
# SQL, Aggregate Function, Grouping, Union, Subqueries, EERD (Day 4)

## Aggregate Functions

⇒Aggregate Functions: count,min,sum,max,avg

Here's the SQL clean code version of the aggregate functions and queries explained, using the `CompanyDB` database:

1. **Count non-null SSNs**

```
SELECT COUNT(SSN) AS NonNullSSNs
FROM Employees
WHERE SSN IS NOT NULL;
```

2. **Count all rows**

```
SELECT COUNT(*) AS TotalRows
FROM Employees;
```

3. **Count non-null `EmployeeName` values (assuming `EmployeeName` replaces `Fname`)**

```
SELECT COUNT(EmployeeName) AS NonNullNames
FROM Employees;
```

4. **Average salary of employees**

```
SELECT AVG(Salary) AS AvgSalary
FROM Employees;
```

5. **Count of salaries with `SSN` grouped**

```
SELECT COUNT(Salary) AS SalaryCount, SSN
FROM Employees
GROUP BY SSN;

SELECT COUNT(Salary) AS SalaryCount, SSN
FROM Employees, Departments
WHERE SSN = '123-45-6789'
GROUP BY SSN;
```

6. **Minimum salary grouped by department**

```
SELECT MIN(Salary) AS MinSalary, DepartmentID
FROM Employees
GROUP BY DepartmentID;
```

7. **Count of employees by gender**

```
SELECT COUNT(SSN) AS GenderCount, Gender
FROM Employees
GROUP BY Gender;
```

8. **Count of employees by gender in departments 10 or 20**

```
SELECT COUNT(SSN) AS GenderCount, Gender
FROM Employees
```

```
WHERE DepartmentID IN (10, 20)
GROUP BY Gender;
```

9. **Count of employees in departments where the address starts with 'a'**

```
SELECT COUNT(SSN) AS EmployeeCount, DepartmentID
FROM Employees
WHERE Address LIKE 'a%'
GROUP BY DepartmentID;
```

## Using `HAVING` to Filter Groups

1. **Sum of salary grouped by gender with filtering**

```
SELECT SUM(Salary) AS TotalSalary, Gender
FROM Employees
GROUP BY Gender
HAVING SUM(Salary) > 10;
```

2. **Sum of salary grouped by department with address filtering**

```
SELECT SUM(Salary) AS TotalSalary, DepartmentID
FROM Employees
WHERE Address LIKE 'a%'
GROUP BY DepartmentID
HAVING SUM(Salary) > 12000;
```

3. **Maximum salary grouped by address, filtered by SSN count**

```
SELECT MAX(Salary) AS MaxSalary, Address
FROM Employees
WHERE DepartmentID IN (10, 30)
GROUP BY Address
HAVING COUNT(SSN) > 3;
```

4. **Average of `DepartmentID` ignoring `NULL` values**

```
SELECT AVG(ISNULL(DepartmentID, 0)) AS AvgDepartmentID
FROM Employees;
```

5. **Sum of `DepartmentID` divided by total count (without ignoring `NULL`)**

```
SELECT SUM(DepartmentID) / COUNT(*) AS AvgDepartmentID
FROM Employees;
```

## Grouping by Multiple Attributes

1. **Sum of salary grouped by department number and department name (using `INNER JOIN` with `Departments` table)**

```
SELECT SUM(e.Salary) AS TotalSalary, e.DepartmentID, d.DepartmentN
ame
FROM Employees e
INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID
GROUP BY e.DepartmentID, d.DepartmentName;
```

2. **Sum of salary grouped by department number and address**

```
SELECT SUM(Salary) AS TotalSalary, DepartmentID, Address
FROM Employees
GROUP BY DepartmentID, Address;
```

## Explanation:

1. **Aggregate Functions**:
   - `COUNT()`: Counts the number of non-null values or all rows depending on how it's used.
   - `AVG()`: Averages the values but ignores `NULL`.
   - `SUM()`, `MIN()`, `MAX()`: Performs sum, minimum, or maximum on the selected column.
2. `GROUP BY`: Required when using aggregate functions alongside normal columns.

3. `HAVING`: Used to filter the results after the `GROUP BY` operation (as opposed to `WHERE` which filters rows before grouping).

# Aggregate Functions with `GROUP BY` `HAVING`

**Rule ⇒ If you are selecting aggregate Function and another attribue you should use group by**

## Queries Using Aggregate Functions with `GROUP BY` and `HAVING`:

1. **Minimum Salary by Department (`Dno`)**:

```
SELECT MIN(Salary) AS MinSalary, DepartmentID
FROM Employees
GROUP BY DepartmentID;
```

   - **Explanation**: This selects the minimum salary for each department by grouping rows by `DepartmentID`.

2. **Count of Employees by Gender**:

```
SELECT COUNT(SSN) AS EmployeeCount, Sex
FROM Employees
GROUP BY Sex;
```

   - **Explanation**: This counts the number of employees grouped by gender (`Sex`).

3. **Count of Employees by Gender for Specific Departments (10, 20)**:

```
SELECT COUNT(SSN) AS EmployeeCount, Sex
FROM Employees
WHERE DepartmentID IN (10, 20)
GROUP BY Sex;
```

- **Explanation**: This counts the number of employees by gender, filtering for departments with IDs 10 or 20.

4. **Count of Employees by Department for Addresses Starting with 'A'**:

```
SELECT COUNT(SSN) AS EmployeeCount, DepartmentID
FROM Employees
WHERE Address LIKE '_a%'
GROUP BY DepartmentID;
```

- **Explanation**: This counts the number of employees in each department where the address starts with 'A'.

## Queries Using `HAVING`:

1. **Total Salary by Gender with Salary Sum Greater than 10**:

```
SELECT SUM(Salary) AS TotalSalary, Sex
FROM Employees
GROUP BY Sex
HAVING SUM(Salary) > 10;
```

- **Explanation**: This calculates the sum of salaries grouped by gender and filters the result to only include groups where the total salary is greater than 10.

2. **Sum of Salaries for Departments with Address Starting with 'A' and Salary Greater than 12,000**:

```
SELECT SUM(Salary) AS TotalSalary, DepartmentID
FROM Employees
WHERE Address LIKE '_a%'
GROUP BY DepartmentID
HAVING SUM(Salary) > 12000;
```

- **Explanation**: This sums the salaries of employees in each department where the address starts with 'A', and only shows departments where the sum exceeds 12,000.

3. **Maximum Salary by Address in Specific Departments (10, 30) with More Than 3 Employees**:

```
SELECT MAX(Salary) AS MaxSalary, Address
FROM Employees
WHERE DepartmentID IN (10, 30)
GROUP BY Address
HAVING COUNT(SSN) > 3;
```

- **Explanation**: This finds the maximum salary for employees grouped by address, but only for departments 10 and 30, and only where there are more than 3 employees per address.

## Working with `NULL` Values and `AVG`:

1. **Average of DepartmentID (Treating `NULL` as Zero)**:

```
SELECT AVG(ISNULL(DepartmentID, 0)) AS AvgDepartmentID
FROM Employees;
```

- **Explanation**: This computes the average of the `DepartmentID` column, treating `NULL` values as zero.

2. **Another Way to Calculate Average Using `SUM` and `COUNT`**:

```
SELECT SUM(DepartmentID) / COUNT(*) AS AvgDepartmentID
FROM Employees;
```

- **Explanation**: This computes the average `DepartmentID` by summing the non-null values and dividing by the total number of rows (including `NULL` values).

## Grouping by Multiple Attributes:

1. **Sum of Salaries by Department and Department Name**:

```
SELECT SUM(e.Salary) AS TotalSalary, e.DepartmentID, d.DepartmentN
ame
FROM Employees e
INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID
GROUP BY e.DepartmentID, d.DepartmentName;
```

- **Explanation**: This sums salaries for each department, grouped by `DepartmentID` and `DepartmentName`.

2. **Sum of Salaries Grouped by Department and Address**:

```
SELECT SUM(Salary) AS TotalSalary, DepartmentID, Address
FROM Employees
GROUP BY DepartmentID, Address;
```

   - **Explanation**: This sums the salaries of employees, grouped by both `DepartmentID` and `Address`.

### Important Rules:

- When selecting both aggregate functions and regular columns, you must use `GROUP BY` on the regular columns.
- After using `GROUP BY`, you can filter groups with the `HAVING` clause. `HAVING` must contain an aggregate function for filtering groups.

## SQL Query Execution Order:

```
SELECT DepartmentID, COUNT(EmployeeID)
FROM Employees
WHERE Salary > 2000
GROUP BY DepartmentID
HAVING COUNT(EmployeeID) > 5
ORDER BY COUNT(EmployeeID) DESC;
```

### Execution Order:

1. **FROM**: The query retrieves data from the `Employees` table.
2. **WHERE**: It filters rows where `Salary > 2000`. **Excute on Rows**
3. **GROUP BY**: The query groups the filtered rows by `DepartmentID`.
4. **HAVING**: It filters the grouped results, keeping only departments with more than 5 employees. **Excute on Groups**

5. **SELECT**: The query selects `DepartmentID` and the count of `EmployeeID` for the result.
6. **ORDER BY**: It sorts the results by the employee count in descending order.

# Subqueries

Let's rewrite your explanations of subqueries, `UNION`, `INTERSECT`, and `EXCEPT` in SQL, while following clean code principles.

## 1. Subqueries:

Subqueries allow you to embed a query within another query. They are often used when you need to compute a value that will be used in a higher-level query.

### Example 1: Error in using aggregate function directly.

```
-- This query would throw an error because you cannot directly compare a column to an aggregate function like `AVG()`.
SELECT *
FROM Employees
WHERE Salary > AVG(Salary); -- This will not work.
```

### Example 2: Correct approach using a subquery.

```
-- This query correctly uses a subquery to calculate the average salary and compares the salary in each row to this value.
SELECT *
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

### Example 3: Selecting employee details and printing the average salary for each row.

```
-- The subquery calculates the average salary once, and it prints the avera
ge salary along with each employee's details.
```

```
SELECT *, (SELECT AVG(Salary) FROM Employees) AS AverageSalary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

### Example 4: Printing a constant value in every row.

```
-- Instead of printing the average salary, this query prints a constant value
(14) in each row.
SELECT *, 14 AS ConstantValue
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

### Example 5: Subquery with another table (Departments).

```
-- This query selects employee details and prints the average department n
umber (Dnum) for each employee.
SELECT *, (SELECT AVG(DepartmentID) FROM Departments) AS AvgDepart
mentID
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

### Example 6: Subquery with `IN` clause.

```
-- This subquery finds all department numbers (Dnum) where at least one
employee is assigned.
SELECT DepartmentID
FROM Departments
WHERE DepartmentID IN (SELECT DISTINCT DepartmentID FROM Employe
es WHERE DepartmentID IS NOT NULL);
```

### Example 7: Better performance using `JOIN`.

```
-- This query achieves the same result as the previous one but performs be
tter due to the use of a `JOIN` instead of a subquery.
SELECT DISTINCT DepartmentName
```

```
FROM Departments d
INNER JOIN Employees e ON d.DepartmentID = e.DepartmentID;
```

## 2. Subquery + DML (Data Manipulation Language):

DML operations such as `UPDATE` can also involve subqueries.

## Example: Update based on subquery.

```
-- This query updates the working hours for all male employees by adding 1
0 hours.
UPDATE Works_For
SET HoursPerWeek = HoursPerWeek + 10
WHERE EmployeeID IN (SELECT EmployeeID FROM Employees WHERE Sex
= 'M');
```

## 3. Union family: `UNION` , `UNION ALL` , `INTERSECT` , `EXCEPT` .

These commands combine results from two or more queries. The rules for `UNION` family are:

- `UNION` : Combines results, removes duplicates, and sorts the result set.
- `UNION ALL` : Combines results without removing duplicates.
- `INTERSECT` : Returns only rows that are present in both queries.
- `EXCEPT` : Returns rows that are present in the first query but not in the second.

## Example 1: `UNION ALL` – Combines rows from two queries without removing duplicates.

```
-- Get first names and last names of employees in one table, with duplicate
s.
SELECT FirstName FROM Employees
UNION ALL
SELECT LastName FROM Employees;
```

## Example 2: `UNION` – Combines and sorts rows, removing duplicates.

```
-- Get first names and last names of employees, but remove duplicates and sort.
SELECT FirstName FROM Employees
UNION
SELECT LastName FROM Employees;
```

- **Note**: In both `UNION` and `UNION ALL`, the number of selected columns from both queries must be the same.

### Example 3: `INTERSECT` – Returns common values between two queries.

```
-- Get names that appear as both first names and last names.
SELECT FirstName FROM Employees
INTERSECT
SELECT LastName FROM Employees;
```

### Example 4: `EXCEPT` – Returns values in the first query that are not in the second.

```
-- Get first names that are not used as last names.
SELECT FirstName FROM Employees
EXCEPT
SELECT LastName FROM Employees;
```

### Example 5: Using `EXCEPT` on multiple columns.

```
-- Select all employees' first and last names, then exclude those who have the same first and last names as others.
SELECT FirstName, LastName FROM Employees
EXCEPT
SELECT FirstName, LastName FROM Employees;
```

### Summary:

- **Subqueries** are useful for comparing a value to an aggregated result or a condition that can only be calculated through a separate query.

- **DML with subqueries** enables you to manipulate data (like updating or deleting rows) based on conditions derived from other queries.
- `UNION` **family** is helpful for combining result sets, with different commands for whether or not you want duplicates ( `UNION` removes them, `UNION ALL` keeps them).
- `INTERSECT` and `EXCEPT` provide ways to compare and filter data across multiple queries.

## Enhanced ERD (EERD) and Inheritance

In an Enhanced Entity-Relationship Diagram (EERD), inheritance (specialization and generalization) plays a key role when dealing with entities that share common attributes but also have their unique attributes.

Enhanced ERD (EERD): (ERD + Inheritance)

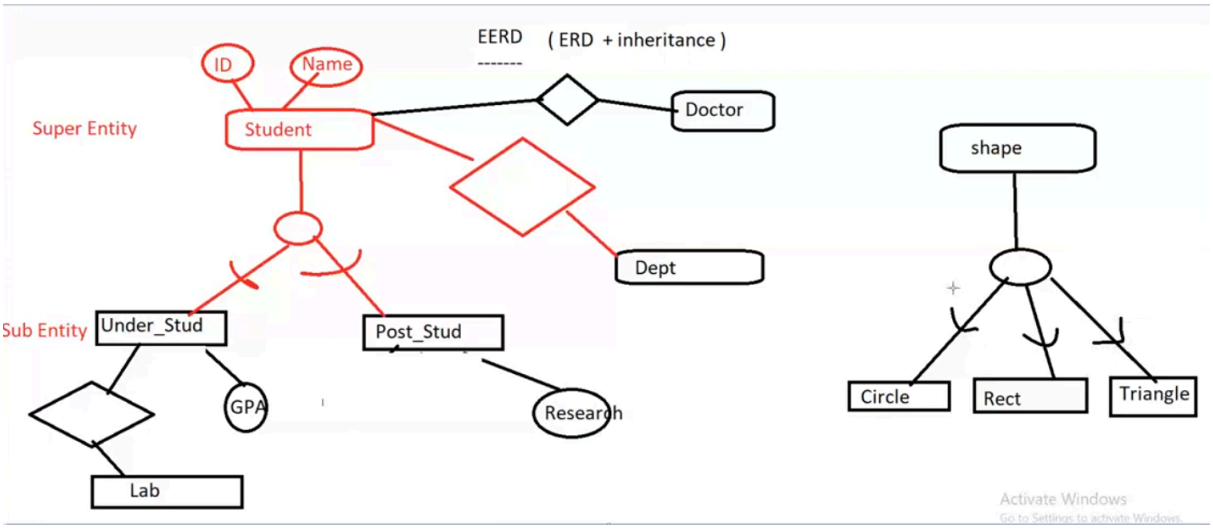
### Scenario:

You have two entity types: **Undergraduate** and **Postgraduate**, which share some common attributes like `ID` and `Name`, but they also have their own unique attributes like `GPA` for undergraduate students and `Research` for postgraduate students.

### Approach:

To model this, you can:

1. **Supertype**: Create a generic entity called `Student` which holds the common attributes ( `ID` , `Name` ).
2. **Subtypes**: Define `Undergraduate` and `Postgraduate` as subtypes that inherit from `Student` . Each subtype will have its specific attributes ( `GPA` for `Undergraduate` and `Research` for `Postgraduate` ).
3. **Relationship with supertype**: Any new table (such as an enrollment table or project table) will relate to the `Student` entity, not directly with `Undergraduate` or `Postgraduate` , as these are specializations of `Student` .




## Definitions:

1. **Subtype**:
   A subgrouping of entities in an entity type (supertype) which has attributes that are distinct from the other subgroups. Example:
   `Undergraduate` and `Postgraduate` are subtypes of `Student`.
2. **Supertype**:
   A generic entity that can be divided into subtypes. The supertype has a relationship with one or more subtypes. Example:
   `Student` is a supertype that holds common attributes of both `Undergraduate` and `Postgraduate`.
3. **Attributes in Subtypes**:
   Any attribute in the supertype (`Student`) is automatically assumed to be part of the subtype (`Undergraduate` and `Postgraduate`). Subtypes have additional attributes that are specific to them.
4. **Generalization and Specialization** :
   - **Generalization**: The process of defining a more general entity (supertype) from a set of more specialized entities (subtypes). In this case, we generalize `Undergraduate` and `Postgraduate` into the supertype `Student`.
   - **Specialization**: The process of defining a set of subtypes from a supertype. This is done when an entity has distinct characteristics in different contexts. For example, we specialize the `Student` entity into `Undergraduate` and `Postgraduate` entities.

## Example EERD Structure:

1. **Supertype (Student)**:
   - **Attributes**: `ID`, `Name`
2. **Subtype (Undergraduate)**:
   - **Attributes**: `GPA` (inherits `ID` and `Name` from `Student`)
3. **Subtype (Postgraduate)**:
   - **Attributes**: `Research` (inherits `ID` and `Name` from `Student`)

---

## SQL Example:

Let's create a relational schema that follows this inheritance model.

### Step 1: Create the `Student` supertype table.

```
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL
);
```

### Step 2: Create the `Undergraduate` subtype table, inheriting from `Student`.

```
CREATE TABLE Undergraduate (
    StudentID INT PRIMARY KEY,
    GPA DECIMAL(3, 2) NOT NULL,
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID)
);
```

### Step 3: Create the `Postgraduate` subtype table, inheriting from `Student`.

```
CREATE TABLE Postgraduate (
    StudentID INT PRIMARY KEY,
    Research VARCHAR(255) NOT NULL,
```

```
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID)
);
```

### Step 4: Any relationship should refer to the supertype `Student`.

```
-- Example: Create a table for course enrollments which relates to the supe
rtype 'Student'.
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseID INT,
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID)
);
```

---

### Conclusion:

By using the EERD concept with inheritance, you ensure that all common attributes are centralized in the `Student` table, and each specific entity (`Undergraduate`, `Postgraduate`) holds its own attributes. Any new relationships you want to create with students can directly link to the `Student` table, ensuring consistency and avoiding redundant data storage.

Let me know if you'd like further clarification or adjustments!

## Constraints in Supertype and Subtype Relations

Let's apply the **supertype/subtype** concept to the entities `Student`, `Undergraduate`, and `Postgraduate`. Here's how we can use the example with **constraints**, **discriminators**, and **mapping** for these entities.

### 1. Constraints in Supertype/Subtype Relationship

### Completeness Constraints:

- **Total Specialization Rule** (Double Line):

  If **every** student is either an `Undergraduate` or a `Postgraduate`, this represents **total specialization**.

Example: No student exists without being classified into one of the subtypes (Undergraduate or Postgraduate).

  - Notation: Double line between `Student` and its subtypes.

- **Partial Specialization Rule** (Single Line):

  If **some** students are not classified into either subtype (i.e., some students don't belong to `Undergraduate` or `Postgraduate`), this is called **partial specialization**.

  Example: A student can exist without being either an undergraduate or postgraduate.

  - Notation: Single line between `Student` and its subtypes.

## Disjointness Constraints:

- **Disjoint Rule (d)**:

  If a student can be **either** an `Undergraduate` or a `Postgraduate`, but **not both** at the same time, we use the **disjoint rule**.

  Example: A student cannot be simultaneously pursuing undergraduate and postgraduate degrees.

  - Notation: Letter "d" is written in the circle between the supertype and subtypes.

- **Overlap Rule (o)**:

  If a student can be **both** an `Undergraduate` and a `Postgraduate`, then the relationship is **overlapping**.

  Example: A student may be pursuing both undergraduate and postgraduate degrees simultaneously.

  - Notation: Letter "o" is written in the circle between the supertype and subtypes.

---

## 2. Subtype Discriminator

The **subtype discriminator** is an attribute that helps in determining to which subtype the entity belongs. There are different rules based on whether the relationship is **disjoint** or **overlap**.

- **Disjoint Rule**:

You would have a **single attribute** in the `Student` supertype that specifies whether the student is an `Undergraduate` or `Postgraduate`.

Example:

- Discriminator attribute: `StudentType` in the `Student` table.
- Possible values: `Undergraduate`, `Postgraduate`.

- **Overlap Rule**:

  You would use **multiple attributes** (a composite attribute) in the `Student` supertype to indicate if the student belongs to one or more subtypes.

  Example:

  - Discriminator attributes: `IsUndergraduate` (Yes/No), `IsPostgraduate` (Yes/No).

---

## 3. Mapping EERD to Tables

When mapping the **Enhanced Entity-Relationship Diagram (EERD)** into a relational database, each entity (supertype and subtypes) becomes a table. The primary key of the supertype is inherited by the subtypes as both the **primary key (PK)** and **foreign key (FK)**.

## Example 1: Total Specialization + Disjoint (d)

- Every student is either an `Undergraduate` or a `Postgraduate`, and a student can be only one of them.

**Tables**:

1. **Student Table** (Supertype):
   - `StudentID` (PK)
   - `Name`
   - `EnrollmentDate`
   - `StudentType` (`Undergraduate` or `Postgraduate`) – discriminator for disjoint.
2. **Undergraduate Table** (Subtype):
   - `StudentID` (PK, FK from Student)
   - `GPA`
3. **Postgraduate Table** (Subtype):

- `StudentID` (PK, FK from Student)
- `ResearchTopic`

### Example 2: Partial Specialization + Overlap (o)

- Some students may not be either an `Undergraduate` or a `Postgraduate`. A student can also be both (i.e., overlap).

**Tables**:

1. **Student Table** (Supertype):
   - `StudentID` (PK)
   - `Name`
   - `EnrollmentDate`
   - `IsUndergraduate` (Yes/No)
   - `IsPostgraduate` (Yes/No)
2. **Undergraduate Table** (Subtype):
   - `StudentID` (PK, FK from Student)
   - `GPA`
3. **Postgraduate Table** (Subtype):
   - `StudentID` (PK, FK from Student)
   - `ResearchTopic`

---

## 4. Examples of Mapping and Interpretation

Let's go through two examples, showing how mapping and relationships work in **total/partial** and **disjoint/overlap** situations.

### Example 1: Total Specialization & Disjoint (d)

- **Total Specialization**: All students are either undergraduates or postgraduates.
- **Disjoint**: A student cannot be both at the same time.
- **Student Table** has 100 rows.
- **Undergraduate Table** has 60 rows.

- **Postgraduate Table** has 40 rows.

This means all 100 students are classified as either undergraduate or postgraduate (total specialization), and no student is in both subtypes (disjoint).

### Example 2: Partial Specialization & Overlap (o)

- **Partial Specialization**: Some students might not belong to either subtype.
- **Overlap**: A student can belong to both subtypes.
- **Student Table** has 100 rows.
- **Undergraduate Table** has 60 rows.
- **Postgraduate Table** has 50 rows.

Here, we see that 10 students (100 - (60 + 50)) do not belong to any subtype (partial specialization), and some students belong to both subtypes because of the overlap.

---

### Conclusion:

In this `Student`, `Undergraduate`, and `Postgraduate` example, we can model **inheritance** using supertype and subtype relationships, specifying **constraints** like completeness (total vs. partial) and disjointness (disjoint vs. overlap), and use **discriminators** to identify which subtype an instance belongs to. Mapping these to tables in the database ensures proper relational structure and integrity based on the designed EERD.

> ⇒ Mapping EERD: every entity become table and the primary key of the supertype become (PK+FK) on the subtypes.

## SQL Functions Overview

1. `GETDATE()`
   - **Description:** Returns the current date and time from the server.
   - **Example:**

     ```
     SELECT GETDATE() AS CurrentDateTime;
     ```

2. `ISNULL(expression, replacement)`

- **Description:** Replaces NULL values with a specified replacement value.
- **Example:**

```
SELECT ISNULL(Salary, 0) AS Salary FROM Employee;
```

3. `COALESCE(value1, value2, ...)`
   - **Description:** Returns the first non-NULL value from the list of expressions.
   - **Example:**

```
SELECT COALESCE(Salary, Bonus, 0) AS TotalCompensation FROM Employee;
```

4. `CONCAT(string1, string2, ...)`
   - **Description:** Concatenates two or more strings into one.
   - **Example:**

```
SELECT CONCAT(FirstName, ' ', LastName) AS FullName FROM Employee;
```

5. `CONVERT(data_type, expression)`
   - **Description:** Converts an expression from one data type to another.
   - **Example:**

```
SELECT CONVERT(VARCHAR(10), GETDATE(), 101) AS FormattedDate;
```

6. `YEAR(date)`
   - **Description:** Extracts the year from a date value.
   - **Example:**

```
SELECT YEAR(GETDATE()) AS CurrentYear;
```

7. `MONTH(date)`

- **Description:** Extracts the month from a date value.
- **Example:**

```
SELECT MONTH(GETDATE()) AS CurrentMonth;
```

8. `SUBSTRING(string, start, length)`
   - **Description:** Returns a part of a string starting at a specified position for a specified length.
   - **Example:**

```
SELECT SUBSTRING(FirstName, 1, 3) AS FirstThreeChars FROM Em
ployee;
```

9. `DB_NAME()`
   - **Description:** Returns the name of the current database.
   - **Example:**

```
SELECT DB_NAME() AS CurrentDatabase;
```

10. `SUSER_NAME()`
    - **Description:** Returns the login name associated with the current security context.
    - **Example:**

```
SELECT SUSER_NAME() AS CurrentUser;
```