Implementing SQL: TechCorp Analytics

Project By Hernandia Zanua Leras

BACKGROUND

TechCorp is an e-commerce company specializing in electronic devices including laptops, smartphones, and accessories, while also providing technical support services to customers. This project analyzes data across products, customers, orders, transaction details, employees, and support tickets to gain comprehensive insights into the company's operational activities. Through SQL queries, raw data is transformed into strategic information such as top-spending customers, unsold inventory, support staff performance, highest-priced product categories, and total revenue. This analysis provides a clear overview of TechCorp's sales performance and support service effectiveness, while demonstrating SQL's role in generating accurate insights for data-driven decision-making.

PROJECT OVERVIEW

Objective

- Analyze customer purchasing behavior and value
- Evaluate product performance and pricing strategy
- Assess employee productivity and revenue metrics

Tools

MySQL

This project utilizes MySQL as the primary database management system for data analysis

SQL ANALYSIS STAGES

- **Database & Table Structure Design**
  Created the TechCorp database from scratch, designed six core tables (products, customers, orders, order_details, employees, support_tickets), and established relational integrity through foreign key constraints.
- **Data Preparation & Cleaning**
  Inserted data into tables, created additional columns, ensured proper data types, and checked data consistency.
- **Analytical Querying**
  Executed queries to identify top customers, unsold products, revenue totals, employee performance, pricing trends, etc.

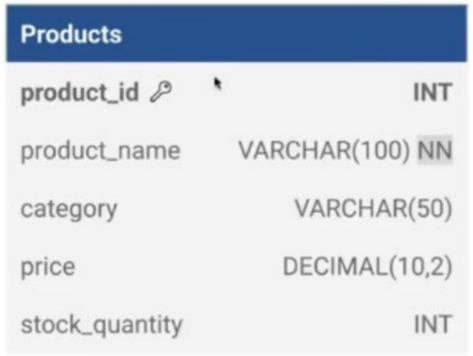
DATABASE SCHEMA

These databases are created manually in MySQL, from designing the table structure to entering the data. The database is as follows:

- Products Table → This table stores information about the products being sold.
- Customers Table → This table stores information about customers.
- Orders Table → This table stores information about orders placed by customers.
- Order Details Table (OrderDetails) →This table stores detailed information for each order.
- Employees Table → This table stores information about employees working at TechCorp.
- Support Tickets Table (SupportTickets) → This table stores information about support tickets submitted by customers.

DATABASE SCHEMA

1. Product Table



Products	
product_id	INT
product_name	VARCHAR(100) NN
category	VARCHAR(50)
price	DECIMAL(10,2)
stock_quantity	INT

- product_id: Unique identifier for each product (Primary Key).
- product_name: Product name.
- category: Product category (e.g., Laptop, Smartphone, Accessories).
- price: Product price.
- stock_quantity: Number of product units available in stock.
- discount: Discount applied to the product price, if any.

2. Customers Table

Customers	
customer_id	INT
first_name	VARCHAR(50) NN
last_name	VARCHAR(50) NN
email	VARCHAR(100)
phone	VARCHAR(20)
address	VARCHAR(200)

- customer_id: Unique identifier for each customer (Primary Key).
- first_name: Customer's first name.
- last_name: Customer's last name.
- email: Customer's email address.
- phone: Customer's phone number.
- address: Customer's residential address.

DATABASE SCHEMA

3. Orders Table

Orders	
order_id	INT
customer_id	INT
order_date	DATE
total_amount	DECIMAL(10,2)

- order_id: Unique identifier for each order (Primary Key).
- customer_id: Customer identifier who placed the order (Foreign Key referencing Customers.customer_id).
- order_date: Date when the order was placed.
- total_amount: Total monetary value of the order.

4. Order Details Table

OrderDetails	
order_detail_id	INT
order_id	INT
product_id	INT
quantity	INT
unit_price	DECIMAL(10,2)

- order_detail_id: Unique identifier for each order detail entry (Primary Key).
- order_id: Order identifier that owns this detail (Foreign Key referencing Orders.order_id).
- product_id: Product identifier included in the order (Foreign Key referencing Products.product_id).
- quantity: Number of products ordered.
- unit_price: Price per unit of the product at the time the order was placed.

DATABASE SCHEMA

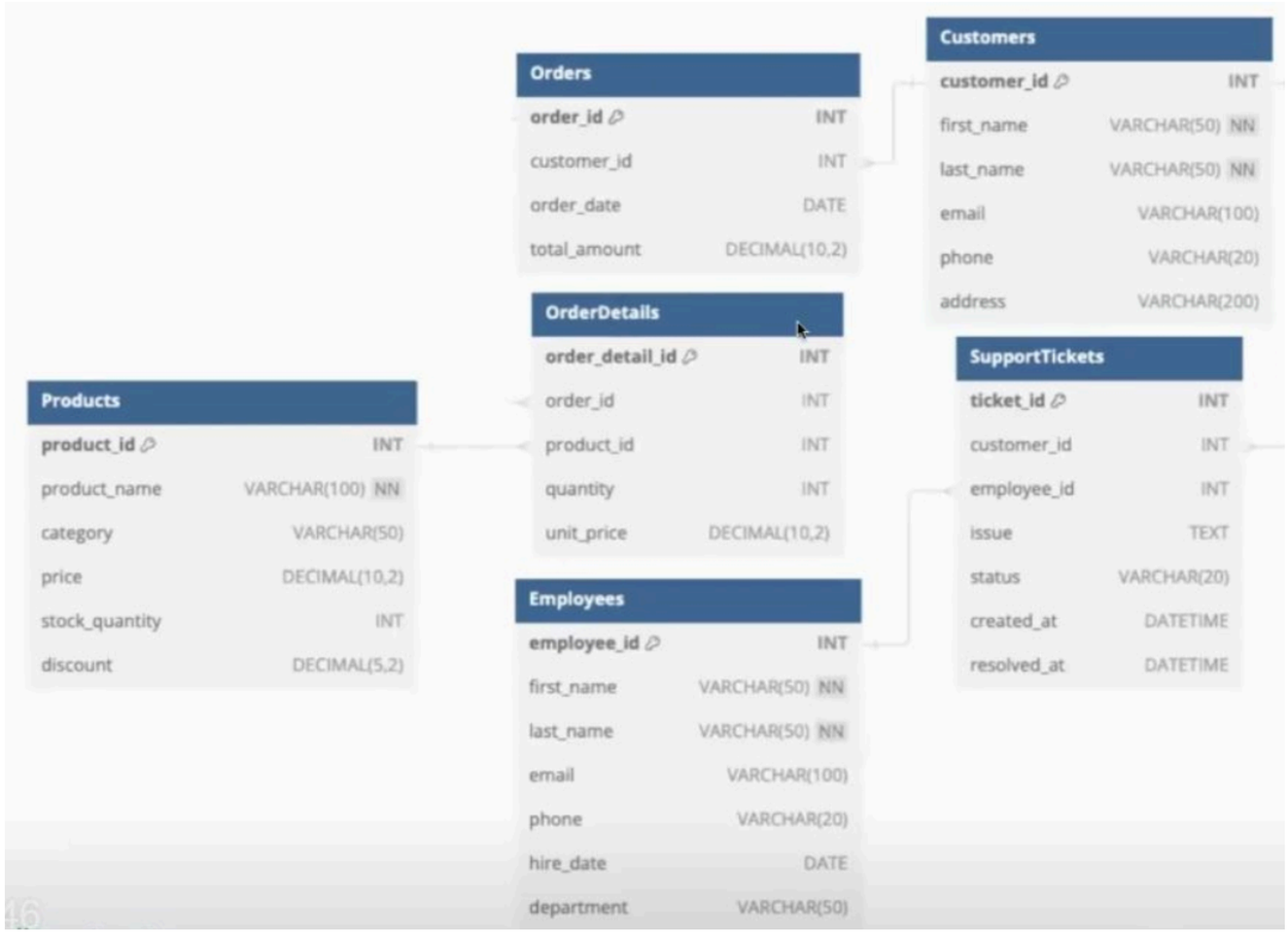
5. Employees Table

- employee_id: Unique identifier for each employee (Primary Key).
- first_name: Employee's first name.
- last_name: Employee's last name.
- email: Employee's email address.
- phone: Employee's phone number.
- hire_date: Date when the employee was hired.
- department: Department where the employee works (e.g., Support, Sales, Development)

6. Support Tickets Table

- ticket_id: Unique identifier for each support ticket (Primary Key).
- customer_id: Customer identifier who created the ticket (Foreign Key referencing Customers.customer_id).
- employee_id: Employee identifier assigned to the ticket (Foreign Key referencing Employees.employee_id).
- issue: Description of the issue reported by the customer.
- status: Current status of the ticket (e.g., open, resolved).
- created_at: Date and time when the ticket was created.
- resolved_at: Date and time when the ticket was resolved (if applicable).

DATABASE SCHEMA



DATABASE SCHEMA

Database Relationships Explanation

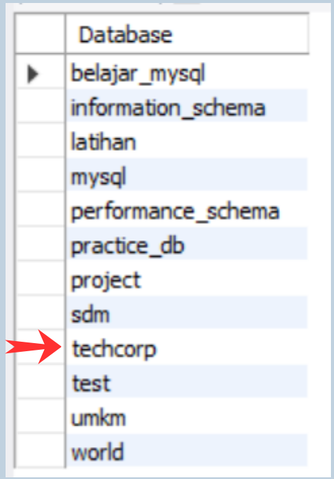
- Orders.customer_id → customers.customer_id This means the orders table can be connected to the customers table through customer_id
- orderdetails.order_id → orders.order_id This means the orderdetails table can be connected to the orders table through order_id
- orderdetails.product_id → products.product_id This means the orderdetails table can be connected to the products table through product_id
- supporttickets.employee_id → employees.employee_id This means the supporttickets table can be connected to the employees table through employee_id
- supporttickets.customer_id → customers.customer_id This means the supporttickets table can be connected to the customers table through customer_id

DATA CREATION & PREPARATION

Query

```
create database techcorp;
show databases;
```

Result



Database
belajar_mysql
information_schema
latihan
mysql
performance_schema
practice_db
project
sdm
techcorp
test
umkm
world

1. CREATE DATABASE

The query successfully creates a new database named techcorp in MySQL. The SHOW DATABASES command then displays all existing databases in the system, with the newly created techcorp database now visible in the list (highlighted with red arrow). This confirms the database has been created and is ready for table creation and data insertion.

DATA CREATION & PREPARATION

Query

```
create table Products (
    product_id int auto_increment primary key,
    product_name varchar (100) not null,
    category varchar (500) not null,
    price decimal (10,2),
    stock_quantity int
);
```

```
INSERT INTO Products (product_name, category, price, stock_quantity, discount)
VALUES
('Laptop Pro 15', 'Laptop', 1500.00, 100, 0),
('Smartphone X', 'Smartphone', 800.00, 200, 0),
('Wireless Mouse', 'Accessories', 25.00, 500, 0),
('USB-C Charger', 'Accessories', 20.00, 300, 0),
('Gaming Laptop', 'Laptop', 2000.00, 50, 10),
('Budget Smartphone', 'Smartphone', 300.00, 150, 5),
('Noise Cancelling Headphones', 'Accessories', 150.00, 120, 15),
('Wireless Earphones', 'Accessories', 100.00, 100, 10);
```

Result

product_id	product_name	category	price	stock_quantity	discount
1	Laptop Pro 15	Laptop	1500.00	100	0.00
2	Smartphone X	Smartphone	800.00	200	0.00
3	Wireless Mouse	Accessories	25.00	500	0.00
4	USB-C Charger	Accessories	20.00	300	0.00
5	Gaming Laptop	Laptop	2000.00	50	10.00
6	Budget Smartphone	Smartphone	300.00	150	5.00
7	Noise Cancelling Headphones	Accessories	150.00	120	15.00
8	Wireless Earphones	Accessories	100.00	100	10.00

2. CREATE PRODUCT TABLE

Query:
This query creates the Products table with six columns: product_id (auto-increment primary key), product_name, category, price, stock_quantity, and discount. After creating the table structure, the query inserts 8 product records across different categories (Laptop, Smartphone, Accessories) using the INSERT INTO statement with multiple value sets.

Result:
The table displays 8 products successfully inserted into the database. The data includes various electronics: 2 laptops (Laptop Pro 15, Gaming Laptop), 2 smartphones (Smartphone X, Budget Smartphone), and 4 accessories (Wireless Mouse, USB-C Charger, Noise Cancelling Headphones, Wireless Earphones). Each product has its corresponding price, stock quantity, and discount value. The product_id was automatically generated sequentially from 1 to 8, confirming the auto-increment functionality works correctly.

DATA CREATION & PREPARATION

Query

```
create table Customers (
    customer_id int auto_increment primary key,
    first_name varchar (100) not null,
    last_name varchar (100) not null,
    email varchar(100) unique,
    phone varchar(20),
    address varchar(100)
);
```

```
INSERT INTO Customers (first_name, last_name, email, phone, address)
VALUES
('John', 'Doe', 'john.doe@example.com', '123-456-7890', '123 Elm Street'),
('Jane', 'Smith', 'jane.smith@example.com', '123-456-7891', '456 Oak Street'),
('Emily', 'Johnson', 'emily.johnson@example.com', '123-456-7892', '789 Pine Street'),
('Michael', 'Brown', 'michael.brown@example.com', '123-456-7893', '101 Maple Street'),
('Sarah', 'Davis', 'sarah.davis@example.com', '123-456-7894', '202 Birch Street');
```

Result

customer_id	first_name	last_name	email	phone	address
1	John	Doe	john.doe@example.com	123-456-7890	123 Elm Street
2	Jane	Smith	jane.smith@example.com	123-456-7891	456 Oak Street
3	Emily	Johnson	emily.johnson@example.com	123-456-7892	789 Pine Street
4	Michael	Brown	michael.brown@example.com	123-456-7893	101 Maple Street
5	Sarah	Davis	sarah.davis@example.com	123-456-7894	202 Birch Street

3. CREATE CUSTOMERS TABLE

Query:

This query creates the Customers table with six columns: customer_id (auto-increment primary key), first_name, last_name, email (unique constraint), phone, and address. After establishing the table structure, the query inserts 5 customer records using the INSERT INTO statement with complete customer information including names, email addresses, phone numbers, and residential addresses.

Result:

The table displays 5 customers successfully inserted into the database. Each customer has a unique auto-generated ID (1-5) and complete contact information: John Doe, Jane Smith, Emily Johnson, Michael Brown, and Sarah Davis. All customers have valid email addresses (following the @example.com format), phone numbers in the format 123-456-XXXX, and street addresses. The unique constraint on the email field ensures no duplicate email addresses can be entered, maintaining data integrity.

DATA CREATION & PREPARATION

Query

```
create table Orders (
    order_id int auto_increment primary key,
    customer_id int,
    order_date date,
    total_amount decimal (10,2),
    foreign key (customer_id) references Customers(customer_id)
);
INSERT INTO Orders (customer_id, order_date, total_amount)
VALUES
(1, '2023-07-01', 1525.00),
(2, '2023-07-02', 820.00),
(3, '2023-07-03', 25.00),
(1, '2023-07-04', 2010.00),
(4, '2023-07-05', 300.00),
(2, '2023-07-06', 315.00),
(5, '2023-07-07', 165.00);
```

Result

order_id	customer_id	order_date	total_amount
1	1	2023-07-01	1525.00
2	2	2023-07-02	820.00
3	3	2023-07-03	25.00
4	1	2023-07-04	2010.00
5	4	2023-07-05	300.00
6	2	2023-07-06	315.00
7	5	2023-07-07	165.00

4. CREATE ORDERS TABLE

Query:

This query creates the Orders table with four columns: order_id (auto-increment primary key), customer_id (integer), order_date (date type), and total_amount (decimal with 10 digits and 2 decimal places). The customer_id is established as a foreign key referencing the Customers table to maintain relational integrity. After creating the table, the query inserts 7 order records with dates ranging from July 1-7, 2023, and varying total amounts.

Result:

The table displays 7 orders successfully inserted into the database. Each order has an auto-generated order_id (1-7), associated customer_id linking to existing customers, an order_date in July 2023, and a total_amount ranging from $25 to $2010. The foreign key relationship ensures that all customer_id values (1-5) correspond to valid customers in the Customers table, maintaining referential integrity between the two tables.

DATA CREATION & PREPARATION

Query

```
create table OrderDetails (
    order_detail_id int auto_increment primary key,
    order_id int,
    product_id int,
    quantity int,
    unit_price decimal (10,2),
    foreign key (order_id) references Orders(order_id),
    foreign key (product_id) references Products(product_id)
);
INSERT INTO OrderDetails (order_id, product_id, quantity, unit_price)
VALUES
(1, 1, 1, 1500.00),
(1, 3, 1, 25.00),
(2, 2, 1, 800.00),
(2, 4, 1, 20.00),
(3, 3, 1, 25.00),
(4, 5, 1, 2000.00),
(4, 6, 1, 10.00),
(5, 6, 1, 300.00),
(6, 6, 1, 300.00),
(7, 7, 1, 150.00),
(7, 4, 1, 15.00);
```

Result

order_detail_id	order_id	product_id	quantity	unit_price
1	1	1	1	1500.00
2	1	3	1	25.00
3	2	2	1	800.00
4	2	4	1	20.00
5	3	3	1	25.00
6	4	5	1	2000.00
7	4	6	1	10.00
8	5	6	1	300.00
9	6	6	1	300.00
10	7	7	1	150.00
11	7	4	1	15.00

5. CREATE ORDERDETAILS TABLE

Query:

This query creates the OrderDetails table with five columns: order_detail_id (auto-increment primary key), order_id (integer), product_id (integer), quantity (integer), and unit_price (decimal with 10 digits and 2 decimal places). Two foreign key constraints are established: order_id references the Orders table and product_id references the Products table, ensuring relational integrity. The query then inserts 11 order detail records, each representing individual items within specific orders.

Result:

The table shows 11 order detail records successfully inserted, with auto-generated order_detail_id values from 1 to 11. Each record references valid order_id (1–7) and product_id (1–7), all with a quantity of 1 and unit prices ranging from $10 to $2000. The foreign key constraints maintain data integrity across related tables and enable tracking multiple products within a single order, such as orders 1, 2, 4, and 7.

DATA CREATION & PREPARATION

Query

```
create table Employees (
    employee_id int auto_increment primary key,
    first_name varchar(100),
    last_name varchar(100),
    email varchar(100),
    phone varchar(20),
    hire_date date,
    department varchar(50)
```

```
INSERT INTO Employees (first_name, last_name, email, phone, hire_date, department)
VALUES
('Alice', 'Williams', 'alice.williams@example.com', '123-456-7895', '2022-01-15', 'Support'),
('Bob', 'Miller', 'bob.miller@example.com', '123-456-7896', '2022-02-20', 'Sales'),
('Charlie', 'Wilson', 'charlie.wilson@example.com', '123-456-7897', '2022-03-25', 'Development'),
('David', 'Moore', 'david.moore@example.com', '123-456-7898', '2022-04-30', 'Support'),
('Eve', 'Taylor', 'eve.taylor@example.com', '123-456-7899', '2022-05-10', 'Sales');
```

Result

employee_id	first_name	last_name	email	phone	hire_date	department
1	Alice	Williams	alice.williams@example.com	123-456-7895	2022-01-15	Support
2	Bob	Miller	bob.miller@example.com	123-456-7896	2022-02-20	Sales
3	Charlie	Wilson	charlie.wilson@example.com	123-456-7897	2022-03-25	Development
4	David	Moore	david.moore@example.com	123-456-7898	2022-04-30	Support
5	Eve	Taylor	eve.taylor@example.com	123-456-7899	2022-05-10	Sales

6. CREATE EMPLOYEES TABLE

Query:

CREATE TABLE statement defines an "Employees" table with 7 columns: employee_id (auto-incrementing primary key), first_name, last_name, email, phone (all varchar with specified lengths), hire_date (date type), and department. Second, an INSERT INTO statement adds 5 employee records with their respective data - Alice Williams, Bob Miller, Charlie Wilson, David Moore, and Eve Taylor - each with complete information including contact details, hire dates ranging from January to May 2022, and department assignments (Support, Sales, or Development).

Result:

The result section displays a database table view showing the successful execution of the query. All 5 employees are listed in rows with their employee_id numbered sequentially from 1 to 5. Each record shows complete data across all columns: names, email addresses in the format name.surname@example.com, phone numbers (123-456-789X), hire dates in YYYY-MM-DD format, and their assigned departments.

DATA CREATION & PREPARATION

Query

```sql
create table SupportTickets(
    ticket_id int auto_increment primary key,
    customer_id int,
    employee_id int,
    issue text,
    status varchar(20),
    created_at datetime,
    resolved_at datetime,
    foreign key (customer_id) references Customers(customer_id),
    foreign key (employee_id) references Employees(employee_id)
);
INSERT INTO SupportTickets (customer_id, employee_id, issue, status, created_at, resolved_at)
VALUES
(1, 1, 'Cannot connect to Wi-Fi', 'resolved', '2023-07-01 10:00:00', '2023-07-01 11:00:00'),
(2, 1, 'Screen flickering', 'resolved', '2023-07-02 12:00:00', '2023-07-02 13:00:00'),
(3, 1, 'Battery drains quickly', 'open', '2023-07-03 14:00:00', NULL),
(4, 2, 'Late delivery', 'resolved', '2023-07-04 15:00:00', '2023-07-04 16:00:00'),
(5, 2, 'Damaged product', 'open', '2023-07-05 17:00:00', NULL),
(1, 3, 'Software issue', 'resolved', '2023-07-06 18:00:00', '2023-07-06 19:00:00'),
(2, 3, 'Bluetooth connectivity issue', 'resolved', '2023-07-07 20:00:00', '2023-07-07 21:00:00'),
(5, 4, 'Account issue', 'open', '2023-07-08 22:00:00', NULL),
(3, 4, 'Payment issue', 'resolved', '2023-07-09 23:00:00', '2023-07-09 23:30:00'),
(4, 5, 'Physical damage', 'open', '2023-07-10 08:00:00', NULL),
(4, 1, 'Laptop blue screen', 'resolved', '2024-01-05 10:00:00', '2024-02-05 12:00:00'),
(5, 1, 'Laptop lagging', 'resolved', '2024-01-06 10:00:00', '2024-01-25 12:00:00'),
(3, 1, 'Some part of laptop broken', 'resolved', '2024-02-05 10:00:00', '2024-03-05 12:00:00');
```

7. CREATE SUPPORTTICKETS TABLE

Query:

This query creates the SupportTickets table with seven columns, including an auto-incrementing primary key, customer and employee references, issue descriptions, ticket status, and creation and resolution timestamps, with foreign key constraints ensuring data integrity. It then inserts 13 support ticket records covering various technical and service issues such as connectivity problems, device malfunctions, delivery issues, and account or payment errors, with data ranging from July 2023 to February 2024. Ticket statuses are marked as either resolved or open, and unresolved tickets are indicated by NULL values in the resolved_at column.

DATA CREATION & PREPARATION

Result

ticket_id	customer_id	employee_id	issue	status	created_at	resolved_at
1	1	1	Cannot connect to Wi-Fi	resolved	2023-07-01 10:00:00	2023-07-01 11:00:00
2	2	1	Screen flickering	resolved	2023-07-02 12:00:00	2023-07-02 13:00:00
3	3	1	Battery drains quickly	open	2023-07-03 14:00:00	NULL
4	4	2	Late delivery	resolved	2023-07-04 15:00:00	2023-07-04 16:00:00
5	5	2	Damaged product	open	2023-07-05 17:00:00	NULL
6	1	3	Software issue	resolved	2023-07-06 18:00:00	2023-07-06 19:00:00
7	2	3	Bluetooth connectivity issue	resolved	2023-07-07 20:00:00	2023-07-07 21:00:00
8	5	4	Account issue	open	2023-07-08 22:00:00	NULL
9	3	4	Payment issue	resolved	2023-07-09 23:00:00	2023-07-09 23:30:00
10	4	5	Physical damage	open	2023-07-10 08:00:00	NULL
11	4	1	Laptop blue screen	resolved	2024-01-05 10:00:00	2024-02-05 12:00:00
12	5	1	Laptop lagging	resolved	2024-01-06 10:00:00	2024-01-25 12:00:00
13	3	1	Some part of laptop broken	resolved	2024-02-05 10:00:00	2024-03-05 12:00:00

7. CREATE SUPPORTTICKETS TABLE

Result:

The result displays a complete table with all 13 support tickets successfully inserted. Each row shows: ticket_id (1-13), customer_id (1-5), employee_id (1-5), issue descriptions, status (resolved/open), created_at timestamps, and resolved_at timestamps. Open tickets (rows 3, 5, 8, 10) show NULL in the resolved_at column, indicating they haven't been closed yet. Resolved tickets display both creation and resolution timestamps, showing the time taken to address each issue. The data demonstrates a functional support ticket tracking system with proper referential integrity through foreign keys connecting to customer and employee records.

ANALYSIS USING SQL

Query

```
select
    c.first_name,
    c.last_name,
    sum(o.total_amount) as total_order_amount
from Customers c
join orders o on c.customer_id = o.customer_id
group by c.customer_id
order by total_order_amount desc
limit 3;
```

Result

first_name	last_name	total_order_amount
John	Doe	3535.00
Jane	Smith	1135.00
Michael	Brown	300.00

1. Top 3 customers based on total order

Query:
This query retrieves customer names along with their total order amounts by selecting first_name and last_name from the Customers table and calculating the total spending using SUM(total_amount) from the Orders table. It joins both tables through customer_id to combine customer and order data, then groups the results by customer to aggregate their purchases. The results are sorted in descending order to highlight the highest-spending customers, with the output limited to the top 3 customers.

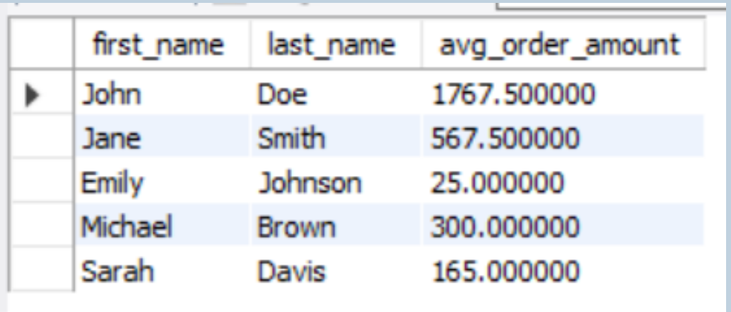
Result:
The result displays the top 3 customers ranked by their total spending. John Doe leads with $3,535 in total orders, followed by Jane Smith with $1,135, and Michael Brown with $300. This ranking clearly identifies the most valuable customers based on their purchasing behavior, which is useful for business decisions like loyalty programs, targeted marketing, or customer retention strategies.

ANALYSIS USING SQL

Query

```
select
    c.first_name,
    c.last_name,
    avg(o.total_amount) as avg_order_amount
from Customers c
join Orders o on c.customer_id=o.customer_id
group by c.customer_id;
```

Result



first_name	last_name	avg_order_amount
John	Doe	1767.500000
Jane	Smith	567.500000
Emily	Johnson	25.000000
Michael	Brown	300.000000
Sarah	Davis	165.000000

2. Average order value for each customer

Query:

This query retrieves customer names and their average order amounts by selecting first_name and last_name from the Customers table and calculating the average of total_amount from the Orders table. The Customers and Orders tables are joined using customer_id, and the results are grouped by customer to compute the average spending per order, showing how much each customer typically spends in a transaction.

Result:

The result shows five customers with different average order values. John Doe has the highest average spending at $1,767.50 per order, followed by Jane Smith at $567.50 and Michael Brown at $300. Sarah Davis and Emily Johnson have lower averages, with Emily Johnson being the lowest at $25. These results help illustrate customer spending patterns and can be used to support marketing decisions, such as targeting high-value customers or identifying opportunities for upselling.

ANALYSIS USING SQL

Query

```
select
    e.first_name,
    e.last_name,
    count(s.status) as resolved_ticket
from Employees e
join SupportTIckets s on e.employee_id=s.employee_id
where s.status = 'resolved'
group by e.employee_id
having count(s.status) > 4;
```

Result

	first_name	last_name	resolved_ticket
▶	Alice	Williams	5

3. Employee with > 4 resolved ticket support

Query:
This query retrieves employee names and the number of support tickets they have resolved by selecting data from the Employees and SupportTickets tables joined through employee_id. It counts only tickets with a resolved status using a WHERE clause, then groups the results by employee to calculate how many cases each employee has completed. The HAVING clause filters the results to display only employees who have resolved more than four tickets, helping identify the most productive support staff.

Result:
The result displays only one employee who meets the criteria. Alice Williams has resolved 5 support tickets, making her the top performer who exceeded the threshold of 4 resolved tickets. This analysis helps identify high-performing support employees, which is useful for performance evaluations, workload distribution, recognition programs, or understanding which team members are most effective at closing customer issues.

ANALYSIS USING SQL

Query

```
select
p.product_name
from products p
left join orderdetails od on p.product_id=od.order_id
where od.order_id is null;
```

Result

	product_name
▶	Wireless Earphones

4. Products that have never been ordered

Query:
This query retrieves product names that have never been ordered by selecting data from the Products table and performing a LEFT JOIN with the OrderDetails table using product_id. The WHERE clause filters rows where order_id is NULL, indicating that the product has no matching order records. As a result, the query identifies products that have never been included in any order, showing items with zero sales or no transaction history.

Result:
The result shows only one product that meets the criteria. "Wireless Earphones" appears as the only product that has never been ordered by any customer. This analysis is valuable for inventory management, helping identify slow-moving or unpopular products that may need promotional efforts, price adjustments, or potential removal from the product catalog to free up warehouse space and reduce holding costs.

ANALYSIS USING SQL

Query

```
select
sum(quantity * unit_price) as total_revenue
from orderdetails;
```

Result

	total_revenue
▶	5145.00

5. Total revenue generated from product sales

Query:
This query is used to calculate the total revenue generated from all product sales. It works by multiplying the quantity sold by the unit price for each order detail, then summing all of these values together. The final result represents the overall revenue earned from all orders in the database.

Result:
The result shows a single value of $5,145, which represents the total revenue generated from all product sales in the system. This is the cumulative amount earned from every product sold across all orders. This metric is crucial for business performance analysis, financial reporting, revenue tracking, and evaluating overall sales effectiveness. It provides a quick snapshot of the company's total sales performance and can be used for comparison against targets, previous periods, or forecasting future revenue.

ANALYSIS USING SQL

Query

```
with avg_price as (
    select
        category,
        round(avg(price),2) as avg_product
    from products
    group by category
)
select * from avg_price
where avg_product > 500;
```

Result

	category	avg_product
▶	Laptop	1750.00
	Smartphone	550.00

6. Find the average price of products for each category and find the categories with an average price > $500

Query:

This query calculates the average price of products for each category using a Common Table Expression (CTE) named avg_price. Inside the CTE, products are grouped by category, and the average price is computed and rounded to two decimal places. After that, the main query selects data from the CTE and filters the results to show only categories where the average product price is greater than $500.

Result:

The result shows two product categories that meet the condition. The Laptop category has the highest average price at $1,750, while the Smartphone category has an average price of $550. These results indicate that only these categories have relatively high-priced products on average, exceeding the $500 threshold.

ANALYSIS USING SQL

Query

```
 select *
from customers
where customer_id in
(select customer_id
from orders
where total_amount > 1000);
```

Result

customer_id	first_name	last_name	email	phone	address
1	John	Doe	john.doe@example.com	123-456-7890	123 Elm Street

7. Find customers who have made at least one order with a total amount over $1000

Query:

This query identifies customers who have placed at least one order with a total amount greater than $1000. It selects all customer data from the Customers table and uses a subquery on the Orders table to find customer_id values with high-value orders. The IN clause then filters customers whose IDs appear in those results, allowing the query to focus on customers who have made significant purchases.

Result:

The result shows one customer who meets the condition, John Doe (customer_id: 1), along with his complete contact information. This indicates that John Doe has made at least one order exceeding $1000, classifying him as a high-value customer. Such insights can be useful for targeted marketing, VIP programs, or prioritizing customers with strong purchasing potential.

CONCLUSIONS

- A small number of customers, especially John Doe, contribute a significant portion of total revenue, showing strong revenue concentration.
- Premium product categories (Laptops and Smartphones) dominate high-value transactions, indicating where most revenue potential lies.
- Customer purchasing behavior is highly uneven, with clear gaps between high-value and low-value customers.
- Operational performance varies across employees, with Alice Williams standing out as a high-impact support contributor.

SUGGESTIONS

- Prioritize high-value customers through VIP programs, personalized offers, or dedicated support to protect key revenue sources.
- Focus sales and marketing efforts on premium categories by strengthening upselling and cross-selling strategies.
- Use targeted campaigns to increase average order value among mid-tier customers.
- Optimize operations by rewarding top-performing employees and reviewing underperforming products like Wireless Earphones for improvement or removal.

LET’S CONNECT

083109376438

hernandiazanua@gmail.com

HERNANDIA ZANUA LERAS

Thank You