SQL Important Questions with Queries

Assumed practice tables include Employee, Department, Student, Exam, Subject, Customer, Orders, Product, OrderItem, Account, and related sample tables. ANSI-style SQL is used where possible; a few procedure/trigger/exception questions are marked with the relevant DBMS style.

Q1. Find all employees whose salary is greater than the average salary.

Answer / SQL Query:

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

Explanation: Uses a scalar subquery to compare each employee's salary with the company-wide average.

Q2. Find the second highest salary from the Employee table.

Answer / SQL Query:

```
SELECT MAX(salary) AS second_highest_salary
FROM Employee
WHERE salary < (SELECT MAX(salary) FROM Employee);
```

Explanation: Finds the maximum salary after excluding the highest salary.

Q3. Find the third highest distinct salary using DENSE_RANK().

Answer / SQL Query:

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

Explanation: DENSE_RANK assigns the same rank to duplicate salaries and returns the third distinct salary.

Q4. Display employees who earn more than their department average.

Answer / SQL Query:

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

Explanation: Calculates each department's average and compares each employee against it.

Q5. Find departments having more than 5 employees.

Answer / SQL Query:

```
SELECT department_id, COUNT(*) AS employee_count
FROM Employee
GROUP BY department_id
HAVING COUNT(*) > 5;
```

Explanation: HAVING filters grouped results after aggregation.

Q6. Find duplicate email addresses in a Student table.

Answer / SQL Query:

```
SELECT email, COUNT(*) AS occurrences
FROM Student
GROUP BY email
HAVING COUNT(*) > 1;
```

Explanation: Groups by email and keeps only repeated values.

Q7. Delete duplicate rows from Student while keeping the lowest student_id.

Answer / SQL Query:

```
WITH cte AS (
   SELECT student_id, email,
        ROW_NUMBER() OVER (PARTITION BY email ORDER BY student_id) AS rn
   FROM Student
)
DELETE FROM Student
WHERE student_id IN (SELECT student_id FROM cte WHERE rn > 1);
```

Explanation: ROW_NUMBER marks duplicates; rows after the first are deleted. Syntax is broadly supported with minor DB-specific variation.

Q8. Find employees who joined in the last 30 days.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE joining_date >= CURRENT_DATE - INTERVAL '30' DAY;
```

Explanation: Filters recent joining dates. Date arithmetic syntax may vary by DBMS.

Q9. Find employees whose names start with 'A' and end with 'n'.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE name LIKE 'A%n';
```

Explanation: The percent wildcard matches any sequence of characters between A and n.

Q10. Find employees whose names contain exactly five characters.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE name LIKE '_____';
```

Explanation: Each underscore represents exactly one character.

Q11. Display the top 3 highest-paid employees.

Answer / SQL Query:

```
SELECT *
FROM Employee
ORDER BY salary DESC
FETCH FIRST 3 ROWS ONLY;
```

Explanation: Sorts salaries in descending order and limits the result to three rows.

Q12. Find the highest salary in each department.

Answer / SQL Query:

```
SELECT department_id, MAX(salary) AS max_salary
FROM Employee
GROUP BY department_id;
```

Explanation: GROUP BY produces one result per department.

Q13. Display the employee(s) receiving the highest salary in each department.

Answer / SQL Query:

```
SELECT e.*
FROM Employee e
JOIN (
   SELECT department_id, MAX(salary) AS max_salary
   FROM Employee
   GROUP BY department_id
) m ON e.department_id = m.department_id
  AND e.salary = m.max_salary;
```

Explanation: Joins employees with the maximum salary calculated for their department.

Q14. Find employees who do not belong to any department.

Answer / SQL Query:

```
SELECT e.*
FROM Employee e
LEFT JOIN Department d ON e.department_id = d.department_id
WHERE d.department_id IS NULL;
```

Explanation: LEFT JOIN plus NULL filtering finds unmatched employee rows.

Q15. Find departments that currently have no employees.

Answer / SQL Query:

```
SELECT d.*
FROM Department d
LEFT JOIN Employee e ON d.department_id = e.department_id
WHERE e.employee_id IS NULL;
```

Explanation: Returns departments with no matching employee.

Q16. Display employee name along with department name.

Answer / SQL Query:

```
SELECT e.name AS employee_name, d.department_name
FROM Employee e
JOIN Department d ON e.department_id = d.department_id;
```

Explanation: An INNER JOIN combines matching employee and department records.

Q17. Find employees whose salary is between 50000 and 80000.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE salary BETWEEN 50000 AND 80000;
```

Explanation: BETWEEN includes both boundary values.

Q18. Find employees from departments 10, 20, or 30.

Answer / SQL Query:

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

Explanation: IN is convenient when matching any value from a fixed set.

Q19. Find employees whose department is neither 10 nor 20.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE department_id NOT IN (10, 20);
```

Explanation: NOT IN excludes the listed department IDs; be cautious when NULLs are involved.

Q20. Count employees in each department and sort from highest to lowest count.

Answer / SQL Query:

```
SELECT department_id, COUNT(*) AS employee_count
FROM Employee
GROUP BY department_id
ORDER BY employee_count DESC;
```

Explanation: Aggregates by department and sorts using the computed count alias.

Q21. Find departments whose average salary is greater than 70000.

Answer / SQL Query:

```
SELECT department_id, AVG(salary) AS avg_salary
FROM Employee
GROUP BY department_id
HAVING AVG(salary) > 70000;
```

Explanation: HAVING is used because the condition depends on an aggregate.

Q22. Find employees earning the same salary as employee 'Ravi'.

Answer / SQL Query:

```
SELECT *
FROM Employee
```

```
WHERE salary = (SELECT salary FROM Employee WHERE name = 'Ravi')
  AND name <> 'Ravi';
```

Explanation: A subquery fetches Ravi's salary, then the outer query finds other employees with the same salary.

Q23. Find the number of distinct departments in Employee.

Answer / SQL Query:

```
SELECT COUNT(DISTINCT department_id) AS department_count
FROM Employee;
```

Explanation: COUNT(DISTINCT ...) counts unique non-NULL department IDs.

Q24. Display salary with a 10% increment without updating the table.

Answer / SQL Query:

```
SELECT employee_id, name, salary, salary * 1.10 AS revised_salary
FROM Employee;
```

Explanation: A calculated expression shows the revised salary only in the result.

Q25. Increase salary by 10% for employees in department 20.

Answer / SQL Query:

```
UPDATE Employee
SET salary = salary * 1.10
WHERE department_id = 20;
```

Explanation: Updates only rows belonging to department 20.

Q26. Replace NULL commission values with 0 in a query result.

Answer / SQL Query:

```
SELECT employee_id, name, COALESCE(commission, 0) AS commission
FROM Employee;
```

Explanation: COALESCE returns the first non-NULL value.

Q27. Find employees whose manager_id is NULL.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE manager_id IS NULL;
```

Explanation: NULL comparisons must use IS NULL rather than = NULL.

Q28. Display employee and manager names using a self join.

Answer / SQL Query:

```
SELECT e.name AS employee_name, m.name AS manager_name
FROM Employee e
LEFT JOIN Employee m ON e.manager_id = m.employee_id;
```

Explanation: The Employee table is joined to itself to map each employee to a manager.

Q29. Find employees who earn more than their managers.

Answer / SQL Query:

```
SELECT e.name AS employee_name, e.salary,
    m.name AS manager_name, m.salary AS manager_salary
FROM Employee e
JOIN Employee m ON e.manager_id = m.employee_id
WHERE e.salary > m.salary;
```

Explanation: A self join makes it possible to compare employee salary with manager salary.

Q30. Find the total salary paid by each department.

Answer / SQL Query:

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

Explanation: SUM calculates the department-wise salary total.

Q31. Find the department with the highest total salary expense.

Answer / SQL Query:

```
SELECT department_id, SUM(salary) AS total_salary
FROM Employee
GROUP BY department_id
ORDER BY total_salary DESC
FETCH FIRST 1 ROW ONLY;
```

Explanation: Aggregates total salary by department and returns the largest one.

Q32. Find employees with the same department and salary as another employee.

Answer / SQL Query:

```
SELECT DISTINCT e1.*
FROM Employee e1
JOIN Employee e2
 ON e1.department_id = e2.department_id
AND e1.salary = e2.salary
AND e1.employee_id <> e2.employee_id;
```

Explanation: A self join finds rows that share both department and salary with another employee.

Q33. Display each department's minimum, maximum, and average salary.

Answer / SQL Query:

```
SELECT department_id,
    MIN(salary) AS min_salary,
    MAX(salary) AS max_salary,
    AVG(salary) AS avg_salary
FROM Employee
GROUP BY department_id;
```

Explanation: Uses multiple aggregate functions in a single grouped query.

Q34. Find employees whose salary is above 60000 and who joined after 1 January 2025.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE salary > 60000
  AND joining_date > DATE '2025-01-01';
```

Explanation: Combines numeric and date conditions using AND.

Q35. Find the latest joined employee.

Answer / SQL Query:

```
SELECT *
FROM Employee
ORDER BY joining_date DESC
FETCH FIRST 1 ROW ONLY;
```

Explanation: Descending date order places the latest joiner first.

Q36. Find all employees who joined on the same date as 'Amit'.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE joining_date = (SELECT joining_date FROM Employee WHERE name = 'Amit')
  AND name <> 'Amit';
```

Explanation: The subquery retrieves Amit's joining date for comparison.

Q37. Find the month-wise count of employees joining the company.

Answer / SQL Query:

```
SELECT EXTRACT(YEAR FROM joining_date) AS join_year,
       EXTRACT(MONTH FROM joining_date) AS join_month,
       COUNT(*) AS employee_count
FROM Employee
GROUP BY EXTRACT(YEAR FROM joining_date), EXTRACT(MONTH FROM joining_date)
ORDER BY join_year, join_month;
```

Explanation: Groups joining records by year and month.

Q38. Find employees with salary in the top 10% using PERCENT_RANK().

Answer / SQL Query:

```
SELECT *
FROM (
   SELECT e.*, PERCENT_RANK() OVER (ORDER BY salary DESC) AS pr
   FROM Employee e
) x
WHERE pr <= 0.10;
```

Explanation: Window ranking can identify employees near the top of the salary distribution.

Q39. Rank employees by salary within each department.

Answer / SQL Query:

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

Explanation: RANK assigns salary positions separately inside each department.

Q40. Assign a unique salary row number within each department.

Answer / SQL Query:

```
SELECT employee_id, name, department_id, salary,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS row_num
FROM Employee;
```

Explanation: ROW_NUMBER guarantees a unique sequence even when salaries are tied.

Q41. Find the difference between an employee's salary and the department average.

Answer / SQL Query:

```
SELECT employee_id, name, department_id, salary,
    salary - AVG(salary) OVER (PARTITION BY department_id) AS difference_from_avg
FROM Employee;
```

Explanation: A windowed AVG keeps individual rows while calculating the department average.

Q42. Show each employee's salary and the previous lower salary in descending order.

Answer / SQL Query:

```
SELECT employee_id, name, salary,
    LAG(salary) OVER (ORDER BY salary DESC) AS previous_salary
FROM Employee;
```

Explanation: LAG accesses the previous row in the salary-sorted result.

Q43. Show each employee's salary and the next salary in descending order.

Answer / SQL Query:

```
SELECT employee_id, name, salary,
    LEAD(salary) OVER (ORDER BY salary DESC) AS next_salary
FROM Employee;
```

Explanation: LEAD accesses the following row without a self join.

Q44. Calculate a running total of salaries ordered by employee_id.

Answer / SQL Query:

```
SELECT employee_id, name, salary,
    SUM(salary) OVER (ORDER BY employee_id) AS running_total
FROM Employee;
```

Explanation: A windowed SUM calculates a cumulative total.

Q45. Find students who scored above the average marks in an Exam table.

Answer / SQL Query:

```
SELECT *
FROM Exam
WHERE marks > (SELECT AVG(marks) FROM Exam);
```

Explanation: Compares each student's marks with the overall average.

Q46. Find the highest marks for each subject.

Answer / SQL Query:

```
SELECT subject_id, MAX(marks) AS highest_marks
FROM Exam
GROUP BY subject_id;
```

Explanation: Returns the subject-wise maximum score.

Q47. Find students who scored the highest marks in each subject.

Answer / SQL Query:

```
SELECT e.*
FROM Exam e
JOIN (
  SELECT subject_id, MAX(marks) AS max_marks
  FROM Exam
  GROUP BY subject_id
) m ON e.subject_id = m.subject_id AND e.marks = m.max_marks;
```

Explanation: Handles ties by joining each subject's maximum back to Exam.

Q48. Find students who have appeared in all subjects.

Answer / SQL Query:

```
SELECT student_id
FROM Exam
GROUP BY student_id
HAVING COUNT(DISTINCT subject_id) = (SELECT COUNT(*) FROM Subject);
```

Explanation: Relational division pattern: the student must have records for every subject.

Q49. Find students who have not appeared in any exam.

Answer / SQL Query:

```
SELECT s.*
FROM Student s
LEFT JOIN Exam e ON s.student_id = e.student_id
WHERE e.student_id IS NULL;
```

Explanation: Finds students with no matching rows in Exam.

Q50. Find the average marks for each student.

Answer / SQL Query:

```
SELECT student_id, AVG(marks) AS avg_marks
FROM Exam
GROUP BY student_id;
```

Explanation: Groups marks by student and calculates the average.

Q51. Find students whose average marks are above 75.

Answer / SQL Query:

```
SELECT student_id, AVG(marks) AS avg_marks
FROM Exam
```

```
GROUP BY student_id
HAVING AVG(marks) > 75;
```

Explanation: HAVING filters students based on their aggregated average.

Q52. Find customers who have placed more than 3 orders.

Answer / SQL Query:

```
SELECT customer_id, COUNT(*) AS order_count
FROM Orders
GROUP BY customer_id
HAVING COUNT(*) > 3;
```

Explanation: Counts orders per customer and filters frequent customers.

Q53. Find customers who have never placed an order.

Answer / SQL Query:

```
SELECT c.*
FROM Customer c
LEFT JOIN Orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

Explanation: Unmatched customers are found using LEFT JOIN.

Q54. Find the total purchase amount for each customer.

Answer / SQL Query:

```
SELECT customer_id, SUM(order_amount) AS total_purchase
FROM Orders
GROUP BY customer_id;
```

Explanation: SUM aggregates customer spending.

Q55. Find the customer with the highest total purchase amount.

Answer / SQL Query:

```
SELECT customer_id, SUM(order_amount) AS total_purchase
FROM Orders
GROUP BY customer_id
ORDER BY total_purchase DESC
FETCH FIRST 1 ROW ONLY;
```

Explanation: Returns the customer with the largest aggregated purchase amount.

Q56. Find orders placed in the current year.

Answer / SQL Query:

```
SELECT *
FROM Orders
WHERE EXTRACT(YEAR FROM order_date) = EXTRACT(YEAR FROM CURRENT_DATE);
```

Explanation: Extracts the year part from the order date and current date.

Q57. Find the number of orders placed each day.

Answer / SQL Query:

```
SELECT order_date, COUNT(*) AS order_count
FROM Orders
GROUP BY order_date
ORDER BY order_date;
```

Explanation: Groups orders by date.

Q58. Find products that have never been ordered.

Answer / SQL Query:

```
SELECT p.*
FROM Product p
LEFT JOIN OrderItem oi ON p.product_id = oi.product_id
WHERE oi.product_id IS NULL;
```

Explanation: Returns products without matching order items.

Q59. Find the best-selling product by quantity.

Answer / SQL Query:

```
SELECT product_id, SUM(quantity) AS total_quantity
FROM OrderItem
GROUP BY product_id
ORDER BY total_quantity DESC
FETCH FIRST 1 ROW ONLY;
```

Explanation: Aggregates total quantity sold per product and returns the highest.

Q60. Find the top 3 products by revenue.

Answer / SQL Query:

```
SELECT product_id, SUM(quantity * unit_price) AS revenue
FROM OrderItem
GROUP BY product_id
ORDER BY revenue DESC
FETCH FIRST 3 ROWS ONLY;
```

Explanation: Calculates product revenue and returns the top three.

Q61. Find orders whose amount is greater than that customer's average order amount.

Answer / SQL Query:

```
SELECT o.*
FROM Orders o
WHERE order_amount > (
  SELECT AVG(o2.order_amount)
  FROM Orders o2
  WHERE o2.customer_id = o.customer_id
);
```

Explanation: A correlated subquery compares an order with the average for the same customer.

Q62. Find employees who work in a department located in 'Bhopal'.

Answer / SQL Query:

```
SELECT e.*
FROM Employee e
JOIN Department d ON e.department_id = d.department_id
WHERE d.location = 'Bhopal';
```

Explanation: Filters employees using a joined department attribute.

Q63. Find the number of employees at each department location.

Answer / SQL Query:

```
SELECT d.location, COUNT(e.employee_id) AS employee_count
FROM Department d
LEFT JOIN Employee e ON d.department_id = e.department_id
GROUP BY d.location;
```

Explanation: Counts employees after joining departments to their locations.

Q64. Find employees whose salary is greater than ALL salaries in department 10.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE salary > ALL (
  SELECT salary FROM Employee WHERE department_id = 10
);
```

Explanation: The employee salary must be greater than every salary returned by the subquery.

Q65. Find employees whose salary is greater than ANY salary in department 10.

Answer / SQL Query:

```
SELECT *
FROM Employee
WHERE salary > ANY (
  SELECT salary FROM Employee WHERE department_id = 10
);
```

Explanation: The salary must be greater than at least one salary in department 10.

Q66. Use EXISTS to find departments that have at least one employee.

Answer / SQL Query:

```
SELECT d.*
FROM Department d
WHERE EXISTS (
  SELECT 1 FROM Employee e
  WHERE e.department_id = d.department_id
);
```

Explanation: EXISTS checks whether at least one matching employee row is present.

Q67. Use NOT EXISTS to find departments with no employees.

Answer / SQL Query:

```
SELECT d.*
FROM Department d
```

```
WHERE NOT EXISTS (
  SELECT 1 FROM Employee e
  WHERE e.department_id = d.department_id
);
```

Explanation: NOT EXISTS is a NULL-safe way to find departments without employees.

Q68. Find the intersection of employee IDs present in Employee_2025 and Employee_2026.

Answer / SQL Query:

```
SELECT employee_id FROM Employee_2025
INTERSECT
SELECT employee_id FROM Employee_2026;
```

Explanation: INTERSECT returns values common to both result sets in DBMSs that support it.

Q69. Combine unique employee names from two branch tables.

Answer / SQL Query:

```
SELECT name FROM BranchA_Employee
UNION
SELECT name FROM BranchB_Employee;
```

Explanation: UNION combines results and removes duplicates.

Q70. Combine all employee names from two branch tables including duplicates.

Answer / SQL Query:

```
SELECT name FROM BranchA_Employee
UNION ALL
SELECT name FROM BranchB_Employee;
```

Explanation: UNION ALL is faster when duplicate removal is not required.

Q71. Find records present in Employee_2025 but not in Employee_2026.

Answer / SQL Query:

```
SELECT employee_id FROM Employee_2025
EXCEPT
SELECT employee_id FROM Employee_2026;
```

Explanation: EXCEPT returns rows from the first result that are absent from the second; Oracle uses MINUS.

Q72. Create a view for high-salary employees.

Answer / SQL Query:

```
CREATE VIEW HighSalaryEmployees AS
SELECT employee_id, name, salary, department_id
FROM Employee
WHERE salary > 80000;
```

Explanation: A view stores a reusable query definition, not a separate copy of the rows in most systems.

Q73. Create an index on Employee(department_id).

Answer / SQL Query:

```
CREATE INDEX idx_employee_department
ON Employee(department_id);
```

Explanation: An index can speed up searches and joins on department_id, with some storage and write overhead.

Q74. Create a composite index on department_id and salary.

Answer / SQL Query:

```
CREATE INDEX idx_emp_dept_salary
ON Employee(department_id, salary);
```

Explanation: A composite index can help queries filtering by department and salary, especially when the leftmost column is used.

Q75. Write a transaction that transfers 5000 from account 101 to 202.

Answer / SQL Query:

```
START TRANSACTION;
UPDATE Account SET balance = balance - 5000 WHERE account_id = 101;
UPDATE Account SET balance = balance + 5000 WHERE account_id = 202;
COMMIT;
```

Explanation: Both updates should succeed as one transaction; in production, validation and rollback logic should also be included.

Q76. Create a stored procedure to increase salary for a department. (MySQL-style)

Answer / SQL Query:

```
DELIMITER //
CREATE PROCEDURE IncreaseDeptSalary(IN p_dept INT, IN p_percent DECIMAL(5,2))
BEGIN
  UPDATE Employee
  SET salary = salary * (1 + p_percent / 100)
  WHERE department_id = p_dept;
END //
DELIMITER ;
```

Explanation: The procedure accepts department and percentage parameters and performs a reusable update.

Q77. Create a trigger that prevents inserting a negative salary. (MySQL-style)

Answer / SQL Query:

```
DELIMITER //
CREATE TRIGGER trg_check_salary
BEFORE INSERT ON Employee
FOR EACH ROW
BEGIN
  IF NEW.salary < 0 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Salary cannot be negative';
  END IF;
END //
DELIMITER ;
```

Explanation: The BEFORE INSERT trigger validates salary and raises an error when the value is invalid.

Q78. Create a trigger to log salary changes. (MySQL-style)

Answer / SQL Query:

```
DELIMITER //
CREATE TRIGGER trg_salary_audit
AFTER UPDATE ON Employee
FOR EACH ROW
BEGIN
  IF OLD.salary <> NEW.salary THEN
    INSERT INTO SalaryAudit(employee_id, old_salary, new_salary, changed_at)
    VALUES (NEW.employee_id, OLD.salary, NEW.salary, CURRENT_TIMESTAMP);
  END IF;
END //
DELIMITER ;
```

Explanation: The trigger writes an audit row only when the salary actually changes.

Q79. Write an exception-handling block for division by zero. (Oracle PL/SQL)

Answer / SQL Query:

```
DECLARE
  v_result NUMBER;
BEGIN
  v_result := 100 / 0;
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Division by zero is not allowed');
END;
/
```

Explanation: The ZERO_DIVIDE exception is caught so the block handles the runtime error gracefully.

Q80. Write an exception-handling block for a missing employee. (Oracle PL/SQL)

Answer / SQL Query:

```
DECLARE
  v_name Employee.name%TYPE;
BEGIN
  SELECT name INTO v_name
  FROM Employee
  WHERE employee_id = 9999;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee not found');
END;
/
```

Explanation: NO_DATA_FOUND is raised when SELECT INTO returns no row.