Table of Contents

Data Management Assignment - Group 28

1. Business Context and Database Design

1.1. Business Context

This report outlines the data strategy for Masteroast (2024), a UK-based company specialising in premium coffee sales. Masteroast sources high-quality coffee beans globally from trusted distributors. Once received at the company's UK warehouse, the beans undergo processing and packaging distribution to customers nationwide.

However, Masteroast faces operational challenges, particularly in delivery performance, product damage, and financial losses due to refunds. Issues such as damaged products and late deliveries impact customer satisfaction and profitability. To address these concerns, Masteroast is implementing a data-driven approach to localise these issues

1.1.1. Key Business Functions

Masteroast sells directly to consumers across the UK via an online platform. Its key functions include client management, product sales, transaction processing, procurement & inventory management, logistics & fulfillment, customer experience & quality control.

1.1.2. Database Mini World

Masteroast's database mini world models its business operations, it's key entities include:

- Clients – Stores customer details, including contact information and location.
- Products – Catalog of coffee offerings, tracking format, pricing, and launch date.
- Sales – Records customer purchases, quantities, and order details.
- Financial Transactions – Logs payments, refunds, net revenue, and shipping costs.
- Distributors – Stores supplier details and shipment information.
- Distributor Supply – Monitors batch shipments, delivery status, and supply chain performance.
- Client Deliveries – Logs delivery times, courier performance, and product damage incidents.
- Product Damage – Tracks damage severity, refund costs, and incident resolution timelines.

- Client Reviews – Stores customer feedback, including ratings, product experience, and location-based trends.

1.1.3. Data Product Overview

The data product provides insights to enhance Masteroast's customer experience, delivery efficiency, and financial performance. It generates four key reports:

1. Customer Review Analysis – Evaluates how coffee origin, format, customer location, and courier service type affect satisfaction.
2. Delivery Performance Analysis – Tracks courier efficiency, late deliveries, and supplier delays.
3. Financial Analysis – Assesses the financial impact of refunds and returns.
4. Product Damage Analysis – Investigates damage severity and the nature of damaged products.

1.2. Database Design

1.2.1. Entity-Relationship (ER) Diagram

Figure 1 outlines the data model for Masteroast, showing how key entities interact. It covers essential details like client information, product specifications, orders, payments, and delivery. Chen's notation was chosen for its clear representation of entities, attributes, and relationships, making it well-suited for Masteroast's straightforward business model and easy to understand for both technical and non-technical stakeholders.

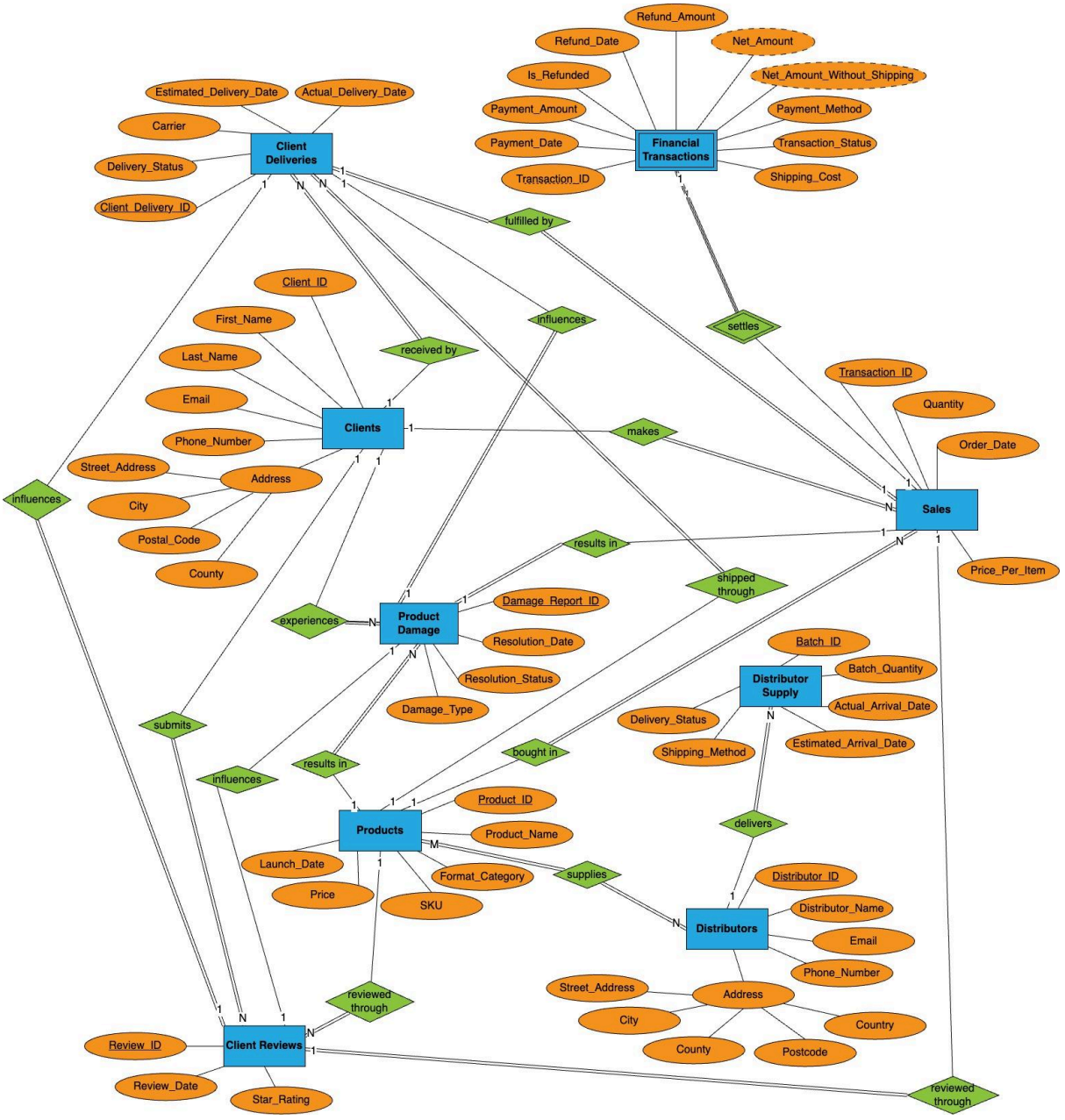


Figure 1. Entity-Relationship Diagram in Chen's Notation

1.2.2. Schema Design

Figure 2 outlines the key entities in Masteroast's database and their attributes, including primary keys (**bolded**) and foreign keys (underlined).

Entity	Purpose	Attributes
Clients	Stores customer details and account-related information	**Client_ID**, First_Name, Last_Name, Email, Phone_Number, Street_Address, City, Postal_Code, County
Products	Stores information about the catalog of products available for purchase	**Product_ID**, Product_Name, Format_Category, SKU, Price, Launch_Date
Sales	Stores overall transaction information	**Transaction_ID**, Client_ID, Product_ID, Order_Date, Quantity, Price_Per_Item
Financial Transactions	Tracks payment and refund information related to sales transactions	**Transaction_ID**, Payment_Date, Payment_Amount, Is_Refunded, Refund_Date, Refund_Amount, Net_Amount, Net_Amount_Without_Shipping, Payment_Method, Transaction_Status, Shipping_Cost
Distributors	Stores information about product distributors and their contact details	**Distributor_ID**, Distributor_Name, Email, Phone_Number, Street_Address, City, County, Postcode, Country
Distributor Supply	Tracks shipment details for distributor deliveries to the warehouse	**Batch_ID**, Distributor_ID, Product_ID, Batch_Quantity, Delivery_Status, Shipping_Method, Estimated_Arrival_Date, Actual_Arrival_Date
Client Deliveries	Stores details related to the shipping of orders to customers	**Client_Delivery_ID**, Transaction_ID, Client_ID, Product_ID, Delivery_Status, Carrier, Estimated_Delivery_Date, Actual_Delivery_Date
Product Damage	Tracks damage reports related to products and shipments	**Damage_Report_ID**, Transaction_ID, Client_ID, Product_ID, Review_ID, Client_Delivery_ID, Damage_Type, Resolution_Status, Resolution_Date
Client Reviews	Records customer feedback on products and transactions	**Review_ID**, Client_ID, Transaction_ID, Product_ID, Review_Date, Star_Rating

Figure 2. Masteroast's Database Schema

1.2.3. Relationship Cardinalities

Figure 3 summarises the interactions between entities in our data model, with details regarding cardinality and the nature of their relationship.

Entity 1	Entity 2	Cardinality	Relationship
Clients	Sales	1:N	A client can make multiple sales transactions for coffee purchases.
Clients	Client Deliveries	1:N	A client can receive multiple deliveries of coffee.
Clients	Product Damage	1:N	A client may experience multiple product damages, with each tied to a separate transaction.
Clients	Client Reviews	1:N	A client can submit multiple reviews — one for each transaction they make.
Sales	Financial Transactions	1:1	Each sale must be settled through a financial transaction.
Sales	Client Deliveries	1:1	Each sale is fulfilled by a single delivery to the client.
Sales	Product Damage	1:1	Each sale may result in product damage, which is recorded in a single report.
Sales	Client Reviews	1:1	Each sale is reviewed through a single client review.
Products	Sales	1:N	A product type (ID) can be bought in multiple sales, once for each transaction.
Products	Client Deliveries	1:N	A product type (ID) can be shipped through multiple client deliveries, once for each transaction.
Products	Product Damage	1:N	A product type (ID) can result in damage multiple times, with one report per transaction.
Products	Client Reviews	1:N	Each product type (ID) can be reviewed multiple times, once for every transaction.
Client Deliveries	Product Damage	1:1	Each delivery may influence a single product damage report.
Client Deliveries	Client Reviews	1:1	Each delivery may influence a single client review.
Product Damage	Client Reviews	1:1	Product damage may influence a single review written by the client.
Distributors	Products	N:M	Each distributor supplies multiple products, and multiple distributors can supply the same product.
Distributors	Distributor Supply	1:N	Each distributor delivers product supplies to the company multiple times.

Figure 3. Entity Relationship Cardinalities

1.2.4. Assumptions

The following assumptions have been made regarding Masteroast's database design:

- Each sales transaction contains only a single product type.
- Each sales transaction is associated with one delivery.
- Each sales transaction results in a single review.

2. Data Implementation and Synthetic Data Generation

2.1. SQL Schema Implementation

The Masteroast database schema is designed with reliability and referential integrity in mind. Identifiers like Client_ID and Product_ID use *VARCHAR(36)* for UUIDs, while shorter codes are used for distributors (*VARCHAR(8)*) and batches (*VARCHAR(6)*). Text fields such as names (*VARCHAR(50)*), emails (*VARCHAR(100)*), and addresses (*VARCHAR(150)*) reflect typical real-world lengths. Whole numbers, such as Quantity and Star_Rating, are stored as *INTEGER*, while financial values like Price and Payment_Amount use *REAL* for decimal precision.

For improved readability, Is_Refunded is stored as *VARCHAR(3)*, allowing "Yes" or "No" instead of *BOOLEAN*. This makes querying and interpreting refund statuses more intuitive. Dates are formatted as *DATE* (YYYY-MM-DD) for consistency, and foreign keys maintain referential integrity across tables.

2.1.1. Clients

```
CREATE TABLE IF NOT EXISTS Clients (
        Client_ID VARCHAR(36) PRIMARY KEY,
        First_Name VARCHAR(50),
        Last_Name VARCHAR(50),
        Email VARCHAR(100),
        Phone_Number VARCHAR(14),
        Street_Address VARCHAR(150),
        City VARCHAR(50),
        Postal_Code VARCHAR(4),
        County VARCHAR(50)
    )
```

2.1.2. Products

```
CREATE TABLE IF NOT EXISTS Products (
        Product_ID VARCHAR(36) PRIMARY KEY,
        Product_Name VARCHAR(20),
        Format_Category VARCHAR(20),
        SKU VARCHAR(13),
        Price REAL,
        Launch_Date DATE
    )
```

2.1.3. Sales

```
CREATE TABLE IF NOT EXISTS Sales (
        Transaction_ID VARCHAR(36) PRIMARY KEY,
        Client_ID VARCHAR(36),
        Product_ID VARCHAR(36),
        Order_Date DATE,
        Quantity INTEGER,
        Price_Per_Item REAL,
        FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
    )
```

2.1.4. Financial Transaction

```
CREATE TABLE IF NOT EXISTS Financial_Transactions (
        Transaction_ID VARCHAR(36) PRIMARY KEY,
        Payment_Date DATE,
        Payment_Amount REAL,
        Is_Refunded VARCHAR(3),
        Refund_Date DATE,
        Refund_Amount REAL,
        Net_Amount REAL,
        Net_Amount_Without_Shipping REAL,
        Payment_Method VARCHAR(20),
        Transaction_Status VARCHAR(20),
        Shipping_Cost REAL,
        FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID)
    )
```

2.1.5. Client Deliveries

```
CREATE TABLE IF NOT EXISTS Client_Deliveries (
        Client_Delivery_ID VARCHAR(36) PRIMARY KEY,
        Transaction_ID VARCHAR(36),
        Client_ID VARCHAR(36),
        Product_ID VARCHAR(36),
        Delivery_Status VARCHAR(15),
        Carrier VARCHAR(20),
        Estimated_Delivery_Date DATE,
        Actual_Delivery_Date DATE,
        FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID),
        FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
    )
```

2.1.6. Client Reviews

```
CREATE TABLE IF NOT EXISTS Client_Reviews (
        Review_ID VARCHAR(36) PRIMARY KEY,
        Client_ID VARCHAR(36),
        Transaction_ID VARCHAR(36),
        Product_ID VARCHAR(36),
        Review_Date DATE,
        Star_Rating INTEGER,
        FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
        FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
    )
```

2.1.7. Product Damage

```
CREATE TABLE IF NOT EXISTS Product_Damage (
        Damage_Report_ID VARCHAR(36) PRIMARY KEY,
        Transaction_ID VARCHAR(36),
        Client_ID VARCHAR(36),
        Product_ID VARCHAR(36),
        Review_ID VARCHAR(36),
        Client_Delivery_ID VARCHAR(36),
        Damage_Type VARCHAR(20),
        Resolution_Status VARCHAR(15),
        Resolution_Date DATE,
        FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID),
        FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID),
        FOREIGN KEY (Review_ID) REFERENCES Client_Reviews(Review_ID),
        FOREIGN KEY (Client_Delivery_ID) REFERENCES
Client_Deliveries(Client_Delivery_ID)
    )
```

2.1.8. Distributors

```
CREATE TABLE IF NOT EXISTS Distributors (
        Distributor_ID VARCHAR(8) PRIMARY KEY,
        Distributor_Name VARCHAR(50),
        Email VARCHAR(100),
        Phone_Number VARCHAR(14),
        Street_Address VARCHAR(150),
        City VARCHAR(50),
        County VARCHAR(50),
        Postcode VARCHAR(10),
        Country VARCHAR(60)
    )
```

2.1.9. Distributor Supply

```
CREATE TABLE IF NOT EXISTS Distributor_Supply (
        Batch_ID VARCHAR(6) PRIMARY KEY,
        Distributor_ID VARCHAR(8),
        Product_ID VARCHAR(36),
        Batch_Quantity INTEGER,
        Delivery_Status VARCHAR(15),
        Shipping_Method VARCHAR(10),
        Estimated_Arrival_Date DATE,
        Actual_Arrival_Date DATE,
        FOREIGN KEY (Distributor_ID) REFERENCES
Distributors(Distributor_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
    )
```

2.1.10. Database Normalisation Checking

A normalisation check was conducted to ensure the database adheres to 1NF, 2NF, and 3NF standards. A Python script extracted table structures and foreign key relationships from SQLite, verifying that all attributes were properly normalised. The results (Appendix L) confirmed that all tables comply with 3NF, requiring no further modifications.

2.2. Synthetic Data Generation

2.2.1. Clients Table

Client_ID	First_Name	Last_Name	Email	Phone_Number	Street_Address	City	Postal_Code	County
35d6d354-df39-4273-a618-19e817f60272	Ruth	Griffiths	ruth.griffiths@gmail.com	+44 7046913810	Flat 32 Sian streets	Edinburgh	EH1	City of Edinburgh
899c128e-d7e7-46ad-ab05-22fa9b64a64b	Janice	Hopkins	janice.hopkins@gmail.com	+44 7223756669	Flat 0 Charlie fort	Leeds	LS1	West Yorkshire
11ae7280-9746-4f4c-a75b-ab4786092777	Leanne	Foster	leanne.foster@gmail.com	+44 7966629388	863 Wright valleys	Edinburgh	EH1	City of Edinburgh
eb14fc12-f806-4749-ae0e-e239e9cfb02c	Thomas	Reed	thomas.reed@gmail.com	+44 7139345092	654 Robin track	London	EC1A	Greater London
54459162-85ac-4f87-850c-e66a698d41c5	Tom	Warner	tom.warner@gmail.com	+44 7836687537	Flat 5 Butler valley	Leeds	LS1	West Yorkshire
30f8b8db-7e86-4f4a-befa-f861a499d5d6	Samuel	Elliott	samuel.elliott@icloud.com	+44 7370291817	18 Norton river	Cardiff	CF10	Cardiff
84379f63-7885-4989-93f4-e16a2fae4a53	Linda	Mellor	linda.mellor@icloud.com	+44 7031429110	3 Robinson walk	Glasgow	G1	Glasgow City
e92712b3-41a4-4ac6-b320-634359af009a	Donald	Taylor	donald.taylor@icloud.com	+44 7547295260	Flat 75k Evans skyway	Cardiff	CF10	Cardiff
fa4c3058-9b2b-4fd1-9792-6f94f7a4122d	Vincent	Harrison	vincent.harrison@gmail.com	+44 7523718431	283 Bethany light	Birmingham	B1	West Midlands
421addab-d578-4cad-8134-5ceb60af6144	Mathew	Scott	mathew.scott@icloud.com	+44 7158181396	503 Guy locks	Manchester	M1	Greater Manchester

Figure 4. Clients Table Sample Data

Column Name	Description
Client_ID (PK)	Unique identifier for each client (UUID)
First_Name	Client's first name
Last_Name	Client's last name
Email	Email address (Gmail/iCloud)
Phone_Number	UK phone number in international format
Street_Address	Client's street address
City	City of residence
Postal_Code	Postal code of the city
County	County associated with the city

Figure 5. Clients Table Data Dictionary

The Clients dataset contains 1,000 synthetic UK-based client records. It assigns each client a unique ID, name, email, phone number, and street address. A random UK city is selected, mapping to its postal code and county. See Appendix C for the data generation code.

2.2.2. Products Table

Product_ID	Product_Name	Format_Category	SKU	Price	Launch_date
0ff5dbba-3be6-496b-8a86-7d04388e21ab	Arabica	Powder	ARA-POW-29382	9.17	April 5, 2015
4cd11909-7c61-4fec-8095-0e633a59a3db	Arabica	Whole Bean	ARA-WHO-43801	14.56	October 22, 2017
7869f5da-e4ef-41d1-9d19-79817863d032	Arabica	Pods	ARA-POD-18783	22.7	June 14, 2019
5d1bcfe2-e31f-4b69-bbad-d53fd3348cb4	Robusta	Powder	ROB-POW-82923	5.99	December 8, 2021
3f9515f3-712b-4c16-b34a-ecf9d61eeeda	Robusta	Whole Bean	ROB-WHO-52746	16.85	March 30, 2016
56179492-4171-4f59-a061-06bb0d5cd6c2	Robusta	Pods	ROB-POD-78116	23.67	July 11, 2014
c2f97ccb-da4a-46dd-bd62-13d184b695d2	Excelsia	Powder	EXC-POW-11620	10.72	January 3, 2018
9a1e1149-8e28-4dc2-8c27-736a31e4bad9	Excelsia	Whole Bean	EXC-WHO-91665	18.7	September 25, 2020
82917a01-a33f-4962-bee7-2d52652652c2	Excelsia	Pods	EXC-POD-64997	25.79	November 17, 2022
86e94322-fab1-40f2-83ef-e9d2432ebc45	Liberica	Powder	LIB-POW-29613	7.89	August 9, 2014
33255307-7e0f-4fa2-9e20-0d2dbead729d	Liberica	Whole Bean	LIB-WHO-87061	21.24	May 4, 2023
f2169c73-55c6-4493-a8e4-014afe069655	Liberica	Pods	LIB-POD-23133	34.47	February 19, 2024

Figure 6. Products Table Data

Column Name	Description
Product_ID (PK)	Unique identifier for each product (UUID)
Product_Name	Name of the coffee product (e.g., Arabica)
Format_Category	Format type (Powder, Whole Bean, Pods)
SKU	Stock keeping unit (unique code)
Price	Product price
Launch_Date	Launch date of the product

Figure 7. Products Table Data Dictionary

The Products dataset includes four coffee types in three formats, with each combination assigned a predefined price range and launch date. Each product is assigned a unique ID, stock keeping unit (SKU) and a random price within its range. See Appendix D for the data generation code.

2.2.3. Sales Table

Transaction_ID	Client_ID	Product_ID	Order_Date	Quantity	Price_Per_Item
c519619c-5c83-44c0-b3bb-d908329aa469	35d6d354-df39-4273-a618-19e817f60272	4cd11909-7c61-4fec-8095-0e633a59a3db	2020-07-14	5	14.56
232e93f5-5da5-40e8-892f-72231c9e92b1	35d6d354-df39-4273-a618-19e817f60272	82917a01-a33f-4962-bee7-2d52652652c2	2023-02-06	6	25.79
c4fac3a8-5fa9-4565-be96-2265ccf1af15	35d6d354-df39-4273-a618-19e817f60272	c2f97ccb-da4a-46dd-bd62-13d184b695d2	2020-05-08	5	10.72
3a6ec892-0ebe-4c87-b3be-d0ff4df1c7a2	35d6d354-df39-4273-a618-19e817f60272	c2f97ccb-da4a-46dd-bd62-13d184b695d2	2024-02-10	8	10.72
c34722df-4fb2-4303-add1-ac4bda3403bb	35d6d354-df39-4273-a618-19e817f60272	86e94322-fab1-40f2-83ef-e9d2432ebc45	2021-01-03	10	7.89
fb114c99-f37e-4b61-ae0c-955f6ae4ffde	35d6d354-df39-4273-a618-19e817f60272	9a1e1149-8e28-4dc2-8c27-736a31e4bad9	2022-12-26	1	18.7
bb66a004-2004-4e70-b908-b943820a1317	35d6d354-df39-4273-a618-19e817f60272	7869f5da-e4ef-41d1-9d19-79817863d032	2023-04-27	5	22.7
837e47b1-af34-4580-bebb-316422f9779b	35d6d354-df39-4273-a618-19e817f60272	c2f97ccb-da4a-46dd-bd62-13d184b695d2	2021-05-02	3	10.72
65138adf-9ad8-4ea8-b5e2-d48ca5d63428	35d6d354-df39-4273-a618-19e817f60272	86e94322-fab1-40f2-83ef-e9d2432ebc45	2022-03-11	10	7.89
07ac2ad1-2c44-47a7-a04a-bcf35de88aed	35d6d354-df39-4273-a618-19e817f60272	33255307-7e0f-4fa2-9e20-0d2dbead729d	2023-10-10	7	21.24

Figure 8. Sales Table Sample Data

Column Name	Description
Transaction_ID (PK)	Unique identifier for the transaction (UUID)
Client_ID (FK)	Foreign key linking to *Clients* entity
Product_ID (FK)	Foreign key linking to *Products* entity
Order_Date	Date of the order
Quantity	Number of items purchased
Price_Per_Item	Price of the product at the time of purchase

Figure 9. Sales Table Data Dictionary

The Sales dataset consists of approximately 5,500 sales transactions. Each client is assigned between 1 and 10 transactions, where a unique ID is generated for each purchase. A product with its associated price is randomly selected from the product catalog, and a random order quantity between 1 and 10 is chosen. The order date is also limited to one within the last decade. See Appendix E for the data generation code.

2.2.4. Financial Transactions Table

Transaction_ID	Payment_Date	Payment_Amount	Is_Refunded	Refund_Date	Refund_Amount	Net_Amount	Net_Amount_Without_Shipping	Payment_Method	Transaction_Status	Shipping_Cost
c519619c-5c83-44c0-b3bb-d908329aa469	2020-07-14	72.8	No		0	72.8	66.8	Credit Card	Pending	6
232e93f5-5da5-40e8-892f-72231c9e92b1	2023-02-06	154.74	No		0	154.74	148.74	Debit Card	Pending	6
c4fac3a8-5fa9-4565-be96-2265ccf1af15	2020-05-08	53.6	No		0	53.6	47.6	Credit Card	Completed	6
3a6ec892-0ebe-4c87-b3be-d0ff4df1c7a2	2024-02-10	85.76	No		0	85.76	79.76	Cash	Pending	6
c34722df-4fb2-4303-add1-ac4bda3403bb	2021-01-03	78.9	No		0	78.9	72.9	Bank Transfer	Completed	6
fb114c99-f37e-4b61-ae0c-955f6ae4ffde	2022-12-26	18.7	No		0	18.7	12.7	Credit Card	Completed	6
bb66a004-2004-4e70-b908-b943820a1317	2023-04-27	113.5	No		0	113.5	107.5	Cash	Completed	6
837e47b1-af34-4580-bebb-316422f9779b	2021-05-02	32.16	No		0	32.16	26.16	Cash	Pending	6
65138adf-9ad8-4ea8-b5e2-d48ca5d63428	2022-03-11	78.9	No		0	78.9	72.9	Cash	Completed	6
07ac2ad1-2c44-47a7-a04a-bcf35de88aed	2023-10-10	148.68	No		0	148.68	142.68	Bank Transfer	Completed	6

Figure 10. Financial Transactions Table Sample Data

Column Name	Description
Transaction_ID (PK, FK)	Foreign key linking to *Sales* entity
Payment_Date	Date the payment was made
Payment_Amount	Total amount paid (before refunds)
Is_Refunded	Indicates if the transaction was refunded
Refund_Date	Date when the refund was issued (if applicable)
Refund_Amount	Amount refunded (if applicable)
Net_Amount	Total payment after refunds
Net_Amount_Without_Shipping	Net payment excluding shipping cost
Payment_Method	Method of payment (Credit Card, PayPal, etc.)
Transaction_Status	Status of the transaction (Completed, Pending, etc.)
Shipping_Cost	Cost of shipping the order

Figure 11. Financial Transactions Table Data Dictionary

The Financial Transactions dataset links each sale to payment details, assigning a payment method and calculating the total amount based on product quantity and price. Shipping costs are determined by the client's city, and a transaction status is randomly assigned. If

refunded, the refund amount and date are recorded, and the net amount is adjusted accordingly. See Appendix F for the data generation code.

2.2.5. Client Deliveries Table

Transaction_ID	Client_ID	Product_ID	Client_Delivery_ID	Delivery_Status	Carrier	Estimated_Delivery_Date	Actual_Delivery_Date
c519619c-5c83-44c0-b3bb-d908329aa469	35d6d354-df39-4273-a618-19e817f60272	4cd11909-7c61-4fec-8095-0e633a59a3db	e17ae4e1-9e64-40b7-a662-087066ea4cd3	Pending			
232e93f5-5da5-40e8-892f-72231c9e92b1	35d6d354-df39-4273-a618-19e817f60272	82917a01-a33f-4962-bee7-2d52652652c2	13219357-251b-4382-812c-034273a2cb3f	Pending			
c4fac3a8-5fa9-4565-be96-2265ccf1af15	35d6d354-df39-4273-a618-19e817f60272	c2f97ccb-da4a-46dd-bd62-13d184b695d2	4aefccf3-e312-4467-8096-47bdf3759d76	Delivered	DPD	2020-05-18	2020-05-18
3a6ec892-0ebe-4c87-b3be-d0ff4df1c7a2	35d6d354-df39-4273-a618-19e817f60272	c2f97ccb-da4a-46dd-bd62-13d184b695d2	6d6f03fb-b2ba-4612-a996-b34dc4791406	Pending			
c34722df-4fb2-4303-add1-ac4bda3403bb	35d6d354-df39-4273-a618-19e817f60272	86e94322-fab1-40f2-83ef-e9d2432ebc45	ab5d656f-b89c-4257-91bc-bf50b6d5bebf	Shipped	Royal Mail	2021-01-05	
fb114c99-f37e-4b61-ae0c-955f6ae4ffde	35d6d354-df39-4273-a618-19e817f60272	9a1e1149-8e28-4dc2-8c27-736a31e4bad9	b7d1993a-6fe4-4a07-b22c-a4f5418cad56	Delivered	DPD	2022-12-31	2022-12-26
bb66a004-2004-4e70-b908-b943820a1317	35d6d354-df39-4273-a618-19e817f60272	7869f5da-e4ef-41d1-9d19-79817863d032	1a5e86a3-dfa0-40d8-99c2-4a42ed9cf977	Delivered	Royal Mail	2023-05-01	2023-04-30
837e47b1-af34-4580-bebb-316422f9779b	35d6d354-df39-4273-a618-19e817f60272	c2f97ccb-da4a-46dd-bd62-13d184b695d2	5273e3a2-5a9e-46d0-867e-726de92c6c1b	Pending			
65138adf-9ad8-4ea8-b5e2-d48ca5d63428	35d6d354-df39-4273-a618-19e817f60272	86e94322-fab1-40f2-83ef-e9d2432ebc45	ec5fcaa5-7087-4e57-9002-d976ceb47e5f	Delivered	DPD	2022-03-13	2022-03-11
07ac2ad1-2c44-47a7-a04a-bcf35de88aed	35d6d354-df39-4273-a618-19e817f60272	33255307-7e0f-4fa2-9e20-0d2dbead729d	fd777da3-3769-4171-ac25-61fd8d92387c	Shipped	Royal Mail	2023-10-15	

Figure 12. Client Deliveries Table Sample Data

Column Name	Description
Transaction_ID (FK)	Foreign key linking to *Sales* entity
Client_ID (FK)	Foreign key linking to *Clients* entity
Product_ID (FK)	Foreign key linking to *Products* entity
Client_Delivery_ID (PK)	Unique identifier for the delivery
Delivery_Status	Status of delivery (Delivered, Pending, etc.)
Carrier	Shipping company (DHL, Royal Mail, etc.)
Estimated_Delivery_Date	Estimated delivery date
Actual_Delivery_Date	Actual delivery date (if delivered)

Figure 13. Client Deliveries Table Data Dictionary

The Client Deliveries dataset links financial transactions to delivery details. Delivery status is based on payment status, with processed orders categorised as "Shipped," "Delivered," "Delayed," or "Cancelled." A carrier is assigned (e.g., DHL, Royal Mail), an estimated and actual delivery date is set after the order date. Delivered items arrive on time 95% of the time, while delays range from 7 to 30 days. See Appendix G for the data generation code.

2.2.6. Client Review Table

Review_ID	Client_ID	Transaction_ID	Product_ID	Review_Date	Star_Rating
057bdd35-bd87-4498-8143-d8b9755cd5dd	35d6d354-df39-4273-a618-19e817f60272	c4fac3a8-5fa9-4565-be96-2265ccf1af15	c2f97ccb-da4a-46dd-bd62-13d184b695d2	2020-05-10	5
58b80ca7-e434-4984-b4c3-7ce62e23682f	35d6d354-df39-4273-a618-19e817f60272	fb114c99-f37e-4b61-ae0c-955f6ae4ffde	9a1e1149-8e28-4dc2-8c27-736a31e4bad9	2023-01-06	4
c36d689d-c294-4fe9-9d03-53701bcebcf7	35d6d354-df39-4273-a618-19e817f60272	bb66a004-2004-4e70-b908-b943820a1317	7869f5da-e4ef-41d1-9d19-79817863d032	2023-04-30	5
38017ea2-2e46-4d4a-be39-c17a86a075bd	35d6d354-df39-4273-a618-19e817f60272	65138adf-9ad8-4ea8-b5e2-d48ca5d63428	86e94322-fab1-40f2-83ef-e9d2432ebc45	2022-03-18	4
ab40cad2-d9eb-42d2-94e3-72fe64875fbc	35d6d354-df39-4273-a618-19e817f60272	07ac2ad1-2c44-47a7-a04a-bcf35de88aed	33255307-7e0f-4fa2-9e20-0d2dbead729d	2023-10-12	5
4b451a41-73f3-4e48-b2ce-36492aae0a9b	899c128e-d7e7-46ad-ab05-22fa9b64a64b	cb85c766-34ad-44e6-b026-a3a6151f3f8c	c2f97ccb-da4a-46dd-bd62-13d184b695d2	2022-04-28	4
435214db-7e9a-4f2b-9c37-926a815af2a7	11ae7280-9746-4f4c-a75b-ab4786092777	7070de93-c05b-466b-8524-f13bbedffb4d	0ff5dbba-3be6-496b-8a86-7d04388e21ab	2020-04-28	4
d468e397-d5e4-4b40-9e93-8ef8e61d8bd3	eb14fc12-f806-4749-ae0e-e239e9cfb02c	5ae461eb-7ce8-4346-b3c1-a3308aed0afa	9a1e1149-8e28-4dc2-8c27-736a31e4bad9	2022-04-10	4
a1b65949-eafd-4af4-91b3-f1cbdec48f86	eb14fc12-f806-4749-ae0e-e239e9cfb02c	36324281-c4d3-4530-be0d-dd856d0ee05e	5d1bcfe2-e31f-4b69-bbad-d53fd3348cb4	2025-01-02	4
7c4c5032-6ade-4a5f-8c15-8b70575d0726	eb14fc12-f806-4749-ae0e-e239e9cfb02c	4cd9409b-eec5-451b-83cf-90dc726257c1	f2169c73-55c6-4493-a8e4-014afe069655	2024-08-15	4

Figure 14. Client Review Table Sample Data

Column Name	Description
Review_ID	Unique identifier for each review (UUID)
Client_ID	Foreign key linking to *Clients* entity
Transaction_ID	Foreign key linking to *Sales* entity
Product_ID	Foreign key linking to *Products* entity
Review_Date	Date when the review was submitted
Star_Rating	Customer rating (1 to 5 stars)

Figure 15. Client Review Table Data Dictionary

The Client Review dataset includes reviews for delivered transactions. It filters out non-delivered orders from the Client Deliveries dataset, assigns a unique Review_ID, and links it to the corresponding Client_ID and Product_ID based on Transaction_ID. The Review_Date is set 2-3 days after delivery. Ratings are based on transaction outcomes: refunds receive 1-2 stars, delayed deliveries get 4 stars, and on-time deliveries receive 4 or 5 stars. See Appendix H for the data generation code.

2.2.7. Product Damage Table

Damage_Report_ID	Transaction_ID	Client_ID	Product_ID	Review_ID	Client_Delivery_ID	Damage_Type	Resolution_Status	Resolution_Date	Batch_ID
c29057a6-c3e3-48c5-8992-bd5d1da65962	c4fac3a8-5fa9-4565-	de3938d1-3c5c-4bc	33255307-7e0f-4fa2	025f3861-89e6-41e	816d797c-4d82-461	Packaging Issue	In Progress		B437726
6edbf90e-3ff6-4be6-8405-2d8f6557c77a	fb114c99-f37e-4b61-	b04cdcf8-e634-4a8c	3f9515f3-712b-4c16	42fb66dc-454a-49c6	ccfa3613-7641-4eb9	Shipping Damage	Pending		B576207
32ba31ae-53d3-4083-8188-ca3ff4d8b354	bb66a004-2004-4e7	14ac6665-3fcc-4a2e	f2169c73-55c6-4493	bd88882d-5dc9-4fa1	6e74efc8-d226-4fe1	Product Defect	Pending		B707065
788278b1-9bfb-4f10-b8e9-03fee841464a	65138adf-9ad8-4ea8	504d9f22-aabf-496f-	56179492-4171-4f55	f4c00458-eb1e-48bc	2a323755-abf5-457(	Missing Item	In Progress		B149889
5b9099d2-7e16-4520-9e75-5486f79a17fc	07ac2ad1-2c44-47a	e7df877b-4c21-4f75	5d1bcfe2-e31f-4b69	5a26ce49-2c8c-406(	d988148c-31ad-4b3	Wrong Item	Pending		B248079
ba1dbb9b-c936-48dc-b1ac-99b6b02e84fc	cb85c766-34ad-44e(	c933c361-738a-462	9a1e1149-8e28-4dc	5a2080c4-cda2-409(	cfc853a1-19b7-464t	Packaging Issue	Resolved	2023-05-01	B529165
c9355fad-bbc2-4ef3-b893-59fd6f0615c3	7070de93-c05b-466	cf07c72f-01f7-4127-	c2f97ccb-da4a-46dd	21e46adb-278f-4b4(	d8421a9b-fe9c-49b2	Missing Item	Resolved	2020-02-17	B128782
6de187f0-f393-4442-a5fe-8c110b296510	5ae461eb-7ce8-434(	5ca4b53f-da32-446(	f2169c73-55c6-4493	123d7abf-4626-465(	14cc065b-6514-422	Product Defect	Pending		B423362
9e32beb1-5c21-4c83-a736-d7ff25304071	36324281-c4d3-453	afa51b28-0ae4-4e3(	4cd11909-7c61-4fec	bdc83415-d029-4a8	1b5a601f-a1d8-49a1	Wrong Item	Pending		B550132
445afa36-3803-49a9-bfa1-45893417c4e4	4cd9409b-eec5-451	6678929c-51cc-4eb	3f9515f3-712b-4c16	70be9592-1dfa-42f8	5e5bc4cd-674b-480	Packaging Issue	In Progress		B172754

Figure 16. Product Damage Table Sample Data

Column Name	Description
Damage_Report_ID	Unique identifier for the damage report (UUID)
Transaction_ID	Foreign key linking to *Sales* entity
Client_ID	Foreign key linking to *Clients* entity
Product_ID	Foreign key linking to *Products* entity
Review_ID	Foreign key linking to *Client Reviews* entity
Client_Delivery_ID	Foreign key linking to *Client Deliveries* entity
Damage_Type	Type of damage (Product Defect, Shipping Damage, etc.)
Resolution_Status	Status of resolution (Pending, Resolved, etc.)
Resolution_Date	Date when the issue was resolved (if applicable)

Figure 17. Product Damage Table Data Dictionary

The Product Damage dataset logs damage reports for delivered transactions. Each report has a unique Damage_Report_ID linked to relevant IDs (Transaction_ID, Client_ID, etc.).

Damage types include "Product Defect," "Shipping Damage," and "Wrong Item." Resolution status is set as "Pending," "In Progress," or "Resolved," with a resolution date if applicable. See Appendix I for the data generation code.

2.2.8. Distributors Table

Distributor_ID	Distributor_Name	Email	Phone_Number	Street_Address	City	County	Postcode	Country
B5614226	Payne, Ryan and Brooks	payneryanandbrooks@gmail.com	+55 2051802512	Flat 79 Barker avenue	Lake Mary	Flintshire	CV3 1DR	Brazil
C3341057	Evans LLC	evansllc@gmail.com	+57 4163119785	Flat 7 Gemma common	Teresastad	Inverclyde	E03 1J	Colombia
V2458591	White-Woods	white-woods@gmail.com	+84 7831113321	Flat 55 Vanessa grove	New Heatherburgh	Eilean Siar	EH2R 4	Vietnam
I1533224	Duncan, Read and George	duncanreadandgeorge@gmail.com	+91 1127978094	64 Phillip port	Phillipfort	Anglesey	LE2M 3	India
M4668136	Morris-Rowe	morris-rowe@gmail.com	+52 3585650756	Flat 32G Green causeway	East Teresaburgh	West Lothian	PH16	Mexico

Figure 18. Distributors Table Data

Column Name	Description
Distributor_ID	Uniqude identifier for the distributor
Distributor_Name	Name of the distributor company
Email	Contact email of the distributor
Phone_Number	Contact phone number of the distributor
Street_Address	Streert address of the distributor
City	City where the distributor is located
County	County of the distributor
Postcode	Postcode of the distributor
Country	Country where the distributor operates

Figure 19. Distributors Table Data Dictionary

The Distributors dataset includes distributors from Brazil, Colombia, Vietnam, India, and Mexico. Each entry has a unique ID , company name, email, phone number, and address. Details are generated using the Faker library, while postcodes are formatted per country standards. See Appendix J for the data generation code.

2.2.9. Distributor Supply Table

Batch_ID	Distributor_ID	Product_ID	Batch_Quantity	Delivery_Status	Shipping_Method	Estimated_Arrival_Date	Actual_Arrival_Date
B00000	B5614226	0ff5dbba-3be6-496b-8a86-7d04388e21ab	343	Delivered	Water	2024-12-11	2024-12-15
B00001	B5614226	4cd11909-7c61-4fec-8095-0e633a59a3db	256	Pending	Road	2024-06-23	
B00002	B5614226	7869f5da-e4ef-41d1-9d19-79817863d032	201	Cancelled	Air	2024-12-25	
B00003	B5614226	5d1bcfe2-e31f-4b69-bbad-d53fd3348cb4	271	Delivered	Air	2024-06-07	2024-06-07
B00004	B5614226	3f9515f3-712b-4c16-b34a-ecf9d61eeeda	224	Delivered	Air	2024-11-04	2024-11-01
B00005	B5614226	56179492-4171-4f59-a061-06bb0d5cd6c2	386	Pending	Road	2024-02-16	
B00006	B5614226	c2f97ccb-da4a-46dd-bd62-13d184b695d2	341	Delivered	Water	2024-11-09	2024-11-14
B00007	B5614226	9a1e1149-8e28-4dc2-8c27-736a31e4bad9	249	Delayed	Air	2024-01-30	2024-01-29
B00008	B5614226	82917a01-a33f-4962-bee7-2d52652652c2	220	Delivered	Air	2024-07-24	2024-07-23
B00009	B5614226	86e94322-fab1-40f2-83ef-e9d2432ebc45	241	Delivered	Air	2024-05-01	2024-04-28

Figure 20. Distributor Supply Table Sample Data

Column Name	Description
Batch_ID	Unique batch identifier
Distributor_ID	Foreign key linking to *Distributors* entity
Product_ID	Foreign key linking to *Products* entity
Batch_Quantity	Quantity of products in the batch
Delivery_Status	Status fo batch delivery (Delivered, Pending, etc.)
Shipping_Method	Transport method (Air, Road, Water)
Estimated_Arrival_Date	Expected date of arrival
Actual_Arrival_Date	Actual date of arrival (if delivered)

Figure 21. Distributor Supply Table Data Dictionary

The Distributor Supply dataset links distributors to the products they deliver to Masteroast. It assigns unique batch IDs, quantities (200-400), delivery statuses, shipping methods (Air, Road, Water), and arrival dates. Delivery statuses include "Delivered," "Pending," "Shipped," "Delayed," and "Cancelled," with estimated and actual arrival dates determined by the shipping method. See Appendix K for the data generation code.

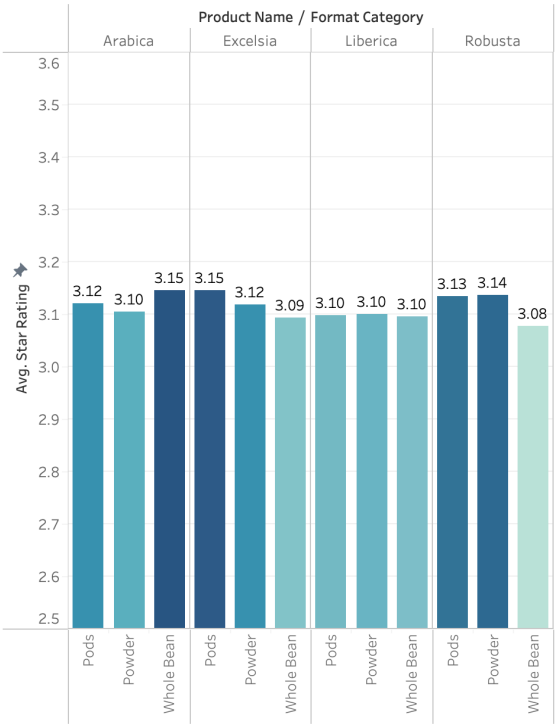
3. Business Insights

3.1. Customer Review

```
SELECT DISTINCT COUNT(ci.Client_ID) CLIENT_ID_COUNT,
        ci.County,
        cd.Delivery_Status,
        cd.Carrier,
        cr.Star_Rating,
        pc.Product_Name,
        pc.Format_Category

FROM Clients ci
LEFT JOIN Sales ss on ss.Client_ID = ci.Client_ID
LEFT JOIN Client_Deliveries cd ON ci.Client_ID = cd.Client_ID
LEFT JOIN Client_Reviews cr ON ci.Client_ID = cr.Client_ID
LEFT JOIN Financial_Transactions ft ON ft.Transaction_ID =
ss.Transaction_ID
LEFT JOIN Products pc ON ss.Product_ID = pc.Product_ID
where cd.Delivery_Status IN ('Delivered','Delayed') and cr.Star_Rating
!= 0
GROUP BY
ci.County,cd.Delivery_Status,cd.Carrier,cr.Star_Rating,pc.Product_Name,
pc.Format_Category
```

This query offers insights into customer review ratings based on different coffee types, formats, and delivery conditions. The ratings differ marginally across different coffee types, counties, delivery statuses, and carriers. The higher rating for delayed deliveries compared to delivered ones suggests potential anomalies or differences in customer expectations that warrant further investigation. Additionally, reviews are analysed based on customer location, delivery status, and the responsible carrier to identify trends and areas for improvement.



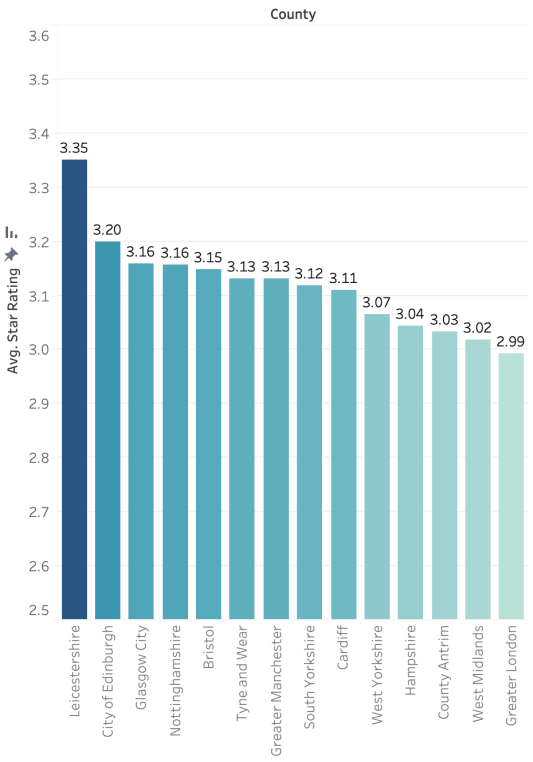


Figure 22. Average Rating Per Product (Left)

Figure 23. Average Rating by Customer County (Right)

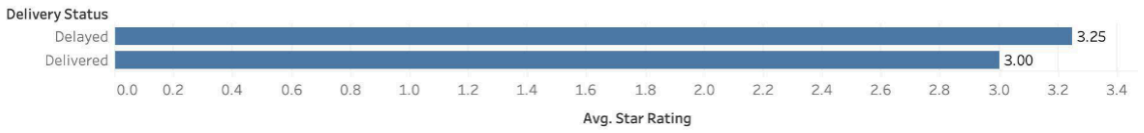


Figure 24. Average Rating by Status of Delivery

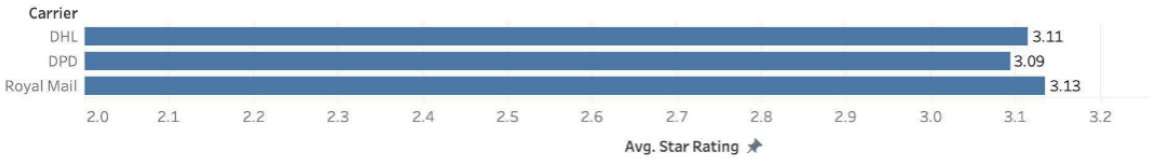


Figure 25. Average Rating by Carrier

3.2. Delivery Performance

```
SELECT
    ci.Client_ID,
    ss.Transaction_ID,
    ci.County,
    cd.Carrier,
    cd.Estimated_Delivery_Date,
    cd.Actual_Delivery_Date,
    julianday(Estimated_Delivery_Date) -
julianday(Actual_Delivery_Date) DELAY_DAYS,
    cd.Delivery_Status,
    ft.Is_Refunded

FROM Clients ci
LEFT JOIN Sales ss ON ss.Client_ID = ci.Client_ID
LEFT JOIN Client_Deliveries cd ON cd.Transaction_ID = ss.Transaction_ID
LEFT JOIN Client_Reviews cr ON cr.Client_ID = ci.Client_ID AND
cr.Product_ID = ss.Product_ID
LEFT JOIN Financial_Transactions ft ON ft.Transaction_ID =
ss.Transaction_ID
where cd.Delivery_Status IN ('Delivered','Delayed')
```

This query provides insights into delivery performance across different carriers, focusing on identifying carriers associated with late deliveries by analysing the difference between estimated and actual delivery times.

Figure 26 shows Royal Mail with the highest average delay (0.74 days), followed by DHL (0.41 days), while DPD averages early deliveries (-0.02 days). However, Figure 27 reveals DPD has the highest delay rate (11.65%), suggesting frequent but shorter delays, whereas DHL and Royal Mail experience fewer but longer delays. Royal Mail's high "Shipped" rate (30.19%) may indicate longer transit times, while cancellation rates remain low across all carriers.

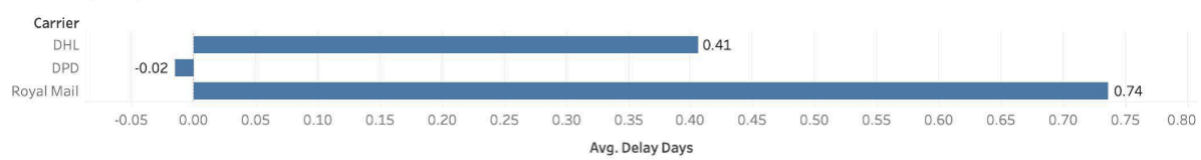


Figure 26. Average Delay Days by Carrier

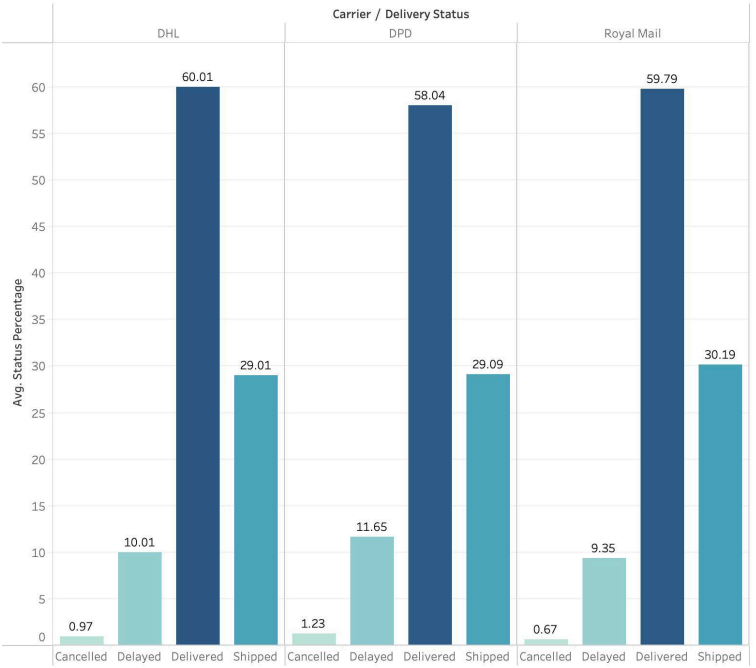


Figure 27. Average Delivery Status by Carrier

3.3. Financial Analysis

```
--  Refund_By_Product - Calculates total refund amount per product
category and name

WITH Refund_By_Product AS (
    SELECT
        pc.Product_Name,
        pc.Format_Category,
        SUM(ft.Refund_Amount) AS Total_Refund_Amount
    FROM Sales ss
    LEFT JOIN Products pc ON ss.Product_ID = pc.Product_ID
    LEFT JOIN Financial_Transactions ft ON ss.Transaction_ID =
ft.Transaction_ID
    WHERE ft.Refund_Amount IS NOT NULL AND ft.Is_Refunded = 'Yes'
    GROUP BY pc.Product_Name, pc.Format_Category
),
```

```
-- Revenue_By_Product - Calculates total revenue per product category
and name
Revenue_By_Product AS (
    SELECT
        pc.Product_Name,
        pc.Format_Category,
        SUM(ft.Payment_Amount) AS Total_Revenue
    FROM Sales ss
    LEFT JOIN Products pc ON ss.Product_ID = pc.Product_ID
    LEFT JOIN Financial_Transactions ft ON ss.Transaction_ID =
ft.Transaction_ID
    WHERE ft.Payment_Amount IS NOT NULL
    GROUP BY pc.Product_Name, pc.Format_Category
)
```

```
-- Combine the results using both Product_Name and Format_Category for
the join
SELECT
    rbp.Product_Name,
    rbp.Format_Category,  -- Use rbp.Format_Category to avoid potential
mismatch
    rbp.Total_Refund_Amount,
    rb.Total_Revenue,
    ROUND((rbp.Total_Refund_Amount * 100.0 / rb.Total_Revenue), 2) AS
Refund_Percentage
FROM Refund_By_Product rbp
LEFT JOIN Revenue_By_Product rb
    ON rbp.Product_Name = rb.Product_Name AND rbp.Format_Category =
rb.Format_Category;
```

This query gives financial performance insights by evaluating refund amounts concerning product categories. In addition, it compares refund amounts to total revenue earned to determine the impact of refunds on overall profitability.

Figures 28 and 29 analyse refund trends across product categories, showing Liberica-derived products have the highest refund costs (25,072), while other categories remain below 16,035, indicating they are more prone to damage and higher refund rates. This insight helps identify products requiring further analysis and quality improvements.

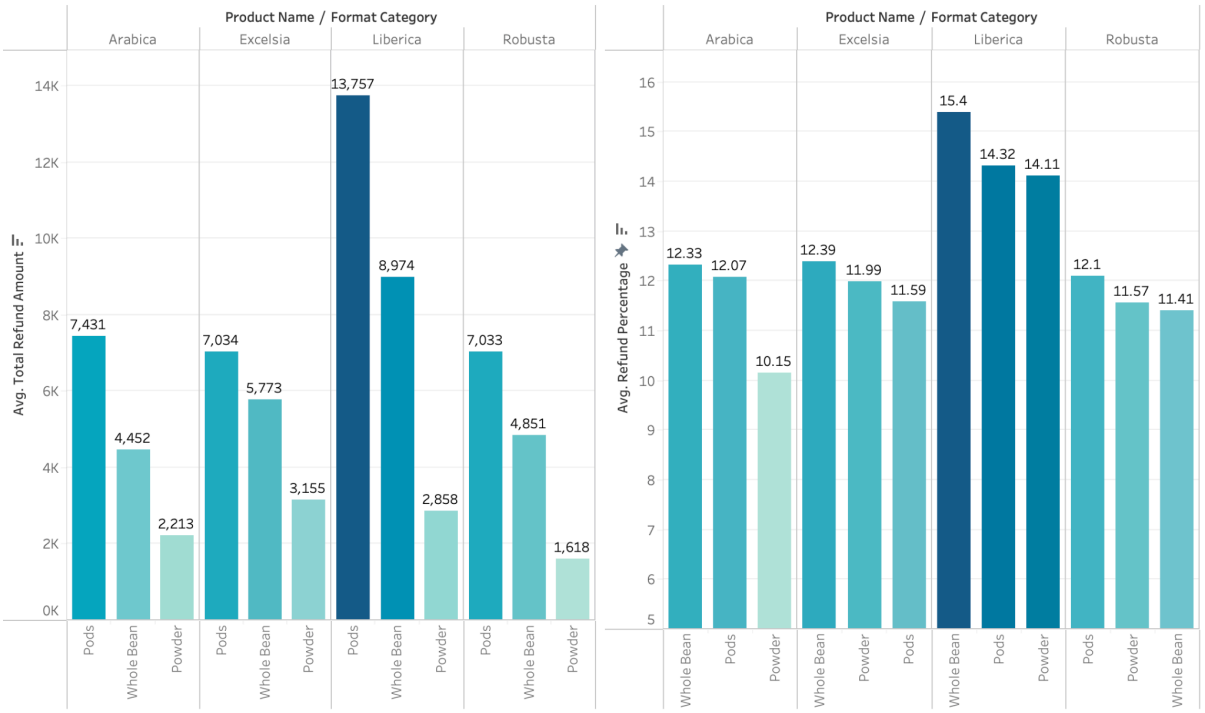


Figure 28. Average Refund Amount by Product Name and Category (Left)

Figure 29. Average Refund Percentage by Product Name and Category (Right)

3.4. Product Damage

```
SELECT
    cd.Carrier,
    cd.Delivery_Status,
    COUNT(*) AS Status_Count,
    (SELECT COUNT(*) FROM Client_Deliveries WHERE Carrier = cd.Carrier)
AS Total_Deliveries,
    ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM Client_Deliveries
WHERE Carrier = cd.Carrier), 2) AS Status_Percentage
FROM Client_Deliveries cd
WHERE cd.Carrier IS NOT NULL AND cd.Carrier <> ''
GROUP BY cd.Carrier, cd.Delivery_Status
```

```
SELECT
    pd.Damage_Type AS Damage_Type,
    'Refund Related' AS Category,
    COUNT(*) AS Status_Count
FROM Product_Damage pd
WHERE pd.Damage_Type IS NOT NULL
GROUP BY pd.Damage_Type
```

The first query investigates product damage trends and their financial impact. Figure 30 indicates that all issue types are reported at similar frequency while Figure 31 shows that almost 50% of transactions have a reported issue

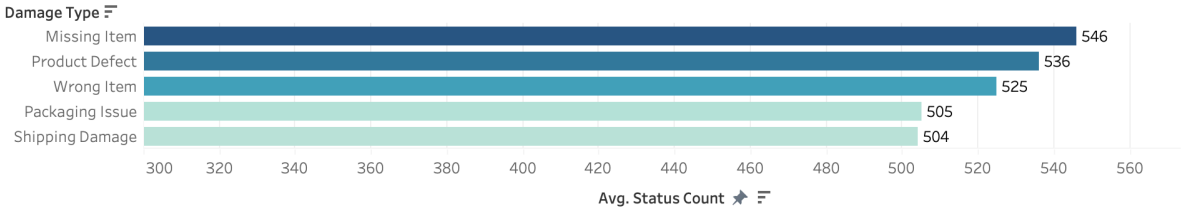


Figure 30. Damage Type (Count)

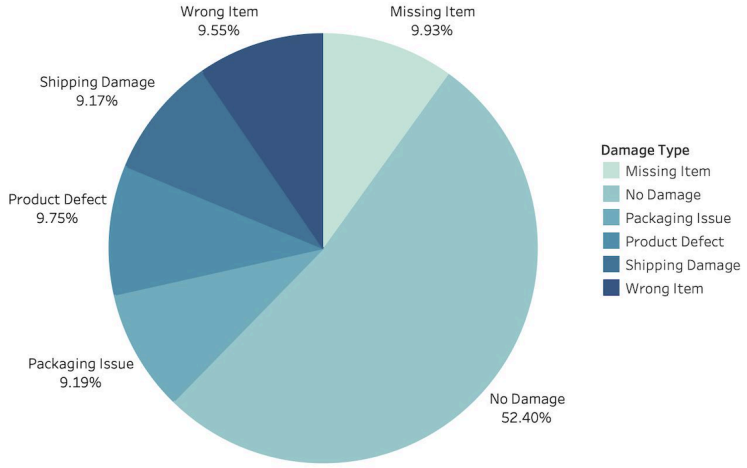


Figure 31. Damage Type (Percentage)

4. Key Takeaways

We faced challenges due to the uncommon nature of generating a fully synthetic database. To tackle this, we worked backward—first identifying the desired insights, then generating the necessary data, and finally designing the database.

For synthetic data generation, the first challenge was ensuring data generated aligned with the conclusions we wanted to draw. While we considered using methods such as Cholesky's Decomposition to induce correlation between sets of random variables, we deemed the method to be impractical for the task (Burgess, 2022). Instead we primarily used categorisation to subdivide our dataset. Then, assigned a different random distribution for each category to induce trends.

When importing the generated data into the SQL database, we faced issues with transitive dependencies on some tables. This was solved by splitting tables that were previously merged for simplicity. Otherwise, our well planned initial database design ensured simplifications would not jeopardise referential integrity.

To conclude, despite some challenges encountered throughout this task, thorough planning of our database design and a well-defined methodology for our data generation ensured successful creation of a realistic database.

References

Burgess, N. (2022) *Correlated Monte Carlo Simulation using Cholesky Decomposition*. Available at: https://ssrn.com/abstract=4066115 (Accessed: 18 March 2025).

Masteroast. (2024) *Home*. Available at: https://www.masteroast.co.uk/ (Accessed: 18 March 2025).

APPENDICES

APPENDIX A: Setup Code

```
#generate customer data
!pip install faker
#generate customer data
import pandas as pd
import numpy as np
import random
from faker import Faker
import uuid
import random
from itertools import cycle
from datetime import datetime, timedelta
import seaborn as sns
import matplotlib.pyplot as plt



# fake = Faker()
fake = Faker("en_GB")
Faker.seed(42)
random.seed(42)
```

APPENDIX B: Fake Data Generation Code

```
import sqlite3
import pandas as pd

# Function to connect to the database and enable foreign key support
def connect_to_db(db_name):
    conn = sqlite3.connect(db_name)
    cursor = conn.cursor()
    cursor.execute("PRAGMA foreign_keys = ON;")  # Enable foreign key
support
    conn.commit()
    return conn, cursor

# Connect to SQLite database (or create it)
conn, cursor = connect_to_db("business_data.db")

# Define the SQL table creation statements with foreign key constraints
```

```
table_creation_queries = [
    """CREATE TABLE IF NOT EXISTS Clients (
        Client_ID VARCHAR(36) PRIMARY KEY,
        First_Name VARCHAR(50),
        Last_Name VARCHAR(50),
        Email VARCHAR(100),
        Phone_Number VARCHAR(14),
        Street_Address VARCHAR(150),
        City VARCHAR(50),
        Postal_Code VARCHAR(4),
        County VARCHAR(50)
    )""",

    """CREATE TABLE IF NOT EXISTS Products (
        Product_ID VARCHAR(36) PRIMARY KEY,
        Product_Name VARCHAR(20),
        Format_Category VARCHAR(20),
        SKU VARCHAR(13),
        Price REAL,
        Launch_Date DATE
    )""",

    """CREATE TABLE IF NOT EXISTS Sales (
        Transaction_ID VARCHAR(36) PRIMARY KEY,
        Client_ID VARCHAR(36),
        Product_ID VARCHAR(36),
        Order_Date DATE,
        Quantity INTEGER,
        Price_Per_Item REAL,
        FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
    )""",

    """CREATE TABLE IF NOT EXISTS Financial_Transactions (
        Transaction_ID VARCHAR(36) PRIMARY KEY,
        Payment_Date DATE,
        Payment_Amount REAL,
        Is_Refunded VARCHAR(3),
        Refund_Date DATE,
        Refund_Amount REAL,
        Net_Amount REAL,
        Net_Amount_Without_Shipping REAL,
        Payment_Method VARCHAR(20),
```

```
    Transaction_Status VARCHAR(20),
    Shipping_Cost REAL,
    FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID)
)""",

"""CREATE TABLE IF NOT EXISTS Client_Deliveries (
    Client_Delivery_ID VARCHAR(36) PRIMARY KEY,
    Transaction_ID VARCHAR(36),
    Client_ID VARCHAR(36),
    Product_ID VARCHAR(36),
    Delivery_Status VARCHAR(15),
    Carrier VARCHAR(20),
    Estimated_Delivery_Date DATE,
    Actual_Delivery_Date DATE,
    FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID),
    FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
    FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
)""",

"""CREATE TABLE IF NOT EXISTS Client_Reviews (
    Review_ID VARCHAR(36) PRIMARY KEY,
    Client_ID VARCHAR(36),
    Transaction_ID VARCHAR(36),
    Product_ID VARCHAR(36),
    Review_Date DATE,
    Star_Rating INTEGER,
    FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
    FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID),
    FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
)""",

"""CREATE TABLE IF NOT EXISTS Product_Damage (
    Damage_Report_ID VARCHAR(36) PRIMARY KEY,
    Transaction_ID VARCHAR(36),
    Client_ID VARCHAR(36),
    Product_ID VARCHAR(36),
    Review_ID VARCHAR(36),
    Client_Delivery_ID VARCHAR(36),
    Damage_Type VARCHAR(20),
    Resolution_Status VARCHAR(15),
    Resolution_Date DATE,
    FOREIGN KEY (Transaction_ID) REFERENCES Sales(Transaction_ID),
    FOREIGN KEY (Client_ID) REFERENCES Clients(Client_ID),
```

```
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID),
        FOREIGN KEY (Review_ID) REFERENCES Client_Reviews(Review_ID),
        FOREIGN KEY (Client_Delivery_ID) REFERENCES
Client_Deliveries(Client_Delivery_ID)
    )""",

    """CREATE TABLE IF NOT EXISTS Distributors (
        Distributor_ID VARCHAR(8) PRIMARY KEY,
        Distributor_Name VARCHAR(50),
        Email VARCHAR(100),
        Phone_Number VARCHAR(14),
        Street_Address VARCHAR(150),
        City VARCHAR(50),
        County VARCHAR(50),
        Postcode VARCHAR(10),
        Country VARCHAR(60)
    )""",

    """CREATE TABLE IF NOT EXISTS Distributor_Supply (
        Batch_ID VARCHAR(6) PRIMARY KEY,
        Distributor_ID VARCHAR(8),
        Product_ID VARCHAR(36),
        Batch_Quantity INTEGER,
        Delivery_Status VARCHAR(15),
        Shipping_Method VARCHAR(10),
        Estimated_Arrival_Date DATE,
        Actual_Arrival_Date DATE,
        FOREIGN KEY (Distributor_ID) REFERENCES
Distributors(Distributor_ID),
        FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
    )"""
]

# Execute each table creation query
for query in table_creation_queries:
    cursor.execute(query)


# Commit changes to save table structures
conn.commit()
```

APPENDIX C: Client Table Data Generation

```
import pandas as pd
import random
import uuid
from faker import Faker

# Set seed for reproducibility
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
fake = Faker()
Faker.seed(RANDOM_SEED)

# Dictionary mapping UK cities to postal codes and counties
uk_locations = {
    "London": {"Postal_Code": "EC1A", "County": "Greater London"},
    "Manchester": {"Postal_Code": "M1", "County": "Greater Manchester"},
    "Birmingham": {"Postal_Code": "B1", "County": "West Midlands"},
    "Leeds": {"Postal_Code": "LS1", "County": "West Yorkshire"},
    "Sheffield": {"Postal_Code": "S1", "County": "South Yorkshire"},
    "Bristol": {"Postal_Code": "BS1", "County": "Bristol"},
    "Newcastle upon Tyne": {"Postal_Code": "NE1", "County": "Tyne and
Wear"},
    "Nottingham": {"Postal_Code": "NG1", "County": "Nottinghamshire"},
    "Leicester": {"Postal_Code": "LE1", "County": "Leicestershire"},
    "Glasgow": {"Postal_Code": "G1", "County": "Glasgow City"},
    "Edinburgh": {"Postal_Code": "EH1", "County": "City of Edinburgh"},
    "Cardiff": {"Postal_Code": "CF10", "County": "Cardiff"},
    "Belfast": {"Postal_Code": "BT1", "County": "County Antrim"},
    "Southampton": {"Postal_Code": "SO14", "County": "Hampshire"}
}

# Custom email generator for Gmail/iCloud
def generate_email(first_name, last_name):
    domain = random.choice(["gmail.com", "icloud.com"])
    return f"{first_name.lower()}.{last_name.lower()}@{domain}"

# Custom UK phone number generator
def generate_uk_phone():
    return f"+44 7{random.randint(0, 9)}{random.randint(10000000,
99999999)}"
```

```
# Generate client data
num_records = 1000
clients = []
clients_ids_list = []

for _ in range(num_records):
    first_name = fake.first_name()
    last_name = fake.last_name()
    client_id = str(uuid.uuid4())
    clients_ids_list.append(client_id)

    # Randomly select a city and extract its details
    city, details = random.choice(list(uk_locations.items()))
    postcode = details["Postal_Code"]
    county = details["County"]

    clients.append([
        client_id,                          # Unique Client ID
        first_name,                         # First Name
        last_name,                          # Last Name
        generate_email(first_name, last_name),  # Gmail/iCloud Email
        generate_uk_phone(),                # UK Phone Number
        fake.street_address(),              # UK Address
        city,                               # Randomly chosen city
        postcode,                           # Corresponding postcode
        county                              # County
    ])

# Create DataFrame
df_clients = pd.DataFrame(
    clients,
    columns=["Client_ID", "First_Name", "Last_Name", "Email",
             "Phone_Number", "Street_Address", "City", "Postal_Code",
"County"]
)

# Improved function to insert data into SQLite table
def insert_data_to_table(df, table_name):


    try:
        # Clear existing data to avoid duplication
        cursor.execute(f"DELETE FROM {table_name};")
```

```
        conn.commit()

        # Insert data row-by-row
        columns = ', '.join(df.columns)
        placeholders = ', '.join(['?' for _ in df.columns])
        query = f"INSERT INTO {table_name} ({columns}) VALUES
({placeholders})"

        for row in df.itertuples(index=False):
            cursor.execute(query, tuple(row))

        conn.commit()
    except Exception as e:
        pass  # Suppress error messages

# Insert data to Clients table
insert_data_to_table(df_clients, "Clients")
```

APPENDIX D: Products Table Data Generation

```
product_combinations = [
    ("Arabica", "Powder"), ("Arabica", "Whole Bean"), ("Arabica",
"Pods"),
    ("Robusta", "Powder"), ("Robusta", "Whole Bean"), ("Robusta",
"Pods"),
    ("Excelsia", "Powder"), ("Excelsia", "Whole Bean"), ("Excelsia",
"Pods"),
    ("Liberica", "Powder"), ("Liberica", "Whole Bean"), ("Liberica",
"Pods")
]

# Price ranges for combinations
price_range = {
    ("Arabica", "Powder"): (5, 10),
    ("Arabica", "Whole Bean"): (12, 20),
    ("Arabica", "Pods"): (22, 30),

    ("Robusta", "Powder"): (4, 9),
    ("Robusta", "Whole Bean"): (10, 18),
    ("Robusta", "Pods"): (20, 28),

    ("Excelsia", "Powder"): (6, 12),
```

```
    ("Excelsia", "Whole Bean"): (14, 22),
    ("Excelsia", "Pods"): (25, 32),

    ("Liberica", "Powder"): (7, 13),
    ("Liberica", "Whole Bean"): (16, 25),
    ("Liberica", "Pods"): (28, 35)
}

# List of dates
dates = [
    "April 5, 2015", "October 22, 2017", "June 14, 2019", "December 8,
2021",
    "March 30, 2016", "July 11, 2014", "January 3, 2018", "September 25,
2020",
    "November 17, 2022", "August 9, 2014", "May 4, 2023", "February 19,
2024"
]

# Generate product data for each combination (one row per combination)
products = []
product_id_list = []

for (product_name, format_category), date in zip(product_combinations,
dates):
    product_id = str(uuid.uuid4())
    product_id_list.append(product_id)
    sku =
f"{product_name[:3].upper()}-{format_category[:3].upper()}-{random.rand
int(10000, 99999)}"
    price = round(random.uniform(*price_range[(product_name,
format_category)]), 2)

    # Append the data with corresponding date
    products.append([
        product_id,             # Primary Key - Product_ID
        product_name,           # Product Name
        format_category,        # Format Category
        sku,                    # SKU Code
        price,                  # Price
        date                    # Launch_date
    ])

# Create DataFrame
```

```
df_products = pd.DataFrame(
    products,
    columns=["Product_ID", "Product_Name", "Format_Category", "SKU",
"Price",
             "Launch_date"]
)

# Display the resulting DataFrame
df_products


# Save as CSV
df_products.to_csv("products_catalog.csv", index=False)

# For Google Colab users: Download the dataset
#from google.colab import files
#files.download("products_catalog.csv")
#insert data
insert_data_to_table(df_products, "Products")
```

APPENDIX E: Sales Table Data Generation

```
transactions = []
transaction_id_list = []
clients = clients_ids_list.copy()
product_ids = product_id_list.copy()

# Ensure each client has between 1 and 10 transactions
for client_id in clients:
    num_orders_for_client = random.randint(1, 10)  # Random number of
orders between 1 and 10
    for _ in range(num_orders_for_client):
        transaction_id = str(uuid.uuid4())  # Unique Transaction ID
        transaction_id_list.append(transaction_id)
        product_id = random.choice(df_products['Product_ID'])  # Random
product from the list of Product_IDs
        order_date = fake.date_this_decade()  # Random order date within
this decade
        quantity = random.randint(1, 10)  # Random quantity between 1
and 10
        # Fetch price from product catalog
        price_per_item = df_products[df_products['Product_ID'] ==
product_id]['Price'].values[0]
```

```
        transactions.append([
            transaction_id,      # Transaction_ID (Primary Key)
            client_id,           # Client_ID (Foreign Key → Client
Information)
            product_id,          # Product_ID (Foreign Key → Product
Catalog Information)
            order_date,          # Order_Date
            quantity,            # Quantity
            price_per_item       # Price_Per_Item (from Product Catalog)
        ])

# If the number of transactions generated is less than
num_transactions, add extra random transactions
remaining_transactions = 100 - len(transactions)  # Calculate how many
more transactions are needed

for _ in range(remaining_transactions):
    transaction_id = str(uuid.uuid4())  # Unique Transaction ID
    if transaction_id not in transaction_id_list:
      transaction_id_list.append(transaction_id)
    client_id = random.choice(clients)  # Random client from the list
    product_id = random.choice(df_products['Product_ID'])  # Random
product from the list
    order_date = fake.date_this_decade()  # Random order date within
this decade
    quantity = random.randint(1, 10)  # Random quantity between 1 and 10
    # Fetch price from product catalog
    price_per_item = df_products[df_products['Product_ID'] ==
product_id]['Price'].values[0]

    transactions.append([
        transaction_id,      # Transaction_ID (Primary Key)
        client_id,           # Client_ID (Foreign Key → Client
Information)
        product_id,          # Product_ID (Foreign Key → Product Catalog
Information)
        order_date,          # Order_Date
        quantity,            # Quantity
        price_per_item       # Price_Per_Item (from Product Catalog)
    ])

# Create DataFrame for transactions
```

```
df_transactions = pd.DataFrame(
    transactions,
    columns=["Transaction_ID", "Client_ID", "Product_ID", "Order_Date",
"Quantity", "Price_Per_Item"]
)

# Display the resulting DataFrame
df_transactions.head(50)


# # Save as CSV
df_transactions.to_csv("sales_transaction.csv", index=False)

# # For Google Colab users: Download the dataset
#from google.colab import files
#files.download("sales_transaction.csv")
#insert data
insert_data_to_table(df_transactions, "Sales")
```

APPENDIX F: Financial Transactions Table Data Generation

```
# Sample Payment Methods and Transaction Statuses
payment_methods = ["Credit Card", "Debit Card", "PayPal", "Bank
Transfer", "Cash"]

# Dictionary mapping UK cities to specific shipping costs
shipping_cost_by_city = {
    "London": 5.0, "Manchester": 6.0, "Birmingham": 4.5, "Leeds": 5.5,
"Sheffield": 4.0,
    "Bristol": 5.0, "Newcastle upon Tyne": 6.5, "Nottingham": 4.0,
"Leicester": 5.5,
    "Glasgow": 7.0, "Edinburgh": 6.0, "Cardiff": 5.0, "Belfast": 8.0,
"Southampton": 4.5
}

# Distribution for Transaction Status
transaction_status_distribution = [
    ("Completed", 0.6),
    ("Pending", 0.2),
    ("Partial Refund", 0.15),
    ("Full Refund", 0.05)
]
```

```
# Generate Financial Transactions Data
transactions = []

# Generate Payment Data
financial_transactions = []

# Link sales transactions to financial transactions
for index, row in df_transactions.iterrows():
    transaction_id = row["Transaction_ID"]
    payment_date = row["Order_Date"]
    payment_amount = round(row["Quantity"] * row["Price_Per_Item"], 2)

    # Shipping cost logic based on Client_ID's city
    client_city = df_clients[df_clients["Client_ID"] ==
row["Client_ID"]]["City"].values[0]
    shipping_cost = shipping_cost_by_city.get(client_city, 5.0)  #
Default to £5 if city not found

    # Refund logic
    transaction_status = random.choices(
        [status[0] for status in transaction_status_distribution],
        [status[1] for status in transaction_status_distribution]
    )[0]

    if transaction_status in ["Partial Refund", "Full Refund"]:
        is_refunded = "Yes"
        refund_quantity = row["Quantity"] if transaction_status == "Full
Refund" else random.randint(1, row["Quantity"])
        refund_amount = round(refund_quantity * row["Price_Per_Item"],
2)-shipping_cost
        refund_date = payment_date + pd.Timedelta(days=10)
    else:
        is_refunded = "No"
        refund_amount = 0
        refund_date = None

    # Net Amount Calculation (excluding shipping cost in refund logic)
    net_amount = round(payment_amount - refund_amount, 2)
    net_amount_with_shipping = round(net_amount - (0 if is_refunded ==
"Yes" else shipping_cost), 2)

    # Random Payment Method
    payment_method = random.choice(payment_methods)
```

```
    financial_transactions.append([
        transaction_id,      # Transaction_ID (Primary Key)
        payment_date,        # Payment_Date
        payment_amount,      # Payment_Amount
        is_refunded,         # Is_Refunded (Yes / No)
        refund_date,         # Refund_Date
        refund_amount,       # Refund_Amount
        net_amount,          # Net_Amount (Payment Amount - Refund
Amount)
        net_amount_with_shipping,  # Net_Amount_With_Shipping
        payment_method,      # Payment_Method
        transaction_status, # Transaction_Status
        shipping_cost        # Shipping_Cost
    ])

# Create DataFrame
df_financial_transactions = pd.DataFrame(
    financial_transactions,
    columns=["Transaction_ID", "Payment_Date", "Payment_Amount",
"Is_Refunded", "Refund_Date",
             "Refund_Amount", "Net_Amount",
"Net_Amount_Without_Shipping", "Payment_Method", "Transaction_Status",
"Shipping_Cost"]
)
# Display sample data
df_financial_transactions.head(20)

# # Save as CSV
df_financial_transactions.to_csv("financial_transactions.csv",
index=False)

# # For Google Colab users: Download the dataset
#from google.colab import files
#files.download("financial_transactions.csv")
#insert data
insert_data_to_table(df_financial_transactions,
"Financial_Transactions")
```

APPENDIX G: Client Deliveries Table Data Generation

```
transactions = transaction_id_list.copy()
client_delivery_ids_list = []
```

```
client_deliveries = []

for index, row in df_financial_transactions.iterrows():
    transaction_id = row["Transaction_ID"]

    # Fetch corresponding Client_ID and Product_ID from df_transactions
    client_info = df_transactions[df_transactions["Transaction_ID"] ==
transaction_id]

    if not client_info.empty:
        client_id = client_info.iloc[0]["Client_ID"]
        product_id = client_info.iloc[0]["Product_ID"]
    else:
        # If no matching data found, skip this transaction
        continue

    client_delivery_id = str(uuid.uuid4())
    client_delivery_ids_list.append(client_delivery_id)

    # Set delivery details based on Transaction Status
    if row["Transaction_Status"] == "Pending":
        delivery_status = "Pending"
        carrier = ""  # No carrier assigned
        estimated_delivery_date = None
        actual_delivery_date = None
    else:
        delivery_status = random.choices(
            ["Shipped", "Delivered", "Delayed", "Cancelled"],
            weights=[0.29, 0.59, 0.1, 0.01]
        )[0]
        carrier = random.choice(["DHL", "Royal Mail", "DPD"])

        # Order date from df_transactions
        order_date = client_info.iloc[0]["Order_Date"]

        # Set Estimated Delivery Date
        estimated_delivery_date = order_date +
timedelta(days=random.randint(1, 10)) if delivery_status in ("Shipped",
"Delivered", "Delayed") else None

        # Set Actual Delivery Date
        actual_delivery_date = None
```

```
        if delivery_status == "Delivered":
            if random.random() < 0.95:  # 95% chance to be on or before
estimated
                actual_delivery_date = estimated_delivery_date -
timedelta(days=random.randint(0, 10))
                if actual_delivery_date < order_date:
                    actual_delivery_date = order_date
            else:
                actual_delivery_date = estimated_delivery_date +
timedelta(days=random.randint(1, 10))
        elif delivery_status == "Delayed":
            actual_delivery_date = estimated_delivery_date +
timedelta(days=random.uniform(7, 30))

    client_deliveries.append([
        transaction_id,     # Transaction_ID (FKey)
        client_id,          # Client_ID (Foreign Key → Client
Information)
        product_id,         # Product_ID (Foreign Key → Product Catalog
Information)
        client_delivery_id, # PKey
        delivery_status,
        carrier,
        estimated_delivery_date,
        actual_delivery_date
    ])

# Create DataFrame for client deliveries
df_client_deliveries = pd.DataFrame(
    client_deliveries,
    columns=["Transaction_ID", "Client_ID", "Product_ID",
"Client_Delivery_ID",
             "Delivery_Status", "Carrier", "Estimated_Delivery_Date",
"Actual_Delivery_Date"]
)

# Display the resulting DataFrame
df_client_deliveries.head(50)

# Save as CSV (optional)
df_client_deliveries.to_csv("client_deliveries.csv", index=False)

# # For Google Colab users: Download the dataset
```

```
#from google.colab import files
#files.download("client_deliveries.csv")
#insert data
insert_data_to_table(df_client_deliveries, "Client_Deliveries")
```

APPENDIX H: Client Reviews Table Data Generation

```
# Number of fake reviews to generate (based on the number of delivered
transactions)
delivered_transactions =
df_client_deliveries[df_client_deliveries['Delivery_Status'] ==
'Delivered']
num_reviews = len(delivered_transactions)

# Generate random Review_IDs (UUIDs for uniqueness)
review_ids = [str(uuid.uuid4()) for _ in range(num_reviews)]

# Extract relevant columns for delivered transactions
client_ids = delivered_transactions["Client_ID"].values
transaction_ids = delivered_transactions["Transaction_ID"].values
product_ids = delivered_transactions["Product_ID"].values  # Added
Product_ID
actual_delivery_dates =
delivered_transactions["Actual_Delivery_Date"].values
estimated_delivery_dates =
delivered_transactions["Estimated_Delivery_Date"].values

# Merge refund statuses to ensure alignment
delivered_with_refund = delivered_transactions.merge(
   df_financial_transactions[["Transaction_ID", "Is_Refunded"]],
   on="Transaction_ID",
   how="left"
).fillna({"Is_Refunded": "No"})  # Fill missing refund status with 'No'

# Extract refund statuses after merging
refund_statuses = delivered_with_refund["Is_Refunded"].values

# Generate Review Dates (2 or 3 days after Actual Delivery Date)
review_dates = [
   actual_delivery_dates[i] + timedelta(days=random.randint(2, 3))
   for i in range(num_reviews)
]
```

```
# Define a function to assign ratings based on refund status and
delivery timing
def assign_review_rating(refund_status, actual_date, estimated_date):
    if refund_status == "Yes":  # Partial or full refund
        return random.choice([1, 2])  # Low rating for refunded orders
    elif actual_date > estimated_date:  # Delayed delivery but not
refunded
        return 4  # Fixed 4-star rating for delayed delivery
    else:
        return random.choice([4, 5])  # High rating for non-refunded
on-time orders

# Assign ratings based on refund status and delivery timing
star_ratings = [
    assign_review_rating(refund_statuses[i], actual_delivery_dates[i],
estimated_delivery_dates[i])
    for i in range(num_reviews)
]

# Create the final DataFrame for reviews
df_reviews = pd.DataFrame({
    "Review_ID": review_ids,
    "Client_ID": client_ids,
    "Transaction_ID": transaction_ids,
    "Product_ID": product_ids,  # Added Product_ID
    "Review_Date": review_dates,
    "Star_Rating": star_ratings
})

# Display the generated reviews DataFrame
df_reviews.head(10)

# Save as CSV (optional)
df_reviews.to_csv("generated_reviews.csv", index=False)

# For Google Colab users: Download the dataset (optional)
#from google.colab import files
#files.download("generated_reviews.csv")
#insert data
insert_data_to_table(df_reviews, "Client_Reviews")
```

APPENDIX I: Product Damage Table Data Generation

```
def generate_product_damage_data(conn, num_records=1000):
    # Fetch existing IDs from referenced tables
    existing_transaction_ids = pd.read_sql_query("SELECT Transaction_ID
FROM Sales;", conn)['Transaction_ID'].tolist()
    existing_client_ids = pd.read_sql_query("SELECT Client_ID FROM
Clients;", conn)['Client_ID'].tolist()
    existing_product_ids = pd.read_sql_query("SELECT Product_ID FROM
Products;", conn)['Product_ID'].tolist()
    existing_review_ids = pd.read_sql_query("SELECT Review_ID FROM
Client_Reviews;", conn)['Review_ID'].tolist()
    existing_client_delivery_ids = pd.read_sql_query("SELECT
Client_Delivery_ID FROM Client_Deliveries;",
conn)['Client_Delivery_ID'].tolist()

    # Ensure IDs are not empty before proceeding
    if not (existing_transaction_ids and existing_client_ids and
existing_product_ids and existing_review_ids and
existing_client_delivery_ids):
        print("One or more required tables are empty. Please check your
database.")
        return

    damage_reports = []

    for _ in range(num_records):
        # Generate unique Damage_Report_ID
        damage_report_id = str(uuid.uuid4())

        # Randomly select valid IDs from existing tables
        try:
            transaction_id = random.choice(existing_transaction_ids)
            client_id = random.choice(existing_client_ids)
            product_id = random.choice(existing_product_ids)
            review_id = random.choice(existing_review_ids)
            client_delivery_id =
random.choice(existing_client_delivery_ids)
        except IndexError:
            print("Error: One or more referenced tables are empty.")
            return

        # Generate random Damage_Type
        damage_type = random.choice([
            "Product Defect", "Shipping Damage", "Wrong Item",
```

```
            "Missing Item", "Packaging Issue"
        ])

        # Generate Resolution Status and Resolution Date if applicable
        resolution_status = random.choice(["Pending", "In Progress",
"Resolved"])
        resolution_date = fake.date_this_decade() if resolution_status
== "Resolved" else None

        # Append generated data to the damage_reports list
        damage_reports.append([
            damage_report_id,
            transaction_id,
            client_id,
            product_id,
            review_id,
            client_delivery_id,
            damage_type,
            resolution_status,
            resolution_date
        ])

    # Create DataFrame from generated data
    df_damage_reports = pd.DataFrame(
        damage_reports,
        columns=[
            "Damage_Report_ID", "Transaction_ID", "Client_ID",
"Product_ID",
            "Review_ID", "Client_Delivery_ID", "Damage_Type",
            "Resolution_Status", "Resolution_Date"
        ]
    )

    # Save generated data to a CSV file
    df_damage_reports.to_csv("damage_reports.csv", index=False)

    # Insert data into Product_Damage table
    insert_data_to_table(df_damage_reports, "Product_Damage", conn)
```

APPENDIX J: Distributors Table Data Generation

```
import sqlite3
```

```
import pandas as pd
import random
import uuid
from faker import Faker
import time
from datetime import datetime, timedelta

# Initialize Faker
fake = Faker()

# Function to create a fresh database connection
def create_connection():
    try:
        conn = sqlite3.connect("business_data.db", timeout=30)
        conn.execute("PRAGMA foreign_keys = ON;")
        conn.execute("PRAGMA synchronous = OFF;")
        conn.execute("PRAGMA journal_mode = DELETE;")
        return conn
    except Exception as e:
        print(f"Error creating connection: {e}")
        return None

# Function to insert data with automatic connection handling
def insert_data_to_table(df, table_name, max_retries=5):
    retries = 0
    while retries < max_retries:
        try:
            conn = create_connection()
            if conn is None:
                raise Exception("Failed to establish a connection.")
            cursor = conn.cursor()

            # Clear existing data
            cursor.execute(f"DELETE FROM {table_name};")
            conn.commit()

            # Insert data in a single transaction (more efficient)
            conn.execute("BEGIN TRANSACTION;")
            columns = ', '.join(df.columns)
            placeholders = ', '.join(['?' for _ in df.columns])
            query = f"INSERT INTO {table_name} ({columns}) VALUES
({placeholders})"
```

```
            cursor.executemany(query, df.values.tolist())
            conn.commit()

            # Clean up the database to prevent locks
            cursor.execute("VACUUM;")
            conn.commit()

            conn.close()
            break  # Break the loop if successful

        except sqlite3.OperationalError as e:
            if "database is locked" in str(e):
                retries += 1
                print(f"Database is locked. Retrying
{retries}/{max_retries}...")
                time.sleep(5)
            else:
                print(f"Error inserting data into {table_name}: {e}")
                break

        except Exception as e:
            print(f"Error inserting data into {table_name}: {e}")
            break

        finally:
            if conn:
                conn.close()  # Ensure connection is closed even if an
error occurs

# Generate distributors data
def generate_distributors_data():
   countries = {
        "Brazil": {"prefix": "B", "phone_code": "+55",
"postcode_format": "#####-###"},
        "Colombia": {"prefix": "C", "phone_code": "+57",
"postcode_format": "######"},
        "Vietnam": {"prefix": "V", "phone_code": "+84",
"postcode_format": "######"},
        "India": {"prefix": "I", "phone_code": "+91", "postcode_format":
"######"},
        "Mexico": {"prefix": "M", "phone_code": "+52",
"postcode_format": "#####" }
   }
```

```
    data = []
    for country, details in countries.items():
        distributor_id = details["prefix"] + str(random.randint(1000000,
9999999))
        name = fake.company()
        email = name.replace(" ", "").replace(",", "").replace(".",
"").lower() + "@gmail.com"
        phone_number = f"{details['phone_code']}
{random.randint(1000000000, 9999999999)}"
        street_address = fake.street_address()
        city = fake.city()
        county = fake.state()
        postcode = fake.postcode()[:len(details["postcode_format"])]

        data.append([
            distributor_id, name, email, phone_number,
            street_address, city, county, postcode, country
        ])

    df_distributor_info = pd.DataFrame(data, columns=[
        "Distributor_ID", "Distributor_Name", "Email", "Phone_Number",
        "Street_Address", "City", "County", "Postcode", "Country"
    ])
    return df_distributor_info

# Generate distributor data
df_distributor_info = generate_distributors_data()

# Insert data into Distributors table
insert_data_to_table(df_distributor_info, "Distributors")
```

APPENDIX K: Distributor Supply Table Data Generation

```
# Load Distributor Information Data
distributor_df = df_distributor_info.copy()
distributor_ids = distributor_df["Distributor_ID"].tolist()

# Dummy Product List (Replace with your actual product ID list)
product_ids = [str(uuid.uuid4()) for _ in range(10)]

# Initialize the global batch counter
```

```
batch_counter = 0

# Function to generate distributor supply data
def generate_supply_data(distributor_ids, product_ids):
    global batch_counter  # Set as global variable
    statuses = ["Delivered"] * 50 + ["Pending"] * 20 + ["Shipped"] * 15
+ ["Delayed"] * 10 + ["Cancelled"] * 5
    shipping_methods = ["Air"] * 50 + ["Road"] * 30 + ["Water"] * 20
    supply_data = []

    for distributor_id in distributor_ids:
        for product_id in product_ids:
            batch_id = f"B{batch_counter:05d}"  # Format as B0000001,
B0000002, etc.
            batch_counter += 1  # Increment counter

            batch_quantity = random.randint(200, 400)
            delivery_status = random.choice(statuses)
            shipping_method = random.choice(shipping_methods)

            base_date = datetime(2024, random.randint(1, 12),
random.randint(1, 28))
            estimated_arrival = base_date +
timedelta(days=random.randint(1, 30))

            if delivery_status in ["Pending", "Cancelled"]:
                actual_arrival = None
            elif shipping_method == "Air":
                actual_arrival = estimated_arrival +
timedelta(days=random.choice([-3, -2, -1, 0]))
            elif shipping_method == "Road":
                actual_arrival = estimated_arrival +
timedelta(days=random.choice([-1, 0, 1, 2]))
            elif shipping_method == "Water":
                actual_arrival = estimated_arrival +
timedelta(days=random.choice([1, 2, 3, 4, 5]))

            supply_data.append({
                "Batch_ID": batch_id,
                "Distributor_ID": distributor_id,
                "Product_ID": product_id,
                "Batch_Quantity": batch_quantity,
                "Delivery_Status": delivery_status,
```

```
                "Shipping_Method": shipping_method,
                "Estimated_Arrival_Date":
estimated_arrival.strftime('%Y-%m-%d'),
                "Actual_Arrival_Date":
actual_arrival.strftime('%Y-%m-%d') if actual_arrival else None
            })

    return pd.DataFrame(supply_data)

# Generate supply data
supply_df = generate_supply_data(distributor_ids, product_ids)

# Save as CSV (optional)
supply_df.to_csv("distributor_supply_data.csv", index=False)

# Insert data into Distributor_Supply table
insert_data_to_table(supply_df, "Distributor_Supply")
```

APPENDIX L: Database Normalisation Check Code

```
import sqlite3
import pandas as pd

# Connect to your SQLite database
conn = sqlite3.connect("business_data.db")

# Retrieve all table names
query = "SELECT name FROM sqlite_master WHERE type='table';"
tables = pd.read_sql_query(query, conn)
table_names = tables['name'].tolist()

# Store the check results
results = []

# Iterate through all tables
for table_name in table_names:
    # Check the table structure (columns information)
    query_structure = f"PRAGMA table_info({table_name});"
    df_structure = pd.read_sql_query(query_structure, conn)

    # Check foreign key information of the table
    query_foreign_keys = f"PRAGMA foreign_key_list({table_name});"
```

```
    df_foreign_keys = pd.read_sql_query(query_foreign_keys, conn)

    # Extract column information
    num_columns = len(df_structure)
    primary_keys = df_structure[df_structure['pk'] > 0]['name'].tolist()
    foreign_keys = df_foreign_keys['from'].tolist()
    non_key_columns = [col for col in df_structure['name'] if col not in
primary_keys]

    print(f"\n=== Checking Table: {table_name} ===")

    # === First Normal Form (1NF) Check ===
    first_nf = all(df_structure['type'].apply(lambda x: x not in
['BLOB', 'LIST', 'ARRAY']))
    if first_nf:
        print(f"1NF (Atomicity): All fields are atomic.")
    else:
        print(f"1NF (Atomicity): Non-atomic fields detected.")

    # === Second Normal Form (2NF) Check ===
    if len(primary_keys) == 1:
        second_nf = True  # A single primary key usually means full
dependency
    else:
        second_nf = all(col in primary_keys or col in foreign_keys for
col in non_key_columns)

    if second_nf:
        print(f"2NF (Full Dependency): No partial dependencies found.")
    else:
        print(f"2NF (Full Dependency): Partial dependencies found.")

    # === Third Normal Form (3NF) Check ===
    third_nf = True
    if len(primary_keys) == 1:
        for column in non_key_columns:
            if column in foreign_keys:
                continue  # Foreign keys are fine for 3NF
            if column.endswith("_ID"):  # Potentially a transitive
dependency
                third_nf = False
                break
    else:
```

```
        for column in non_key_columns:
            if column not in primary_keys and column not in
foreign_keys:
                third_nf = False
                break

    if third_nf:
        print(f"3NF (No Transitive Dependency): No transitive
dependencies found.")
    else:
        print(f"3NF (No Transitive Dependency): Transitive dependencies
detected.")

    # Store the results for each table
    results.append({
        "Table": table_name,
        "Primary Keys": primary_keys,
        "Foreign Keys": foreign_keys,
        "1NF (Atomicity)": "0" if first_nf else "1",
        "2NF (Full Dependency)": "0" if second_nf else "1",
        "3NF (No Transitive Dependency)": "0" if third_nf else "1"
    })

# Close the database connection
conn.close()

# Display the results
df_results = pd.DataFrame(results)
print("\n=== Summary of Normalization Checks ===")
print(df_results)
```

APPENDIX M: SQL Querying for 1st KPI (Customer Review)

```
# Connect to the SQLite database
conn = sqlite3.connect("business_data.db")

# Create a cursor object
cursor = conn.cursor()

# SQL query to fetch all data from the Clients table
query = """
SELECT DISTINCT COUNT(ci.Client_ID) CLIENT_ID_COUNT,
```

```
        ci.County,
        cd.Delivery_Status,
        cd.Carrier,
        cr.Star_Rating,
        pc.Product_Name,
        pc.Format_Category

FROM Clients ci
LEFT JOIN Sales ss on ss.Client_ID = ci.Client_ID
LEFT JOIN Client_Deliveries cd ON ci.Client_ID = cd.Client_ID
LEFT JOIN Client_Reviews cr ON ci.Client_ID = cr.Client_ID
LEFT JOIN Financial_Transactions ft ON ft.Transaction_ID =
ss.Transaction_ID
LEFT JOIN Products pc ON ss.Product_ID = pc.Product_ID
where cd.Delivery_Status IN ('Delivered','Delayed') and cr.Star_Rating
!= 0
GROUP BY
ci.County,cd.Delivery_Status,cd.Carrier,cr.Star_Rating,pc.Product_Name,
pc.Format_Category
"""

# Execute the query and fetch the results into a Pandas DataFrame
df_clients = pd.read_sql_query(query, conn)

# Close the connection
conn.close()

# Display the first few rows of the DataFrame
df_clients.head(5)

# Save as CSV
df_clients.to_csv("kpi1.csv", index=False)

# For Google Colab users: Download the dataset (optional)
from google.colab import files
files.download("kpi1.csv")
```

APPENDIX N: SQL Querying for 2nd KPI (Delivery Performance)

```
# Connect to the SQLite database
conn = sqlite3.connect("business_data.db")

# Create a cursor object
```

```
cursor = conn.cursor()

# SQL query to fetch all data from the Clients table
query = """
SELECT
   ci.Client_ID,
   ss.Transaction_ID,
   ci.County,
   cd.Carrier,
   cd.Estimated_Delivery_Date,
   cd.Actual_Delivery_Date,
   julianday(Estimated_Delivery_Date) - julianday(Actual_Delivery_Date)
DELAY_DAYS,
   cd.Delivery_Status,
   ft.Is_Refunded

FROM Clients ci
LEFT JOIN Sales ss ON ss.Client_ID = ci.Client_ID
LEFT JOIN Client_Deliveries cd ON cd.Transaction_ID = ss.Transaction_ID
LEFT JOIN Client_Reviews cr ON cr.Client_ID = ci.Client_ID AND
cr.Product_ID = ss.Product_ID
LEFT JOIN Financial_Transactions ft ON ft.Transaction_ID =
ss.Transaction_ID
where cd.Delivery_Status IN ('Delivered','Delayed')
"""

# Execute the query and fetch the results into a Pandas DataFrame
df_deliver = pd.read_sql_query(query, conn)

# Close the connection
conn.close()

# Display the first few rows of the DataFrame
df_deliver.head(5)

# Save as CSV
df_deliver.to_csv("kpi2.csv", index=False)

# For Google Colab users: Download the dataset (optional)
from google.colab import files
files.download("kpi2.csv")
```

APPENDIX O: SQL Querying for 3rd KPI (Financial Analysis)

```
# Connect to the SQLite database
conn = sqlite3.connect("business_data.db")
cursor = conn.cursor()

# Improved SQL Query
query = """
-- Calculate refund amounts and total revenue for each product category
--  Refund_By_Product - Calculates total refund amount per product
category and name

WITH Refund_By_Product AS (
   SELECT
       pc.Product_Name,
       pc.Format_Category,
       SUM(ft.Refund_Amount) AS Total_Refund_Amount
   FROM Sales ss
   LEFT JOIN Products pc ON ss.Product_ID = pc.Product_ID
   LEFT JOIN Financial_Transactions ft ON ss.Transaction_ID =
ft.Transaction_ID
   WHERE ft.Refund_Amount IS NOT NULL AND ft.Is_Refunded = 'Yes'
   GROUP BY pc.Product_Name, pc.Format_Category
),

-- Revenue_By_Product - Calculates total revenue per product category
and name
Revenue_By_Product AS (
   SELECT
       pc.Product_Name,
       pc.Format_Category,
       SUM(ft.Payment_Amount) AS Total_Revenue
   FROM Sales ss
   LEFT JOIN Products pc ON ss.Product_ID = pc.Product_ID
   LEFT JOIN Financial_Transactions ft ON ss.Transaction_ID =
ft.Transaction_ID
   WHERE ft.Payment_Amount IS NOT NULL
   GROUP BY pc.Product_Name, pc.Format_Category
)

-- Combine the results using both Product_Name and Format_Category for
the join
SELECT
```

```
    rbp.Product_Name,
    rbp.Format_Category,  -- Use rbp.Format_Category to avoid potential
mismatch
    rbp.Total_Refund_Amount,
    rb.Total_Revenue,
    ROUND((rbp.Total_Refund_Amount * 100.0 / rb.Total_Revenue), 2) AS
Refund_Percentage
FROM Refund_By_Product rbp
LEFT JOIN Revenue_By_Product rb
    ON rbp.Product_Name = rb.Product_Name AND rbp.Format_Category =
rb.Format_Category;

"""

# Execute the query and store the result in a DataFrame
df_finance = pd.read_sql_query(query, conn)

# Close the database connection
conn.close()

# Save the result to a CSV file
df_finance.to_csv("kpi3.csv", index=False)

# Display the first few rows of the DataFrame
df_finance.head(5)

# For Google Colab users: Download the dataset (optional)
from google.colab import files
files.download("kpi3.csv")
```

APPENDIX P: SQL Querying for 4th KPI (Product Damage Analysis)

```
# Connect to the SQLite database
conn = sqlite3.connect("business_data.db")

# SQL query for Delivery Status Calculation (KPI 4.1)
query_delivery_status = """
SELECT
    cd.Carrier,
    cd.Delivery_Status,
    COUNT(*) AS Status_Count,
```

```
   (SELECT COUNT(*) FROM Client_Deliveries WHERE Carrier = cd.Carrier)
AS Total_Deliveries,
   ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM Client_Deliveries
WHERE Carrier = cd.Carrier), 2) AS Status_Percentage
FROM Client_Deliveries cd
WHERE cd.Carrier IS NOT NULL AND cd.Carrier <> ''
GROUP BY cd.Carrier, cd.Delivery_Status
"""

# SQL query for Refund & Damage Type Calculation (KPI 4.2)
query_refund_damage = """
SELECT
   pd.Damage_Type AS Damage_Type,
   'Refund Related' AS Category,
   COUNT(*) AS Status_Count
FROM Product_Damage pd
WHERE pd.Damage_Type IS NOT NULL
GROUP BY pd.Damage_Type
"""

# Fetch data into DataFrames
df_delivery_status = pd.read_sql_query(query_delivery_status, conn)
df_refund_damage = pd.read_sql_query(query_refund_damage, conn)

# Close the connection
conn.close()

# Save results to CSV files
df_delivery_status.to_csv("Delivery_Status.csv", index=False)
df_refund_damage.to_csv("Refund_Damage.csv", index=False)

# Display first few rows of both DataFrames for verification
print("Delivery Status Data:")
print(df_delivery_status.head())
print("\nRefund & Damage Data:")
print(df_refund_damage.head())

# For Google Colab users: Download the files (optional)
from google.colab import files
files.download("Delivery_Status.csv")
files.download("Refund_Damage.csv")
```

APPENDIX Q: List of Figures