues: If there's a problem adding a category (like a technical glitch), it will let you know without stopping the whole process.
6. Wrapping Up: Once all the categories from the file have been checked and added, it closes the connection to the database.

This process repeats for each file in a list of category files, ensuring all new product categories are added to the database efficiently and correctly.

```
                  category_id cat_name
1 01HQZSYXN5D9YD5YEVE62CZY5T  Jewelry
```

SUPPLIERS

This code is about adding new supplier information to an e-commerce database. It works like this:

1. Connecting: First, it sets up a connection with the database where supplier information needs to be stored.
2. Checking Requirements: The code expects each supplier's information to include an ID, name, address, phone number, and email. It makes sure these details are present and correctly formatted, especially the email.
3. Ensuring Quality: The code also checks for incomplete records, particularly making sure that each supplier has both an ID and a name.
4. Adding Suppliers: One by one, it attempts to add suppliers to the database. Before adding, it checks to ensure the supplier isn't already in the database to avoid duplicates.

5. Dealing with Problems: If there's an issue while adding a supplier, like a mistake in the data or a technical glitch, the code notes the problem but continues with the rest.
6. Finishing Up: After working through all suppliers in the list, the connection to the database is closed.

This process is repeated for each file that contains a list of suppliers, making sure all new supplier data is added systematically and correctly to the database.

GIFT CARDS

This code is about adding new gift card information to an e-commerce database, ensuring each entry is complete and unique. Here's a simpler explanation:

1. Setting Up: It connects to the database where gift card details need to be stored.
2. Checking the Basics: The code looks for specific pieces of information for each gift card: an ID, a code, some details, and its status. If any expected information is missing, it stops the process.
3. Ensuring Completeness: It checks to make sure the essential details (ID, code, and status) aren't missing for any gift card.
4. Detail Adjustment: The code converts the 'detail' section into a numeric format, perhaps to standardize the data.
5. Adding Gift Cards: It goes through the list, adding each gift card to the database. If a gift card with the same ID already exists, it skips adding it to avoid duplication.
6. Handling Errors: If there's a problem adding a gift card (like incorrect data format or a database issue), it notes the error and moves on.
7. Finishing Up: After all the gift cards in the file have been processed, it closes the connection to the database.

This process is repeated for each file in a set of gift card files, ensuring all new gift card data is correctly added to the database.

```
                          gift_card_id gift_card_code detail status
1 3bb1655b-9007-415c-b78a-c6c10a386882    5XT6GQ9XQ72    0.3 UNUSED
```

PRODUCTS

This code is about adding new product information to an e-commerce database. It's like checking and organizing new stock in a store. Here's a simpler breakdown:

1. Connection Setup: The code first connects to the store's database, ready to update the inventory.
2. Checking the List: It ensures each product comes with specific information: an ID, name, price, quantity in stock, category, and supplier. If any info is missing, the process halts.
3. Preparing the Stock: The quantity of each product is confirmed to be a whole number, possibly to avoid errors with partial products.

4. Ensuring Essentials: It checks that vital information (product ID, stock quantity, category, and supplier) isn't missing from any product.
5. Stocking Shelves: For each product, the code checks if that product is already in the system to avoid duplicates. If it's new, the product's details are added to the database.
6. Troubleshooting: If there's an issue adding a product, like incorrect details or a system error, the code notes the issue but keeps going.
7. Closing Time: Once all products in the list have been processed, the database connection is closed.

This process repeats for each file in a set of product files, ensuring all new products are accurately added to the store's database.

```
     product_id              product_name price stock_quantity
1 5116-vjq-2956 Pampers Swaddlers Diapers    25            222
                  category_id                supplier_id
1 01HQZSYXN9NDEKZ0KDTXG7GWAR 01HQZS3CHR3Z0C3RDD0QYFT566
```

ORDER

This code is about processing new orders in an e-commerce system, Here's a simple explanation:

1. Connection Setup: First, it connects to the store's database to start updating the order records.
2. Checking the Order List: The code makes sure each order includes all the necessary information like order ID, customer ID, product ID, and so on. If something's missing, the process stops.
3. Verifying Order Details: It ensures crucial details like the order ID, the quantity of items, and the order status are present and correct for every order.
4. Processing Orders: For each order, it checks if that order already exists in the database to avoid recording it twice. It also checks if the quantity of items is valid (more than zero and a number).
5. Recording Orders: After passing the checks, each order's details are added to the database.
6. Handling Errors: If there's an issue with adding an order (like incorrect data or a technical glitch), the code notes the problem but continues with the next order.
7. Finishing Up: Once all orders have been processed, it disconnects from the database.

This way, each new order is carefully checked and recorded in the system, ensuring the database is up-to-date and accurate.

```
  order_id                customer_id    product_id
1 ORD-0001 01HQZS38YDTF2DBFZMBDXF6WZ6 3672-agb-8683
```

```
                          gift_card_id payment_method quantity order_timestamp
1 3014edd1-7db0-4e6e-b19d-5bc9ff355b9c         PayPal        4      2024-02-01
    payment_timestamp order_status shipment_id
1 2024-02-02 18:00:00      Shipped   SHIP00295
```

SHIPMENTS

This code operates as a data ingestion module for a logistics system, systematically validating and storing each shipment's metadata within the organization's database:

1. Connecting: It links to the database to start processing shipments.
2. Checking: It verifies each shipment has an ID and status before filing.
3. Filing: For each shipment, it checks for duplicates, then files its dispatch and delivery details, along with its status.
4. Handling Issues: If there's a filing error, it's noted, but the process continues.
5. Wrapping Up: After all shipments are filed, it secures the database connection.

This ensures the shipment records are consistently updated and accurate.

```
  shipment_id dispatch_timestamp delivered_timestamp     status
1   SHIP00295         2024-03-14                  NA In Transit
```

2.3.1 Check Referential Integrity

ORDERS Table

These code snippets act as integrity checks for the "ORDERS" table, verifying links to "CUSTOMERS," "PRODUCTS," "GIFT_CARD," and "SHIPMENT" tables:

1. Customer ID Check: Validates that each customer ID in orders corresponds to an entry in the customer table, flagging any discrepancies.
2. Product ID Check: Ensures each product ID in orders matches an item in the product table, highlighting any nonexistent product references.
3. Gift Card ID Check: Confirms gift card IDs in orders exist in the gift card table, identifying any invalid uses.
4. Shipment ID Check: Verifies that each shipment ID in orders is present in the shipment table, detecting any unrecorded or nonexistent shipments.

These checks aim to identify and rectify database inconsistencies, maintaining the accuracy and trustworthiness of the order system.

customer_id check

```
dbGetQuery(my_connection,
           "SELECT
              DISTINCT o.customer_id as customer_id,
              c.customer_id as customer_id,
              first_name ||' '|| last_name as customer_name
           FROM ORDERS as o
           LEFT JOIN CUSTOMERS as c ON c.customer_id = o.customer_id
           WHERE c.customer_id is NULL
           ;")
```

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

product_id check

```
dbGetQuery(my_connection,
           "SELECT
              DISTINCT o.product_id as product_id,
              p.product_id as check_product_id,
              product_name as product_name
           FROM ORDERS as o
           LEFT JOIN PRODUCTS as p ON o.product_id = p.product_id
           WHERE p.product_id is NULL
           ;")
```

```
[1] product_id       check_product_id product_name
<0 rows> (or 0-length row.names)
```

gift_card_id

```
dbGetQuery(my_connection,
           "SELECT
              DISTINCT o.gift_card_id as gif_card_id,
              g.gift_card_id as check_gift_card_id,
              gift_card_code
           FROM ORDERS as o
           LEFT JOIN GIFT_CARD as g ON g.gift_card_id = o.gift_card_id
           WHERE o.gift_card_id is NULL
           ;")
```

```
[1] gif_card_id        check_gift_card_id gift_card_code
<0 rows> (or 0-length row.names)
```

shipment_id

```
dbGetQuery(my_connection,
           "SELECT
              DISTINCT o.shipment_id as shipment_id,
              s.shipment_id as check_shipment_id
           FROM ORDERS as o
           LEFT JOIN SHIPMENT as s ON s.shipment_id = o.shipment_id
           WHERE o.shipment_id is NULL
           ORDER BY o.shipment_id
           ;")
```

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

PRODUCTS

These code snippets validate the "PRODUCTS" table, ensuring products are correctly linked to existing suppliers and categories:

1. Supplier ID Check: Verifies each product's supplier ID against the "SUPPLIERS" table, flagging any mismatches which could indicate incorrect or outdated supplier links.
2. Category ID Check: Confirms each product's category ID with the "PRODUCT_CATEGORY" table, identifying any nonexistent category links, which could suggest mislabeling or missing categories.

These validations are essential for maintaining database integrity and supporting efficient inventory and order management.

supplier_id

```
dbGetQuery(my_connection,
           "SELECT
              DISTINCT p.supplier_id,
              s.supplier_id as check_supplier_id,
              s.supplier_name
           FROM PRODUCTS as p
           LEFT JOIN SUPPLIERS as s ON p.supplier_id = s.supplier_id
           WHERE s.supplier_id is NULL
           ORDER BY p.supplier_id
           ;")
```

```
[1] supplier_id       check_supplier_id supplier_name
<0 rows> (or 0-length row.names)
```

category_id

```
dbGetQuery(my_connection,
           "SELECT
              DISTINCT p.category_id,
              c.category_id as check_catergory_id,
              cat_name
           FROM PRODUCTS as p
           LEFT JOIN PRODUCT_CATEGORY as c ON c.category_id = p.category_id
           WHERE p.category_id is NULL
           ORDER BY p.category_id
           ;")
```

```
[1] category_id        check_catergory_id cat_name
<0 rows> (or 0-length row.names)
```

3 Data Pipeline Generation

3.1 GitHub Repository and Workflow Setup and GitHub Actions for Continuous Integration

- **.github/workflows**: This directory contains definitions for GitHub Actions workflows, which automate schema creation, data generation, validation, insertion and data analysis
- **R**: This directory is where all R scripts and code files are stored.
- **database**: Contains files related to the project's database. These include database files.
- **database_schema**: This contains SQL scripts defining the structure of the database used in the project.
- **datasets**: This directory stores data files that the R scripts would process.

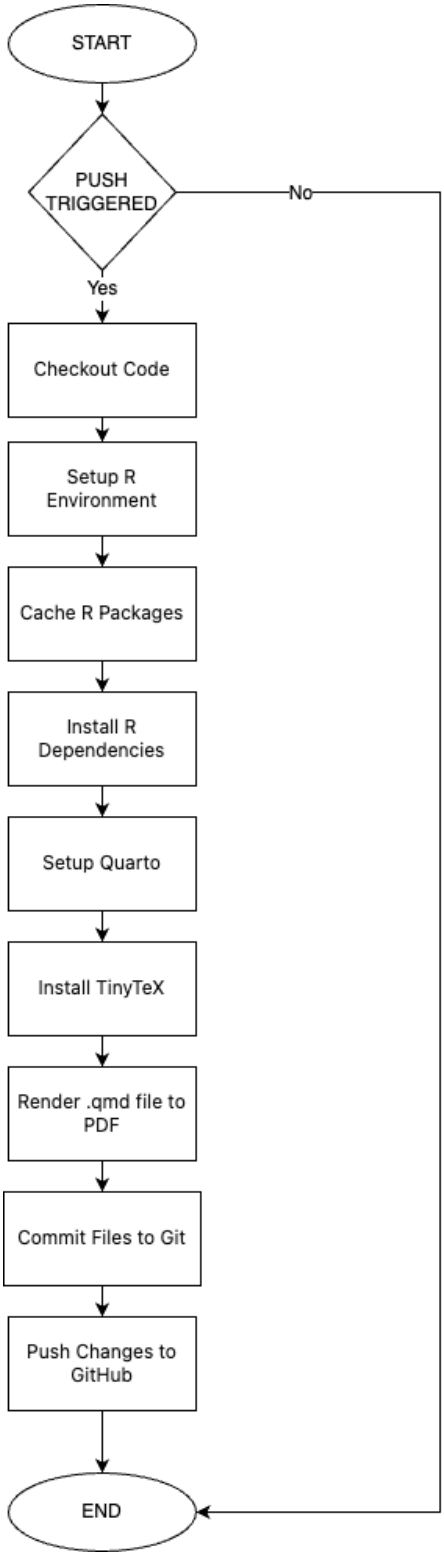


Figure 4: Github Action Workflow

4 Data Analysis and Reporting with Quarto in R

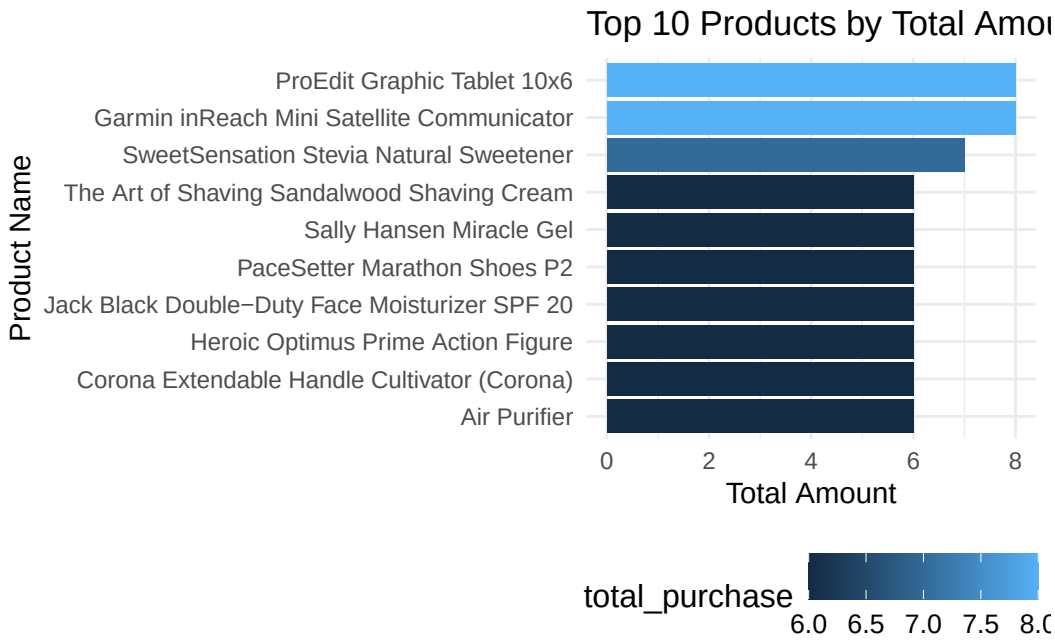
4.1 Advanced Data Analysis in R

4.2 Comprehensive Reporting with Quarto

1. Top 10 Products - Overall (by Quantity)

```
# Define the SQL query
query_1 <- dbGetQuery(my_connection,
           "SELECT
              ORDERS.product_id,
              product_name,
              count(quantity) as total_purchase
           FROM ORDERS
           JOIN PRODUCTS ON ORDERS.product_id = PRODUCTS.product_id
           WHERE lower(order_status) in ('shipped','delivered')
           GROUP BY ORDERS.product_id,product_name
           ORDER BY total_purchase desc
           LIMIT 10
           ;")

# Visualize the result using ggplot2
ggplot(query_1, aes(x = reorder(product_name, total_purchase),
                    y = total_purchase, fill = total_purchase)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  coord_flip() +
  labs(title = "Top 10 Products by Total Amount",
       x = "Product Name",
       y = "Total Amount") +
  theme_minimal() +
  theme(legend.title = element_text(size = 12),
        legend.text = element_text(size = 10),
        legend.position = "bottom")
```



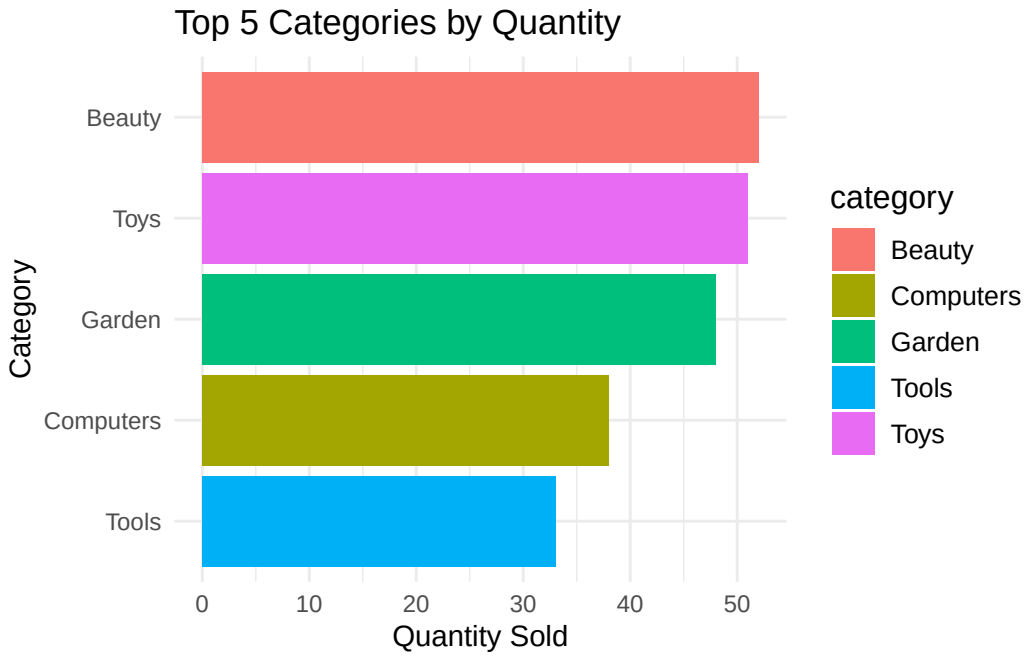
The bar chart shows the top 10 products by total sales amount. Leading the chart, the 'ProEdit Graphic Tablet 10x6' and the 'Garmin in Reach Mini Satellite Communicator' indicate strong sales, suggesting high consumer demand. The diversity of products, including computers, groceries, and other items, reflects varied consumer interests and potential market segments for focus.

2. Top 5 Categories (by Quantity)

```
# SQL query to fetch top 5 categories by quantity
query_2 <- dbGetQuery(my_connection,
           "SELECT
              cat_name as category,
              count(quantity) as total_purchase
           FROM ORDERS
           JOIN PRODUCTS ON ORDERS.product_id = PRODUCTS.product_id
           JOIN PRODUCT_CATEGORY
              ON PRODUCTS.category_id = PRODUCT_CATEGORY.category_id
           WHERE lower(order_status) in ('shipped','delivered')
           GROUP BY cat_name
           ORDER by total_purchase desc
           LIMIT 5
           ;")

# Plot using ggplot2
```

```r
ggplot(query_2, aes(x = reorder(category, total_purchase),
                    y = total_purchase, fill = category)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  coord_flip() +
  labs(title = "Top 5 Categories by Quantity",
       x = "Category",
       y = "Quantity Sold") +
  theme_minimal() +
  theme(legend.title = element_text(size = 12),
        legend.text = element_text(size = 10))
```



This bar chart depicts the top five product categories ranked by quantity sold. Beauty products lead, indicating high customer demand, followed by toys, which may suggest popularity or a large customer base for these items.

3. Top Products across categories (by Total Amount)

```r
query_3 <- dbGetQuery(my_connection,
           "WITH product AS (
              SELECT
                DISTINCT p.product_id,
                pc.cat_name,
                p.product_name
```

```
              FROM PRODUCTS as p
              JOIN PRODUCT_CATEGORY as pc ON pc.category_id = p.category_id
            ),
            order_amount AS (
              SELECT
                o.product_id AS product_id,
                SUM(o.quantity * p.price) AS total_amount
              FROM ORDERS as o
              JOIN PRODUCTS as p ON o.product_id = p.product_id
              WHERE LOWER(o.order_status) IN ('shipped', 'delivered')
              GROUP BY o.product_id
            ),
            rnk AS (
              SELECT
                pr.cat_name,
                pr.product_name,
                oa.total_amount,
                ROW_NUMBER() OVER (PARTITION BY pr.cat_name
                ORDER BY oa.total_amount DESC) AS rnk
              FROM order_amount as oa
              JOIN product as pr ON oa.product_id = pr.product_id
            )
            SELECT
              cat_name,
              SUBSTR(product_name,1,20) as product_name,
              total_amount
            FROM rnk
            WHERE rnk IN (1)
            ORDER BY total_amount DESC
            ;")

kable(query_3, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

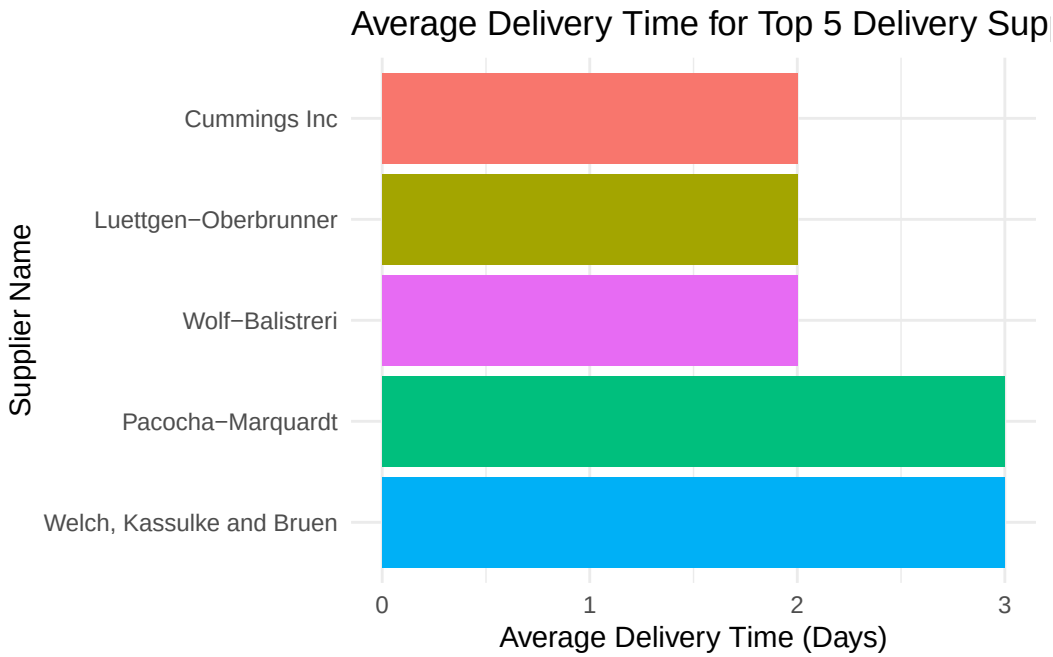
According to this bar chart, the 'Toro TimeMaster 30' leads significantly, meaning a strong market preference. The 'InfinityPad Tablet 12.9" Pro' and the 'Garmin inReach Mini Satellite Communicator' follow, suggesting diverse consumer interests or needs in garden, computer and outdoor categories.

4. Average delivery time for orders across top 5 delivery suppliers

cat_name	product_name	total_amount
Garden	Toro TimeMaster 30-I	14000
Computers	InfinityPad Tablet 1	11200
Outdoors	Garmin inReach Mini	5500
Tools	SmartSaw Table Saw T	5500
Baby	Nanit Plus Smart Bab	2000
Beauty	Clarisonic Mia Smart	1500
Toys	Rival Prometheus MXV	1330
Shoes	PaceSetter Marathon	1250
Health	Air Purifier	1200
Grocery	SweetSensation Stevi	84

```
# Define the SQL query for average delivery time for orders across top 5 delivery suppliers
query_4 <- dbGetQuery(my_connection,
            "SELECT
                sup.supplier_id,
                sup.supplier_name AS supplier_name,
                AVG(julianday(s.delivered_timestamp)
                   - julianday(s.dispatch_timestamp)) AS delivery_time
            FROM SHIPMENT AS s
            JOIN ORDERS AS o ON o.shipment_id = s.shipment_id
            JOIN PRODUCTS AS p ON p.product_id = o.product_id
            JOIN SUPPLIERS AS sup ON sup.supplier_id = p.supplier_id
            WHERE LOWER(s.status) = 'delivered'
            GROUP BY sup.supplier_id, sup.supplier_name
            ORDER BY delivery_time, supplier_name desc
            LIMIT 5;")

# Plot using ggplot2
ggplot(query_4, aes(x = reorder(supplier_name, delivery_time),
                    y = delivery_time, fill = supplier_name)) +
  geom_bar(stat = "identity") +
  coord_flip() +  # Flip the coordinates so that supplier names are on the y-axis
  scale_x_discrete(limits = rev(levels(reorder(query_4$supplier_name, query_4$delivery_time)))
  labs(title = "Average Delivery Time for Top 5 Delivery Suppliers",
       x = "Supplier Name",
       y = "Average Delivery Time (Days)") +
  theme_minimal() +
  theme(legend.position = "none")
```



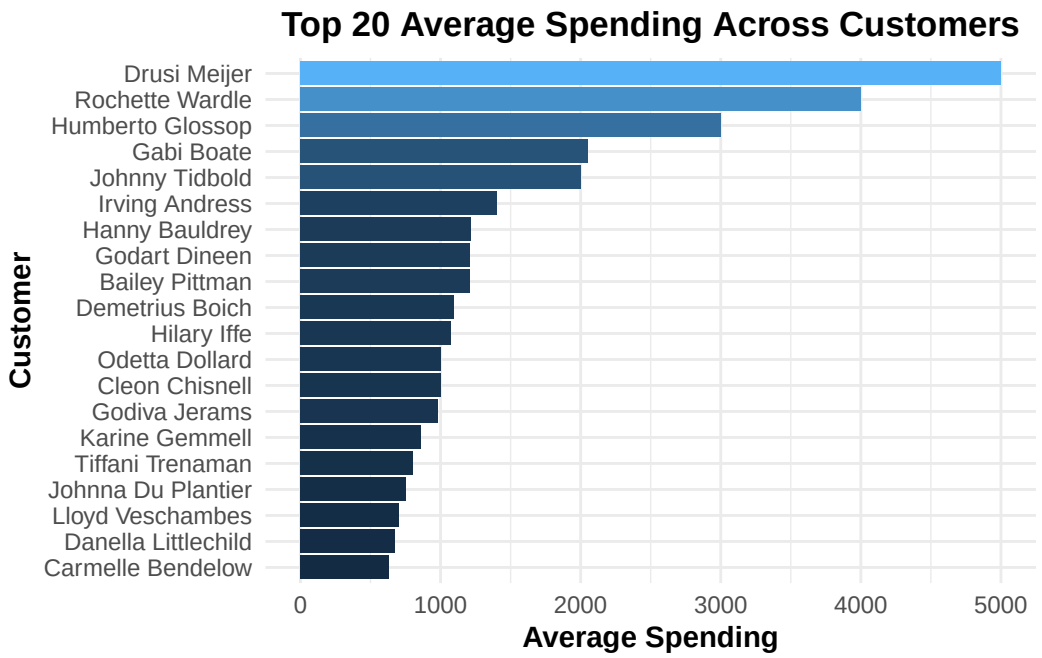
The bar chart displays the average delivery time for the five leading suppliers. Cummings Inc, Luettgen−Oberbrunner and Wolf−Balistrer have the fastest delivery time while the other two suppliers have over 2 days for delivery.

«««< HEAD **5. Top 20 Average Spending across customers** ======= 5. Top 20 Customers based on Average Spending »»»> 0ce7d51ab8a3a244bfbefc823e513dcb0eb23e0b

```
# Define the SQL query
query_5 <- dbGetQuery(my_connection,
           "SELECT
              o.customer_id as customer_id,
              c.first_name ||' '|| c.last_name as customer_name,
              AVG(p.price*o.quantity) as avg_amount,
              SUM(p.price*o.quantity) as total_amount
            FROM ORDERS as o
            JOIN CUSTOMERS as c ON o.customer_id = c.customer_id
            JOIN PRODUCTS as p ON p.product_id = o.product_id
            WHERE LOWER(o.order_status) IN ('shipped', 'delivered')
            GROUP BY o.customer_id,customer_name
            ORDER BY avg_amount DESC
            limit 20
           ;")

# Plot using ggplot2
```

```
ggplot(query_5, aes(x = reorder(customer_name, avg_amount),
                    y = avg_amount, fill = avg_amount)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  labs(title = "Top 20 Average Spending Across Customers",
       x = "Customer",
       y = "Average Spending") +
  theme_minimal() +
  theme(axis.title.x = element_text(face = "bold"),
        axis.title.y = element_text(face = "bold"),
        plot.title = element_text(hjust = 0.5, face = "bold"),
        legend.position = "none")
```



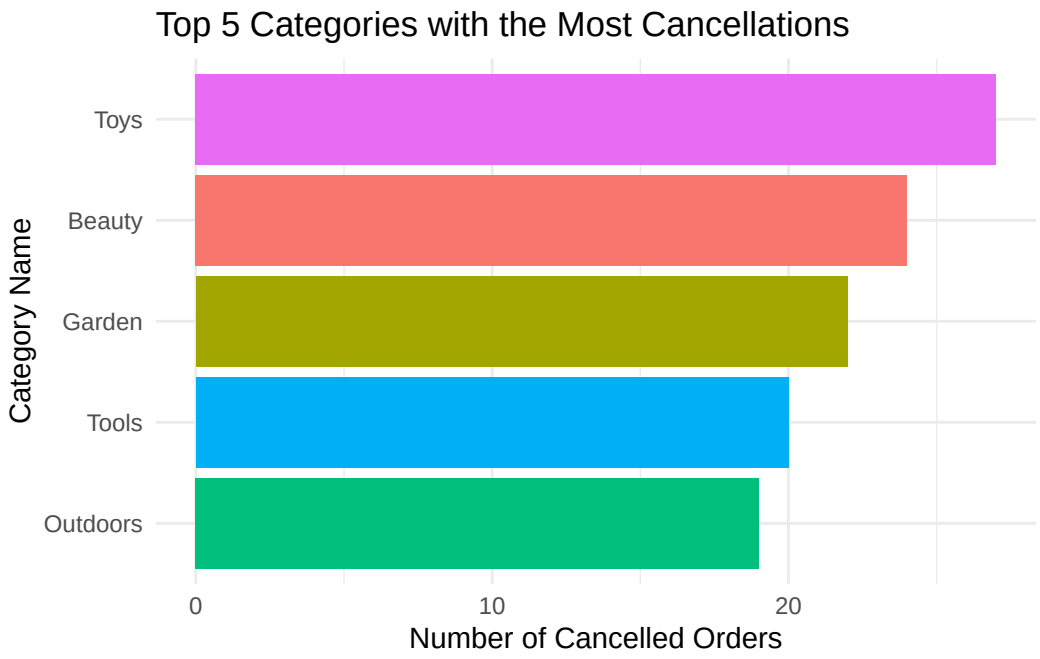
Among the top 20 customers, Drusi Meijer is significantly leading, with average spending over £5000 indicating high-value transactions or frequent purchases. This suggests a potential segment of premium customers who contribute substantially to sales revenue.

6. Top 5 Categories with the Most Cancellations

```
query_6 <- dbGetQuery(my_connection,
           "SELECT
              cat_name,
              COUNT(o.quantity) as total_cancelled
```

```
            FROM ORDERS as o
            JOIN PRODUCTS as p ON p.product_id = o.product_id
            JOIN PRODUCT_CATEGORY as pc on pc.category_id = p.category_id
            WHERE LOWER(order_status) = 'cancelled'
            GROUP BY cat_name
            ORDER BY total_cancelled DESC
            LIMIT 5
           ;")

# Visualization
ggplot(query_6, aes(x = reorder(cat_name, total_cancelled),
                    y = total_cancelled, fill = cat_name)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  labs(title = "Top 5 Categories with the Most Cancellations",
       x = "Category Name",
       y = "Number of Cancelled Orders") +
  theme_minimal() +
  theme(legend.position = "none")
```



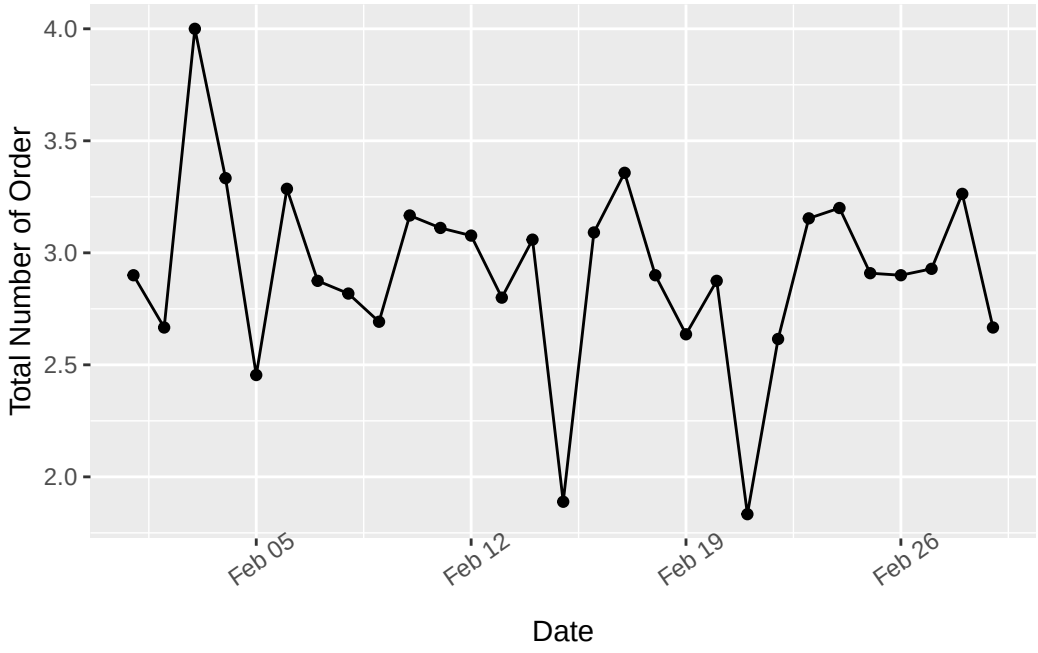
The bar plot reveals that the toy category has the highest number of cancelled orders, followed closely by the beauty category. This underscores the importance for the business to closely

monitor these categories and investigate the reasons for cancellations, whether they were initiated from the buyer's side or seller's side, before proceeding with any further actions.

7. Average number of orders across time

```
query_7 <- dbGetQuery(my_connection,
           "SELECT
              order_timestamp as date,
              AVG(o.quantity) as total_order
            FROM ORDERS as o
            WHERE LOWER(order_status) IN ('shipped', 'delivered')
            GROUP BY order_timestamp
            ORDER BY date
           ;")
```

```
ggplot(query_7,aes(x=as.Date(date),y=total_order,group=1))+
      geom_point(stat="identity")+
      geom_line(stat="identity")+
      labs(x="Date",y="Total Number of Order")+
        theme(axis.text.x=element_text(angle=35))
```

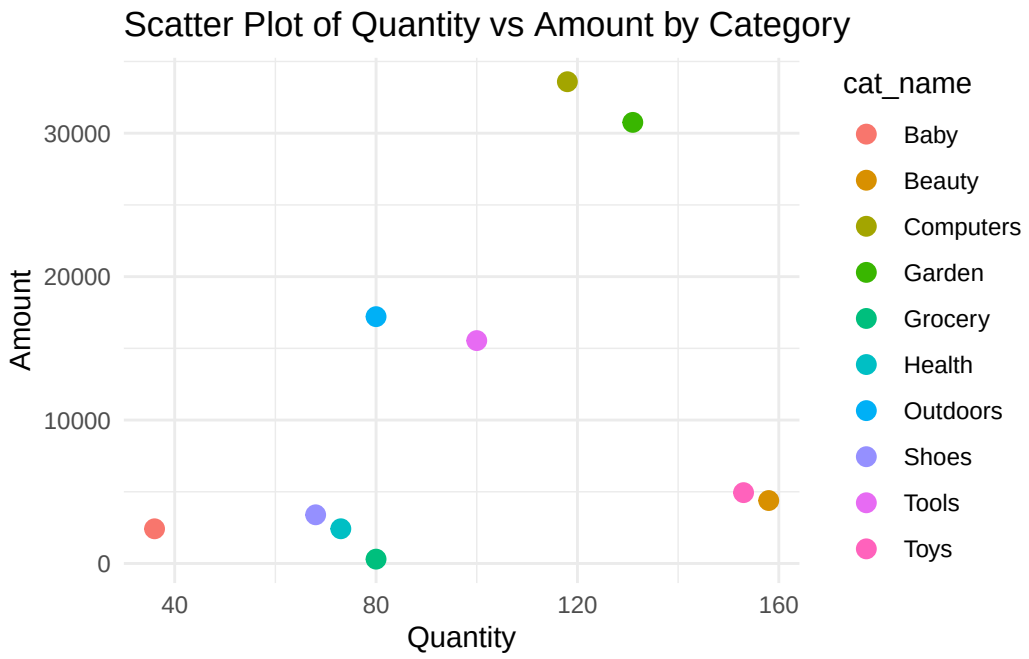


The average number of orders fluctuates significantly over time with no consistent overall trend. Peaks reach an average of 4 orders per day, while lows drop below 2 orders per day.

8. Scatter plot for revenue across quantity (by category)

```
query_8 <- dbGetQuery(my_connection,
           "SELECT
              cat_name,
              SUM(o.quantity) as quantity,
              SUM(p.price * o.quantity) as amount
            FROM ORDERS as o
            JOIN PRODUCTS as p ON p.product_id = o.product_id
            JOIN PRODUCT_CATEGORY as pc on pc.category_id = p.category_id
            WHERE LOWER(order_status) IN ('shipped', 'delivered')
            GROUP BY cat_name
           ;")

ggplot(query_8, aes(x = quantity, y = amount, color = cat_name)) +
  geom_point(size = 3) +
  theme_minimal() +
  labs(title = "Scatter Plot of Quantity vs Amount by Category",
       x = "Quantity",
       y = "Amount") +
  theme(legend.position = "right")
```



The scatter plot indicates varied sales across categories. High-value item likes Computers show

substantial sales amounts, while Outdoors products suggest high revenue with lower quantities sold. Category like Beauty exhibits highest quantities, indicating most frequent purchases of items.