Table of Contents

List of Figures

1. Business Context

Cartify has been operating online for a year, initially experiencing strong sales and a growing customer base. However, as the market has become more competitive, the company has struggled to maintain customer engagement, leading to a decline in repeat purchases and an increasing churn rate. Despite investing in marketing campaigns and promotional strategies, Cartify lacks the necessary insights to determine which efforts are most effective.

One of the key challenges is the absence of a structured, data-driven approach. Without proper tracking and analysis of customer-behavior, marketing decisions are often made based on assumptions rather than concrete evidence. This leads to ineffective targeting, inefficient budget allocation, and missed opportunities for personalising customer interactions. As a result, Cartify faces lost revenue and diminishing customer loyalty.

1.1. Implementing a Strategic Data Solution

To address these challenges, we propose a comprehensive database design that will allow Cartify to track marketing campaigns, analyse customer interactions and make informed decisions. This approach will enable Cartify to identify connections between marketing strategies and customer behavior, resulting in more focused and impactful campaigns.

2. Database Design

The Entity-Relationship diagram for the database was created using Crow's notation, and the description of the cardinalities between the entities is provided in Table A (see Appendix A).

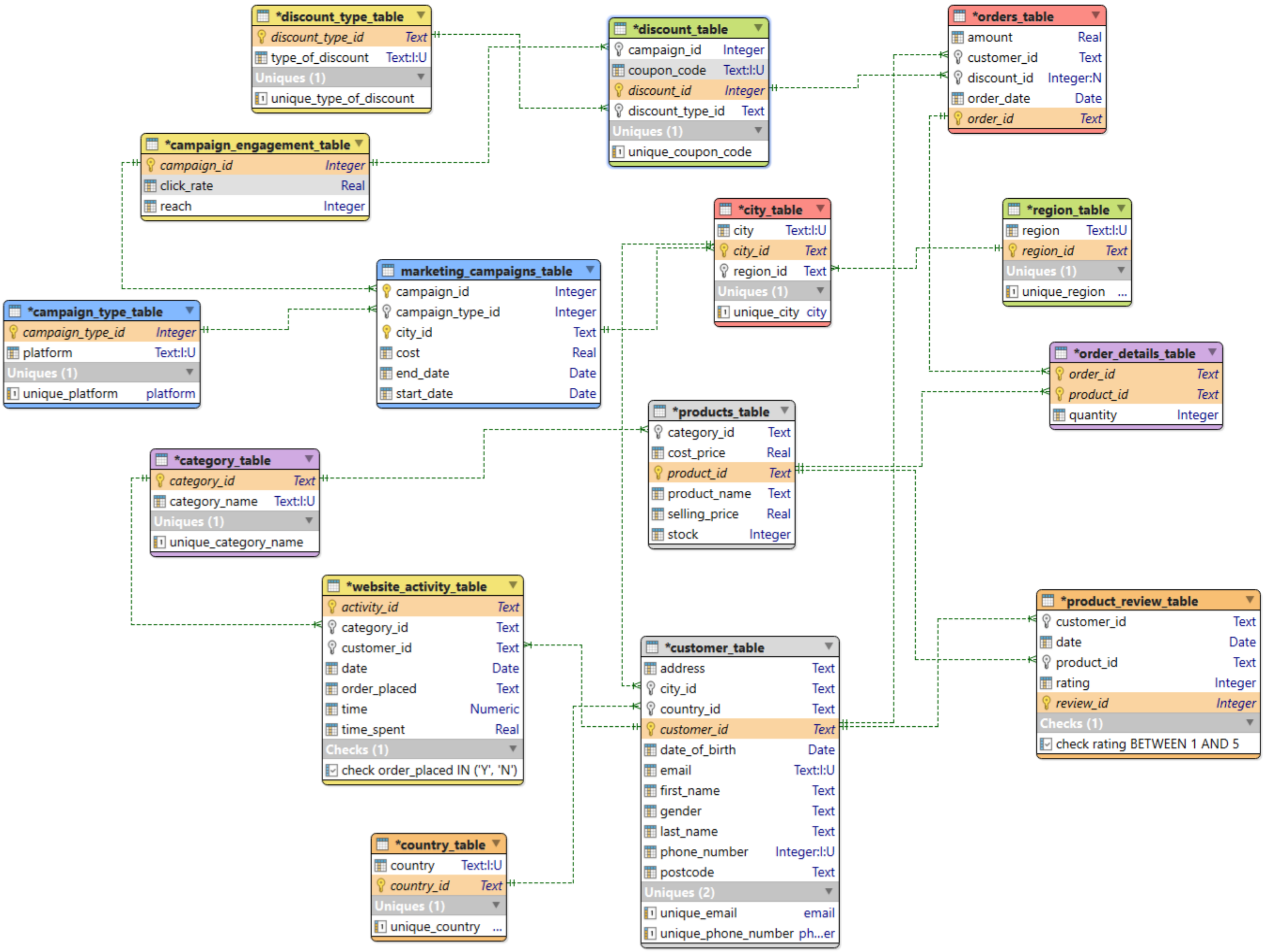

Figure 1: Cartify's Entity-Relationship Diagram for Customer Engagement, Marketing, and Sales Analytics

3. SQL Schema Implementation

The database schema enforces data integrity using PRIMARY KEY, FOREIGN KEY, UNIQUE, and CHECK constraints. The customer table ensures uniqueness with constraints on the email and phone number fields, while its foreign keys reference the city and country tables, enforcing cascading deletions when referenced entries are removed. The orders table maintains relationships with the customer and discount tables, ensuring referential integrity. The order details table uses composite keys to uniquely identify each product within an order while referencing the orders and products tables.

The products table enforces foreign keys to associate products with categories. The product review table ensures that only registered customers can review purchased products and enforces a rating range of 1 to 5. The website activity table restricts the order status field to 'Y' or 'N', maintaining consistency.

Discounts are linked to specific marketing campaigns and discount types, supporting structured promotional management. The marketing_campaigns_table, using composite primary keys, connects with city data and campaign types to ensure each campaign operates within a defined location. The campaign engagement table tracks marketing campaign effectiveness. Geographic segmentation follows a hierarchical structure from regions to cities and countries, ensuring consistency in location-based data.

The schema enforces ON DELETE CASCADE in multiple relationships, ensuring the automatic removal of dependent records when a referenced entity is deleted. Unique constraints prevent duplication and maintain consistency in fields such as coupon codes, category names, and city names. Overall, it preserves data integrity, enforces logical relationships, and simplifies data management.

Data types were selected to optimize efficiency and storage. TEXT fields store variable-length strings for identifiers and attributes, such as customer_id, product_name, and category_name. INTEGER fields define numerical identifiers like discount_id and campaign_id, supporting indexing and efficient lookups. Primary keys defined as INTEGER use the AUTOINCREMENT attribute to generate unique, sequential values. REAL numbers represent financial values such as selling_price and cost_price, allowing decimal precision.

DATE and TIME fields ensure consistency in tracking orders, marketing campaigns, and customer activity.

Finally, the SQL code for database definition and table creation in SQLite is provided in Appendix B.

3.1. Normalisation

The database schema is designed to ensure data integrity, eliminate redundancy, and establish clear relationships between entities through normalisation. It organizes data into separate tables for each entity, with foreign keys linking them, reducing duplication and ensuring consistency.

Customer information is stored in a dedicated table, referencing city and country data through foreign keys, while the region table stores broader geographical classifications. Orders, discounts, products, and categories are managed in separate tables to avoid redundancy. Order details use a composite key to maintain unique relationships between orders and products, while product reviews and website activity are stored in their tables to prevent cluttering the customer or product tables.

Marketing campaigns, including types and engagement metrics, are managed in separate tables, with cities and campaign types providing flexibility in campaign management. The location-based tables (country, city, and region) maintain consistency and avoid duplication through foreign key constraints.

Overall, the schema separates data into distinct tables, linking them logically with foreign keys. This structure reduces redundancy, enhances data integrity, and improves scalability and efficiency by minimizing the risk of anomalies (such as insertion, deletion, or update errors).

4. Synthetic Data Generation

For the customer database, we utilised Python along with Faker and Generative AI to generate a sample dataset of 698 customers all located in the UK. We ensured realistic dates of birth by confirming all customers were above the legal age of 18 in the UK. Initially, we included both "County" and "City" fields, but because adding a "County" would cause invalid postcodes, we replaced it with "Regions" and "Cities" instead. To reflect the distribution of customers, we focused on more populous UK cities for postcodes, while also incorporating smaller cities to increase diversity and data quality. The postcode format was restricted to 5-7 characters. For email addresses, we used the domains of the four major providers (Gmail, Outlook, Hotmail, and Yahoo), and customer usernames were derived from their first and last names. UK-specific telephone numbers were generated to follow the official format.

For the Cities and Regions tables, we employed Generative AI to create datasets, generating 111 cities, each mapped to their corresponding region in the UK. These city IDs were kept consistent with those in the Customers table to ensure proper alignment.

In the product database, we defined several product categories, including Miscellaneous, Health & Beauty, Food & Beverages, Home & Decor, Electronics, Toys & Gifts, Art & Craft, Clothes & Accessories, Home & Kitchen, Footwear, and Music Equipment. We then used ChatGPT to generate 19,975 product records corresponding to each of these categories.

For the Orders table, Generative AI was used to generate 2,000 unique orders while maintaining consistency across related tables. The order_id values matched order_details.csv, and customer_id was sourced from customers.csv. We created an order details table with 6,949 records. Although there are 2,000 unique order IDs, each order can include multiple products, which is why the number of rows in the order details table exceeds the number of unique orders. In the order details table, each order is linked to 2 to 5 products from the products table, with a random quantity between 1 and 9. Order dates were spread across 2024, with 70% of customers making a second purchase between January and March, and another between September and December, to reflect return behavior after marketing campaigns launched in August. The amount for each order was calculated from the quantity and selling price, while discount_id was applied to only 9% of orders before July 31st, and to 90% of orders after August 1st.

For the Discounts table, we generated 111 records, each with its own discount_id. We also created 20 different discount types, allowing for various promotional strategies to be applied across different orders.

Generative AI was also used to generate the marketing_campaigns, campaign_engagement, and website_activity tables. In the marketing_campaigns table, we generated 111 records to ensure that each city is covered by one marketing campaign. For the campaign_engagement table, 15 records were created, one for each campaign, tracking the click rates and reach for each. The website_activity table generated 700 records, linking customer IDs to the Customers table and category IDs to the Category Data table. Date and time fields were distributed across 2024, with random values for time spent and orders placed, simulating browsing behavior and purchases.

Finally, the product review table contained 1,000 records, capturing customer feedback on specific products. The country table contains only 1 record, as the e-commerce platform currently operates only in the UK.

A sample of each generated dataset is provided in Appendix C, while the Python code used to generate the data with the Faker library is included in Appendix D.

5. Transforming Data into Actionable Insights

The following insights were derived from the proposed database structure for Cartify. Each metric highlights a specific aspect of Cartify's operations and customer interactions. The SQL code used to calculate these insights, along with others shown in the presentation, can be found in Appendix E.

5.1. Customer Retention Rate

A customer retention rate of 78% suggests that Cartify has been highly successful in retaining a significant portion of its customer base. This result demonstrates the effectiveness of Cartify’s marketing campaigns in engaging customers and addressing the initial problem of high churn rates that the company faced at the beginning of 2024. Thanks to the proposed database design, which allows for better tracking and analysis of customer behavior, Cartify can now identify and target customers more effectively, leading to stronger customer retention. However, the percentage of acquisition of new customers is low, at 22%, which indicates that while Cartify is now successful in retaining existing customers, there may be opportunities to focus more on acquiring new customers to further expand its customer base.

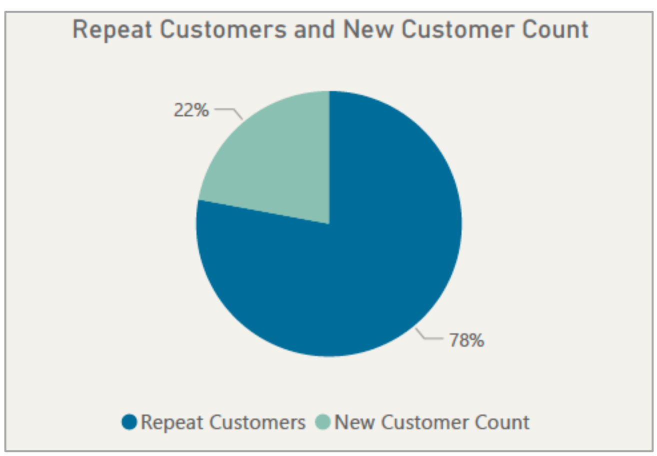

Figure 2: Distribution of Repeat vs. New Customers in Cartify

5.2. Order Abandonment Rate

The order abandonment rate of 45% indicates that nearly half of the customers who visit the website choose not to complete their purchases. This could point to potential issues in the customer journey, such as a complicated checkout process, concerns about pricing, or high shipping fees. Addressing these factors and improving the checkout process could significantly reduce this abandonment rate and help convert more visitors into buyers.

5.3. Average Order Count

With an average of 3.07 orders per customer, Cartify sees a relatively healthy frequency of repeat purchases. This suggests that customers are coming back multiple times to buy, but there's still room for improvement. Increasing the average order count could be achieved by enhancing customer engagement, offering incentives for repeat purchases, or encouraging customers to purchase more per transaction.

5.4. Campaign ROI

A campaign ROI of 6.83% means that for every pound spent on marketing campaigns, Cartify generates approximately 6.83% in profit. While this indicates a positive return, it's relatively modest. The company might want to optimise its campaigns further by refining targeting, improving ad content, or adjusting the budget allocation to increase the effectiveness of marketing spend.

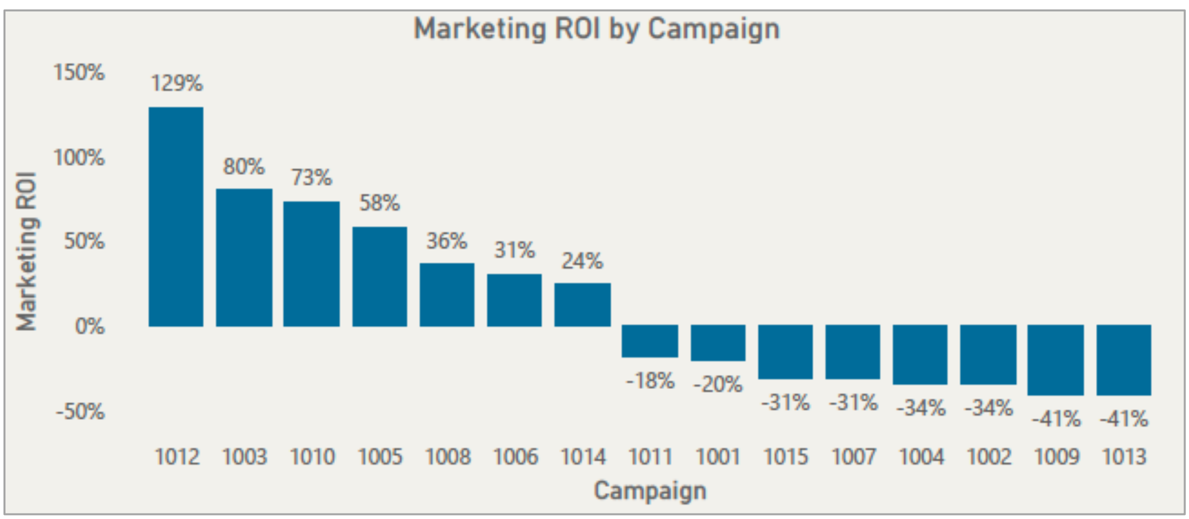

Figure 3: Marketing ROI Performance Across Cartify Campaigns

5.5. Average Revenue Generated by Campaign

On average, each campaign generates £83,724.25 in revenue. This metric shows that Cartify's marketing campaigns are able to drive significant sales. Additionally, the performance of each individual campaign was analysed, providing deeper insights into which campaigns have been the most successful. These insights can be visualised in the following graph, which illustrates the revenue generated by each campaign. By identifying high-performing campaigns, Cartify can optimise future marketing efforts and allocate resources to the most effective strategies.

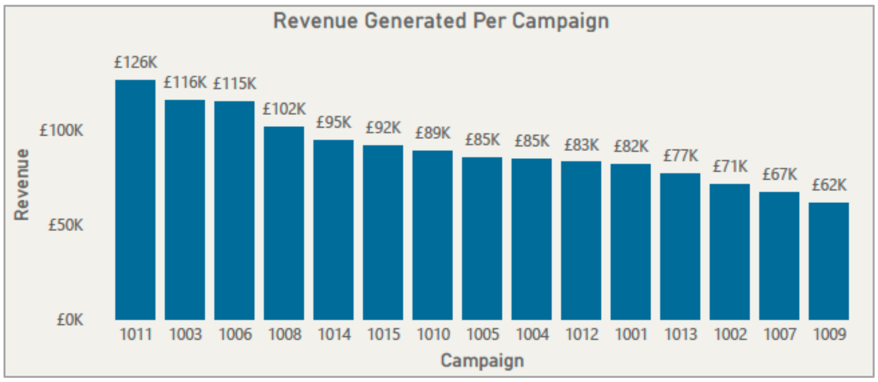

Figure 4: Average Revenue Generated Per Campaign

5.6. Cost Per Click

A cost per click (CPC) of £3.60 means Cartify is paying an average of £3.60 for each customer who clicks through its marketing campaigns. This figure is reasonable for many industries, but it's important to continuously monitor CPC to ensure it stays efficient in terms of generating high-quality traffic. Lowering the CPC through better targeting or ad optimisation could help increase the profitability of marketing efforts.

5.7. Sales Across Months (With and Without Discounts)

Below is a breakdown of sales across months, comparing revenue from purchases made with a discount versus those made without a discount:

From the data, it is evident that months with discounts tend to show significant variations in sales, with some months (like August, September, October, and November) showing a sharp spike in discount-related sales. However, sales without discounts consistently remain high, particularly in months like January and March, indicating that Cartify generates significant revenue even without relying heavily on discounts.

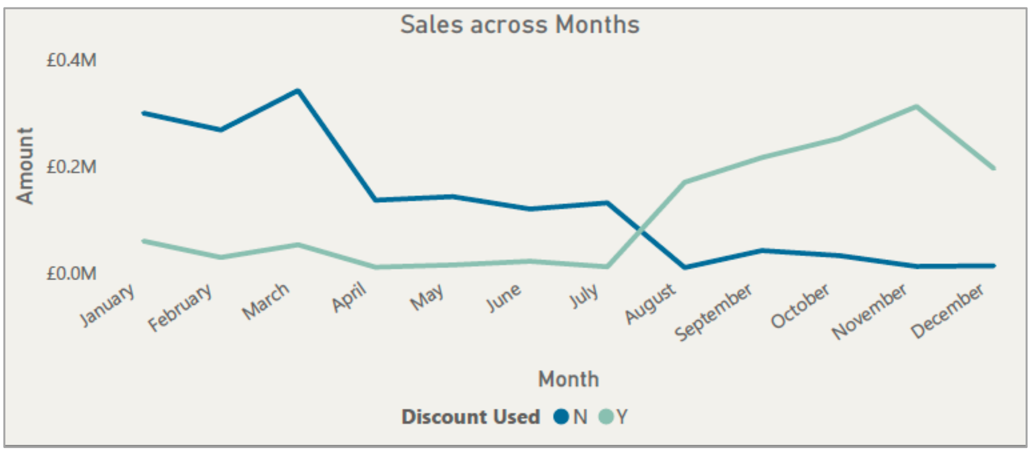

Figure 5: Sales Trends Across Months with and without Discounts

Appendices

Appendix A: Cardinalities between the entities

Table A: Cardinalities between the entities:

Tables	Cardinality	Explanation
customer_table and orders_table	One-to-Many	One customer can have many orders
customer_table and product_review_table	One-to-Many	One customer can write many reviews
customer_table and website_activity_table	One-to-Many	One customer can have many website activities
customer_table and city_table	Many-to-One	Many customers can be in the same city
customer_table and country_table	Many-to-One	Many customers can be in the same country
orders_table and order_details_table	One-to-Many	One order can have many order details
orders_table and discount_table	Many-to-One	Many orders can use the same discount
orders_details_table and products_table	Many-to-One	Many order details can refer to one product
products_table and product_review_table	One-to-Many	One product can have many reviews
products_table and category_table	Many-to-One	Many products can belong to one category
category_table and website_activity_table	One-to-Many	One category can be viewed in many activities
discount_table and discount_type_table	Many-to-One	Many discounts can be of one type
campaign_type_table and marketing_campaigns_table	One-to-Many	One campaign type can be used in many campaigns
marketing_campaigns_table and city_table	One-to-Many	One campaign can target many cities
marketing_campaigns_table and campaign_engagment_table	Many-to-One	Multiple rows in marketing_campaign_table correspond to a single row in campaign_engagement_table based on campaign_id. campaign_id in marketing_campaigns_table repeat several times because the primary key there is a composite key (campaign_id, city_id)
city_table and region_table	Many-to-One	Many cities can belong to one region
campaign_engagement and discount table	One-to-Many	One campaign can have many discounts.

Appendix B: SQL of the database definition and table creation

STEP 1: CREATE the SQLite database:

```
import sqlite3

# Establish a connection to the database file (or create it if it
doesn't exist)
conn = sqlite3.connect('Cartify.db')
cursor = conn.cursor()

# Customer Table
cursor.execute('''
CREATE TABLE customer_table (
    customer_id TEXT PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    gender TEXT NOT NULL,
    date_of_birth DATE NOT NULL,
    email TEXT UNIQUE NOT NULL,
    phone_number INTEGER UNIQUE NOT NULL,
    address TEXT NOT NULL,
    city_id TEXT NOT NULL,
    postcode TEXT NOT NULL,
    country_id TEXT NOT NULL,
    FOREIGN KEY (city_id) REFERENCES city_table(city_id) ON DELETE
CASCADE
    FOREIGN KEY (country_id) REFERENCES country_table(country_id) ON
DELETE CASCADE
);
''')

# Orders Table
cursor.execute('''
CREATE TABLE orders_table (
    order_id TEXT PRIMARY KEY,
    customer_id TEXT NOT NULL,
    order_date DATE NOT NULL,
    discount_id INTEGER,
    amount REAL NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customer_table(customer_id)
ON DELETE CASCADE,
    FOREIGN KEY (discount_id) REFERENCES discount_table(discount_id)
ON DELETE CASCADE
```

```
);
''')

# Order Details Table
cursor.execute('''
CREATE TABLE order_details_table (
    order_id TEXT,
    product_id TEXT,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders_table(order_id) ON DELETE
CASCADE,
    FOREIGN KEY (product_id) REFERENCES products_table(product_id) ON
DELETE CASCADE
);
''')

# Products Table
cursor.execute('''
CREATE TABLE products_table (
    product_id TEXT PRIMARY KEY,
    product_name TEXT NOT NULL,
    selling_price REAL NOT NULL,
    category_id TEXT NOT NULL,
    cost_price REAL NOT NULL,
    stock INTEGER NOT NULL,
    FOREIGN KEY (category_id) REFERENCES category_table(category_id)
ON DELETE CASCADE
);
''')

# Product Review Table
cursor.execute('''
CREATE TABLE product_review_table (
    review_id INTEGER PRIMARY KEY AUTOINCREMENT,
    customer_id TEXT NOT NULL,
    product_id TEXT NOT NULL,
    rating INTEGER CHECK (rating BETWEEN 1 AND 5) NOT NULL,
    date DATE NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customer_table(customer_id)
ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products_table(product_id) ON
DELETE CASCADE
);
''')
```

```
# Category Table
cursor.execute('''
CREATE TABLE category_table (
    category_id TEXT PRIMARY KEY,
    category_name TEXT UNIQUE NOT NULL
);
''')

# Website Activity Table
cursor.execute('''
CREATE TABLE website_activity_table (
    activity_id TEXT PRIMARY KEY,
    category_id TEXT NOT NULL,
    date DATE NOT NULL,
    time TIME NOT NULL,
    customer_id TEXT NOT NULL,
    time_spent REAL NOT NULL,
    order_placed TEXT CHECK (order_placed IN ('Y', 'N')) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customer_table(customer_id)
ON DELETE CASCADE,
    FOREIGN KEY (category_id) REFERENCES category_table(category_id)
ON DELETE CASCADE
);
''')

# Discount Table
cursor.execute('''
CREATE TABLE discount_table (
    discount_id INTEGER PRIMARY KEY AUTOINCREMENT,
    campaign_id INTEGER NOT NULL,
    discount_type_id TEXT NOT NULL,
    coupon_code TEXT UNIQUE NOT NULL,
    FOREIGN KEY (discount_type_id) REFERENCES
discount_type_table(discount_type_id) ON DELETE CASCADE,
    FOREIGN KEY (campaign_id) REFERENCES
campaign_engagement_table(campaign_id) ON DELETE CASCADE
);
''')

# Discount Type Table
cursor.execute('''
CREATE TABLE discount_type_table (
    discount_type_id TEXT PRIMARY KEY,
    type_of_discount TEXT UNIQUE NOT NULL
```

```
);
''')

# Campaign Type Table
cursor.execute('''
CREATE TABLE campaign_type_table (
    campaign_type_id INTEGER PRIMARY KEY AUTOINCREMENT,
    platform TEXT UNIQUE NOT NULL
);
''')

# Marketing Campaigns Table
cursor.execute('''
CREATE TABLE marketing_campaigns_table (
    campaign_id INTEGER NOT NULL,
    city_id TEXT NOT NULL,
    campaign_type_id INTEGER NOT NULL,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    cost REAL NOT NULL,
    PRIMARY KEY (campaign_id, city_id),
    FOREIGN KEY (campaign_type_id) REFERENCES
campaign_type_table(campaign_type_id) ON DELETE CASCADE,
    FOREIGN KEY (city_id) REFERENCES city_table(city_id) ON DELETE
CASCADE
);
''')

# Campaign Engagement Table
cursor.execute('''
CREATE TABLE campaign_engagement_table (
    campaign_id INTEGER PRIMARY KEY AUTOINCREMENT,
    click_rate REAL NOT NULL,
    reach INTEGER NOT NULL,
    FOREIGN KEY (campaign_id) REFERENCES
marketing_campaigns_table(campaign_id) ON DELETE CASCADE
);
''')

# City Table
cursor.execute('''
CREATE TABLE city_table (
    city_id TEXT PRIMARY KEY,
    city TEXT UNIQUE NOT NULL,
```

```
    region_id TEXT NOT NULL,
    FOREIGN KEY (region_id) REFERENCES region_table(region_id) ON
DELETE CASCADE
);
''')

# Region Table
cursor.execute('''
CREATE TABLE region_table (
    region_id TEXT PRIMARY KEY,
    region TEXT UNIQUE NOT NULL
);
''')

# Country Table
cursor.execute('''
CREATE TABLE country_table (
    country_id TEXT PRIMARY KEY,
    country TEXT UNIQUE NOT NULL
);
''')

# Save the changes to the database
conn.commit()

print("Database and tables created successfully!")
```

STEP 2: Check Tables Created:

```
cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
tables = cursor.fetchall()

for table_name in tables:
    print(f"Table: {table_name[0]}")
    cursor.execute(f"PRAGMA table_info({table_name[0]});")
    columns = cursor.fetchall()
    for col in columns:
        print(f"  Column: {col[1]}, Type: {col[2]}, NotNull: {col[3]},
DefaultVal: {col[4]}, PrimaryKey: {col[5]}")
    print("-" * 20)
```

STEP 3: Load CSV files into the database tables:

```
import csv

def import_csv_to_table(csv_file, table_name):
    #opens the file aas read only 'r', doesn't allow the origianl csv
to be changed.
    with open(csv_file, 'r', encoding='utf-8') as file:
        csv_reader = csv.reader(file, delimiter=';')
        next(csv_reader)  # Skip header row if present
        for row in csv_reader:
            #? creates a placeholder for each column in the CSV file.
['?','?','?'] - Join makes it a string so it can then be inserted.
            # use of the '?' reduce risk of SQL injection
            placeholders = ', '.join(['?' for _ in row])
            #Assumes that the CSV and table have the same structure
(this could be an issue) Would have to specify column names if
different.
            sql = f"INSERT INTO {table_name} VALUES ({placeholders})"
            cursor.execute(sql, row)

# Import data from CSV files into the relevant table - Student_Table
goes into student table.  teh import_csv_to_table is the function,
passing the two values across.
try:
    import_csv_to_table('customers.csv', 'customer_table')
    import_csv_to_table('orders.csv', 'orders_table')
    import_csv_to_table('orders_details.csv', 'order_details_table')
    import_csv_to_table('product_data.csv', 'products_table')
    import_csv_to_table('products_review.csv', 'product_review_table')
    import_csv_to_table('marketing_campaigns.csv',
'marketing_campaigns_table')
    import_csv_to_table('campaign_type.csv', 'campaign_type_table')
    import_csv_to_table('discount.csv', 'discount_table')
    import_csv_to_table('discount_types.csv', 'discount_type_table')
    import_csv_to_table('campaign_engagement.csv',
'campaign_engagement_table')
    import_csv_to_table('website_activity.csv',
'website_activity_table')
    import_csv_to_table('category_data.csv', 'category_table')
    import_csv_to_table('cities.csv', 'city_table')
    import_csv_to_table('region.csv', 'region_table')
    import_csv_to_table('country.csv', 'country_table')
    conn.commit()
```

```
    print("Data imported successfully!")
except Exception as e:
    print(f"An error occurred: {e}")
    conn.rollback()  # Rollback changes if an error occurred
```

STEP 4: Check Data has loaded:

```
import pandas as pd
# Query each table and load into pandas DataFrames
customer_df = pd.read_sql_query("SELECT * FROM customer_table", conn)
orders_df = pd.read_sql_query("SELECT * FROM orders_table", conn)
order_details_df = pd.read_sql_query("SELECT * FROM 
order_details_table", conn)
products_df = pd.read_sql_query("SELECT * FROM products_table", conn)
product_review_df = pd.read_sql_query("SELECT * FROM 
product_review_table", conn)
marketing_campaigns_df = pd.read_sql_query("SELECT * FROM 
marketing_campaigns_table", conn)
campaign_type_df = pd.read_sql_query("SELECT * FROM 
campaign_type_table", conn)
discount_df = pd.read_sql_query("SELECT * FROM discount_table", conn)
discount_types_df = pd.read_sql_query("SELECT * FROM 
discount_type_table", conn)
campaign_engagement_df = pd.read_sql_query("SELECT * FROM 
campaign_engagement_table", conn)
website_activity_df = pd.read_sql_query("SELECT * FROM 
website_activity_table", conn)
category_df = pd.read_sql_query("SELECT * FROM category_table", conn)
city_df = pd.read_sql_query("SELECT * FROM city_table", conn)
region_df = pd.read_sql_query("SELECT * FROM region_table", conn)
country_df = pd.read_sql_query("SELECT * FROM country_table", conn)

# Show the first 5 lines of each DataFrame
print("Customer Table:")
print(customer_df.head(5))
print("\nOrders Table:")
print(orders_df.head(5))
print("\nOrder Details Table:")
print(order_details_df.head(5))
print("\nProducts Table:")
print(products_df.head(5))
print("\nProduct Review Table:")
print(product_review_df.head(5))
print("\nMarketing Campaigns Table:")
```

```
print(marketing_campaigns_df.head(5))
print("\nCampaign Type Table:")
print(campaign_type_df.head(5))
print("\nDiscount Table:")
print(discount_df.head(5))
print("\nDiscount Types Table:")
print(discount_types_df.head(5))
print("\nCampaign Engagement Table:")
print(campaign_engagement_df.head(5))
print("\nWebsite Activity Table:")
print(website_activity_df.head(5))
print("\nCategory Table:")
print(category_df.head(5))
print("\nCity Table:")
print(city_df.head(5))
print("\nRegion Table:")
print(region_df.head(5))
print("\nCountry Table:")
print(country_df.head(5))
```

Appendix C: Sample of each table generated

Table C1: Customers Table:

customer_id	first_name	last_name	gender	date_of_birth	email	phone_number	address	city_id	postcode	country_id
CUST97034	Victoria	Hill	Female	1994-06-09	victoria.hill@yahoo.com	7957263192	944 Williams circle	Y2VK	E1 0AB	C001
CUST78652	Ethan	Sharp	Male	2000-06-14	ethan.sharp@outlook.com	7808474504	59 Clive lock	B5VW	M2 1BC	C001
CUST62025	Charlotte	Browne	Female	1982-10-15	charlotte.browne@hotmail.com	7972272513	15 Derek land	XU5I	B3 2CD	C001
CUST38259	Connor	Smith	Male	1986-08-11	connor.smith@gmail.com	7881519870	327 Butler parks	30AN	G4 3DE	C001
CUST53488	Holly	Evans	Female	1997-12-07	holly.evans@hotmail.com	7658408505	390 Lynda bypass	KJRQ	L5 4EF	C001
CUST71644	Linda	Chambers	Female	2003-11-09	linda.chambers@hotmail.com	7582935175	745 Mitchell fall	92R4	LS6 5FG	C001
CUST77363	Erin	Davie	Female	1983-11-23	erin.davie@yahoo.com	7656833626	203 Mohammad rue	PYH6	EH7 6GH	C001
CUST64028	Noah	Owens	Male	1999-12-22	noah.owens@gmail.com	7806311711	354 Simpson walks	7W41	BS8 7HI	C001
CUST93065	Samuel	Buckley	Male	1985-12-03	samuel.buckley@outlook.com	7524886106	921 Pickering point	C6WB	S9 8IJ	C001
CUST24370	Edward	Barnes	Male	1993-01-21	edward.barnes@hotmail.com	7788267644	974 Richardson motorway	XBR7	NG1 9JK	C001
CUST73374	David	Brown	Male	1979-02-06	david.brown@yahoo.com	7592166675	772 Davison way	Y2VK	E2 0KL	C001
CUST72814	Phoebe	Watts	Female	1998-08-22	phoebe.watts@hotmail.com	7154551307	804 Jackson branch	B5VW	M3 1LM	C001

Table C2: Orders Table:

order_id	customer_id	order_date	discount_id	amount
ZYY98	CUST51271	2024-12-13	670	163.17
ZYGRH	CUST73447	2024-09-09	698	9643.52
ZXYG6	CUST72546	2024-07-22		20870.11
ZXXOY	CUST47951	2024-04-25		637.78
ZXWHD	CUST92125	2024-02-05		1016.39
ZXUE3	CUST17932	2024-11-28	555	579.93
ZXN5I	CUST58041	2024-12-01	318	2588.19
ZW9A6	CUST38643	2024-03-06		146.79
ZUVAQ	CUST27097	2024-03-12		612.24
ZUOBF	CUST74378	2024-08-11	223	4004.0
ZTX1Z	CUST24370	2024-02-23		484.86
ZT7H1	CUST71864	2024-03-05		4492.06

Table C3: Product Table:

product_id	product_name	selling_price	category_id	cost_price	stock
B00XLHSZ74	Aqualona Premium	26.99	E387	12.28	46
B07Z1YTD9R	Leone Leggins Donna	25.49	E387	11.41	28
B0BHJ1MNY4	Skang Gilet for Women UK	37.59	E387	17.53	65
B0CGX6XR41	Women's Loose Sweaters O-Neck	49.43	E387	23.3	89
B01BL8720K	Mustela PN	9.77	TG47	4.3	76
B07PDJ9JFF	Echo Flex Voice control smart	9.99	E387	4.65	83
B016KN36QA	Craghoppers Mens Fleece	15.99	E387	5.71	44
B003U77IG4	Lime Fusion Finnish Vodka 70cl Bottle	16.56	ZP25	7.38	69
B08H81BRKX	Deconovo 2 Pack Burgundy Velvet	4.49	O4BF	1.56	93
B097C3B6S8	Sink Tap Rotatable Water	5.99	E387	2.93	16
B0B27XM5PK	Xinxuan 5/10-PCS Trampoline	3.79	E387	1.6	33
B0BXH36SB2	Ring Light Kit	117.54	7EE3	48.7	30

Table C4: Category Table:

category_id	category_name
E387	Miscellaneous
TG47	Health & Beauty
ZP25	Food & Beverages
O4BF	Home & Decor
7EE3	Electronics
C3E5	Toys & Gift
A2CA	Art & Craft
7FE3	Clothes & Accessories
715T	Home & Kitchen
0DCP	Footwear
2D5Y	Music Equipment

Table C5: Orders Details Table:

order_id	product_id	quantity
80GPC	B0C5LTPT2Q	8
80GPC	B0CKLKWV6T	8
80GPC	B09WMN46JS	5
YSC9L	B0992XK6M8	9
YSC9L	B09QMN67V2	2
YSC9L	B0CB9H1511	1
YSC9L	B0C4BJSQGB	8
YSC9L	B0013ISRFI	4
F95CU	B07FFVTPCP	2
F95CU	B00DHMICOE	5
F95CU	B0CG9XTK69	7
F95CU	B0CDRT77WH	1
NHBUJ	B07S1V1YSL	8
NHBUJ	B09ZV84XDM	1
9Q9DH	B08SW52KBD	5
9Q9DH	B0BHSQ4F2Q	4
9Q9DH	B0BMVZGWG8	3
OSN1F	B09N7MKXQP	6
OSN1F	B09CDNK1LX	9

Table C6: Discount Type Table:

discount_type_id	type_of_discount
D001	Exclusive App-Only Discount
D002	Buy One Get One Free
D003	Holiday Sale
D004	Flash Sale
D005	Seasonal Discount
D006	Loyalty Program Discount
D007	First Purchase Discount
D008	Student Discount
D009	Senior Citizen Discount
D010	Referral Discount
D011	Bulk Purchase Discount

Table C7: Discount Table:

discount_id	campaign_id	discount_type_id	coupon_code
194	1001	D010	CCTSX
890	1002	D009	4OWWB
580	1003	D001	KK5UU
314	1004	D010	0HTJN
250	1005	D011	HIZZP
871	1006	D007	R1V7Z
778	1007	D006	2XAHC
133	1008	D020	YTRPX
722	1009	D010	T8F0N
195	1010	D010	R0DCP
763	1011	D003	J0KOO

Table C8: Product Review Table:

review_id	customer_id	product_id	rating	date
261390	CUST35004	B09B9HM391	2	2024-05-11
303495	CUST14987	B09DD5QNGG	5	2024-10-30
110593	CUST34279	B0CFXB11RJ	2	2024-03-07
824168	CUST28877	B08MFQJ8CH	5	2024-05-05
422354	CUST25063	B08X15S4VH	5	2024-02-17
217042	CUST34002	B0CGNSTT7Y	3	2024-09-23
173287	CUST14987	B09CKCT8BS	2	2024-07-07
785962	CUST70883	B07VT7PKVY	5	2024-07-09
774832	CUST17373	B09FS4QSG1	3	2024-11-16
588303	CUST60806	B0C6ZQRHSV	2	2024-08-31
172209	CUST26924	B00SBDUYHC	3	2024-03-14
558690	CUST83587	B0BDRQLNR9	2	2024-03-04

Table C9: City Table:

city_id	city	region_id
KJRQ	Liverpool	L3D2X
Y2VK	London	G7B9Q
B5VW	Manchester	L3D2X
7W41	Bristol	W4K2T
LR65	Milton keynes	N2C5R
30AN	Glasgow	NIZUV
8GPB	Chesterfield	P7X8Z
XU5I	Birmingham	Q5B1W
18KT	North warwickshire	Q5B1W
285N	North devon	W4K2T
30AN	Glasgow	NIZUV
33ZJ	Aberdeen city	18Y3Q
3H9C	Wyre	L3D2X
3UIK	Greenwich	G7B9Q

Table C10: Region Table:

region_id	region
G7B9Q	Greater London
L3D2X	North West England
Q5B1W	West Midlands
NIZUV	Glasgow & Argyll
R6Q5Y	Yorkshire and the Humber
MWKHZ	Lothian
W4K2T	South West England
P7X8Z	East Midlands
N2C5R	South East England
V8T3C	North East England
428JC	Central & Fife

Table C11: Country Table:

country_id	name
C001	United Kingdom

Table C12: Marketing Campaign Table:

campaign_id	city_id	campaign_type_id	start_date	end_date	cost
1001	Y2VK	5	2024-08-01	2024-08-15	6000
1002	B5VW	3	2024-08-11	2024-08-25	7500
1003	XU5I	7	2024-08-21	2024-09-04	4500
1004	30AN	2	2024-09-02	2024-09-16	9000
1005	KJRQ	6	2024-09-12	2024-09-26	3500
1006	92R4	1	2024-09-22	2024-10-06	6500
1007	PYH6	4	2024-10-03	2024-10-17	8000
1008	7W41	3	2024-10-13	2024-10-27	5500
1009	C6WB	7	2024-10-23	2024-11-06	7000
1010	XBR7	5	2024-11-04	2024-11-18	4000
1011	LUVM	1	2024-11-14	2024-11-28	8500

Table C13: Campaign Engagement Table:

campaign_id	click_rate	reach
1001	2.5	200000
1002	3.4	350000
1003	4.7	480000
1004	5.2	600000
1005	6.3	720000
1006	2.6	250000
1007	3.8	400000
1008	4.1	520000
1009	5.6	650000
1010	2.9	300000
1011	3.2	420000

Table C14: Campaign Type Table:

campaign_type_id	platform
1	Instagram
2	Facebook
3	TikTok
4	Email
5	Google Ads
6	Youtube
7	X

Table C15: Website Activity Table:

activity_id	category_id	date	time	customer_id	time_spent	order_placed
X7KIY	E387	2024-07-30	14:19:45	CUST84253	6.8	N
XFLYV	TG47	2024-06-22	12:28:55	CUST82606	9.0	N
9F4D8	A2CA	2024-05-01	21:37:43	CUST72915	7.7	Y
MZ1X3	7FE3	2024-03-23	08:44:36	CUST62592	6.1	N
FWXLE	A2CA	2024-04-03	13:55:59	CUST71535	8.8	N
5ER0J	0DCP	2024-06-25	19:00:15	CUST32519	5.8	Y
MRC1D	TG47	2024-03-09	01:13:58	CUST49436	8.0	N
VRT2L	TG47	2024-05-26	09:06:45	CUST56641	5.2	Y
1ECMT	7EE3	2024-07-27	15:09:11	CUST23368	7.0	N
N8A2D	715T	2024-06-29	22:25:40	CUST36714	1.6	N
QHVW1	7EE3	2024-04-23	12:31:56	CUST71075	6.2	N

Appendix D: Python code using the Faker library

```
pip install requests faker pandas
```

```
import requests
import pandas as pd
import random
import string
import re
import datetime
from faker import Faker

# Initialize Faker for UK locale
fake = Faker("en_GB")

# List of 10 major cities in the UK (Most customers will be from
these)
big_cities = ["London", "Manchester", "Birmingham", "Glasgow",
"Liverpool",
              "Leeds", "Edinburgh", "Bristol", "Sheffield",
"Nottingham"]

# Expanding the list of first names for Male, Female, and Other
categories
male_names = [
    "James", "John", "Robert", "Michael", "William", "David",
"Richard", "Joseph", "Thomas", "Charles",
    "Daniel", "Matthew", "Luke", "Edward", "Harry", "George", "Jack",
"Oliver", "Henry", "Samuel",
    "Jake", "Nathan", "Lewis", "Ryan", "Oscar", "Alex", "Ethan",
"Liam", "Benjamin", "Joshua",
    "Noah", "Charlie", "Adam", "Connor", "Zachary", "Harrison",
"Toby", "Callum", "Jayden", "Arthur"
]

female_names = [
    "Mary", "Patricia", "Jennifer", "Linda", "Elizabeth", "Barbara",
"Susan", "Jessica", "Sarah", "Karen",
    "Emily", "Hannah", "Charlotte", "Sophie", "Olivia", "Isabella",
"Amelia", "Megan", "Abigail", "Emma",
    "Lucy", "Katie", "Ellie", "Lauren", "Rebecca", "Holly", "Jasmine",
"Eleanor", "Phoebe", "Freya",
    "Madison", "Alice", "Isla", "Anna", "Mia", "Amber", "Daisy",
"Harriet", "Erin", "Victoria"
]
```

```
# Other gender-neutral names (Only 4% of dataset)
other_names = [
    "Taylor", "Jordan", "Morgan", "Casey", "Jamie", "Alexis", "Robin",
"Skyler", "Avery", "Riley",
    "Finley", "Sasha", "Dakota", "Phoenix", "Quinn", "Eden", "River",
"Rowan", "Harper", "Indigo"
]

# Merging all names into one dictionary with category labels
all_names = {
    "Male": male_names,
    "Female": female_names,
    "Other": other_names
}

# Adjusted gender distribution: 53% Female, 43% Male, 4% Other
name_distribution = ["Female"] * 53 + ["Male"] * 43 + ["Other"] * 4

# Function to generate a name and assign gender accordingly
def generate_name_and_gender():
    gender = random.choice(name_distribution)  # Assign gender based
on adjusted distribution
    first_name = random.choice(all_names[gender])  # Select a name
from the respective category
    return first_name, gender

# Function to generate a unique customer ID (CUST followed by 5
digits)
def generate_customer_id():
    return f"CUST{random.randint(10000, 99999)}"

# Function to fetch UK addresses, prioritizing big cities
def get_address_details():
    if random.random() < 0.75:  # 75% chance to choose a big city
        city = random.choice(big_cities)
        return {
            "postcode": f"{city[:2].upper()}{random.randint(1, 9)}
{random.choice('ABCDEFGHJKLMNPRSTUVWXYZ')}{random.randint(1,
9)}{random.choice('ABCDEFGHJKLMNPRSTUVWXYZ')}",
            "country": "United Kingdom",
            "city": city,
            "street": fake.street_name()
        }
    else:
```

```
        response =
requests.get("https://api.postcodes.io/random/postcodes")
        if response.status_code == 200:
            data = response.json()["result"]
            return {
                "postcode": data["postcode"],
                "country": "United Kingdom",
                "city": data["admin_district"],
                "street": fake.street_name()
            }
    return None

# Function to generate a random UK mobile number
def generate_random_uk_mobile_number():
    return f"07{random.randint(100000000, 999999999)}"

# Function to validate UK phone number format (Must start with 07 and
be 11 digits long)
def validate_uk_mobile_number(number):
    return re.match(r"^07\d{9}$", number) is not None  # Checks for
correct format

# Function to validate email format
def validate_email(email):
    email_regex = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]+$"
    allowed_tlds = (".com", ".co.uk", ".org", ".net")  # Only valid
TLDs

    if not re.match(email_regex, email) or not
email.endswith(allowed_tlds):
        return None
    return email

# Function to generate a realistic date of birth with most in 1985-
2005 range
def generate_dob():
    if random.random() < 0.7:  # 70% chance for DOB between 1985 and
2005
        year = random.randint(1985, 2005)
    else:  # 30% chance for DOB between 1965 and 1984 or 2006-2010
        year = random.choice(range(1965, 1985)) if random.random() <
0.5 else random.choice(range(2006, 2010))
    month = random.randint(1, 12)
    day = random.randint(1, 28)  # Keep within valid range for all
months
```

```
    return datetime.date(year, month, day).strftime("%Y-%m-%d")

# Generate customer data with at least 700 unique records
num_rows = 700
customer_data = set()

while len(customer_data) < num_rows:
    customer_id = generate_customer_id()
    first_name, gender = generate_name_and_gender()  # Generate name
and gender
    last_name = fake.last_name()

    email =
validate_email(f"{first_name.lower()}.{last_name.lower()}@{random.choi
ce(['gmail.com', 'yahoo.com', 'outlook.com', 'hotmail.com'])}")

    if not email:
        continue  # Skip invalid emails

    phone = None
    while not phone:
        random_phone = generate_random_uk_mobile_number()
        if validate_uk_mobile_number(random_phone):  # Check phone
format
            phone = random_phone

    address_details = get_address_details()
    if not address_details:
        continue  # Skip if address details are not found

    dob = generate_dob()  # Generate Date of Birth

    customer_data.add((
        customer_id, gender, first_name, last_name, dob, email, phone,
        address_details["street"], address_details["city"],
address_details["postcode"], address_details["country"]
    ))

# Convert set to DataFrame
customer_df = pd.DataFrame(list(customer_data), columns=[
    "CustomerID", "Gender", "First Name", "Last Name", "Date of
Birth", "Email", "Phone",
    "Street", "City", "Postcode", "Country"
])
```

```python
# Save the dataset to CSV
customer_df.to_csv("customer_uk_database.csv", index=False)

print("CSV file 'customer_uk_database.csv' generated successfully!")
```

Appendix E: SQL code of the different insights

Code E1: <u>Customer Retention Rate SQL</u>:

```
Customer_Retention_df = pd.read_sql_query("""
WITH OrderCounts AS (
    SELECT customer_id, COUNT(*) AS OrderCount
    FROM orders_table
    GROUP BY customer_id
)
SELECT
    COUNT(DISTINCT c.customer_id) AS total_customers,
    COUNT(DISTINCT CASE WHEN oc.OrderCount >= 2 THEN c.customer_id
END)  AS repeating_customers,
    ROUND(
        CASE
            WHEN COUNT(DISTINCT c.customer_id) = 0 THEN 0
            ELSE (COUNT(DISTINCT CASE WHEN oc.OrderCount >= 2 THEN
c.customer_id END) * 100.0) / COUNT(DISTINCT c.customer_id)
        END, 0
    ) || '%' AS RepeatCustomersPercentage
FROM customer_table c
LEFT JOIN OrderCounts oc
  ON c.customer_id = oc.customer_id;

""", conn)

print(Customer_Retention_df)
```

```
   total_customers  repeating_customers RepeatCustomersPercentage
0              698                  544                     78.0%
```

Code E2: **Order Abandonment Rate:**

```
Order_Abandonment_Rate_df = pd.read_sql_query("""
SELECT
    t.total_customers,
    n.notbuying_customers,
    ROUND(
      CASE
        WHEN t.total_customers = 0 THEN 0
        ELSE (n.notbuying_customers * 100.0) / t.total_customers
      END, 0
    ) || '%' AS order_abandonment_rate
FROM
    (SELECT COUNT(order_placed) AS total_customers FROM
website_activity_table) AS t
JOIN
    (SELECT COUNT(order_placed) AS notbuying_customers
     FROM website_activity_table
     WHERE order_placed = 'N') AS n;
""", conn)

print(Order_Abandonment_Rate_df)
```

```
   total_customers  notbuying_customers order_abandonment_rate
0              700                  318                  45.0%
```

Code E3: **Average Order Count:**

```
Avg_Order_Count_df = pd.read_sql_query("""
SELECT
    ROUND(AVG(order_count), 2) AS avg_order_count
FROM (
    SELECT
        customer_id,
        COUNT(DISTINCT order_id) AS order_count
    FROM orders_table
    GROUP BY customer_id
) AS OrderCounts;
""", conn)

print(Avg_Order_Count_df)
```

```
   avg_order_count
0             3.07
```

Code E4: Average campaign ROI:

```
Company_ROI_df = pd.read_sql_query("""
SELECT
    ROUND(
        CASE
            WHEN cc.total_marketing_cost = 0 THEN 0
            ELSE ((cp.total_profit - cc.total_marketing_cost) /
NULLIF(cc.total_marketing_cost, 0)) * 100
        END, 2
    ) || '%' AS Company_ROI
FROM
    (SELECT SUM((p.selling_price - p.cost_price) * od.quantity) AS
total_profit
     FROM orders_table o
     JOIN order_details_table od
       ON o.order_id = od.order_id
     JOIN products_table p
       ON od.product_id = p.product_id
     WHERE o.order_date >= '2024-08-01'
    ) AS cp
JOIN
    (SELECT SUM(cost) AS total_marketing_cost FROM
marketing_campaigns_table) AS cc;
""", conn)

print(Company_ROI_df)
```

```
  Company_ROI
0       6.83%
```

Code E5: <u>Average Revenue Generated per Campaign:</u>

```
Avg_Revenue_Per_Campaign_df = pd.read_sql_query("""
SELECT
    CASE
        WHEN tc.total_campaign = 0 THEN 0
        ELSE ROUND(tr.total_revenue / NULLIF(tc.total_campaign, 0), 2)
    END AS Avg_Revenue_Per_Campaign
FROM
    (SELECT SUM(amount) AS total_revenue
     FROM orders_table
     WHERE order_date >= '2024-08-01'
    ) AS tr
JOIN
    (SELECT COUNT(campaign_id) AS total_campaign
     FROM campaign_engagement_table
    ) AS tc;
""", conn)

print(Avg_Revenue_Per_Campaign_df)
```

```
   Avg_Revenue_Per_Campaign
0                  83724.95
```

Code E6: <u>Cost per Click:</u>

```
Cost_Per_Click_df = pd.read_sql_query("""
SELECT
    ROUND(AVG(cost_per_campaign.cost_per_click), 2) AS cost_per_click
FROM (
    SELECT
        cc.campaign_id,
        cc.total_cost,
        COALESCE(c.total_clicks, 0) AS total_clicks,
        CASE
            WHEN COALESCE(c.total_clicks, 0) = 0 THEN 0
            ELSE ROUND(cc.total_cost / NULLIF(c.total_clicks, 0), 2)
        END AS cost_per_click
    FROM (
        SELECT campaign_id, SUM(cost) AS total_cost
        FROM marketing_campaigns_table
        GROUP BY campaign_id
    ) AS cc
```

```
    LEFT JOIN (
        SELECT campaign_id, SUM(click_rate * reach / 100.0) AS
total_clicks
        FROM campaign_engagement_table
        GROUP BY campaign_id
    ) AS c
    ON cc.campaign_id = c.campaign_id
) AS cost_per_campaign;

""", conn)

print(Cost_Per_Click_df)
```

```
   cost_per_click
0             3.6
```

Code E7: <u>Sales across months (with and without discount):</u>

```
import pandas as pd

# 1. Orders Discount Used DataFrame:
Orders_Discount_Used_df = pd.read_sql_query("""
SELECT
    order_id,
    order_date,
    amount,
    CASE
        WHEN discount_id IS NULL OR discount_id = '' THEN 'N'
        ELSE 'Y'
    END AS discount_used
FROM orders_table;
""", conn)

# 2. Monthly Amount by Discount Usage DataFrame:
Monthly_Amount_by_Discount_df = pd.read_sql_query("""
SELECT
    strftime('%Y-%m', order_date) AS Month,
    SUM(CASE WHEN discount_id IS NOT NULL AND discount_id <> '' THEN
amount ELSE 0 END) AS Discount_Used_Amount,
    SUM(CASE WHEN discount_id IS NULL OR discount_id = '' THEN amount
ELSE 0 END) AS Discount_Not_Used_Amount
FROM orders_table
GROUP BY strftime('%Y-%m', order_date)
ORDER BY Month;
""", conn)
```

```
print(Monthly_Amount_by_Discount_df)
```

```
      Month  Discount_Used_Amount  Discount_Not_Used_Amount
0   2024-01              59716.43                 299454.90
1   2024-02              29068.93                 267936.70
2   2024-03              52840.41                 341915.81
3   2024-04              10775.14                 136278.47
4   2024-05              14982.25                 142987.12
5   2024-06              21853.94                 119786.77
6   2024-07              11499.10                 131311.12
7   2024-08             169919.87                  10071.71
8   2024-09             216252.57                  41918.94
9   2024-10             252066.14                  32479.67
10  2024-11             311935.88                  12200.12
11  2024-12             195560.13                  13469.22
```

Code E8: **ROI per Camapign:**

```
import pandas as pd

# 1. Product Profit DataFrame: Calculate profit per product
Product_Profit_df = pd.read_sql_query("""
SELECT
    product_id,
    product_name,
    selling_price,
    cost_price,
    (selling_price - cost_price) AS profit
FROM products_table;
""", conn)


# 2. Total Profit DataFrame: Calculate profit per order detail record
Total_Profit_df = pd.read_sql_query("""
SELECT
    od.product_id,
    od.quantity,
    p.product_name,
    p.selling_price,
    p.cost_price,
    ((p.selling_price - p.cost_price) * od.quantity) AS total_profit
FROM order_details_table AS od
JOIN products_table AS p
```

```
  ON od.product_id = p.product_id;
""", conn)


# 3. Order Total Profit DataFrame: Aggregate total profit per order
Order_Total_Profit_df = pd.read_sql_query("""
SELECT
    o.order_id,
    SUM((p.selling_price - p.cost_price) * od.quantity) AS
total_profit
FROM orders_table AS o
JOIN order_details_table AS od
  ON o.order_id = od.order_id
JOIN products_table AS p
  ON od.product_id = p.product_id
GROUP BY o.order_id;
""", conn)

# 4. Campaign ROI DataFrame:
#    For orders on or after 2024-08-01, compute per campaign:
#    - Sum total_profit (aggregated via discount_table)
#    - Sum marketing_cost from marketing_campaigns_table
#    - Compute Campaign_ROI_percentage as a percentage value rounded
to 2 decimals with a "%" sign.
Campaign_ROI_df = pd.read_sql_query("""
WITH order_profit AS (
    SELECT
         o.order_id,
         o.discount_id,
         SUM((p.selling_price - p.cost_price) * od.quantity) AS
total_profit
    FROM orders_table o
    JOIN order_details_table od
      ON o.order_id = od.order_id
    JOIN products_table p
      ON od.product_id = p.product_id
    WHERE o.order_date >= '2024-08-01'
    GROUP BY o.order_id, o.discount_id
),
profit_per_campaign AS (
    SELECT
         d.campaign_id,
         SUM(op.total_profit) AS total_profit
    FROM order_profit op
    JOIN discount_table d
```

```
      ON op.discount_id = d.discount_id
    GROUP BY d.campaign_id
),
cost_per_campaign AS (
    SELECT
         campaign_id,
         SUM(cost) AS total_marketing_cost
    FROM marketing_campaigns_table
    GROUP BY campaign_id
)
SELECT
    p.campaign_id,
    c.total_marketing_cost,
    p.total_profit,
    ROUND(((p.total_profit - c.total_marketing_cost) /
c.total_marketing_cost) * 100, 2) || '%' AS Campaign_ROI
FROM profit_per_campaign p
JOIN cost_per_campaign c
  ON p.campaign_id = c.campaign_id;
""", conn)

print(Campaign_ROI_df)
```

```
    campaign_id  total_marketing_cost  total_profit Campaign_ROI
0          1001               48000.0      38409.37      -19.98%
1          1002               60000.0      39738.76      -33.77%
2          1003               36000.0      64804.27       80.01%
3          1004               72000.0      47690.27      -33.76%
4          1005               28000.0      44375.74       58.48%
5          1006               52000.0      67894.98       30.57%
6          1007               56000.0      38560.37      -31.14%
7          1008               38500.0      52360.71        36.0%
8          1009               49000.0      28978.09      -40.86%
9          1010               28000.0      48504.50       73.23%
10         1011               59500.0      48581.41      -18.35%
11         1012               21000.0      48047.82       128.8%
12         1013               66500.0      39222.97      -41.02%
13         1014               35000.0      43510.32       24.32%
14         1015               70000.0      48508.20       -30.7%
```

Code E9: <u>Marketing Reach vs Cost for Different Campaign:</u>

```
Campaign_Cost_Reach_df = pd.read_sql_query("""
SELECT
    c.campaign_id,
    c.total_cost,
    COALESCE(r.total_reach, 0) AS total_reach
FROM
    (SELECT campaign_id, SUM(cost) AS total_cost
     FROM marketing_campaigns_table
     GROUP BY campaign_id
    ) AS c
LEFT JOIN
    (SELECT campaign_id, SUM(reach) AS total_reach
     FROM campaign_engagement_table
     GROUP BY campaign_id
    ) AS r
ON c.campaign_id = r.campaign_id;
""", conn)

print(Campaign_Cost_Reach_df)
```

```
    campaign_id  total_cost  total_reach
0          1001     48000.0       200000
1          1002     60000.0       350000
2          1003     36000.0       480000
3          1004     72000.0       600000
4          1005     28000.0       720000
5          1006     52000.0       250000
6          1007     56000.0       400000
7          1008     38500.0       520000
8          1009     49000.0       650000
9          1010     28000.0       300000
10         1011     59500.0       420000
11         1012     21000.0       550000
12         1013     66500.0       700000
13         1014     35000.0       270000
14         1015     70000.0       380000
```

Code E10: Profit per Category:

```
Profit_per_Category_df = pd.read_sql_query("""
SELECT
    c.category_id,
    c.category_name,
    ROUND(SUM((p.selling_price - p.cost_price) * od.quantity), 2) AS
total_profit
FROM orders_table o
JOIN order_details_table od
    ON o.order_id = od.order_id
JOIN products_table p
    ON od.product_id = p.product_id
JOIN category_table c
    ON p.category_id = c.category_id
GROUP BY c.category_id, c.category_name
ORDER BY total_profit DESC;
""", conn)

print(Profit_per_Category_df)
```

```
   category_id          category_name  total_profit
0         E387          Miscellaneous    1419009.24
1         7EE3            Electronics     199748.35
2         04BF           Home & Decor     105313.94
3         0DCP               Footwear      24019.29
4         C3E5             Toys & Gift      18966.68
5         7FE3  Clothes & Accessories      18719.43
6         715T         Home & Kitchen       5571.10
7         ZP25       Food & Beverages       2486.97
8         TG47        Health & Beauty       2208.35
9         A2CA            Art & Craft        380.75
10        2D5Y        Music Equipment          8.85
```

Code E11: <u>Revenue Per Campaign:</u>

```
Revenue_per_Campaign_df = pd.read_sql_query("""
SELECT
    c.campaign_id,
    ROUND(SUM(o.amount), 2) AS total_revenue
FROM orders_table o
JOIN discount_table d
    ON o.discount_id = d.discount_id
JOIN campaign_engagement_table c
    ON d.campaign_id = c.campaign_id
GROUP BY c.campaign_id
ORDER BY total_revenue DESC;
""", conn)


print(Revenue_per_Campaign_df)
```

```
    campaign_id  total_revenue
0          1011      126093.47
1          1003      115728.65
2          1006      115119.90
3          1008      101762.79
4          1014       94555.57
5          1015       91661.59
6          1010       89056.20
7          1005       85449.24
8          1004       84585.46
9          1012       82976.96
10         1001       82190.65
11         1013       77002.20
12         1002       71480.62
13         1007       67255.37
14         1009       61552.12
```

Code E12: <u>Discount Effectiveness:</u>

```
Discount_effective_df = pd.read_sql_query("""
SELECT
    dtt.type_of_discount,  -- Select type_of_discount from
discount_type_table
    COUNT(ot.order_id) AS total_orders,
    SUM(ot.amount) AS total_revenue -- Assuming 'amount' column in
orders_table represents total amount
FROM orders_table ot  -- Alias orders_table as ot
```

```
JOIN discount_table dt ON ot.discount_id = dt.discount_id  -- Join
with discount_table using discount_id
JOIN discount_type_table dtt ON dt.discount_type_id =
dtt.discount_type_id --Join with discount type table
WHERE ot.order_date >= '1/08/24'
GROUP BY dtt.type_of_discount  -- Group by type_of_discount
ORDER BY total_revenue DESC;
""", conn)

print(Discount_effective_df)
```

```
               type_of_discount  total_orders  total_revenue
0           Early Bird Discount            51      134919.92
1                  Holiday Sale            87      117338.09
2      Loyalty Program Discount            81      107164.02
3             Referral Discount            52      100844.58
4      Weekend Special Discount            80       89772.44
5       First Purchase Discount            55       86777.79
6        Bulk Purchase Discount            55       66719.69
7    Friends and Family Discount            31       64369.69
8    Exclusive App-Only Discount            43       62513.03
9             Seasonal Discount            38       61144.24
10           Buy One Get One Free           31       58218.25
11                   Flash Sale            28       57670.55
12               Clearance Sale            44       57157.95
13             Student Discount            46       56403.49
14       Senior Citizen Discount            60       55482.94
15           Limited-Time Offer            36       51292.66
16               Cashback Offer            27       49371.78
17          VIP Member Discount            27       39883.19
18             Anniversary Sale            28       17280.22
19             Mystery Discount             4       12146.27
```

Code E12: <u>Revenue per City:</u>

```
Revenue_Per_City_df = pd.read_sql_query("""
SELECT
    c.city_id,
    ct.city,
    ROUND(SUM(o.amount), 2) AS total_revenue
FROM orders_table o
JOIN customer_table c
    ON o.customer_id = c.customer_id
JOIN city_table ct
    ON c.city_id = ct.city_id
GROUP BY c.city_id, ct.city
ORDER BY total_revenue DESC;
""", conn)
```

```
print(Revenue_Per_City_df.head(10))
```

```
  city_id        city  total_revenue
0    Y2VK      London      309123.15
1    C6WB   Sheffield      276798.42
2    KJRQ   Liverpool      272225.13
3    7W41     Bristol      241757.51
4    92R4       Leeds      241377.13
5    XBR7  Nottingham      238636.38
6    B5VW  Manchester      236244.81
7    PYH6   Edinburgh      216451.61
8    30AN     Glasgow      198162.62
9    XU5I  Birmingham      183467.61
```