Step 1: Dataset Preparation and Import

I split the original customer support dataset into four normalized tables—**customers**, **orders**, **agents**, and **tickets**—to simulate a relational database structure. These CSV files were then imported into SQLite for performing SQL-based analysis. The structured tables enabled efficient querying using JOINs, aggregate functions, and filtering. All files have been added to my GitHub repository.

Data path : https://github.com/Reenakalkandha1234/Task-3-Elevate-labs

Step 2: SQL Query Execution and Analysis:

Using the structured tables imported into SQLite, I performed various SQL queries to extract insights and analyze customer support data. The queries made use of core SQL clauses and operations such as:

- SELECT, WHERE, ORDER BY, GROUP BY for filtering and grouping data
- Aggregate functions like SUM() and AVG() to calculate total and average values
- JOIN operations between orders and customers tables to analyze location-based sales
- Grouped analysis on agents and tickets tables to understand CSAT scores and ticket categories

The queries were written, executed, and tested to ensure accuracy and efficiency. Screenshots of each query and its output were captured and included in the below:

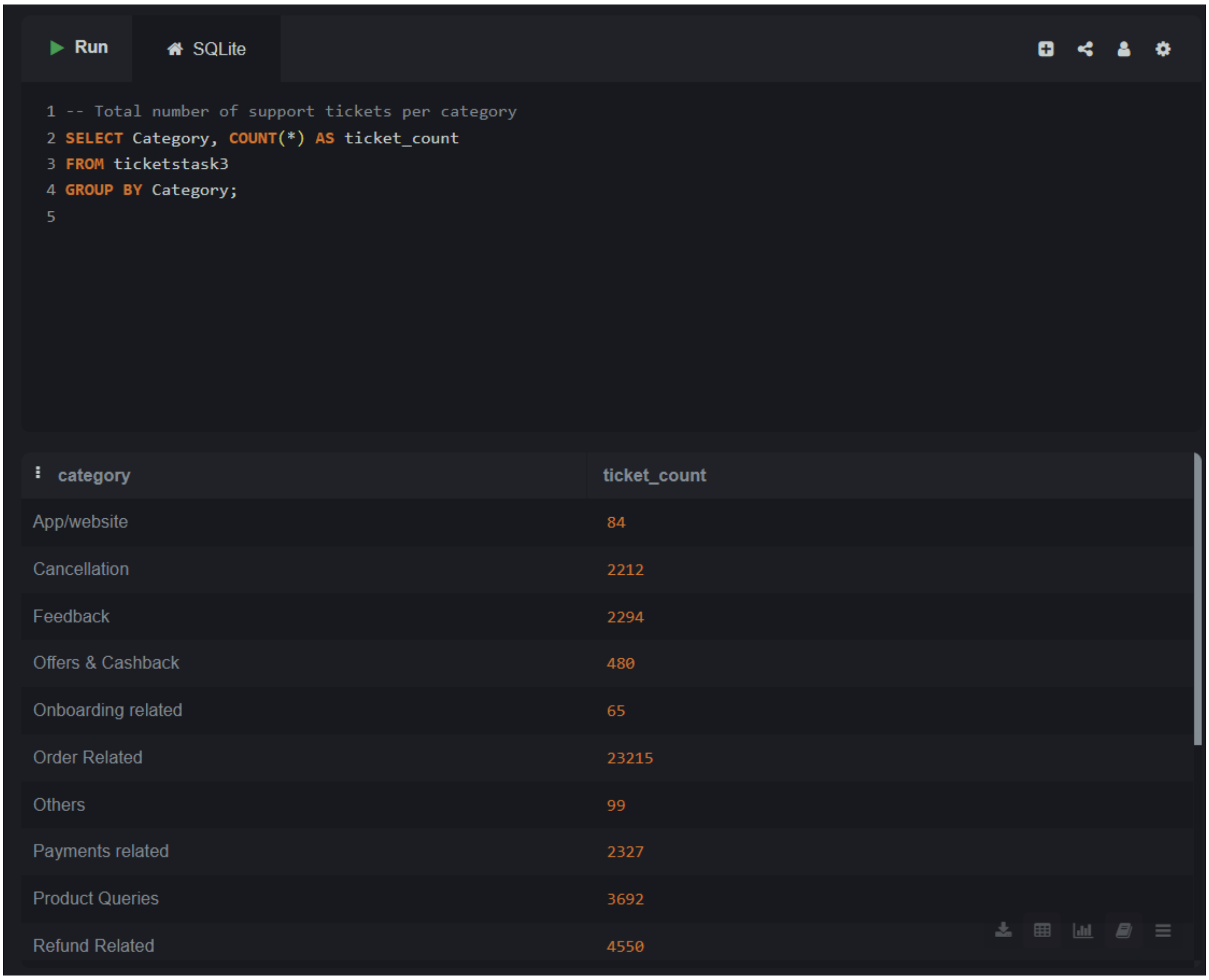
Basic Queries:

1st) Quarry

```
-- Total number of support tickets per category

SELECT Category, COUNT(*) AS ticket_count

FROM tickets

GROUP BY Category;
```

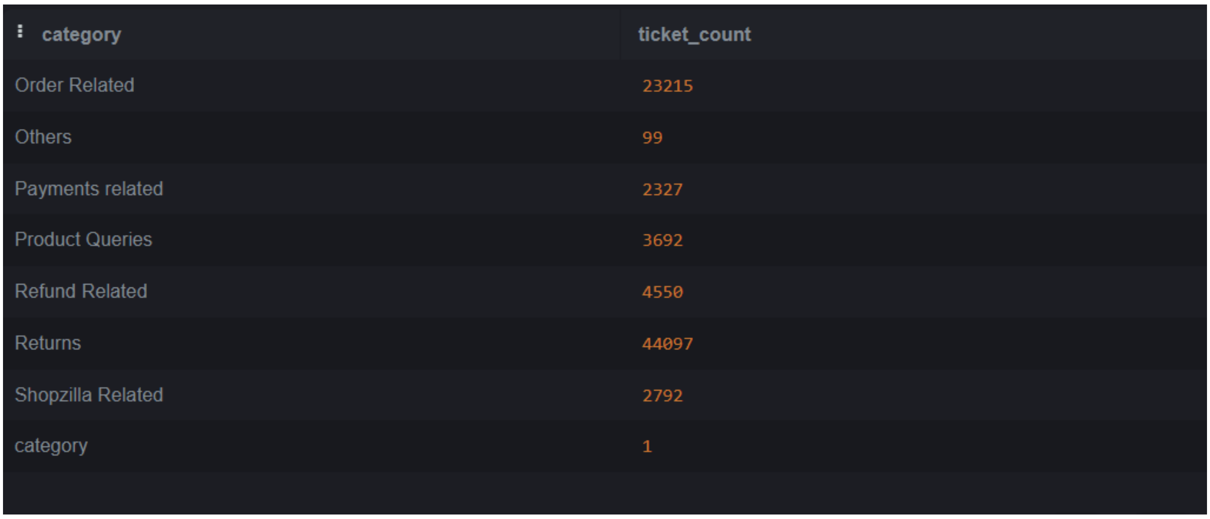
OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```sql
-- Total number of support tickets per category
SELECT Category, COUNT(*) AS ticket_count
FROM ticketstask3
GROUP BY Category;

```

category	ticket_count
App/website	84
Cancellation	2212
Feedback	2294
Offers & Cashback	480
Onboarding related	65
Order Related	23215
Others	99
Payments related	2327
Product Queries	3692
Refund Related	4550



category	ticket_count
Order Related	23215
Others	99
Payments related	2327
Product Queries	3692
Refund Related	4550
Returns	44097
Shopzilla Related	2792
category	1

--Use of SELECT, WHERE, ORDER BY, GROUP BY:

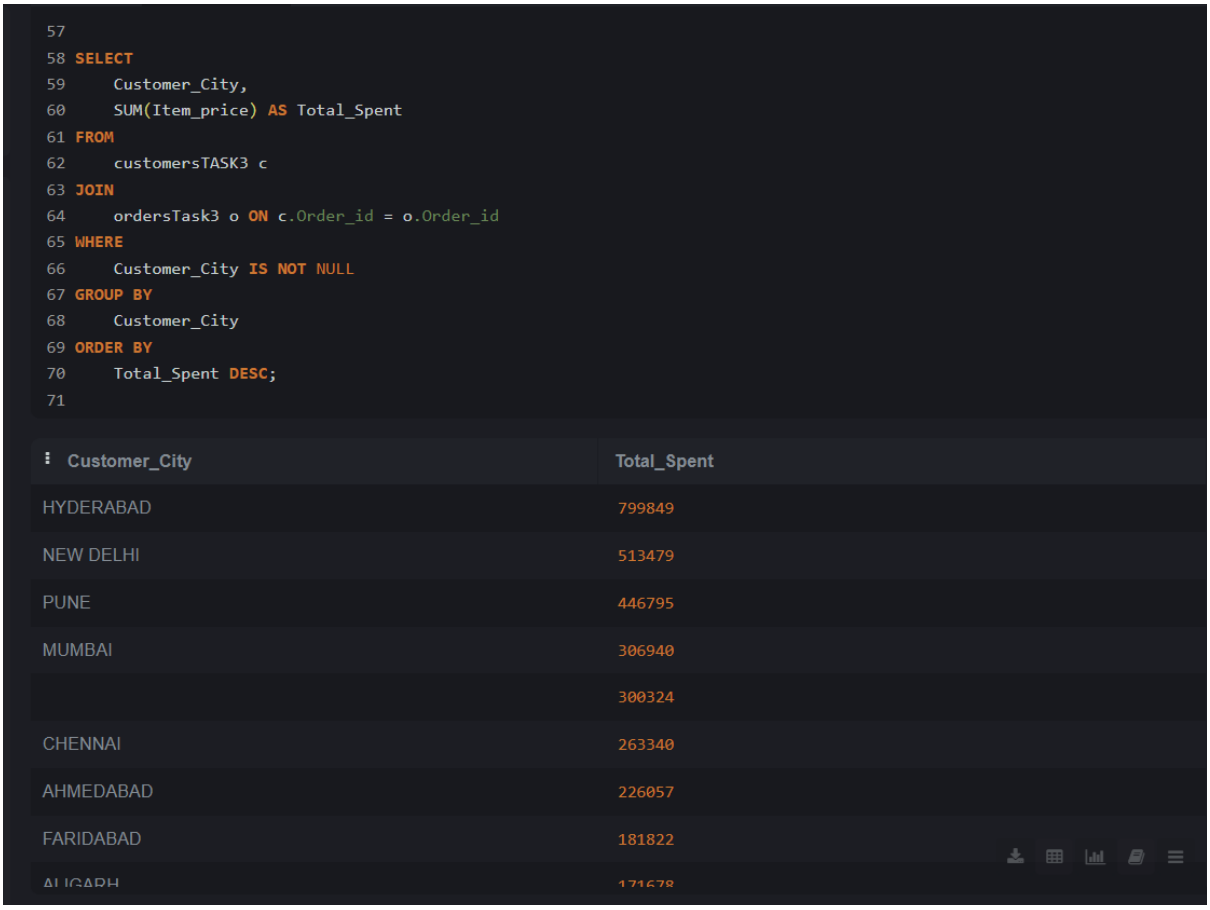
I used fundamental SQL clauses to retrieve, filter, sort, and group the data effectively.

This query selects and groups customer spending by city, filters out null values using WHERE, and sorts the results in descending order of total amount spent.

```
SELECT
    Customer_City,
    SUM(Item_price) AS Total_Spent
FROM
    customersTASK3 c
JOIN
    ordersTask3 o ON c.Order_id = o.Order_id
WHERE
    Customer_City IS NOT NULL
GROUP BY
    Customer_City
ORDER BY
    Total_Spent DESC;
```

OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```sql
SELECT
    Customer_City,
    SUM(Item_price) AS Total_Spent
FROM
    customersTASK3 c
JOIN
    ordersTask3 o ON c.Order_id = o.Order_id
WHERE
    Customer_City IS NOT NULL
GROUP BY
    Customer_City
ORDER BY
    Total_Spent DESC;
```

Customer_City	Total_Spent
HYDERABAD	799849
NEW DELHI	513479
PUNE	446795
MUMBAI	306940
	300324
CHENNAI	263340
AHMEDABAD	226057
FARIDABAD	181822
ALIGARH	171678

--Use of JOINS (INNER, LEFT, RIGHT)

I used JOIN operations to combine data from multiple related tables based on common fields like Order_id.

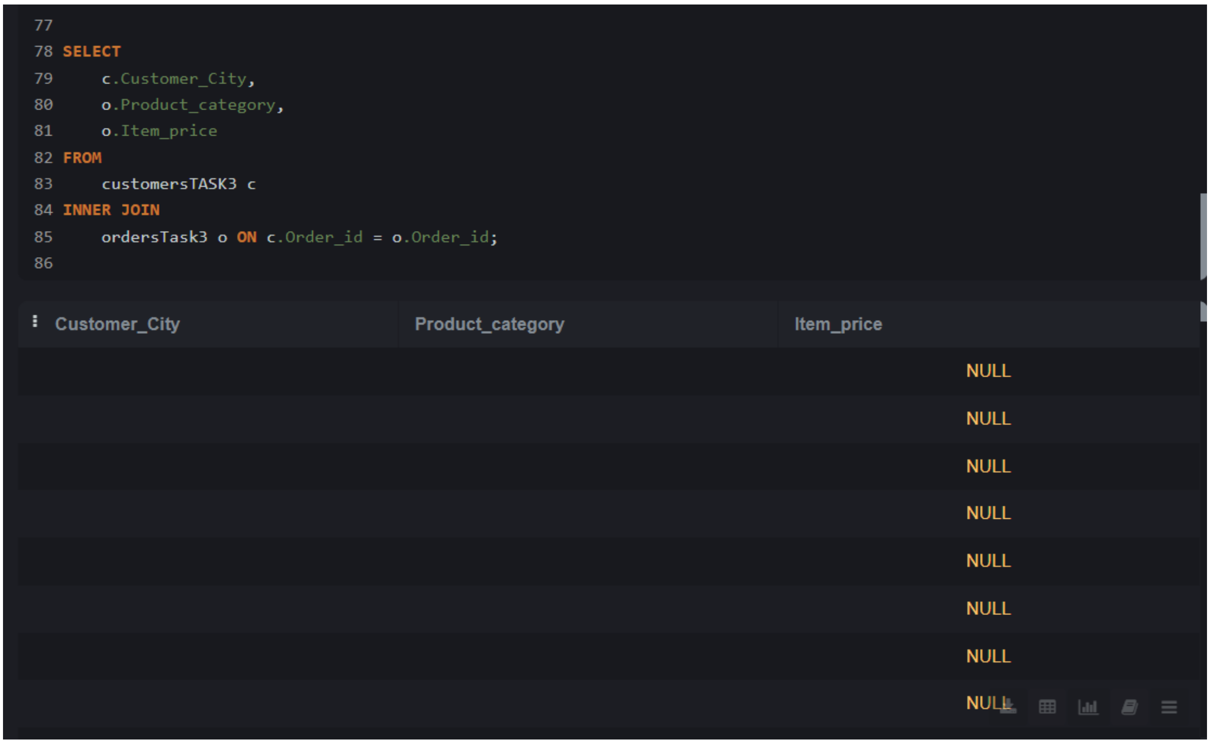
INNER JOIN Example:

```
SELECT
    c.Customer_City,
    o.Product_category,
    o.Item_price
FROM
    customersTASK3 c
INNER JOIN
    ordersTask3 o ON c.Order_id = o.Order_id;
```

OUTPUT:

Screenshots of queries and their respective outputs are provided below:

Combines customers and orders to show city-wise purchase details



```
SELECT
    c.Customer_City,
    o.Product_category,
    o.Item_price
FROM
    customersTASK3 c
INNER JOIN
    ordersTask3 o ON c.Order_id = o.Order_id;
```

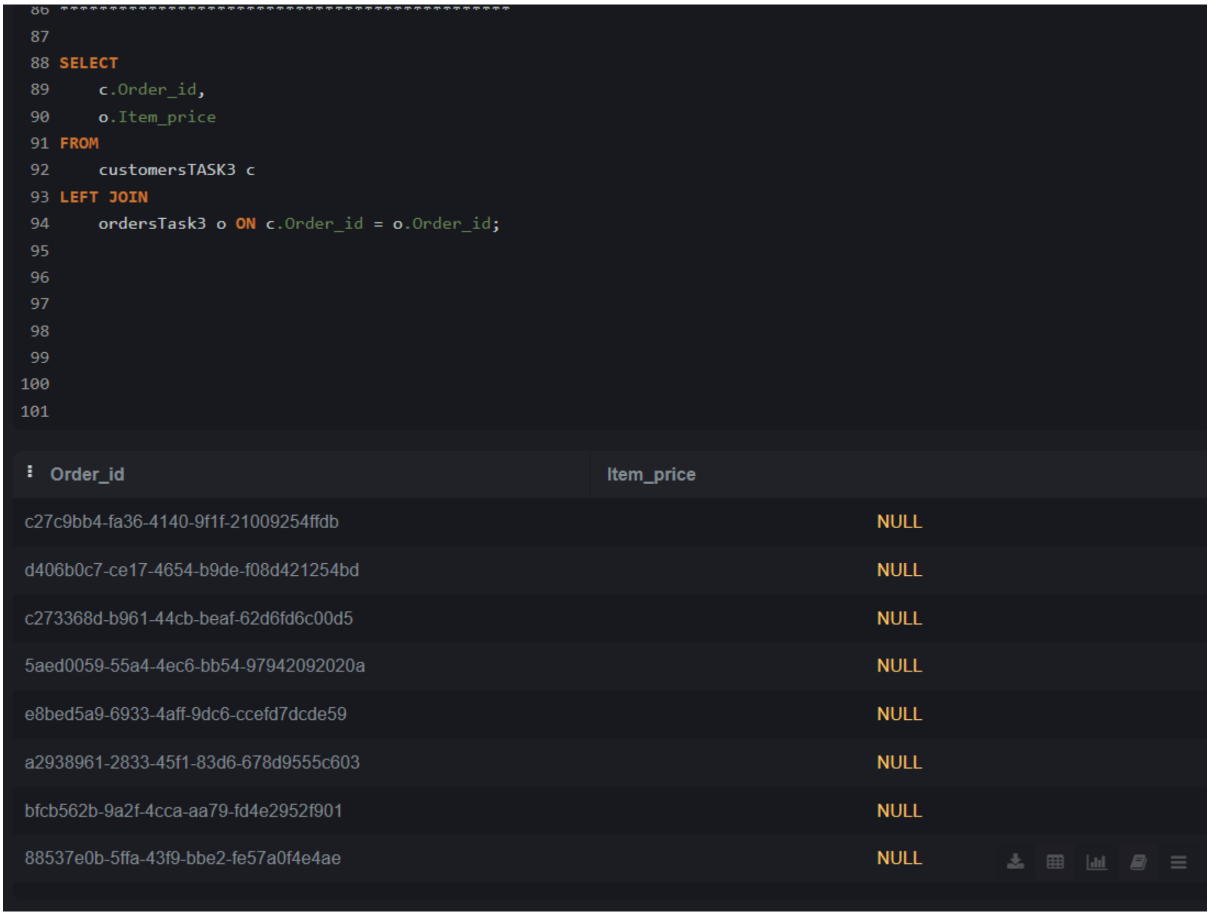
Customer_City	Product_category	Item_price
		NULL
		NULL
		NULL
		NULL
		NULL
		NULL
		NULL
		NULL

LEFT JOIN Example:

```
 SELECT
    c.Order_id,
    o.Item_price
FROM
    customers c
LEFT JOIN
    orders o ON c.Order_id = o.Order_id;
```

OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```sql
**********************************************

SELECT
    c.Order_id,
    o.Item_price
FROM
    customersTASK3 c
LEFT JOIN
    ordersTask3 o ON c.Order_id = o.Order_id;
```

Order_id	Item_price
c27c9bb4-fa36-4140-9f1f-21009254ffdb	NULL
d406b0c7-ce17-4654-b9de-f08d421254bd	NULL
c273368d-b961-44cb-beaf-62d6fd6c00d5	NULL
5aed0059-55a4-4ec6-bb54-97942092020a	NULL
e8bed5a9-6933-4aff-9dc6-ccefd7dcde59	NULL
a2938961-2833-45f1-83d6-678d9555c603	NULL
bfcb562b-9a2f-4cca-aa79-fd4e2952f901	NULL
88537e0b-5ffa-43f9-bbe2-fe57a0f4e4ae	NULL

RIGHT JOIN :

SELECT

  o.Order_id,

  c.Customer_City

FROM

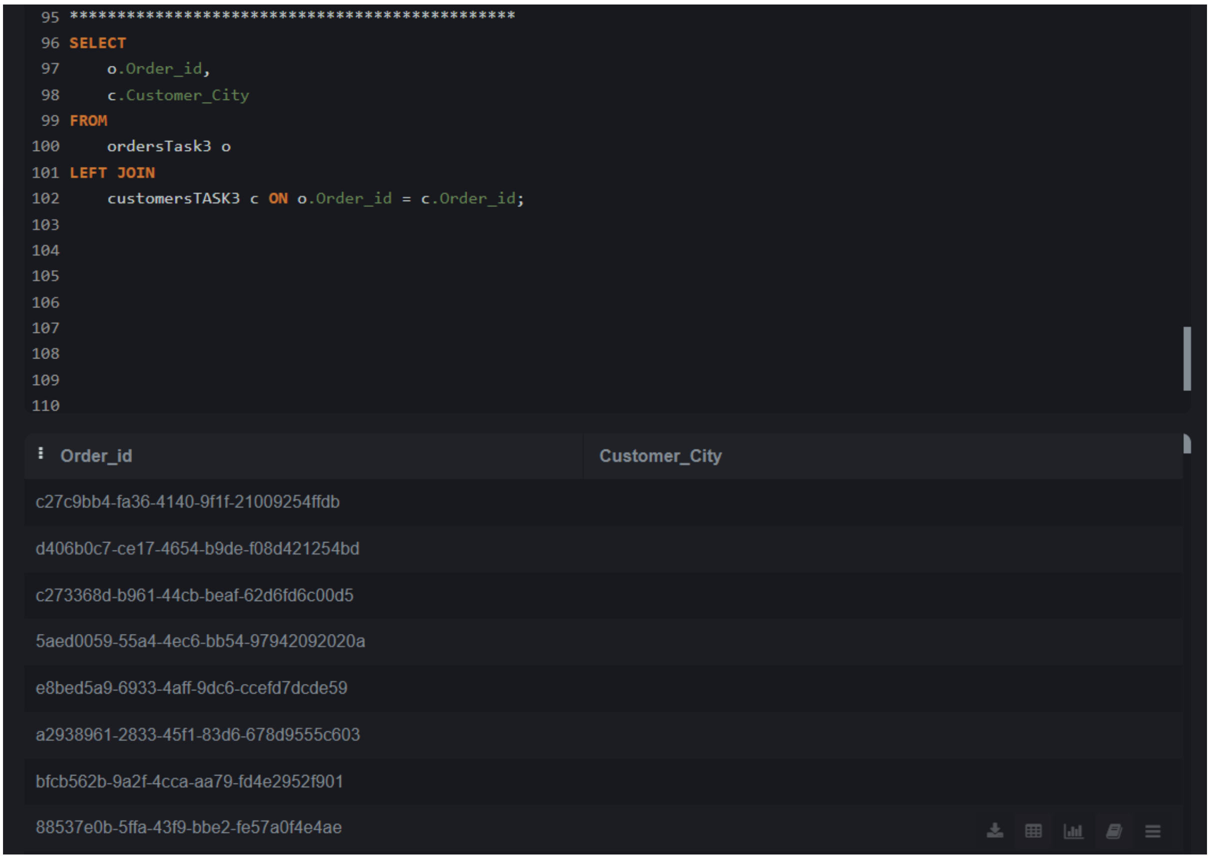
  ordersTask3 o

LEFT JOIN

  customersTASK3 c ON o.Order_id = c.Order_id;

OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```sql
***********************************************
SELECT
    o.Order_id,
    c.Customer_City
FROM
    ordersTask3 o
LEFT JOIN
    customersTASK3 c ON o.Order_id = c.Order_id;
```

Order_id	Customer_City
c27c9bb4-fa36-4140-9f1f-21009254ffdb	
d406b0c7-ce17-4654-b9de-f08d421254bd	
c273368d-b961-44cb-beaf-62d6fd6c00d5	
5aed0059-55a4-4ec6-bb54-97942092020a	
e8bed5a9-6933-4aff-9dc6-ccefd7dcde59	
a2938961-2833-45f1-83d6-678d9555c603	
bfcb562b-9a2f-4cca-aa79-fd4e2952f901	
88537e0b-5ffa-43f9-bbe2-fe57a0f4e4ae	

Use of Subqueries:

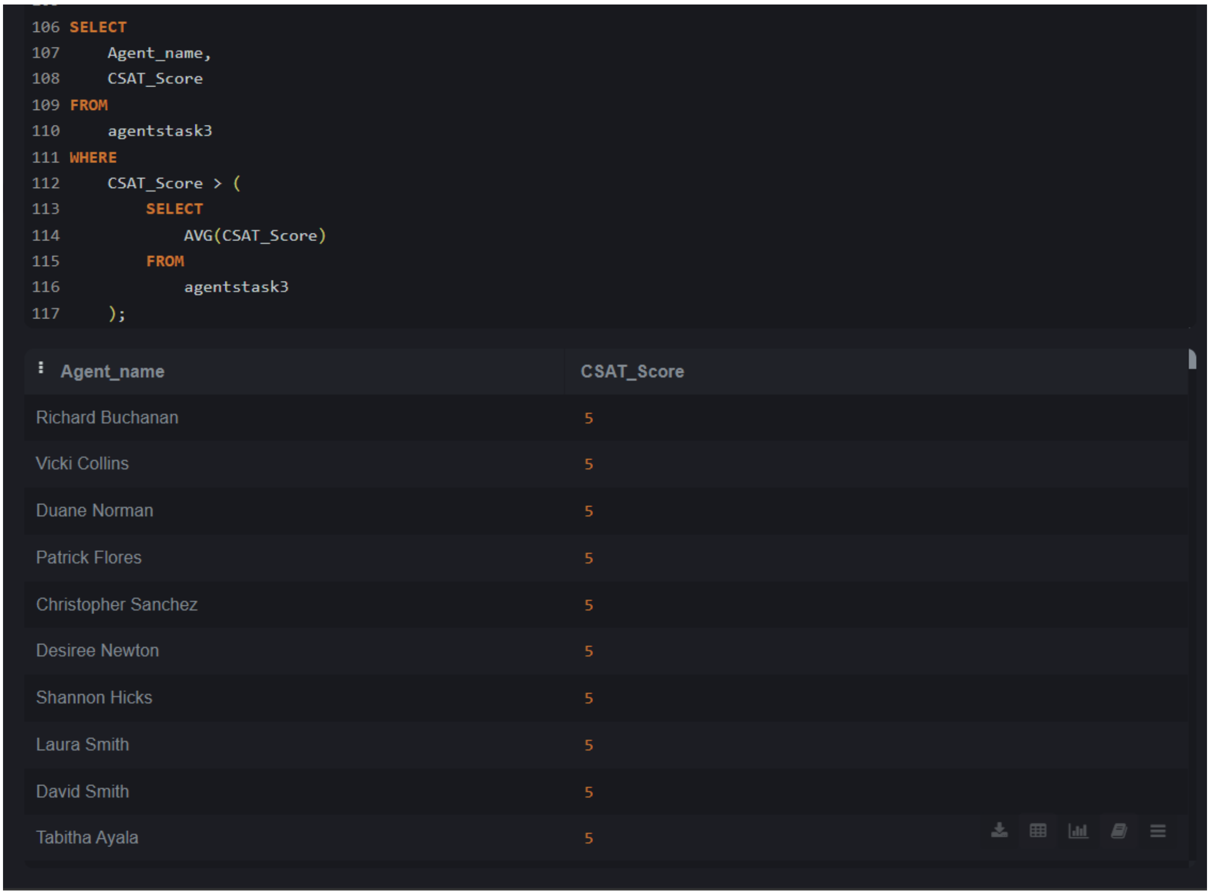
I used subqueries to perform calculations within a query, such as filtering based on aggregate values.

```sql
SELECT
    Agent_name,
    CSAT_Score
FROM
    agentstask3
WHERE
    CSAT_Score > (
        SELECT
            AVG(CSAT_Score)
        FROM
            agentstask3
    );
```

OUTPUT:

Screenshots of queries and their respective outputs are provided below:

This returns all agents whose CSAT score is above the average score of all agents



```sql
SELECT
    Agent_name,
    CSAT_Score
FROM
    agentstask3
WHERE
    CSAT_Score > (
        SELECT
            AVG(CSAT_Score)
        FROM
            agentstask3
    );
```

Agent_name	CSAT_Score
Richard Buchanan	5
Vicki Collins	5
Duane Norman	5
Patrick Flores	5
Christopher Sanchez	5
Desiree Newton	5
Shannon Hicks	5
Laura Smith	5
David Smith	5
Tabitha Ayala	5

Join Example:

-- Total spending by each customer city

SELECT c.Customer_City, SUM(o.Item_price) AS Total_Spent

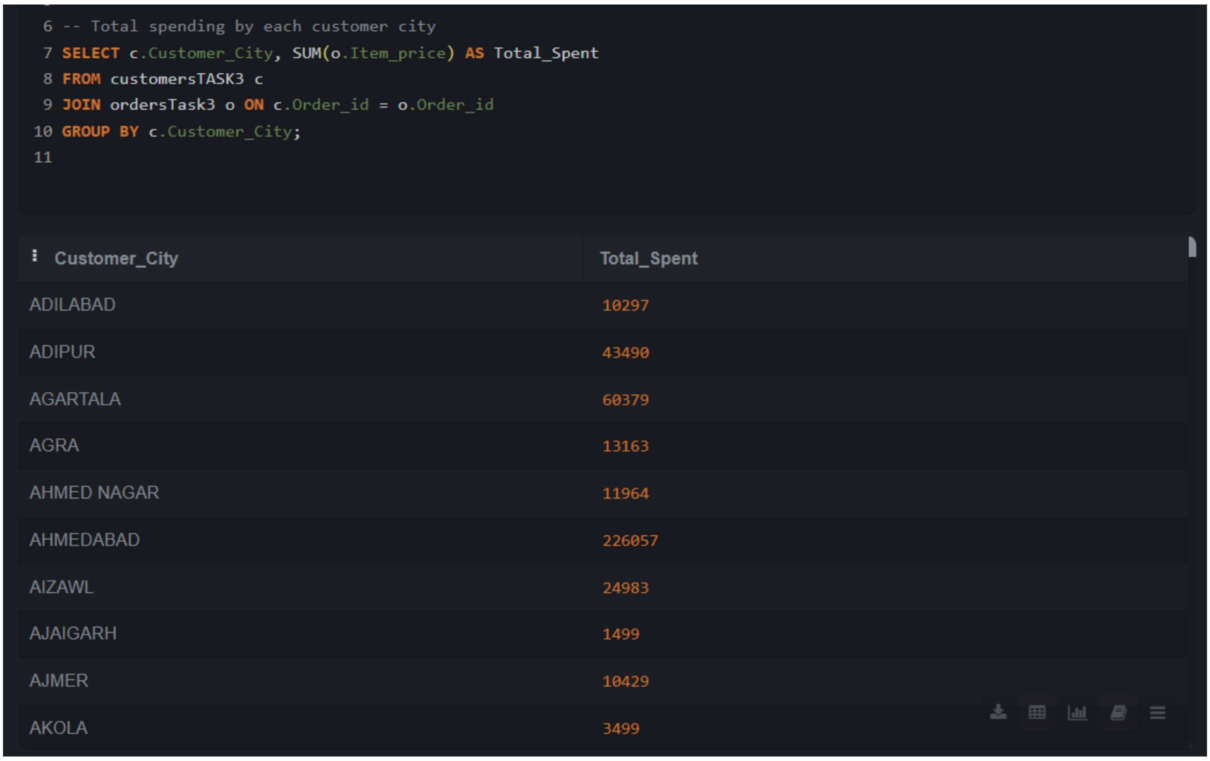
FROM customersTASK3 c

JOIN ordersTask3 o ON c.Order_id = o.Order_id

GROUP BY c.Customer_City;

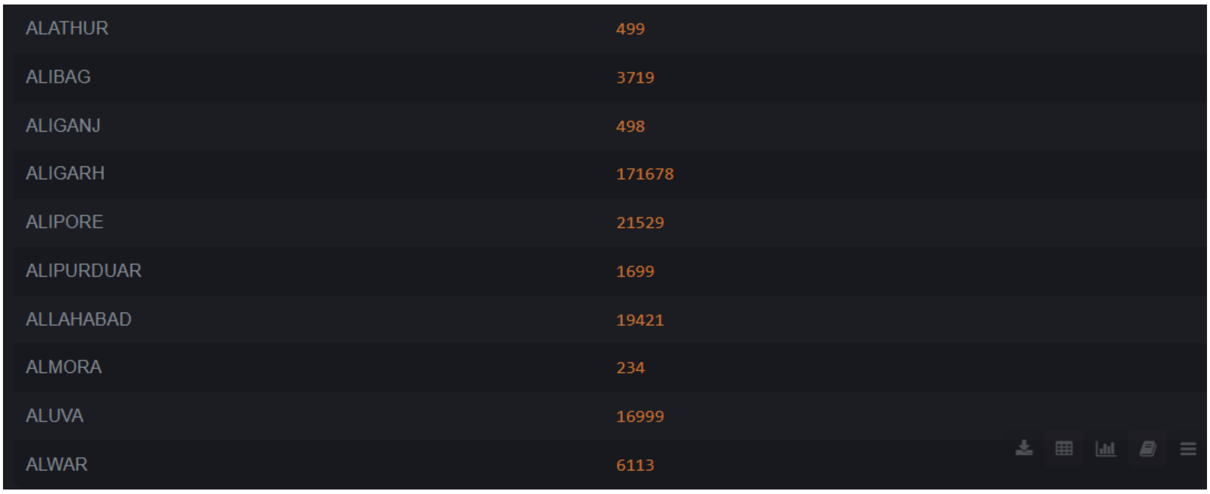
OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```sql
-- Total spending by each customer city
SELECT c.Customer_City, SUM(o.Item_price) AS Total_Spent
FROM customersTASK3 c
JOIN ordersTask3 o ON c.Order_id = o.Order_id
GROUP BY c.Customer_City;

```

Customer_City	Total_Spent
ADILABAD	10297
ADIPUR	43490
AGARTALA	60379
AGRA	13163
AHMED NAGAR	11964
AHMEDABAD	226057
AIZAWL	24983
AJAIGARH	1499
AJMER	10429
AKOLA	3499



ALATHUR	499
ALIBAG	3719
ALIGANJ	498
ALIGARH	171678
ALIPORE	21529
ALIPURDUAR	1699
ALLAHABAD	19421
ALMORA	234
ALUVA	16999
ALWAR	6113

Use aggregate functions (SUM, AVG)

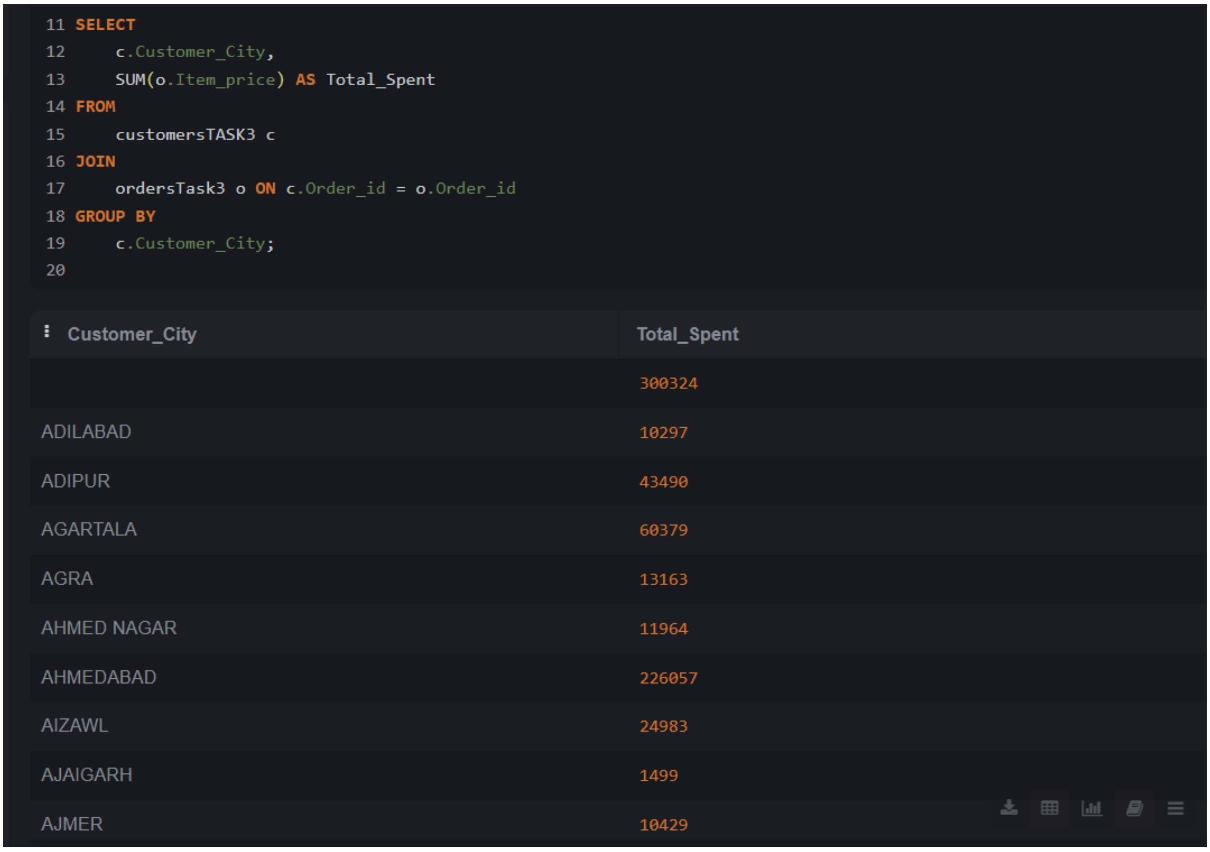
--1.)Total Item Price by City (SUM):

SELECT

```
    c.Customer_City,
    SUM(o.Item_price) AS Total_Spent
FROM
    customersTASK3 c
JOIN
    ordersTask3 o ON c.Order_id = o.Order_id
GROUP BY
    c.Customer_City;
```

OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```
SELECT
    c.Customer_City,
    SUM(o.Item_price) AS Total_Spent
FROM
    customersTASK3 c
JOIN
    ordersTask3 o ON c.Order_id = o.Order_id
GROUP BY
    c.Customer_City;
```

Customer_City	Total_Spent
	300324
ADILABAD	10297
ADIPUR	43490
AGARTALA	60379
AGRA	13163
AHMED NAGAR	11964
AHMEDABAD	226057
AIZAWL	24983
AJAIGARH	1499
AJMER	10429

Output more which is in csv

-- 2.)Average Item Price by Product Category (AVG)

```
SELECT
    Product_category,
```

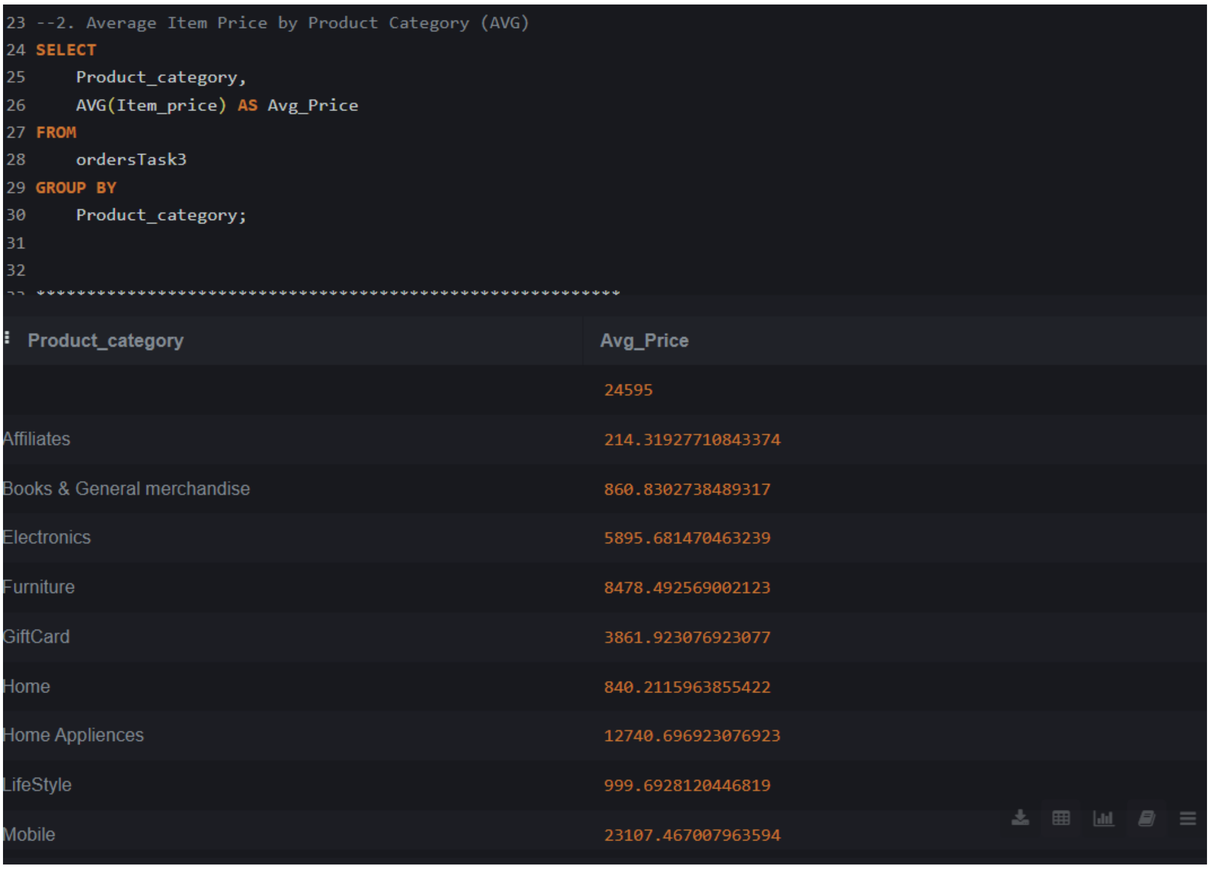
AVG(Item_price) AS Avg_Price

FROM

ordersTask3

GROUP BY

Product_category;



```
--2. Average Item Price by Product Category (AVG)
SELECT
    Product_category,
    AVG(Item_price) AS Avg_Price
FROM
    ordersTask3
GROUP BY
    Product_category;
```

Product_category	Avg_Price
	24595
Affiliates	214.31927710843374
Books & General merchandise	860.8302738489317
Electronics	5895.681470463239
Furniture	8478.492569002123
GiftCard	3861.923076923077
Home	840.2115963855422
Home Appliences	12740.696923076923
LifeStyle	999.6928120446819
Mobile	23107.467007963594

--3). Total Tickets by Category (COUNT + GROUP BY)

SELECT

Category,

COUNT(*) AS Ticket_Count

FROM

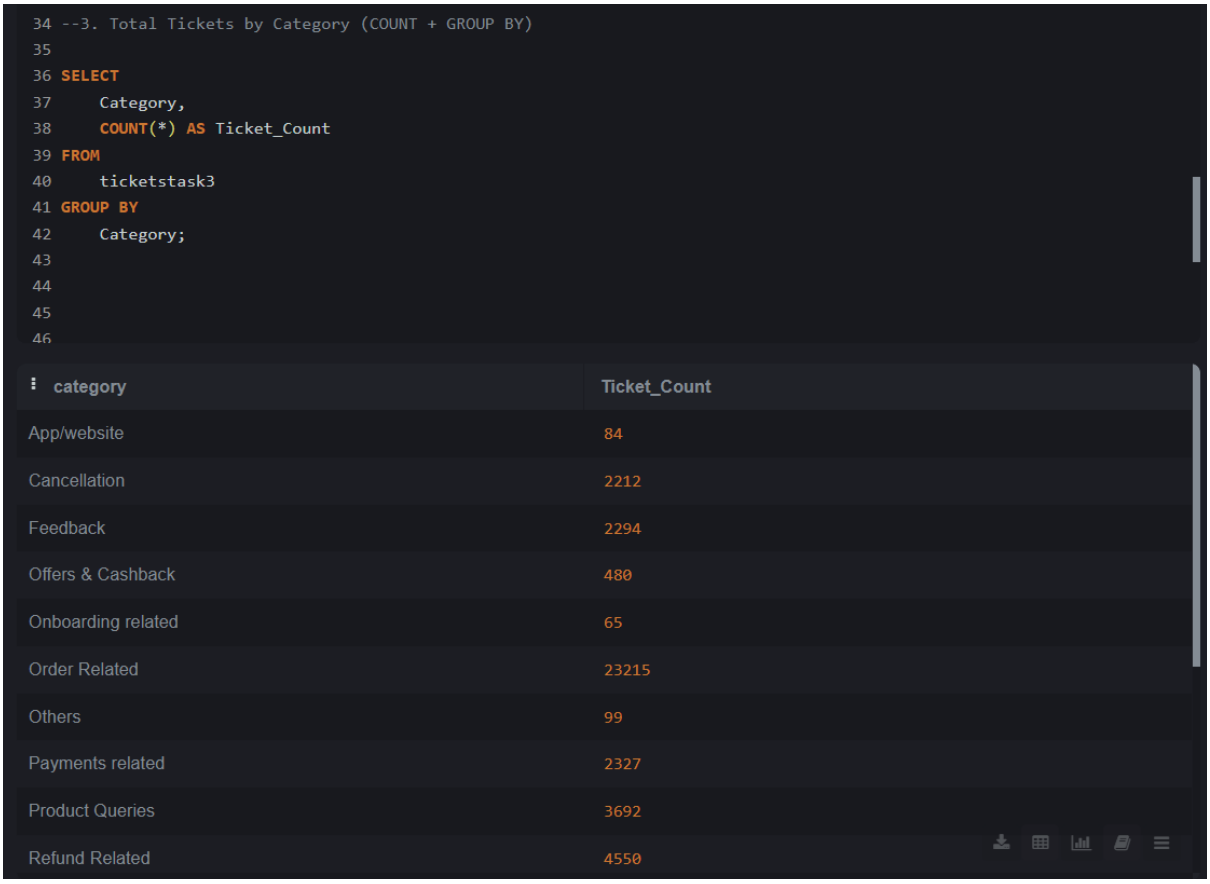
ticketstask3

GROUP BY

Category;

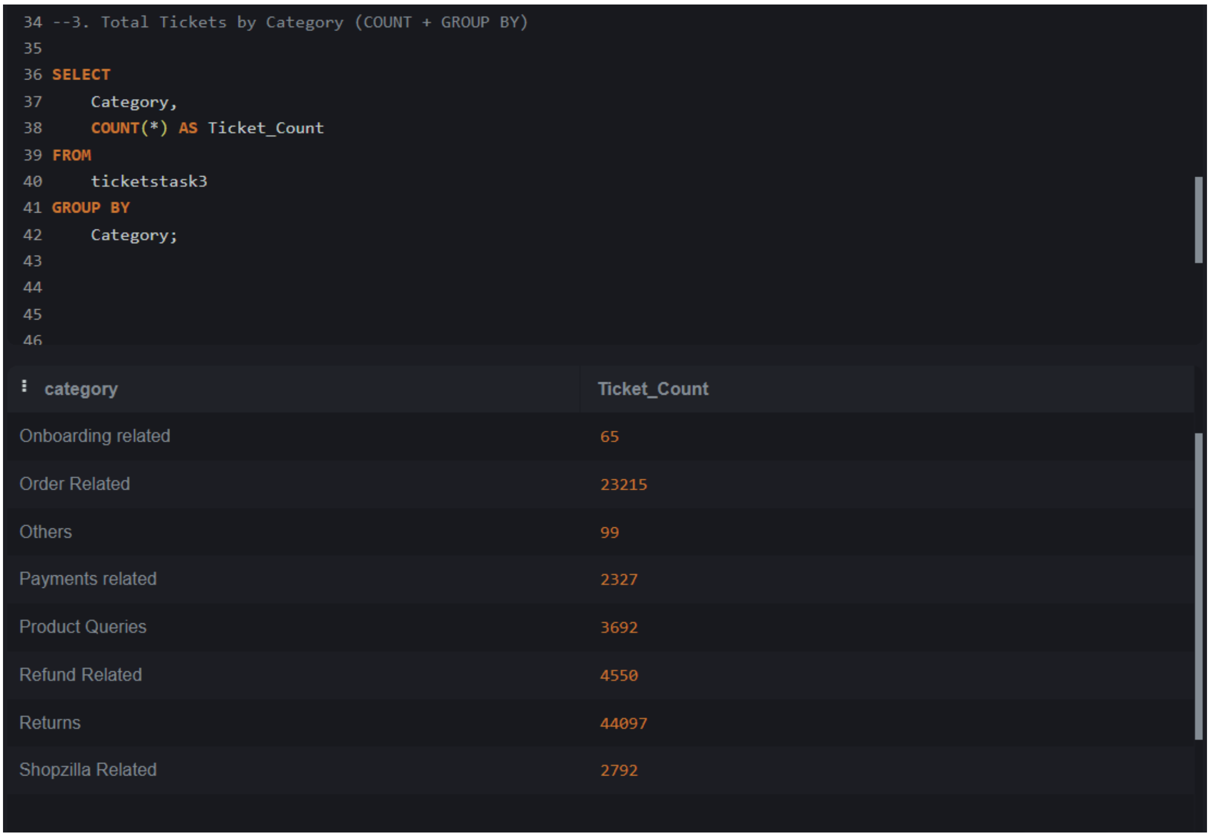
OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```
--3. Total Tickets by Category (COUNT + GROUP BY)

SELECT
    Category,
    COUNT(*) AS Ticket_Count
FROM
    ticketstask3
GROUP BY
    Category;
```

category	Ticket_Count
App/website	84
Cancellation	2212
Feedback	2294
Offers & Cashback	480
Onboarding related	65
Order Related	23215
Others	99
Payments related	2327
Product Queries	3692
Refund Related	4550



```
--3. Total Tickets by Category (COUNT + GROUP BY)

SELECT
    Category,
    COUNT(*) AS Ticket_Count
FROM
    ticketstask3
GROUP BY
    Category;
```

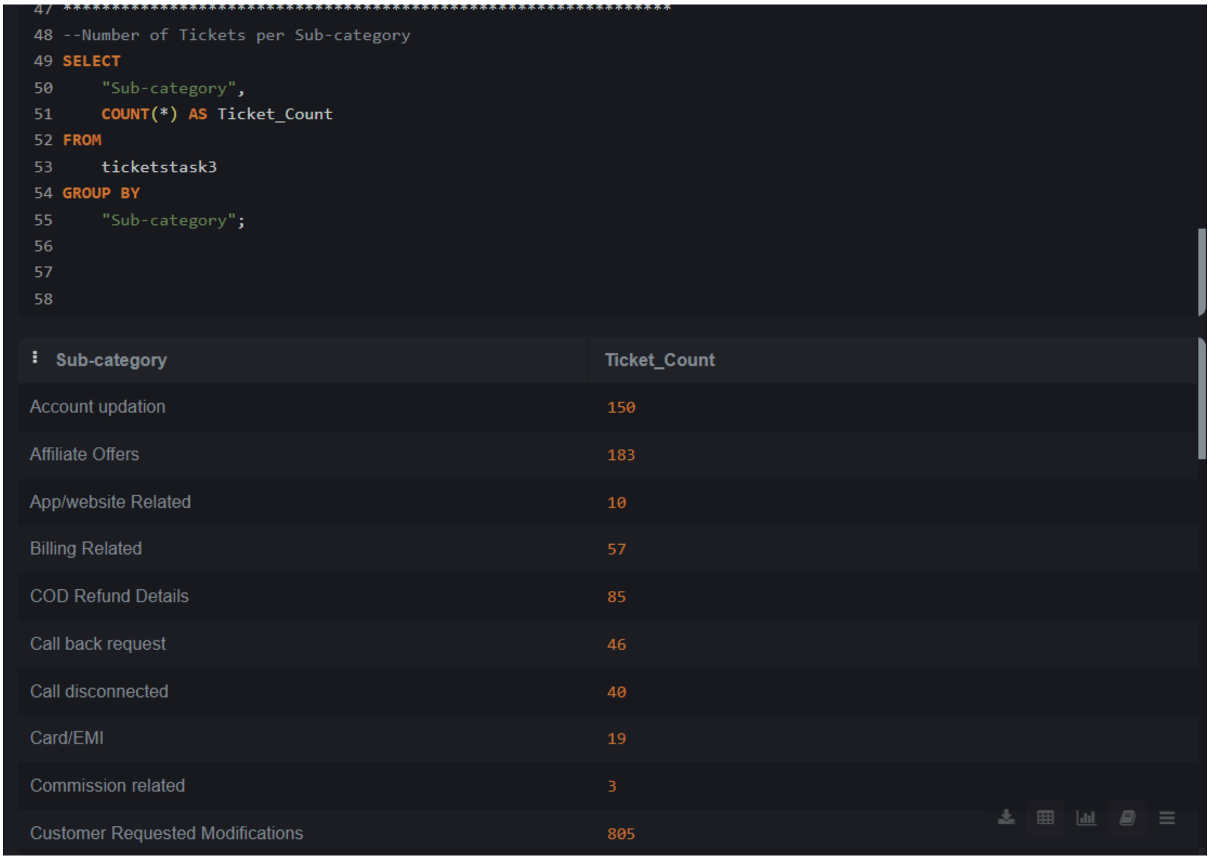
category	Ticket_Count
Onboarding related	65
Order Related	23215
Others	99
Payments related	2327
Product Queries	3692
Refund Related	4550
Returns	44097
Shopzilla Related	2792

--4.) Number of Tickets per Sub-category:

```
SELECT
    "Sub-category",
    COUNT(*) AS Ticket_Count
FROM
    ticketstask3
GROUP BY
    "Sub-category";
```

OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```
***************************************************************
--Number of Tickets per Sub-category
SELECT
    "Sub-category",
    COUNT(*) AS Ticket_Count
FROM
    ticketstask3
GROUP BY
    "Sub-category";



```

Sub-category	Ticket_Count
Account updation	150
Affiliate Offers	183
App/website Related	10
Billing Related	57
COD Refund Details	85
Call back request	46
Call disconnected	40
Card/EMI	19
Commission related	3
Customer Requested Modifications	805

--5.) Average CSAT Score by Agent Shift:

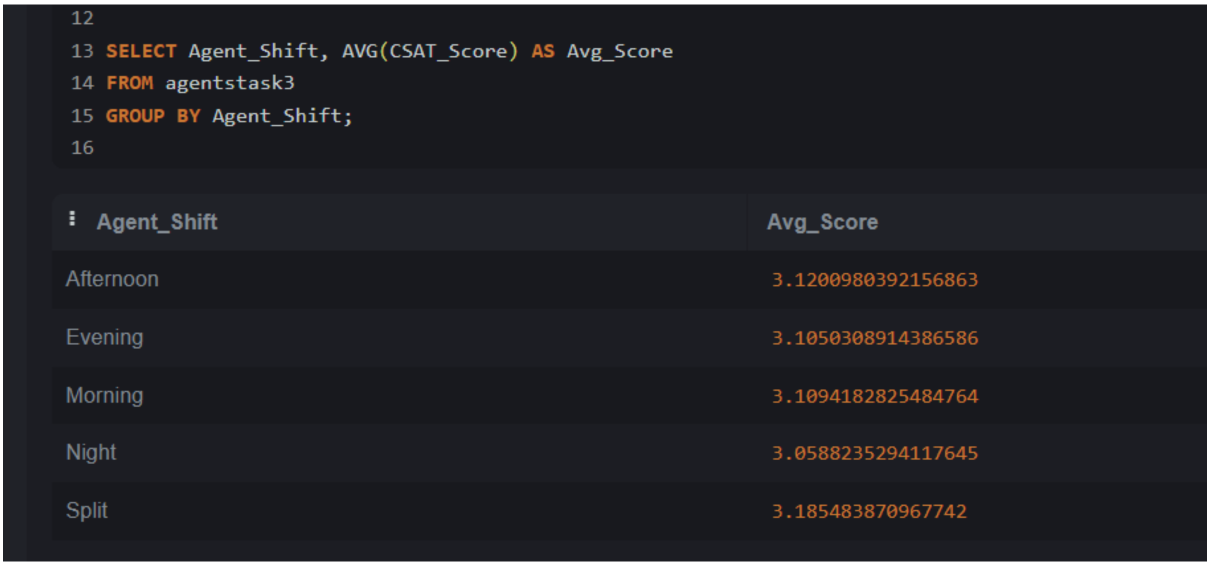
SELECT Agent_Shift, AVG(CSAT_Score) AS Avg_Score

FROM agentstask3

GROUP BY Agent_Shift;

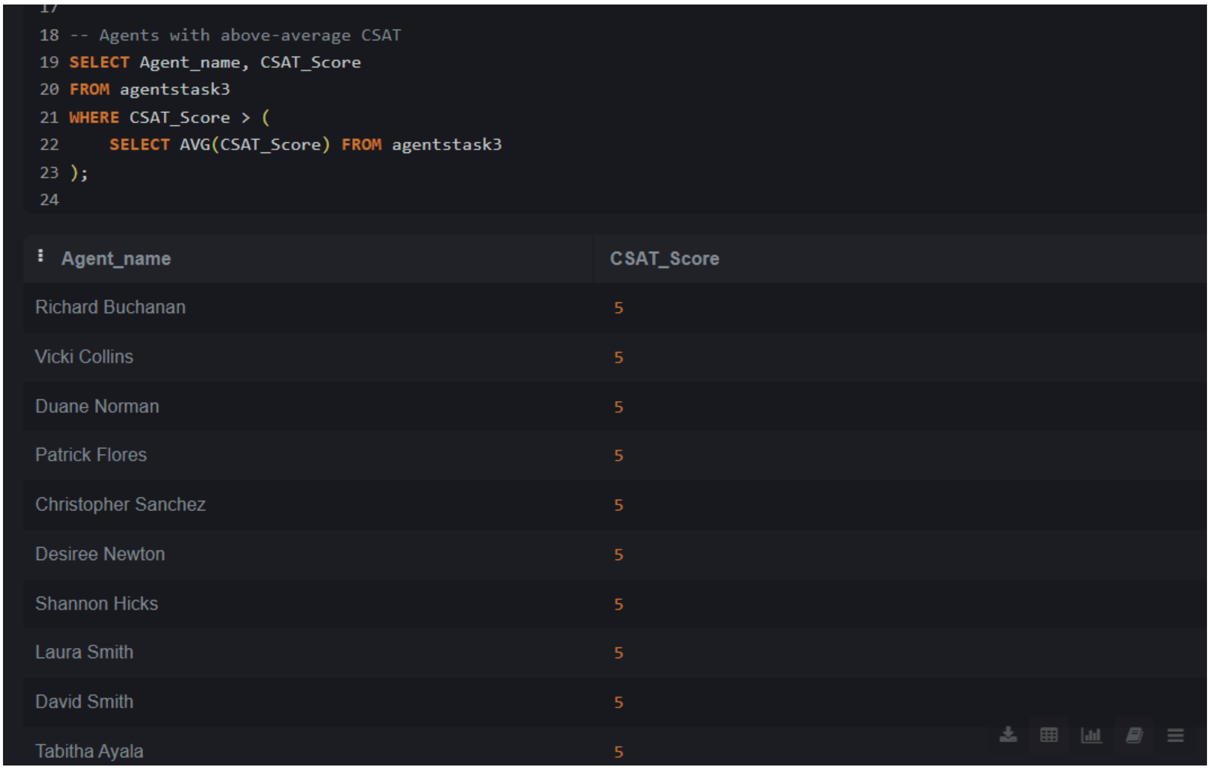
OUTPUT:

Screenshots of queries and their respective outputs are provided below:



```
SELECT Agent_Shift, AVG(CSAT_Score) AS Avg_Score
FROM agentstask3
GROUP BY Agent_Shift;
```

Agent_Shift	Avg_Score
Afternoon	3.1200980392156863
Evening	3.1050308914386586
Morning	3.1094182825484764
Night	3.0588235294117645
Split	3.185483870967742



```
-- Agents with above-average CSAT
SELECT Agent_name, CSAT_Score
FROM agentstask3
WHERE CSAT_Score > (
    SELECT AVG(CSAT_Score) FROM agentstask3
);
```

Agent_name	CSAT_Score
Richard Buchanan	5
Vicki Collins	5
Duane Norman	5
Patrick Flores	5
Christopher Sanchez	5
Desiree Newton	5
Shannon Hicks	5
Laura Smith	5
David Smith	5
Tabitha Ayala	5

Step 3 : Create views for analysis

To simplify repetitive queries and enable easier analysis, I created a **SQL View** using the CREATE VIEW statement. This view aggregates important agent performance metrics such as CSAT scores and shift details.

Code :

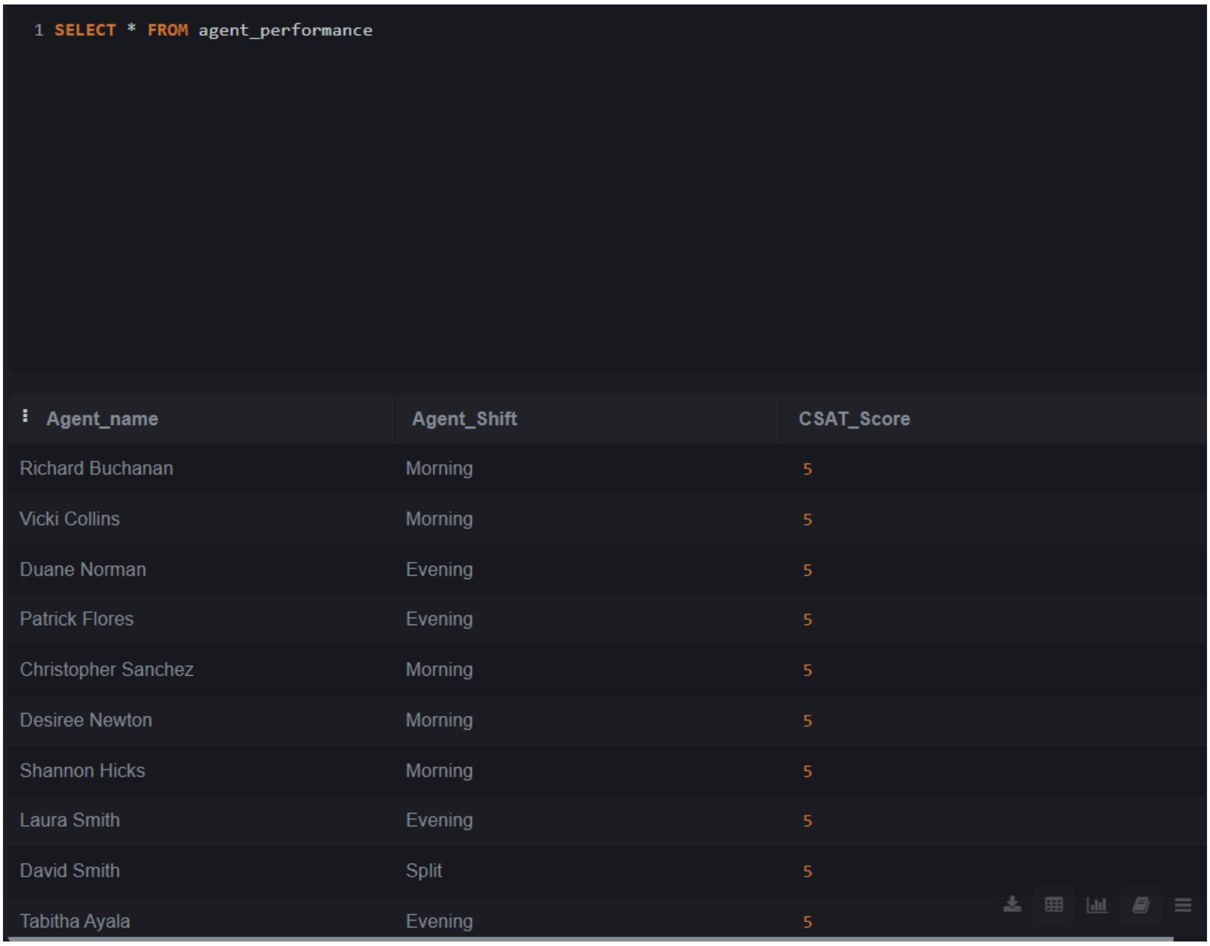
```
CREATE VIEW agent_performance AS
SELECT Agent_name, Agent_Shift, CSAT_Score
FROM agentstask3
WHERE CSAT_Score >= 4;
```

Output :

Screenshots of queries and their respective outputs are provided below:

Create a view as agent_performance

And view agent_performance is below:



```
SELECT * FROM agent_performance
```

Agent_name	Agent_Shift	CSAT_Score
Richard Buchanan	Morning	5
Vicki Collins	Morning	5
Duane Norman	Evening	5
Patrick Flores	Evening	5
Christopher Sanchez	Morning	5
Desiree Newton	Morning	5
Shannon Hicks	Morning	5
Laura Smith	Evening	5
David Smith	Split	5
Tabitha Ayala	Evening	5

This view helps in quickly retrieving high-performing agents based on CSAT scores and organizing them by shift and tenure. It can be reused in future queries for agent-related performance evaluations.

Step 4 : Optimize queries with indexes

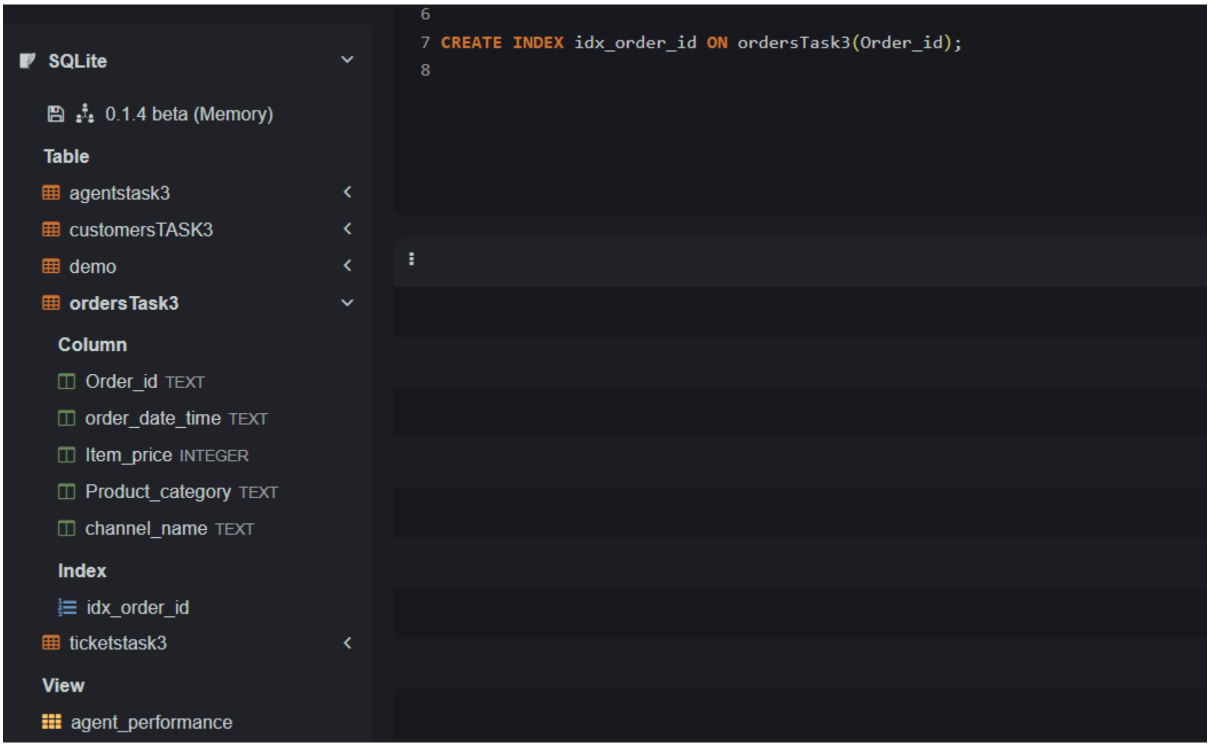
To improve query performance, especially for JOIN and WHERE operations on frequently searched columns, I implemented indexing using the CREATE INDEX statement. Indexes help speed up data retrieval without scanning the entire table.

The following index was created:

CREATE INDEX idx_order_id ON ordersTask3(Order_id);

Output :

Screenshots of queries and their respective outputs are provided below:



```sql

CREATE INDEX idx_order_id ON ordersTask3(Order_id);

```

This index optimizes queries that join the orders table with customers or tickets using Order_id, which is a common key across multiple tables. By indexing this column, the database can access matching rows more efficiently, leading to faster query execution.