SQL

Nice, this one is super important for this JD (.NET + SQL Server).

Here's a **focused question bank only on:**

- SQL queries
- Joins / subqueries
- Stored procedures & functions
- Real-time scenarios with SQL Server

(Questions only – so you can use as checklist.)

🔷 A. Basic to Intermediate SQL Queries

1) Write a query to fetch all employees with salary > 50,000.

■ Explanation

This query filters rows based on a condition using the `WHERE` clause.

■ When/Why is it used

Used when you want to retrieve only employees who meet a minimum salary requirement, often for reporting or analytics.

■ Example / Code snippet

```
SELECT *
FROM Employees
WHERE Salary > 50000;
```

■ Short summary line

Use `WHERE` to filter employees with salary greater than a given threshold.

---

2) Write a query to get all employees in “IT” department.

■ Explanation

We filter records based on department name using `WHERE`.

■ When/Why is it used

Used to fetch department-specific data for HR, payroll, auditing, or analytics.

■ Example / Code snippet

```
SELECT *
FROM Employees
WHERE Department = 'IT';
```

■ Short summary line

Filter by department using a simple `WHERE` condition.

---

3) Write a query to get employees whose name starts with ‘A’.

■ Explanation

The `LIKE` operator with `%` wildcard is used for pattern matching.

■ When/Why is it used

Used for searching names alphabetically or performing partial text searches.

■ Example / Code snippet

```
SELECT *
FROM Employees
WHERE Name LIKE 'A%';
```

■ Short summary line

Use `LIKE 'A%'` to fetch names starting with a specific letter.

---

4) Write a query to get employees whose name contains ‘an’.

■ **Explanation**

`%an%` finds any name where the substring “an” appears anywhere.

■ **When/Why is it used**

Useful for flexible text search, auto-suggestions, and filtering user lists.

■ **Example / Code snippet**

```
SELECT *
FROM Employees
WHERE Name LIKE '%an%';
```

■ **Short summary line**

Use `LIKE '%an%'` for substring-based search in names.

---

5) How do you select only distinct values from a column?

■ **Explanation**

`DISTINCT` removes duplicate values from the result set.

■ **When/Why is it used**

Used when analyzing unique data—such as unique departments, job titles, or cities.

■ **Example / Code snippet**

```
SELECT DISTINCT Department
FROM Employees;
```

■ **Short summary line**

`DISTINCT` is used to fetch unique values and remove duplicates.

6) How to sort results by multiple columns (e.g., dept ASC, salary DESC)?

■ Explanation

We use the `ORDER BY` clause and specify multiple columns with asc/desc sorting.

■ When/Why is it used

Used when displaying organized reports—for example, sorting by department first and then sorting employees within each department by salary.

■ Example / Code snippet

```
SELECT *
FROM Employees
ORDER BY Department ASC, Salary DESC;
```

■ Short summary line

Use `ORDER BY column1 ASC, column2 DESC` to apply multi-column sorting.

---

7) How do you limit top N records? (TOP, LIMIT, FETCH NEXT)

■ Explanation

Different SQL engines use different keywords; SQL Server uses `TOP`, SQL standard uses `FETCH`.

■ When/Why is it used

Used for pagination, dashboards, top-performers list, or restricting large datasets.

■ Example / Code snippet (SQL Server)

```
SELECT TOP 5 *
FROM Employees;
```

■ Using OFFSET–FETCH (SQL Server / Standard SQL)

```
SELECT *
FROM Employees
ORDER BY Salary DESC
OFFSET 0 ROWS FETCH NEXT 5 ROWS ONLY;
```

■ Short summary line

Use `TOP` in SQL Server or `OFFSET...FETCH` for paginated limited results.

---

8) Write a query to count total employees in each department.

■ Explanation

We use `GROUP BY` to aggregate rows department-wise and `COUNT()` to count employees.

■ When/Why is it used

Used for HR dashboards, reporting team size, and department-level analysis.

■ Example / Code snippet

```
SELECT Department, COUNT(*) AS TotalEmployees
FROM Employees
GROUP BY Department;
```

■ Short summary line

`GROUP BY` + `COUNT()` gives employee counts for each department.

---

9) Write a query to find min, max, avg salary in a table.

■ Explanation

Aggregate functions like MIN, MAX, AVG are used on numeric columns to analyze salary distribution.

■ When/Why is it used

Used in payroll, analytics, and calculating salary ranges for budgeting.

■ **Example / Code snippet**

```
SELECT
   MIN(Salary) AS MinSalary,
   MAX(Salary) AS MaxSalary,
   AVG(Salary) AS AvgSalary
FROM Employees;
```

■ **Short summary line**

Use `MIN`, `MAX`, and `AVG` to get basic statistical salary insights.

---

10) How to group data by department and filter groups with AVG(salary) > X?

■ **Explanation**

We use `HAVING` to filter aggregated results **after** grouping.

■ **When/Why is it used**

Used to identify high-paying departments or departments exceeding certain performance thresholds.

■ **Example / Code snippet**

```
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY Department
HAVING AVG(Salary) > 60000;
```

■ **Short summary line**

Use `HAVING` to filter aggregated groups such as departments with high average salary.

---

11) Difference between WHERE and HAVING? Give examples

■ **Explanation**

- `WHERE` filters rows **before grouping**
- `HAVING` filters results **after grouping**

■ **When/Why is it used**

- Use `WHERE` for normal row filtering
- Use `HAVING` when filtering aggregated results like AVG, SUM, COUNT

■ **Example / Code snippet**

✔ **WHERE example:**

```
SELECT *
FROM Employees
WHERE Salary > 50000;
```

✔ **HAVING example:**

```
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY Department
HAVING AVG(Salary) > 60000;
```

■ **Short summary line**

`WHERE` filters rows, `HAVING` filters groups formed by `GROUP BY`

---

12) How to find records between two dates?

■ **Explanation**

To filter rows between two date/time values you can use `BETWEEN` (inclusive) or explicit range comparisons (`>=` and `<`) — the latter is safer for datetime boundaries (avoids off-by-one when times are involved).

■ **When/Why is it used**

Used to get records in a date window such as last month's transactions, weekly reports, or audit logs.

■ **Example / Code snippet**

```
-- Using BETWEEN (inclusive of start and end)
SELECT *
FROM Orders
WHERE OrderDate BETWEEN '2025-01-01' AND '2025-01-31';

-- Safer approach for datetime: inclusive start, exclusive end
SELECT *
FROM Orders
WHERE OrderDate >= '2025-01-01'
  AND OrderDate <  '2025-02-01';
```

■ Short summary line

Use `BETWEEN` for inclusive ranges, but prefer `>= start AND < end` for datetime ranges to avoid boundary/time issues.

---

13) How to get current date and time in SQL Server?

■ Explanation

SQL Server provides built-in functions: `GETDATE()` returns the current datetime (precision up to milliseconds), and `SYSDATETIME()` returns a more precise datetime2 value.

■ When/Why is it used

Used for timestamps on inserts/updates, logging, calculating time differences, and default values.

■ Example / Code snippet

```
-- Current date & time (datetime)
SELECT GETDATE() AS CurrentDateTime;

-- Higher precision (datetime2)
SELECT SYSDATETIME() AS CurrentDateTime2;
```

```
-- Only date (SQL Server 2008+)
SELECT CAST(GETDATE() AS DATE) AS TodayDate;
```

■ Short summary line

Use `GETDATE()` for current datetime and `SYSDATETIME()` if you need higher precision.

---

14) Write a query to find employees who don't have manager assigned (NULL).

■ Explanation

In SQL `NULL` represents missing values; use `IS NULL` (not `= NULL`) to test for absence.

■ When/Why is it used

Used to identify top-level employees, incomplete data, or records that need attention (e.g., missing foreign key).

■ Example / Code snippet

```
SELECT EmployeeId, Name, ManagerId
FROM Employees
WHERE ManagerId IS NULL;
```

■ Short summary line

Use `IS NULL` to find rows that have no value in a column (e.g., employees without managers).

---

15) How to update a specific column for specific rows?

■ Explanation

`UPDATE` with a `SET` clause changes specified columns; always include a `WHERE` clause to limit affected rows, otherwise the whole table is updated.

■ When/Why is it used

Used for correcting data, applying business rules, or bulk changes (salary hikes, status changes).

■ Example / Code snippet

```
-- Give a 10% raise to employees in 'IT' with salary < 60000
UPDATE Employees
SET Salary = Salary * 1.10
WHERE Department = 'IT'
  AND Salary < 60000;

-- Always check before update
SELECT * FROM Employees WHERE Department = 'IT' AND Salary < 6000
0;
```

■ Short summary line

Use `UPDATE ... SET ... WHERE ...` and always verify the `WHERE` condition before executing to avoid mass updates.

---

16) Write query to delete records older than 1 year.

■ Explanation

Use `DELETE` with a date comparison using `DATEADD()` to compute the cutoff; consider transaction safety and backups for destructive operations.

■ When/Why is it used

Used for cleanup tasks like archival, log pruning, or GDPR retention enforcement.

■ Example / Code snippet

```
-- Delete old audit logs older than 1 year from today
BEGIN TRAN;
DELETE FROM AuditLogs
WHERE EventDate < DATEADD(year, -1, GETDATE());
-- Optionally check rows affected, then
COMMIT;
```

```
-- Safer pattern: move to an archive table first
BEGIN TRAN;
INSERT INTO Archive_AuditLogs
SELECT * FROM AuditLogs
WHERE EventDate < DATEADD(year, -1, GETDATE());

DELETE FROM AuditLogs
WHERE EventDate < DATEADD(year, -1, GETDATE());
COMMIT;
```

■ Short summary line

Use `DELETE ... WHERE DateColumn < DATEADD(year, -1, GETDATE())` and prefer transactional or archive-first approaches for safety.

---

17) How to rename a column in result set (alias)?

■ Explanation

Column aliases rename output columns using `AS` (optional keyword) — this doesn't change the underlying table schema, only the result set header.

■ When/Why is it used

Used to improve readability of results, match API/DTO field names, or present computed columns.

■ Example / Code snippet

```
-- Simple alias
SELECT EmployeeId AS Id, Name AS EmployeeName, Salary AS CurrentSal
ary
FROM Employees;

-- Alias for computed/concatenated column
SELECT EmployeeId,
      FirstName + ' ' + LastName AS FullName,
```

```
    Salary AS CurrentSalary
FROM Employees;
```

■ Short summary line

Use `AS` to provide friendly or API-aligned column names in query results without modifying the table.

---

18) What is the difference between `IN` and `EXISTS`?

■ Explanation

`IN` tests membership against a (possibly static) list or subquery result; `EXISTS` tests whether a correlated subquery returns any row. `EXISTS` short-circuits on first match and is usually better for correlated checks; `IN` compares values and may behave differently with `NULL`s.

■ When/Why is it used

- Use `IN` when you have a small, non-correlated list or subquery that returns distinct scalar values.
- Use `EXISTS` when checking existence of related rows (often correlated) or when you want performance benefit on large datasets with proper indexes.

■ Example / Code snippet

```
-- Using IN (non-correlated)
SELECT e.*
FROM Employees e
WHERE e.DepartmentId IN (1, 2, 3);

-- Using IN with subquery
SELECT e.*
FROM Employees e
WHERE e.DepartmentId IN (
    SELECT DepartmentId FROM Departments WHERE IsActive = 1
);

-- Using EXISTS (correlated subquery)
```

```
SELECT e.*
FROM Employees e
WHERE EXISTS (
    SELECT 1
    FROM Departments d
    WHERE d.DepartmentId = e.DepartmentId
      AND d.IsActive = 1
);

-- Note on NULL behavior:
-- If subquery returns NULL as a member, IN may yield unexpected results;
EXISTS is null-safe in correlated existence checks.
```

■ Short summary line

`IN` checks membership against a list or subquery values; `EXISTS` checks for the presence of rows and often performs better for correlated-existence checks and large datasets.

🔷 B. Joins – Very Important

1) What is a JOIN? Why do we use it?

■ Explanation

A JOIN combines rows from two or more tables based on a related column (usually a foreign key). It connects relational data stored in different tables into a single result set.

■ When/Why is it used

Used when you need meaningful combined information—for example, joining Employees with Departments to fetch department names. It ensures normalized tables can be read together meaningfully.

■ Example / Code snippet

```
SELECT e.Name, d.DepartmentName
FROM Employees e
```

```
JOIN Departments d
  ON e.DepartmentId = d.DepartmentId;
```

■ **Short summary line**

Joins link related tables using a common key to return combined, meaningful data.

---

2) Explain INNER JOIN with an example.

■ **Explanation**

INNER JOIN returns only the matching rows from both tables. If a row does not have a match in the other table, it is excluded.

■ **When/Why is it used**

Used when you want only the valid, fully-connected data—like employees who belong to a department that exists.

■ **Example / Code snippet**

```
SELECT e.Name, d.DepartmentName
FROM Employees e
INNER JOIN Departments d
   ON e.DepartmentId = d.DepartmentId;
```

Result: Only employees whose DepartmentId exists in Departments will appear.

■ **Short summary line**

INNER JOIN gives only matching records from both tables.

---

3) Explain LEFT JOIN with an example.

■ **Explanation**

LEFT JOIN returns all rows from the left table and matching rows from the right table. If a match is missing, the right table's columns become NULL.

■ **When/Why is it used**

Used when you want all left-side data—even if corresponding data is missing.

Example: fetch all employees, including those not assigned to any department.

■ **Example / Code snippet**

```
SELECT e.Name, d.DepartmentName
FROM Employees e
LEFT JOIN Departments d
    ON e.DepartmentId = d.DepartmentId;
```

Result: Employees with no department will show `NULL` for DepartmentName.

■ **Short summary line**

LEFT JOIN keeps all left table rows and fills missing right-side values with NULL.

---

4) RIGHT JOIN vs LEFT JOIN — when to use?

■ **Explanation**

- LEFT JOIN → all rows from left table + matched from right
- RIGHT JOIN → all rows from right table + matched from left

Functionally they are mirror images.

■ **When/Why is it used**

Use whichever direction keeps the “complete” dataset readable.

- Use **LEFT JOIN** if your primary table is on left (most common case).
- Use **RIGHT JOIN** if your base table is on the right side and you want all its rows.

■ **Example / Code snippet**

```
-- Left join (all employees)
SELECT e.Name, d.DepartmentName
FROM Employees e
LEFT JOIN Departments d
    ON e.DepartmentId = d.DepartmentId;

-- Right join (all departments)
SELECT e.Name, d.DepartmentName
```

```
FROM Employees e
RIGHT JOIN Departments d
    ON e.DepartmentId = d.DepartmentId;
```

■ **Short summary line**

LEFT JOIN keeps all left rows; RIGHT JOIN keeps all right rows — choose based on which table you want fully preserved.

---

5) FULL OUTER JOIN use-case

■ **Explanation**

FULL OUTER JOIN returns **all rows from both tables**, whether matched or unmatched. Unmatched values are filled with NULLs.

■ **When/Why is it used**

Used when you need a **complete comparison** of both tables—for example:

- audit two tables for mismatched records
- find which employees have invalid departments
- find departments with no employees

■ **Example / Code snippet**

```
SELECT e.Name, d.DepartmentName
FROM Employees e
FULL OUTER JOIN Departments d
    ON e.DepartmentId = d.DepartmentId;
```

■ **Short summary line**

FULL OUTER JOIN returns everything from both tables, matching where possible and showing NULLs where not.

---

6) What is CROSS JOIN? Where is it used?

■ **Explanation**

CROSS JOIN returns the **Cartesian product** of two tables — every row of table A combined with every row of table B. It does **not** require any join condition.

■ **When/Why is it used**

Used rarely, mainly for:

- Generating combinations (e.g., all dates × all shifts)
- Creating sample/mock data
- Producing matrix-style reports
- Building dimension tables in BI

■ **Example / Code snippet**

```
SELECT e.Name, d.DepartmentName
FROM Employees e
CROSS JOIN Departments d;
```

If Employees = 10 rows and Departments = 5, result = 50 rows.

■ **Short summary line**

CROSS JOIN creates all possible row combinations and is used for generating combinations or synthetic datasets.

---

7) Write query to get employee name and department name using JOIN.

■ **Explanation**

We join Employees with Departments using a foreign key (DepartmentId).

■ **When/Why is it used**

Used in almost all real projects to show employee details along with department info in UI/API responses.

■ **Example / Code snippet**

```
SELECT e.Name AS EmployeeName,
       d.DepartmentName
FROM Employees e
```

```
INNER JOIN Departments d
    ON e.DepartmentId = d.DepartmentId;
```

■ **Short summary line**

Use INNER JOIN between Employees and Departments to fetch combined meaningful info.

---

8) Query to fetch all departments even if they have no employees.

■ **Explanation**

We want all departments, so we start from the `Departments` table and use a **LEFT JOIN**.

■ **When/Why is it used**

Useful for admin dashboards where empty departments must still be shown.

■ **Example / Code snippet**

```
SELECT d.DepartmentName,
       e.Name AS EmployeeName
FROM Departments d
LEFT JOIN Employees e
    ON d.DepartmentId = e.DepartmentId;
```

■ **Short summary line**

Use LEFT JOIN starting from Departments to show departments with or without employees.

---

9) Self Join — what is it? Example use-case (manager–employee).

■ **Explanation**

A self join is when a table is joined with itself. Useful when rows have hierarchical or parent-child relationships.

■ **When/Why is it used**

Used for:

- Employee–manager hierarchy
- Category–subcategory structure
- Tree or recursive relationships

■ **Example / Code snippet**

```
SELECT e.Name AS Employee,
       m.Name AS Manager
FROM Employees e
LEFT JOIN Employees m
    ON e.ManagerId = m.EmployeeId;
```

■ **Short summary line**

Self join is joining a table with itself to represent hierarchical relationships like employee–manager.

---

10) Query to list manager and their direct report employees.

■ **Explanation**

We join the Employees table with itself — an employee's ManagerId points to another EmployeeId.

■ **When/Why is it used**

Useful for org charts, reporting structure, workflow approvals.

■ **Example / Code snippet**

```
SELECT
    m.Name AS Manager,
    e.Name AS Employee
FROM Employees e
INNER JOIN Employees m
    ON e.ManagerId = m.EmployeeId;
```

■ **Short summary line**

Use a self join (employee → manager) to list managers with their reporting employees.

---

11) Difference between JOIN and Subquery — which is faster generally?

■ **Explanation**

- JOIN: Combines data from multiple tables in a single result set.
- Subquery: A query inside another query; can return values or datasets.

■ **When/Why is it used**

- Use **JOIN** when needing combined datasets or relational linking.
- Use **Subquery** when performing isolated filtering, aggregation, or existence checks.

■ **Performance (Generally)**

- **JOIN is usually faster** because SQL optimizes them better and works with indexes directly.
- **Subqueries (especially correlated ones)** may run row-by-row and be slower.

■ **Example / Code snippet**

✔ **JOIN approach (faster usually):**

```
SELECT e.Name, d.DepartmentName
FROM Employees e
JOIN Departments d
  ON e.DepartmentId = d.DepartmentId;
```

✔ **Subquery approach:**

```
SELECT Name,
       (SELECT DepartmentName
        FROM Departments d
        WHERE d.DepartmentId = e.DepartmentId) AS DepartmentName
FROM Employees e;
```

■ **Short summary line**

JOINs are generally faster and preferred, while subqueries are useful for hierarchical or filtered logic.

🔷 C. Subqueries & CTE

1) What is a subquery?

■ **Explanation**

A subquery is a query written inside another query. It provides a result that the outer query uses for filtering, comparison, or computation.

■ **When/Why is it used**

Used when a value or a set of values must be calculated dynamically—for example, comparing employee salary with average salary, or checking existence in another table.

■ **Example / Code snippet**

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

■ **Short summary line**

A subquery is an inner query whose result is used by an outer query for filtering or computation.

---

2) Where can we use subquery? (SELECT, FROM, WHERE, HAVING)

■ **Explanation**

Subqueries can appear in multiple clauses depending on what output they generate—scalar, list, or table.

■ **When/Why is it used**

Used for flexible filtering, dynamic calculations, and complex conditions without needing joins.

■ **Example / Code snippet**

✔ **In SELECT (scalar subquery)**

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

✔ **In FROM (inline view / derived table)**

```
SELECT *
FROM (SELECT DepartmentId, COUNT(*) AS TotalEmployees
     FROM Employees GROUP BY DepartmentId) deptSummary;
```

✔ **In WHERE**

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

✔ **In HAVING**

```
SELECT DepartmentId, AVG(Salary) AS AvgSal
FROM Employees
GROUP BY DepartmentId
HAVING AVG(Salary) > (SELECT AVG(Salary) FROM Employees);
```

■ **Short summary line**

Subqueries can appear in SELECT, FROM, WHERE, and HAVING clauses to supply dynamic or calculated values.

---

3) Write a query to get employees whose salary is above average salary (subquery).

■ **Explanation**

We compare each employee's salary with the result of an aggregate function (AVG) returned by a subquery.

■ **When/Why is it used**

Used for identifying above-average performers or running salary audits.

■ **Example / Code snippet**

```
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

■ **Short summary line**

Use a scalar subquery with AVG to filter employees whose salary is above the organization's average.

---

4) What is a correlated subquery? Example.

■ **Explanation**

A correlated subquery is a subquery that depends on each row of the outer query. It runs once per row, making it row-by-row evaluation.

■ **When/Why is it used**

Used when filtering depends on related data for each row—like checking each employee’s salary vs. department average.

■ **Example / Code snippet**

```
SELECT e.Name, e.Salary
FROM Employees e
WHERE e.Salary >
    (SELECT AVG(Salary)
     FROM Employees
     WHERE DepartmentId = e.DepartmentId);
```

The inner query uses `e.DepartmentId` — hence it’s correlated.

■ Short summary line

A correlated subquery depends on outer query values and executes once per row.

---

5) What is CTE (Common Table Expression)?

■ Explanation

A CTE is a temporary named result set defined using `WITH`. It's like a readable alternative to subqueries and can be referenced multiple times.

■ When/Why is it used

Used to improve query readability, perform recursive operations (like hierarchy), and break complex queries into logical steps.

■ Example / Code snippet

```
WITH DeptCount AS (
    SELECT DepartmentId, COUNT(*) AS TotalEmployees
    FROM Employees
    GROUP BY DepartmentId
)
SELECT *
FROM DeptCount
WHERE TotalEmployees > 5;
```

■ Short summary line

A CTE is a named temporary result set created with `WITH`, used for cleaner queries and recursive operations.

---

6) Difference between CTE and temporary table?

■ Explanation

A CTE (Common Table Expression) is a named, in-memory result set defined with `WITH` that exists only for the single statement that follows it. A temporary

table (e.g., `#TempTable`) is a real table object created in tempdb that persists for the user session (or procedure scope) and can be indexed, have statistics, and be reused across multiple statements within that session.

■ When/Why is it used

- Use a **CTE** for readable, composable queries, breaking complex logic into logical steps, or for simple recursive queries. It's best when you only need the intermediate result inside one statement and when readability matters.
- Use a **temporary table** when you need to persist intermediate results across multiple statements, create indexes or statistics for performance on large datasets, perform incremental processing, or when you need to inspect intermediate data during debugging.

■ Example / Code snippet

```
-- Example CTE (single-statement use, no indexes)
WITH DeptTotals AS (
    SELECT DepartmentId, COUNT(*) AS EmpCount
    FROM Employees
    GROUP BY DepartmentId
)
SELECT d.DepartmentId, d.EmpCount
FROM DeptTotals d
WHERE EmpCount > 5;

-- Example temporary table (persist across multiple statements, can index)
SELECT DepartmentId, COUNT(*) AS EmpCount
INTO #DeptTotals
FROM Employees
GROUP BY DepartmentId;

-- Create an index if needed for further joins / filters
CREATE INDEX IX_DeptTotals_EmpCount ON #DeptTotals(EmpCount);

-- Use temp table in another statement
SELECT d.DepartmentId, d.EmpCount
FROM #DeptTotals d
WHERE d.EmpCount > 5;
```

```
DROP TABLE #DeptTotals;
```

Key practical differences to mention in interview:

- **Scope:** CTE — single statement; Temp table — session/proc scope.
- **Materialization:** CTE is generally not materialized (optimizer can inline it); temp table physically stored in tempdb (has stats).
- **Performance control:** Temp tables allow indexes/statistics and can improve performance on large intermediate datasets; CTEs rely on query optimizer and may be re-evaluated.
- **Recursion:** CTE supports recursion natively; temp tables do not provide built-in recursive syntax.

■ Short summary line

CTEs are lightweight, single-statement named queries great for readability and recursion; temp tables are physical, session-scoped objects you use when you need persistence, indexing, or multi-statement processing.

---

7) Write recursive CTE to get hierarchy (e.g., categories, org tree).

■ Explanation

A recursive CTE has two parts: an anchor member that selects root rows, and a recursive member that references the CTE to walk the hierarchy. SQL Server repeats the recursive member until no more rows are produced. You can add a `LEVEL` / `Depth` column and a `Path` to show the chain and optionally control recursion with `OPTION (MAXRECURSION n)`.

■ When/Why is it used

Used to traverse parent–child relationships (organization charts, category trees, BOMs) in a clear, set-based way without writing procedural loops.

■ Example / Code snippet

```
-- Table schema (example)
-- Employees(EmployeeId INT PRIMARY KEY, Name NVARCHAR(100), Man
agerId INT NULL)
```

```
-- Recursive CTE to get org hierarchy with depth and path
WITH OrgCTE AS (
    -- Anchor member: top-level managers (no manager)
    SELECT
        EmployeeId,
        Name,
        ManagerId,
        0 AS Level,
        CAST(Name AS NVARCHAR(MAX)) AS HierarchyPath
    FROM Employees
    WHERE ManagerId IS NULL

    UNION ALL

    -- Recursive member: join employees to previously found managers
    SELECT
        e.EmployeeId,
        e.Name,
        e.ManagerId,
        c.Level + 1 AS Level,
        CAST(c.HierarchyPath + ' > ' + e.Name AS NVARCHAR(MAX)) AS Hier
archyPath
    FROM Employees e
    INNER JOIN OrgCTE c
        ON e.ManagerId = c.EmployeeId
)
SELECT EmployeeId, Name, ManagerId, Level, HierarchyPath
FROM OrgCTE
ORDER BY Level, ManagerId, EmployeeId
OPTION (MAXRECURSION 0); -- 0 = no limit, or set a number to prevent ru
naway recursion
```

Notes / practical tips:

- Use `OPTION (MAXRECURSION n)` to protect against infinite loops — default is 100.
- If your hierarchy is deep and `HierarchyPath` grows large, prefer `VARCHAR(MAX)` / `NVARCHAR(MAX)` and be mindful of performance.

- For very large trees, consider materializing intermediate results into a temp table for indexing and further processing.

■ **Short summary line**

Recursive CTEs provide a concise, set-based way to traverse parent–child hierarchies, producing depth and path information with an anchor and recursive member.

🔷 D. Classic Query Puzzles (Almost Always Asked)

1) Query to find 2nd highest salary from employee table.

■ **Explanation**

We want the salary that is the second largest value in the Salary column. Common safe approaches avoid `DISTINCT` pitfalls with NULLs and handle duplicates properly.

■ **When/Why is it used**

Used in analytics (e.g., show runner-up compensation), ranking scenarios, or business rules that depend on relative positions.

■ **Example / Code snippet (three concise approaches)**

a) Using a subquery (works when there is at least one lower value):

```
SELECT MAX(Salary) AS SecondHighestSalary
FROM Employees
WHERE Salary < (SELECT MAX(Salary) FROM Employees);
```

b) Using DENSE_RANK() to handle ties (recommended):

```
WITH Ranked AS (
  SELECT Salary,
         DENSE_RANK() OVER (ORDER BY Salary DESC) AS rnk
  FROM Employees
)
SELECT Salary AS SecondHighestSalary
```

```
FROM Ranked
WHERE rnk = 2;
```

c) Using OFFSET–FETCH (SQL Server):

```
SELECT DISTINCT Salary
FROM Employees
ORDER BY Salary DESC
OFFSET 1 ROWS FETCH NEXT 1 ROW ONLY;  -- second distinct salary
```

■ Short summary line

Use `DENSE_RANK()` to reliably get the 2nd highest value (handles ties); simpler subqueries work if duplicates aren't a concern.

---

2) Multiple ways to find 2nd highest salary (TOP, subquery, CTE).

■ Explanation

There are several patterns: aggregate + subquery, window functions (ROW_NUMBER/RANK/DENSE_RANK), TOP with OFFSET or TOP with nested queries. Choice depends on tie-handling and readability.

■ When/Why is it used

Different interviewers expect different approaches — show knowledge of aggregates, window functions, and pagination methods.

■ Example / Code snippet (compact showcase)

TOP with nested query:

```
-- If you want the single second highest (may ignore ties)
SELECT TOP 1 Salary
FROM (
  SELECT DISTINCT TOP 2 Salary
  FROM Employees
  ORDER BY Salary DESC
) AS t
ORDER BY Salary ASC;
```

CTE + ROW_NUMBER (returns one row per employee even with same salary):

```
WITH C AS (
  SELECT Salary,
        ROW_NUMBER() OVER (ORDER BY Salary DESC) AS rn
  FROM Employees
)
SELECT Salary FROM C WHERE rn = 2;
```

CTE + DENSE_RANK (handles ties and returns all employees at 2nd rank):

```
WITH C AS (
  SELECT Salary,
        DENSE_RANK() OVER (ORDER BY Salary DESC) AS dr
  FROM Employees
)
SELECT Salary FROM C WHERE dr = 2;
```

■ Short summary line

Show multiple options: `TOP+n` nested subquery, `ROW_NUMBER()` for exact position, `DENSE_RANK()` to handle ties — prefer window functions for clarity.

---

3) Query to find Nth highest salary.

■ Explanation

Generalize the ranking approach: use window functions to rank and then filter for the desired rank (N). This handles ties and is efficient with proper indexing.

■ When/Why is it used

When interview asks for Nth (not just 2nd) or when building dynamic reporting where N is parameterized.

■ Example / Code snippet

```
-- Replace @N with desired N (e.g., 3 for 3rd highest)
DECLARE @N INT = 3;

WITH Ranked AS (
```

```
  SELECT Salary,
        DENSE_RANK() OVER (ORDER BY Salary DESC) AS dr
  FROM Employees
)
SELECT Salary
FROM Ranked
WHERE dr = @N;
```

If you need the Nth row (including duplicates treated separately):

```
WITH RN AS (
  SELECT Salary,
        ROW_NUMBER() OVER (ORDER BY Salary DESC) AS rn
  FROM Employees
)
SELECT Salary FROM RN WHERE rn = @N;
```

■ Short summary line

Use `DENSE_RANK()` for Nth distinct highest salary and `ROW_NUMBER()` when you want the Nth row position (including duplicates as separate rows).

---

4) Query to find employees with duplicate email IDs.

■ Explanation

Find emails that appear more than once via `GROUP BY` + `HAVING`, then join back to list employees with those emails.

■ When/Why is it used

Used in data quality checks, user account deduplication, and migration validation.

■ Example / Code snippet

```
-- Find duplicate email values
SELECT Email, COUNT(*) AS Cnt
FROM Employees
GROUP BY Email
```

```
HAVING COUNT(*) > 1;

-- List employees who share duplicate emails
SELECT e.EmployeeId, e.Name, e.Email
FROM Employees e
JOIN (
    SELECT Email
    FROM Employees
    GROUP BY Email
    HAVING COUNT(*) > 1
) dup ON e.Email = dup.Email
ORDER BY e.Email, e.EmployeeId;
```

■ Short summary line

Group by Email with `HAVING COUNT(*) > 1` then join back to list the duplicate rows for cleanup.

---

5) Query to delete duplicate rows and keep only one.

■ Explanation

Use a CTE with `ROW_NUMBER()` partitioned by the duplicate-defining columns and delete rows where row number > 1 — this keeps the first occurrence and removes others.

■ When/Why is it used

Use when cleaning datasets after imports, or enforcing uniqueness before adding a unique constraint.

■ Example / Code snippet

```
-- Assume duplicates defined by (Email) and we keep the lowest Employeel
d
WITH Dups AS (
  SELECT *,
        ROW_NUMBER() OVER (PARTITION BY Email ORDER BY EmployeeId)
AS rn
  FROM Employees
```

```
)
DELETE FROM Dups
WHERE rn > 1;
```

Alternative (if you can't delete directly from CTE in some dialects):

```
WITH Dups AS (
  SELECT EmployeeId,
       ROW_NUMBER() OVER (PARTITION BY Email ORDER BY EmployeeId)
AS rn
  FROM Employees
)
DELETE e
FROM Employees e
JOIN Dups d ON e.EmployeeId = d.EmployeeId
WHERE d.rn > 1;
```

Safety tip: first `SELECT * FROM Dups WHERE rn > 1` to review rows before deleting.

■ **Short summary line**

Use `ROW_NUMBER() OVER (PARTITION BY <dup_cols> ORDER BY <keep_rule>)` and delete rows with `rn > 1` to remove duplicates safely.

---

6) Find department with maximum number of employees.

■ **Explanation**

Aggregate employee counts per department and pick the department with the highest count. Use window functions to handle ties elegantly.

■ **When/Why is it used**

Used for reporting top teams, capacity planning, resource allocation.

■ **Example / Code snippet**

Simple: TOP 1

```
SELECT TOP 1 DepartmentId, COUNT(*) AS EmpCount
FROM Employees
```

```
GROUP BY DepartmentId
ORDER BY COUNT(*) DESC;
```

Handle ties (all departments that share the max):

```
WITH DeptCount AS (
  SELECT DepartmentId, COUNT(*) AS EmpCount,
         RANK() OVER (ORDER BY COUNT(*) DESC) AS rnk
  FROM Employees
  GROUP BY DepartmentId
)
SELECT DepartmentId, EmpCount
FROM DeptCount
WHERE rnk = 1;
```

■ Short summary line

Group by DepartmentId and use `ORDER BY COUNT(*) DESC` (or `RANK()` to return all tied top departments) to find the largest team.

---

7) Get top 3 highest salaries per department.

■ Explanation

Use `ROW_NUMBER()` (or `RANK()` / `DENSE_RANK()` depending on tie-handling) partitioned by department and ordered by salary descending, then filter where row number ≤ 3.

■ When/Why is it used

Used for per-department leaderboards, compensation reviews, or dashboards that show top performers.

■ Example / Code snippet

```
-- Top 3 rows per department; ROW_NUMBER() returns exactly 3 per dept even if ties exist
WITH Ranked AS (
  SELECT EmployeeId, Name, DepartmentId, Salary,
         ROW_NUMBER() OVER (PARTITION BY DepartmentId ORDER BY Salar
```

```
y DESC) AS rn
  FROM Employees
)
SELECT EmployeeId, Name, DepartmentId, Salary
FROM Ranked
WHERE rn <= 3
ORDER BY DepartmentId, rn;
```

If you want to include ties (all employees sharing the 3rd highest salary):

```
WITH R AS (
  SELECT EmployeeId, Name, DepartmentId, Salary,
        DENSE_RANK() OVER (PARTITION BY DepartmentId ORDER BY Salary
DESC) AS dr
  FROM Employees
)
SELECT * FROM R WHERE dr <= 3;
```

■ Short summary line

Partition by department and use `ROW_NUMBER()` (or `DENSE_RANK()` to include ties) to pick top-N per group.

---

8) Query to find employees not present in another table (NOT IN / NOT EXISTS).

■ Explanation

To find rows in A not present in B, prefer `NOT EXISTS` (null-safe and typically better performance) or a `LEFT JOIN` filter `WHERE b.key IS NULL`. `NOT IN` can be problematic if the subquery returns NULLs.

■ When/Why is it used

Used for orphan detection (e.g., employees with no assigned accounts), synchronization checks, or anti-join patterns.

■ Example / Code snippet

Using NOT EXISTS (recommended):

```
-- Employees not in EmployeesArchive (example)
SELECT e.*
FROM Employees e
WHERE NOT EXISTS (
  SELECT 1 FROM EmployeesArchive a
  WHERE a.EmployeeId = e.EmployeeId
);
```

Using LEFT JOIN anti-join:

```
SELECT e.*
FROM Employees e
LEFT JOIN EmployeesArchive a
  ON e.EmployeeId = a.EmployeeId
WHERE a.EmployeeId IS NULL;
```

Avoid if possible: NOT IN (NULL pitfalls):

```
SELECT * FROM Employees
WHERE EmployeeId NOT IN (SELECT EmployeeId FROM EmployeesArchiv
e);
-- If EmployeesArchive.EmployeeId contains NULL, this returns no rows
```

■ Short summary line

Use `NOT EXISTS` or `LEFT JOIN ... WHERE right IS NULL` to find rows absent in another table; avoid `NOT IN` when NULLs may appear.

---

9) Query to list all employees who never placed any order (customer–orders type).

■ Explanation

This is an anti-join problem: list employees not referenced in the Orders table. Use `NOT EXISTS` or `LEFT JOIN ... IS NULL` to solve safely.

■ When/Why is it used

Used for identifying inactive users/customers, re-engagement campaigns, or data reconciliation.

■ Example / Code snippet

```
-- Assuming Orders has CreatedByEmployeeId FK referencing Employees.E
mployeeId
SELECT e.EmployeeId, e.Name, e.Email
FROM Employees e
WHERE NOT EXISTS (
  SELECT 1
  FROM Orders o
  WHERE o.CreatedByEmployeeId = e.EmployeeId
);

-- Alternative using LEFT JOIN
SELECT e.EmployeeId, e.Name, e.Email
FROM Employees e
LEFT JOIN Orders o
  ON o.CreatedByEmployeeId = e.EmployeeId
WHERE o.OrderId IS NULL;
```

If you want to ensure performance: make sure Orders.CreatedByEmployeeId is indexed for the `NOT EXISTS` or join to use indexes efficiently.

■ Short summary line

Use `NOT EXISTS` (or left anti-join) to list employees who have never created any orders, ensuring the FK column is indexed for performance.

🔷 E. Window Functions (Nice Bonus Points)

1) What are window functions?

■ Explanation

Window functions perform calculations *across a set of rows related to the current row*, without collapsing rows into a single result (unlike GROUP BY). Each row keeps its identity while still having access to aggregated or ranked values.

■ When/Why is it used

Used for ranking, running totals, moving averages, cumulative sums, and performing analytics without losing row-level detail.

■ Example / Code snippet

```
SELECT Name, Salary,
      AVG(Salary) OVER () AS AvgSalary
FROM Employees;
```

Each row shows employee salary plus the overall average salary — without grouping.

■ Short summary line

Window functions provide row-wise analytics like ranking and running totals without grouping or losing individual rows.

---

2) Difference between GROUP BY and window functions.

■ Explanation

- `GROUP BY` **collapses** rows into groups and returns one row per group.
- Window functions **preserve** individual rows and add calculated values alongside them.

■ When/Why is it used

Use `GROUP BY` when you need aggregated summaries.

Use window functions when you need both *detail* and *aggregates* together.

■ Example / Code snippet

GROUP BY (loses row detail):

```
SELECT DepartmentId, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY DepartmentId;
```

Window function (keeps row detail):

```
SELECT Name, DepartmentId, Salary,
       AVG(Salary) OVER (PARTITION BY DepartmentId) AS AvgDeptSalary
FROM Employees;
```

■ **Short summary line**

GROUP BY collapses rows; window functions keep all rows while adding analytical values.

---

3) Use of ROW_NUMBER(), RANK(), DENSE_RANK().

■ **Explanation**

These functions assign rankings to rows:

- **ROW_NUMBER()** → Always unique numbers (1,2,3...) even if values tie.
- **RANK()** → Same rank for ties; **skips** next rank (1,1,3).
- **DENSE_RANK()** → Same rank for ties; **does not skip** (1,1,2).

■ **When/Why is it used**

Used for top-N results, pagination, finding duplicates, and ranked reporting.

■ **Example / Code snippet**

```
SELECT Name, Salary,
       ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum,
       RANK()       OVER (ORDER BY Salary DESC) AS RankVal,
       DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRankVal
FROM Employees;
```

■ **Short summary line**

Use ROW_NUMBER for unique ordering, RANK for competition-style ranking, and DENSE_RANK when you don’t want gaps after ties.

---

4) Query to assign serial numbers to rows using ROW_NUMBER().

■ **Explanation**

ROW_NUMBER generates sequential numbering based on order criteria.

■ **When/Why is it used**

Used for pagination, exporting grids, ordering UI lists, and debugging queries.

■ **Example / Code snippet**

```
SELECT
   ROW_NUMBER() OVER (ORDER BY EmployeeId) AS SrNo,
   EmployeeId, Name, Salary
FROM Employees;
```

■ **Short summary line**

ROW_NUMBER assigns sequential serial numbers based on defined ordering.

---

5) Query to get 2nd highest salary using RANK/DENSE_RANK.

■ **Explanation**

Rank-based window functions simplify top-N logic.

Use **RANK** to include ties but skip ranks; use **DENSE_RANK** to include ties continuously.

■ **When/Why is it used**

Used when tie-handling is important (multiple employees with same salary).

■ **Example / Code snippet (using DENSE_RANK – recommended)**

```
WITH SalaryRank AS (
  SELECT Salary,
        DENSE_RANK() OVER (ORDER BY Salary DESC) AS rnk
  FROM Employees
)
SELECT Salary
FROM SalaryRank
WHERE rnk = 2;
```

■ Short summary line

DENSE_RANK with “rnk = 2” cleanly returns the 2nd highest salary, including ties.

---

6) Running total (cumulative sum) using window function.

■ Explanation

A running total is a cumulative sum of a column ordered by date or ID.

`SUM() OVER (ORDER BY ...)` handles this perfectly.

■ When/Why is it used

Used in financial dashboards, transaction statements, monthly expense tracking, analytics, and time-series reporting.

■ Example / Code snippet

```
SELECT
    TransactionId,
    Amount,
    SUM(Amount) OVER (ORDER BY TransactionId) AS RunningTotal
FROM Transactions;
```

Date-based running total:

```
SELECT
    Date,
    Amount,
    SUM(Amount) OVER (ORDER BY Date) AS CumulativeAmount
FROM Payments;
```

■ Short summary line

Use `SUM() OVER (ORDER BY ...)` to compute cumulative totals without grouping.

🔷 F. Stored Procedures – Core Topic

1) What is a stored procedure?

■ **Explanation**

A stored procedure is a precompiled SQL program stored in the database. It contains reusable SQL statements (SELECT/INSERT/UPDATE/DELETE/logic) that run on the SQL Server side.

■ **When/Why is it used**

Used when you want consistent, secure, and performant database operations. APIs frequently call stored procedures for business logic, validation, and data processing.

■ **Example / Code snippet**

```
CREATE PROCEDURE GetAllEmployees
AS
BEGIN
    SELECT * FROM Employees;
END;
```

■ **Short summary line**

Stored procedures are precompiled SQL programs used to execute reusable and optimized database logic.

2) Advantages of stored procedures over plain queries.

■ **Explanation**

Stored procedures offer performance and maintainability benefits compared to direct SQL queries from applications.

■ **When/Why is it used**

Used when you want optimized, secure, maintainable, and modular database logic—common in enterprise applications.

■ **Example / Key Points**

✔️ Precompiled → faster execution

✔️ Encapsulate business logic → central place for updates

✔️ Prevent SQL injection → parameters ensure safety

✔️ Reduced network traffic → logic runs server-side

✔️ Reusability across different applications

✔️ Can use transactions, loops, error handling inside

■ **Short summary line**

Stored procedures improve performance, security, maintainability, and encapsulate business logic at the DB layer.

---

3) How to create a simple stored procedure (no parameters)?

■ **Explanation**

A basic stored procedure includes only SQL statements and no input/output parameters.

■ **When/Why is it used**

Used for fixed operations like fetching all rows or running cleanup tasks.

■ **Example / Code snippet**

```
CREATE PROCEDURE GetAllDepartments
AS
BEGIN
   SELECT DepartmentId, DepartmentName
   FROM Departments;
END;
```

To call it:

```
EXEC GetAllDepartments;
```

■ **Short summary line**

Simple stored procedures contain fixed SQL logic and are executed using `EXEC`.

---

4) Stored procedure with input parameter — example.

■ **Explanation**

Input parameters allow dynamic filtering or logic by passing values at runtime.

■ **When/Why is it used**

Used when APIs need to fetch or update data based on user input like ID, date range, or status.

■ **Example / Code snippet**

```
CREATE PROCEDURE GetEmployeeById
   @EmployeeId INT
AS
BEGIN
   SELECT EmployeeId, Name, Salary
   FROM Employees
   WHERE EmployeeId = @EmployeeId;
END;
```

To call:

```
EXEC GetEmployeeById @EmployeeId = 101;
```

■ **Short summary line**

Input parameters make stored procedures dynamic and reusable for different inputs.

---

5) Stored procedure with multiple input parameters.

■ **Explanation**

You can pass multiple parameters to filter or update using multiple conditions.

■ **When/Why is it used**

Used when queries require multiple conditions—like filtering by department + salary range.

■ **Example / Code snippet**

```
CREATE PROCEDURE GetEmployeesByDeptAndSalary
    @DeptId INT,
    @MinSalary DECIMAL(10,2)
AS
BEGIN
    SELECT Name, Salary
    FROM Employees
    WHERE DepartmentId = @DeptId
      AND Salary >= @MinSalary;
END;
```

To execute:

```
EXEC GetEmployeesByDeptAndSalary
    @DeptId = 2,
    @MinSalary = 50000;
```

■ **Short summary line**

Multiple parameters allow combining various filters in a single reusable stored procedure.

---

6) Stored procedure with output parameter — example.

■ **Explanation**

Output parameters return values back to the caller (like count, status, message). Useful when you want scalar results instead of full datasets.

■ **When/Why is it used**

Useful for returning generated IDs, messages, counts, or results to an API (.NET, Java, etc).

■ **Example / Code snippet**

```
CREATE PROCEDURE GetEmployeeCountByDept
   @DeptId INT,
   @EmpCount INT OUTPUT
AS
BEGIN
   SELECT @EmpCount = COUNT(*)
   FROM Employees
   WHERE DepartmentId = @DeptId;
END;
```

To call:

```
DECLARE @Count INT;
EXEC GetEmployeeCountByDept
   @DeptId = 3,
   @EmpCount = @Count OUTPUT;

SELECT @Count AS TotalEmployees;
```

■ Short summary line

Output parameters help return scalar values like counts or IDs back to the calling application.

---

7) How to call stored procedure in .NET / Java code?

■ Explanation

Calling a stored procedure is done via the database client: you prepare a command object, set the command type to `StoredProcedure`, add parameters (input/output), execute and read results. In .NET you commonly use ADO.NET, Dapper or EF Core; in Java you use JDBC (or Microsoft JDBC extensions).

■ When/Why is it used

Used whenever business logic is encapsulated in the DB (reports, complex joins, transactional work) and you want the application to invoke that logic securely and efficiently.

■ Example / Code snippet

.NET — ADO.NET (synchronous):

```
using (var conn = new SqlConnection(connString))
using (var cmd = new SqlCommand("GetEmployeeById", conn))
{
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@EmployeeId", 101);
    conn.Open();
    using (var rdr = cmd.ExecuteReader())
    {
        while (rdr.Read())
        {
            Console.WriteLine(rdr["Name"]);
        }
    }
}
```

.NET — Dapper (simpler mapping):

```
using (var conn = new SqlConnection(connString))
{
    var employee = conn.QuerySingleOrDefault<Employee>(
        "GetEmployeeById",
        new { EmployeeId = 101 },
        commandType: CommandType.StoredProcedure);
}
```

.NET — EF Core (FromSql / ExecuteSqlRaw):

```
// If SP returns entities
var employees = context.Employees
    .FromSqlRaw("EXEC GetEmployeesByDept @p0", deptId)
    .ToList();
```

Java — JDBC (CallableStatement):

```
String sql = "{call GetEmployeeById(?)}";
try (Connection conn = DriverManager.getConnection(url, user, pass);
     CallableStatement cs = conn.prepareCall(sql)) {
    cs.setInt(1, 101);
    try (ResultSet rs = cs.executeQuery()) {
        while (rs.next()) {
            System.out.println(rs.getString("Name"));
        }
    }
}
```

Java — calling SP with OUT param:

```
String sql = "{call GetEmployeeCountByDept(?, ?)}"; // IN dept, OUT count
try (CallableStatement cs = conn.prepareCall(sql)) {
    cs.setInt(1, 3);                // IN
    cs.registerOutParameter(2, Types.INTEGER); // OUT
    cs.execute();
    int count = cs.getInt(2);
}
```

■ Short summary line

Use `SqlCommand` / `CallableStatement` with `CommandType.StoredProcedure`, add parameters safely, then execute and read results — Dapper/EF Core simplify common scenarios.

---

8) How to return multiple result sets from a stored procedure?

■ Explanation

A stored procedure can execute multiple `SELECT` statements; the caller receives multiple result sets. Clients iterate results using data reader methods that advance to next result (e.g., `SqlDataReader.NextResult()` or JDBC `getMoreResults()`).

■ When/Why is it used

Used to return related datasets in one call (e.g., header + lines, master + lookup lists) to reduce round-trips and simplify transactional reads.

■ **Example / Code snippet**

SQL Server SP (multiple SELECTs):

```
CREATE PROCEDURE GetCustomerWithOrders
   @CustomerId INT
AS
BEGIN
   SELECT * FROM Customers WHERE CustomerId = @CustomerId;
   SELECT * FROM Orders WHERE CustomerId = @CustomerId;
END;
```

.NET — read multiple result sets with SqlDataReader:

```
using (var cmd = new SqlCommand("GetCustomerWithOrders", conn))
{
   cmd.CommandType = CommandType.StoredProcedure;
   cmd.Parameters.AddWithValue("@CustomerId", 1);
   using (var rdr = cmd.ExecuteReader())
   {
      // First result set: Customer
      if (rdr.Read())
      {
         var name = rdr["Name"].ToString();
      }
      // Move to next
      if (rdr.NextResult())
      {
         while (rdr.Read()) // Orders
         {
            Console.WriteLine(rdr["OrderId"]);
         }
      }
   }
}
```

Java — JDBC (multiple result sets):

```
CallableStatement cs = conn.prepareCall("{call GetCustomerWithOrders
(?)}");
cs.setInt(1, 1);
boolean hasResult = cs.execute();
if (hasResult) {
    try (ResultSet rs1 = cs.getResultSet()) {
        // process customers
    }
}
if (cs.getMoreResults()) {
    try (ResultSet rs2 = cs.getResultSet()) {
        // process orders
    }
}
```

■ Short summary line

Return multiple `SELECT`s from the SP and iterate result sets on the client with `NextResult()` / `getMoreResults()` to process them sequentially.

---

9) How to handle exceptions in stored procedure (TRY...CATCH)?

■ Explanation

SQL Server supports `TRY...CATCH`. Put risky statements in `TRY`; handle errors in `CATCH` using functions like `ERROR_NUMBER()`, `ERROR_MESSAGE()`, and optionally `THROW` to propagate or rethrow. In CATCH you can rollback transactions, log errors, set output params, or return consistent error codes.

■ When/Why is it used

Used to maintain transactional integrity, provide meaningful error information, and ensure predictable behavior for callers (applications can react to standardized error outputs).

■ Example / Code snippet

```
CREATE PROCEDURE SafeUpdateEmployeeSalary
    @EmployeeId INT,
```

```
    @NewSalary DECIMAL(10,2)
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRAN;

        UPDATE Employees
        SET Salary = @NewSalary
        WHERE EmployeeId = @EmployeeId;

        -- commit if OK
        COMMIT TRAN;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRAN;

        -- Capture error info
        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @ErrNo INT = ERROR_NUMBER();

        -- Optionally rethrow to caller with original details
        THROW @ErrNo, @ErrMsg, 1;
    END CATCH
END;
```

Notes:

- Use `THROW` (SQL Server 2012+) to rethrow preserving original error number and message.
- Use `XACT_STATE()` to determine transaction state before commit/rollback.

■ Short summary line

Use `TRY...CATCH`, check `XACT_STATE()` for safe rollback, capture `ERROR_...()` info, and rethrow with `THROW` to provide consistent error behavior.

---

10) How to use transactions inside stored procedure?

■ Explanation

Wrap related statements in `BEGIN TRAN` / `COMMIT` / `ROLLBACK` to ensure atomicity. Inside SPs, handle nested or caller transactions carefully by checking `XACT_STATE()` and `@@TRANCOUNT`, and ensure you do not accidentally commit a caller's transaction.

■ When/Why is it used

Used to ensure a set of DB operations either all succeed or none do — crucial for financial updates, inventory adjustments, and any multi-statement consistency requirement.

■ Example / Code snippet

```
CREATE PROCEDURE TransferFunds
   @FromAccount INT,
   @ToAccount INT,
   @Amount DECIMAL(10,2)
AS
BEGIN
   SET NOCOUNT ON;
   BEGIN TRY
      BEGIN TRAN;

      UPDATE Accounts SET Balance = Balance - @Amount WHERE Accoun
tId = @FromAccount;
      UPDATE Accounts SET Balance = Balance + @Amount WHERE Accou
ntId = @ToAccount;

      COMMIT TRAN;
   END TRY
   BEGIN CATCH
      IF XACT_STATE() <> 0
         ROLLBACK TRAN;

      THROW; -- rethrow original error
```

```
    END CATCH
END;
```

Best practices:

- Use `TRY...CATCH` with `XACT_STATE()` checks.
- Avoid unconditional `COMMIT` if you didn't start the outermost transaction in multi-layered callers; consider `SAVEPOINT` (`SAVE TRANSACTION`) for partial rollbacks.
- Keep transactions short to reduce locks and contention.

■ Short summary line

Use `BEGIN TRAN` + `COMMIT` / `ROLLBACK` inside `TRY...CATCH`, check `XACT_STATE()` and keep transactions short to maintain consistency and performance.

---

11) When would you choose SP vs writing logic in application code?

■ Explanation

Choice depends on concerns like performance, security, maintainability, deployment cadence, and team skills. Stored procedures centralize DB logic; application code centralizes business logic with better tooling and CI/CD.

■ When/Why is it used

Prefer Stored Procedure when:

- You need server-side performance gains (set-based ops, heavy joins).
- You want to reduce network round-trips (multiple result sets / batch ops).
- You require DB-level security or controlled access.
- You need reuse across multiple applications or legacy integrations.

Prefer Application Code when:

- Business logic is complex, versioned frequently, or tightly coupled with domain code and tests.
- You need advanced language features, libraries, or dependency injection.

- You prefer centralized CI/CD and code review in app repos rather than DB scripts.

■ **Example / Code snippet (decision hints)**

- Use SP for `bulk updates`, `archival`, `complex aggregation`, `audit logging` that must be atomic in DB.
- Use app code for `validation`, `workflow orchestration`, and `UI-driven logic`.

Interview answer sample sentence:

"I use stored procedures for performance-critical set-based operations and secure data access; I implement business orchestration and domain logic in the application for easier testing and deployment."

■ **Short summary line**

Choose SPs for performance, security, and atomic DB work; choose application code for complex business logic, testing, and agile deployment.

---

12) How to debug a stored procedure that is running slow?

■ **Explanation**

Debugging slow SPs is investigative: analyze execution plan, measure IO/time, check indexes/statistics, look for parameter sniffing, examine blocking/waits, and profile with server tools.

■ **When/Why is it used**

Used when production queries exceed SLAs or a particular stored procedure causes user-facing latency or high resource usage.

■ **Example / Debug checklist & commands**

1. **Capture execution plan** (actual plan in SSMS) and identify scans, expensive operators, missing index recommendations.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
-- then EXEC yourProc
```

```
SET STATISTICS IO OFF;
SET STATISTICS TIME OFF;
```

2. **Get actual plan and look for** high-cost operators, sorts, hash joins, large I/O.
3. **Check wait stats and blocking** using `sys.dm_os_wait_stats` and `sys.dm_exec_requests` / `sys.dm_tran_locks`.
4. **Check parameter sniffing**: try `OPTION (RECOMPILE)` or `WITH RECOMPILE` on the SP to see if plan change helps.
5. **Update statistics / consider indexing**: `UPDATE STATISTICS`, add appropriate indexes, or rewrite query to be set-based.
6. **Test with representative parameters** and use `SET STATISTICS IO/TIME` to measure.
7. **Consider temp tables** if intermediate datasets are large and reused, or rewrite using hashed partitions.
8. **Profiler / Extended Events / Query Store**: capture runtime behavior and regressions.

■ Short summary line

Use execution plans, `SET STATISTICS IO/TIME`, Query Store, wait stats, and indexing/statistics fixes; test for parameter sniffing and consider plan recompilation or query rewrite.

---

13) How do you recompile a stored procedure?

■ Explanation

Recompilation forces SQL Server to generate a fresh execution plan. Options include `sp_recompile`, `WITH RECOMPILE` when creating/executing, or using `OPTION (RECOMPILE)` in the query.

■ When/Why is it used

Used to address bad cached plans (parameter sniffing) or after schema/statistics changes that make the old plan suboptimal.

■ Example / Code snippet

```
-- Force next execution to recompile
EXEC sp_recompile 'dbo.MyProcedure';

-- Create / alter proc with immediate recompile behavior
ALTER PROCEDURE dbo.MyProcedure
WITH RECOMPILE
AS
BEGIN
   -- body
END;

-- Force recompile for a single execution (no SP change)
EXEC dbo.MyProcedure WITH RECOMPILE;

-- Or use query-level option
SELECT * FROM Employees WHERE Salary > @p OPTION (RECOMPILE);
```

Note: Frequent forced recompiles can increase CPU overhead — use judiciously.

■ Short summary line

Use `sp_recompile`, `WITH RECOMPILE`, or `OPTION (RECOMPILE)` to regenerate the execution plan and mitigate bad cached plans.

---

14) How do you pass table-valued parameter (TVP) to stored procedure?

■ Explanation

SQL Server TVP allows passing a set of rows as a parameter by defining a user-defined table type. The SP accepts the TVP as `READONLY`. Clients send a DataTable/structured type to the server — efficient for batch inserts/updates.

■ When/Why is it used

Used for bulk operations (bulk insert, multi-row updates) to reduce round-trips and avoid constructing large SQL strings; safer and performant for passing many rows from app to DB.

■ Example / Code snippet

1) Create a user-defined table type (SQL Server):

```
CREATE TYPE dbo.EmployeeType AS TABLE
(
   EmployeeId INT,
   Name NVARCHAR(200),
   Salary DECIMAL(10,2)
);
```

2) Stored procedure accepting TVP:

```
CREATE PROCEDURE InsertEmployees_TVP
   @Employees dbo.EmployeeType READONLY
AS
BEGIN
   INSERT INTO Employees (EmployeeId, Name, Salary)
   SELECT EmployeeId, Name, Salary
   FROM @Employees;
END;
```

3) .NET — pass TVP using `DataTable` and `SqlParameter` (ADO.NET):

```
var table = new DataTable();
table.Columns.Add("EmployeeId", typeof(int));
table.Columns.Add("Name", typeof(string));
table.Columns.Add("Salary", typeof(decimal));
table.Rows.Add(1001, "Alice", 60000m);
table.Rows.Add(1002, "Bob", 55000m);

using (var cmd = new SqlCommand("InsertEmployees_TVP", conn))
{
   cmd.CommandType = CommandType.StoredProcedure;
   var param = cmd.Parameters.AddWithValue("@Employees", table);
   param.SqlDbType = SqlDbType.Structured;
   param.TypeName = "dbo.EmployeeType";
   cmd.ExecuteNonQuery();
}
```

4) Java — using Microsoft JDBC driver (SQLServerDataTable)

```
// uses com.microsoft.sqlserver.jdbc.SQLServerDataTable
SQLServerDataTable tvp = new SQLServerDataTable();
tvp.addColumnMetadata("EmployeeId", java.sql.Types.INTEGER);
tvp.addColumnMetadata("Name", java.sql.Types.NVARCHAR);
tvp.addColumnMetadata("Salary", java.sql.Types.DECIMAL);

tvp.addRow(1001, "Alice", new BigDecimal("60000.00"));

CallableStatement cstmt = conn.prepareCall("{call InsertEmployees_TVP
(?)}");
cstmt.setObject(1, tvp);
cstmt.execute();
```

(Requires the Microsoft JDBC driver and SQLServerDataTable support.)

■ Short summary line

Define a user table type, accept it `READONLY` in the SP, and pass a `DataTable`/structured object from .NET (or `SQLServerDataTable` from Microsoft JDBC) for efficient multi-row operations.

🔷 G. Functions (SQL Server)

1) Difference between Stored Procedure and Function

■ Explanation

A Stored Procedure is a precompiled block of SQL statements that can perform operations like SELECT, INSERT, UPDATE, DELETE.

A Function returns a single value or a table and must be used inside SQL queries.

■ When/Why is it used

Use SP for business logic, data modification, or multi-step operations.

Use Function when you need reusable calculations inside SELECT/WHERE or use it like an expression.

■ Example / Code snippet

Feature	Stored Procedure	Function
Return type	0 or many result sets	Must return 1 value or table
Allows INSERT/UPDATE/DELETE	✔ Yes	❌ No
Can be used in SELECT	❌ No	✔ Yes
Can handle transactions	✔ Yes	❌ No
Have output params	✔ Yes	❌ No

■ Short summary line

Stored procedures perform full operations, functions return values and are used inside SQL expressions.

---

2) Scalar function vs Table-valued function

■ Explanation

- **Scalar Function:** Returns a single value (INT, VARCHAR, DATE).
- **Table-Valued Function (TVF):** Returns a table (like a virtual table you can query).

■ When/Why is it used

Use scalar functions for small reusable calculations.

Use TVFs for reusable queries that return multiple rows.

■ Example / Code snippet

```
-- Scalar
CREATE FUNCTION GetYearOnly(@date DATE)
RETURNS INT
AS
BEGIN
   RETURN YEAR(@date);
END;
```

```
-- Table-valued
CREATE FUNCTION GetEmployeesByDept(@DeptId INT)
RETURNS TABLE
AS
RETURN (
   SELECT EmployeeId, Name FROM Employees WHERE DepartmentId = @
DeptId
);
```

■ **Short summary line**

Scalar functions return a value, TVFs return a reusable table result.

---

3) Example of scalar function (e.g., FormatName(@first, @last))

■ **Explanation**

A scalar function takes input parameters and returns one formatted value.

■ **When/Why is it used**

Used for formatting, data cleaning, reusable expressions inside SELECT queries.

■ **Example / Code snippet**

```
CREATE FUNCTION FormatName
(
   @FirstName NVARCHAR(50),
   @LastName NVARCHAR(50)
)
RETURNS NVARCHAR(100)
AS
BEGIN
   RETURN @FirstName + ' ' + @LastName;
END;
```

Usage:

```
SELECT dbo.FormatName(FirstName, LastName) AS FullName
FROM Employees;
```

■ **Short summary line**

Scalar functions return a single calculated value such as formatted strings.

---

4) Example of inline table-valued function

■ **Explanation**

An inline TVF returns a set of rows using a single SELECT statement (most efficient TVF type).

■ **When/Why is it used**

Used for reusable filtering or lookup queries that return rows, similar to a parameterized view.

■ **Example / Code snippet**

```
CREATE FUNCTION GetHighSalaryEmployees(@MinSalary DECIMAL(10,2))
RETURNS TABLE
AS
RETURN
(
    SELECT EmployeeId, Name, Salary
    FROM Employees
    WHERE Salary >= @MinSalary
);
```

Usage:

```
SELECT * FROM dbo.GetHighSalaryEmployees(60000);
```

■ **Short summary line**

Inline TVFs return a table and behave like lightweight parameterized views.

---

5) Can a function modify table data? Why/why not?

■ Explanation

No. SQL functions **cannot** perform INSERT, UPDATE, DELETE, or modify table state.

■ When/Why is it used

Because functions are expected to be deterministic and safe inside SELECT statements, SQL restricts them to being read-only.

■ Example / Code snippet

Disallowed inside functions:

```
UPDATE Employees SET Salary = Salary + 1000;  -- ❌ Not allowed
```

Allowed only in Stored Procedures.

■ Short summary line

Functions cannot modify data because SQL Server enforces them as deterministic, read-only expressions.

---

6) When should you prefer function over stored procedure?

■ Explanation

Choose a function when you need reusable logic inside queries and predictable return values.

■ When/Why is it used

Use a function when:

- You need logic inside SELECT/WHERE/HAVING
- You need to return a single value repeatedly
- You want a reusable filter or lookup table (TVF)
- You need deterministic computation (e.g., tax calculation, formatting)

Do **NOT** use it for data modifications or transactions.

■ **Example / Code snippet**

```
SELECT Name,
      dbo.GetTaxAmount(Salary) AS Tax
FROM Employees;
```

■ **Short summary line**

Prefer a function when logic must be used inside SQL queries as an expression or table source.

---

7) Built-in functions you commonly use (LEN, SUBSTRING, CHARINDEX, ISNULL, COALESCE, CAST, CONVERT, DATEADD, DATEDIFF)

■ **Explanation**

SQL Server provides powerful built-in scalar functions for string manipulation, null handling, type conversion, and date arithmetic.

■ **When/Why is it used**

Used in reporting, filtering, formatting, analytics, and preparing data for business logic.

■ **Example / Code snippets**

String functions

```
SELECT LEN(Name) AS Length;
SELECT SUBSTRING(Name, 1, 3);
SELECT CHARINDEX('an', Name); -- find substring position
```

NULL handling

```
SELECT ISNULL(MiddleName, 'N/A');
SELECT COALESCE(MiddleName, NickName, 'Unknown');
```

Type conversion

```
SELECT CAST(Salary AS VARCHAR(20));
SELECT CONVERT(VARCHAR(10), HireDate, 120); -- yyyy-mm-dd
```

Date functions

```
SELECT DATEADD(year, 1, HireDate) AS NextYear;
SELECT DATEDIFF(month, HireDate, GETDATE()) AS MonthsWorked;
```

■ Short summary line

Built-in functions help in string processing, null handling, type conversion, and date calculations essential for data preparation.

🔷 H. Transactions & Error Handling

1) What is a transaction?

■ Explanation

A transaction is a logical unit of work that groups multiple SQL operations together so that they either **all succeed** or **all fail**.

It ensures consistent and reliable data handling.

■ When/Why is it used

Used when multiple related updates must be treated as one atomic operation — examples: money transfer, order placement, inventory updates.

■ Example / Code snippet

```
BEGIN TRAN;
UPDATE Accounts SET Balance = Balance - 1000 WHERE Id = 1;
UPDATE Accounts SET Balance = Balance + 1000 WHERE Id = 2;
COMMIT;
```

■ Short summary line

A transaction guarantees that a group of operations behaves as a single all-or-nothing unit.

---

2) ACID properties — explain each

■ **Explanation**

ACID is the set of guarantees that ensure reliable transaction processing.

■ **When/Why is it used**

Used to maintain data integrity in banking, financial apps, and critical systems.

■ **Example / Code snippet**

Property	Meaning
Atomicity	Entire transaction succeeds or fails; no partial updates.
Consistency	Transaction brings DB from one valid state to another.
Isolation	Concurrent transactions do not interfere with each other.
Durability	Once committed, data persists even after crashes.

■ **Short summary line**

ACID ensures transactions are atomic, consistent, isolated, and durable.

---

3) Example of transaction using BEGIN TRAN, COMMIT, ROLLBACK

■ **Explanation**

This structure defines a transactional block where COMMIT saves changes and ROLLBACK undoes them.

■ **When/Why is it used**

Used for critical multi-step updates where partial failure is unacceptable.

■ **Example / Code snippet**

```
BEGIN TRAN;

UPDATE Employees SET Salary = Salary + 2000 WHERE EmployeeId = 10;

IF @@ERROR <> 0
BEGIN
   ROLLBACK TRAN;
   RETURN;
```

```
END

UPDATE Employees SET Bonus = Bonus + 500 WHERE EmployeeId = 10;

COMMIT TRAN;
```

■ **Short summary line**

Transactions use `BEGIN TRAN`, `COMMIT`, and `ROLLBACK` to control multi-step operations safely.

---

4) What happens if error occurs in middle of transaction?

■ **Explanation**

If an error occurs, the transaction enters a failed state and must be either rolled back manually or via TRY...CATCH logic.

■ **When/Why is it used**

Used to protect data integrity when some statements succeed but others fail.

■ **Example / Code snippet**

```
BEGIN TRAN;

UPDATE A ...;
-- Error occurs here
UPDATE B ...;

-- Transaction must be rolled back
ROLLBACK TRAN;
```

■ **Short summary line**

If a transaction fails mid-way, SQL Server keeps it open until you explicitly roll it back or handle it inside TRY...CATCH.

---

5) How to use TRY…CATCH with transaction in SQL Server?

■ Explanation

TRY…CATCH ensures transactions are committed only on success and rolled back safely on failure.

■ When/Why is it used

Used in stored procedures or batch operations to ensure clean rollback and proper error reporting.

■ Example / Code snippet

```
BEGIN TRY
   BEGIN TRAN;

   UPDATE A SET ...;
   UPDATE B SET ...;

   COMMIT TRAN;
END TRY
BEGIN CATCH
   IF XACT_STATE() <> 0 ROLLBACK TRAN;

   DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
   THROW;  -- rethrow error
END CATCH;
```

■ Short summary line

Use TRY…CATCH with `XACT_STATE()` to guarantee controlled COMMIT or ROLLBACK.

---

6) What is deadlock? How do you avoid it?

■ Explanation

A deadlock occurs when two or more transactions wait on each other's locked resources, creating a circular dependency; SQL Server kills one transaction (victim).

■ **When/Why is it used**

Understanding deadlocks is crucial in concurrent systems with frequent updates.

■ **Example / Code snippet (concept)**

```
Txn1 locks Row A → wants Row B
Txn2 locks Row B → wants Row A
→ Both wait forever → Deadlock
```

■ **How to avoid it**

✔️ Access tables in same order across code paths

✔️ Keep transactions short

✔️ Use proper indexes (avoid long scans)

✔️ Avoid unnecessary locks (no long SELECT * with serialization)

✔️ Use READ COMMITTED SNAPSHOT to reduce locking

✔️ Break big updates into batches

■ **Short summary line**

Deadlocks happen due to circular locking; avoid by consistent access order, good indexing, and short transactions.

---

7) What is lock escalation?

■ **Explanation**

Lock escalation is SQL Server's automatic process of converting many fine-grained locks (row/page) into a larger lock (table) to reduce memory overhead.

■ **When/Why is it used**

It improves system performance by reducing lock tracking overhead, but can block other sessions.

■ **Example / Code snippet (concept)**

```
Many row locks → SQL Server escalates → Table lock
```

■ **How to prevent problems**

✔ Keep transactions short

✔ Filter rows effectively (use indexes)

✔ Avoid scanning entire tables inside a transaction

✔ Use ROWLOCK hints only when necessary

■ **Short summary line**

Lock escalation converts many small locks into a bigger lock to save resources, but may increase blocking.

🔷 I. Index & Performance Around Queries

1) What is an index?

■ **Explanation**

An index is a database object that improves the speed of data retrieval by creating a sorted reference structure (like a book index).

Instead of scanning the whole table, SQL uses the index to locate rows efficiently.

■ **When/Why is it used**

Used when tables grow large and SELECT queries become slow.

Helps speed up WHERE, JOIN, ORDER BY, and GROUP BY operations.

■ **Example / Code snippet**

```
CREATE INDEX IX_Employees_Salary
ON Employees(Salary);
```

■ **Short summary line**

Indexes speed up data retrieval by allowing SQL Server to find rows without scanning the whole table.

---

2) Clustered vs Non-clustered index

■ Explanation

- **Clustered Index** → Physically sorts and stores table data based on the key. (1 per table)
- **Non-clustered Index** → Logical index that stores pointers (like a lookup map) to actual data rows.

■ When/Why is it used

- Use clustered index for primary keys or columns used frequently for sorting/searching.
- Use non-clustered indexes for filtering columns, joins, or frequently queried attributes.

■ Example / Code snippet

Clustered Index

```
-- Usually created automatically on Primary Key
CREATE CLUSTERED INDEX IX_Employees_Id
ON Employees(EmployeeId);
```

Non-clustered Index

```
CREATE NONCLUSTERED INDEX IX_Employees_Department
ON Employees(DepartmentId);
```

■ Short summary line

Clustered index sorts the table physically; non-clustered index stores separate lookup structure pointing to data.

---

3) How index helps a SELECT query?

■ Explanation

Indexes allow SQL Server to directly navigate to required rows instead of scanning the entire table (seek operation).

They convert expensive table scans into efficient index seeks.

■ **When/Why is it used**

Used when queries filter on specific columns frequently, such as searching by email, department, date, or ID.

■ **Example / Code snippet**

Without index → full table scan:

```
SELECT * FROM Employees WHERE Email = 'abc@test.com';
```

With index:

```
CREATE INDEX IX_Employees_Email ON Employees(Email);
```

Query now uses **Index Seek**, significantly faster.

■ **Short summary line**

Indexes enable fast lookups by turning slow scans into quick seeks.

---

4) When index can hurt INSERT/UPDATE/DELETE performance?

■ **Explanation**

Indexes speed up SELECTs but slow down writes because SQL Server must update the index entries for each row change.

■ **When/Why is it used**

Happens when:

- Table has too many indexes
- High insert/update workloads
- Frequent bulk operations

■ **Example / Code snippet**

```
INSERT INTO Employees(Name, Salary) VALUES ('A', 50000);
-- SQL must update:
```

```
-- 1) Clustered index
-- 2) All non-clustered indexes
```

■ **Short summary line**

Indexes slow down write operations because SQL must update every related index after data changes.

---

5) How do you find which query is taking more time?

■ **Explanation**

SQL Server provides multiple tools to identify slow queries: Query Store, DMVs, Profiler, Extended Events, Execution Plans.

■ **When/Why is it used**

Used during performance tuning, debugging production issues, or analyzing high CPU/IO queries.

■ **Example / Code snippet**

Using Query Store (SQL Server 2016+)

```
SELECT TOP 10 *
FROM sys.query_store_runtime_stats
ORDER BY avg_duration DESC;
```

Using DMVs

```
SELECT TOP 10
    total_elapsed_time/ execution_count AS AvgTime,
    text
FROM sys.dm_exec_query_stats
CROSS APPLY sys.dm_exec_sql_text(sql_handle)
ORDER BY AvgTime DESC;
```

■ **Short summary line**

Use Query Store, DMVs, or Profiler to identify the slowest queries and analyze their execution patterns.

---

6) What is execution plan? How do you read it?

■ Explanation

An execution plan shows how SQL Server executes a query — which indexes it uses, join methods, scans vs seeks, and overall cost.

■ When/Why is it used

Used to troubleshoot slow queries and understand performance bottlenecks like missing indexes, expensive sorts, or table scans.

■ Example / Code snippet

In SSMS:

```
Click: Display Actual Execution Plan (Ctrl + M)
Run your query
```

What to look for:

- **Index Seek (Good)** → Efficient
- **Index Scan / Table Scan (Bad)** → Missing or ineffective index
- **Hash Match / Sort** → Expensive operators
- **Key Lookup** → Non-clustered index missing included columns
- **Missing Index Suggestion** → Consider adding it

■ Short summary line

Execution plans reveal how SQL Server runs queries, helping locate scans, missing indexes, and expensive operations.

---

7) Techniques to optimize slow query

■ Explanation

Slow queries can be improved using indexing, query rewrite, statistics update, or avoiding unnecessary scans and joins.

■ When/Why is it used

Used when queries exceed SLAs, degrade application performance, or cause high CPU/IO usage.

■ **Example / Code snippet**

✔ Techniques

1. **Create appropriate non-clustered indexes**

```
CREATE INDEX IX_Orders_CustomerId ON Orders(CustomerId);
```

1. **Add INCLUDE columns to avoid key lookups**

```
CREATE INDEX IX_Orders ON Orders(CustomerId) INCLUDE (OrderDate, A
mount);
```

1. **Rewrite SELECT * to specific columns**
2. **Update statistics**

```
UPDATE STATISTICS Employees;
```

1. **Avoid functions on indexed columns**

   (e.g., avoid `WHERE YEAR(DateColumn) = 2024`)
2. **Use EXISTS instead of IN for large sets**
3. **Use proper JOINs and avoid cross joins**
4. **Use pagination (OFFSET-FETCH)**

```
ORDER BY Date
OFFSET 0 ROWS FETCH NEXT 50 ROWS ONLY;
```

1. **Avoid nested subqueries when JOIN works better**
2. **Use execution plan to identify bottlenecks**

■ **Short summary line**

Optimize slow queries by adding correct indexes, updating statistics, avoiding full scans, rewriting inefficient logic, and analyzing execution plans.

🔷 J. Real-Time / Backend Integration Style Questions

1) How do your APIs interact with SQL Server? (end-to-end flow)

■ Explanation

APIs expose HTTP endpoints (controllers) that accept requests, validate input, call service layer logic, which in turn calls repository/DAO code that executes parameterized SQL (via EF Core / Dapper / ADO.NET). The DB returns result sets which are mapped to DTOs, business logic applies transformations, and the controller returns JSON to the frontend. Cross-cutting concerns (transactions, logging, retries, caching, metrics, error handling) run around these layers.

■ When/Why is it used

This layered flow enforces separation of concerns, testability, security (parameterization), and performance (pooled DB connections, efficient queries). It's used in production to ensure maintainable code and predictable DB interactions.

■ Example / Code snippet

```
// Controller
[HttpGet("{id}")]
public async Task<IActionResult> GetEmployee(int id) ⇒
    Ok(await _employeeService.GetByIdAsync(id));

// Service
public async Task<EmployeeDto> GetByIdAsync(int id) ⇒
    await _repo.GetByIdAsync(id);

// Repository (Dapper)
public async Task<EmployeeDto> GetByIdAsync(int id)
{
    using var conn = new SqlConnection(_connString);
    const string sql = "SELECT EmployeeId, Name, Salary FROM Employees
WHERE EmployeeId = @id";
```

```
    return await conn.QuerySingleOrDefaultAsync<EmployeeDto>(sql, new {
id });
}
```

Operational notes: use connection pooling (SqlConnection), parameterized queries, transactions for multi-step updates, retry logic for transient errors, Query Store / monitoring for slow queries, and map DB models to DTOs before returning to Angular.

■ Short summary line

APIs call service → repository → parameterized DB calls (EF/Dapper/ADO.NET) → map results to DTOs → return JSON, with transactions/logging/monitoring as cross-cutting concerns.

---

2) Have you written any stored procedures for your Expense Tracker / Fraud Service / Automation apps? Explain one.

■ Explanation

Yes — for the Expense Tracker I wrote a stored procedure that returns a monthly summary and category breakdown in one call. The procedure aggregates expenses (total, count), returns per-category totals, and supports date-range parameters to reduce round-trips and keep heavy aggregation close to the data.

■ When/Why is it used

Used to provide the frontend dashboard (single API call) with summary KPIs and category charts, improve performance by pushing aggregation to SQL Server, and ensure consistent business logic for summaries.

■ Example / Code snippet

```
CREATE PROCEDURE GetMonthlyExpenseSummary
    @UserId INT,
    @StartDate DATE,
    @EndDate DATE
```

```
AS
BEGIN
    SET NOCOUNT ON;

    -- 1) Total summary
    SELECT
        SUM(Amount) AS TotalAmount,
        COUNT(*) AS TransactionCount
    FROM Expenses
    WHERE UserId = @UserId AND ExpenseDate BETWEEN @StartDate AND
@EndDate;

    -- 2) Category breakdown
    SELECT
        Category,
        SUM(Amount) AS CategoryTotal,
        COUNT(*) AS CategoryCount
    FROM Expenses
    WHERE UserId = @UserId AND ExpenseDate BETWEEN @StartDate AND
@EndDate
    GROUP BY Category
    ORDER BY CategoryTotal DESC;
END;
```

Call from .NET (Dapper reading multiple result sets):

```
using var grid = await conn.QueryMultipleAsync("GetMonthlyExpenseSum
mary",
    new { UserId = uid, StartDate = s, EndDate = e }, commandType: Comma
ndType.StoredProcedure);

var summary = await grid.ReadSingleAsync<ExpenseSummaryDto>();
var categories = (await grid.ReadAsync<CategoryDto>()).ToList();
```

■ Short summary line

I built SPs that aggregate and return multiple result sets (summary + breakdown) so the API can deliver dashboard data in one efficient DB call.

---

3) How do you pass parameters safely to prevent SQL injection?

■ Explanation

Always use parameterized queries or stored procedures with parameters. Do **not** concatenate user input into SQL strings. Use ORM parameter support (EF Core parameters / FromSqlRaw with parameters), Dapper (anonymous object), or ADO.NET `SqlParameter`. For dynamic SQL, use sp_executesql with parameters only.

■ When/Why is it used

Prevents attackers from injecting malicious SQL, ensures proper type handling, and enables plan reuse (better performance).

■ Example / Code snippet

```
// Dapper (safe)
var sql = "SELECT * FROM Users WHERE Username = @user";
var user = await conn.QuerySingleOrDefaultAsync<User>(sql, new { user = username });

// ADO.NET (safe)
using var cmd = new SqlCommand("SELECT * FROM Users WHERE Username = @user", conn);
cmd.Parameters.Add(new SqlParameter("@user", SqlDbType.NVarChar, 100) { Value = username });
using var rdr = await cmd.ExecuteReaderAsync();
```

If dynamic SQL is unavoidable:

```
DECLARE @sql NVARCHAR(MAX) = N'SELECT * FROM Orders WHERE OrderDate >= @p1';
EXEC sp_executesql @sql, N'@p1 DATE', @p1 = @StartDate;
```

■ Short summary line

Always use parameterized queries / stored-proc parameters (never string concatenation) to prevent SQL injection and improve plan reuse.

---

4) Why parameterized queries are important?

■ **Explanation**

Parameterized queries separate SQL code from data. They prevent SQL injection, enforce type safety, and allow SQL Server to cache and reuse execution plans, which reduces CPU and improves performance.

■ **When/Why is it used**

Used for all user-driven queries (search, filters, inserts) and any scenario where inputs come from clients (Angular forms, APIs).

■ **Example / Code snippet**

```
// Example showing plan reuse benefit
await conn.ExecuteAsync("UPDATE Products SET Price = @price WHERE P
roductId = @id",
    new { price = 99.99m, id = 123 });
```

The database stores a compiled plan keyed by the parameterized template rather than the literal SQL string.

■ **Short summary line**

Parameterized queries prevent injection, enable plan caching, and enforce type safety — they are mandatory for secure and performant DB access.

---

5) In your project, what was the most complex SQL query/SP you wrote?

■ **Explanation**

The most complex SP I wrote aggregated multi-source data for a fraud-risk score: it joined transactional events, account history, and external lookup tables, applied weighted scoring rules, used window functions for session-based features, and returned both summary and flagged transaction lists in one call. It also used temp tables for intermediate deduplication and a TVP for input events.

■ **When/Why is it used**

Used in the Fraud Services Middleware to compute risk in near-real-time, minimizing data movement and latency, and returning both aggregated scores and actionable transaction IDs for downstream processing.

■ **Example / Code snippet (simplified excerpt)**

```
CREATE PROCEDURE ComputeFraudScore
    @Events dbo.EventTVP READONLY,
    @UserId INT
AS
BEGIN
    SET NOCOUNT ON;
    -- de-dupe and prepare
    SELECT DISTINCT EventId, UserId, Amount, EventTime
    INTO #Events
    FROM @Events;

    -- session aggregation with window function
    SELECT UserId,
        SUM(Amount) OVER (PARTITION BY UserId, SessionId ORDER BY Ev
entTime ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS SessionTotal,
        DENSE_RANK() OVER (PARTITION BY UserId ORDER BY EventTime)
AS SessionRank
    INTO #SessionAgg
    FROM #Events;

    -- compute weighted score joining lookup tables
    SELECT e.EventId, e.UserId,
        COALESCE(lk.Weight, 1) * (e.Amount / NULLIF(sa.SessionTotal,0)) A
S Score
    FROM #Events e
    JOIN #SessionAgg sa ON e.UserId = sa.UserId
    LEFT JOIN RiskLookup lk ON e.EventType = lk.EventType;

    -- final aggregation and return
    SELECT UserId, SUM(Score) AS RiskScore FROM #Events GROUP BY Us
erId;
```

```
    SELECT EventId FROM #Events WHERE /*flagging logic*/ Score > 0.8;
END;
```

■ Short summary line

I built SPs that combine TVPs, temp tables, window functions and lookups to compute multi-stage results (risk scores + flagged items) in a single efficient call.

---

6) How did you debug a production SQL issue?

■ Explanation

Debugging production SQL issues is a methodical process: reproduce, capture metrics and plans, identify root cause (index/stats/plan/parameter sniffing/locking), test fixes, and deploy with rollback safety. Use monitoring tools (Query Store, Extended Events, DMVs) and safe, reversible changes.

■ When/Why is it used

Used when users report slowness, timeouts, or errors — to fix performance regressions or correctness issues without causing downtime.

■ Example / Step-by-step

1. **Reproduce** with representative params in lower environment or use Query Store to find slow queries.
2. **Capture actual execution plan** and `SET STATISTICS IO/TIME ON` to measure I/O and CPU.
3. **Check wait stats/blocking** (`sys.dm_os_waiting_tasks`, `sys.dm_tran_locks`).
4. **Check parameter sniffing** (test with `OPTION (RECOMPILE)` or `WITH RECOMPILE`).
5. **Try quick safe fixes**: create/adjust index, update statistics, or add `OPTION (RECOMPILE)` for problem SP.
6. **Validate** performance improvement with realistic loads.
7. **Deploy change** during low traffic or via controlled deployment; monitor Query Store for regressions.
8. **If deeper issue**: roll back, add telemetry, and plan a robust refactor.

Tools I used: SSMS Actual Execution Plan, Query Store, Extended Events, Application logs, and APM (NewRelic/AppInsights).

■ Short summary line

Reproduce, collect execution plan and metrics, identify root cause (indexes/stats/plan), apply safe fix (index/stats/recompile), validate, and deploy with monitoring.

---

7) How do you handle pagination from backend side? (OFFSET...FETCH)

■ Explanation

Server-side pagination uses `ORDER BY` + `OFFSET ... FETCH NEXT ...` for page-based pagination. For large/real-time datasets, keyset (seek) pagination using a stable ordered key is preferred for performance and consistency.

■ When/Why is it used

Used to return only needed rows to the client (reduce payload), support UI infinite-scroll or page navigation, and improve query performance.

■ Example / Code snippet

OFFSET-FETCH (page-based):

```
-- Page 3 with pageSize 20
SELECT Id, Name, CreatedAt
FROM Users
ORDER BY CreatedAt DESC
OFFSET 40 ROWS FETCH NEXT 20 ROWS ONLY;
```

.NET (EF Core)

```
var page = 3; var size = 20;
var items = await context.Users
    .OrderByDescending(u ⇒ u.CreatedAt)
    .Skip((page-1) * size)
    .Take(size)
    .ToListAsync();
```

Keyset pagination (recommended for large sets):

```
-- LastSeenCreatedAt is the cursor from previous page
SELECT TOP (20) Id, Name, CreatedAt
FROM Users
WHERE CreatedAt < @LastSeenCreatedAt
ORDER BY CreatedAt DESC;
```

Trade-offs:

- OFFSET-FETCH is simple but becomes inefficient for high offsets (deep pages).
- Keyset pagination is faster and consistent for large datasets but requires a cursor and restricts jumping to arbitrary page numbers.

■ Short summary line

Use `OFFSET...FETCH` for simple pagination and prefer keyset (cursor) pagination for large datasets to maintain performance and consistency.
