SQL Aggregate Functions

Solve the following SQL queries using aggregate functions to calculate the following from the Employee table:
a. Find the total salary of all employees.
b. Find the average salary of employees in the IT department.
c. Find the department with the highest total salary.
d. Find the total salary paid in each department.

Here are the SQL queries using **aggregate functions** to answer each part based on an `Employee` table that includes columns like `Salary`, `Department`, etc.:

---

a. Find the total salary of all employees:

```sql
SELECT SUM(Salary) AS TotalSalary
FROM Employee;
```

---

b. Find the average salary of employees in the IT department:

```sql
SELECT AVG(Salary) AS AverageITSalary
FROM Employee
WHERE Department = 'IT';
```

---

c. Find the department with the highest total salary:

```
SELECT Department, SUM(Salary) AS TotalDeptSalary
FROM Employee
GROUP BY Department
ORDER BY TotalDeptSalary DESC
LIMIT 1;
```

---

d. Find the total salary paid in each department:

```sql
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employee
GROUP BY Department;
```

Let me know if you want to include employee names, handle nulls, or filter further!

Solve the following SQL queries using date functions to display the following:
a. Get the current system date.
b. Display employees' names and the number of years they have worked.
c. Find employees hired on a weekend (Saturday or Sunday).
d. Find the employee who has been working the longest

Here are the SQL queries using **date functions** to solve each part from the `Employee` table (assuming there are columns like `Name` and `HireDate`):

---

a. Get the current system date:

```sql
```

```sql
SELECT CURRENT_DATE AS SystemDate;
```

> In some systems, you can also use `GETDATE()` or `SYSDATE`.

---

b. Display employees' names and the number of years they have worked:

```sql
SELECT Name,
       TIMESTAMPDIFF(YEAR, HireDate, CURDATE()) AS YearsWorked
FROM Employee;
```

> If you're using **Oracle**, use:

```sql
SELECT Name,
       FLOOR(MONTHS_BETWEEN(SYSDATE, HireDate) / 12) AS YearsWorked
FROM Employee;
```

---

c. Find employees hired on a weekend (Saturday or Sunday):

```sql
SELECT Name, HireDate
FROM Employee
WHERE DAYOFWEEK(HireDate) IN (1, 7);
```

> `1` = Sunday, `7` = Saturday in MySQL.
> In **Oracle**, use:

```sql
```

```
WHERE TO_CHAR(HireDate, 'DY') IN ('SAT', 'SUN');
```

---

d. Find the employee who has been working the longest:

```sql
SELECT Name, HireDate
FROM Employee
ORDER BY HireDate
LIMIT 1;
```

> In **SQL Server**, use `TOP 1`, and in **Oracle**, use `FETCH FIRST 1 ROWS ONLY`.

Let me know your SQL dialect (MySQL, Oracle, SQL Server, etc.) if you want exact compatibility!

Solve the following SQL queries on Employee & Department Schemas

a. To Perform a LEFT JOIN between the Employee and Department tables to display: Employee name, Department name & Salary. All employees should be displayed, with NULL for those without a department.
b. To select Only employees who are assigned a department should be displayed(Inner Join)
c. Convert all employee names to uppercase
d. Find employees whose name contains ‘o’ as the second letter

Here are the SQL queries for each part based on **Employee** and **Department** tables:

---

a. **LEFT JOIN**: Show all employees with department name (NULL if no department)

```sql
SELECT E.Name AS EmployeeName,
       D.DeptName AS DepartmentName,
       E.Salary
FROM Employee E
LEFT JOIN Department D
ON E.DeptID = D.DeptID;
```

---

b. **INNER JOIN**: Show only employees who are assigned a department

```sql
SELECT E.Name AS EmployeeName,
       D.DeptName AS DepartmentName,
       E.Salary
FROM Employee E
INNER JOIN Department D
ON E.DeptID = D.DeptID;
```

---

c. Convert all employee names to uppercase

```sql
SELECT UPPER(Name) AS UpperCaseName
FROM Employee;
```

---

d. Find employees whose name contains ‘o’ as the second letter

```sql
SELECT Name
FROM Employee
WHERE Name LIKE '_o%';
```

> `_` represents one character, and `%` means any sequence of characters.

Let me know if you want to add sorting, filtering, or use aliases for better readability!

Solve the following SQL queries
a. Write a SQL query using the IN operator to Find employees who work in departments with IDs 10, 20, or 30.
b. Write a SQL query using the NOT IN operator to Find Employees Who Are NOT in HR and IT Departments.

Here are the SQL queries for your requirements using the `IN` and `NOT IN` operators:

---

a. Find employees who work in departments with IDs 10, 20, or 30:

```sql
SELECT *
FROM Employee
WHERE DeptID IN (10, 20, 30);
```

---

b. Find employees who are NOT in HR and IT departments

(Assuming you have to check based on department **names**, and the `Department` table has columns like `DeptID` and `DeptName`):

```sql
```

```sql
SELECT *
FROM Employee
WHERE DeptID NOT IN (
    SELECT DeptID
    FROM Department
    WHERE DeptName IN ('HR', 'IT')
);
```

> If `DeptName` is directly in the `Employee` table, then simply:

```sql
SELECT *
FROM Employee
WHERE DeptName NOT IN ('HR', 'IT');
```

Let me know if you'd like to see this joined with department names for clarity!

Solve the following SQL queries
a. Write a SQL query using the EXISTS operator to Find employees who are not assigned to any department.
b. Write a SQL query using the ANY operator to Find employees earning more than at least one employee in department 10.

Here are the SQL queries using the `EXISTS` and `ANY` operators:

---

a. Find employees who are not assigned to any department using `EXISTS`:

```sql
SELECT *
FROM Employee E
WHERE NOT EXISTS (
    SELECT 1
    FROM Department D
    WHERE E.DeptID = D.DeptID
);
```

> Alternatively, if you just want to check for unassigned employees (i.e., `DeptID IS NULL`), this is simpler:

```sql
SELECT *
FROM Employee
WHERE DeptID IS NULL;
```

---

b. Find employees earning more than at least one employee in department 10 using `ANY`:

```sql
SELECT *
FROM Employee
WHERE Salary > ANY (
    SELECT Salary
    FROM Employee
    WHERE DeptID = 10
);
```

> This will return employees whose salary is greater than the **lowest salary** in Dept 10.

Let me know if you want to adjust these queries to use JOINs or filter further!

Consider the following un normalized table:
StudentID StudentName Course Instructor
InstructorDept
101 Alice DBMS, OS Dr. Smith, Dr. John CS, IT

102 Bob DBMS Dr. Smith CS
103 Charlie OS Dr. John IT

a. Identify the anomalies in the table.
b. Normalize the table to 1NF, 2NF, 3NF, and BCNF by creating proper relations.
c. Explain how each normalization step resolves anomalies.

Let's walk through the normalization process for the given **unnormalized table** and answer each part clearly:

🔹 Given Unnormalized Table:

StudentID	StudentName	Course	Instructor	InstructorDept
101	Alice	DBMS, OS	Dr. Smith, Dr. John	CS, IT
102	Bob	DBMS	Dr. Smith	CS
103	Charlie	OS	Dr. John	IT

---

a. Anomalies in the Table:

1. **Insertion Anomaly**:
   - Can't insert a new instructor or course unless a student takes it.
2. **Update Anomaly**:
   - If Dr. John changes department from IT to AI, we must update all rows where he appears—risk of inconsistent data.
3. **Deletion Anomaly**:
   - If Alice is deleted, and she’s the only one taking "OS" with Dr. John, we lose all info about that course and instructor.

4. **Redundancy**:
    - Instructor names and departments repeat for each student-course combo.
5. **Multivalued Attributes**:
    - `Course`, `Instructor`, and `InstructorDept` contain multiple values in a single field—this violates **1NF**.

---

b. Normalization Steps

🔸 1NF (First Normal Form)

➡️ Eliminate multivalued attributes by creating one row per atomic value:

StudentID	StudentName	Course	Instructor	InstructorDept
101	Alice	DBMS	Dr. Smith	CS
101	Alice	OS	Dr. John	IT
102	Bob	DBMS	Dr. Smith	CS
103	Charlie	OS	Dr. John	IT

🔸 2NF (Second Normal Form)

➡️ Remove partial dependencies (i.e., non-prime attributes depending only on part of the composite key).
➡️ Assume the composite primary key is `(StudentID, Course)`.

Split into two tables:

StudentCourse Table:

StudentID	Course
101	DBMS
101	OS
102	DBMS
103	OS

Student Table:

StudentID	StudentName
101	Alice
102	Bob
103	Charlie

CourseInstructor Table:

Course	Instructor	InstructorDept
DBMS	Dr. Smith	CS
OS	Dr. John	IT

🔸 3NF (Third Normal Form)

➡ Remove transitive dependencies (non-prime attribute depending on another non-prime attribute).

Here, `InstructorDept` depends on `Instructor`, not directly on the `Course`.

Split further:

CourseInstructor Table:

Course	Instructor
DBMS	Dr. Smith
OS	Dr. John

Instructor Table:

Instructor	InstructorDept
Dr. Smith	CS
Dr. John	IT

🔸 BCNF (Boyce-Codd Normal Form)

➡ All determinants must be candidate keys.

In **3NF**, if each course has only one instructor (1:1), then **BCNF = 3NF** here.

If the same course can have multiple instructors, we'd need to consider a more complex key, but for now, **this design satisfies BCNF**.

---

c. How Each Step Resolves Anomalies:

Normal Form	Fixes These Anomalies
1NF	Removes multivalued fields (atomicity)
2NF	Removes partial dependency (avoids duplication of student names)
3NF	Removes transitive dependency (separates instructor and department info)
BCNF	Ensures all dependencies are on candidate keys (eliminates redundancy further)

Would you like these tables converted into SQL `CREATE TABLE` and `INSERT` statements too?

Given a relation R (A, B, C, D) and Functional Dependency set FD = {AB → CD, B → C}, determine whether
the given R is in 2NF? If not convert it into 2 NF.

Let's analyze whether the relation **R(A, B, C, D)** with **FD = {AB → CD, B → C}** is in **2NF**, and convert it if not.

🔹 Step 1: Check for 2NF

✅ What is 2NF?

A relation is in **2NF** if:

1. It is in **1NF** (i.e., no multi-valued or composite attributes), and
2. **No partial dependency** exists — i.e., **no non-prime attribute depends on a part of a candidate key**.

🔸 Identify candidate keys:

- Given **AB → CD** (closure of AB includes all attributes), so **AB is a candidate key**.

Now check for any partial dependencies:

- **FD: B → C**
  - 🔸 Here, `B` is **part** of candidate key `AB`, and `C` is a **non-prime** attribute.

  ✅ **This is a partial dependency**, which violates 2NF.

❌ Conclusion: R is NOT in 2NF because of partial dependency `B → C`.

🔹 Step 2: Convert R to 2NF

We need to remove the partial dependency `B → C`.

🔸 Decompose into two relations:

1. **R1(B, C)** — to capture the partial dependency `B → C`
2. **R2(A, B, D)** — remove `C` from the original relation

🔸 Final 2NF Relations:

R1(B, C)
FD: `B → C`

R2(A, B, D)
FD: `AB → D` (after removing `C`, `AB → D` still holds)

✅ Now, both **R1** and **R2** are in **2NF** — no partial dependencies remain.

Let me know if you'd like to check 3NF or write SQL `CREATE TABLE` statements for this!

Assume a scenario where salary updates are being made in the Employee table. Solve the following SQL Queries:
a. BEGIN TRANSACTION to start a transaction.
b. SAVEPOINT to mark a point in the transaction.
c. ROLLBACK to undo changes in case of an error.
d. COMMIT to finalize the changes.

Here’s how you can use **transaction control** in SQL for salary updates in the `Employee` table:

🔹 a. Start the transaction

sql

```sql
BEGIN TRANSACTION;
-- or simply:
START TRANSACTION;
```

🔹 b. Set a savepoint

sql

```sql
SAVEPOINT BeforeSalaryUpdate;
```

🔹 c. Rollback to savepoint (if needed)

sql

```sql
ROLLBACK TO BeforeSalaryUpdate;
```

> This undoes changes **after the savepoint**, but keeps the transaction open.

🔹 d. Commit the transaction

sql

```sql
COMMIT;
```

> This **finalizes** all changes made since the transaction began.

✅ Example Scenario

```sql
BEGIN TRANSACTION;

UPDATE Employee
SET Salary = Salary + 1000
WHERE DeptID = 10;

SAVEPOINT BeforeSalaryUpdate;

-- Suppose an error occurs in the next update:
UPDATE Employee
SET Salary = Salary + 500
WHERE DeptID = 'XYZ'; -- Invalid DeptID

-- Rollback the second update
ROLLBACK TO BeforeSalaryUpdate;

-- Finalize the first update
COMMIT;
```

Let me know if you want this with error handling in stored procedures or PL/SQL!

a. Create a stored Procedure to Retrieve Student Details by ID.
b. Create a Procedure to Retrieve Highest and Lowest Marks.

Sure! Here's how you can write stored procedures in SQL (assuming you're using **MySQL** syntax; let me know if you're using Oracle, SQL Server, etc.).

🔹 a. Stored Procedure to Retrieve Student Details by ID

```sql
DELIMITER //

CREATE PROCEDURE GetStudentByID(IN student_id INT)
```

```
BEGIN
    SELECT *
    FROM Student
    WHERE StudentID = student_id;
END //

DELIMITER ;
```

> ✅ Call the procedure like:

```sql
CALL GetStudentByID(101);
```

🔹 b. Stored Procedure to Retrieve Highest and Lowest Marks

```sql
DELIMITER //

CREATE PROCEDURE GetHighestAndLowestMarks()
BEGIN
    SELECT
        MAX(Marks) AS HighestMark,
        MIN(Marks) AS LowestMark
    FROM Student;
END //

DELIMITER ;
```

> ✅ Call the procedure like:

```sql
CALL GetHighestAndLowestMarks();
```

Let me know if your table has a different structure (like a separate `Marks` table), or if you want to return the student names along with the highest/lowest marks!

a. On schema Employees(Empid,Name,Department,salary) Create a View which displays IT department
employees whose salary is greater than 60000
b. Create a View which displays IT department information
c. Create trigger to ensure that employee records are not permanently lost by copying them to a backup table
before deletion.

Here's how you can write the required **SQL view and trigger statements** based on the schema:

🔹 a. Create a view to display IT department employees with salary > 60000

```sql
CREATE VIEW IT_HighEarners AS
SELECT Empid, Name, Department, Salary
FROM Employees
WHERE Department = 'IT' AND Salary > 60000;
```

> ✅ Use:

```sql
SELECT * FROM IT_HighEarners;
```

🔹 b. Create a view to display only IT department information

```sql
```

```
CREATE VIEW IT_Department AS
SELECT *
FROM Employees
WHERE Department = 'IT';
```

> ✅ Use:

```sql
SELECT * FROM IT_Department;
```

🔹 c. Create a trigger to back up employee records before deletion

First, create a backup table:

```sql
CREATE TABLE Employees_Backup (
    Empid INT,
    Name VARCHAR(100),
    Department VARCHAR(100),
    Salary DECIMAL(10,2),
    DeletedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Then, create the trigger:

```sql
DELIMITER //

CREATE TRIGGER BeforeEmployeeDelete
BEFORE DELETE ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO Employees_Backup (Empid, Name, Department, Salary)
    VALUES (OLD.Empid, OLD.Name, OLD.Department, OLD.Salary);
END //
```

```
DELIMITER ;
```

> ✅ This trigger automatically copies the deleted record to `Employees_Backup` before it's removed from `Employees`.

Let me know if you want similar triggers for updates or other departments too!