Practice Problem

21_AIE_304

Big Data Analysis– SEM-V
Professor – SanjanaSri Mam

Submitted By: Vikhyat Bansal [CB.EN.U4AIE21076]

Practice Problem - 1

Refer to product.csv and perform the following.

1. Count the total number of transactions in the dataset.

```
select count(*) from product_tbl;
```

OR

```
select count(transaction_id) from product_tbl;
```

OR

```
select count(*) number_of_transactions from product_tbl;
```

2. Calculate the total revenue generated from all transactions.

```
select sum(price*quantity) Total_Revenue from product_tbl;
```

3. Find Product with highest price.

```
select product_name,price from product_tbl where price= (select max(price) from product_tbl);
```

4. List the distinct products that were sold in the "Electronics" category.

```
select distinct(product_name) from product_tbl where category='Electronics';
```

5. Calculate the average price of products in each category .

```
select category,avg(price) from product_tbl group by category;
```

6. Calculate the total quantity sold for each product.

```
select product_name,sum(quantity) from product_tbl group by product_name;
```

7. Calculate the revenue generated for each month.

```
select (month(transaction_date)),sum(price*quantity) from product_tbl group by month(transaction_date);
```

8. List the top 5 customers who spent the most.

```
select customer_id,sum(price*quantity) as money_spent from product_tbl group by customer_id order by money_spent desc limit 0,5;
```

9. Find the products purchased by a specific customer.

```
select product_name from product_tbl where customer_id = 1001;
```

OR

Change 1001 to any number XXXX to get a specific customer

10. List the quantity of products sold in each category.

```
select category,sum(quantity) from product_tbl group by category;
```

11. Calculate the average quantity of products sold per transaction.

```
select avg(quantity) Quantity_sold_per_transac from product_tbl;
```

OR

```
select sum(quantity)/count(transaction_id) from product_tbl;
```

12. List products that were sold at least twice along with their total quantity sold.

```
select product_name,category,quantity from product_tbl where quantity > 1;
```

OR

```
select product_name,sum(quantity) from product_tbl group by product_name having sum(quantity)>1;
```

13. Find customers who made purchases in both the "Electronics" and "Clothing" categories.

```
select customer_id from product_tbl where category = 'Electronics'
intersect
select customer_id  from product_tbl where category = 'Clothing';
```

14. List products sold on a specific transaction date along with their quantity sold.

```
select product_name,category,quantity from product_tbl where transaction_date = 20230701;
```

15. List the top N categories by total revenue generated.

```
select category,sum(quantity*price) as revenue from product_tbl group by category order by revenue desc limit 0,3;
```

Instead of 3 we can write N.

16. Find customers who have made more than one transaction along with the number of transactions.

```
select customer_id,count(transaction_id) as number_transactions from product_tbl group by customer_id having number_transactions > 1;
```

17. List products sold between a specific date range along with the quantity sold.

```
select product_name,quantity from product_tbl where transaction_date between '20230701' and '20230713';
```

where dates can be changed before and after the BETWEEN keyword

18. Calculate the total revenue generated by each customer.

```
select customer_id,sum(quantity*price) as Revenue_customerwise from product_tbl group by customer_id order by Revenue_customerwise ;
```

19. Calculate the percentage of revenue contributed by each category to the total revenue.

```
select category,(sum(quantity*price*100)/(select sum(quantity*price) from product_tbl)) as percentage_contri from product_tbl group by category;
```

20. Calculate the total quantity sold and total revenue for products in the low, medium, and high price ranges.

```
select sum(quantity) as Total_quantity,sum(quantity*price) as Total_revenue, case when price between 0 and 100 then 'Low Range' when price between 99 and 300 then 'Medium Range' else 'High Range' end as Price_ranges from product_tbl group by Price_Ranges;
```

{For getting the total quantity sold and total revenue from sold product in a price range}

OR

```
select price,sum(quantity) as Total_quantity,sum(quantity*price) as Total_revenue, case when price between 0 and 100 then 'Low Range' when price between 99 and 300 then 'Medium Range' else 'High Range' end as Price_ranges from product_tbl group by price order by price;
```

{For getting each product price tag in a range with quantity and revenue earned from it.}

21. Count the number of products in each category from the products table

```
select category,count(product_id) as Number_of_products from product_tbl group by category;
```

22. Calculate the total sales for each product category from the sales table.

select category,sum(quantity) as Number_of_products from product_tbl group by category;

23. Retrieve products from the products table whose names contain the word 'Laptop'.

select product_id,category,price from product_tbl where product_name like %Laptop%;

OR

select * from product_tbl where product_name like %Laptop%;

(For all the details regarding the product containing name Laptop).

24. Retrieve orders from the orders table placed between January 1, 2022, and December 31, 2022.

select * from product_tbl where transaction_date between '20220101' and '20221231';

Refer to employee2.csv and perform the following.

1. Calculate the average salary of employees in each department with a salary greater than $40,000 from the employees table.

select dpt_name,avg(salary) from emp_tbl where salary>40000 group by dpt_name;

2. Retrieve employees from the employees table who have a salary greater than the average salary.

select emp_id,concat(emp_fname,' ',emp_lname) as emp_name from emp_tbl where salary>(select avg(salary) from emp_tbl);

3. Retrieve employees from the employees table with a salary greater than $60,000 and job title is 'Manager'.

select emp_id from emp_tbl where salary>50000 intersect select emp_id from emp_tbl where job_desc like '%Manager%' group by emp_id;

4. Retrieve employees from the employees table whose last name starts with 'S'.

select * from emp_tbl where emp_lname like 'S%';

5. Update the salary of an employee with employee_id 101 to $55,000 in the employees table.

```
update emp_tbl set salary=55000 where emp_id = 101;
```

6. Retrieve employees from the employees table who are in the 'Sales' department and have a salary between $40,000 and $50,000.

```
select * from emp_tbl where dpt_name = 'Sales' and salary between 40000 and 50000;
```

7. Retrieve employees from the employees table who have not been assigned to any department (department is NULL).

```
select * from emp_tbl where dpt_name = '';
```

8. Retrieve employees' full names and a calculated column for their annual bonus (10% of salary) from the employees table.

```
select emp_id,concat(emp_fname, ' ',emp_lname) as emp_fullname,(salary + ((salary*10)/100))
as bonus from emp_tbl;
```

9. Retrieve distinct job titles from the employees table.

```
select distinct(dpt_name) from emp_tbl;
```

10. Retrieve employees from the employees table who are in the 'Sales' or 'Marketing' departments.

```
select * from emp_tbl where dpt_name = 'Sales'
union
select * from emp_tbl where dpt_name = 'Marketing';
```

11. Retrieve employees from the employees table who are in the 'Sales' and 'Marketing' departments.

```
select emp_id,concat(emp_fname,' ',emp_lname) as emp_fullname from emp_tbl where
dpt_name = 'Marketing' intersect  select emp_id,concat(emp_fname,' ',emp_lname) as
emp_fullname from emp_tbl where dpt_name = 'Sales';
```

12. Retrieve employees from the employees table who are not in the 'Sales' department.

```
select * from emp_tbl where dpt_name != 'Sales';
```

13. Retrieve employees from the employees table, ordered first by department in ascending order, and then by salary in descending order.

```
select * from emp_tbl order by dpt_name,salary desc;
```

14. Retrieve employees' full names and a column indicating whether their salary is above $60,000 in the employees table.

```
select emp_id,concat(emp_fname,' ',emp_lname) as emp_fullname,salary, case  when salary>60000 then 'Yes!!Salary is above threshold' else 'No!Salary is less' end as 'Threshold' from emp_tbl;
```

15. You want to return values in multiple columns as one column. For example, you would like to produce this result set from a query against the EMP table:
CLARK WORKS IN IT
KING WORKS in HR

```
select concat(emp_fname,' ',emp_lname,' ','works in',' ',dpt_name) as emp_details from emp_tbl;
```

16. Get the random records from but limit the size to .

```
Select * from emp_tbl order by rand() limit 5;
```

Inside rand parenthesis goes the seed for the rand function.

17. Return employee names and departments from the table employee and sort by the last two characters in the name field.

```
select concat(emp_fname,' ',emp_lname) as emp_fullname,dpt_name from emp_tbl order by substring(emp_fullname,-2);
```

OR

```
select concat(emp_fname,' ',emp_lname) as emp_fullname,dpt_name from emp_tbl order by right(emp_fullname,2);
```

18. Display the Full Name of the employee whose salary is maximum.

```
select concat(emp_fname,' ',emp_lname) as emp_fullname,salary from emp_tbl order by salary desc limit 0,1;
```

OR

```
select concat(emp_fname,' ',emp_lname) as emp_fullname,salary from emp_tbl where salary = (select max(salary) from emp_tbl);
```

19. Select an attribute of your choice and make a constraint in table if a value is missing by default, the attribute should be filled with a default value and not NULL.

alter table emp_tbl alter salary set default 30000;
insert into emp_tbl values(91,'lol','hmm','IT',1,2,default,23);

Remember there is a diff between missing value and NULL value.

20. Get the products with invalid product_id.

Select * from product_tbl where product_id like "[0-9]…."

Whatever type of pattern matching you want to do.

Practice Problem – 2

1. Create table that establish many to many relationship and explain the same.

```
CREATE TABLE Employee (
EmployeeID INT NOT NULL AUTO_INCREMENT,
EmployeeName VARCHAR(100) NOT NULL,
PRIMARY KEY (EmployeeID)
);
CREATE TABLE SkillDescription (
SkillID INT NOT NULL AUTO_INCREMENT,
SkillName VARCHAR(100) NOT NULL,
PRIMARY KEY (SkillID)
);
CREATE TABLE EmployeeSkill (
EmployeeID INT NOT NULL,
SkillID INT NOT NULL,
PRIMARY KEY (EmployeeID, SkillID),
FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID),
FOREIGN KEY (SkillID) REFERENCES SkillDescription(SkillID)
);
```

2. You are working for an online bookstore, and your manager has asked you to create a report that displays the discount amount for each book in the inventory based on its price. Books priced differently will have different discount rates applied to them.
Here are the requirements:
If the book's price is less than $20, apply a 10% discount.
If the book's price is between $20 and $50 (inclusive), apply a 20% discount.
If the book's price is greater than $50, apply a 30% discount.
Write an SQL query to generate a report that includes the book's title, original price, and the discounted price based on the criteria above.

```
SELECT title, price, CASE WHEN price < 20 THEN price * 0.9 WHEN price >= 20 AND price <= 50 THEN price * 0.8 ELSE price * 0.7 END AS discounted_price FROM books;
```

3. You are managing a student course registration system. Create a table called **`Registrations`** with the following attributes:
`student_id (not unique, same id given to students of different semester,year and section)`
`course_id`
`semester`
`year`
`section`
Perform the following

•Identify the super key(s) for the Registrations table.
Super key is a set of attributes that uniquely identifies each row in a table. In this case, the super key is the combination of student_id, course_id, semester, year, and section.

•Determine the candidate key(s) among the attributes.
Candidate key is a minimal set of attributes that can uniquely identify each row in a table. In this case, the candidate key is the combination of student_id, course_id, semester, year, and section.

•Specify which attribute(s) serve as the primary key.
Primary key is a candidate key that is chosen to uniquely identify each row in a table. In this case, the primary key is the combination of student_id, course_id, semester, year, and section.

• Explain the rationale behind your choices for candidate and primary keys.
The combination of student_id, course_id, semester, year, and section is the primary key because it uniquely identifies each row in the table. The student_id alone is not sufficient to uniquely identify each row because the same student can register for the same course in different semesters, years, and sections.

4. The client wants you to create an efficient registration database. The client has come up with attributes, student_id, semester, section,course,instructor name, department, first name, last name.
Hint: Perform normalisation over this, find keys and create table . Also explain your choice.

Let us assume that Primary Key for the above data is only student_id taking that student_id is unique for each student.

Now performing normalization on the above data, we get the following tables:
From the above given attributes we knew that if the data is already in 1 NF, then we will look for 2 NF where attributes like instructor name,department,course are not dependent on primary key, so that is why we create separate table such as Course Table,Instructor Table,Registration Table, then finally if there exists some kind of transitive dependency we will look for it and if it doesn't then we have our attributes in 3NF.

Student Table: student_id, first_name, last_name

Primary Key of Student Table: student_id

Foreign Key of Student Table: student_id in Registration Table

Course Table: course_id, course_name, department

Primary Key of Course Table: course_id

Foreign Key of Course Table: course_id in Registration Table

Instructor Table: instructor_id, instructor_name, department

Primary Key of Instructor Table: instructor_id

Foreign Key of Instructor Table: instructor_id in Registration Table

Registration Table: student_id, course_id,instructor_id

Primary Key of Registration Table: student_id

Foreign Key of Registration Table: student_id in Student Table, course_id in Course Table, instructor_id in Instructor Table

Student Class Detail Table: student_id, semester, section

Primary Key of Student Class Detail Table: student_id, semester, section

Foreign Key of Student Class Detail Table: student_id in Student Table

5. Consider the following table representing a company's employee information:
EmployeeID –unique to each employee, EmployeeName Department ManagerID, ManagerName (manager is also one of the employee) Salary HireDate.

Normalize the table to at least 3NF. Pay special attention to the hierarchical relationship between employees and managers.

Employee Table: EmployeeID, EmployeeName, Department, ManagerID, ManagerName,Salary, HireDate

In the above table the primary key is EmployeeID. The ManagerID is the foreign key referencing the EmployeeID in the same table. The ManagerName is the derived attribute.

The above table is in 1NF because all the attributes are atomic.

The above table is not in 2 NF because manager name is a derived attribute and it is dependent on the ManagerID. So, we need to remove the ManagerName attribute from the above table and create a new table.

Employee Table: EmployeeID, EmployeeName, Department, ManagerID, Salary, HireDate

Manager Table: ManagerID, ManagerName

The above tables are in 2NF because the ManagerName is dependent on the ManagerID and the ManagerID is the primary key of the Manager Table.

The above tables are not in 3NF because the Department attribute is dependent on the EmployeeID. So, we need to remove the Department attribute from the Employee Table and create a new table.

Employee Table: EmployeeID, EmployeeName, ManagerID, Salary, HireDate

Manager Table: ManagerID, ManagerName

Department Table: EmployeeID, Department

The above tables are in 3NF because the Department attribute is dependent on the EmployeeID and the EmployeeID is the primary key of the Employee Table.

6. Imagine a university database that keeps track of courses, instructors, and student enrollments. The original table is as follows:
CREATE TABLE University (CourseID INT, CourseName VARCHAR(50), InstructorID INT, InstructorName VARCHAR(50), StudentID INT, StudentName VARCHAR(50), Grade CHAR(1), PRIMARY KEY (CourseID, InstructorID, StudentID));
Normalize the table to at least 3NF.

The primary key of the above table is (CourseID, InstructorID, StudentID).

The above tables are in 1NF because all the attributes are atomic.

The above tables are not in 2NF because the Grade attribute is dependent on the StudentID. So, we need to remove the Grade attribute from the above table and create a new table.

University Table: CourseID, CourseName, InstructorID, InstructorName, StudentID, StudentName

Grade Table: StudentID, Grade

The above tables are not in 3NF because the InstructorName attribute is dependent on the InstructorID. So, we need to remove the InstructorName attribute from the University Table and create a new table.

University Table: CourseID, CourseName, InstructorID, StudentID, StudentName

Instructor Table: InstructorID, InstructorName

Grade Table: StudentID, Grade

CREATE TABLE Instructor (InstructorID INT, InstructorName VARCHAR(50), PRIMARY KEY (InstructorID));

CREATE TABLE Grade (StudentID INT, Grade CHAR(1), PRIMARY KEY (StudentID));

CREATE TABLE University (CourseID INT, CourseName VARCHAR(50), InstructorID INT, StudentID INT, StudentName VARCHAR(50), PRIMARY KEY (CourseID, InstructorID, StudentID), FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID), FOREIGN KEY (StudentID) REFERENCES Grade(StudentID));

7. You have a table representing library transactions:

```
CREATE TABLE LibraryTransactions (
TransactionID INT PRIMARY KEY,
BookID INT,
BookTitle VARCHAR(100), AuthorName VARCHAR(50),
MemberID INT,
MemberName VARCHAR(50),
CheckOutDate DATE,
ReturnDate DATE );
```

Check whether the table is normalized. If not, normalize the table to at least 3NF.

The primary key of the above table is TransactionID.

The above table is in 1NF because all the attributes are atomic i.e containing atmost one value.

The above table is not in 2NF because the BookTitle and AuthorName attributes are dependent on the BookID. So, we need to remove the BookTitle and AuthorName attributes from the above table and create a new table.

LibraryTransactions Table: TransactionID, BookID, MemberID, MemberName, CheckOutDate, ReturnDate
Book Table: BookID, BookTitle, AuthorName

The above tables are not in 3NF because the MemberName attribute is dependent on the MemberID. So, we need to remove the MemberName attribute from the LibraryTransactions Table and create a new table.

LibraryTransactions Table: TransactionID, BookID, MemberID, CheckOutDate, ReturnDate
Book Table: BookID, BookTitle, AuthorName
Member Table: MemberID, MemberName

Creating above table with appropriate foreign key constraints:

```
CREATE TABLE Book ( BookID INT, BookTitle VARCHAR(100), AuthorName VARCHAR(50), PRIMARY KEY (BookID) );

CREATE TABLE Member ( MemberID INT, MemberName VARCHAR(50), PRIMARY KEY (MemberID) );

CREATE TABLE LibraryTransactions ( TransactionID INT PRIMARY KEY, BookID INT, MemberID INT, CheckOutDate DATE, ReturnDate DATE, FOREIGN KEY (BookID) REFERENCES Book(BookID), FOREIGN KEY (MemberID) REFERENCES Member(MemberID) );
```

8. Create a university database with the following attribute: student_id
first_name
last_name
department_name
course_name
grade

Normalize if needed. Also, write an SQL query that retrieves a report containing the following information for each student:

- student_id
- Full name (`first_name` + " " + `last_name`)
- department_name
- The list of courses the student is enrolled in along with the corresponding grades.
- A column indicating whether the student passed or failed each course. Consider a pass if the grade is greater than or equal to 60.

The primary key of the above table is student_id.

The above table is in 1NF because all the attributes are atomic.

The above table is not in 2NF because the department_name attribute is dependent on the student_id. So, we need to remove the department_name attribute from the above table and create a new table.

Student Table: student_id, first_name, last_name
Department Table: student_id, department_name

The above tables are not in 3NF because the course_name attribute is dependent on the student_id. So, we need to remove the course_name attribute from the above table and create a new table.

Student Table: student_id, first_name, last_name
Department Table: student_id, department_name
Course Table: student_id, course_name

The above tables are not in 3NF because the grade attribute is dependent on the student_id. So, we need to remove the grade attribute from the above table and create a new table.

Student Table: student_id, first_name, last_name
Department Table: student_id, department_name
Course Table: student_id, course_name
Grade Table: student_id, grade

Creating above table with appropriate foreign key constraints:

CREATE TABLE Student (student_id INT, first_name VARCHAR(50), last_name VARCHAR(50), PRIMARY KEY (student_id));

CREATE TABLE Department (student_id INT, department_name VARCHAR(50), FOREIGN KEY (student_id) REFERENCES Student(student_id));

CREATE TABLE Course (student_id INT, course_name VARCHAR(50), FOREIGN KEY (student_id) REFERENCES Student(student_id));

CREATE TABLE Grade (student_id INT, grade CHAR(1), FOREIGN KEY (student_id) REFERENCES Student(student_id));

Write an SQL query that retrieves a report containing the following information for each student:

- student_id

SELECT student_id FROM Student;

- Full name (first_name + " " + last_name)

SELECT CONCAT(first_name, " ", last_name) AS full_name FROM Student;

- department_name

SELECT department_name FROM Department;

- The list of courses the student is enrolled in along with the corresponding grades.

SELECT course_name, grade FROM Course JOIN Grade on Course.student_id = Grade.student_id;

- A column indicating whether the student passed or failed each course. Consider a pass if the grade is greater than or equal to 60.

SELECT course_name, CASE WHEN grade >= 60 THEN "Pass" ELSE "Fail" END AS pass_fail FROM Course JOIN Grade on Course.student_id = Grade.student_id;

9. Create a retail database with the following tables: customer_name
email
product_name
price
order_date
quantity
discount_percent

Normalize if needed. Also, write an SQL query that calculates the total amount spent by each customer in terms of the total order value after applying discounts. Assume that the discount is in percentage and is applied to each product in an order. The query should include:

- customer_id
- customer_name
- Total amount spent by the customer (considering discounts)

The primary key of the above table is a new attribute called customer_id.

The above table is in 1NF because all the attributes are atomic.

The above table is not in 2NF because the customer_name attribute is dependent on the customer_id. So, we need to remove the customer_name attribute from the above table and create a new table.

Customer Table: customer_id, customer_name, email
Order Table: customer_id, product_name, price, order_date, quantity, discount_percent

The above tables are not in 3NF because the customer_name attribute is dependent on the customer_id. So, we need to remove the customer_name attribute from the Customer Table and create a new table.

Customer Table: customer_id,customer_name, email
Order Table: customer_id, product_id, order_date, quantity,
Product Table: product_id, product_name, price, discount_percent

The above tables are in 3NF because the customer_name attribute is dependent on the customer_id. So, we need to remove the customer_name attribute from the Customer Table and create a new table.

Creating above table with appropriate foreign key constraints:

CREATE TABLE Customer (customer_id INT, customer_name VARCHAR(50), email VARCHAR(50), PRIMARY KEY (customer_id));

CREATE TABLE product (product_id INT, product_name VARCHAR(50), price INT, discount_percent INT, PRIMARY KEY (product_id));

CREATE TABLE order (customer_id INT, product_id INT, order_date DATE, quantity INT, FOREIGN KEY (customer_id) REFERENCES customer(customer_id), FOREIGN KEY (product_id) REFERENCES product(product_id));

Write an SQL query that calculates the total amount spent by each customer in terms of the total order value after applying discounts. Assume that the discount is in percentage and is applied to each product in an order. The query should include:

- customer_id
- customer_name
- Total amount spent by the customer (considering discounts)

SELECT customer_id, customer_name, SUM(price * quantity * (1 - discount_percent / 100)) AS total_amount_spent FROM Customer JOIN on Customer.customer_id = Order.customer_id GROUP BY customer_id;

10. Consider a weather database with the following tables: Cities:
city_id (Primary Key)
city_name
country
TemperatureReadings:
reading_id (Primary Key)
city_id (Foreign Key)
temperature
reading_date

Write an SQL query to find the average temperature for each city in the last week. Include the following information in the result:

- city_name
- country
- Average temperature for the last week

Creating above table with appropriate foreign key constraints:

CREATE TABLE Cities (city_id INT, city_name VARCHAR(50), country VARCHAR(50), PRIMARY KEY (city_id));

CREATE TABLE TemperatureReadings (reading_id INT, city_id INT, temperature INT, reading_date DATE, FOREIGN KEY (city_id) REFERENCES Cities(city_id));

Write an SQL query to find the average temperature for each city in the last week. Include the following information in the result:

- ***city_name***
- ***country***
- ***Average temperature for the last week***

SELECT city_name, country, AVG(temperature) AS average_temperature FROM Cities JOIN TemperatureReadings on Cities.city_id = TemperatureReadings.city_id WHERE reading_date >= DATE_SUB(CURDATE(), INTERVAL 7 DAY) GROUP BY city_name, country;

Practice Problem – 3

1. A) Extract the domain from the page_url.

```
val data = Seq( (1, "https://example.com/spark/page1"),
(2, "https://example.com/spark/page2"),
(1, https://example.com/spark/page3),
(3, "https://example.com/hadoop/page1"),
(2, https://example.com/spark/page4),
(3, "https://example.com/spark/page5"),
(1, https://anotherdomain.com/page6),
(2, "https://anotherdomain.com/page7") )
// Create dataframe from the data
val df = data.toDF("user_id", "page_url")

// Extract the domain from the page_url.
// Hint: Use the regexp_extract function.
val df2 = df.withColumn("domain", regexp_extract($"page_url", "https?://([^/]+).*", 1));
```

B) Count the number of visits for each user on each domain.

```
val df3 = df2.groupBy("user_id", "domain").count()
```

C) Find the top domain for each user based on the number of visits.

```
// Without using row_number(),PartitionBy().
val df4 = df3.groupBy("user_id").agg(max(struct($"count",
$"domain")).as("max")).select($"user_id", $"max.domain", $"max.count")
// Using Window Partition By
import org.apache.spark.sql.expressions.Window
val df4 = df3.withColumn("rank",
row_number().over(Window.partitionBy("user_id").orderBy($"count".desc)))
```

2. Suppose you have a CSV file named **`shopping_data.csv`** with the following attributes: **`customer_id,product_id,quantity,price.`** Compute the below

Total Spending per Customer: Calculate the total spending for each customer and display the result.

```
// Create a DataFrame from the CSV file.
val df = spark.read.format("csv").option("header", "true").load("shopping_data.csv")
; Total Spending per Customer: Calculate the total spending for each customer and display the result.
val df2 = df.groupBy("customer_id").agg(sum($"quantity" * $"price").as("total_spending"))
```

Most Purchased Product: Identify the product that has been purchased the most and display its details.

```
val df3 = df.groupBy("product_id").agg(sum($"quantity").as("total_quantity"))
```

Average Price per Product: Calculate the average price for each product and display the result.

```
val df4 = df.groupBy("product_id").agg((sum($"price")/sum($"quantity")).as("avg_price"));
```

3. Suppose you have a DataFrame containing information about employees, and you want to add a new column called "performance_category" based on the "performance_score" column. The categorization should be as follows:

; 1. If the performance_score is greater than or equal to 90, the performance_category is "Excellent".
; 2. If the performance_score is greater than or equal to 80 and less than 90, the performance_category is "Good".
; 3. If the performance_score is greater than or equal to 70 and less than 80, the performance_category is "Average".
; 4. If the performance_score is greater than or equal to 60 and less than 70, the performance_category is "Poor".

```
val df = data.toDF("employee_id", "employee_name", "performance_score")

val df2 = df.withColumn("performance_category", when($"performance_score" >= 90, "Excellent").when($"performance_score" >= 80, "Good").when($"performance_score" >= 70, "Average").when($"performance_score" >= 60, "Poor").otherwise("Poor"))
```

4. You have three datasets: employees, departments, and projects. The employees dataset contains information about employees, the departments dataset contains information about departments, and the projects dataset contains information about projects assigned to employees. You need to perform the following tasks:

   Create a DataFrame from each dataset.

```
val employees = spark.read.format("csv").option("header",
"true").load("employees.csv")
val departments = spark.read.format("csv").option("header",
"true").load("departments.csv")
val projects = spark.read.format("csv").option("header",
"true").load("projects.csv")
```

   Task 1: Inner Join - Employee and Department
   Join the employees and departments datasets using an inner join based on the department_id column.

```
val df1 = employees.join(departments, employees("department_id") ===
departments("department_id"), "inner")
```

   Task 2: Left Join - Employee and Project
   Join the employees and projects datasets using a left join based on the employee_id column.

```
val df2 = employees.join(projects, employees("employee_id") ===
projects("employee_id"), "left")
```

   Task 3: Right Join - Project and Employee
   Join the projects and employees datasets using a right join based on the employee_id column.

```
val df3 = projects.join(employees, projects("employee_id") ===
employees("employee_id"), "right")
```

   Task 4: Full Outer Join - Employee,Department and Project
   Join the employees, departments, and projects datasets using a full outer join based on common columns.

```
val df4 = employees.join(departments, employees("department_id") ===
departments("department_id"), "fullouter").join(projects,
employees("employee_id") === projects("employee_id"), "fullouter")
```

5. Suppose you have a dataset of marketing campaign results with columns like "campaign_id," "conversion_rate," and "cost_per_conversion." The goal is to analyze the effectiveness of each campaign and calculate the overall marketing ROI.

```
; The sample data is
; val data = Seq( ("campaign_1", 0.1, 50.0), ("campaign_2", 0.15, 60.0), ("campaign_3", 0.12,
55.0), ("campaign_4", 0.2, 70.0), ("campaign_5", 0.18, 65.0) )

; Create dataframe from the data
val df = data.toDF("campaign_id", "conversion_rate", "cost_per_conversion")
```

; Calculate the total number of conversions for each campaign.
val df2 = df.withColumn("campaign_roi", $"conversion_rate" * $"cost_per_conversion")

6. Suppose you have a dataset of travel bookings with columns like "booking_id," "destination," and "travel_date." The goal is to identify popular travel destinations and analyze booking trends.

The sample data is
val data = Seq(("booking_1", "City A", "2023-01-01"), ("booking_2", "City B", "2023-01-02"), ("booking_3", "City A", "2023-01-03"), ("booking_4", "City C", "2023-01-04"), ("booking_5", "City B", "2023-01-05"))

; Create dataframe from the data
val df = data.toDF("booking_id", "destination", "travel_date")

; Find the top 3 most popular destinations.
val df2 = df.groupBy("destination").count().orderBy($"count".desc)

; Find the number of bookings for each destination by month.
val df3 = df.withColumn("month", month($"travel_date")).groupBy("destination", "month").count().orderBy($"month".asc, $"count".desc).

Practice Problem – 4

1. Use sales_data.csv for the following questions

   Get the most popular day/s of the week

```
val df = spark.read.format("csv").option("header", "true").load("sales_data.csv")

df.withColumn("day_of_week",dayofweek($"timestamp")).groupBy("day_of_week").count().orderBy($"count".desc).show()
```

   Get the most popular day of the month.

```
df.withColumn("day_of_month",dayofmonth($"timestamp")).groupBy("day_of_month").count().orderBy($"count".desc).show()
```

   Get the cummulative sales of each customer.

```
df.groupBy("customer_id").agg(sum($"quantity"*$"price").as("total_sales")).orderBy($"total_sales".desc).show()
```

Get the the total quantity sold for each product and rank them.

```
val df2 = df.groupBy("product_id").agg(sum($"quantity").as("total_quantity")).orderBy($"total_quantity".desc)

df2.withColumn("rank", rank().over(Window.orderBy($"total_quantity".desc))).show()
```

Get the purchase frequency for each customer and rank them

```
val df2 = df.groupBy("customer_id").agg(count($"customer_id").as("purchase_frequency")).orderBy($"purchase_frequency".desc)

df2.withColumn("rank", rank().over(Window.orderBy($"purchase_frequency".desc))).show()
```

Get the rolling average of sales quantity for each product

```
df.withColumn("rolling_average",
avg($"quantity").over(Window.partitionBy("product_id").orderBy("timestamp"))).sho
w()
```

Get the running total for each customer's purchase amount.

```
df.withColumn("running_total", sum($"quantity" *
$"price").over(Window.partitionBy("customer_id").orderBy("timestamp"))).show()
```

Estimate the age of each customer based on their first purchase date.

CANNOT BE DONE USING GIVEN DATA AS NO SPECIFIC TABLE ABOUT

```
val df2 = df.groupBy("customer_id").agg(min($"timestamp").as("first_purchase_date"))

df2.withColumn("estimated_age", datediff(current_date(),
$"first_purchase_date")/365).show()
```

Identify the gap in days between the availability of a product and its purchase.

CANNOT BE DONE USING GIVEN DATA AS NO SPECIFIC TABLE ABOUT PRODUCT IS GIVEN.

Find customers who made a purchase in the last 3 months.

```
df.filter(months_between(current_date(), $"timestamp") <= 3).show()
```

Calculate the number of months since each customer's first purchase.

```
df.groupBy("customer_id").agg(min($"timestamp").as("first_purchase_date")).withColu
mn("months_since_first_purchase", months_between(current_date(),
$"first_purchase_date")).show()
```

Detect seasonal patterns in customer purchases, such as spikes during holidays or specific seasons.

```
df.groupBy(month($"timestamp").as("month")).agg(sum($"quantity" *
$"price").as("total_sales")).orderBy($"total_sales".desc).show()
```

Now from above data, we can calculate percent change or variance by subtracting each month expenditure with max expenditure.

Identify High-Value Customers with Weekly Ranks: Rank customers based on their weekly total sales within each customer partition. This can help you identify high-value customers on a weekly basis.

```
val df2 = df.withColumn("week", weekofyear($"timestamp")).groupBy("customer_id", "week").agg(sum($"quantity" * $"price").as("total_sales")).orderBy($"customer_id".asc, $"week".asc)

df2.withColumn("rank", rank().over(Window.partitionBy("customer_id").orderBy($"total_sales".desc))).show()
```

Remember the partition is done based on week of year and accordingly the ranks were given.

2. Use weather_data.csv to achieve following

Create dataframe using weather_data.csv

```
val df = spark.read.format("csv").option("header", "true").load("/home/leagueflyer/Downloads/weather_data.csv")
```

Give query Convert the timezone to american timezone

```
df.withColumn("timestamp", from_utc_timestamp($"timestamp", "America/Los_Angeles")).show()
```

Get to know the current timezone and convert the timestamp column to standard time (UTC timezone).

To get the current time zone,

```
val current_timezone = spark.conf.get("spark.sql.session.timeZone")
```

Then use withColumn function to convert the timestamp column to UTC timezone

```
df.withColumn("timestamp", from_utc_timestamp($"timestamp", "UTC")).show()
```

Practice Problem - 5

mongoimport --type csv -d test -c sales_data --headerline --drop sales_data.csv

- Give mongodb query to get Average Quantity Sold per Customer

```
db.sales_data.aggregate([{$group: {_id: "$customer_id", avg_quantity_sold: {$avg: "$quantity"}}}])
```

- Give mongodb query to get Total Sales per Product Category

```
db.sales_data.aggregate([{$group: {_id: "$product_category", total_sales: {$sum: {$multiply: ["$quantity", "$price"]}}}}])
```

- Give mongodb query to get Most Sold Product

```
db.sales_data.aggregate([{$group: {_id: "$product_id", total_quantity: {$sum: "$quantity"}}}, {$sort: {total_quantity: -1}}, {$limit: 1}])
```

- Sales Trend Over Time (weekly and Monthly basis)

```
db.sales_data.aggregate([{$group: {_id: {$week: {$toDate: "$timestamp"}}, total_sales: {$sum: {$multiply: ["$quantity", "$price"]}}}}, {$sort: {_id: 1}}])
```

After toDate replace the word “week” by word “month” to get MONTH based output.

- Average Price per Product Category

```
db.sales_data.aggregate([{$group: {_id: "$product_category", avg_price: {$avg: "$price"}}}])
```

- Monthly Sales Breakdown for a Specific Product Category

```
db.sales_data.aggregate([{$match: {product_category: "Electronics"}}, {$group: {_id: {$month: {$toDate: "$timestamp"}}, total_sales: {$sum: {$multiply: ["$quantity", "$price"]}}}}, {$sort: {_id: 1}}])
```

- Top N Customers by Total Spending

```
db.sales_data.aggregate([{$group: {_id: "$customer_id", total_spending: {$sum: {$multiply: ["$quantity", "$price"]}}}}, {$sort: {total_spending: -1}}, {$limit: 5}])
```

- Average Price Over Time for a Specific Product

```
db.sales_data.aggregate([{$match: {product_id: 101}}, {$group: {_id: {$toDate: "$timestamp"}, avg_price: {$avg: "$price"}}}, {$sort: {_id: 1}}])
```

- Sales Contribution Percentage by Product Category

```
db.sales_data.aggregate([{$group: {_id: "$product_category", total_sales: {$sum: {$multiply: ["$quantity", "$price"]}}}}, {$project: {product_category: "$_id", total_sales: 1, sales_contribution_percentage: {$multiply: [{$divide: ["$total_sales", {$sum: "$total_sales"}]}, 100]}}}, {$sort: {sales_contribution_percentage: -1}}])
```

- Give mongodb query to get Repeated customers

```
db.sales_data.aggregate([{$group: {_id: "$customer_id",
total_purchases: {$sum: 1}}}, {$match: {total_purchases: {$gt: 1}}}])
```

- Calculate Average Time Between Purchases for Each Customer

```
db.sales_data.aggregate([{$group: {_id: "$customer_id",
min_purchase_date: {$min: {$toDate: "$timestamp"}}}}, {$project:
{customer_id: "$_id", min_purchase_date: 1,
avg_time_between_purchases: {$divide: [{$subtract: [new Date(),
"$min_purchase_date"]}, 86400000]}}}, {$sort:
{avg_time_between_purchases: -1}}])
```

- Identify Peak Shopping Hours Across All Days

```
db.sales_data.aggregate([{$group: {_id: {$hour: {$toDate:
"$timestamp"}}, total_sales: {$sum: {$multiply: ["$quantity", "$price"]}}}},
{$sort: {total_sales: -1}}])
```

- to select products whose names contain either "Phone" or "Laptop" and have a price between $700 and $1500. Additionally, you want to calculate a 15% discount for products with a quantity greater than 1.

```
db.sales_data.aggregate([{$match: {$and: [{product_name: {$regex:
"Phone|Laptop"}}, {price: {$gte: 700, $lte: 1500}}]}}, {$project:
{product_name: 1, price: 1, quantity: 1, discount: {$cond: [{$gt:
["$quantity", 1]}, {$multiply: [0.15, "$price"]}, 0]}}}, {$project:
{product_name: 1, price: 1, quantity: 1, discount: 1, discounted_price:
{$subtract: ["$price", "$discount"]}}}])
```

- to calculate a discount for the selected products based on certain conditions. Let's say you want to apply a 10% discount for products with a price greater than $1000

```
db.sales_data.aggregate([{$match: {$and: [{product_name: {$regex: "Phone|Laptop"}}, {price: {$gte: 700, $lte: 1500}}]}}, {$project: {product_name: 1, price: 1, quantity: 1, discount: {$cond: [{$gt: ["$price", 1000]}, {$multiply: [0.1, "$price"]}, 0]}}}, {$project: {product_name: 1, price: 1, quantity: 1, discount: 1, discounted_price: {$subtract: ["$price", "$discount"]}}}])
```