IB9HP Report - Group 11

Introduction

Our project revolves around an online shoe retailer. Our database is tailored to track sales, manage inventory, and monitor vital business metrics. This report will provide insights into our data generation, storage procedures, automatic updating mechanisms, and a comprehensive data analysis.

1. Entity-Relationship (E-R) Diagram Design

The e-commerce website database design aims to create an efficient system for selling shoes online. In this report, we present the Entity-Relationship (E-R) diagram and the corresponding SQL database schema, ensuring data integrity and minimizing redundancy.

1.1 Initial E-R Diagram

To create the ER diagram, each team member created their own idea of what the ER diagram for a shoes e-commerce company should look like. We then came together and merged our ideas. The following figure shows an early draft of out ER diagram.

The E-R diagram captures the essential components of the e-commerce system: 1. Entities: - Inventory: Representing the shoes available for sale. - Customer: Storing information about website users. - Category: Grouping products based on shoe types (e.g., heels, boots, trainers, casual). - Suppliers: Managing information about shoe suppliers. - Shipping Methods: Describing available shipping options. - Payment: Recording payment details. - Transactions: Linking orders and payments. - Advert: Any advertisements that were created in relation to a specific product. - Rating: A numerical ranking assigned to each of the products with each order. - Discount: Lists discount codes that can be applied to orders.

The relationships between these entities will be described in the next section of the report.

The ER diagram presented here is not the final version. As we worked through the project we made changes based on our findings from data generation and other work, as well as changes to make sure the ER diagram followed correct structuring rules. These changes will be presented throughout the report as well as including a final version.

One of these changes was removing the double relationship between transaction and order. As we are considering order as a relationship, transaction can be directly linked to it. We thus combined transaction and payment, as having 2 entities here was over complicating the database. That portion of the ER diagram then looked as follows:

Originally we planned to offer discounts through advertisements that customers would find online. When generating the data, we realized that it would be impractical to offer discounts solely through advertising. Therefore, we edited the ER diagram so that the ads were directly related to the products and discounts were created as a separate entity. For simplicity, the discounts entity also included site-wide sales. During the time period of the profile, we believe that 4 discounts are reasonable for the size of our e-commerce company.

Moreover, the ER-diagram was updated to improve the database's ability to handle complex business requirements and enhance data management. The supplier entity was removed from the ER diagram, due to changes in business operations and a simplification of the database schema. The supplier-related data can now be managed by separate systems or through an integrated procurement process that does not need to be directly represented in this particular schema. We also decided that for a small e-commerce company selling

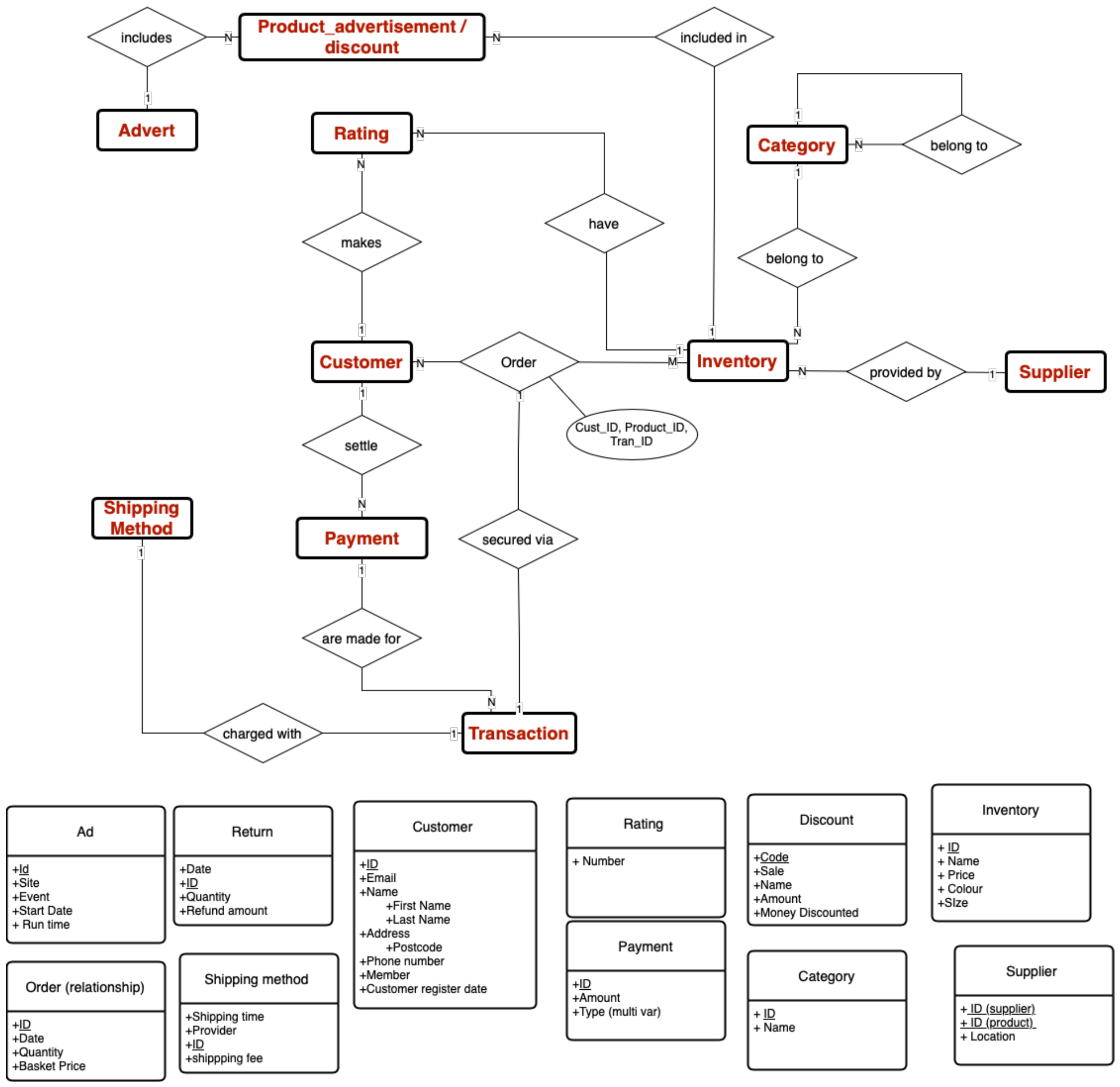


Figure 1: Early ER diagram for shoes e-commerce business

only 100 products, it was unrealistic to have multiple suppliers so we assume from now on the company runs out of one warehouse and this information is contained within the inventory table.

Advertisement is now tied to sales through the Order table, as a column was added to see which orders were placed via an advertisement. Shipping details have been expanded to now include courier, speed and cost, designed to enhance logistics tracking. The introduction of a separate order-related discount entity signals a shift toward order-specific discounts rather than product-based discounts. Finally, the product rating system is decoupled from customers, in order to keep ratings anonymous and to simplify review management.

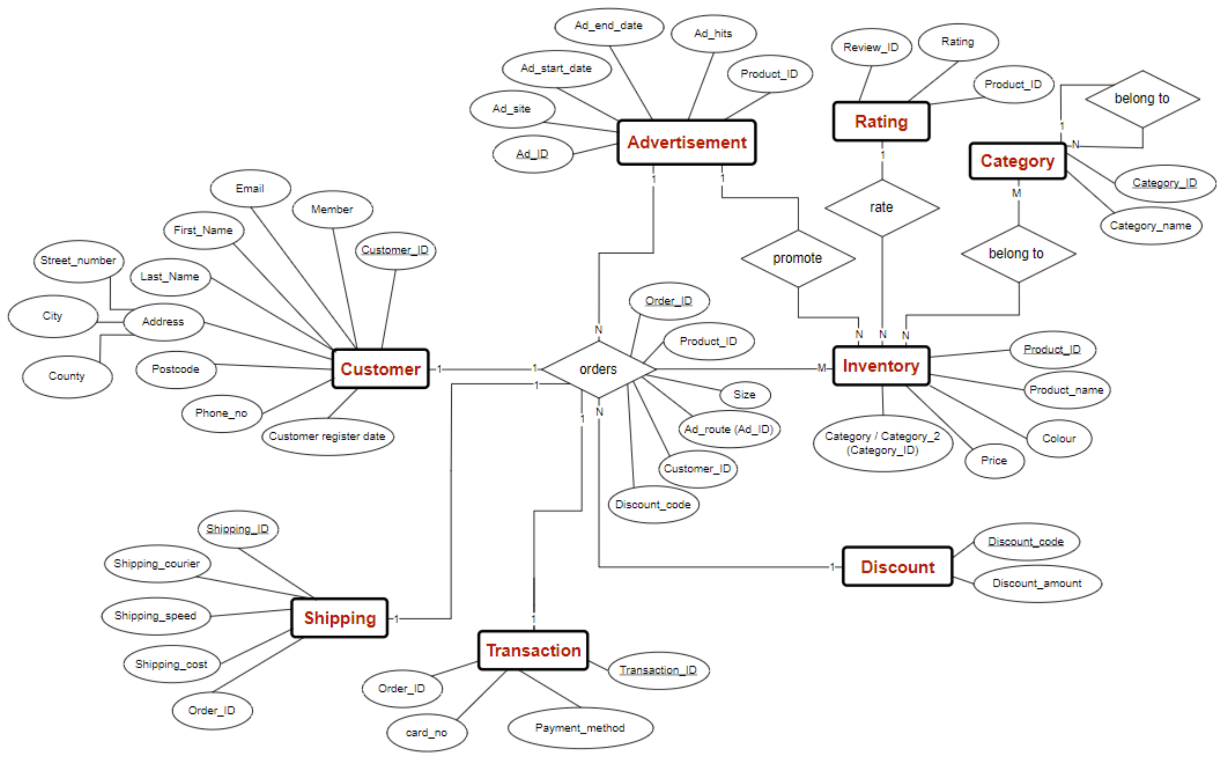


Figure 2: ER Diagram Final Version

Throughout the ER diagram generation we made a number of assumptions which simplified the way we collected and stored our data. - Shipping fee is charged depending on the speed of the delivery rather than the customer's desired delivery location. - The data was generated over a three month period spanning from November 2023 to January 2024. - Certain columns will contain missing values to simulate real-world data. - Only one discount code can be applied per order.

1.2 Relationships:

- Customer-Order: One-to-One (Customers can place 1 order and each order is related to one customer.)
- Discount-Order: One-to-Many (Each Discount can apply to multiple Orders, but each Order has at most one Discount applied to it.)
- Transaction-Order: One-to-One (Each transaction is linked to a single order, which indicates that every order results in one transaction.)
- Shipping-Order: One-to-One (Each order has a unique shipping entry, which details the shipping process for that order.)
- Advertisement-Order: Many-to-One (Each order can only be linked to one advertisement, but each ad can be linked to multiple orders.)
- Advertisement-Inventory: One-to-Many (Each advertisement is promoting a single inventory item but a product can be promoted in more than one advertisement.)

- Rating-Inventory: Many-to-One (Each inventory item can have multiple ratings, but each rating is associated with one inventory item.)
- Category-Inventory: Many-to-Many (An product can belong to up to 2 different categories, and each category can contain multiple products.)
- Category-Category: One-to-Many (A category can be contained within another category)

Relationship sets (reference figure) show how these relationships work.

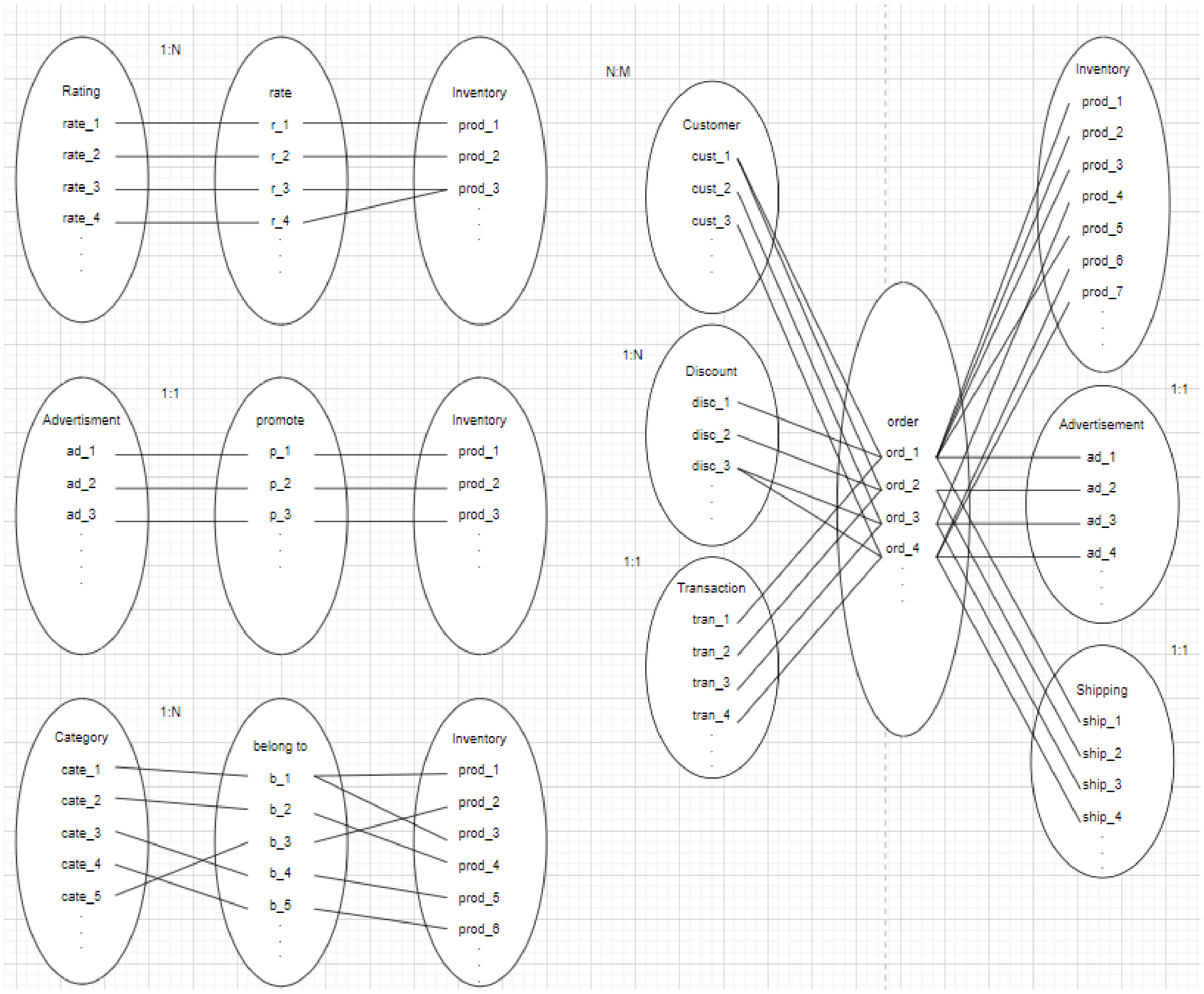


Figure 3: Transaction Edit

1.3 Cardinalities:

o One: Denoted as “1” (e.g., one customer places many orders).

o Many: Denoted as “N” (or “M) (e.g., many products can belong to a category).

Logical Schema: The ER diagram was then converted into a logical schema with foreign keys included for the relevant entities.

Inventory($\underline{product-id}$, product_name, colour, price, $category\_id$)

orders($\underline{order-id}$, product(1-5), size(1-5), $discount\_code$, $customer\_id$, ad_id)

customer($\underline{customer-id}$, email, first_name, last_name, member, address, postcode, phone_no, customer_reg_date)

shipping($\underline{shipping-id}$, shipping_courier, shipping_cost, shipping_speed, *order_id*)

transaction($\underline{transaction-id}$, *order_id*, transaction_type, card_no)

advertisement($\underline{ad-id}$, ad_site, ad_start_date, ad_end_date, ad_hits, *product_id*) category($\underline{catagory-id}$, category_name)

discount($\underline{discount-code}$, discount_amount)

2. Data Generation and Schema Population

2.1 Synthetic Data Generation

In order to populate the populate the SQL database artificial data must be generated. There are multiple techniques that can be used to generate the data and we found a combination of these gives the best results- for our database Mockaroo (mockaroo.com) and R where the main tools used.

The LLM Google Gemini was also used to aid in the generation or product names. We prompted the LLM by asking for names linked to floral, river and places as well as some more common unisex names. Gemini was used so that we could produce names to fit our hypothetical brand as the names generated by Mockaroo were not appropriate. The names generated were then checked and manually edited to fit our needs. When the data was imported into R, checks were implemented for uniqueness, as although 100 unique names were requested, this is something the LLM struggled to provide.

The prompt sequence used to start generation: "Please write me a list of 100 unique names separated by a comma. Can the names be influenced by factors such as cities, flowers, rivers etc and they can be common male female or unisex names" .

Workflow of data generation:
1. Consider each entity and its components individually, use Mockaroo to generate simple entries such as custom lists and curated IDs. For each variable consider the amount of missing entries that would occur in this data set (as would be realistic in real life data- as for example sometimes not all fields are compulsory e.g last name or phone number)
2. Import Mockaroo generated data into R for additions of further variables, data edits and quality checks 3. Ensure that for foreign keys referential integrity is upheld- eg using one set of product IDs which correspond to named products. 4. Populate SQL database using generated data and analyse it using SQL commands 5. Ensure to continually update ER diagram and assumptions based on findings made in data generation

The data set spans Nov 2023 to Jan 2024 and in that time there were:

Get summary using code * 1000 orders * 1000 customers * 100 products available to purchase * 150 adverts * 989 anonymous reviews made * 4 shipping couriers which each have a next day, 3-5 day, and eco saver option* 4 types of discount code used

The process of data generation underlines that there were updates that could improve our model that we had not previously considered.

Some assumptions about out e commerce company that we updated during the Data generation include: basket limit of 5 items, one order per customer in our time frame, discount and sale become one variable (as discount usually only applied to full price items in e commerce so basket can either be made when there is a sitewide sale or a discount is applied to the basket).

A complete table summarising how each variable was generated can be seen in the appendix. Variables in R are generated using R code and those generated in Mockaroo are generated through a mixture of mockaroos pre-built options and Ruby code. Although this requried us to learn how Ruby code worked it meant we were able to build whole tables in Mockaroo and ensure that referential integrity was maintained in data generation.

Examples of the Ruby code used in Mockaroo are:

Conditional statements to ensure correct referecing. This can be seen for example in shipping cost which is coded with the following Ruby Code

```
if discount_code== 'stu20'then '0.2'
elsif discount_code== 'xmas10'then '0.1'
elsif discount_code== 'jan5'then '0.05'
elsif discount_code== 'join15'then '0.15'
end
```

Pulling data from other columns to create ID that includes customer info

```
concat(lower(email)[0,2], cust_num)
```

Examples of the R code used to generate variables include:

Code that generated whether or not orders were made after a customer visited the site via an ad:

```
ad_op= advertisement_$ad_id

zeros = rep(0, 400)

options= c(ad_op, zeros)

ad_route = sample(options, 1000, replace=TRUE)

order$ad_route= sample(options, 1000, replace=TRUE) # to reflect real patterns
```

One key consideration in data generation was how many missing values there would be and where they would arise. All real-world data sets contain missing values which can be for multiple reasons. In the example of our e commerce data some of the data is collected via customer input and not all customer entities are mandatory in the order process. The following variables were coded as they are to follow realistic patterns:

Review scores- generated after analysing review patterns on different e commerce websites that sell shoes such as ASOS.

Items ordered- every order contains 1 item but the probability of ordering each additional item decreases.

Orders made via ad routes- as shown in R code, we estimate roughly 28% of orders are made after accessing the site from an advertisement.

2.2 Data Import and Quality Assurance

Once we had generated data for all the tables in our schema we were able to populate the schema.

To start, we downloaded the generated data into a folder that would be used to hold the data required for the schema.

The total list of tables that we generated data for can be seen below.

```
library(RSQLite)
library(readr)
connection <- RSQLite::dbConnect(RSQLite::SQLite(), "my_database.db")
all_files <- list.files("shoes_data/")
for (variable in all_files) {
  this_filepath <- paste0("shoes_data/",variable)
  this_file_contents <- readr::read_csv(this_filepath, show_col_types = FALSE)
  number_of_rows <- nrow(this_file_contents)
  number_of_columns <- ncol(this_file_contents)
  print(paste0("The file: ",variable,
              " has: ",
              format(number_of_rows,big.mark = ","),
              " rows and ",
              number_of_columns," columns"))
}
```

```
## [1] "The file: advertisement.csv has: 150 rows and 6 columns"
## [1] "The file: category.csv has: 12 rows and 2 columns"
## [1] "The file: customer.csv has: 1,000 rows and 9 columns"
## [1] "The file: discount.csv has: 4 rows and 2 columns"
## [1] "The file: inventory.csv has: 100 rows and 6 columns"
## [1] "The file: orders.csv has: 1,000 rows and 14 columns"
## [1] "The file: rating.csv has: 989 rows and 3 columns"
## [1] "The file: shipping.csv has: 1,000 rows and 5 columns"
## [1] "The file: transactions.csv has: 1,000 rows and 4 columns"
```

Then we created the physical schema we needed to populate uaing the physical schema developed in part 1 of the project:

```
connection <- RSQLite::dbConnect(RSQLite::SQLite(), "my_database.db")
```

```
CREATE TABLE category (
  category_id INT NOT NULL PRIMARY KEY,
  category_name TEXT NOT NULL
);
```

```
CREATE TABLE inventory (
  product_id INT PRIMARY KEY,
  product_name TEXT NOT NULL,
  colour TEXT NOT NULL,
  price DECIMAL(5,2) NOT NULL,
  category INT NOT NULL,
  category_2 INT,
  FOREIGN KEY (category)REFERENCES category ('category_id'),
  FOREIGN KEY (category_2) REFERENCES category('category_id')
);
```

```
CREATE TABLE advertisement (
  ad_id INT PRIMARY KEY,
  ad_site TEXT NOT NULL,
  ad_start_date DATE NOT NULL,
  ad_end DATE NOT NULL,
  ad_hits INT,
  product_id INT NOT NULL,
  FOREIGN KEY (product_id) REFERENCES inventory(product_id)
);
```

```
CREATE TABLE customer (
  email TEXT NOT NULL,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  customer_id INT PRIMARY KEY,
  member TINYINT NOT NULL,
  address VARCHAR(50) NOT NULL,
  postcode VARCHAR(50) NOT NULL,
```

```sql
  reg_date DATETIME DEFAULT CURRENT_TIMESTAMP,
  phone_num VARCHAR(20)
);
```

```sql
CREATE TABLE shipping (
  shipping_id INT PRIMARY KEY ,
  shipping_courier VARCHAR(100) NOT NULL,
  shipping_speed TEXT NOT NULL,
  shipping_cost DECIMAL(5,2) NOT NULL,
  order_id VARCHAR(50),
  FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

```sql
CREATE TABLE rating (
  rating INT,
  product INT,
  review_id VARCHAR(50) PRIMARY KEY,
  CHECK (rating >= 1 AND rating <= 5),
  FOREIGN KEY (product) REFERENCES Inventory(product_id)
);
```

```sql
CREATE TABLE discount (
  discount_code VARCHAR(50) PRIMARY KEY,
  amount DECIMAL(2,2)
  );
```

```sql
CREATE TABLE orders(
  order_id VARCHAR(15) PRIMARY KEY,
  product_1 INT NOT NULL,
  product_2 INT,
  product_3 INT,
  product_4 INT,
  product_5 INT,
  size_1 INT NOT NULL,
  size_2 INT,
  size_3 INT,
  size_4 INT,
  size_5 INT,
  discount_code VARCHAR(50),
  customer_id NOT NULL,
  ad_route INT,
  FOREIGN KEY (discount_code) REFERENCES discount(discount_code),
  FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

```sql
CREATE TABLE transactions(
  transaction_id VARCHAR(20) PRIMARY KEY,
  payment_method VARCHAR(50),
  card_no INT NOT NULL,
  order_id INT,
  FOREIGN KEY(order_id) REFERENCES Order_(order_id)
);
```

The generated data was then appended to the schema as follows:

```
connection <- RSQLite::dbConnect(RSQLite::SQLite(), "my_database.db")
for (variable in all_files) {
  this_filepath <- paste0("shoes_data/", variable)
  this_file_contents <- read_csv(this_filepath, show_col_types = FALSE)
  table_name <- gsub(".csv", "", variable)
  dbAppendTable(connection, table_name, this_file_contents)
}
```

2.3 Data Quality Checks

Then a range of data quality checks were carried out to ensure that data follows the correct formatting, duplicated are not present where uniqueness is requires and the whole schema has referential integrity.

First we checked to make sure that all of the primary keys were unique.

```
for (variable in all_files) {
  this_filepath <- paste0("shoes_data/", variable)
  this_file_contents <- readr::read_csv(this_filepath, show_col_types = FALSE)
  number_of_rows <- nrow(this_file_contents)
  print(paste0("Checking uniqueness for: ", variable))
  print(paste0(" and it is ", nrow(unique(this_file_contents[,1])) == number_of_rows))
}
```

```
## [1] "Checking uniqueness for: advertisement.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: category.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: customer.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: discount.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: inventory.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: orders.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: rating.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: shipping.csv"
## [1] " and it is TRUE"
## [1] "Checking uniqueness for: transactions.csv"
## [1] " and it is TRUE"
```

Thus each primary key is unique for all of our tables. Next we checked that the data types were correct for each of the columns in our tables. An example of this is shown below with the discount table:

```sql
SELECT typeof(discount_code), typeof(amount)
FROM discount
WHERE typeof(amount) != 'real'
OR typeof(discount_code) != 'text'
```

Table 1: 0 records

typeof(discount_code)	typeof(amount)

Thus we can see that the generated data for the discount percentage (labelled ‘amount’) are all decimals and the discount codes are all text values as we would expect.

Furthermore, we also checked that some of our columns which aren't primary keys were unique. For example, in our inventory table we wanted to check that each product has a unique name.

```
SELECT COUNT(DISTINCT product_name) AS total_products
FROM inventory
```

Table 2: 1 records

total_products
100

We can also perform sense checks such as confirming that the start date of an advertisment is always before the end dat. Within the assumptions of our data, we also specified that all advertisements should be ran for a minimum of 10 days and a maximum of 60 days.

```
SELECT *
 FROM advertisement
WHERE
(julianday(ad_end) - julianday(ad_start_date)) BETWEEN 10 AND 60
AND ad_start_date BETWEEN '2023-09-01' AND '2024-01-31'
AND ad_end BETWEEN '2023-09-01' AND '2024-01-31'
```

Table 3: 0 records

ad_id	ad_site	ad_start_date	ad_end	ad_hits	product_id

Following this, referential checks were carried out to ensure that the foreign keys within the data had worked successfully. For example, we checked that the customer ID's assigned to each of the orders existed within the customer table.

```
if (any(!orders$customer_id %in% customer$customer_id)) {
  print("Invalid customer IDs. Some customer IDs do not exist in the Customer table.")
} else {
  print("Valid customer IDs")
}
```

```
## [1] "Valid customer IDs"
```

This was completed for each of our foreign keys.

We also checked to make sure that each of the products ordered within the orders table existed within the inventory table.

```
violations <- FALSE
for (i in 1:nrow(orders)) {
  for (j in 1:5) {
    product_id <- orders[i, paste0("product_", j)]
    if (is.na(product_id) || product_id == "") {
      next
    }
    if (!product_id %in% inventory$product_id) {
      violations <- TRUE
      cat("Row", i, ": Product", j, "(", product_id, ") does not exist in the inventory.\n")
    }
  }
}
if (!violations) {
  print("All products exist in the inventory.")
}
```

```
## [1] "All products exist in the inventory."
```

A further example of data validation can be seen for the shipping table below. This checks that there are no empty rows within the columns, all related orders exist within the order table, and the shipping cost is greater than zero.

```
na_shipment <- sapply(shipping, function(x) sum(is.na(x)))
order_numbers <- unique(shipping$order_id)
if (all(na_shipment == 0) &&
    all(shipping$shipping_cost > 0) &&
    all(order_numbers %in% orders$order_id)) {
  print("Shipment data is valid.")
  # Load the data into the database
} else {
  print("Shipment data is not valid. Please correct the errors.")
}
```

```
## [1] "Shipment data is valid."
```

These quality checks can be repeated whenever new data is gathered.

Note: Due to the method of generating data and assessing schema in the process data redundancy is not present and data is normalised to 3NF.

2.4 Calculated Variables

There are some key results that are not stored in the data schema as they are calculated variables. However these variables do contain useful information for business inference, thus they are coded using SQL so they can be easily calculated from the schema:

Total Basket Price: Total Items: Money saved via discount: Total Basket Price

The total items ordered in a basket is calculates as follows:

```
SELECT o.order_id,
SUM(CASE WHEN o.product_1 >= 0 THEN 1
   WHEN o.product_1 IS NULL THEN 0
   ELSE 0 END)+
 SUM(CASE WHEN o.product_2 >= 0 THEN 1
     WHEN o.product_2 IS NULL THEN 0
     ELSE 0 END)+
 SUM(CASE WHEN o.product_3 >= 0 THEN 1
     WHEN o.product_3 IS NULL THEN 0
     ELSE 0 END) +
 SUM(CASE WHEN o.product_4 >= 0 THEN 1
     WHEN o.product_4 IS NULL THEN 0
     ELSE 0 END) +
 SUM(CASE WHEN o.product_5 >= 0 THEN 1
     WHEN o.product_5 IS NULL THEN 0
     ELSE 0 END)
AS number_of_items
```

```sql
FROM orders o
GROUP BY o.order_id
```

Table 4: Displaying records 1 - 10

order_id	number_of_items
#AA1553	2
#AA8927	3
#AB6823	3
#AF6260	4
#AF7695	4
#AI6989	4
#AK1679	2
#AM5703	4
#AN4518	4
#AR3083	3

Then the basket price and total transaction amount can be calculated as follows.

```sql
WITH OrderedItems AS (
  SELECT o.order_id,
         o.product_1 AS item_id,
         COALESCE(i1.price, 0) AS price
  FROM orders o
  LEFT JOIN inventory i1 ON o.product_1 = i1.product_id

  UNION ALL

  SELECT o.order_id,
         o.product_2 AS item_id,
         COALESCE(i2.price, 0) AS price
  FROM orders o
  LEFT JOIN inventory i2 ON o.product_2 = i2.product_id

  UNION ALL

  SELECT o.order_id,
         o.product_3 AS item_id,
         COALESCE(i3.price, 0) AS price
  FROM orders o
  LEFT JOIN inventory i3 ON o.product_3 = i3.product_id

  UNION ALL

  SELECT o.order_id,
         o.product_4 AS item_id,
         COALESCE(i4.price, 0) AS price
  FROM orders o
  LEFT JOIN inventory i4 ON o.product_4 = i4.product_id

  UNION ALL

  SELECT o.order_id,
        o.product_5 AS item_id,
        COALESCE(i5.price, 0) AS price
  FROM orders o
  LEFT JOIN inventory i5 ON o.product_5 = i5.product_id
  )

  SELECT o.order_id,
  SUM(COALESCE(oi.price, 0)) AS basket_price,
  ROUND((
    SUM(COALESCE(oi.price, 0)) * (1 - d.amount)
  ) + s.shipping_cost, 2) AS transaction_amount
  FROM orders o

  INNER JOIN OrderedItems oi ON o.order_id = oi.order_id
  INNER JOIN discount d ON o.discount_code = d.discount_code
  INNER JOIN shipping s ON o.order_id = s.order_id
  GROUP BY o.order_id;
```

Table 5: Displaying records 1 - 10

order_id	basket_price	transaction_amount
#AK1679	80.69	70.98
#AM5703	229.06	198.69
#AN4518	288.49	278.06
#AV2501	194.50	159.59
#AW8102	234.50	193.59
#Ai8568	215.51	197.95
#An0582	193.08	158.45
#As2890	186.46	151.56
#Au5760	97.34	86.73
#Aw9543	35.92	38.11

And then average rating can be calculated as follows

```
SELECT p.product_id,
ROUND(AVG(r.rating), 1) AS product_rating
FROM inventory p
INNER JOIN rating r ON p.product_id = r.product
GROUP BY p.product_id
```

Table 6: Displaying records 1 - 10

product_id	product_rating
10199	2.8
12675	2.7
13506	3.6
13562	3.6
15720	3.3
16458	2.7
17273	2.3
18638	2.8
19337	2.9
19842	3.5

product_id	product_rating

These calculated values can be used for data analysis.

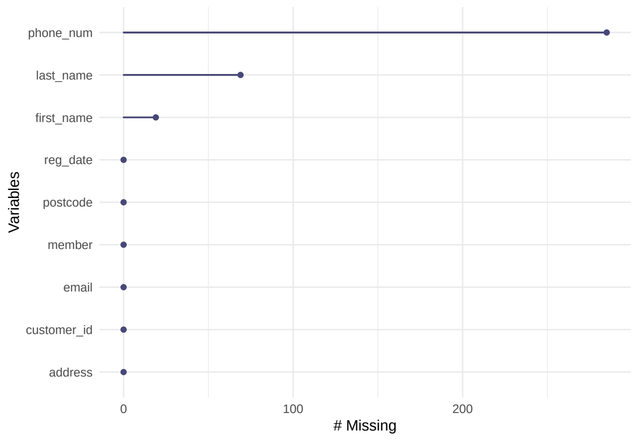
3 Git Hub

4 Data Analysis

This report presents an analysis of a shoe sales dataset using various R packages such as `dplyr`, `ggplot2`, `RSQLite`, `tidyr`, and `lubridate`. The dataset includes information on advertisements, categories, customers, discounts, inventory, orders, ratings, shipping, transactions, and more. The analysis aims to provide insights into top-rated products, top categories, top-selling products, advertisement performance, discount utilization, and other key metrics. The graphs get updated as new data is added to the tables and are saved in the analysis graphs folder on our github which helps us to know how our products are performing with time.

During the data generation stage we selected appropriate columns that would include missing data. An example of this is the customer table where it is quite common that some customers would not fill out all of their information.

```r
library(naniar)
gg_miss_var(customer)
```

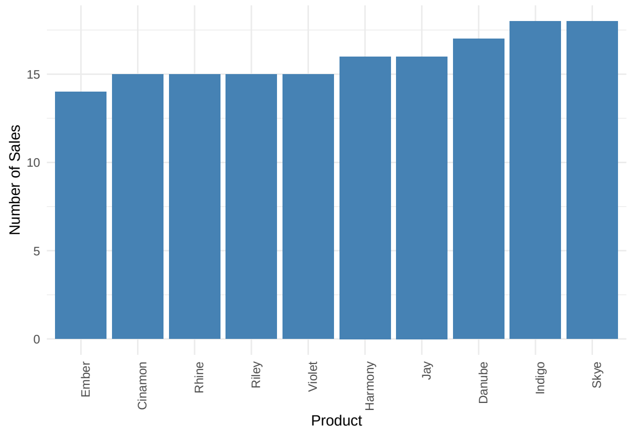


Next, we investigated which products were most popular for our customers. We see that the most ordered products were 'Indigo' from the running section and 'Skye' from the flats section, each with a total of 18 sales.

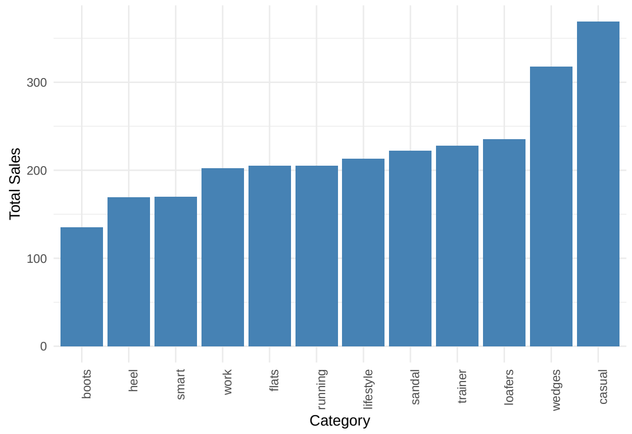
```sql
SELECT i.product_id,
       i.product_name,
       c.category_name,
       COUNT(*) AS num_sales
FROM inventory AS i
INNER JOIN rating AS r ON i.product_id = r.product
INNER JOIN category AS c ON i.category = c.category_id
GROUP BY i.product_id, i.product_name, c.category_name
ORDER BY num_sales DESC
LIMIT 10;
```

Table 7: Displaying records 1 - 10

product_id	product_name	category_name	num_sales
54268	Indigo	running	18
96257	Skye	flats	18
23267	Danube	lifestyle	17
83731	Harmony	flats	16
84955	Jay	running	16
28093	Violet	work	15
42435	Rhine	heel	15
82823	Riley	heel	15
87193	Cinamon	flats	15
13506	Ember	trainer	14

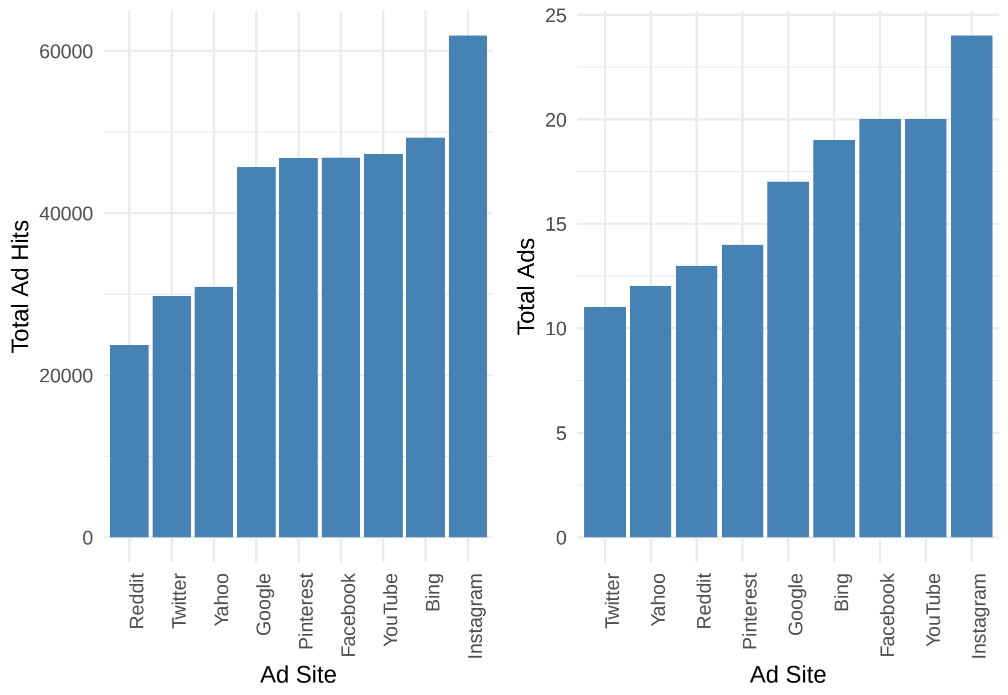


Expanding on this, we can then see which categories are most popular using the chart below.

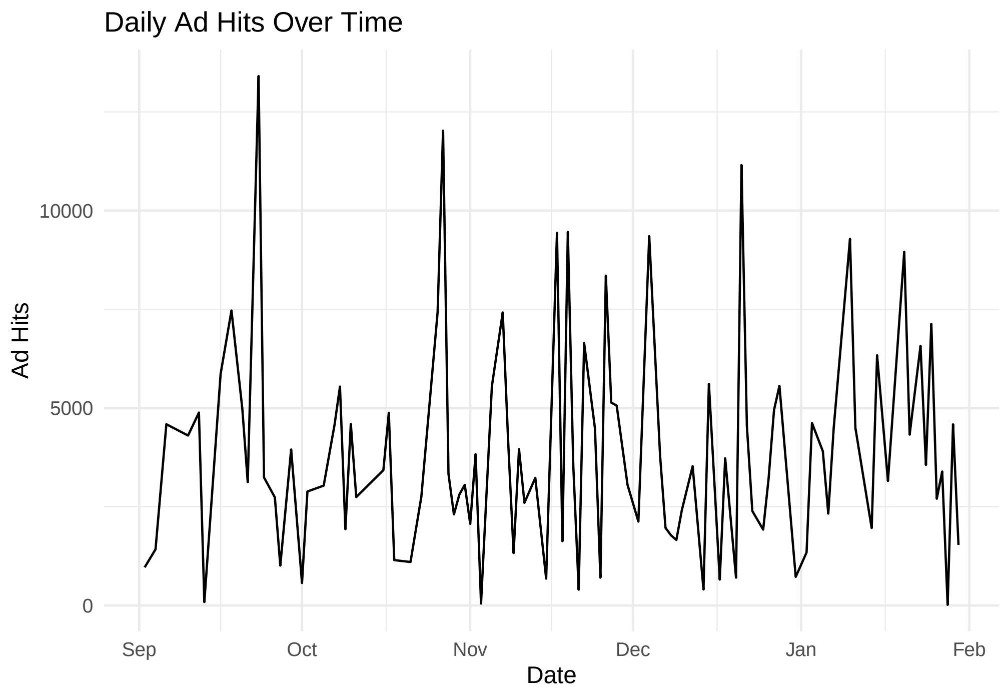


We see that the best selling category is casual, this could be because it also incorporates other categories such as flats. The category with the least units sold was boots.

The performance of advertisements across different sites was analysed based on the number of times an advert generated a sale for the company. We see that Instagram had the best performance and generated the most sales for the business whereas Reddit was the worst with a total of 23,643 hits, compared to Instagram's 61,867 hits. however, in the chart on the right we see how many times each site was used to promote the products. Here we see that Instagram was used 11 more times than Reddit to advertise products.



Furthermore, we can track how the ads performed between 01/09/2023 to 31/01/2024, using the following graph.



We see that there were peaks leading up the Christmas period, as we would expect, and this subsided as January begins.

From the table below we can see the products with the highest ratings:

```
SELECT i.product_id,
       ROUND(AVG(r.rating), 1) AS product_rating,
       i.product_name
```

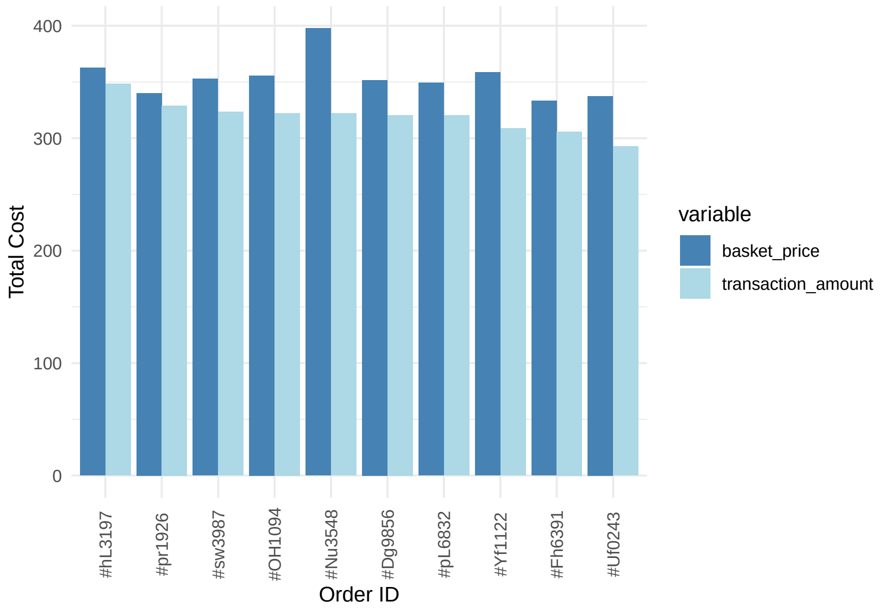
```
FROM inventory AS i
INNER JOIN rating AS r ON i.product_id = r.product
GROUP BY i.product_id
ORDER BY product_rating DESC
LIMIT 5;
```

Table 8: 5 records

product_id	product_rating	product_name
84955	3.9	Jay
53572	3.8	Journey
26232	3.8	Laurel
99608	3.7	Tokyo
96257	3.7	Skye

We see our best selling flat shoe 'Skye' ranked in the top 5 and our joint third best selling shoe 'Jay' rated first overall.

The next chart shows the five largest transactions that were made during the period. We see the total basket price calculated by summing each individual product's price and their respective transaction price which incorporates shipping costs and discounts.



We see that the largest transaction amount was just under £350 with a total basket price of slightly over £360. Overall we see that the average basket price was £150 and the average transaction amount was £167.

Table 9: 1 records

avg_transaction_amount	avg_basket_price
166.9178	150.8236

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