🔗 Types of SQL Joins

A JOIN combines rows from two or more tables based on a related column.

Assume we have two tables:

透 Employees

EmpID	Name	DeptID
1	Alice	10
2	Bob	20
3	Charlie	30
4	David	NULL

 Departments

DeptID	DeptName
10	HR
20	IT
30	Finance
40	Marketing

INNER JOIN

顗 Returns rows that have matching values in both tables.

```
SELECT e.EmpID, e.Name, d.DeptName

FROM Employees e

INNER JOIN Departments d

ON e.DeptID = d.DeptID;
```

Result: (only matching DeptID)

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance

✕ David not shown (no DeptID)

✕ Marketing not shown (no employee)

▢ LEFT JOIN (LEFT OUTER JOIN)

頡 Returns all rows from the left table, plus matched rows from right.

If no match → NULL.

```
SELECT e.EmpID, e.Name, d.DeptName

FROM Employees e

LEFT JOIN Departments d

ON e.DeptID = d.DeptID;
```

Result:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance
4	David	NULL

✅ David is included (no dept → NULL)

▢ RIGHT JOIN (RIGHT OUTER JOIN)

頡 Returns all rows from the right table, plus matched rows from left.

If no match → NULL.

```
SELECT e.EmpID, e.Name, d.DeptName

FROM Employees e

RIGHT JOIN Departments d

ON e.DeptID = d.DeptID;
```

Result:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance
NULL	NULL	Marketing

✅ Marketing included (no employee → NULL)

FULL OUTER JOIN

顥 Returns all rows from both tables, matched where possible.

If no match → NULL.

```sql
SELECT e.EmpID, e.Name, d.DeptName

FROM Employees e

FULL OUTER JOIN Departments d

ON e.DeptID = d.DeptID;
```

Result:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance
4	David	NULL
NULL	NULL	Marketing

✅ Everything is included.

▢ CROSS JOIN (Cartesian Product)

顥 Returns all combinations of rows from both tables (no condition).

```sql
SELECT e.Name, d.DeptName

FROM Employees e

CROSS JOIN Departments d;
```

If 4 employees × 4 departments → 16 rows.

Result example (first few):

Name	DeptName
Alice	HR
Alice	IT
Alice	Finance
Alice	Marketing
Bob	HR

…	…

SELF JOIN

䫎 A table joined with itself (useful for hierarchy, manager–employee relations).

Example: Employees have a ManagerID column pointing to another employee.

SELECT e.Name AS Employee, m.Name AS Manager

FROM Employees e

LEFT JOIN Employees m

ON e.ManagerID = m.EmpID;

NATURAL JOIN (Oracle specific)

䫎 Automatically joins tables on columns with the same name and data type.

⚠ Risky because it joins by all matching column names.

SELECT *

FROM Employees

NATURAL JOIN Departments;

EQUI JOIN vs NON-EQUI JOIN

Equi Join → uses = operator (basically INNER JOIN).

Non-Equi Join → uses other operators (<, >, BETWEEN).

Example (find employees in salary ranges):

SELECT e.Name, s.Grade

FROM Employees e

JOIN SalaryGrade s

ON e.Salary BETWEEN s.MinSalary AND s.MaxSalary;

 Summary (Quick Table)

Join Type	Returns
INNER JOIN	Only matching rows
LEFT JOIN	All left + matching right
RIGHT JOIN	All right + matching left
FULL JOIN	All from both tables

CROSS JOIN	All combinations
SELF JOIN	Table joined with itself
NATURAL JOIN	Auto-join on same column names
NON-EQUI JOIN	Join with <, >, BETWEEN

Show Venn-diagram style visuals for each join (easy to memorize).

Explain SELF JOIN in more detail with an example.

▯ SQL Joins with Venn Diagrams

Imagine:

Left circle 透 = Employees table

Right circle ▯ = Departments table

▢ INNER JOIN

✅ Only rows that match in both tables.

透 ∩ ▯

▢ LEFT JOIN (LEFT OUTER JOIN)

✅ All rows from Employees + matching from Departments.

✕ If no match → NULL on right.

透 + (透 ∩ ▯)

▢ RIGHT JOIN (RIGHT OUTER JOIN)

✅ All rows from Departments + matching from Employees.

✕ If no match → NULL on left.

▯ + (透 ∩ ▯)

▢ FULL OUTER JOIN

✅ All rows from both tables.

✕ If no match, NULL on the missing side.

透 ∪ ▯

CROSS JOIN

✅ All combinations (Cartesian product).

⚠ Can create a huge result.

透 × 	 (every row with every row)

SELF JOIN

✅ A table joins with itself.

Think of 2 copies of the same table: Employees → Employee vs Manager.

Employees 透 join Employees 透

 SELF JOIN Explained (Step by Step)

Let’s add more data.

透 Employees Table

EmpID	Name	ManagerID
1	Alice	NULL
2	Bob	1
3	Charlie	1
4	David	2

Goal: Find Employee → Manager Name.

顗 Employee and Manager are in the same table.

Alice has no manager.

Bob’s manager is EmpID 1 → Alice.

Charlie’s manager is EmpID 1 → Alice.

David’s manager is EmpID 2 → Bob.

SQL Query:

```
SELECT e.Name AS Employee, m.Name AS Manager

FROM Employees e

LEFT JOIN Employees m

ON e.ManagerID = m.EmpID;
```

How it Works:

We take 2 "copies" of the same table:

e = Employees (for Employee)

m = Employees (for Manager)

Then we match e.ManagerID = m.EmpID.

Result:

Employee	Manager
Alice	NULL
Bob	Alice
Charlie	Alice
David	Bob

✅ This is why it's called SELF JOIN – we join the table with itself.

Q) We had 3 tables:

Emp (empid, firstname, lastname, deptid, ...)

dept (deptid, dname, location, ...)

project (project_id, pname, deptid, managed, ...)

Q1: Find the department names (dname) that manage more than one project. Also, display the total number of projects managed.

We need to join dept and project, then group by department and count projects.

```
SELECT d.dname, COUNT(p.project_id) AS total_projects

FROM dept d

JOIN project p ON d.deptid = p.deptid

GROUP BY d.dname

HAVING COUNT(p.project_id) > 1;
```

✅ This gives all department names with more than 1 project and total projects managed.

Q2: Display the deptid and dname of departments that are not managing any projects.

This means departments that do not appear in the project table. We can use LEFT JOIN or NOT EXISTS.

Option 1 (LEFT JOIN):

```
SELECT d.deptid, d.dname

FROM dept d

LEFT JOIN project p ON d.deptid = p.deptid
```

```
WHERE p.project_id IS NULL;
```

Option 2 (NOT EXISTS):

```
SELECT d.deptid, d.dname
FROM dept d
WHERE NOT EXISTS (
    SELECT 1 FROM project p WHERE p.deptid = d.deptid
);
```

✅ Both give departments without projects.

Q3: Get the first name, last name, and department name of employees who work in a department located in 'CHICAGO' and whose department manages at least one project.

Steps:

Join emp with dept (to get department and location).

Ensure location = 'CHICAGO'.

Ensure department manages ≥1 project → check using EXISTS or JOIN.

```
SELECT e.firstname, e.lastname, d.dname
FROM emp e
JOIN dept d ON e.deptid = d.deptid
WHERE d.location = 'CHICAGO'
  AND EXISTS (
      SELECT 1 FROM project p
      WHERE p.deptid = d.deptid
  );
```

✅ This ensures only employees in Chicago-based departments that manage at least one project are returned.

⚡ So in short:

Q1 → Use JOIN + GROUP BY + HAVING.

Q2 → Use LEFT JOIN ... IS NULL or NOT EXISTS.

Q3 → Use JOIN + EXISTS.

1. What is a package in PL/SQL?

A package is a collection of related procedures, functions, variables, cursors, and exceptions stored together as a single unit in the database.It has two parts:

Specification – declares the public items (procedures, functions, variables).

Body – contains the implementation of those items.

✅ Example:

```
-- Package Specification
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE get_emp_details(p_emp_id IN NUMBER);
  FUNCTION get_salary(p_emp_id IN NUMBER) RETURN NUMBER;
END emp_pkg;
/

-- Package Body
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE get_emp_details(p_emp_id IN NUMBER) IS
    v_name VARCHAR2(50);
    v_salary NUMBER;
  BEGIN
    SELECT first_name, salary INTO v_name, v_salary
    FROM employees WHERE employee_id = p_emp_id;
    DBMS_OUTPUT.PUT_LINE('Name: ' || v_name || ' Salary: ' || v_salary);
  END;

  FUNCTION get_salary(p_emp_id IN NUMBER) RETURN NUMBER IS
    v_salary NUMBER;
  BEGIN
```

```
    SELECT salary INTO v_salary FROM employees WHERE employee_id = p_emp_id;
    RETURN v_salary;
  END;
END emp_pkg;
/
```

2. How do you handle errors in nested PL/SQL blocks?

Use EXCEPTION handling inside nested blocks.

The inner block handles its errors first. If not handled, the error propagates to the outer block.

✅ Example:

```
BEGIN
  BEGIN
    -- Inner block
    DECLARE v_sal NUMBER;
    BEGIN
      SELECT salary INTO v_sal FROM employees WHERE employee_id = 9999; -- invalid
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee not found in inner block');
    END;
  END;

  -- Outer block
  DBMS_OUTPUT.PUT_LINE('Continuing in outer block...');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error handled in outer block');
END;
/
```

3. What is a REF CURSOR and when would you use it?

A REF CURSOR is a pointer to a result set.

Useful when you want to return query results dynamically from procedures/functions.

✅ Example:

```
DECLARE
  TYPE ref_cursor IS REF CURSOR;
  c_emp ref_cursor;
  v_name employees.first_name%TYPE;
BEGIN
  OPEN c_emp FOR SELECT first_name FROM employees WHERE department_id = 10;
  LOOP
    FETCH c_emp INTO v_name;
    EXIT WHEN c_emp%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_name);
  END LOOP;
  CLOSE c_emp;
END;
/
```

4. Explain transaction control in PL/SQL.

PL/SQL supports transaction control statements:

COMMIT → Save changes permanently.

ROLLBACK → Undo changes.

SAVEPOINT → Mark a point to rollback partially.

✅ Example:

```
BEGIN
  INSERT INTO employees(employee_id, first_name, salary, department_id)
  VALUES (3001, 'John', 5000, 10);
  SAVEPOINT sp1;
```

```
   UPDATE employees SET salary = 6000 WHERE employee_id = 3001;

   ROLLBACK TO sp1; -- undo update, keep insert

   COMMIT; -- save insert

END;

/
```

5. What is %ROWCOUNT and how is it used?

%ROWCOUNT is an implicit cursor attribute that shows how many rows were affected by the last DML.

✅ Example:

```
BEGIN

   UPDATE employees SET salary = salary + 1000 WHERE department_id = 20;

   DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated.');

END;

/
```

6. How do you pass parameters to procedures and functions?

Parameters are passed in procedure/function declarations.

They can be IN, OUT, or IN OUT.

✅ Example:

```
-- Procedure with parameters

CREATE OR REPLACE PROCEDURE update_salary(p_emp_id IN NUMBER, p_new_sal IN NUMBER) IS

BEGIN

   UPDATE employees SET salary = p_new_sal WHERE employee_id = p_emp_id;

   COMMIT;

END;

/

-- Calling procedure

BEGIN

   update_salary(101, 7000);

END;
```

```
/
```

7. What are IN, OUT, and IN OUT parameters?

IN → Passes a value into the procedure (default).

OUT → Returns a value out from the procedure.

IN OUT → Passes a value in and also returns it.

✅ Example:

```
CREATE OR REPLACE PROCEDURE param_demo(
   p_in    IN NUMBER,
   p_out   OUT VARCHAR2,
   p_inout IN OUT NUMBER
) IS
BEGIN
   p_out := 'Square is ' || (p_in * p_in);
   p_inout := p_inout + 10;
END;
/
-- Call procedure
DECLARE
   v_out VARCHAR2(100);
   v_inout NUMBER := 5;
BEGIN
   param_demo(4, v_out, v_inout);
   DBMS_OUTPUT.PUT_LINE(v_out);      -- Output: Square is 16
   DBMS_OUTPUT.PUT_LINE(v_inout);    -- Output: 15
END;
/
```

8. How can you use bind variables in PL/SQL?

Bind variables improve performance and security (prevent SQL injection).

Used with : prefix in SQL*Plus or tools like Oracle Forms.

✅ Example:

```
VARIABLE b_emp_id NUMBER
EXEC :b_emp_id := 101;

BEGIN
   UPDATE employees SET salary = salary + 500 WHERE employee_id = :b_emp_id;
   COMMIT;
END;
/
```

9. How can you optimize PL/SQL performance?

Use bulk operations (BULK COLLECT, FORALL) to reduce context switching.

Use bind variables.

Avoid unnecessary SQL inside loops.

Use packages (loaded once into memory).

Use collection types.

✅ Example (BULK COLLECT + FORALL):

```
DECLARE
   TYPE t_ids IS TABLE OF employees.employee_id%TYPE;
   v_ids t_ids;
BEGIN
   SELECT employee_id BULK COLLECT INTO v_ids FROM employees WHERE department_id = 30;

   FORALL i IN 1..v_ids.COUNT
      UPDATE employees SET salary = salary + 500 WHERE employee_id = v_ids(i);

   COMMIT;
END;
/
```

10. What is an autonomous transaction?

An autonomous transaction is independent of the main transaction.

Used for logging, auditing, etc.

Declared with PRAGMA AUTONOMOUS_TRANSACTION.

✅ Example:

```
CREATE OR REPLACE PROCEDURE log_error(p_msg VARCHAR2) IS

PRAGMA AUTONOMOUS_TRANSACTION;

BEGIN

  INSERT INTO error_log(log_message, log_date)

  VALUES(p_msg, SYSDATE);

  COMMIT; -- must commit because it’s separate

END;

/

-- Main transaction

BEGIN

  UPDATE employees SET salary = salary/0; -- error

EXCEPTION

  WHEN ZERO_DIVIDE THEN

    log_error('Division by zero occurred');

END;
```

Indexes: Clustered vs Non-Clustered

 What is an Index?

An index is like a "table of contents" in a book.

It helps the database find rows faster without scanning the whole table.

 Clustered Index

Only one per table (because it defines the physical order of data).

The table data itself is stored in the index order.

Acts like a phonebook: data is stored in sorted order.

Example: In SQL Server/MySQL (InnoDB), the primary key is usually the clustered index.

Example:

```
CREATE CLUSTERED INDEX idx_emp_id
ON Emp(empid);
```

Here, the Emp table’s rows will be physically stored in the order of empid.

Use case: Best for columns often used in range queries or sorting (e.g., dates, IDs).

 Non-Clustered Index

Many allowed per table.

Data is stored separately from the index.

Index holds a pointer to the actual row.

Works like the index at the back of a book → it tells you where to find the data.

Example:

```
CREATE NONCLUSTERED INDEX idx_emp_lastname
ON Emp(lastname);
```

Here, the database creates a separate structure mapping lastname to empid (or row address).

Use case: Best for columns used in search conditions (WHERE, JOIN) but not necessarily sorted.

 Quick Comparison Table

Feature	Clustered Index	Non-Clustered Index
Number per table	1	Many
Storage	Data is stored in index order	Separate index with row pointers
Speed	Faster for range queries	Faster for lookups/search
Example	Primary Key	Secondary indexes like email, name

18. How to Optimize Slow Queries

When a query is slow, possible causes include missing indexes, poor joins, large data scans, etc.

Here’s how to optimize step by step:

1) Use Indexes

Ensure WHERE, JOIN, ORDER BY, and GROUP BY columns are indexed.

Example (slow):

```
SELECT * FROM Emp WHERE lastname = 'Sharma';
```

If no index on lastname, DB scans all rows (full table scan).

Optimized:

```
CREATE INDEX idx_emp_lastname ON Emp(lastname);

SELECT * FROM Emp WHERE lastname = 'Sharma';
```

Now it quickly finds matching rows.

2) Avoid SELECT *

Only fetch required columns → reduces I/O.

Bad:

```
SELECT * FROM Emp WHERE deptid = 10;
```

Good:

```
SELECT firstname, lastname FROM Emp WHERE deptid = 10;
```

3) Use Joins Efficiently

Prefer explicit JOIN over subqueries when possible.

Bad (subquery):

```
SELECT firstname
FROM Emp
WHERE deptid IN (SELECT deptid FROM Dept WHERE location = 'CHICAGO');
```

Good (join):

```
SELECT e.firstname
FROM Emp e
JOIN Dept d ON e.deptid = d.deptid
WHERE d.location = 'CHICAGO';
```

4) Analyze Execution Plan

Use EXPLAIN (MySQL/Postgres) or EXPLAIN PLAN (Oracle) to check if query uses indexes or scans.

```
EXPLAIN SELECT * FROM Emp WHERE lastname = 'Sharma';
```

If it shows full table scan, add an index.

5) Optimize WHERE Clauses

Avoid functions on indexed columns → prevents index use.

Bad:

```
SELECT * FROM Emp WHERE YEAR(hire_date) = 2023;
```

(DB can’t use index on hire_date here)

Good:

```
SELECT * FROM Emp
WHERE hire_date >= '2023-01-01' AND hire_date < '2024-01-01';
```

6) Use Proper Data Types

Match column types in joins (int vs varchar mismatch = slow).

Use smallest data type possible (e.g., INT instead of BIGINT if values fit).

7) Limit Result Sets

Use LIMIT or pagination to avoid returning unnecessary rows.

```
SELECT * FROM Emp ORDER BY empid LIMIT 50;
```

8) Denormalization / Caching

For very large queries:

Precompute aggregates into summary tables.

Use caching (Redis, Memcached).

✅ Summary

Clustered Index: defines physical storage (only one per table, usually PK).

Non-Clustered Index: extra lookup structure (many per table).

Optimize slow queries: use indexes, avoid SELECT *, write efficient joins, check execution plan, rewrite WHERE conditions, limit result sets.

1. What are the different types of SQL statements?

Answer:

SQL statements are mainly divided into:

DDL (Data Definition Language): CREATE, ALTER, DROP, TRUNCATE

DML (Data Manipulation Language): INSERT, UPDATE, DELETE

DQL (Data Query Language): SELECT

DCL (Data Control Language): GRANT, REVOKE

TCL (Transaction Control Language): COMMIT, ROLLBACK, SAVEPOINT

Example:

```
CREATE TABLE employees (
   emp_id NUMBER PRIMARY KEY,
   name VARCHAR2(50),
   salary NUMBER
);
```

2. What is the difference between DELETE, TRUNCATE, and DROP?

Answer:

DELETE → Removes rows from a table, can use WHERE, logs entries, can ROLLBACK.

TRUNCATE → Removes all rows, faster, no WHERE, minimal logging, cannot ROLLBACK.

DROP → Removes entire table structure + data.

Example:

```
DELETE FROM employees WHERE emp_id = 101;
TRUNCATE TABLE employees;
DROP TABLE employees;
```

3. What are Joins? Explain types with examples.

Answer:

A JOIN is used to fetch data from multiple tables.

INNER JOIN → Common records from both tables.

LEFT JOIN → All records from left table + matching from right.

RIGHT JOIN → All records from right table + matching from left.

FULL OUTER JOIN → All records from both tables.

CROSS JOIN → Cartesian product.

Example:

```
SELECT e.emp_id, e.name, d.dname
FROM employees e
INNER JOIN department d ON e.dept_id = d.dept_id;
```

4. What is a Primary Key and a Foreign Key?

Answer:

Primary Key → Uniquely identifies each record (cannot be NULL).

Foreign Key → References primary key of another table, ensures referential integrity.

Example:

```
CREATE TABLE department (
    dept_id NUMBER PRIMARY KEY,
    dname VARCHAR2(50)
);

CREATE TABLE employees (
    emp_id NUMBER PRIMARY KEY,
    name VARCHAR2(50),
    dept_id NUMBER,
    CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);
```

5. What is the difference between WHERE and HAVING?

Answer:

WHERE → Filters rows before grouping.

HAVING → Filters groups after aggregation.

Example:

```
SELECT dept_id, COUNT(*)
FROM employees
WHERE salary > 30000
GROUP BY dept_id
HAVING COUNT(*) > 5;
```

6. What are Views in Oracle?

Answer:

A View is a virtual table based on the result of a SQL query.

Example:

```
CREATE VIEW emp_dept_view AS
SELECT e.name, d.dname
FROM employees e
JOIN department d ON e.dept_id = d.dept_id;
```

7. What are Indexes? Types in Oracle?

Answer:

Indexes improve query performance.

B-tree Index (default)

Bitmap Index (used in low-cardinality columns)

Unique Index

Composite Index

Example:

```
CREATE INDEX idx_emp_name ON employees(name);
```

8. What is Normalization? Explain 1NF, 2NF, 3NF.

Answer:

Normalization reduces redundancy.

1NF: Atomic values (no repeating groups).

2NF: 1NF + no partial dependency.

3NF: 2NF + no transitive dependency.

9. What is a Subquery? What are types?

Answer:

Single-row subquery → Returns one value.

Multi-row subquery → Returns multiple values.

Correlated subquery → Executes for each row of outer query.

Example:

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

10. What are Sequences in Oracle?

Answer:

A Sequence generates unique numbers.

Example:

```
CREATE SEQUENCE emp_seq START WITH 1 INCREMENT BY 1;
INSERT INTO employees (emp_id, name) VALUES (emp_seq.NEXTVAL, 'John');
```

11. What is a Trigger? Give an example.

Answer:

A Trigger automatically executes on certain events (INSERT, UPDATE, DELETE).

Example:

```
CREATE OR REPLACE TRIGGER emp_audit_trg
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('New employee inserted: ' || :NEW.name);
END;
```

12. What is a Cursor? Explain explicit and implicit.

Answer:

Implicit cursor: Automatically created for single-row queries.

Explicit cursor: Manually declared for multi-row queries.

Example:

```
DECLARE
  CURSOR emp_cur IS SELECT name, salary FROM employees;
  v_name employees.name%TYPE;
  v_salary employees.salary%TYPE;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO v_name, v_salary;
```

```
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_name || ' earns ' || v_salary);
   END LOOP;
   CLOSE emp_cur;
END;
```

13. What are Transactions in Oracle?

Answer:

Transaction = group of SQL statements executed as one unit.

COMMIT → Save changes permanently.

ROLLBACK → Undo changes.

SAVEPOINT → Mark point in transaction to rollback partially.

Example:

```
UPDATE employees SET salary = salary + 5000 WHERE emp_id = 101;
SAVEPOINT sp1;
UPDATE employees SET salary = salary + 2000 WHERE emp_id = 102;
ROLLBACK TO sp1;  -- Undo only second update
COMMIT;
```

14. Difference between UNION and UNION ALL?

Answer:

UNION → Returns unique records.

UNION ALL → Returns all including duplicates.

磋 When to Use UNION vs UNION ALL

Operator	Removes Duplicates	Faster	When to Use
UNION	✔ Yes	✖ Slower	When you need unique results
UNION ALL	✖ No	✔ Faster	When duplicates are allowed

Example:

```
SELECT name FROM employees

UNION

SELECT name FROM managers;
```

15. How do you optimize slow queries in Oracle?

Answer:

Use proper indexes.

Avoid SELECT *.

Use EXPLAIN PLAN to analyze.

Partition large tables.

Avoid correlated subqueries if JOIN can be used.

Use bind variables instead of literals.

Example (EXPLAIN PLAN):

```
EXPLAIN PLAN FOR
SELECT * FROM employees WHERE name = 'John';

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

. Find the second highest salary from the employees table.

Answer:

```
-- Using Subquery
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);

-- Using RANK()
SELECT salary
FROM (
   SELECT salary, RANK() OVER (ORDER BY salary DESC) rnk
   FROM employees
)
WHERE rnk = 2;
```

2. Find Nth highest salary (say 3rd highest).

Answer:

```
SELECT salary
FROM (
   SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) rnk
   FROM employees
)
WHERE rnk = 3;
```

3. Find employees with duplicate salaries.

Answer:

```
SELECT salary, COUNT(*) AS cnt
FROM employees
GROUP BY salary
HAVING COUNT(*) > 1;
```

4. Find the first 5 highest salaries.

Answer:

```
SELECT *
FROM (
   SELECT name, salary, DENSE_RANK() OVER (ORDER BY salary DESC) rnk
   FROM employees
)
WHERE rnk <= 5;
```

5. Find employees who do not belong to any department.

Answer:

```
SELECT e.emp_id, e.name
FROM employees e
LEFT JOIN department d ON e.dept_id = d.dept_id
WHERE d.dept_id IS NULL;
```

6. Find employees who earn more than the average salary.

Answer:

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

7. Display the department with the maximum number of employees.

Answer:

```
SELECT dept_id, COUNT(*) AS emp_count
FROM employees
GROUP BY dept_id
ORDER BY emp_count DESC
FETCH FIRST 1 ROWS ONLY;
```

8. Find employees with the same name.

Answer:

```
SELECT name, COUNT(*)

FROM employees

GROUP BY name

HAVING COUNT(*) > 1;
```

9. Retrieve the employee(s) with the highest salary in each department.

Answer:

```
SELECT emp_id, name, dept_id, salary

FROM (

  SELECT emp_id, name, dept_id, salary,

         RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) rnk

  FROM employees

)

WHERE rnk = 1;
```

10. Write a query to transpose rows into columns (Pivot).

Answer:

Suppose we want to show department-wise salary sum.

```
SELECT *

FROM (

  SELECT dept_id, salary FROM employees

)

PIVOT (
  SUM(salary) FOR dept_id IN (10 AS dept10, 20 AS dept20, 30 AS dept30)
);
```

11. How to fetch only odd/even rows?

Answer:

```
-- Odd rows
SELECT * FROM (
  SELECT e.*, ROWNUM rn FROM employees e
)
WHERE MOD(rn, 2) = 1;

-- Even rows
SELECT * FROM (
  SELECT e.*, ROWNUM rn FROM employees e
)
WHERE MOD(rn, 2) = 0;
```

12. Find the 3rd highest salary without using RANK/ROWNUM.

Answer:

```
SELECT MIN(salary) AS third_highest
FROM (
  SELECT DISTINCT salary
  FROM employees
  ORDER BY salary DESC
)
```

```
WHERE ROWNUM <= 3;
```

13. Find all employees who joined in the last 30 days.

Answer:

```
SELECT name, hire_date
FROM employees
WHERE hire_date >= SYSDATE - 30;
```

14. How to find employees whose names start and end with the same letter?

Answer:

```
SELECT name
FROM employees
WHERE LOWER(SUBSTR(name, 1, 1)) = LOWER(SUBSTR(name, -1, 1));
```

15. Find the highest paid employee in the company.

Answer:

```
SELECT *
FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees);
```

Basic DBMS Questions

What is DBMS? How is it different from RDBMS?

顲 Answer:

DBMS (Database Management System) → Software to store, manage, and retrieve data (e.g., file system-based, hierarchical).

RDBMS (Relational DBMS) → Stores data in tables with relationships (MySQL, Oracle, PostgreSQL). Supports normalization, ACID properties.

What are the different types of keys in DBMS?

顲 Answer:

Primary Key → Unique + Not Null (e.g., EmployeeID).

Foreign Key → References a primary key in another table.

Candidate Key → All possible unique keys.

Composite Key → Key made of 2+ columns.

Alternate Key → Candidate key not chosen as primary.

撤 Example:

```
CREATE TABLE Employee (
  EmpID INT PRIMARY KEY,
  EmpName VARCHAR(50),
  DeptID INT,
  FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);
```

What are the different types of relationships in DBMS?

顏 Answer:

One-to-One

One-to-Many (most common)

Many-to-Many (resolved using a junction table)

 SQL-Based Questions (Infosys loves practical queries)

What is the difference between DDL, DML, and DCL commands?

顏 Answer:

DDL (Data Definition Language): CREATE, ALTER, DROP

DML (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE

DCL (Data Control Language): GRANT, REVOKE

Write a SQL query to fetch the 2nd highest salary from Employee table.

顏 Answer:

```
SELECT MAX(Salary)
FROM Employee
WHERE Salary < (SELECT MAX(Salary) FROM Employee);
```

What is the difference between TRUNCATE, DELETE, and DROP?

顏 Answer:

DELETE → Removes rows (can use WHERE, rollback possible).

TRUNCATE → Removes all rows (faster, cannot rollback).

DROP → Deletes the table structure itself.

What is a Join? Explain its types with examples.

頻 Answer:

INNER JOIN → Common records in both tables.

LEFT JOIN → All records from left, matching from right.

RIGHT JOIN → All records from right, matching from left.

FULL JOIN → All records from both tables.

撤 Example:

```
SELECT e.EmpName, d.DeptName
FROM Employee e
INNER JOIN Department d
ON e.DeptID = d.DeptID;
```

 Intermediate DBMS Questions

What is Normalization? Explain its types.

頻 Answer:

Process of removing redundancy and ensuring data integrity.

1NF → Atomic values only

2NF → 1NF + No partial dependency

3NF → 2NF + No transitive dependency

BCNF → Stronger form of 3NF

What are Transactions in DBMS? Explain ACID properties.

頴 Answer:

Transaction → Unit of work (INSERT/UPDATE/DELETE).

ACID:

Atomicity → All or nothing

Consistency → Data valid after commit

Isolation → Transactions don't affect each other

Durability → Once committed, data is permanent

What is the difference between Clustered and Non-Clustered Index?

頴 Answer:

Clustered Index → Rearranges physical data (only 1 per table, usually on primary key).

Non-Clustered Index → Separate structure, points to data (multiple allowed).

 Advanced / Scenario-Based Infosys Questions

How do you optimize a slow SQL query?

 Answer:

Create proper indexes

Avoid SELECT *

Use joins instead of subqueries (if better)

Check execution plan

Partitioning large tables

What is Deadlock in DBMS? How do you prevent it?

 Answer:

Deadlock → Two transactions waiting for each other's resources.

Prevention:

Use same order of locking

Set transaction timeouts

Use smaller transactions

What is the difference between OLTP and OLAP systems?

 Answer:

OLTP (Online Transaction Processing) → Handles daily transactions (banking system).

OLAP (Online Analytical Processing) → Handles analytics/reporting (data warehouse).

Explain Stored Procedure, Function, and Trigger.

頴 Answer:

Stored Procedure → Precompiled SQL code, can return multiple values.

Function → Returns single value, used in queries.

Trigger → Executes automatically when an event occurs (INSERT, UPDATE, DELETE).

撤 Example:

```
CREATE TRIGGER trg_after_insert
AFTER INSERT ON Employee
FOR EACH ROW
BEGIN
  INSERT INTO AuditLog VALUES (NEW.EmpID, NOW());
END;
```

 Infosys-Specific Tips

Infosys usually asks scenario-based SQL → e.g. "Find duplicate records", "Get nth highest salary", "Find employees without managers".

They check if you know concepts + can write queries quickly.

Sometimes they combine Java + DBMS → "How would you connect Java to DB?" (Answer: JDBC, Spring Data JPA, Hibernate ORM).

1)What is difference between UNION and UNION ALL operator?

2)Explain Collections with types and where it use?

3)What makes materialized view different than simple or complex view?

4)what are windows functions?

5)What are ACID properties in sql?

6)The procedure is taking more time to execute. How to optimize it?

7) Are you heard about DBMS_PROFILER? Tell me its operation uses?

8)Can we update view? if update then how ,if not then why ?explain in brief

9)What is Bulk Upload and For all?

10) What is Partitioning of table?its types?

11)What is Index? can we use function on indexed column?

12) give me examples of package calling procedure and procedure calling packages?

13)What is call by reference and call by value? can you distinguish them?

Apart from theory part they ask Query and scenario based questions listed below:-

1)get the position of 'K' in 'OMKAR'?

2)How to fetch duplicate record without group by clause?

3)Now delete fetch record using CTE?

4) Imagine you are working on a complex SQL query that joins multiple tables and takes several minutes to execute. This query is critical for generating daily reports, but its slow performance is causing delay. Could you describe your approach to optimize the query? What tool and technique would you use to improve its performance?

5)Tell me the Logical execution of select statement.

Let's assume we have these two tables:

Employee

emp_id	name	dept_id	salary	joining_date
1	John	10	50000	2021-05-12

2	Alice	20	60000	2022-02-18
3	Bob	10	55000	2020-12-01
4	Sarah	30	45000	2023-06-10
5	David	20	70000	2021-09-25

Department

dept_id	dept_name
10	IT
20	Finance
30	HR

1. 透 Basic Queries

1.1 Select all employees

```
SELECT * FROM Employee;
```

1.2 Get names of employees in IT department

```
SELECT name
FROM Employee
WHERE dept_id = 10;
```

1.3 Get employees with salary > 50,000

```
SELECT name, salary
FROM Employee
WHERE salary > 50000;
```

1.4 Sort employees by salary (highest first)

```
SELECT name, salary
FROM Employee
ORDER BY salary DESC;
```

1.5 Find unique department IDs

```
SELECT DISTINCT dept_id
FROM Employee;
```

1.6 Get top 3 highest-paid employees

```
SELECT name, salary
FROM Employee
ORDER BY salary DESC
LIMIT 3;
```

(For Oracle use FETCH FIRST 3 ROWS ONLY.)

2. □ Aggregate Functions with GROUP BY & HAVING

2.1 Find total salary paid in each department

```
SELECT dept_id, SUM(salary) AS total_salary
FROM Employee
GROUP BY dept_id;
```

2.2 Find average salary in IT department

```
SELECT AVG(salary) AS avg_salary
FROM Employee
WHERE dept_id = 10;
```

2.3 Find departments having more than 1 employee

```
SELECT dept_id, COUNT(*) AS emp_count
FROM Employee
GROUP BY dept_id
HAVING COUNT(*) > 1;
```

3. □ Joins

3.1 Show employee names with their department names

SELECT e.name, d.dept_name

FROM Employee e

INNER JOIN Department d ON e.dept_id = d.dept_id;

3.2 Get all employees even if they don’t have a department

SELECT e.name, d.dept_name

FROM Employee e

LEFT JOIN Department d ON e.dept_id = d.dept_id;

3.3 Find departments that have no employees

SELECT d.dept_name

FROM Department d

LEFT JOIN Employee e ON e.dept_id = d.dept_id

WHERE e.emp_id IS NULL;

4. [illegible] Subqueries

4.1 Find employees earning more than the average salary

SELECT name, salary

FROM Employee

WHERE salary > (SELECT AVG(salary) FROM Employee);

4.2 Find second highest salary

SELECT MAX(salary)

FROM Employee

WHERE salary < (SELECT MAX(salary) FROM Employee);

4.3 Find employees who joined after ‘John’

```
SELECT name
FROM Employee
WHERE joining_date > (
   SELECT joining_date
   FROM Employee
   WHERE name = 'John'
);
```

5. □ Window Functions

(Very common in 2+ year interviews)

5.1 Rank employees by salary

```
SELECT name, salary,
     RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM Employee;
```

5.2 Find the highest salary per department

```
SELECT name, dept_id, salary
FROM (
   SELECT name, dept_id, salary,
        RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rnk
   FROM Employee
) AS ranked
WHERE rnk = 1;
```

5.3 Calculate running total of salaries

```
SELECT name, salary,
     SUM(salary) OVER (ORDER BY emp_id) AS running_total
```

```
FROM Employee;
```

6. 讚 CTE (Common Table Expressions)

Find employees whose salary is above their department average

```
WITH dept_avg AS (
   SELECT dept_id, AVG(salary) AS avg_salary
   FROM Employee
   GROUP BY dept_id
)
SELECT e.name, e.salary, e.dept_id
FROM Employee e
JOIN dept_avg d ON e.dept_id = d.dept_id
WHERE e.salary > d.avg_salary;
```

7. 廳 DML & DDL Questions

7.1 Insert a new employee

```
INSERT INTO Employee (emp_id, name, dept_id, salary, joining_date)
VALUES (6, 'Emma', 30, 48000, '2023-09-01');
```

7.2 Update salary of an employee

```
UPDATE Employee
SET salary = 75000
WHERE name = 'Alice';
```

7.3 Delete employees in HR department

```
DELETE FROM Employee
WHERE dept_id = 30;
```

7.4 Create a new table

```
CREATE TABLE Projects (
    proj_id INT PRIMARY KEY,
    proj_name VARCHAR(50),
    dept_id INT
);
```

8. 👉 Real-World Problem Queries

8.1 Find highest-paid employee in each department

```
SELECT e.name, e.salary, e.dept_id
FROM Employee e
WHERE e.salary = (
    SELECT MAX(salary)
    FROM Employee
    WHERE dept_id = e.dept_id
);
```

8.2 Find department with maximum employees

```
SELECT dept_id, COUNT(*) AS emp_count
FROM Employee
GROUP BY dept_id
ORDER BY emp_count DESC
LIMIT 1;
```

8.3 Retrieve employees who joined in 2022

```
SELECT name, joining_date
FROM Employee
WHERE YEAR(joining_date) = 2022;
```

there was employer table details emplyoid ,joining of date, gender and etc...so i have to write a query to count number of male and female from the table how i will write the orcale query?

Good question 跤

Let’s say your table name is EMPLOYEE_DETAILS, and it has a column GENDER that stores either 'Male' or 'Female'.

Here’s how you can write the Oracle SQL query to count the number of males and females

✅ Query 1 — Using GROUP BY

```
SELECT gender, COUNT(*) AS total_count

FROM employee_details

GROUP BY gender;
```

透 Explanation:

GROUP BY gender groups all rows by each gender value.

COUNT(*) counts how many rows are in each group.s

This will return something like:

The SQL ANY and ALL Operators

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

The SQL ANY Operator

The ANY operator:

returns a boolean value as a result

returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

ANY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

Note: The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

The SQL ALL Operator

The ALL operator:

returns a boolean value as a result

returns TRUE if ALL of the subquery values meet the condition

is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

ALL Syntax With WHERE or HAVING

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Syntax

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

NOT NULL - Ensures that a column cannot have a NULL value

UNIQUE - Ensures that all values in a column are different

PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

FOREIGN KEY - Prevents actions that would destroy links between tables

CHECK - Ensures that the values in a column satisfies a specific condition

DEFAULT - Sets a default value for a column if no value is specified

CREATE INDEX - Used to create and retrieve data from the database very quickly

What Are Constraints?

Constraints are rules applied on table columns to restrict invalid data — ensuring data accuracy, integrity, and reliability.

Examples of constraints:

NOT NULL

UNIQUE

PRIMARY KEY

FOREIGN KEY

CHECK

DEFAULT

□ 1. Column-Level Constraints

➡ These are applied while defining a column — i.e., right after the column name in the CREATE TABLE statement.

➡ The constraint applies only to that column.

✅ Example:

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,        -- Column-level constraint
   emp_name VARCHAR(50) NOT NULL, -- Column-level constraint
   age INT CHECK (age >= 18),     -- Column-level constraint
   salary DECIMAL(10,2) DEFAULT 25000.00
);
```

⅜ Explanation:

PRIMARY KEY — ensures emp_id is unique and not null.

NOT NULL — ensures emp_name cannot be empty.

CHECK (age >= 18) — ensures age is at least 18.

DEFAULT — gives a default salary value if none provided.

✅ All of these constraints are defined at the column level.

 2. Table-Level Constraints

➡ These are defined after all columns are declared (i.e., at the end of the table definition).

➡ They can apply to one or more columns.

✅ Example:

```
CREATE TABLE orders (
   order_id INT,
   product_id INT,
   customer_id INT,
   quantity INT,
   PRIMARY KEY (order_id),                    -- Table-level constraint
   FOREIGN KEY (customer_id) REFERENCES customers(customer_id),  -- Table-level
   CHECK (quantity > 0)                       -- Table-level constraint
);
```

⅜ Explanation:

PRIMARY KEY (order_id) — defined after all columns → table-level.

FOREIGN KEY (customer_id) — references another table's column → table-level.

CHECK (quantity > 0) — applies to column(s) at the table level.

 3. Difference Between Column-Level & Table-Level Constraints

Feature	Column-Level Constraint	Table-Level Constraint
Placement	Immediately after the column name	After all columns are defined
Applies to	Single column only	One or more columns
Common examples	NOT NULL, DEFAULT, simple CHECK	PRIMARY KEY, FOREIGN KEY, composite CHECK
When used	When the constraint affects only one column	When the constraint involves multiple columns or references other tables

易 Example with Both Types Together

```
CREATE TABLE students (
   student_id INT NOT NULL,         -- Column-level
   first_name VARCHAR(50) NOT NULL, -- Column-level
   last_name VARCHAR(50),
   age INT CHECK (age >= 18),       -- Column-level
   course_id INT,
   PRIMARY KEY (student_id),        -- Table-level
   FOREIGN KEY (course_id) REFERENCES courses(course_id) -- Table-level
);
```

責 Interview Tip:

You can say:

"Column-level constraints are defined directly after a column definition and apply to that single column, while table-level constraints are defined at the end of the table definition and can apply to one or multiple columns — for example, composite primary keys or foreign keys."

What Are Constraints?

Constraints are rules applied on table columns to restrict invalid data — ensuring data accuracy, integrity, and reliability.

Examples of constraints:

- `NOT NULL`

- `UNIQUE`
- `PRIMARY KEY`
- `FOREIGN KEY`
- `CHECK`
- `DEFAULT`

---

1. **`Column-Level Constraints`**

➡ These are applied **`while defining a column`** — i.e., right after the column name in the `CREATE TABLE` statement.
➡ The constraint applies **`only to that column`**.

✅ Example:

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,         -- Column-level constraint
    emp_name VARCHAR(50) NOT NULL, -- Column-level constraint
    age INT CHECK (age >= 18),     -- Column-level constraint
    salary DECIMAL(10,2) DEFAULT 25000.00
);
```

⅔ Explanation:

- `PRIMARY KEY` — ensures `emp_id` is unique and not null.
- `NOT NULL` — ensures `emp_name` cannot be empty.
- `CHECK (age >= 18)` — ensures age is at least 18.
- `DEFAULT` — gives a default salary value if none provided.

✅ **`All of these constraints are defined at the column level`**.

---

2. **`Table-Level Constraints`**

➡ These are defined **`after all columns are declared`** (i.e., at the end of the table definition).
➡ They can apply to **`one or more columns`**.

✅ Example:

```
CREATE TABLE orders (
    order_id INT,
    product_id INT,
    customer_id INT,
    quantity INT,
    PRIMARY KEY (order_id),                          -- Table-level constraint
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id),  -- Table-level
    CHECK (quantity > 0)                             -- Table-level constraint
);
```

⅔ Explanation:

- `PRIMARY KEY (order_id)` — defined **after** all columns → **table-level**.
- `FOREIGN KEY (customer_id)` — references another table’s column → table-level.
- `CHECK (quantity > 0)` — applies to column(s) at the table level.

---

3. Difference Between Column-Level & Table-Level Constraints

Feature	Column-Level Constraint	Table-Level Constraint
Placement	Immediately after the column name	After all columns are defined
Applies to	Single column only	One or more columns
Common examples	NOT NULL, DEFAULT, simple CHECK	PRIMARY KEY, FOREIGN KEY, composite CHECK
When used	When the constraint affects only one column	When the constraint involves multiple columns or references other tables

---

易 Example with Both Types Together

```
CREATE TABLE students (
    student_id INT NOT NULL,         -- Column-level
    first_name VARCHAR(50) NOT NULL, -- Column-level
    last_name VARCHAR(50),
    age INT CHECK (age >= 18),       -- Column-level
    course_id INT,
    PRIMARY KEY (student_id),        -- Table-level
    FOREIGN KEY (course_id) REFERENCES courses(course_id) -- Table-level
);
```

責 Interview Tip:

You can say:

> “Column-level constraints are defined directly after a column definition and apply to that single column, while table-level constraints are defined at the end of the table definition and can apply to one or multiple columns — for example, composite primary keys or foreign keys.”

Would you like me to show a small query to alter (add/remove) these constraints after table creation (e.g., using ALTER TABLE)? That’s another frequent follow-up question.

You said:

explain all the constraints in detail with examples in mysql.

ChatGPT said:

Perfect 敃 — this is one of the most frequently asked interview topics in SQL.
Let’s go step by step and explain all constraints in MySQL clearly, with syntax, examples, and output.

𥫱 SQL Constraints in MySQL

里 What are Constraints?

Constraints are rules applied to columns in a table to ensure data accuracy, consistency, and reliability.
They restrict invalid data from being entered into the database.

Types of Constraints in MySQL

Constraint	Purpose
NOT NULL	Ensures a column cannot have a NULL (empty) value
UNIQUE	Ensures all values in a column are unique
PRIMARY KEY	Uniquely identifies each record (combines NOT NULL + UNIQUE)
FOREIGN KEY	Enforces a relationship between two tables
CHECK	Ensures that data in a column meets a specific condition
DEFAULT	Assigns a default value when no value is provided
AUTO_INCREMENT	Automatically generates unique numbers for each record (used with primary key)

1️⃣ NOT NULL Constraint

顲 Ensures a column must have a value; it cannot be left empty.

```
CREATE TABLE employees (
    emp_id INT,
    emp_name VARCHAR(50) NOT NULL,
    department VARCHAR(30)
);
```

⅜ Example:

```
INSERT INTO employees (emp_id, department)
VALUES (101, 'HR');
```

✕ Error: Column 'emp_name' cannot be null

✅ Correct:

```
INSERT INTO employees VALUES (101, 'Nisha', 'HR');
```

2️⃣ UNIQUE Constraint

頴 Ensures that all values in a column are different (no duplicates allowed).

```
CREATE TABLE employees (
    emp_id INT UNIQUE,
    emp_name VARCHAR(50),
    email VARCHAR(50) UNIQUE
);
```

% Example:

```
INSERT INTO employees VALUES (101, 'Nisha', 'nisha@gmail.com');
INSERT INTO employees VALUES (102, 'Ravi', 'nisha@gmail.com');
```

✕ Error: Duplicate entry 'nisha@gmail.com' for key email

✅ Every email must be unique.

3️⃣ PRIMARY KEY Constraint

頴 A combination of NOT NULL and UNIQUE.
It uniquely identifies each row in a table.
Each table can have only one primary key.

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    department VARCHAR(30)
);
```

% Example:

```
INSERT INTO employees VALUES (101, 'Nisha', 'HR');
INSERT INTO employees VALUES (101, 'Ravi', 'IT');
```

✕ Error: Duplicate entry '101' for key PRIMARY.

✅ emp_id uniquely identifies each employee.

Composite Primary Key (Multiple Columns)

If one column cannot uniquely identify a row, use multiple columns:

```
CREATE TABLE student_course (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Now the combination of both columns must be unique.

---

4 FOREIGN KEY Constraint

顤 Creates a link between two tables.

The foreign key in one table refers to the primary key in another table.

Example:

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

⅜ Explanation:

- You can insert a dept_id in employees only if it already exists in the departments table.

✅ Valid:

```
INSERT INTO departments VALUES (1, 'HR');
INSERT INTO employees VALUES (101, 'Nisha', 1);
```

✕ Invalid:

```
INSERT INTO employees VALUES (102, 'Ravi', 5);
```

Error → Cannot add or update a child row: a foreign key constraint fails

---

5 CHECK Constraint

顤 Ensures that data in a column meets a condition.

(Supported in MySQL 8.0+)

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    age INT CHECK (age >= 18)
);
```

✅ Valid:

```
INSERT INTO employees VALUES (101, 'Nisha', 25);
```

✕ Invalid:

```
INSERT INTO employees VALUES (102, 'Ravi', 15);
```

Error → Check constraint 'employees_chk_1' is violated

---

6 DEFAULT Constraint

顉 Provides a default value when no value is given during insert.

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    city VARCHAR(50) DEFAULT 'Pune'
);
```

✅ Example:

```
INSERT INTO employees (emp_id, emp_name)
VALUES (101, 'Nisha');

SELECT * FROM employees;
```

喆 Output:

emp_id	emp_name	city
101	Nisha	Pune

---

7 AUTO_INCREMENT Constraint

顉 Automatically generates a unique number (usually for primary keys).

```
CREATE TABLE employees (
    emp_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_name VARCHAR(50)
);
```

✅ Example:

```
INSERT INTO employees (emp_name) VALUES ('Nisha');
INSERT INTO employees (emp_name) VALUES ('Ravi');

SELECT * FROM employees;
```

喆 Output:

emp_id	emp_name
1	Nisha
2	Ravi

易 Summary Table

Constraint	Description	Example
NOT NULL	Column must have a value	emp_name VARCHAR(50) NOT NULL
UNIQUE	No duplicate values	email VARCHAR(50) UNIQUE
PRIMARY KEY	Uniquely identifies each	PRIMARY KEY(emp_id)

Constraint	Description	Example
	record	
FOREIGN KEY	References a key in another table	FOREIGN KEY(dept_id) REFERENCES departments(dept_id)
CHECK	Restricts values to a condition	CHECK (age >= 18)
DEFAULT	Assigns default value	DEFAULT 'Pune'
AUTO_INCREMENT	Generates sequence numbers	emp_id INT AUTO_INCREMENT

1) Column-level constraint — definition

A column-level constraint is declared as part of the column definition. It applies to the values stored in that one column only.

Common column-level constraints:

- NOT NULL — value must be present
- UNIQUE — values in this column must be unique (some DBMS allow inline UNIQUE here)
- PRIMARY KEY — designate this column as the primary key (usually for a single-column PK)
- CHECK — validate values for that single column (support varies by DBMS)
- DEFAULT — default value for the column

Column-level example

```
CREATE TABLE employees (
  emp_id     INT PRIMARY KEY,      -- column-level PRIMARY KEY
  first_name VARCHAR(50) NOT NULL, -- column-level NOT NULL
  email      VARCHAR(255) UNIQUE,  -- column-level UNIQUE
  age        INT CHECK (age >= 18) -- column-level CHECK (DBMS must support
CHECK)
);
```

In this example:

- emp_id primary key constraint is attached to that column only.
- first_name cannot be NULL.
- email must be unique among all rows.
- age check applies only to age.

2) Table-level constraint — definition

A table-level constraint is declared after the column definitions (inside the `CREATE TABLE` statement) or added later with `ALTER TABLE`. Table-level constraints can reference **`one or`**

`more columns` and are required when you want to express a constraint that involves multiple columns (composite keys, multi-column unique, multi-column check logic) or when declaring foreign keys that reference columns in another table.

Common table-level constraints:

- `PRIMARY KEY (col1, col2) — composite primary key`
- `UNIQUE (col1, col2) — composite unique constraint`
- `CHECK (expression involving multiple columns) — multi-column validation`
- `FOREIGN KEY (col) REFERENCES other_table(col) — foreign key`

Table-level examples

Composite primary key and foreign key example:

```
CREATE TABLE orders (
  order_id   INT,
  product_id INT,
  qty        INT NOT NULL,
  PRIMARY KEY (order_id, product_id)        -- table-level composite PK
);

CREATE TABLE order_items (
  order_id   INT,
  product_id INT,
  price      DECIMAL(10,2),
  FOREIGN KEY (order_id, product_id)        -- table-level FK referencing
composite PK
    REFERENCES orders (order_id, product_id)
);
```

Or a composite unique and a multi-column check:

```
CREATE TABLE registrations (
  event_id INT,
  user_id  INT,
  seat_no  INT,
  UNIQUE (event_id, seat_no),            -- table-level composite UNIQUE
  CHECK (seat_no >= 1 AND seat_no <= 100) -- table-level CHECK (involves seat_no
only here)
);
```

3) Differences summarized

- **Scope**
  - **Column-level: applies to one column.**
  - **Table-level: can apply to one or multiple columns (required for multi-column constraints).**
- **Syntax location**
  - **Column-level: declared inline with the column definition.**

    - **Table-level: declared after the column list in `CREATE TABLE` or via `ALTER TABLE`.**
- **Composite constraints**
    - **Column-level: cannot express composite (multi-column) constraints.**
    - **Table-level: used for composite primary keys, composite unique constraints, and composite foreign keys.**
- **Foreign keys**
    - **Commonly defined at table-level (especially if they reference more than one column).**
- **Naming**
    - **Table-level constraints are often named explicitly (`CONSTRAINT fk_name FOREIGN KEY (...)`) which helps with maintenance/error messages. Column-level inline constraints are often unnamed by default (DBMS will generate a name).**
- **Portability notes**
    - **Some older versions of certain DBMSs had limited support for `CHECK` or handled inline `UNIQUE` differently; if portability matters, check your database's docs (MySQL, PostgreSQL, SQL Server, Oracle have slightly different behaviors).**

4) Equivalents and when either can be used

- `NOT NULL must be column-level (there's no separate table-level NOT NULL syntax).`
- `PRIMARY KEY on a` **`single`** `column can be declared either inline (col INT PRIMARY KEY) or at table level (PRIMARY KEY(col)); both create the same effect for one column.`
- `UNIQUE for one column can be inline or table-level — functionally equivalent. For multiple columns you must use table-level.`
- `FOREIGN KEY is typically table-level.`

Same-effect example (single-column UNIQUE or PRIMARY KEY)

Inline:

```
username VARCHAR(50) UNIQUE
```

Table-level equivalent:

```
username VARCHAR(50),
UNIQUE (username)
```

5) How to add constraints later

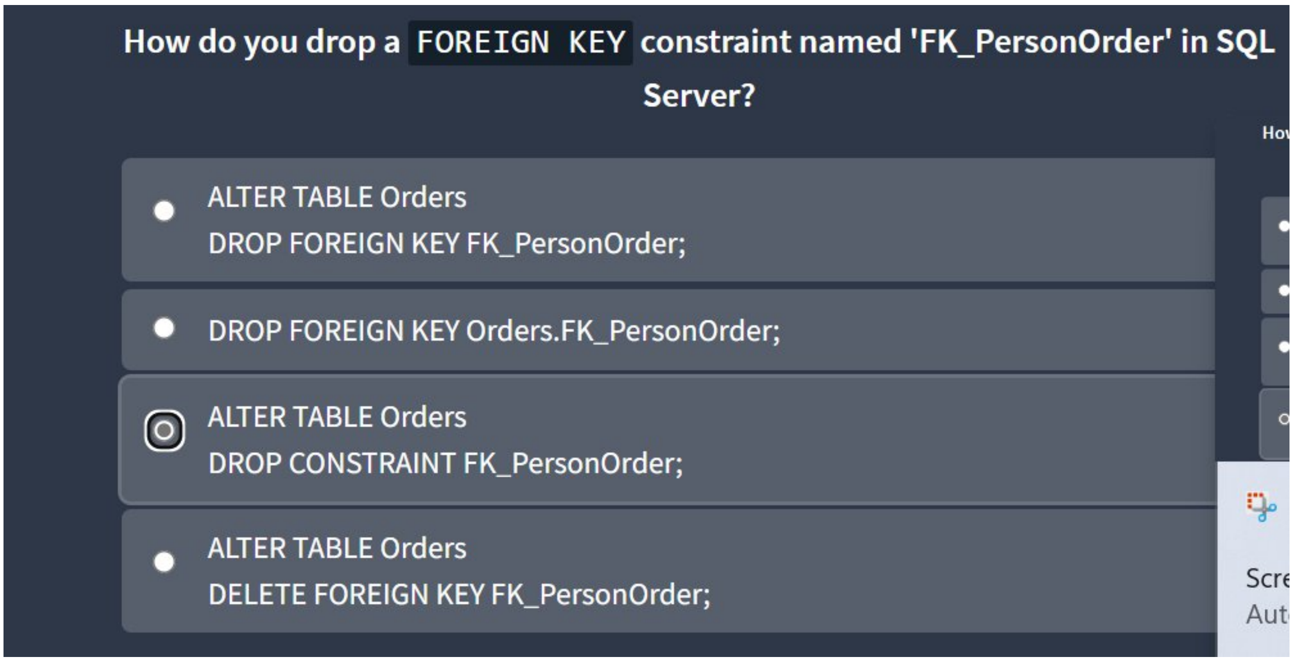
Use ALTER TABLE:

```
ALTER TABLE employees
  ADD CONSTRAINT emp_email_unique UNIQUE (email);

ALTER TABLE employees
  ADD CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES departments(id);
```

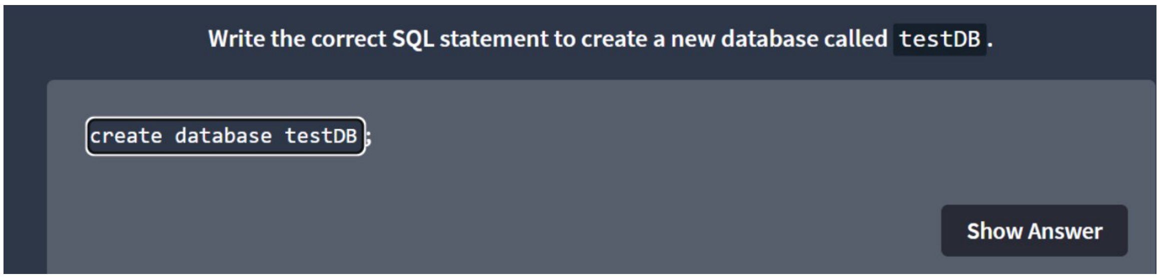
6) Practical guidelines

- Use column-level for simple, single-column rules (NOT NULL, DEFAULT, inline UNIQUE, single-column PRIMARY KEY) because it's concise and readable.
- Use table-level when rules need to:
    - cover multiple columns (composite keys/uniques),
    - define foreign keys (especially composite),
    - require explicit CONSTRAINT names for maintenance,
    - express multi-column CHECK logic.



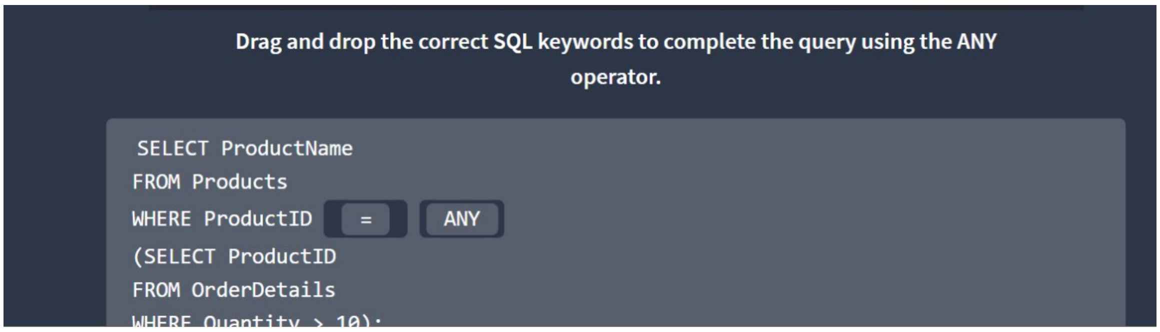
How do you drop a `FOREIGN KEY` constraint named 'FK_PersonOrder' in SQL Server?

- ALTER TABLE Orders
  DROP FOREIGN KEY FK_PersonOrder;
- DROP FOREIGN KEY Orders.FK_PersonOrder;
- [✓ ALTER TABLE Orders
  DROP CONSTRAINT FK_PersonOrder;]
- ALTER TABLE Orders
  DELETE FOREIGN KEY FK_PersonOrder;



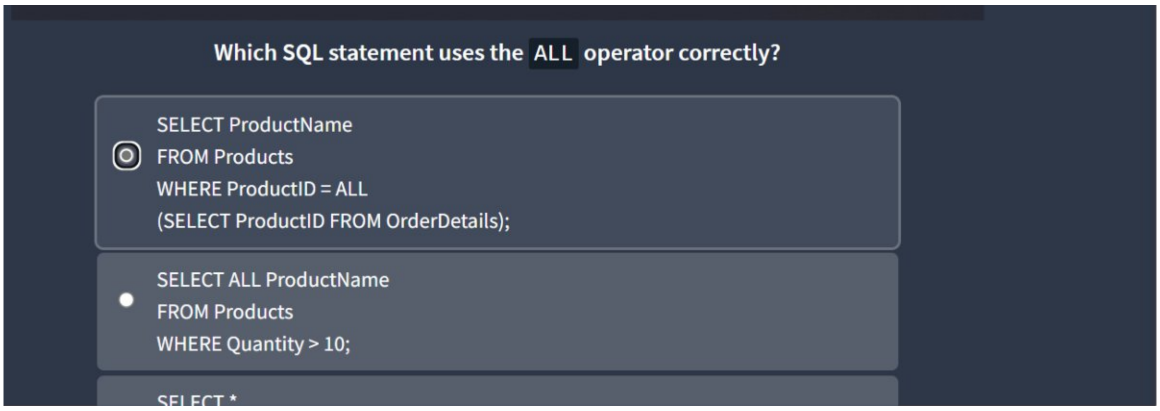
Write the correct SQL statement to create a new database called `testDB`.

```
create database testDB;
```



Drag and drop the correct SQL keywords to complete the query using the ANY operator.

```
SELECT ProductName
FROM Products
WHERE ProductID  =  ANY
(SELECT ProductID
FROM OrderDetails
WHERE Quantity > 10);
```

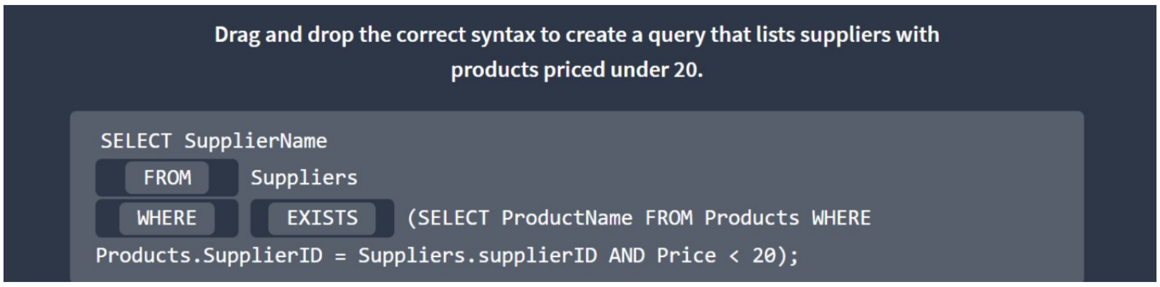


Which SQL statement uses the ALL operator correctly?

[✓ SELECT ProductName
FROM Products
WHERE ProductID = ALL
(SELECT ProductID FROM OrderDetails);]

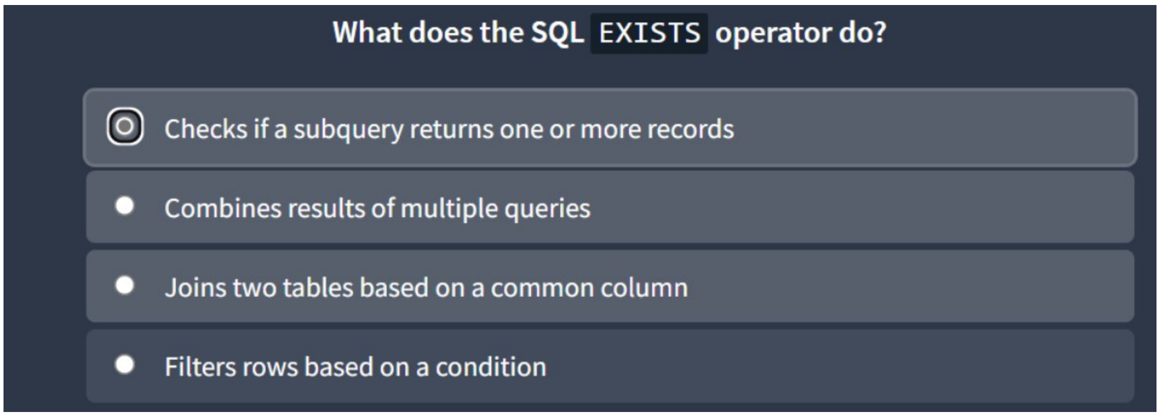
● SELECT ALL ProductName
FROM Products
WHERE Quantity > 10;

SELECT *



Drag and drop the correct syntax to create a query that lists suppliers with products priced under 20.

```
SELECT SupplierName
FROM  Suppliers
WHERE  EXISTS  (SELECT ProductName FROM Products WHERE
Products.SupplierID = Suppliers.supplierID AND Price < 20);
```



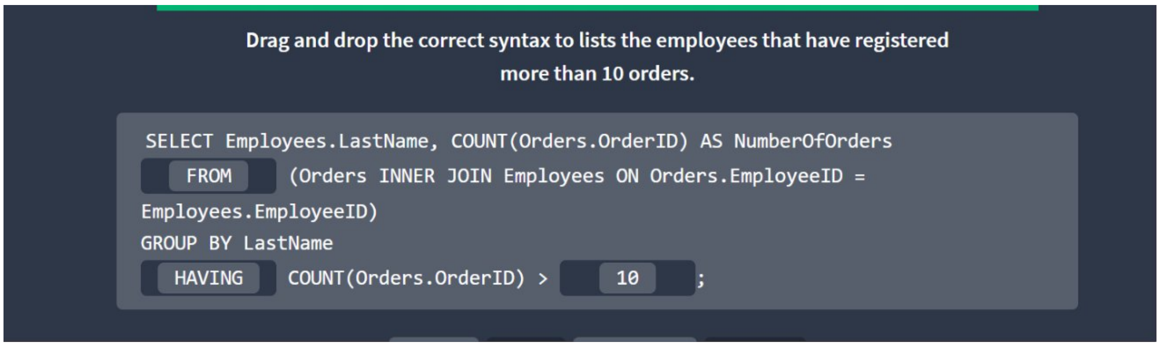
What does the SQL EXISTS operator do?

[✓ Checks if a subquery returns one or more records]

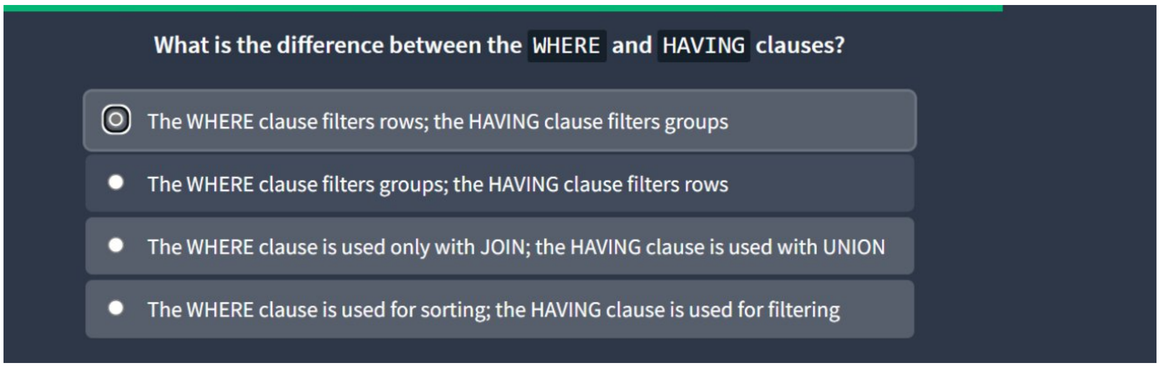
● Combines results of multiple queries

● Joins two tables based on a common column

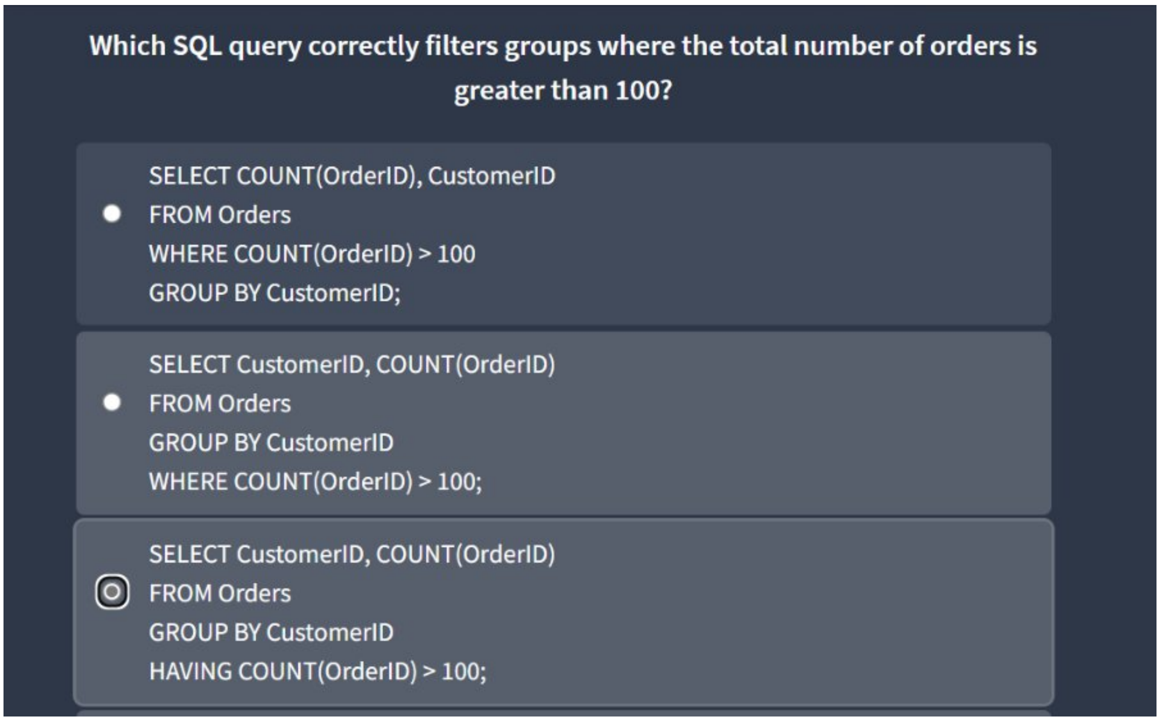
● Filters rows based on a condition


Drag and drop the correct syntax to lists the employees that have registered more than 10 orders.

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM (Orders INNER JOIN Employees ON Orders.EmployeeID =
Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10 ;
```

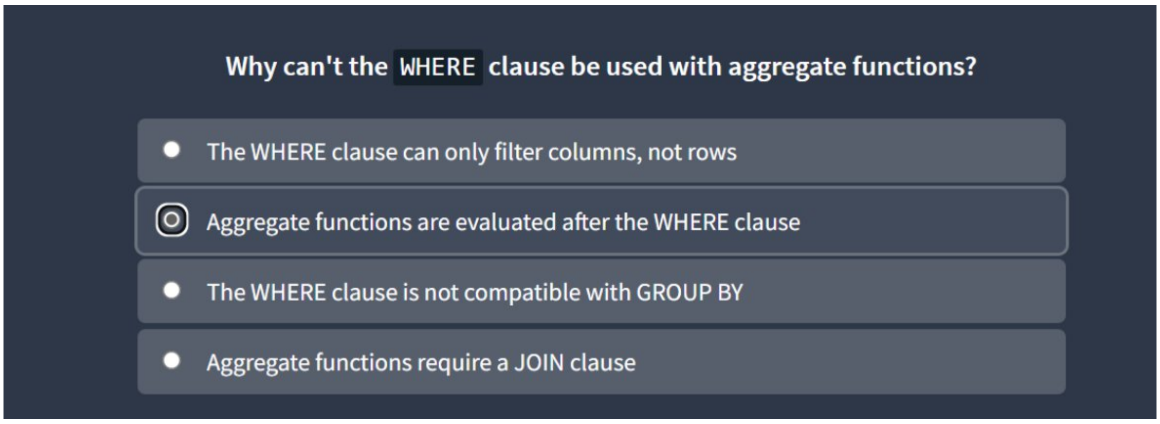

What is the difference between the `WHERE` and `HAVING` clauses?

- [✓ The WHERE clause filters rows; the HAVING clause filters groups]
- The WHERE clause filters groups; the HAVING clause filters rows
- The WHERE clause is used only with JOIN; the HAVING clause is used with UNION
- The WHERE clause is used for sorting; the HAVING clause is used for filtering


Which SQL query correctly filters groups where the total number of orders is greater than 100?

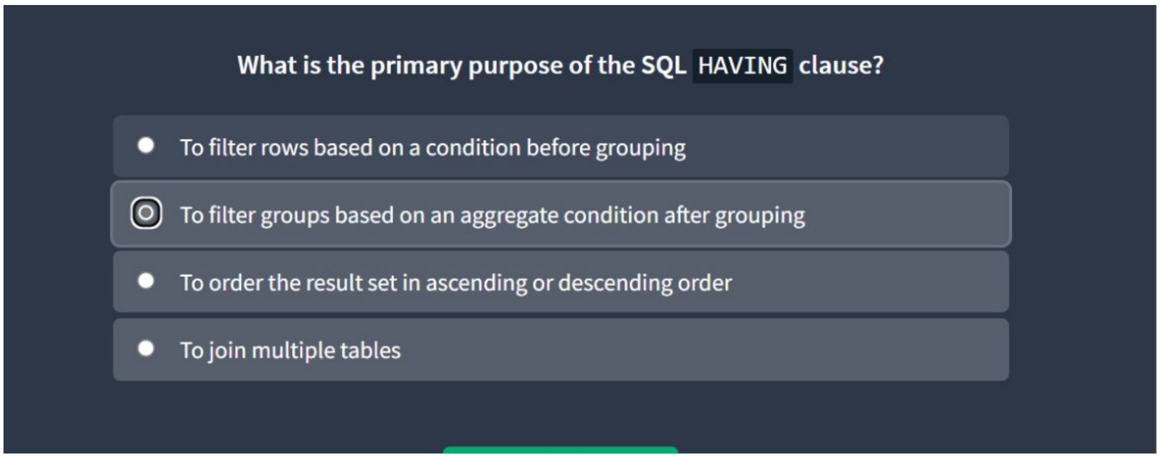
-
  ```
  SELECT COUNT(OrderID), CustomerID
  FROM Orders
  WHERE COUNT(OrderID) > 100
  GROUP BY CustomerID;
  ```

-
  ```
  SELECT CustomerID, COUNT(OrderID)
  FROM Orders
  GROUP BY CustomerID
  WHERE COUNT(OrderID) > 100;
  ```
- [✓]
  ```
  SELECT CustomerID, COUNT(OrderID)
  FROM Orders
  GROUP BY CustomerID
  HAVING COUNT(OrderID) > 100;
  ```



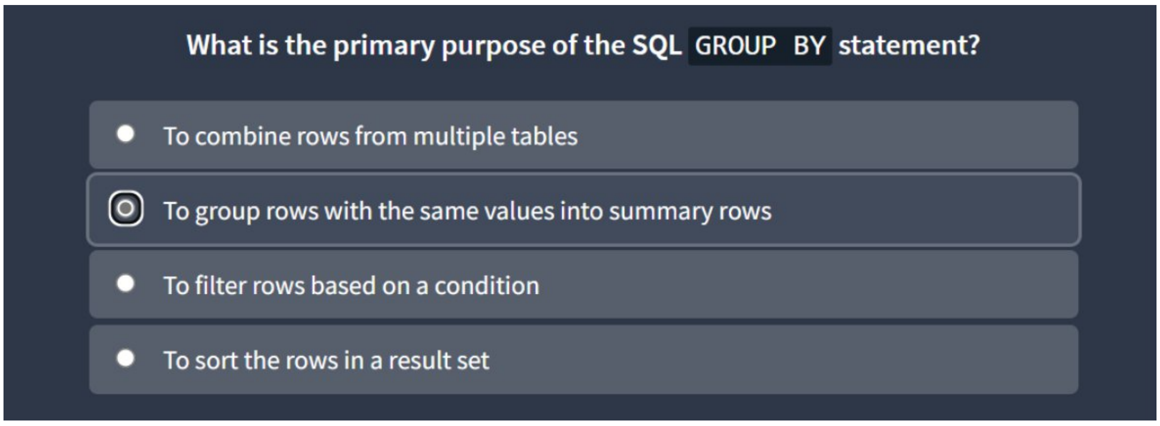
Why can't the `WHERE` clause be used with aggregate functions?

- The WHERE clause can only filter columns, not rows
- [✓ Aggregate functions are evaluated after the WHERE clause]
- The WHERE clause is not compatible with GROUP BY
- Aggregate functions require a JOIN clause



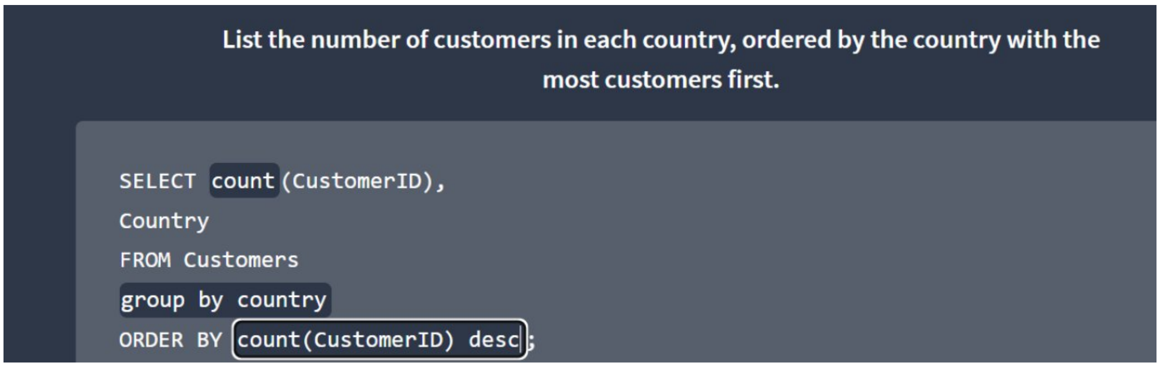
What is the primary purpose of the SQL `HAVING` clause?

- To filter rows based on a condition before grouping
- [✓ To filter groups based on an aggregate condition after grouping]
- To order the result set in ascending or descending order
- To join multiple tables



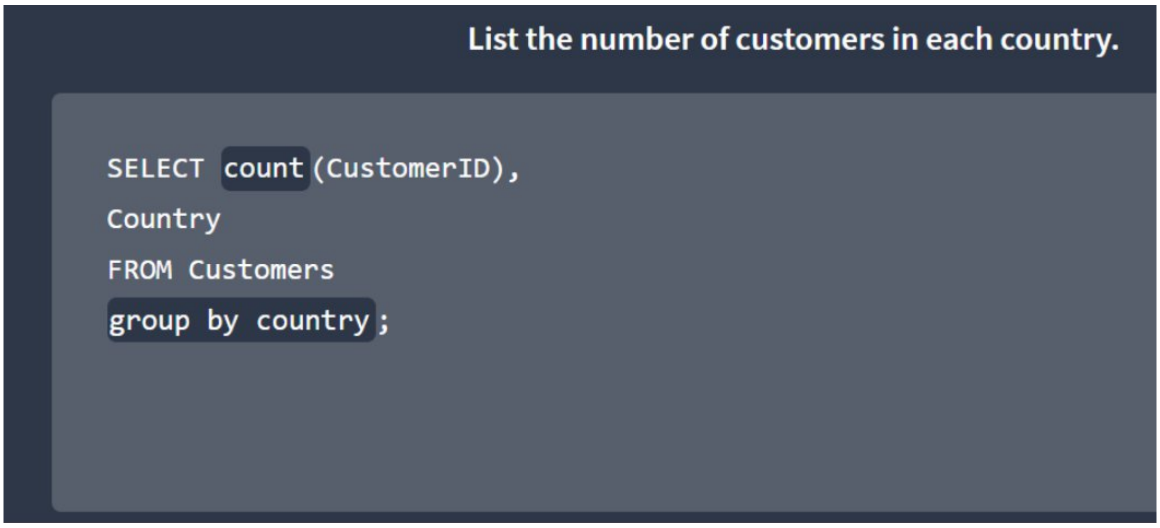
What is the primary purpose of the SQL `GROUP BY` statement?

- To combine rows from multiple tables
- [✓ To group rows with the same values into summary rows]
- To filter rows based on a condition
- To sort the rows in a result set

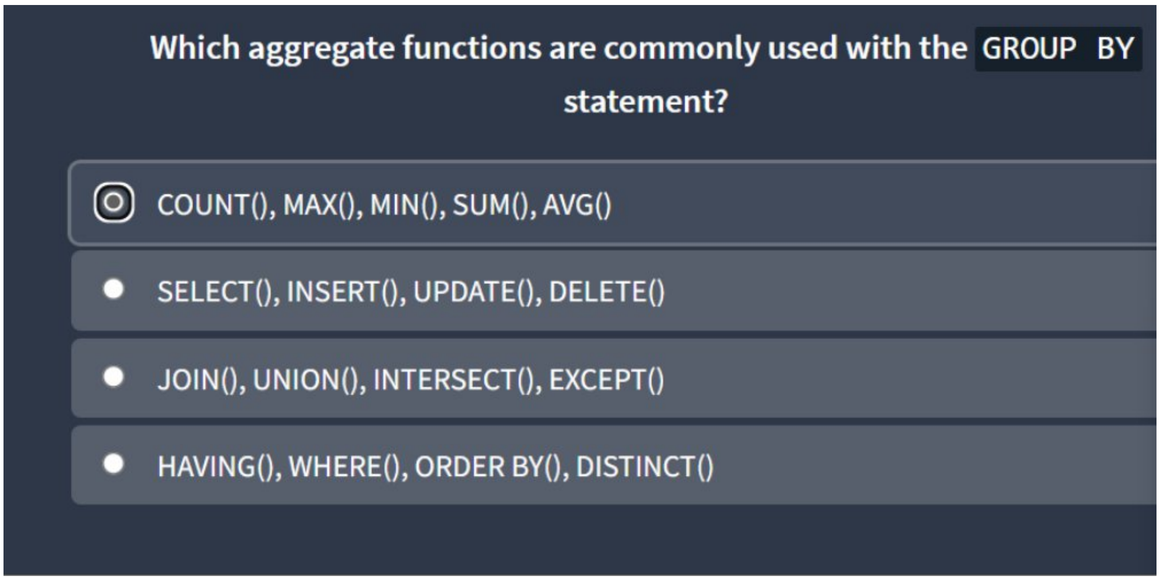


List the number of customers in each country, ordered by the country with the most customers first.

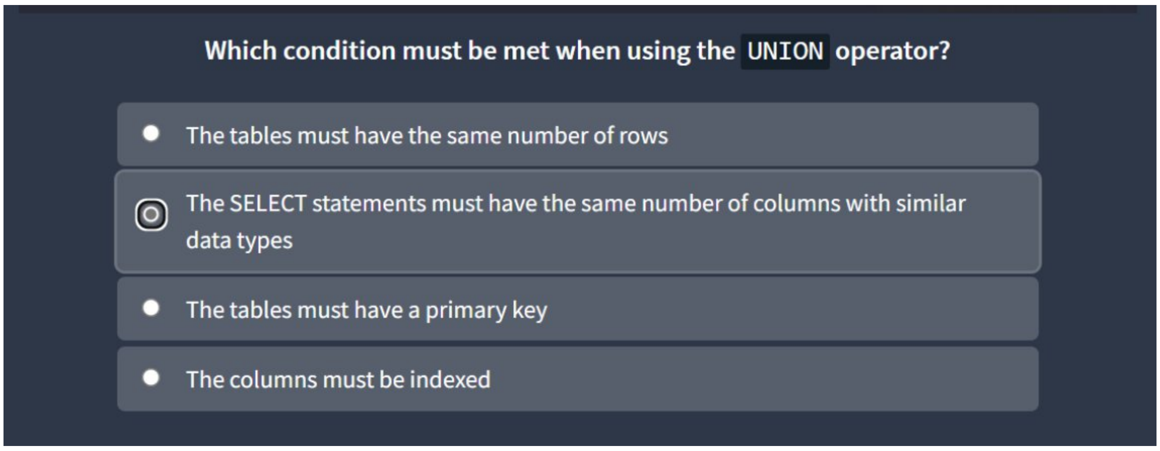
```
SELECT count(CustomerID),
Country
FROM Customers
group by country
ORDER BY count(CustomerID) desc;
```


List the number of customers in each country.

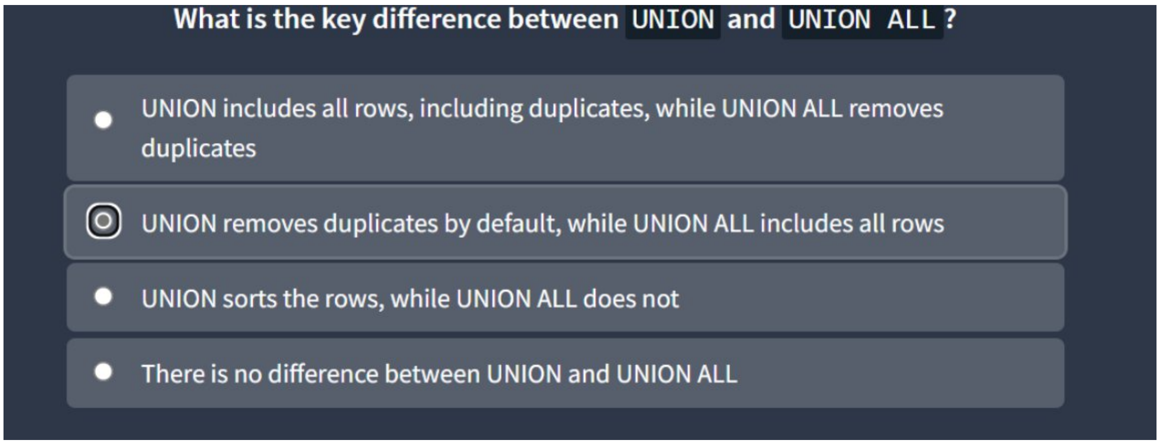
```
SELECT count(CustomerID),
Country
FROM Customers
group by country;
```


Which aggregate functions are commonly used with the `GROUP BY` statement?

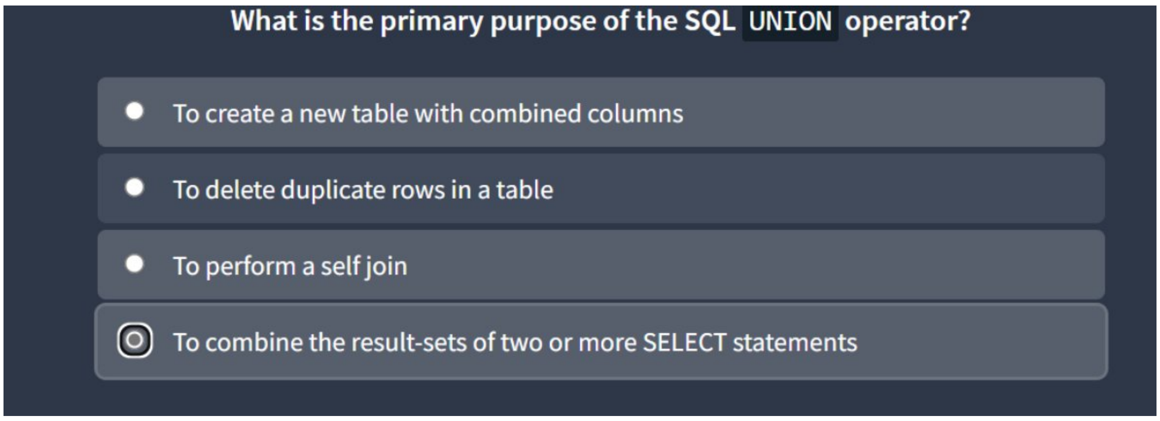
- [✓ COUNT(), MAX(), MIN(), SUM(), AVG()]
- SELECT(), INSERT(), UPDATE(), DELETE()
- JOIN(), UNION(), INTERSECT(), EXCEPT()
- HAVING(), WHERE(), ORDER BY(), DISTINCT()


Which condition must be met when using the `UNION` operator?

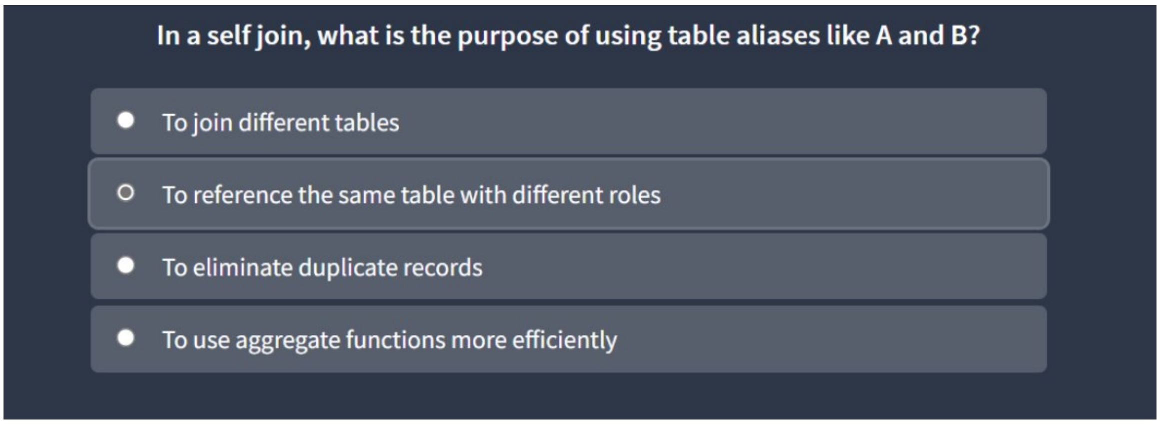
- The tables must have the same number of rows
- [✓ The SELECT statements must have the same number of columns with similar data types]
- The tables must have a primary key
- The columns must be indexed


What is the key difference between `UNION` and `UNION ALL`?

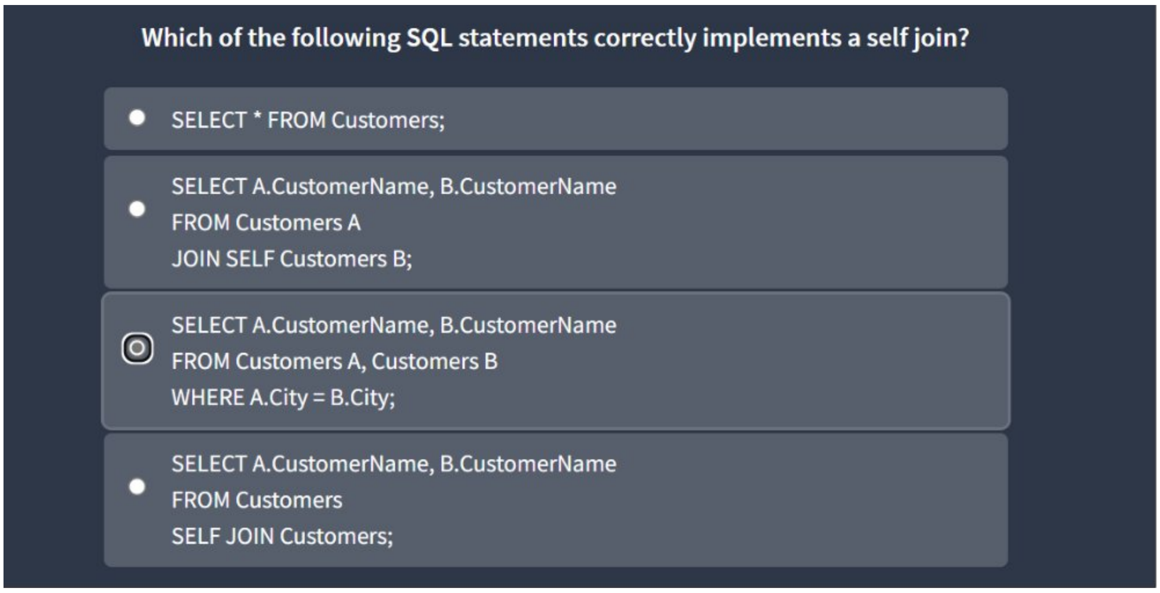
- UNION includes all rows, including duplicates, while UNION ALL removes duplicates
- [✓ UNION removes duplicates by default, while UNION ALL includes all rows]
- UNION sorts the rows, while UNION ALL does not
- There is no difference between UNION and UNION ALL


What is the primary purpose of the SQL `UNION` operator?

- To create a new table with combined columns
- To delete duplicate rows in a table
- To perform a self join
- [✓ To combine the result-sets of two or more SELECT statements]


In a self join, what is the purpose of using table aliases like A and B?

- To join different tables
- [✓ To reference the same table with different roles]
- To eliminate duplicate records
- To use aggregate functions more efficiently

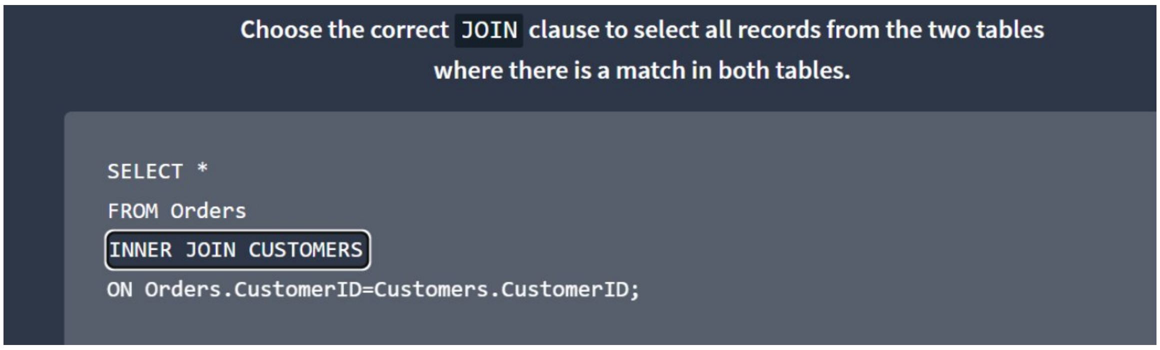


Which of the following SQL statements correctly implements a self join?

-
  ```
  SELECT * FROM Customers;
  ```

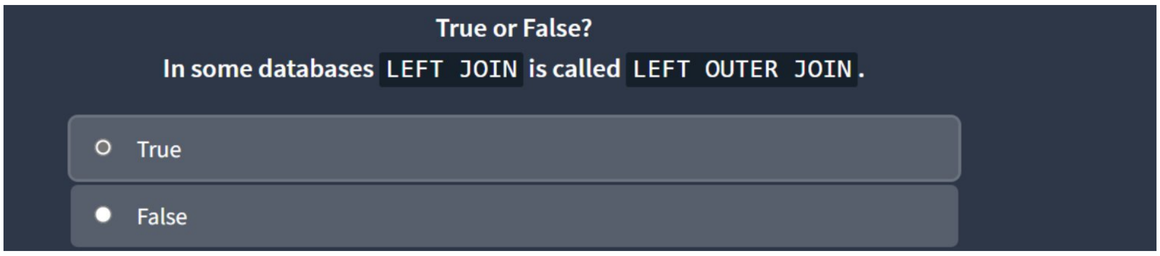
-
  ```
  SELECT A.CustomerName, B.CustomerName
  FROM Customers A
  JOIN SELF Customers B;
  ```
- [✓
  ```
  SELECT A.CustomerName, B.CustomerName
  FROM Customers A, Customers B
  WHERE A.City = B.City;
  ```
  ]

-
  ```
  SELECT A.CustomerName, B.CustomerName
  FROM Customers
  SELF JOIN Customers;
  ```



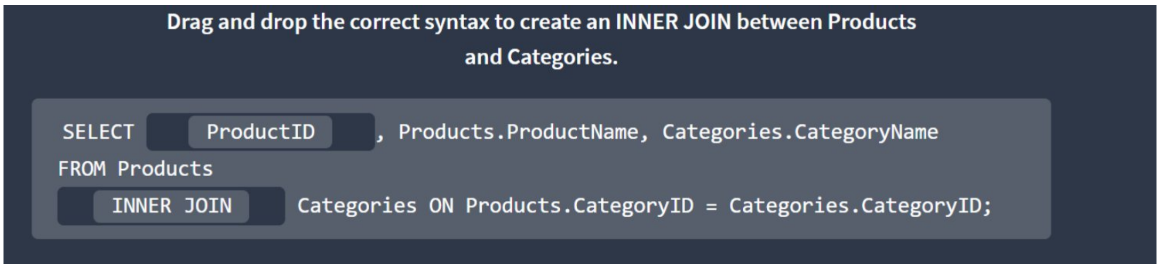
Choose the correct `JOIN` clause to select all records from the two tables where there is a match in both tables.

```
SELECT *
FROM Orders
INNER JOIN CUSTOMERS
ON Orders.CustomerID=Customers.CustomerID;
```



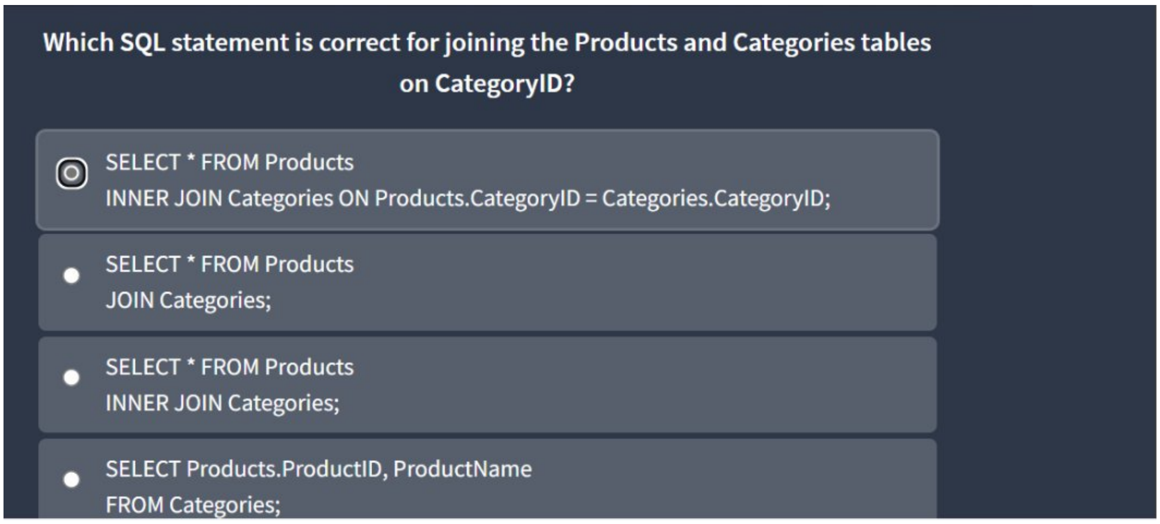
True or False?
In some databases `LEFT JOIN` is called `LEFT OUTER JOIN`.

- [✓ True]
- False



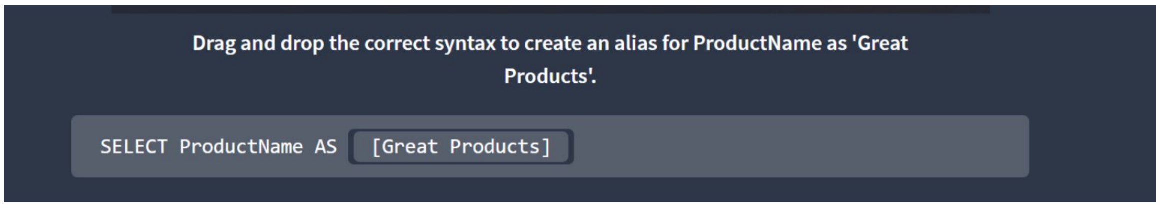
Drag and drop the correct syntax to create an INNER JOIN between Products and Categories.

```
SELECT  ProductID , Products.ProductName, Categories.CategoryName
FROM Products
 INNER JOIN  Categories ON Products.CategoryID = Categories.CategoryID;
```



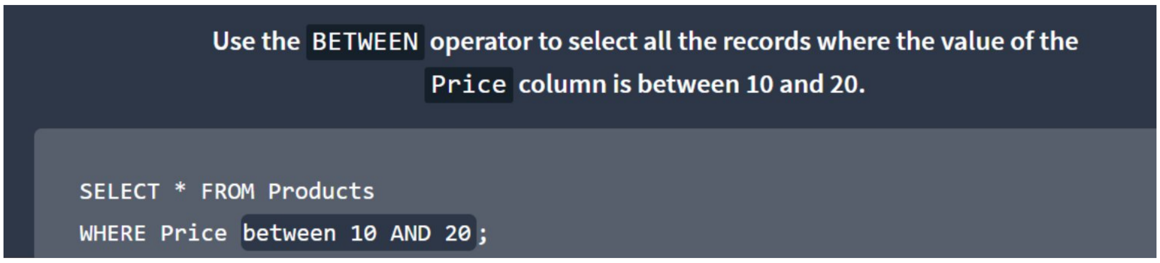
Which SQL statement is correct for joining the Products and Categories tables on CategoryID?

- [✓ SELECT * FROM Products
  INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID;]
- SELECT * FROM Products
  JOIN Categories;
- SELECT * FROM Products
  INNER JOIN Categories;
- SELECT Products.ProductID, ProductName
  FROM Categories;



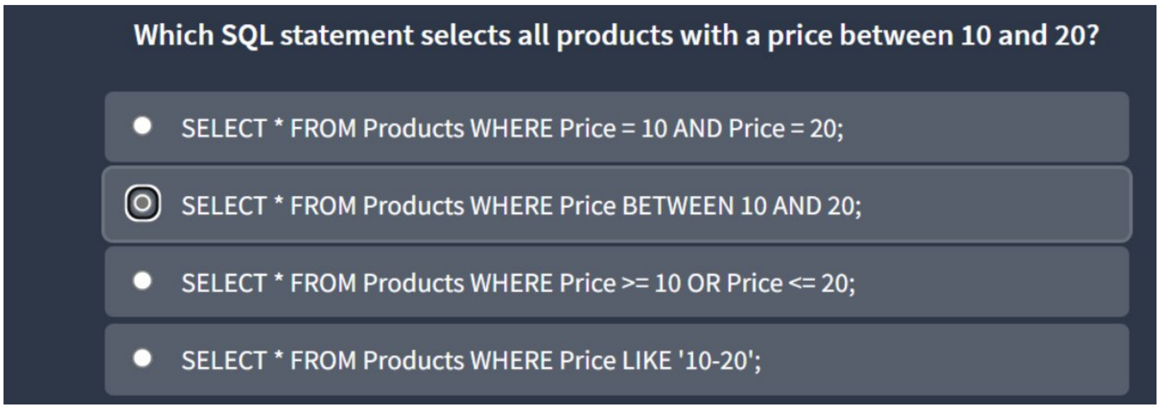
Drag and drop the correct syntax to create an alias for ProductName as 'Great Products'.

```
SELECT ProductName AS [Great Products]
```



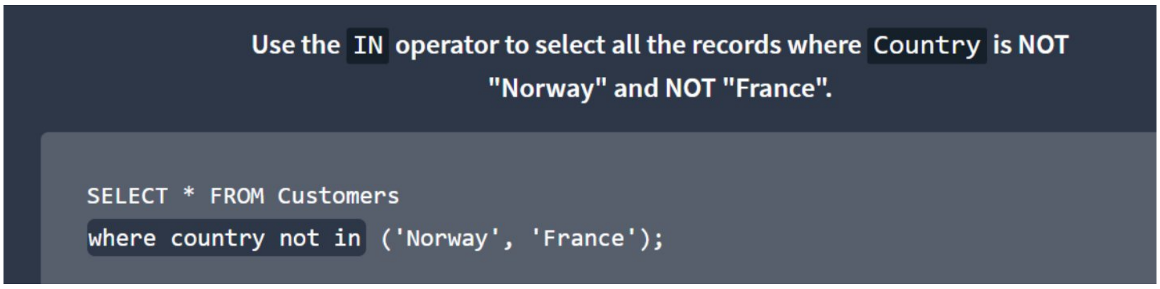
Use the `BETWEEN` operator to select all the records where the value of the `Price` column is between 10 and 20.

```
SELECT * FROM Products
WHERE Price between 10 AND 20;
```

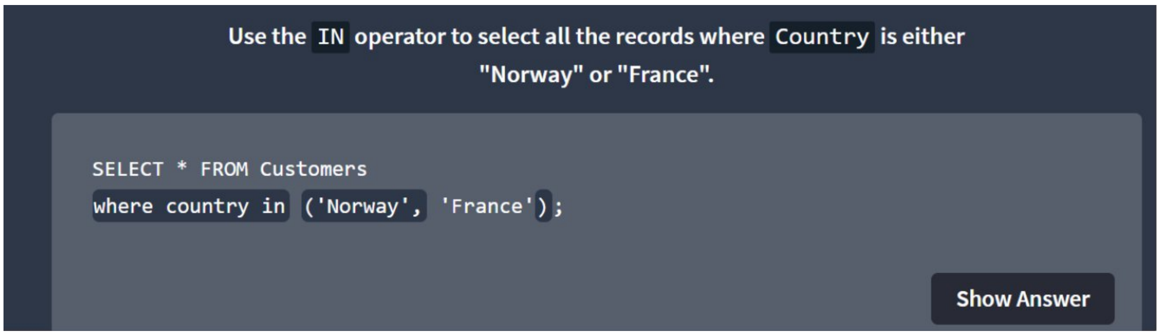


Which SQL statement selects all products with a price between 10 and 20?

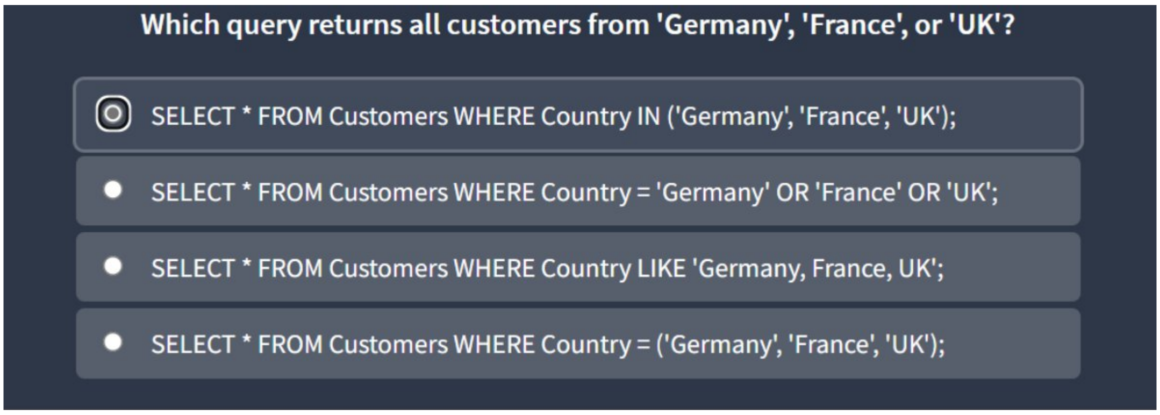
- SELECT * FROM Products WHERE Price = 10 AND Price = 20;
- [✓ SELECT * FROM Products WHERE Price BETWEEN 10 AND 20;]
- SELECT * FROM Products WHERE Price >= 10 OR Price <= 20;
- SELECT * FROM Products WHERE Price LIKE '10-20';


Use the `IN` operator to select all the records where `Country` is NOT "Norway" and NOT "France".

```
SELECT * FROM Customers
where country not in ('Norway', 'France');
```


Use the `IN` operator to select all the records where `Country` is either "Norway" or "France".

```
SELECT * FROM Customers
where country in ('Norway', 'France');
```

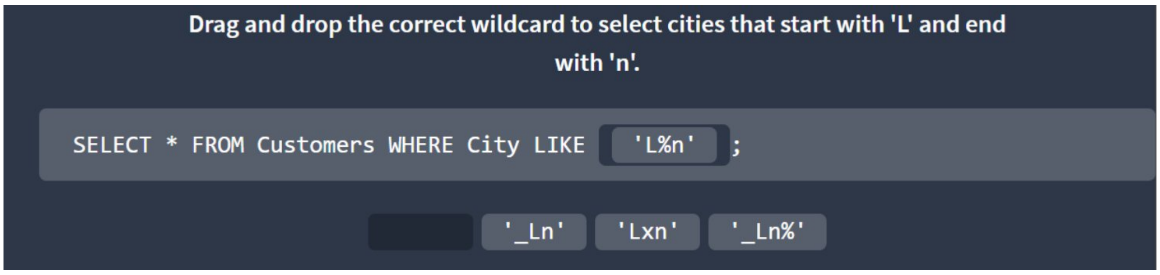

Which query returns all customers from 'Germany', 'France', or 'UK'?

[✓ SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'UK');]

SELECT * FROM Customers WHERE Country = 'Germany' OR 'France' OR 'UK';

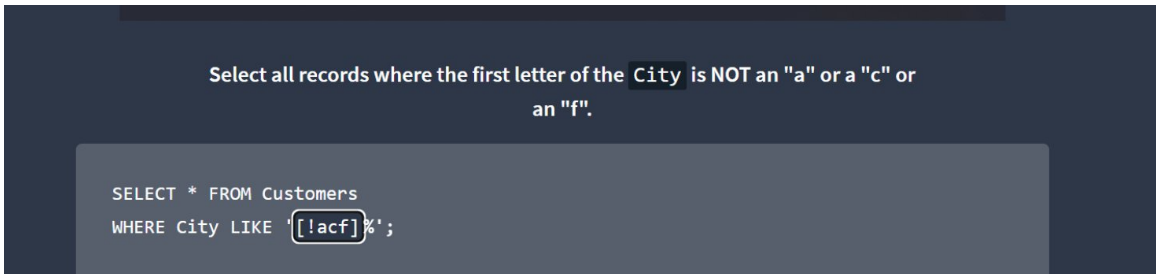
SELECT * FROM Customers WHERE Country LIKE 'Germany, France, UK';

SELECT * FROM Customers WHERE Country = ('Germany', 'France', 'UK');

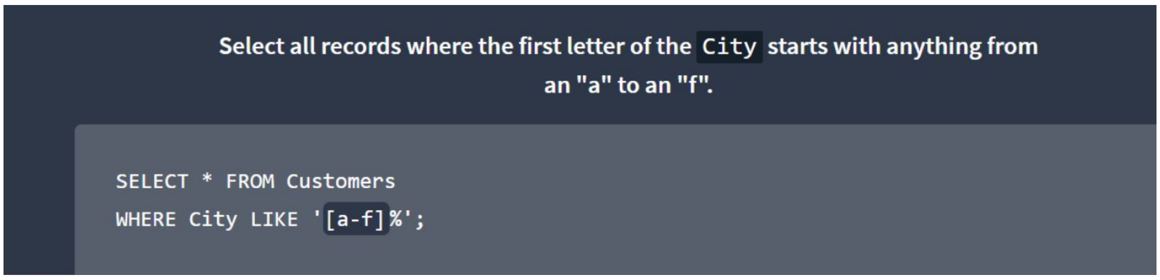

Drag and drop the correct wildcard to select cities that start with 'L' and end with 'n'.

```
SELECT * FROM Customers WHERE City LIKE 'L%n' ;
```

`'_Ln'` `'Lxn'` `'_Ln%'`

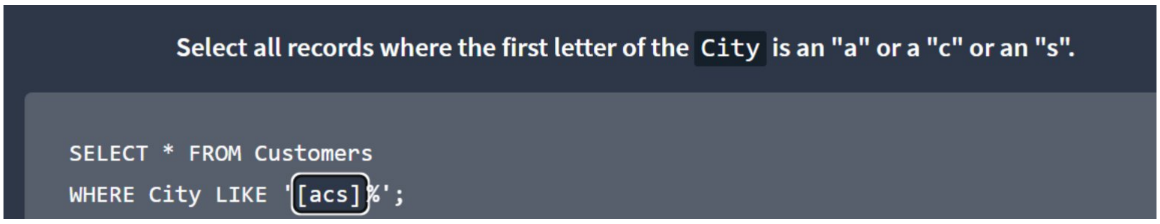

Select all records where the first letter of the `City` is NOT an "a" or a "c" or an "f".

```
SELECT * FROM Customers
WHERE City LIKE '[!acf]%';
```



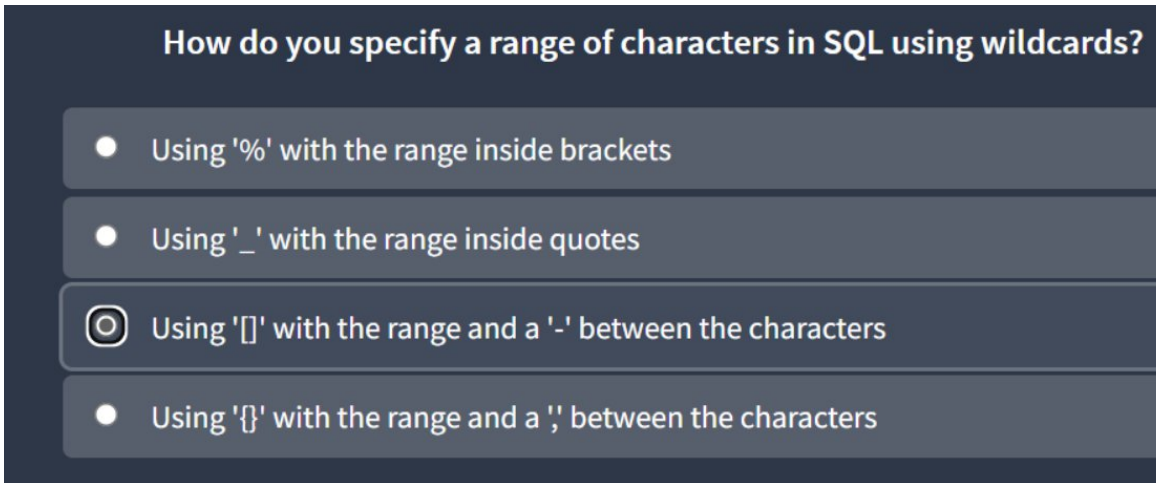
Select all records where the first letter of the `City` starts with anything from an "a" to an "f".

```
SELECT * FROM Customers
WHERE City LIKE '[a-f]%';
```



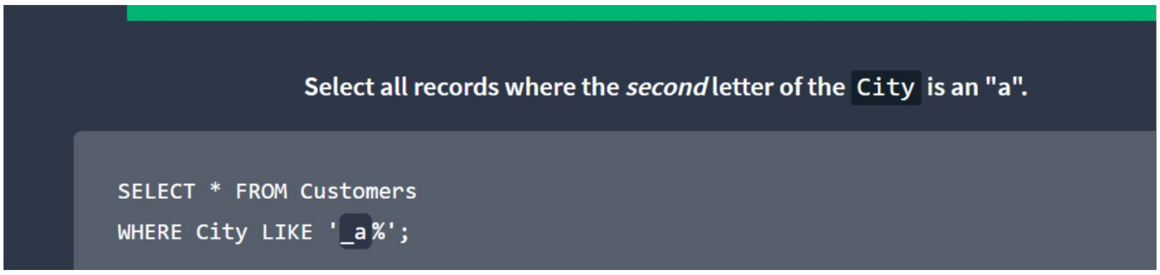
Select all records where the first letter of the `City` is an "a" or a "c" or an "s".

```
SELECT * FROM Customers
WHERE City LIKE '[acs]%';
```



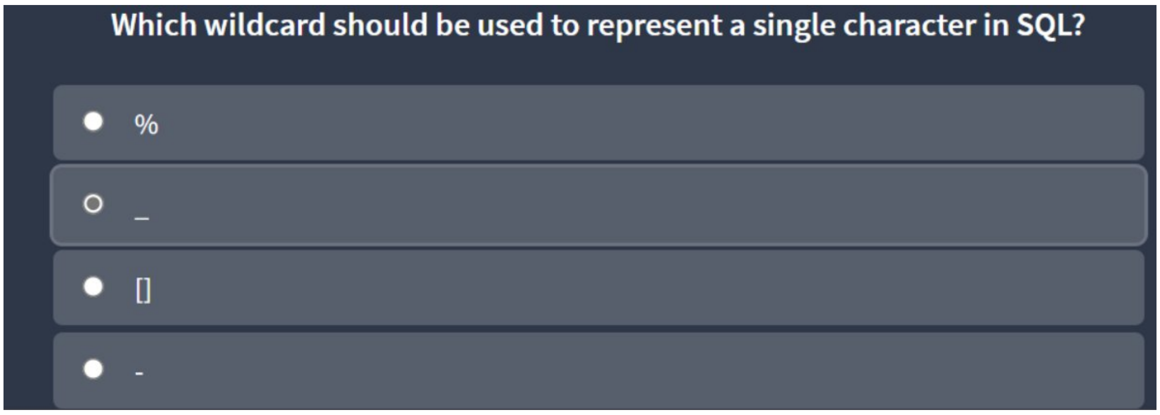
How do you specify a range of characters in SQL using wildcards?

- ● Using '%' with the range inside brackets
- ● Using '_' with the range inside quotes
- [✓ Using '[]' with the range and a '-' between the characters]
- ● Using '{}' with the range and a ',' between the characters

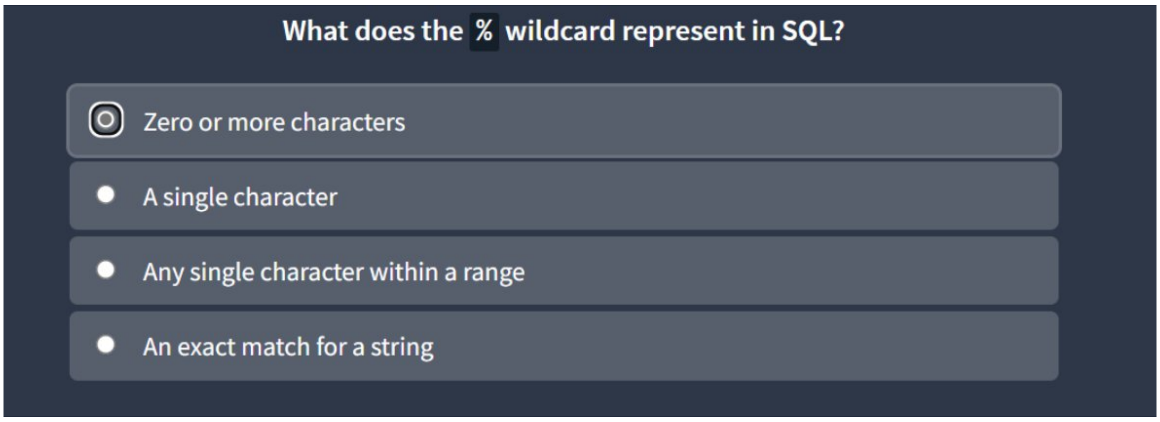


Select all records where the *second* letter of the `City` is an "a".

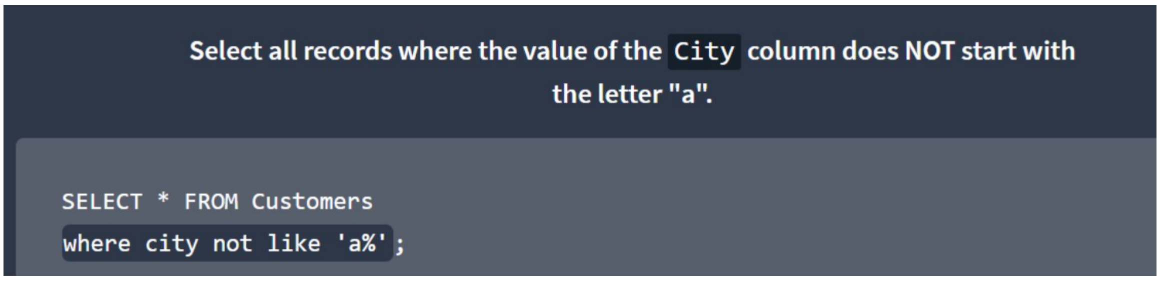
```
SELECT * FROM Customers
WHERE City LIKE '_a%';
```


Which wildcard should be used to represent a single character in SQL?

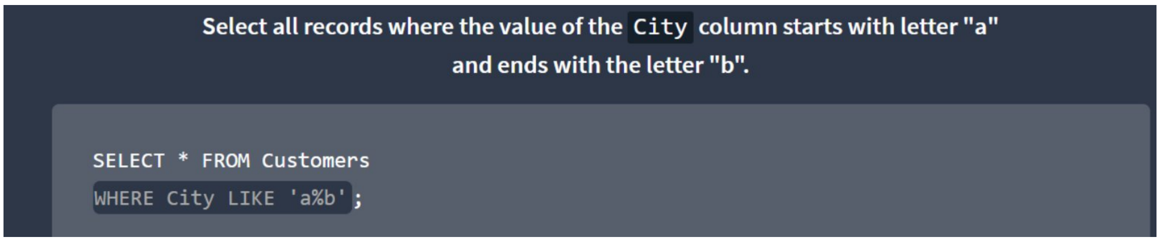
- %
- [✓ _]
- []
- -


What does the `%` wildcard represent in SQL?

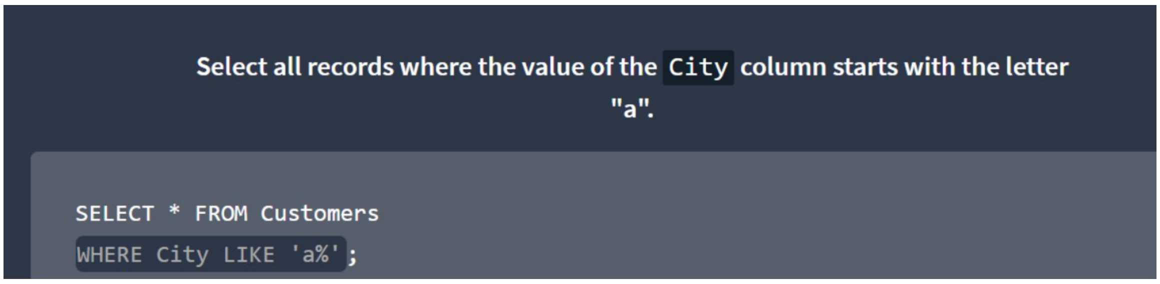
- [✓ Zero or more characters]
- A single character
- Any single character within a range
- An exact match for a string


Select all records where the value of the `City` column does NOT start with the letter "a".

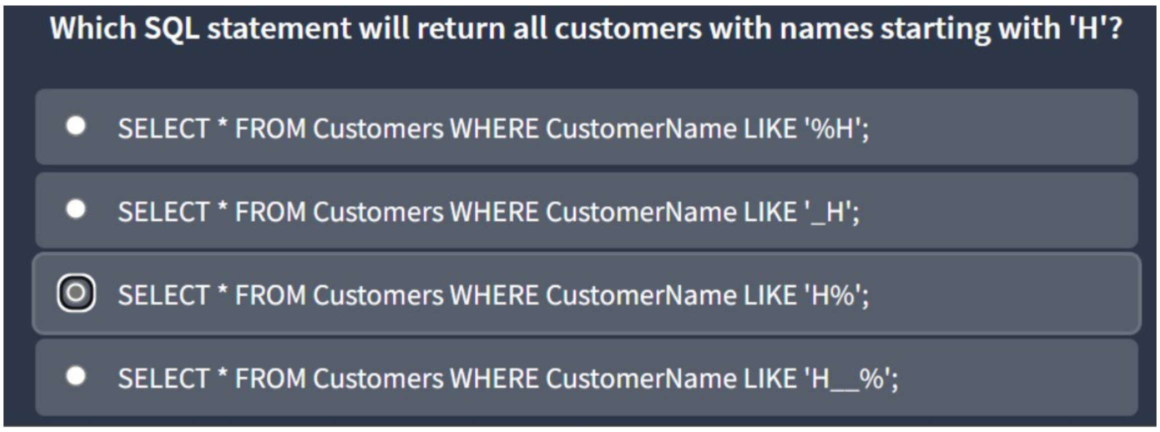
```
SELECT * FROM Customers
where city not like 'a%';
```


Select all records where the value of the `City` column starts with letter "a" and ends with the letter "b".

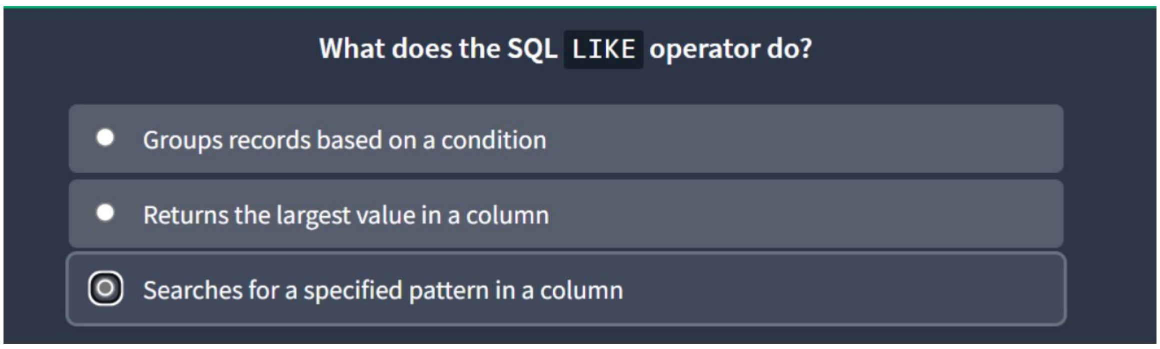
```
SELECT * FROM Customers
WHERE City LIKE 'a%b';
```


Select all records where the value of the `City` column starts with the letter "a".

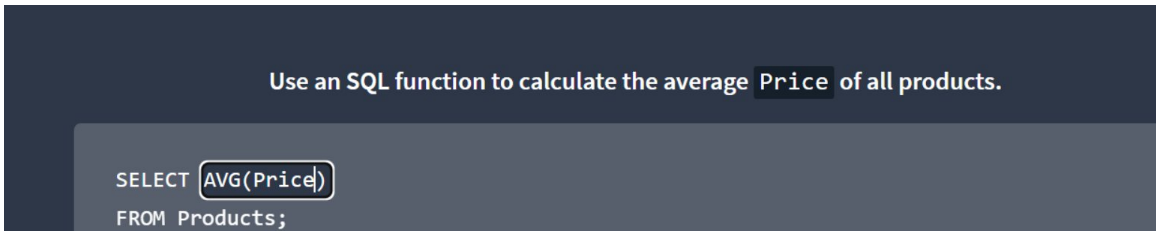
```
SELECT * FROM Customers
WHERE City LIKE 'a%';
```


Which SQL statement will return all customers with names starting with 'H'?

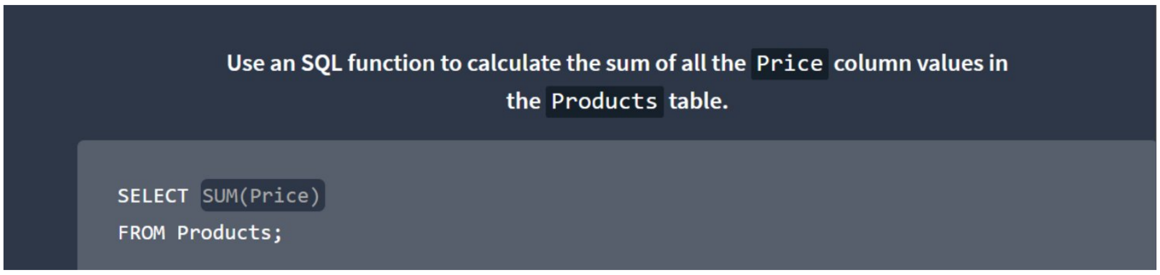
- SELECT * FROM Customers WHERE CustomerName LIKE '%H';
- SELECT * FROM Customers WHERE CustomerName LIKE '_H';
- [✓ SELECT * FROM Customers WHERE CustomerName LIKE 'H%';]
- SELECT * FROM Customers WHERE CustomerName LIKE 'H__%';


What does the SQL `LIKE` operator do?

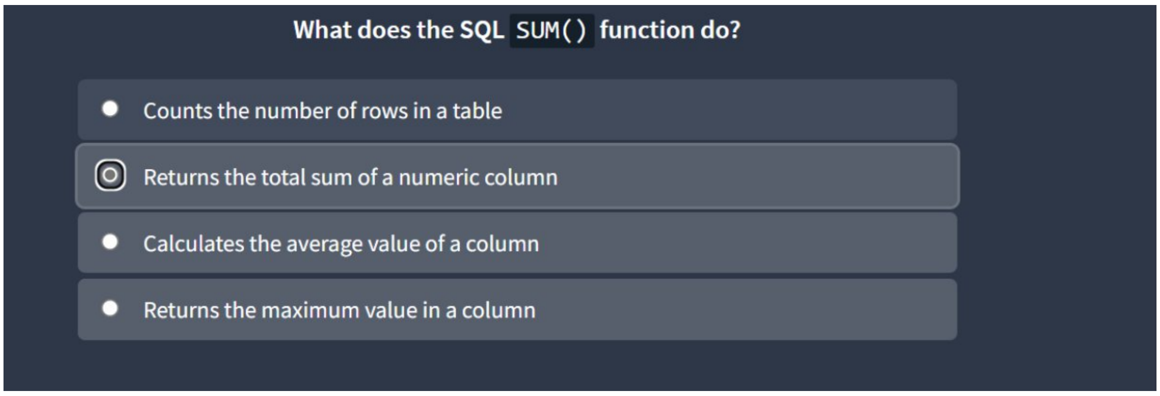
- Groups records based on a condition
- Returns the largest value in a column
- [✓ Searches for a specified pattern in a column]


Use an SQL function to calculate the average `Price` of all products.

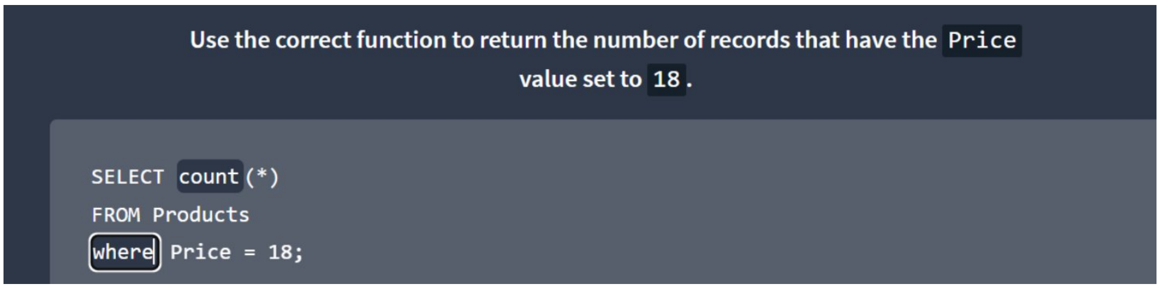
```
SELECT AVG(Price)
FROM Products;
```


Use an SQL function to calculate the sum of all the `Price` column values in the `Products` table.

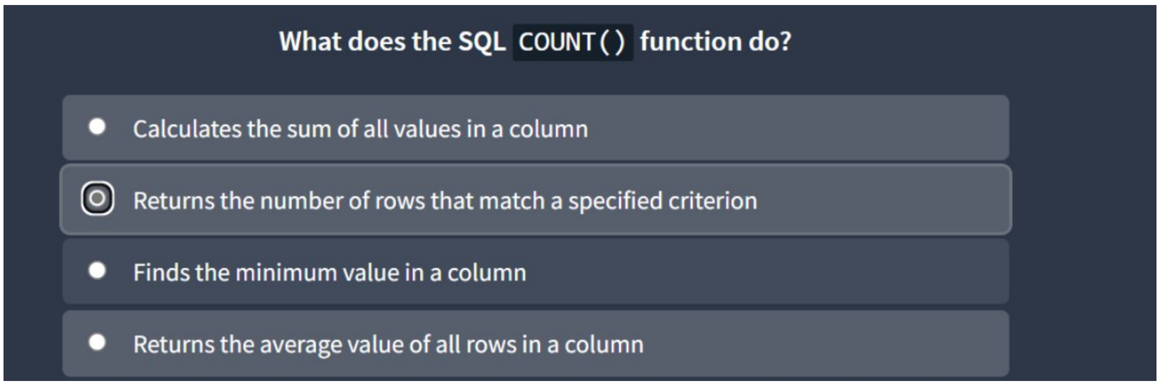
```
SELECT SUM(Price)
FROM Products;
```


What does the SQL `SUM()` function do?

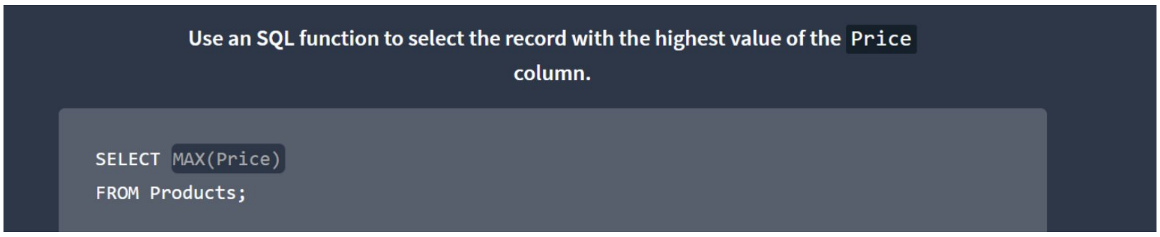
- Counts the number of rows in a table
- [✓ Returns the total sum of a numeric column]
- Calculates the average value of a column
- Returns the maximum value in a column


Use the correct function to return the number of records that have the `Price` value set to `18`.

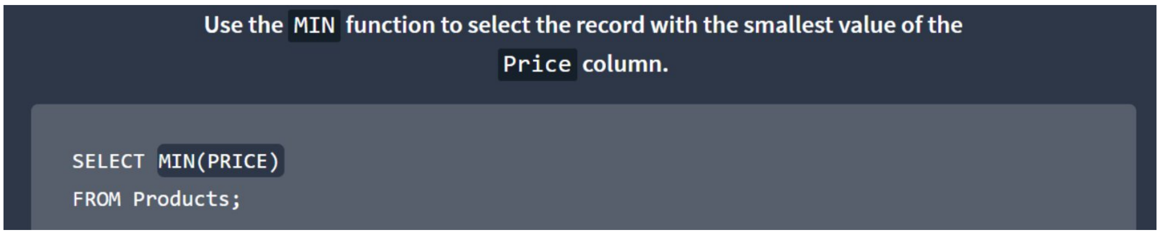
```
SELECT count(*)
FROM Products
where Price = 18;
```


What does the SQL `COUNT()` function do?

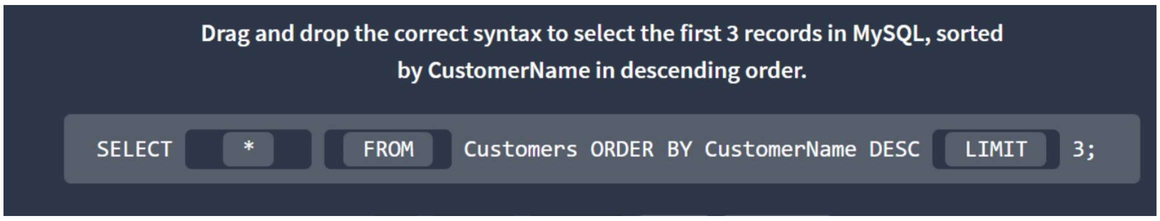
- Calculates the sum of all values in a column
- [✓ Returns the number of rows that match a specified criterion]
- Finds the minimum value in a column
- Returns the average value of all rows in a column


Use an SQL function to select the record with the highest value of the `Price` column.

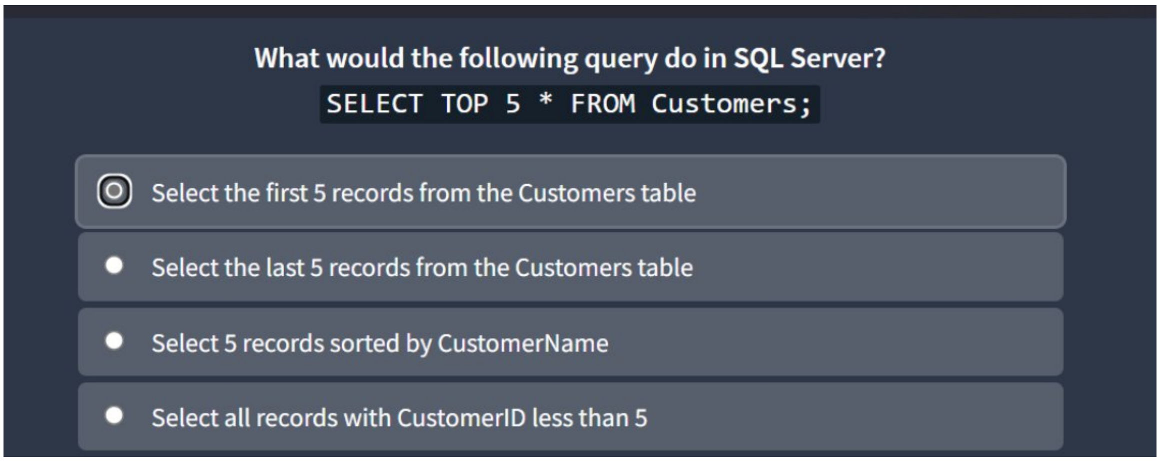
```
SELECT MAX(Price)
FROM Products;
```


Use the `MIN` function to select the record with the smallest value of the `Price` column.

```
SELECT MIN(PRICE)
FROM Products;
```

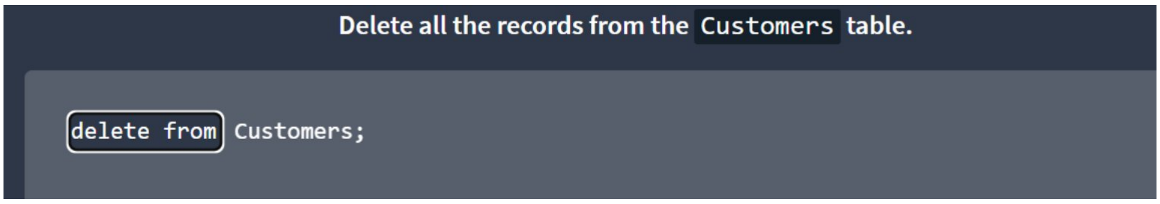

Drag and drop the correct syntax to select the first 3 records in MySQL, sorted by CustomerName in descending order.

```
SELECT * FROM Customers ORDER BY CustomerName DESC LIMIT 3;
```

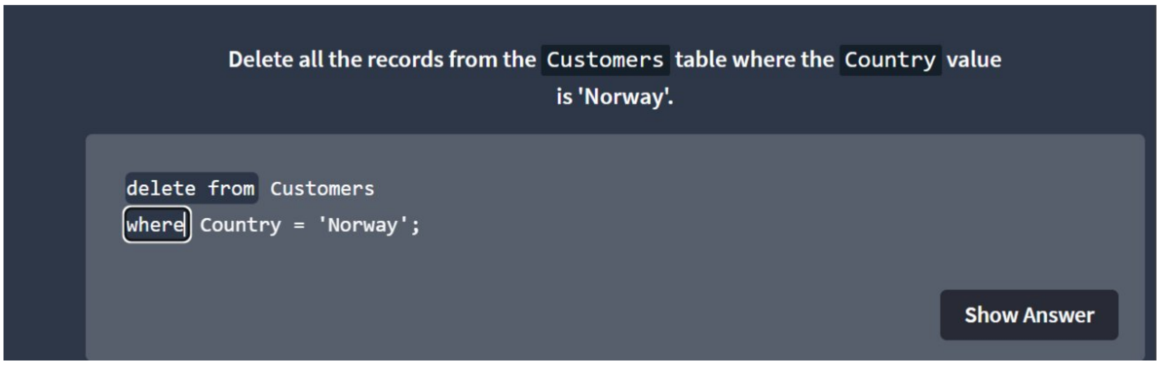

What would the following query do in SQL Server?

```
SELECT TOP 5 * FROM Customers;
```

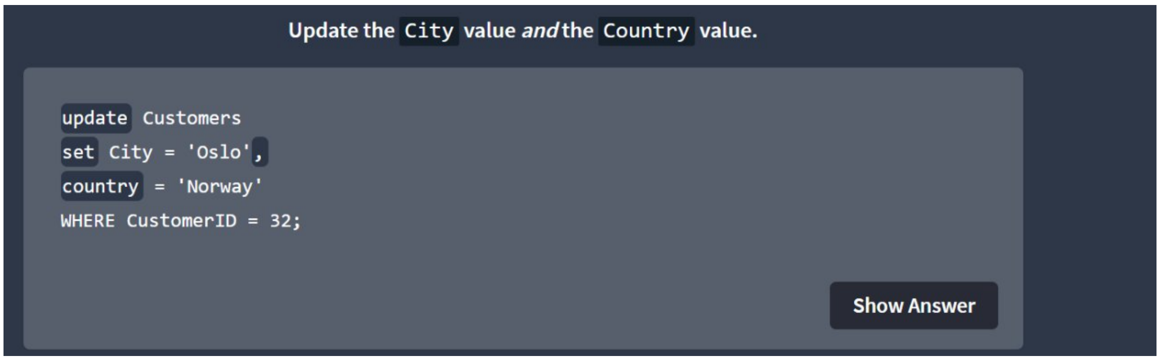
[✓ Select the first 5 records from the Customers table]
● Select the last 5 records from the Customers table
● Select 5 records sorted by CustomerName
● Select all records with CustomerID less than 5


Delete all the records from the `Customers` table.

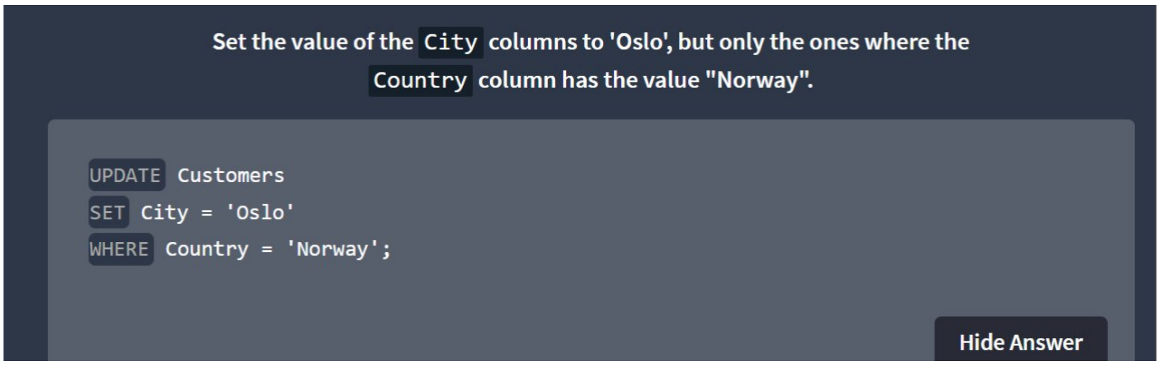
```
delete from Customers;
```


Delete all the records from the `Customers` table where the `Country` value is 'Norway'.

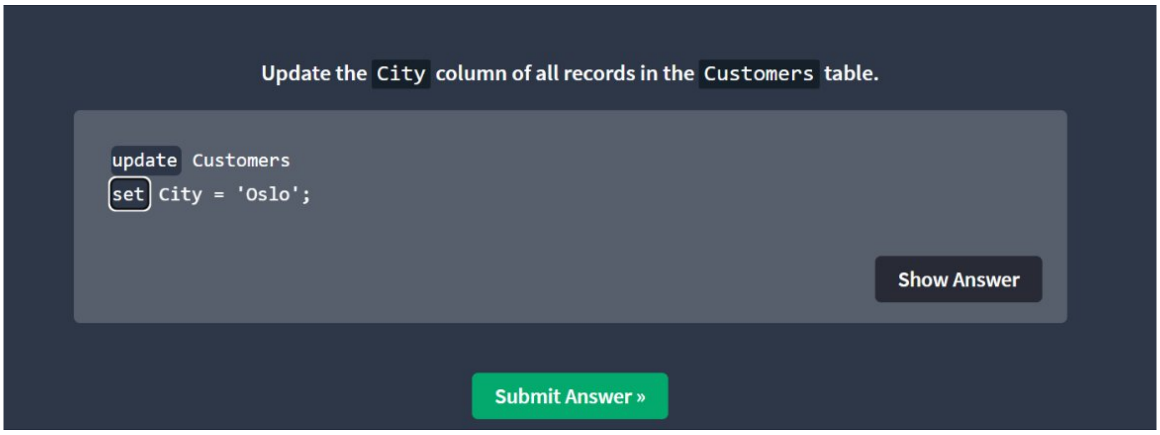
```
delete from Customers
where Country = 'Norway';
```


Update the `City` value *and* the `Country` value.

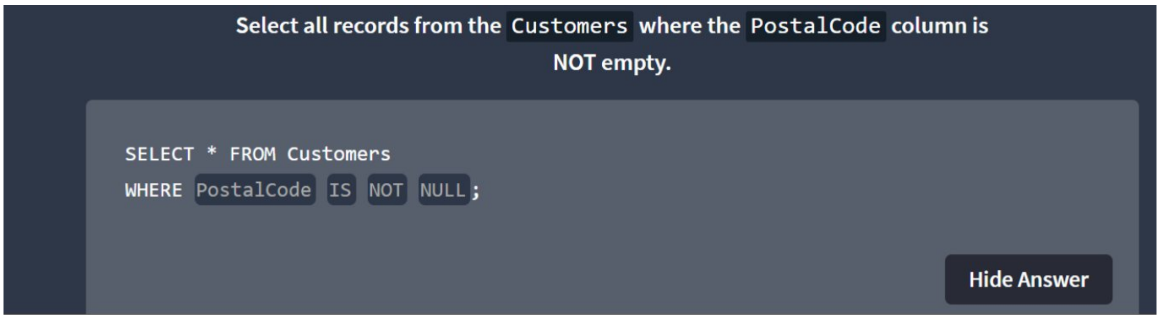
```
update Customers
set City = 'Oslo',
country = 'Norway'
WHERE CustomerID = 32;
```


Set the value of the `City` columns to 'Oslo', but only the ones where the `Country` column has the value "Norway".

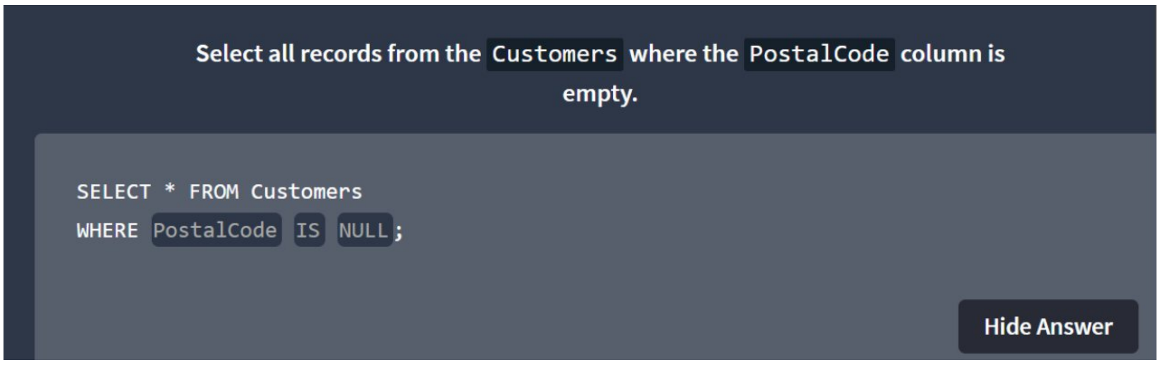
```
UPDATE Customers
SET City = 'Oslo'
WHERE Country = 'Norway';
```


Update the `City` column of all records in the `Customers` table.

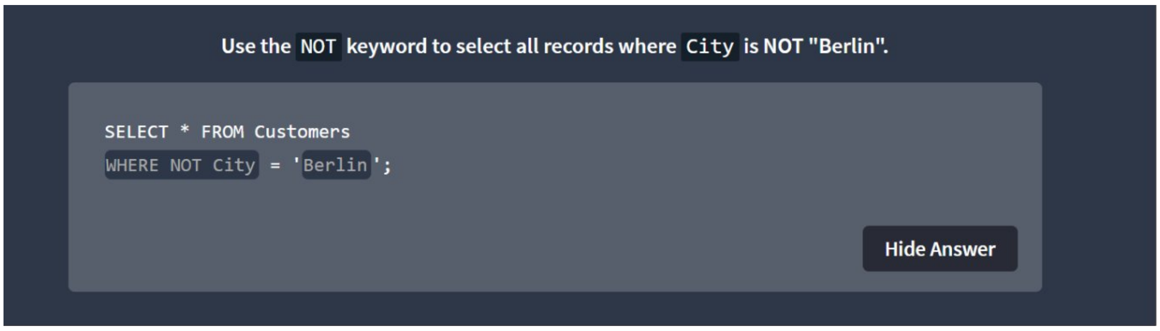
```
update Customers
set City = 'Oslo';
```


Select all records from the `Customers` where the `PostalCode` column is NOT empty.

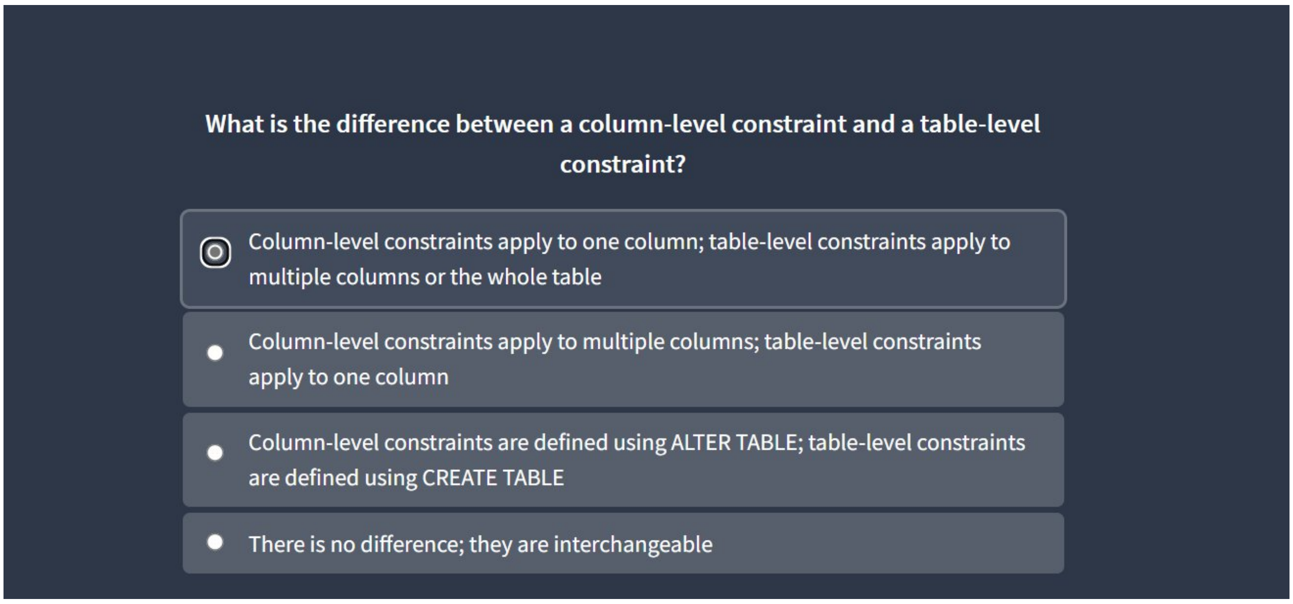
```
SELECT * FROM Customers
WHERE PostalCode IS NOT NULL;
```


Select all records from the `Customers` where the `PostalCode` column is empty.

```
SELECT * FROM Customers
WHERE PostalCode IS NULL;
```


Use the `NOT` keyword to select all records where `City` is NOT "Berlin".

```
SELECT * FROM Customers
WHERE NOT City = 'Berlin';
```


What is the difference between a column-level constraint and a table-level constraint?

- [✓ Column-level constraints apply to one column; table-level constraints apply to multiple columns or the whole table]
- Column-level constraints apply to multiple columns; table-level constraints apply to one column
- Column-level constraints are defined using ALTER TABLE; table-level constraints are defined using CREATE TABLE
- There is no difference; they are interchangeable

```
GENDER   TOTAL_COUNT

-------  ------------

Male         25

Female        30
```

1. Overview

Feature	DELETE	TRUNCATE
Purpose	Removes rows based on a condition (can delete some or all rows)	Removes all rows from a table
WHERE Clause	✅ Yes, can use WHERE	❌ No, deletes all rows
Transaction Control	✅ Can be rolled back (DML)	❌ Cannot be rolled back (DDL)
Speed	Slower (row-by-row)	Faster (deallocates pages)
Triggers	✅ Fires DELETE triggers	❌ Does not fire triggers
Auto Increment / Identity Reset	MySQL: No reset by default	MySQL: Resets to initial value
Locks	Locks rows (in some engines)	Locks the whole table
Type of Command	DML (Data Manipulation Language)	DDL (Data Definition Language)

---

2. DELETE Command

➤ Definition:

Used to remove specific rows from a table using a condition.

➤ Syntax:

```
DELETE FROM table_name
WHERE condition;
```

If you omit the WHERE clause, all rows will be deleted.

➤ Example (MySQL or Oracle):

Suppose you have a table:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(50)
);
```

Insert data:

```
INSERT INTO employees VALUES (1, 'Amit', 'HR');
INSERT INTO employees VALUES (2, 'Priya', 'IT');
INSERT INTO employees VALUES (3, 'Raj', 'Finance');
```

Delete only one record:

```
DELETE FROM employees WHERE department = 'HR';
```

✅ This removes only rows where department = 'HR'.

To delete all rows:

```
DELETE FROM employees;
```

You can ROLLBACK this if inside a transaction:

```
START TRANSACTION;   -- MySQL
DELETE FROM employees;
ROLLBACK;  -- Undo delete
```

In Oracle, you'd use:

```
BEGIN;
DELETE FROM employees;
ROLLBACK;
```

---

3. TRUNCATE Command

➤ Definition:

Used to remove all rows from a table instantly, freeing up the storage space.

➤ Syntax:

```
TRUNCATE TABLE table_name;
```

➤ Example:

```
TRUNCATE TABLE employees;
```

✅ This removes all records from the table quickly.

- No WHERE clause allowed.
- Auto increment (in MySQL) resets:

```
-- Before truncate
INSERT INTO employees (name, department) VALUES ('Meena', 'IT');
-- id = 4 (next)
TRUNCATE TABLE employees;
INSERT INTO employees (name, department) VALUES ('Ravi', 'IT');
-- id = 1 (reset)
```

- In Oracle, space is deallocated, and rollback is not possible once executed.

---

4. Practical Difference

Action	DELETE	TRUNCATE
Deletes specific rows?	✅ Yes	❌ No
Fires triggers?	✅ Yes	❌ No
Can be rolled back?	✅ Yes	❌ No (unless in special Oracle cases)
Resets AUTO_INCREMENT?	❌ No	✅ Yes (MySQL)
Speed	Slower	Much faster
Removes structure?	❌ No	❌ No (keeps table structure)
Space released	Partially	Fully

---

5. Example Comparison

```
-- Before cleanup
SELECT COUNT(*) FROM employees;  -- 3 rows

-- Using DELETE
DELETE FROM employees WHERE department = 'IT';
SELECT COUNT(*) FROM employees;  -- 2 rows remain

-- Using TRUNCATE
TRUNCATE TABLE employees;
SELECT COUNT(*) FROM employees;  -- 0 rows
```

---

6. In Summary

Aspect	DELETE	TRUNCATE
Clause support	WHERE supported	WHERE not allowed
Rollback possible	✅ Yes	❌ No
Trigger execution	✅ Yes	❌ No
Performance	Slower	Faster
Transaction log usage	Logs each row	Logs table deallocation
Type	DML	DDL

✅ When to Use

- Use DELETE → when you need to remove specific rows or when rollback is needed.
- Use TRUNCATE → when you want to empty the entire table quickly and don't need rollback.

Q. Find the 3rd highest salary from the employer table .
SELECT DISTINCT salary
FROM employer
ORDER BY salary DESC
LIMIT 1 OFFSET 2;

1. executeQuery()

- Used for SELECT statements (queries that return data).
- Returns a ResultSet object that contains the data produced by the query.

✅ Example:

```
String sql = "SELECT * FROM employees WHERE department = 'HR'";
Statement stmt = connection.createStatement();

// Execute SELECT query
ResultSet rs = stmt.executeQuery(sql);

// Iterate through result
while (rs.next()) {
    System.out.println("Employee ID: " + rs.getInt("id"));
    System.out.println("Employee Name: " + rs.getString("name"));
}
```

➡ Key Point:
executeQuery() → only for queries that return data (like SELECT).

---

2. executeUpdate()

- Used for INSERT, UPDATE, DELETE, or DDL (CREATE, DROP, ALTER) statements.
- Returns an integer — the number of rows affected by the query.

✅ **Example:**

```
String sql = "UPDATE employees SET salary = 60000 WHERE department = 'HR'";
Statement stmt = connection.createStatement();

// Execute UPDATE query
int rowsAffected = stmt.executeUpdate(sql);
System.out.println(rowsAffected + " rows updated.");
```

➡ Key Point:
executeUpdate() → returns count of affected rows, used for write operations.

---

3. execute()

- Used when you don't know whether the SQL will return a result set or an update count.
- Returns a boolean:
    - `true → if the result is a ResultSet (like SELECT)`
    - `false → if the result is an update count (like UPDATE, INSERT, etc.)`
- `You can then get the result using:`
    - `getResultSet() or`
    - `getUpdateCount()`

✅ **Example:**

```
String sql = "DELETE FROM employees WHERE department = 'HR'";
Statement stmt = connection.createStatement();

// Execute any SQL statement
boolean isResultSet = stmt.execute(sql);

if (isResultSet) {
    ResultSet rs = stmt.getResultSet();
    while (rs.next()) {
        System.out.println(rs.getString("name"));
    }
} else {
    int count = stmt.getUpdateCount();
    System.out.println("Rows affected: " + count);
}
```

➡ Key Point:
execute() → flexible method used when you don’t know the SQL type.

易 Summary Table

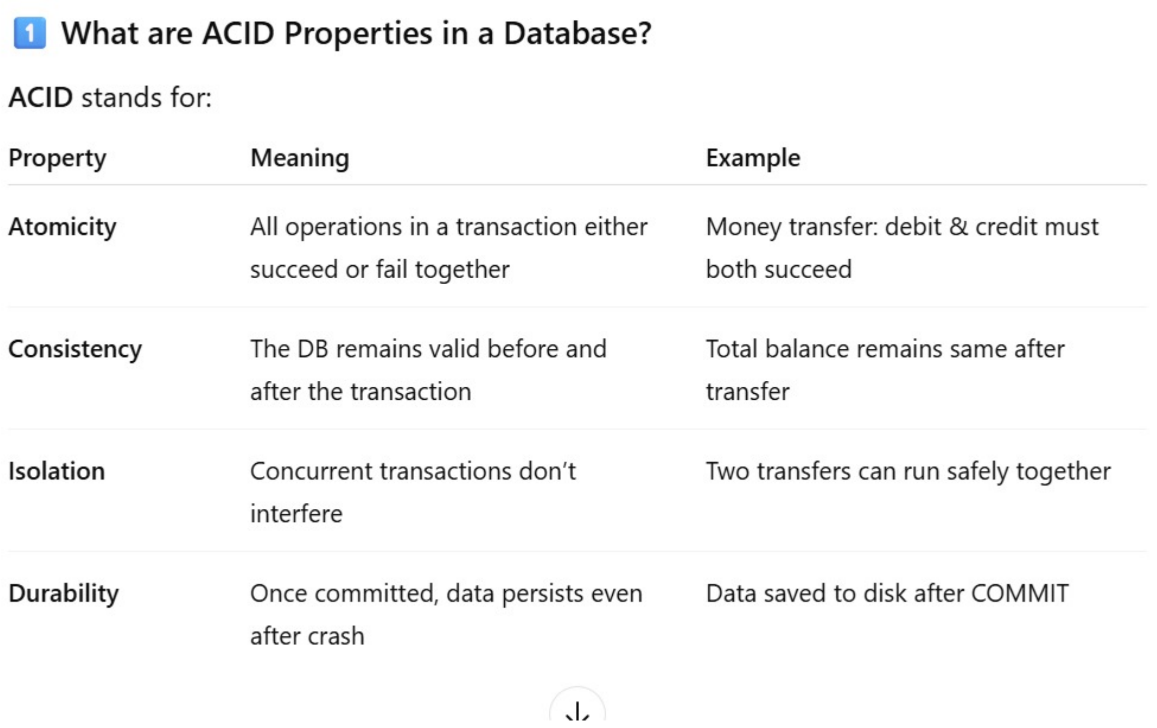
Method	Use For	Return Type	Example Query	Notes
executeQuery()	SELECT	ResultSet	SELECT * FROM employees	For reading data
executeUpdate()	INSERT / UPDATE / DELETE / DDL	int (rows affected)	UPDATE employees SET salary=50000	For modifying data
execute()	Any (SELECT / DML / DDL)	boolean (true → ResultSet, false → update count)	CREATE TABLE, SELECT, DELETE	Flexible but less specific

SQL – employees with salary above average

```
SELECT *
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

SQL – users with more than 5 transactions per day

```
SELECT user_id, txn_date, COUNT(*) AS txn_count
FROM transactions
GROUP BY user_id, txn_date
HAVING COUNT(*) > 5;
```



1️⃣ What are ACID Properties in a Database?

ACID stands for:

Property	Meaning	Example
Atomicity	All operations in a transaction either succeed or fail together	Money transfer: debit & credit must both succeed
Consistency	The DB remains valid before and after the transaction	Total balance remains same after transfer
Isolation	Concurrent transactions don't interfere	Two transfers can run safely together
Durability	Once committed, data persists even after crash	Data saved to disk after COMMIT

lk

```
employees(
   emp_id,
   emp_name,
   department,
   salary
)
```

⭐ Q36. Return employees earning more than the average salary in their department

Logic:
Calculate average salary per department, then select employees above that average.

✅ SQL Answer (Most Common & Recommended)

sql
Copy code

```sql
SELECT e.*
FROM employees e
JOIN (
   SELECT department, AVG(salary) AS avg_salary
   FROM employees
   GROUP BY department
) d
ON e.department = d.department
WHERE e.salary > d.avg_salary;
```

頴 Interview Explanation

First calculate avg salary per department (subquery).

Join original table to filter employees above avg.

Very common SQL interview pattern.

⭐ Q37. Find 2nd highest salary (3 ways)

Method 1 — Using MAX with a subquery

sql
Copy code
```
SELECT MAX(salary) AS second_highest_salary
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```
➤ Explanation
Find the maximum salary.

Then find max salary less than that.

Guaranteed correct.

Method 2 — Using ORDER BY + LIMIT
(Works for MySQL, PostgreSQL)

sql
Copy code
```
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```
➤ Explanation
Sort by salary descending.

Skip the first row (highest salary).

Return the next one.

Method 3 — Using ROW_NUMBER() or DENSE_RANK()
Using DENSE_RANK() (Recommended in interviews)
sql
Copy code
```
SELECT salary
FROM (
   SELECT salary,
          DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
   FROM employees
) t
WHERE rnk = 2;
```
➤ Explanation
DENSE_RANK() handles duplicates properly.

Rank 1 = highest, rank 2 = second highest.

☆ Q38. Return employees who have same salary (duplicate salary groups)
This means finding employees whose salary appears more than once.

Step 1 — Find salaries occurring more than once
sql
Copy code
```
SELECT salary
FROM employees
```

GROUP BY salary
HAVING COUNT(*) > 1;
Step 2 — Get all employees with those salaries
sql
Copy code
```
SELECT *
FROM employees
WHERE salary IN (
   SELECT salary
   FROM employees
   GROUP BY salary
   HAVING COUNT(*) > 1
);
```
➤ Explanation
Identify salaries that repeat.

Return all employees with those salaries.

⭐ Q39. Get top 3 highest salaries from each department
This is a very popular SQL problem.

Use DENSE_RANK() window function.

✅ SQL Answer
sql
Copy code
```
SELECT emp_id, emp_name, department, salary
FROM (
   SELECT e.*,
        DENSE_RANK() OVER (
           PARTITION BY department
           ORDER BY salary DESC
        ) AS rnk
   FROM employees e
) t
WHERE rnk <= 3;
```
➤ Explanation
Partition salaries by department.

Rank them inside each department.

Pick first 3.

✔ Works even if duplicate salaries exist
Because DENSE_RANK() handles ties correctly.`

SQL – Employee Name with Manager Name

Table: employee

emp_id	emp_name	manager_id
1	A	null
2	B	1
3	C	1
4	D	2

✅ Self-Join Query

```
SELECT e.emp_name AS Employee,
       m.emp_name AS Manager
FROM employee e
LEFT JOIN employee m
ON e.manager_id = m.emp_id;
```

⅔ Explanation:

- Same table joined with itself
- `e → employee`
- `m → manager`
- `LEFT JOIN keeps top manager`

撤 **Interview line:**

> *This is a classic self-join scenario.*