**Practice Problems Database:**

**CREATE TABLE:**

-- Create the Departments table

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(50),

Location VARCHAR(50)

);

-- Create the Employees table

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(50),

DepartmentID INT,

Salary DECIMAL(10, 2),

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

-- Inserting sample data into Departments

INSERT INTO Departments (DepartmentID, DepartmentName, Location) VALUES

(1, 'Sales', 'New York'),

(2, 'Marketing', 'San Francisco'),

(3, 'IT', 'Chicago'),

(4, 'Finance', 'New York'),

(5, 'HR', 'Los Angeles');

-- Inserting sample data into Employees

INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary) VALUES

(1, 'Alice', 1, 60000.00), -- Sales

(2, 'Bob', 2, 55000.00), -- Marketing

(3, 'Charlie', 3, 75000.00), -- IT

(4, 'David', 4, 80000.00), -- Finance

(5, 'Eve', 1, 70000.00), -- Sales

(6, 'Frank', 5, 45000.00), -- HR

(7, 'Grace', 3, 95000.00), -- IT

(8, 'Hannah', 2, 50000.00), -- Marketing

(9, 'Ivy', 4, 85000.00), -- Finance

(10, 'Jack', 3, 72000.00); -- IT

**Simple Nested Query using where attribute IN/NOT IN (Query) clause, Any and All.**

Problem 1: Find the names of all employees who work in departments located in 'New York'.

Problem 2: List the employees who are **not** in the 'Marketing' department.

Problem 3: Find employees who earn more than the highest salary in the 'IT' department.

Problem 4: Find the employees who earn more than everyone in the 'Finance' department.

**Aggregate functions (sql) (Count, max, min, sum, avg) Group by and use of Having clause**

Problem 1: Count the number of employees in each department.

Problem 2: Find the highest and lowest salary in each department.

Problem 3: Calculate the total salary expenditure in each department.

Problem 4: Find the average salary in each department, but only for departments with more than 2 employees.

Problem 5: Find the departments where the total salary expenditure exceeds 200,000.

Problem 6: Find the department with the highest average salary.

Problem 7: List the employees who have the highest salary in each department.

Problem 8: Find the departments where the average salary is above 60,000 and there are more than 3 employees.

Problem 9: Find the departments that have an average salary between 50,000 and 80,000.

Problem 10: Find the department with the most employees.

**Scenario: University Merit Calculation**

In a university, students are assessed based on two main components:

1. **Test Scores** (30% of the final merit)
2. **Annual Exam Scores** (70% of the final merit)

We will calculate the **final merit** score for each student based on the weighted average of these two components.

**Entities:**

* **Student**: StudentID, StudentName, StudentEmail
* **TestResult**: ResultID, StudentID, TestScore
* **AnnualExamResult**: ResultID, StudentID, AnnualExamScore

**Query Objective:**

Calculate the **final merit score** for each student by taking **30% of their Test Score** and **70% of their Annual Exam Score**.

**Database Tables:**

1. **Student** Table:
   * StudentID (Primary Key)
   * StudentName
   * StudentEmail
2. **TestResult** Table:
   * ResultID (Primary Key)
   * StudentID (Foreign Key)
   * TestScore (Out of 100)
3. **AnnualExamResult** Table:
   * ResultID (Primary Key)
   * StudentID (Foreign Key)
   * AnnualExamScore (Out of 100)

**Calculation Logic:**

* **Merit Calculation**:
  + **Test Score** contributes **30%** of the final score.
  + **Annual Exam Score** contributes **70%** of the final score.

The formula for the **final merit score** for each student is:

Final Merit Score=(Test Score×0.30)+(Annual Exam Score×0.70)\text{Final Merit Score} = (\text{Test Score} \times 0.30) + (\text{Annual Exam Score} \times 0.70)Final Merit Score=(Test Score×0.30)+(Annual Exam Score×0.70)

**Example Query:**

This SQL query calculates the **final merit score** for each student by joining the **TestResult** and **AnnualExamResult** tables with the **Student** table, and applying the weights to both scores.

SELECT

S.StudentID,

S.StudentName,

S.StudentEmail,

TR.TestScore,

AER.AnnualExamScore,

(

TR.TestScore \* 0.30 +

AER.AnnualExamScore \* 0.70

) AS FinalMeritScore

FROM

Student S

JOIN

TestResult TR ON S.StudentID = TR.StudentID

JOIN

AnnualExamResult AER ON S.StudentID = AER.StudentID;

**Example Data:**

1. **Student** Table:

|  |  |  |
| --- | --- | --- |
| **StudentID** | **StudentName** | **StudentEmail** |
| 1 | John Doe | john@example.com |
| 2 | Jane Smith | jane@example.com |
| 3 | Alice Green | alice@example.com |

1. **TestResult** Table:

|  |  |  |
| --- | --- | --- |
| **ResultID** | **StudentID** | **TestScore** |
| 101 | 1 | 80 |
| 102 | 2 | 85 |
| 103 | 3 | 90 |

1. **AnnualExamResult** Table:

|  |  |  |
| --- | --- | --- |
| **ResultID** | **StudentID** | **AnnualExamScore** |
| 201 | 1 | 75 |
| 202 | 2 | 80 |
| 203 | 3 | 95 |

**Sample Output:**

After running the above SQL query, the output would look like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **StudentID** | **StudentName** | **StudentEmail** | **TestScore** | **AnnualExamScore** | **FinalMeritScore** |
| 1 | John Doe | john@example.com | 80 | 75 | 77.50 |
| 2 | Jane Smith | jane@example.com | 85 | 80 | 81.50 |
| 3 | Alice Green | alice@example.com | 90 | 95 | 93.50 |

**Question: Find the first two topper based on total merit among above.**

**Question: Find the last student based on the total merit.**

**Multi-table Select (Cross Product), Joins(Inner, Natural, left outer join, right outer join, equi join)**

**1. Cross Product (Cartesian Product)**

The **Cross Product** returns the Cartesian product of two tables, meaning every row from the first table is combined with every row from the second table. **Note**: This is rarely used in practice, as it results in large result sets.

**Query:**

SELECT \*

FROM Student, Course;

**Explanation:**

This query produces a combination of every row from the Student table with every row from the Course table. For the example data, it would return 9 rows (3 students × 3 courses).

**Sample Output:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **StudentID** | **FirstName** | **LastName** | **Email** | **CourseID** | **CourseName** | **Credits** |
| 1 | John | Doe | john.doe@example.com | 101 | Database Systems | 3 |
| 1 | John | Doe | john.doe@example.com | 102 | Web Development | 4 |
| 1 | John | Doe | john.doe@example.com | 103 | Data Science | 3 |
| 2 | Jane | Smith | jane.smith@example.com | 101 | Database Systems | 3 |
| 2 | Jane | Smith | jane.smith@example.com | 102 | Web Development | 4 |
| 2 | Jane | Smith | jane.smith@example.com | 103 | Data Science | 3 |
| 3 | Alice | Johnson | alice.johnson@example.com | 101 | Database Systems | 3 |
| 3 | Alice | Johnson | alice.johnson@example.com | 102 | Web Development | 4 |
| 3 | Alice | Johnson | alice.johnson@example.com | 103 | Data Science | 3 |

**2. Inner Join**

An **Inner Join** returns only the rows that have matching values in both tables. It excludes rows that don't have a match.

SELECT S.StudentID, S.FirstName, C.CourseName

FROM Student S

INNER JOIN Enrollment E ON S.StudentID = E.StudentID

INNER JOIN Course C ON E.CourseID = C.CourseID;

**Explanation:**

This query finds all students who are enrolled in at least one course and lists the course names they're enrolled in. It combines data from the Student, Enrollment, and Course tables using inner joins.

**Sample Output:**

|  |  |  |
| --- | --- | --- |
| **StudentID** | **FirstName** | **CourseName** |
| 1 | John | Database Systems |
| 1 | John | Web Development |
| 2 | Jane | Database Systems |
| 3 | Alice | Data Science |

**3. Natural Join**

A **Natural Join** automatically joins tables based on all columns with the same name in both tables. It is typically used when the column names are identical and there’s no need to specify the join condition.

**Query:**

SELECT Student.StudentID, Student.FirstName, Course.CourseName

FROM Student

NATURAL JOIN Enrollment

NATURAL JOIN Course;

**Explanation:**

In this case, the NATURAL JOIN will join the tables Student, Enrollment, and Course based on common column names (in this case, StudentID and CourseID).

**Sample Output:**

|  |  |  |
| --- | --- | --- |
| **StudentID** | **FirstName** | **CourseName** |
| 1 | John | Database Systems |
| 1 | John | Web Development |
| 2 | Jane | Database Systems |
| 3 | Alice | Data Science |

**4. Left Outer Join**

A **Left Outer Join** returns all the rows from the left table and the matching rows from the right table. If there’s no match, the result is NULL on the side of the right table.

**Query:**

SELECT Student.StudentID, Student.FirstName, Course.CourseName

FROM Student

LEFT OUTER JOIN Enrollment ON Student.StudentID = Enrollment.StudentID

LEFT OUTER JOIN Course ON Enrollment.CourseID = Course.CourseID;

**Explanation:**

This query will return all students, even those who are not enrolled in any course. If a student is not enrolled in any course, the CourseName will be NULL.

**Sample Output:**

|  |  |  |
| --- | --- | --- |
| **StudentID** | **FirstName** | **CourseName** |
| 1 | John | Database Systems |
| 1 | John | Web Development |
| 2 | Jane | Database Systems |
| 3 | Alice | Data Science |
| 4 | Bob | NULL |

Here, if Bob is not enrolled in any courses, the result will still show him with NULL for CourseName.

**5. Right Outer Join**

A **Right Outer Join** returns all the rows from the right table and the matching rows from the left table. If there’s no match, the result is NULL on the side of the left table.

**Query:**

SELECT Student.StudentID, Student.FirstName, Course.CourseName

FROM Student

RIGHT OUTER JOIN Enrollment ON Student.StudentID = Enrollment.StudentID

RIGHT OUTER JOIN Course ON Enrollment.CourseID = Course.CourseID;

**Explanation:**

This query will return all courses, even those that don't have any enrolled students. If a course has no students, the StudentID and FirstName will be NULL.

**Sample Output:**

|  |  |  |
| --- | --- | --- |
| **StudentID** | **FirstName** | **CourseName** |
| 1 | John | Database Systems |
| 1 | John | Web Development |
| 2 | Jane | Database Systems |
| 3 | Alice | Data Science |
| NULL | NULL | Machine Learning |

Here, if a course like "Machine Learning" has no enrolled students, the result will still show that course with NULL for StudentID and FirstName.

**6. Equi Join**

An **Equi Join** is a type of inner join where the join condition uses the equality operator (=). It is one of the most common types of joins, where tables are joined based on the equality of columns.

**Query:**

SELECT Student.StudentID, Student.FirstName, Course.CourseName

FROM Student

JOIN Enrollment ON Student.StudentID = Enrollment.StudentID

JOIN Course ON Enrollment.CourseID = Course.CourseID;

**Explanation:**

This is essentially an **Inner Join** based on equality between columns. The JOIN keyword here is equivalent to an **Equi Join**, where the condition is explicitly checking for equality of columns (e.g., S.StudentID = E.StudentID).

**Sample Output:**

|  |  |  |
| --- | --- | --- |
| **StudentID** | **FirstName** | **CourseName** |
| 1 | John | Database Systems |
| 1 | John | Web Development |
| 2 | Jane | Database Systems |
| 3 | Alice | Data Science |

**Introduction to SQL (DML) Select statement without conditions Wild card (\*) vs Select statement with conditions & Where Clause with comparison operator (, <=, >=, < >, = =) Where clause, IS NULL, IS NOT NULL Special operators Between, like wild cards (\_, %), order by, distinct, limit**

Let's go through the different **SQL DML (Data Manipulation Language)** concepts and examples based on the queries you're asking for.

**1. Select Statement Without Conditions:**

A simple SELECT statement is used to retrieve data from a table. Without conditions, it fetches all records from the table.

**Example:**

SELECT \* FROM Student;

* This query retrieves all columns and all rows from the Student table.

If you want specific columns, you can specify those:

SELECT StudentID, FirstName, LastName FROM Student;

**. Wildcard (\*) vs <attribute list>**

* The \* wildcard retrieves **all columns** from the specified table.
* A specific attribute list retrieves only the **specified columns**.

**Example with \*:**

SELECT \* FROM Course;

**3. Select Statement with Conditions (Using WHERE Clause)**

The WHERE clause allows filtering of records based on specified conditions. Conditions can use comparison operators such as <, >, <=, >=, <> (not equal), and =.

**Examples of WHERE Clause with Comparison Operators:**

* **Equal to (=):**

SELECT \* FROM Student

WHERE StudentID = 1;

* **Greater than (>):**

SELECT \* FROM Student

WHERE Age > 20;

* **Less than (<):**

SELECT \* FROM Student

WHERE Age < 18;

* **Greater than or equal to (>=):**

SELECT \* FROM Student

WHERE Age >= 18;

* **Less than or equal to (<=):**

SELECT \* FROM Student

WHERE Age <= 22;

* **Not equal to (<> or !=):**

SELECT \* FROM Student

WHERE FirstName <> 'John';

**4. IS NULL and IS NOT NULL**

The IS NULL and IS NOT NULL operators are used to check if a column contains **NULL** values.

* **IS NULL**: Filters records where the value is NULL.

SELECT \* FROM Student

WHERE Email IS NULL;

**IS NOT NULL**: Filters records where the value is **not NULL**.

SELECT \* FROM Student

WHERE Email IS NOT NULL;

**5. Special Operators: BETWEEN, LIKE, ORDER BY, DISTINCT, LIMIT**

* **BETWEEN**: Used to filter records within a certain range (inclusive)

SELECT \* FROM Student

WHERE Age BETWEEN 18 AND 22;

**LIKE**: Used to filter records based on a pattern match. The % symbol is used as a wildcard for any sequence of characters, and \_ is used as a wildcard for a single character.

* **% (wildcard for zero or more characters):**

SELECT \* FROM Student

WHERE FirstName LIKE 'J%';

This query will return all students whose first name starts with 'J' (e.g., 'John', 'Jane').

 **\_ (wildcard for exactly one character):**

SELECT \* FROM Student

WHERE FirstName LIKE '\_ohn';

**ORDER BY**: Used to sort the results of a query in ascending or descending order.

* **Ascending Order (default):**

SELECT \* FROM Student

ORDER BY LastName;

**Descending Order:**

SELECT \* FROM Student

ORDER BY LastName DESC;

**DISTINCT**: Used to return only unique (non-duplicate) values from a column

SELECT DISTINCT CourseName FROM Enrollment;

**LIMIT**: Used to limit the number of records returned by the query.

SELECT \* FROM Student

LIMIT 5;

**Example: Select the Top 3 Students Based on Their Grades**

Assume we have a table called StudentGrades with the columns StudentID, FirstName, LastName, and Grade. We want to display the top 3 students based on their Grade in descending order (highest grade first).

**SQL Query:**

SELECT StudentID, FirstName, LastName, Grade

FROM StudentGrades

ORDER BY Grade DESC

LIMIT 3;

**Another Example: Top 3 Most Recent Students by Enrollment Date**

Assume we have an Enrollment table with columns StudentID, CourseID, and EnrollmentDate. If you want to find the top 3 most recent students who enrolled in courses, you would order by the EnrollmentDate in descending order:

**SQL Query:**

SELECT StudentID, CourseID, EnrollmentDate

FROM Enrollment

ORDER BY EnrollmentDate DESC

LIMIT 3;

**Example: Select the Bottom 3 Students Based on Their Grades**

Assume we have a table called StudentGrades with the columns StudentID, FirstName, LastName, and Grade, and we want to display the **bottom 3 students** based on their Grade in ascending order (lowest grade first).

**SQL Query:**

SELECT StudentID, FirstName, LastName, Grade

FROM StudentGrades

ORDER BY Grade ASC

LIMIT 3;

**Alter table (add, modify, drop) Insert Statement and variations (all attribute, row with null attribute, Optional attribute) Update Statement, Delete Statement**

-- Step 1: Create the Student table

CREATE TABLE Student (

StudentID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Age INT,

Email VARCHAR(100) NULL

);

-- Step 2: Insert data into the Student table

INSERT INTO Student (StudentID, FirstName, LastName, Age, Email)

VALUES (1, 'John', 'Doe', 20, 'john.doe@example.com');

INSERT INTO Student (StudentID, FirstName, LastName, Age)

VALUES (2, 'Alice', 'Smith', 22, NULL);

-- Step 3: Alter the table

ALTER TABLE Student

ADD EnrollmentDate DATE; -- Add a new column

ALTER TABLE Student

MODIFY FirstName VARCHAR(100); -- Modify an existing column

ALTER TABLE Student

DROP COLUMN Email; -- Drop the Email column

-- Step 4: Update a student's data

UPDATE Student

SET FirstName = 'Johnathan', Age = 21

WHERE StudentID = 1;

-- Step 5: Delete a student's record

DELETE FROM Student

WHERE StudentID = 2;

-- Step 6: Insert new data with the modified table structure

INSERT INTO Student (StudentID, FirstName, LastName, Age, EnrollmentDate)

VALUES (3, 'Bob', 'Johnson', 25, '2024-09-01');

**What happens in this sequence?**

1. **Table Creation:** The Student table is created with the original columns.
2. **Insert Data:** Two students are inserted. One has an Email, and the other has NULL for the Email.
3. **Alter Table:**
   * The EnrollmentDate column is added.
   * The FirstName column is modified to allow up to 100 characters.
   * The Email column is dropped.
4. **Update:** The first student's name and age are updated.
5. **Delete:** The student with StudentID = 2 is deleted.
6. **Insert New Data:** A new student is inserted with the updated table structure, including the new EnrollmentDate column.