



Higher Diploma in Science in Data Analytics

ASSESSMENT

Module Code: **B8IT101**

Module Description: **Databases and Business Applications**

Student Name: **Javier Ignacio Astorga Di Pauli**

Student ID: **20023978**

ASSESSMENT

Question 1 - SQL Queries

1. Counting the Number of Tables in the Database

The purpose of the following SQL query is to ascertain the total number of tables present within the database. This is achieved by utilising the 'sqlite_master' table, which is a meta-table containing the schema definitions for the database in SQLite. It holds information about all database objects such as tables, indexes, views, and triggers. By executing a 'COUNT' on this meta-table, we can retrieve the number of table objects. The 'WHERE' clause is specifically targeting rows where the 'type' column equals 'table', ensuring that only the count of tables is returned.

```
1 -- 1. How many tables does this database contain?
2 SELECT COUNT(*) 'Number of Tables'
3 FROM sqlite_master
4 WHERE type='table'
```

```
SELECT COUNT(*) 'Number of Tables'
FROM sqlite_master
WHERE type='table'
```

Output:

	Number of Tables
1	13

The output of the query is 13. This number includes two additional tables that SQLite automatically adds to each project. These are system tables used by SQLite for internal purposes.

To get the count of only the user-defined tables in the dataset, which is 11 in this case, the query should exclude these system tables. This can be achieved by modifying the WHERE clause to:

```
WHERE type='table' AND name NOT LIKE 'sqlite_%'
```

This modification ensures that the system tables, which have names starting with 'sqlite_', are not counted in the output.

2. Invoice Quantities within the Database

The purpose of the following SQL query is to ascertain the total number of invoices recorded in the database. This is achieved by employing the COUNT function, which is designed to tally the entries within the InvoiceId column of the invoices table. The query is intentionally straightforward to ensure clarity and precision in the results obtained.

```
7  -- 2. How many invoices are listed in this database?
8  SELECT count(invoiceid) 'Number Of Invoices'
9  FROM invoices
```

```
SELECT count(invoiceid) 'Number Of Invoices'
FROM invoices
```

Output:

	Number Of Invoices
1	412

3. Customers named 'John'

The query is designed to determine the existence of any customers named 'John' within the database. It aims to count the number of entries where 'John' appears in the FirstName field of the customers table. The SQL script utilises the LIKE operator with wildcard characters '%' before and after 'John' to ensure a comprehensive search. This approach captures any entries where 'John' may be part of a longer name or includes additional characters, thus accounting for instances where 'John' could be a middle name or have prefixes/suffixes.

```
11 -- 3. Is there any customer with the name 'John'?
12 SELECT count(FirstName) "John's"
13 FROM customers
14 where customers.FirstName like '%John%'
```

```
SELECT count(FirstName) "John's"
FROM customers
where customers.FirstName like '%John%'
```

Output:

	John's
1	1

4. Longest album in the Database

The following SQL query is designed to identify the longest album within the database. It employs the MAX() function to retrieve the greatest value from the 'Milliseconds' column, which represents the duration of each track in milliseconds. By using a JOIN operation, the query combines data from the 'tracks' table with the corresponding 'album' table to ensure that the output includes the album titles alongside their maximum track length.

```

16 -- 4. What is the longest album in the database?
17 SELECT albums.Title 'Album Title',max(tracks.Milliseconds) 'Lenght In Miliseconds'
18 FROM albums
19 JOIN tracks on tracks.AlbumId = albums.AlbumId

```

```

SELECT albums.Title 'Album Title',max(tracks.Milliseconds) 'Lenght In
Milliseconds'
FROM albums
JOIN tracks on tracks.AlbumId = albums.AlbumId

```

Output:

	Album Title	Lenght In Miliseconds
1	Battlestar Galactica, Season 3	5286953

5. Average length of a track

The purpose of the following SQL query is to calculate the average length of a track within the database. To achieve this, the AVG function is employed to compute the mean value of the 'Milliseconds' column from the tracks table. The result is then rounded to two decimal places using the ROUND function, which enhances readability by simplifying the output to just two decimal points.

```

21 -- 5. What is the average length of a track in the database?
22 SELECT round(avg(tracks.Milliseconds),2) as 'Average Track Length'
23 FROM tracks

```

```
SELECT round(avg(tracks.Milliseconds),2) as 'Average Track Length'
FROM tracks
```

Output:

	Average Track Length
1	393599.21

6. Artist with more albums

This query is designed to identify the artist with the most albums in the database. By employing the GROUP BY clause, we group the records by artist, which allows us to count the number of albums associated with each artist. The results are then ordered in descending order to bring the artist with the highest album count to the top. To ensure that we only get the single top result, the LIMIT function is used, restricting the output to just one row, which represents the artist with the most albums.

```
25 -- 6. What is the artist with more albums?
26 SELECT artists.ArtistId, artists.Name 'Artist Name', count(albums.albumId) 'Number of Albums'
27 FROM artists
28 JOIN albums on albums.ArtistId = artists.ArtistId
29 GROUP by artists.Name
30 ORDER by count(albums.albumId) DESC
31 LIMIT 1
```

```
SELECT artists.ArtistId, artists.Name 'Artist Name',
count(albums.albumId) 'Number of Albums'
FROM artists
JOIN albums on albums.ArtistId = artists.ArtistId
GROUP by artists.Name
ORDER by count(albums.albumId) DESC
LIMIT 1
```

Output:

	ArtistId	Artist Name	Number of Albums
1	90	Iron Maiden	21

7. Top 5 Artists with the Highest Sales

This query is designed to identify the top five artists with the highest sales. To achieve this, we aggregate the total sales by summing the price of each song from the 'invoice_items' table. Each track is then linked to its respective album using the 'AlbumId' from the 'tracks' table, and subsequently, each album is associated with its artist using the 'ArtistId' from the 'albums' table. This relational structure allows us to accurately calculate the total sales for each artist. The final output is a list of artists and their corresponding sales totals, ordered from highest to lowest, limited to the top five.

```
33 -- 7. List the top 5 artists with more sales.
34 SELECT ar.Name, round(sum(ii.UnitPrice),2) 'Total of Sales'
35 FROM invoice_items ii
36 JOIN tracks t on t.TrackId = ii.TrackId
37 JOIN albums al on al.AlbumId = t.AlbumId
38 JOIN artists ar on ar.ArtistId = al.ArtistId
39 GROUP by ar.Name
40 ORDER by sum(ii.UnitPrice) DESC
41 LIMIT 5
```

```
SELECT ar.Name, round(sum(ii.UnitPrice),2) 'Total of Sales'
FROM invoice_items ii
JOIN tracks t on t.TrackId = ii.TrackId
JOIN albums al on al.AlbumId = t.AlbumId
JOIN artists ar on ar.ArtistId = al.ArtistId
GROUP by ar.Name
ORDER by sum(ii.UnitPrice) DESC
LIMIT 5
```

Output:

	Name	Total of Sales
1	Iron Maiden	138.6
2	U2	105.93
3	Metallica	90.09
4	Led Zeppelin	86.13
5	Lost	81.59

8. Query to List Tracks Containing 'you' in Their Title

The objective of this SQL query is to retrieve a list of songs that include the word 'you' within their titles. To accomplish this, the query extracts the track names from the 'tracks' table, album titles from the 'albums' table, and artist names from the 'artists' table. A WHERE clause is employed to filter the results to only those tracks where 'you' appears in the song title. Additionally, table aliases 't', 'al', and 'ar' are used to abbreviate the names of the tables, making the code more concise and readable.

```

44 -- 8. List the tracks that contain 'you' in their title. List the album and artist of these
45 -- tracks.
46 SELECT t.Name SongName, al.Title Album, ar.name Artist
47 FROM tracks t
48 JOIN albums al on al.AlbumId = t.AlbumId
49 JOIN artists ar on ar.ArtistId = al.ArtistId
50 WHERE t.name like '%you%'
51

```

```

SELECT t.Name SongName, al.Title Album, ar.name Artist
FROM tracks t
JOIN albums al on al.AlbumId = t.AlbumId
JOIN artists ar on ar.ArtistId = al.ArtistId
WHERE t.name like '%you%'

```

Output:

	SongName	Album	Artist
1	For Those About To Rock (We ...	For Those About To Rock We ...	AC/DC
2	Put The Finger On You	For Those About To Rock We ...	AC/DC
3	You Oughta Know	Jagged Little Pill	Alanis Morissette
4	Right Through You	Jagged Little Pill	Alanis Morissette
5	You Learn	Jagged Little Pill	Alanis Morissette
6	You Oughta Know (Alternate)	Jagged Little Pill	Alanis Morissette
7	We Die Young	Facelift	Alice In Chains
8	Put You Down	Facelift	Alice In Chains
9	I Know Something (Bout You)	Facelift	Alice In Chains

9. Identifying Invoices with Above-Average Track Counts

The following SQL query is designed to determine the average number of tracks per invoice and subsequently identify all invoices containing a higher than average track count. The process begins by calculating the average track count per invoice. This is achieved through a subquery that counts the tracks associated with each invoice and then computes the average of these counts. Utilising the HAVING clause allows for the filtering of groups created by the GROUP BY clause based on the conditions set by the subquery. In essence, the query first establishes a benchmark average and then, through a

comparison operation, retrieves only those invoices where the track count exceeds this average.

```
52 -- 9. What is the average number of items per invoice? List all invoices that have more
53 -- than the average number of items in them.
54
55 -- SELECT avg(ItemCount)
56 -- FROM (
57 --     SELECT count(TrackId) ItemCount
58 --     FROM invoice_items
59 --     GROUP by InvoiceId)
60 -----
61 SELECT InvoiceId, count(TrackId)
62 FROM invoice_items
63 GROUP by InvoiceId
64 HAVING count(TrackId) > (
65     SELECT avg(ItemCount)
66     FROM (
67         SELECT count(TrackId) ItemCount
68         FROM invoice_items
69         GROUP by InvoiceId)
70 )
```

```
SELECT InvoiceId, count(TrackId)
FROM invoice_items
GROUP by InvoiceId
HAVING count(TrackId) > (
    SELECT avg(ItemCount)
    FROM (
        SELECT count(TrackId) ItemCount
        FROM invoice_items
        GROUP by InvoiceId)
```

Output:

Within the main SQL script, there is a subquery that, although commented out, serves a crucial role in the overall execution. This subquery is not executed independently as it is embedded within the main code. Its purpose is to calculate the average number of tracks per invoice. For clarity and understanding, if this subquery were to be run separately, it would yield the average track count, which is a pivotal figure in the context of this analysis.

To illustrate, executing the subquery alone would result in the following average number of tracks per invoice:

	avg(ItemCount)
1	5.4368932038835

Upon running this subquery, we find that the average number of tracks per invoice is 5,43. This figure is then used as a benchmark to filter the main query results.

When the principal code is executed, it provides a list of invoices where the number of tracks exceeds this average. The result set not only confirms the invoices that have a higher than average track count but also underscores the efficiency of embedding the subquery within the main query to streamline the data retrieval process.

	InvoiceId	count(TrackId)
1	3	6
2	4	9
3	5	14
4	10	6
5	11	9
6	12	14
7	17	6
8	18	9
9	19	14

10. Annual Invoices

In the analysis of the invoices dataset, the strftime function was employed to format the InvoiceDate column. This function is particularly useful for manipulating specific components of date or time values within a column. In this instance, the formatting was applied to extract the year from the InvoiceDate, although it is versatile enough to format dates by year, month, day, etc. Utilising this function facilitates the grouping of invoice records by year, enabling the calculation of the total number of invoices and the aggregate sales amount for each year. The resulting data is then ordered chronologically by year to observe trends over time.

```
72 -- 10. How many invoices are there per year? What is the total amount in sales per
73 -- year?
74
75 SELECT strftime('%Y', InvoiceDate) Year, count(invoiceid) 'Amount of Invoices', sum(Total) 'Total of Sales'
76 FROM invoices
77 GROUP by year
78 order by year
```

```

SELECT strftime('%Y', InvoiceDate) Year, count(invoiceid) 'Amount of
Invoices', sum(Total) 'Total of Sales'
FROM invoices
GROUP by year
order by year

```

Output:

	Year	Amount of Invoices	Total of Sales
1	2009	83	449.46
2	2010	83	481.45
3	2011	83	469.58
4	2012	83	477.53
5	2013	80	450.58

Question 2 - Database Design & Implementation

Part 1

Introduction to the Gaming and Hobby Store

The Gaming and Hobby Store is a retail business focused on providing a diverse range of gaming and hobby products. Its inventory includes Board Games, Card Games, Role-Playing Games, Strategy Games, Miniatures Games, Model Kits, Collectibles, and Craft Supplies. The store is staffed by five employees, each capable of managing multiple transactions, with each transaction being managed by one employee.

With a network of 50 suppliers, the store ensures that each product is uniquely supplied by one, while a supplier can provide a variety of products. The store's extensive product range includes over 400 different items, catering to a wide array of customer interests and hobbies.

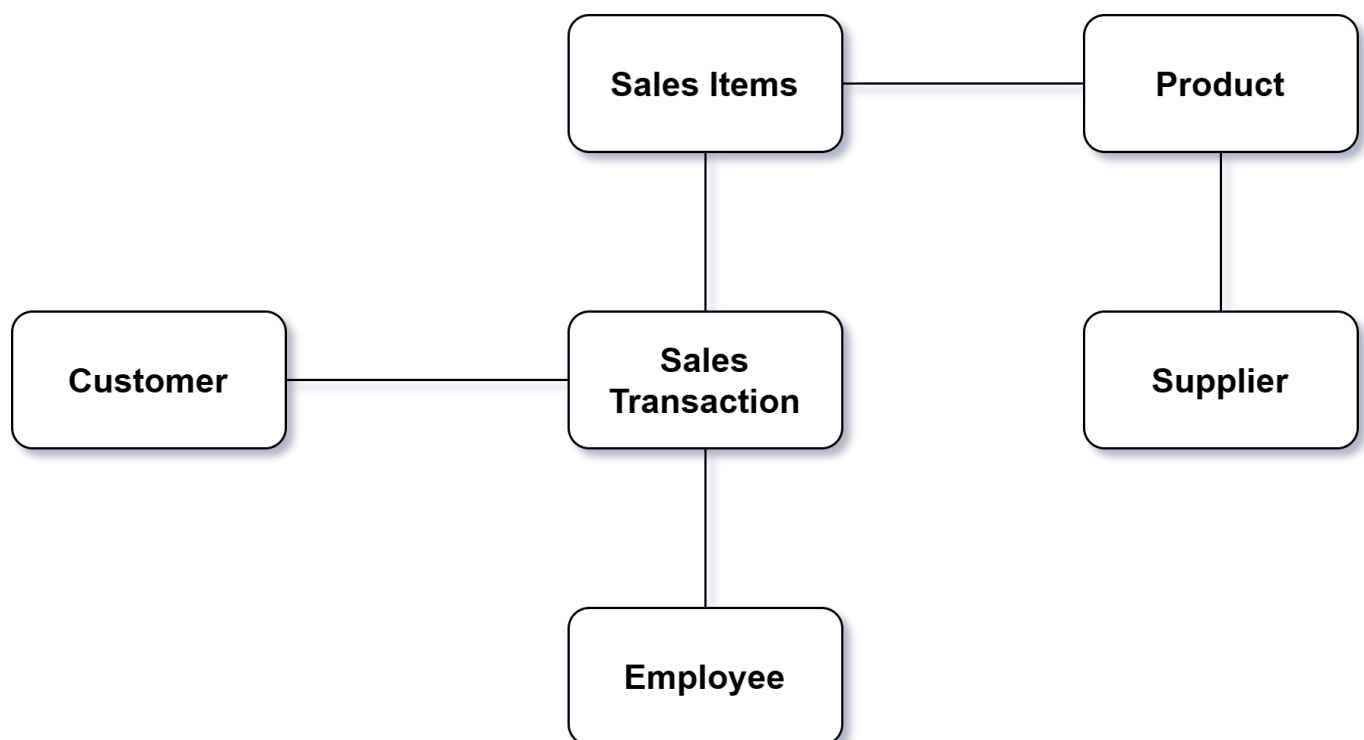
The store maintains a detailed database of sales transactions, where each transaction can include multiple sales items, but each item is part of one specific transaction. This system also records that each sales transaction is associated with one customer, although a customer can engage in multiple transactions over time.

In its operations, the store ensures that each sales item is linked to a specific product, while a product can be part of several sales items. This approach aids in efficient inventory management and understanding customer preferences.

A key operational feature is the separate recording of billing and shipping addresses, acknowledging that these are often different and both crucial for the business's smooth functioning. Each sale involves an employee, emphasizing the store's commitment to personalized customer service.

Conceptual Database Design for a Gaming and Hobby Store

In designing a database for a Gaming and Hobby Store, which includes a range of products like card games, board games, hobby supplies, and collectibles, it's crucial to identify the key entities and their relationships. This will ensure efficient data management and retrieval. Below is a detailed breakdown of the entities, their attributes, and the relationships between them.



- **Product:** Connected to *Supplier* (indicating where the product comes from) and *Sales Items* (indicating that products are part of sales transactions).
- **Supplier:** Connected to *Product*.
- **Customer:** Connected to *Sales Transaction* (indicating that customers make transactions).
- **Sales Transaction:** Connected to *Customer* and *Sales Items*.
- **Sales Items:** Connected to *Sales Transaction* and *Product*.
- **Employee:** Connected to *Sales Transaction* (if employees are involved in transactions).

Entities and Their Attributes:

1. Product:

- Attributes:
 - **product_id** (Primary Key)
 - **product_name**
 - **genre**
 - **price**
 - **stock_quantity**
 - **type**
 - **supplier_id** (Foreign Key)

2. Supplier:

- Attributes:
 - **supplier_id** (Primary Key)
 - **supplier_name**
 - **contact**
- Note: It might be possible to add more information about the address, city, country, and postal code to complete the supplier's information. However, for simplicity in the purpose of this work, it will not be done.

3. Customer:

- Attributes:
 - **customer_id** (Primary Key)
 - **customer_name**
 - **contact**
 - **address**
 - **city**
 - **country**
 - **postal_code**
- Note: Splitting **contact** into **email** and **phone_number** could provide more detailed contact information.

4. Sales Transaction:

- Attributes:
 - **invoice_id** (Primary Key)
 - **date**
 - **customer_id** (Foreign Key)
 - **billing_address**
 - **billing_city**
 - **billing_state**
 - **billing_country**
 - **billing_postal_code**
 - **total**
 - **employee_id** (Foreign Key)
- Note: Ensure **total** reflects the total amount of the transaction.

5. Sales Items:

- Attributes:
 - **invoice_line_id** (Primary Key)
 - **invoice_id** (Foreign Key)
 - **product_id** (Foreign Key)
 - **quantity**
- Note: This table represents the line items in a sales transaction.

6. Employee:

- Attributes:
 - **employee_id** (Primary Key)

- **employee_name**
- **role**
- **contact_information**
- Note: **contact_information** could be expanded to include **email**, **phone_number**, and **address**.

Relationships and Cardinalities:

The relationships between entities along with their cardinalities are described as follows:

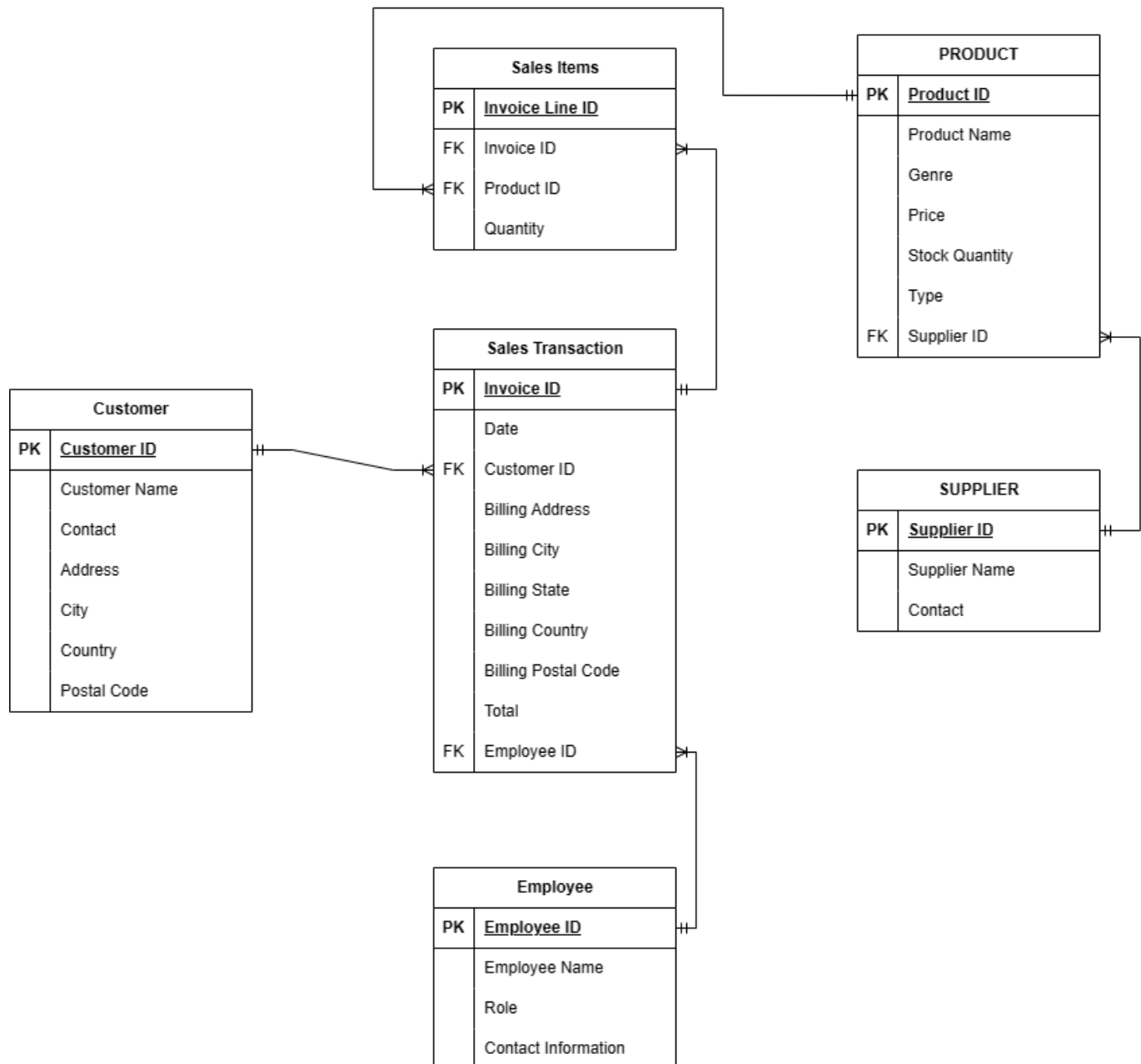
- **Product-Supplier Relationship:** Many-to-one; each product is supplied by one supplier, but a supplier can provide many products.
- **Sales Transaction-Customer Relationship:** Many-to-one; each sales transaction is associated with one customer, but a customer can have many transactions.
- **Sales Transaction-Sales Items Relationship:** One-to-many; each sales transaction can include multiple sales items, but each sales item is part of one transaction.
- **Sales Items-Product Relationship:** Many-to-one; each sales item corresponds to one product, but a product can be part of many sales items.
- **Employee-Sales Transaction Relationship:** One-to-many; each employee can handle multiple sales transactions, but each transaction is managed by one employee.

Assumptions and Business Rules:

- Customers can have multiple transactions, suggesting a one-to-many relationship from customers to sales transactions.
- A product can be involved in multiple sales transactions, and a sales transaction can include multiple products.
- Employees are responsible for handling multiple sales transactions, but each transaction is handled by a single employee.\

Normalization to 3rd Normal Form:

- The design ensures that all attributes are dependent only on the primary key, thereby eliminating transitive dependencies.
- Non-key attributes are not dependent on other non-key attributes, ensuring data integrity and avoiding redundancy.



Part 2:

Physical Design and Data Population of the Gaming and Hobby Store Database

In the development of the Gaming and Hobby Store database, the Data Definition Language (DDL) and Data Manipulation Language (DML) were employed to create and populate the database using SQLite. The process involved the following steps:

1. Database Creation with DDL:

- The database was initiated in SQLite, a lightweight, file-based system. The structure of the database, including tables and columns, was defined using SQL commands.

2. Table Creation with DDL:

○ Product Table:

- This table was structured to include attributes such as product ID, name, genre, price, stock quantity, supplier ID, and type. The

data incorporated movie names as product names, realistic genres, and types of products.

Field Name	Type	Options
product_id	Row Number	blank: 0 % Σ X
product_name	Movie Title	blank: 0 % Σ X
genre	Custom List	Family, Party, Educational, Cooperative, Competitive, Fantasy, Sci-Fi, Historical, Warfare random blank: 0 % Σ X
price	Number	min: 1 max: 200 decimals: 1 blank: 0 % Σ X
stock_quantity	Number	min: 0 max: 100 decimals: 0 blank: 0 % Σ X
supplier_id	Number	min: 1 max: 50 decimals: 0 blank: 0 % Σ X
type	Custom List	Board Game, Card Game, Role-Playing Game, Strategy Game, Miniatures Game, Model Kit, Collectible, C random blank: 0 % Σ X

+ ADD ANOTHER FIELD GENERATE FIELDS USING AI...













Rows: 400 Format: SQL Table Name: product ☒ include CREATE TABLE



```
CREATE TABLE "product" (
    "product_id" INT UNIQUE,
    "product_name" VARCHAR(50),
    "genre" VARCHAR(11),
    "price" DECIMAL(4, 1),
    "stock_quantity" INT,
    "supplier_id" INT,
    "type" VARCHAR(17),
    PRIMARY KEY("product_id")
);


insert into product (product_id, product_name, genre, price,
stock_quantity, supplier_id, type) values (1, 'Taking Sides', 'Family', 64.4,
21, 10, 'Model Kit');

insert into product (product_id, product_name, genre, price,
stock_quantity, supplier_id, type) values (2, 'Journey for Margaret',
'Fantasy', 31.5, 48, 12, 'Card Game');
```

- **Supplier Table:**
 - The Supplier table was created with essential attributes like supplier ID, name, and contact email.

Field Name	Type	Options
 supplier_id	Row Number 	blank: 0 %  
 supplier_name	Company Name 	blank: 0 %  
 contact	Email Address 	blank: 0 %  

 ADD ANOTHER FIELD
  GENERATE FIELDS USING AI...

Rows: 50
 Format: SQL 
 Table Name: supplier
 ☒ include CREATE TABLE

```
CREATE TABLE "supplier" (
    "supplier_id" INT UNIQUE,
    "supplier_name" VARCHAR(50),
    "contact" VARCHAR(50),
    PRIMARY KEY("supplier_id")
);

insert into supplier (supplier_id, supplier_name, contact) values (1,
'Gigaclub', 'ckalinsky0@umich.edu');

insert into supplier (supplier_id, supplier_name, contact) values (2,
'Bluezoom', 'kbeat1@cnet.com');
```

- **Sales Items Table:**
 - Comprising item ID, invoice ID (FK), product ID (FK), and quantity, the Sales Items table was designed to maintain consistency with the original data in the Product and Sales Transaction tables.

Field Name	Type	Options
invoice_line_id	Row Number	blank: 0 % Σ X
invoice_id	Number	min: 1 max: 300 decimals: 0 blank: 0 % Σ X
product_id	Number	min: 1 max: 400 decimals: 0 blank: 0 % Σ X
quantity	Number	min: 1 max: 5 decimals: 0 blank: 0 % Σ X

+ ADD ANOTHER FIELD
 [GENERATE FIELDS USING AI...](#)

Rows: 700
 Format: SQL
 Table Name: sales_items
 ☒ include CREATE TABLE

```

CREATE TABLE "sales_items" (
    "invoice_line_id" INT UNIQUE,
    "invoice_id" INT,
    "product_id" INT,
    "quantity" INT,
    FOREIGN KEY("invoice_id") REFERENCES
    "sales_transaction"("invoice_id"),
    FOREIGN KEY("product_id") REFERENCES "product"("product_id"),
    PRIMARY KEY("invoice_line_id")
);

insert into sales_items (invoice_line_id, invoice_id, product_id, quantity)
values (1, 128, 281, 2);

insert into sales_items (invoice_line_id, invoice_id, product_id, quantity)
values (2, 185, 56, 3);

```

- **Sales Transaction Table:**
 - This table captures the details of sales transactions within the specified date range (01/01/2023 - 31/12/2023) and includes location information.

Field Name	Type	Options
invoice_id	Row Number	blank: 0% Σ X
date	Datetime	01/01/2023 to 12/31/2023 format: dd/mm/yyyy blank: 0% Σ X
customer_id	Number	min: 1 max: 200 decimals: 0 blank: 0% Σ X
billing_address	Street Address	blank: 0% Σ X
billing_city	City	blank: 0% Σ X
billing_state	State	restrict states... All Countries blank: 0% Σ X
billing_country	Country	restrict countries... blank: 0% Σ X
billing_postal_code	Postal Code	blank: 0% Σ X
total	Money	between 10 and 300 in € blank: 0% Σ X
employee_id	Number	min: 1 max: 5 decimals: 0 blank: 0% Σ X

+ ADD ANOTHER FIELD GENERATE FIELDS USING AI...

Rows: 300 Format: SQL Table Name: sales_transaction ☒ include CREATE TABLE

```

CREATE TABLE "sales_transaction" (
  "invoice_id" INT UNIQUE,
  "date" DATE,
  "customer_id" INT,
  "billing_address" VARCHAR(50),
  "billing_city" VARCHAR(50),
  "billing_state" VARCHAR(50),
  "billing_country" VARCHAR(50),
  "billing_postal_code" INTEGER,
  "total" NUMERIC,
  "employee_id" INT,
  FOREIGN KEY("customer_id") REFERENCES
"customer"("customer_id"),
  FOREIGN KEY("employee_id") REFERENCES
"employee"("employee_id"),
  PRIMARY KEY("invoice_id")
);

insert into sales_transaction (invoice_id, date, customer_id,
billing_address, billing_city, billing_state, billing_country,
billing_postal_code, total, employee_id) values (1, '25/09/2023', 53, '4440

```

Paget Parkway', 'Boston', 'Massachusetts', 'United States', '02283',
'€57,10', 1);

insert into sales_transaction (invoice_id, date, customer_id,
billing_address, billing_city, billing_state, billing_country,
billing_postal_code, total, employee_id) values (2, '02/03/2023', 66, '2510
Westerfield Center', 'Daliuhao', null, 'China', null, '€228,30', 2);

- **Customer Table:**

- The Customer table was structured with all necessary attributes as per the project requirements.

Field Name	Type	Options
customer_id	Row Number	blank: 0% Σ X
customer_name	Full Name	blank: 0% Σ X
contact	Email Address	blank: 0% Σ X
address	Street Address	blank: 0% Σ X
city	City	blank: 0% Σ X
country	Country	restrict countries... blank: 0% Σ X
postal_code	Postal Code	blank: 0% Σ X

+ ADD ANOTHER FIELD GENERATE FIELDS USING AI...

Rows: 200 Format: SQL Table Name: customer ☒ include CREATE TABLE

```
CREATE TABLE "customer" (
    "customer_id" INT UNIQUE,
    "customer_name" VARCHAR(50),
    "contact" VARCHAR(50),
    "address" VARCHAR(50),
    "city" VARCHAR(50),
    "country" VARCHAR(50),
    "postal_code" VARCHAR(50),
    PRIMARY KEY("customer_id")
);
```

insert into customer (customer_id, customer_name, contact, address,
city, country, postal_code) values (1, 'Mel Bocke', 'mbocke0@rambler.ru',
'20687 Mayfield Point', 'Sasaguri', 'Japan', '811-2405');

insert into customer (customer_id, customer_name, contact, address, city, country, postal_code) values (2, 'Moise Lune', 'mlune1@yelp.com', '6326 Daystar Plaza', 'Nam Đàn', 'Vietnam', null);

- **Employee Table:**

- For the Employee table, roles were defined using a custom list that reflects realistic job roles in a retail environment.

Field Name	Type	Options
employee_id	Row Number	blank: 0 %
employee_name	Full Name	blank: 0 %
role	Custom List	Store Manager, Assistant Manager, Sales Associate, Inventory Specialist, Cashier
contact_information	Email Address	blank: 0 %

+ ADD ANOTHER FIELD GENERATE FIELDS USING AI...

Rows: 5 Format: SQL Table Name: employee ☒ Include CREATE TABLE

```
CREATE TABLE "employee" (
    "employee_id"      INT UNIQUE,
    "employee_name"    VARCHAR(50),
    "role"             VARCHAR(21),
    "contact_information" VARCHAR(50),
    PRIMARY KEY("employee_id")
);

insert into employee (employee_id, employee_name, role,
contact_information) values (1, 'Teodoor McCurt', 'Inventory Speacialist',
'tmccurt0@walmart.com');

insert into employee (employee_id, employee_name, role,
contact_information) values (2, 'Arabella Blewis', 'Assistant Manager',
'ablewis1@mediafire.com');
```

3. Data Population with DML:

- The tables were populated using DML insert statements. Data generators like Mockaroo were utilized to create realistic and coherent data, ensuring a sample one-year transaction data (01/01/2023 - 31/12/2023) for each table. Below the images, a selection of these insert statements will be included to illustrate the process.

4. Ensuring Data Consistency:

- Care was taken to ensure that the quantity of data in tables related by foreign keys matched the data in their corresponding original tables. For instance, the number of **supplier_id** entries in the Product table corresponds to the number of entries in the Supplier table.

Part 3:

1. Displaying Details of Products Priced Above 100

For this query, fundamental SQL codes were employed to select and display all details from the 'Product' table for products priced above 100. The results were ordered by price in ascending order for better visualization.

```

1  -- 1. Show all the details of the products that have a price greater than 100.
2  SELECT *
3  FROM Product
4  WHERE price > 100
5  ORDER by price ASC;

```

SELECT *

FROM Product

WHERE price > 100

ORDER by price ASC;

Output:

	product_id	product_name	genre	price	stock_quantity	supplier_id	type
1	113	Devil's Rain, The	Sci-Fi	100.5	72	14	Miniatures Game
2	148	Rich, Young and Pretty	Fantasy	100.7	69	28	Craft Supplies
3	311	Children Shouldn't Play with ...	Historical	100.7	8	32	Miniatures Game
4	74	SLC Punk!	Competitive	101.7	14	28	Card Game
5	253	The Infinite Man	Party	102.1	45	4	Card Game
6	96	Sleepaway Camp	Cooperative	104.2	51	33	Model Kit
7	67	Marrying Man, The (Too Hot to...	Cooperative	104.8	7	20	Board Game
8	291	Jack Goes Boating	Cooperative	105.6	77	18	Miniatures Game
9	214	Sol	Educational	106.1	95	11	Collectible

2. Displaying Products with Supplier Details

For this query, fundamental SQL techniques were employed. Aliases were assigned to tables for ease of reference, and a JOIN operation was used to merge information from the supplier and product tables. This approach allows us to determine which supplier provides each product.

```

8  -- 2. Show all the products along with the supplier detail who supplied the products.
9  SELECT p.product_id, p.product_name, p.genre, p.type, p.price, s.supplier_id, s.supplier_name, s.contact
10 FROM product p
11 JOIN supplier s on s.supplier_id = p.supplier_id
12

```

```

SELECT p.product_id, p.product_name, p.genre, p.type, p.price,
s.supplier_id, s.supplier_name, s.contact

```

```

FROM product p

```

```

JOIN supplier s on s.supplier_id = p.supplier_id

```

Output:

	product_id	product_name	genre	type	price	supplier_name	contact
1	1	Taking Sides	Family	Model Kit	64.4	Flashset	mhucknall9@oakley.com
2	2	Journey for Margaret	Fantasy	Card Game	31.5	Skyvu	lthurbyb@icio.us
3	3	Family Viewing	Sci-Fi	Miniatures Game	58.7	Skinte	rmatzelu@hc360.com
4	4	In the Name of the King: A ...	Educational	Model Kit	99.4	Yabox	asargeant6@imageshack.us
5	5	Heima	Educational	Miniatures Game	152	Agivu	hromanetw@surveymonkey.com
6	6	The Wind in the Willows	Sci-Fi	Collectible	19.2	Jayo	lmarlandy@miibeian.gov.cn
7	7	Hatchet for the Honeymoon ...	Sci-Fi	Miniatures Game	97	Agivu	rcastelluzzid@icio.us
8	8	They Came Back (Les ...	Historical	Collectible	41	Agimba	hplummer1b@trellian.com
9	9	Time to Live, a Time to Die, A	Family	Collectible	155.5	Agimba	hplummer1b@trellian.com

The output of this query displays a comprehensive list of products, including details such as product ID, name, genre, type, and price, along with corresponding supplier information like supplier ID, name, and contact. This comprehensive view is instrumental in understanding the supply chain for each product.

3. Identifying Products Bought by Top 5 Customers

In this query, the objective was to identify all products purchased by the top 5 customers in terms of total spending at the store. To achieve this, a subquery was used to determine who the top 5 customers are. This was accomplished by summing all purchases made by each customer, ordering the results in descending order, and limiting the list to the top 5. Joins were employed to connect the customer ID, customer name, and product name, which are located in different tables. The final output was ordered by customer ID for better visualization.

```

13 -- 3. Show all the products bought by the top 5 customers (the customers that spent the most in the store).
14 SELECT st.customer_id, c.customer_name, p.product_name
15 FROM sales_transaction st
16 JOIN sales_items si on st.invoice_id = si.invoice_id
17 JOIN product p on p.product_id = si.product_id
18 JOIN customer c on st.customer_id = c.customer_id
19 WHERE st.customer_id IN (
20     SELECT st.customer_id
21     FROM sales_transaction st
22     GROUP by customer_id
23     ORDER by sum(total) DESC
24     Limit 5
25 )
26 ORDER by st.customer_id ASC

```

SELECT st.customer_id, c.customer_name, p.product_name

FROM sales_transaction st

JOIN sales_items si on st.invoice_id = si.invoice_id

JOIN product p on p.product_id = si.product_id

JOIN customer c on st.customer_id = c.customer_id

WHERE st.customer_id IN (

SELECT st.customer_id

FROM sales_transaction st

GROUP by customer_id

ORDER by sum(total) DESC

Limit 5

)

ORDER by st.customer_id ASC

Output:

	customer_id	customer_name	product_name
1	193	Chick Turfus	Nina Takes a Lover
2	193	Chick Turfus	Colt Comrades
3	193	Chick Turfus	Cashback
4	193	Chick Turfus	Sidekicks
5	194	Obidiah Leftly	Flower of Evil, The (Fleur du ...
6	194	Obidiah Leftly	Genealogies of a Crime ...
7	194	Obidiah Leftly	Burt's Buzz
8	194	Obidiah Leftly	Hard, Fast and Beautiful
9	194	Obidiah Leftly	Who's That Knocking at My

The output of this query effectively displays the products bought by the store's top 5 customers, organized by customer ID. This approach ensures that the data is presented in a clear and concise manner, making it easy to identify the purchasing patterns of the store's most valuable customers.

4. Top 3 Suppliers with the Most Products

In this query, the goal was to identify the top three suppliers based on the number of products they provide. To achieve this, a SQL query was written that sums up all the products supplied by each supplier. The query also employs a JOIN operation to enhance the visibility of the table, allowing for the display of the supplier's name and contact information alongside the count of products they supply.

```

28 -- 4. Show the information of the top 3 suppliers with more products.
29 SELECT product.supplier_id, supplier.supplier_name, count(product.supplier_id) Product_Count, supplier.contact
30 FROM product
31 JOIN supplier on supplier.supplier_id = product.supplier_id
32 GROUP by product.supplier_id
33 ORDER by Product_Count DESC
34 LIMIT 3
35

```

```

SELECT product.supplier_id, supplier.supplier_name,
count(product.supplier_id) Product_Count, supplier.contact

```

```

FROM product

```

```

JOIN supplier on supplier.supplier_id = product.supplier_id

```

```

GROUP by product.supplier_id

```


ORDER by Product_Count DESC

LIMIT 3

Output:

	supplier_id	supplier_name	Product_Count	contact
1	37	Wikivu	14	mle noire10@list-manage.com
2	22	Divavu	14	elewingtonl@pen.io
3	14	Agivu	13	rcastelluzzid@icio.us

The output of this query displays the supplier ID, supplier name, the total count of products supplied by each supplier (Product_Count), and the supplier's contact information. The results are ordered in descending order of the Product_Count, showing the top three suppliers who have the highest number of products in the store. This output is straightforward and does not require further detailed description.

5. Identifying the Highest Value Product in Sales

To determine the product with the highest sales value, I executed a SQL query that involved multiple steps. The process required joining tables, aggregating data, and sorting the results. Specifically, the query multiplied the price of each product from the 'product' table with the sum of the quantities sold from the 'sales items' table. This calculation was achieved by joining the 'sales items' table with the 'sales transaction' and 'product' tables. The data was then grouped by product ID and ordered in descending order based on the total sales value. The query was limited to return only the top result, ensuring that the product with the highest sales value was identified.

```
36 -- 5. What is the product with the highest value of sales?
37 SELECT p.product_id, p.product_name, p.genre, p.type, p.price * sum(si.quantity) as total_sold
38 FROM sales_items si
39 JOIN sales_transaction st on st.invoice_id = si.invoice_id
40 JOIN product p on p.product_id = si.product_id
41 GROUP by si.product_id
42 ORDER by total_sold DESC
43 LIMIT 1
```

```
SELECT p.product_id, p.product_name, p.genre, p.type, p.price *
sum(si.quantity) as total_sold
```

```
FROM sales_items si
```

```
JOIN sales_transaction st on st.invoice_id = si.invoice_id
```

```
JOIN product p on p.product_id = si.product_id
```

```
GROUP by si.product_id
```

ORDER by total_sold DESC

LIMIT 1

Output:

	product_id	product_name	genre	type	total_sold
1	378	Bloody Mama	Educational	Board Game	2949.0

The output of this query provides the details of the product with the highest sales value, including its ID, name, genre, type, and the total sales amount. This information is crucial for understanding which product is the most successful in terms of sales within the specified time frame.

6. Summing Total Sales by Month

In this code, special attention was needed regarding the formatting of dates and prices in SQLite. For the strftime function to work correctly, dates must be in the YYYY-MM-DD format. Additionally, prices need to be in a numeric format.

Note: In my case, the initial date format was DD/MM/YYYY, which complicated the use of the strftime function. Therefore, I had to change the format of the entire column and all its rows. Similarly, the 'total' column initially imported from Mockaroo was in text format with the Euro symbol, necessitating a change to numeric format for simpler and more efficient SQL execution.

```
45 -- 6. Sum the total sales by month.  
46 SELECT strftime('%m', date) AS Month, SUM(total) AS TotalSales  
47 FROM Sales_Transaction  
48 GROUP BY strftime('%m', date)  
49 ORDER BY strftime('%m', date)
```

```
SELECT strftime('%m', date) AS Month, SUM(total) AS TotalSales  
  
FROM Sales_Transaction  
  
GROUP BY strftime('%m', date)  
  
ORDER BY strftime('%m', date)
```

Output:

	Month	TotalSales
1	01	4656.0
2	02	2440.0
3	03	5739.0
4	04	4118.0
5	05	2034.0
6	06	3843.0
7	07	3014.0
8	08	4249.0
9	09	3330.0

7. Creating a View to Display Customer Purchases in October 2023

In SQL, a "view" is essentially a virtual table based on the result of an SQL query. For this task, a view named 'OctoberPurchases' was created to show the total number of items each customer bought from the business in October 2023, along with the total price. This required joining several tables: 'sales_items' with 'sales_transaction' (to connect 'customer_id' with 'invoice_id'), 'customer' (to identify which customer made each purchase), and 'product' (to determine the price of each product).

```

51 -- 7. Create a view that shows the total number of items a customer bought
52 -- from the business in October 2023 along with the total price.
53 CREATE VIEW OctoberPurchases AS
54 SELECT c.customer_id CustomerID, c.customer_name,
55        count(si.quantity) NumberOfItems, sum(p.price * si.quantity) TotalPrice
56 FROM sales_items si
57 JOIN sales_transaction st on st.invoice_id = si.invoice_id
58 JOIN customer c on c.customer_id = st.customer_id
59 JOIN product p on p.product_id = si.product_id
60 WHERE st.date BETWEEN '2023-10-01' AND '2023-10-31'
61 GROUP by c.customer_id
62

```

CREATE VIEW OctoberPurchases AS

SELECT c.customer_id CustomerID, c.customer_name,
count(si.quantity) NumberOfItems, sum(p.price * si.quantity) TotalPrice

FROM sales_items si

JOIN sales_transaction st on st.invoice_id = si.invoice_id

```

JOIN customer c on c.customer_id = st.customer_id

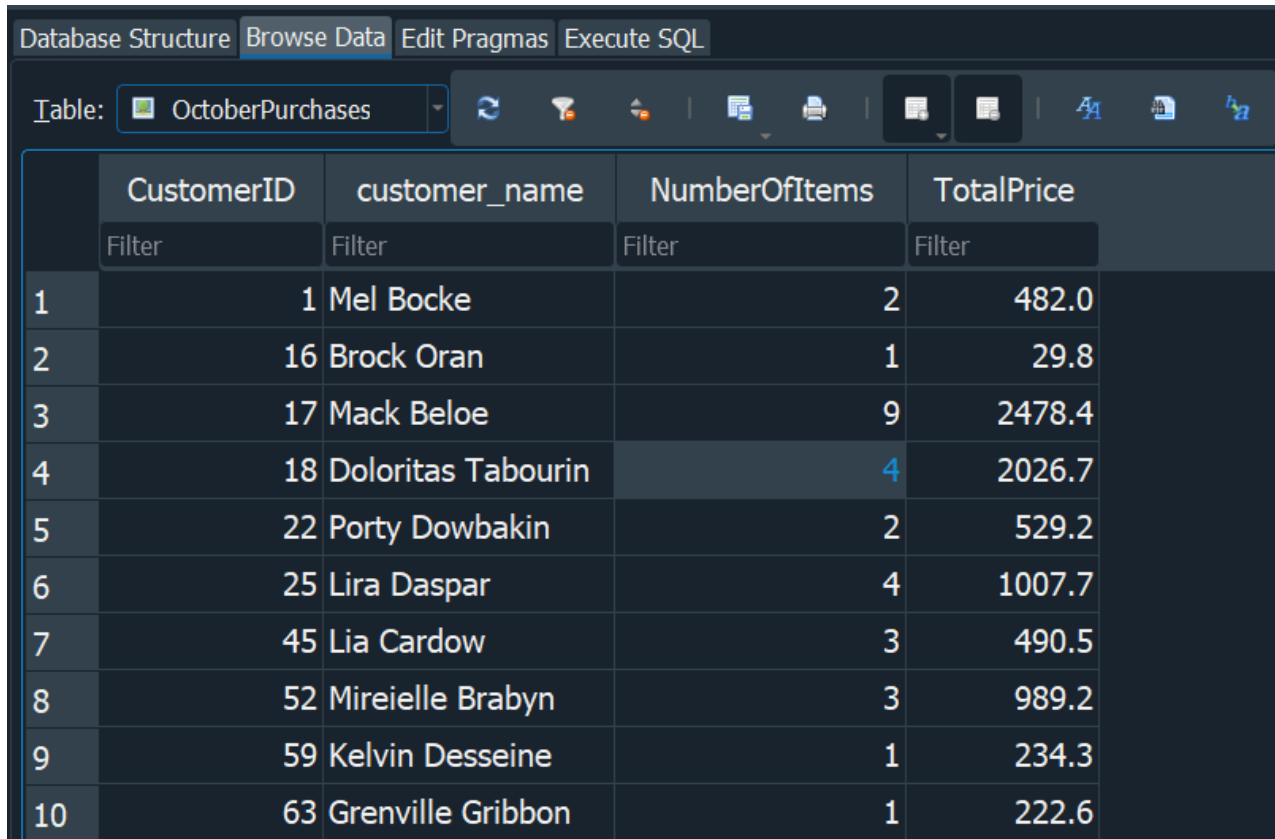
JOIN product p on p.product_id = si.product_id

WHERE st.date BETWEEN '2023-10-01' AND '2023-10-31'

GROUP by c.customer_id

```

Output:



	CustomerID	customer_name	NumberOfItems	TotalPrice
	Filter	Filter	Filter	Filter
1	1	Mel Bocke	2	482.0
2	16	Brock Oran	1	29.8
3	17	Mack Beloe	9	2478.4
4	18	Doloritas Tabourin	4	2026.7
5	22	Porty Dowbakin	2	529.2
6	25	Lira Daspar	4	1007.7
7	45	Lia Cardow	3	490.5
8	52	Mireielle Brabyn	3	989.2
9	59	Kelvin Desseine	1	234.3
10	63	Grenville Gribbon	1	222.6

Regarding the output, it's important to note that creating a view does not return immediate results like a standard query. Instead, it creates a new table within the database structure. The 'OctoberPurchases' view will display the data as defined in the SQL code. This view simplifies the process of querying total purchases and prices for each customer in October 2023, making the data more accessible and easier to interpret.

8. Deleting Customers Who Never Bought a Product

In this part of the SQL assignment, the objective was to write a query to delete all customers from the 'customer' table who have never made a purchase from the business. This was achieved by identifying customers whose IDs do not appear in the 'sales_transaction' table, indicating that they have not engaged in any transaction.

```

62 -- 8. Delete all customers who never bought a product from the business.
63 DELETE FROM customer
64 WHERE customer_id NOT IN (
65     SELECT DISTINCT customer_id
66     FROM sales_transaction
67 )

```

```

DELETE FROM customer

WHERE customer_id NOT IN (

    SELECT DISTINCT customer_id

    FROM sales_transaction

)

```

Output:

```

Execution finished without errors.
Result: query executed successfully. Took 1ms, 41 rows affected
At line 63:
DELETE FROM customer
WHERE customer_id NOT IN (
    SELECT DISTINCT customer_id
    FROM sales_transaction
)

```

The output of this query does not display data but rather indicates the number of rows affected. It shows how many customer records were deleted from the 'Customers' table, specifically those who had no corresponding entries in the 'Sales_Transactions' table, signifying they have never made a purchase.

9. Query to List Customers Whose Names Start with 'B'

For this task, I wrote a simple SQL query to retrieve the names of all customers whose names begin with the letter 'B'. The query utilises the LIKE operator with the pattern 'B%' to match any customer names starting with 'B'. The '%' symbol following 'B' ensures that the search includes all names that start with 'B', regardless of what follows after.

```

69 -- 9. List all the customers whose name starts with B.
70 SELECT *
71 FROM customer
72 WHERE customer_name
73 LIKE 'B%'
74

```

```

SELECT *

```

```

FROM customer

WHERE customer_name

LIKE 'B%'

```

Output:

	customer_id	customer_name	contact	address	city	country	postal_code
1	6	Brynn Christie	bchristie5@google.de	68 Forest Junction	Bicaj	Albania	NULL
2	16	Brock Oran	boranf@imageshack.us	584 Dexter Road	Kalety	Poland	42-660
3	29	Burnaby Maundrell	bmaundrells@netlog.com	0728 Buell Circle	Hanggan	Philippines	6406
4	36	Brynn Goare	bgoarez@cocolog-nifty.com	43170 Ridge Oak Alley	Sneek	Netherlands	8604
5	48	Becki Chipperfield	bchipperfield1b@gov.uk	1593 High Crossing Drive	Maluno Sur	Philippines	3332
6	77	Berta Atchly	batchly24@reference.com	05 Londonderry Circle	Zongga	China	NULL
7	83	Bobbie Achurch	bachurch2a@youtube.com	67 Welch Road	Koltushi	Russia	188680
8	127	Beilul Fedynski	bfedynski3i@instagram.com	21 Columbus Drive	Vágia	Greece	NULL
9	150	Benedict Rzehorz	brzehorz45@hao123.com	4166 Bultman Park	Sobang	Indonesia	NULL
10	162	Brandais McGovern	bmcgovern4h@simplemachine...	1940 Fieldstone Avenue	Jiuchenggong	China	NULL

The output of this query displays all records from the 'customer' table where the 'customer_name' begins with 'B'. This includes the complete row of data for each qualifying customer, such as customer ID, name, contact details, and address. The description of the output is straightforward, as the result is a simple list of customers filtered by the specified condition.

10. Identifying the Top Supplier in Product Sales

In this task, I aimed to determine which supplier sold the most products. To achieve this, I utilized the 'product' table, which contains columns for product ID and supplier ID. I performed a join with the 'sales_items' table to ascertain the quantity of products sold. Additionally, I joined the 'supplier' table to obtain the names of the suppliers. The final step involved grouping the data by supplier, sorting it in descending order based on the quantity of products sold, and limiting the output to the top supplier. This approach effectively answered the question of which supplier sold the most products.

```

75 -- 10. What supplier sold more products?
76 SELECT p.supplier_id, s.supplier_name, sum(si.quantity) products_sold
77 FROM product p
78 JOIN sales_items si on si.product_id = p.product_id
79 JOIN supplier s on s.supplier_id = p.supplier_id
80 GROUP by p.supplier_id
81 ORDER by products_sold DESC
82 LIMIT 1

```

```

SELECT p.supplier_id, s.supplier_name, sum(si.quantity) products_sold

FROM product p

JOIN sales_items si on si.product_id = p.product_id

```

JOIN supplier s on s.supplier_id = p.supplier_id

GROUP by p.supplier_id

ORDER by products_sold DESC

LIMIT 1

Output:

	supplier_id	supplier_name	products_sold
1	16	Fatz	92

The output of this SQL query displays the supplier ID, supplier name, and the total number of products sold by the top supplier.

End