SQL Interview Questions for Experienced Candidates (3+ years)

Data Types & Miscellaneous Concepts

Indexing Strategies & Keys

1. What is a SQL index and what are different types of indexes?

Answer:

An **index** in SQL is a performance-tuning structure that allows the database engine to locate and access data quickly, similar to an index in a book. Indexes reduce the amount of data the database must scan, thus speeding up SELECT queries, but they require additional storage and can slow down data modification operations.

Types of Indexes:

Index Type	Description	Example Syntax
Clustered	Determines the physical order of data in the table. Only one per table.	`CREATE CLUSTERED INDEX idx_emp_id ON Employees(EmployeeID);`
Non-Clustered	Separate structure from table data, contains pointers to actual rows. Multiple allowed.	`CREATE NONCLUSTERED INDEX idx_emp_lastname ON Employees(LastName);`
Unique	Ensures all values in the indexed column(s) are unique.	`CREATE UNIQUE INDEX idx_emp_email ON Employees(Email);`
Composite	Index on two or more columns.	`CREATE INDEX idx_emp_dept_salary ON Employees(DepartmentID, Salary);`
Full-Text	Optimized for text searching.	(DB-specific syntax)
Spatial	Used for spatial data types (geometry, geography).	(DB-specific syntax)

Visual Example:

```
-- Clustered Index
CREATE CLUSTERED INDEX idx_employee_id ON Employees(EmployeeID);

-- Non-Clustered Index
```

```sql
CREATE NONCLUSTERED INDEX idx_employee_lastname ON Employees(LastName);

-- Unique Index
CREATE UNIQUE INDEX idx_employee_email ON Employees(Email);

-- Composite Index
CREATE INDEX idx_employee_dept_salary ON Employees(DepartmentID, Salary);
```

> **Tip:**
>
> - Indexes speed up SELECT queries but can slow down INSERT, UPDATE, and DELETE operations.
> - Use indexes judiciously based on query patterns and data modification frequency.

---

2. What is the difference between a heap (no clustered index) and a table with a clustered index? How can you identify a heap table?

Answer:

- **Heap Table:**
  A table without a clustered index. Data is stored in no particular order, and row locations are tracked by row identifiers (RIDs).
  - *Identification:* In SQL Server, query `sys.indexes` and look for `index_id = 0`.
- **Clustered Index Table:**
  Data rows are physically ordered by the clustered index key.
  - *Identification:* `index_id = 1`.

Example:

```sql
-- Heap table (no clustered index)
CREATE TABLE HeapTable (
        ID INT,
        Name VARCHAR(50)
);

-- Add a clustered index
CREATE CLUSTERED INDEX idx_id ON HeapTable(ID);

-- Identify heap in SQL Server
SELECT name, index_id
FROM sys.indexes
WHERE object_id = OBJECT_ID('HeapTable');
```

> **Summary:**
>
> - Heaps are faster for bulk inserts but slower for searches.
> - Clustered indexes improve search performance and eliminate forwarding pointers.

3. What is the difference between a PRIMARY KEY and a UNIQUE KEY in SQL?

Answer:

Feature	PRIMARY KEY	UNIQUE KEY (or Unique Index)
Uniqueness	Enforces uniqueness	Enforces uniqueness
NULLs Allowed	Not allowed	Allowed (one NULL in most DBs)
Per Table	Only one	Multiple allowed
Index Type	Unique clustered (if none exists)	Unique non-clustered (by default)

Example:

```
CREATE TABLE Employees (
        EmployeeID INT PRIMARY KEY,
        Email VARCHAR(100) UNIQUE
);

ALTER TABLE Employees ADD CONSTRAINT uq_emp_phone UNIQUE (PhoneNumber);
```

> **Summary:**
> Use PRIMARY KEY for the main identifier; use UNIQUE KEY for alternate unique constraints.

---

4. What are index “forwarding pointers” in a heap table, and how do they affect query performance?

Answer:

- **Forwarding pointers** occur in heap tables when a row is updated and moved to a new location (due to size increase). The original location stores a pointer to the new location.
- **Impact:** Causes extra I/O as the engine must follow the pointer, degrading performance.
- **Resolution:** Rebuild the table or add a clustered index.

Example:

```
UPDATE HeapTable SET Name = REPLICATE('A', 1000) WHERE ID = 1;
```

> **Summary:**
> Forwarding pointers degrade performance in heaps with frequent updates.

---

5. What is a composite index, and how do you choose the order of columns in it for optimal performance?

Answer:

A **composite index** is an index on two or more columns. The order of columns is crucial:

- Place the most selective (most unique) column first.
- The index is most effective when queries filter on the leading column(s).

Example:

```
CREATE INDEX idx_dept_salary ON Employees(DepartmentID, Salary);
```

> **Summary:**
> Choose column order based on query patterns and selectivity.

---

6. When should you use a covering index, and how does it improve the performance of a query?

Answer:

A **covering index** includes all columns needed by a query, so the database can satisfy the query using only the index.

- **Use Case:** Frequently run queries that select a small set of columns.
- **Performance:** Reduces I/O by avoiding lookups in the base table.

Example:

```
CREATE INDEX idx_covering ON Employees(DepartmentID) INCLUDE (Salary, FirstName);
```

> **Summary:**
> Covering indexes are powerful for read-heavy workloads with predictable queries.

---

7. How does the existence of an index on a column affect INSERT, UPDATE, and DELETE performance on a table?

Answer:

Operation	Effect of Indexes
INSERT	Indexes must be updated for each new row, increasing insert time.
UPDATE	If indexed columns are updated, the index must be modified, adding overhead.
DELETE	Index entries must be removed, which can slow down deletes.

Example:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName) VALUES (1, 'Ashish',
'Zope');
```

> **Summary:**
> More indexes = faster reads, slower writes. Balance based on workload.

---

8. What is index selectivity, and why is it important for query optimization?

Answer:

- **Index selectivity** is the ratio of the number of distinct values in an indexed column to the total number of rows.
- **High selectivity:** Many unique values; index is very effective.
- **Low selectivity:** Many duplicates; index is less effective.

Example:

```
-- High selectivity
CREATE INDEX idx_employee_id ON Employees(EmployeeID);

-- Low selectivity
CREATE INDEX idx_gender ON Employees(Gender);
```

> **Summary:**
> Use indexes on columns with high selectivity for best performance.

---

9. How many clustered indexes can a table have, and why?

Answer:

- **Only one clustered index** per table, because the data rows can be physically ordered in only one way.
- Multiple non-clustered indexes are allowed.

Example:

```
CREATE CLUSTERED INDEX idx_emp_id ON Employees(EmployeeID);
CREATE NONCLUSTERED INDEX idx_emp_email ON Employees(Email);
CREATE NONCLUSTERED INDEX idx_emp_phone ON Employees(PhoneNumber);
```

> **Summary:**
> One clustered index per table; unlimited non-clustered indexes (within system limits).

---

10. What is index fragmentation, and how can it be resolved or mitigated in a large database?

Answer:

- **Index fragmentation** occurs when the logical order of index pages does not match the physical order, leading to inefficient I/O and slower queries.
- **Causes:** Frequent INSERT, UPDATE, DELETE operations.
- **Resolution:** Rebuild or reorganize indexes.

Example:

```
ALTER INDEX idx_emp_id ON Employees REBUILD;
ALTER INDEX idx_emp_id ON Employees REORGANIZE;
```

> **Summary:**
> Regular index maintenance is essential for optimal performance in large databases.

Mastering SQL Joins

SQL Joins

1. What are the different types of SQL joins and when would you use each?

Answer:

Join Type	Description	Use Case
INNER JOIN	Returns only rows with matching values in both tables.	Records present in both tables
LEFT JOIN	Returns all rows from the left table and matched rows from the right table.	All records from left, even if no match
RIGHT JOIN	Returns all rows from the right table and matched rows from the left table.	All records from right, even if no match
FULL JOIN	Returns all rows when there is a match in either table.	All records from both tables
CROSS JOIN	Returns the Cartesian product of both tables.	Generating all possible combinations

Examples:

```
-- INNER JOIN
SELECT a.*, b.* FROM TableA a INNER JOIN TableB b ON a.id = b.a_id;

-- LEFT JOIN
SELECT a.*, b.* FROM TableA a LEFT JOIN TableB b ON a.id = b.a_id;
```

```
-- RIGHT JOIN
SELECT a.*, b.* FROM TableA a RIGHT JOIN TableB b ON a.id = b.a_id;

-- FULL OUTER JOIN
SELECT a.*, b.* FROM TableA a FULL OUTER JOIN TableB b ON a.id = b.a_id;

-- CROSS JOIN
SELECT a.*, b.* FROM TableA a CROSS JOIN TableB b;
```

> **Summary:**
>
> - Use INNER JOIN for matched records only.
> - Use LEFT/RIGHT JOIN to include all records from one side.
> - Use FULL JOIN for all records from both sides.
> - Use CROSS JOIN for all combinations (rare).

---

2. What is the difference between a CROSS JOIN and a FULL OUTER JOIN?

Answer:

Feature	CROSS JOIN	FULL OUTER JOIN
Purpose	Cartesian product (all combinations)	All rows from both tables, matching where possible
Join Condition	None	Required (ON clause)
Result Size	Rows in A × Rows in B	All matched + unmatched rows from both tables
Use Case	Generating combinations	Combining all data, showing matches and non-matches

Example:

```
-- CROSS JOIN
SELECT a.*, b.* FROM TableA a CROSS JOIN TableB b;

-- FULL OUTER JOIN
SELECT a.*, b.* FROM TableA a FULL OUTER JOIN TableB b ON a.id = b.a_id;
```

---

3. Write a SQL query to retrieve the first and last names of employees along with the names of their managers.

Answer:

```
SELECT
        e.FirstName AS EmployeeFirstName,
        e.LastName AS EmployeeLastName,
        m.FirstName AS ManagerFirstName,
        m.LastName AS ManagerLastName
FROM Employees e
LEFT JOIN Managers m ON e.ManagerID = m.ManagerID;
```

- **LEFT JOIN** includes employees without managers (manager fields will be NULL).

---

4. Write a SQL query to find the average salary for each department.

Answer:

```
SELECT
        d.DepartmentName,
        AVG(e.Salary) AS AverageSalary
FROM Departments d
LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID
GROUP BY d.DepartmentName;
```

- **LEFT JOIN** ensures all departments are listed, even those without employees.

---

5. Write a SQL query to list all products that have never been ordered.

Answer:

```
SELECT p.ProductID, p.ProductName
FROM Product p
LEFT JOIN Orders o ON p.ProductID = o.ProductID
WHERE o.OrderID IS NULL;
```

- **Alternative:** Use `NOT EXISTS` or `NOT IN` for the same result.

---

6. Write a SQL query to list all employees who are also managers (self-join).

Answer:

```
SELECT DISTINCT
        e.EmployeeID,
        e.FirstName,
        e.LastName
```

```
FROM Employees e
INNER JOIN Employees m ON e.EmployeeID = m.ManagerID;
```

- **Self-join** finds employees who are referenced as managers by others.

---

7. What is a self-join, and when might you use it? Provide an example scenario.

Answer:

A **self-join** joins a table to itself, useful for hierarchical data (e.g., employees and their managers).

Example:

```
SELECT
        e.FirstName AS EmployeeFirstName,
        e.LastName AS EmployeeLastName,
        m.FirstName AS ManagerFirstName,
        m.LastName AS ManagerLastName
FROM Employees e
LEFT JOIN Employees m ON e.ManagerID = m.EmployeeID;
```

---

8. How would you join more than two tables in a single SQL query? What factors affect performance?

Answer:

- **Multiple Joins:** Chain JOIN clauses, using aliases for clarity.
- **Performance Factors:** Indexes, join order, join type, data volume, query complexity.

Example:

```
SELECT
        e.FirstName AS EmployeeFirstName,
        d.DepartmentName,
        m.FirstName AS ManagerFirstName
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID
LEFT JOIN Managers m ON e.ManagerID = m.ManagerID;
```

---

9. Explain how an OUTER JOIN works when one side has no matching rows. How does this differ from an INNER JOIN?

Answer:

- **OUTER JOIN:** Returns all rows from one (or both) tables, filling with NULLs where there is no match.

- **INNER JOIN:** Returns only rows with matches in both tables.

Example:

```
-- LEFT OUTER JOIN
SELECT e.FirstName, d.DepartmentName
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

Working with Views

Views

1. What is a database view, and can you update data in the base tables through it?

Answer:

A **view** is a virtual table based on a SQL query.

- **Updatable:** Simple views (single table, no aggregates).
- **Read-only:** Complex views (joins, aggregates).

Example:

```
CREATE VIEW vw_EmployeeNames AS
SELECT EmployeeID, FirstName, LastName FROM Employees;

UPDATE vw_EmployeeNames SET FirstName = 'John' WHERE EmployeeID = 1;
```

---

2. What is the difference between a standard view and a materialized (or indexed) view?

Feature	Standard View	Materialized/Indexed View
Storage	No data stored	Stores result set physically
Performance	Slower for complex queries	Faster for repeated access
Refresh	Always current	Needs refresh (manual/automatic)

---

3. What happens if a materialized view is being refreshed and a user queries it at the same time?

Answer:

- Most databases serve **old data** until refresh completes.
- Some may block queries or return an error if unavailable.

---

4. When would you use a view in a database design? What benefits do views provide?

- **Security:** Restrict access to sensitive data.
- **Abstraction:** Hide complex logic.
- **Simplification:** Provide a simple interface.
- **Consistency:** Centralize business logic.

---

5. Can you create an index on a view? What are the implications?

Answer:

- Yes, in some databases (e.g., SQL Server: **indexed view**).
- **Benefits:** Improves performance for complex queries.
- **Drawbacks:** Increases storage and maintenance; restrictions on view definition.

---

6. How do you modify or drop a view if the underlying table schema changes?

Answer:

- **Modify:** `CREATE OR REPLACE VIEW` or `ALTER VIEW`.
- **Drop:** `DROP VIEW view_name;`
- If a column is dropped from the base table, the view becomes invalid.

---

7. What is the difference between a view and a temporary table?

Aspect	View	Temporary Table
Persistence	Definition persists; data always live	Exists only for session/transaction
Storage	No data stored (unless materialized)	Physically stores data
Use Case	Abstraction, security	Intermediate results, complex logic

---

Stored Procedures & Functions

Stored Procedures & Functions

1. What is the difference between a stored procedure and a user-defined function in SQL?

Aspect	Stored Procedure	User-Defined Function
Can modify data	Yes	No (except special cases)
Use in SELECT	No	Yes
Transaction control	Yes	No
Return type	None, scalar, resultset	Scalar or table

---

2. What are the advantages and disadvantages of using stored procedures?

Advantages:

- Encapsulate business logic
- Improve performance (precompiled)
- Enhance security
- Reduce network traffic

Disadvantages:

- Harder to version/deploy
- May increase DB server load
- Logic split complicates maintenance

---

3. Can you perform INSERT/UPDATE/DELETE operations inside a SQL function?

Answer:
In most databases, **user-defined functions cannot perform DML** (INSERT/UPDATE/DELETE) to ensure determinism and safety in queries.

---

4. What is a table-valued function and when would you use one?

Answer:
A **table-valued function (TVF)** returns a table and is used in the FROM clause for reusable query logic.

Example:

```
CREATE FUNCTION dbo.GetEmployeesByDept(@DeptID INT)
RETURNS TABLE
AS
RETURN (
        SELECT EmployeeID, FirstName, LastName
        FROM Employees
        WHERE DepartmentID = @DeptID
);

SELECT * FROM dbo.GetEmployeesByDept(10);
```

---

5. When would you use a stored procedure instead of inline SQL queries in an application?

- Centralize/reuse business logic
- Improve security
- Reduce SQL injection risk
- Optimize complex/repetitive operations
- Batch multiple statements

6. How do you pass parameters to and receive results from stored procedures?

Example (SQL Server):

```
CREATE PROCEDURE GetEmployeeByID
        @EmpID INT,
        @FirstName NVARCHAR(50) OUTPUT
AS
BEGIN
        SELECT @FirstName = FirstName FROM Employees WHERE EmployeeID = @EmpID;
END;

DECLARE @Name NVARCHAR(50);
EXEC GetEmployeeByID 1, @Name OUTPUT;
SELECT @Name;
```

7. How would you debug or test a slow or failing stored procedure in production?

- Capture execution plans
- Use logging/print statements
- Test with sample data
- Check for blocking/deadlocks
- Review schema/data changes
- Use profiling tools

8. How do you grant a user permission to execute a specific stored procedure?

```
GRANT EXECUTE ON OBJECT::GetEmployeeByID TO username;
```

Triggers & Automation

Triggers

1. What is a trigger in SQL, and when would you use one?

Answer:

A **trigger** is a special stored procedure that automatically executes in response to events (INSERT, UPDATE, DELETE) on a table or view.

Use Cases:
Auditing, enforcing constraints, cascading updates/deletes, logging, automation.

Example:

```
CREATE TRIGGER trg_AuditSalaryChange
ON Employees
AFTER UPDATE
AS
BEGIN
        INSERT INTO SalaryAudit (EmployeeID, OldSalary, NewSalary, ChangedAt)
        SELECT i.EmployeeID, d.Salary, i.Salary, GETDATE()
        FROM inserted i
        JOIN deleted d ON i.EmployeeID = d.EmployeeID
        WHERE i.Salary <> d.Salary;
END;
```

---

2. What is the difference between an AFTER trigger and an INSTEAD OF trigger?

- **AFTER trigger:** Executes after the triggering event.
- **INSTEAD OF trigger:** Replaces the triggering event (often used on views).

---

3. What are the “inserted” and “deleted” magic tables in SQL Server triggers?

- **inserted:** Holds new rows for INSERT/UPDATE.
- **deleted:** Holds old rows for DELETE/UPDATE.

---

4. How can triggers be used to enforce business rules or data integrity?

- **Auditing:** Log changes.
- **Enforcing Rules:** Prevent deletion if child rows exist.
- **Data Integrity:** Maintain derived columns.

Example:

```
CREATE TRIGGER trg_PreventDeptDelete
ON Departments
INSTEAD OF DELETE
AS
BEGIN
        IF EXISTS (SELECT 1 FROM Employees e JOIN deleted d ON e.DepartmentID =
d.DepartmentID)
                RAISERROR('Cannot delete department with employees.', 16, 1);
        ELSE
                DELETE FROM Departments WHERE DepartmentID IN (SELECT DepartmentID
FROM deleted);
END;
```

---

5. What are the potential drawbacks of using triggers?

- **Performance:** Extra processing on DML operations.
- **Hidden Logic:** Harder to debug/maintain.
- **Complexity:** Nested/recursive triggers can cause issues.
- **Portability:** Syntax/behavior varies by DB.

---

6. How do INSTEAD OF triggers on a view work?

- Intercept DML on a view and define custom logic for handling those operations.

Example:

```
CREATE TRIGGER trg_UpdateEmpDept
ON vw_EmployeeDept
INSTEAD OF UPDATE
AS
BEGIN
        UPDATE Employees
        SET FirstName = i.FirstName
        FROM inserted i
        WHERE Employees.EmployeeID = i.EmployeeID;
END;
```

---

7. Can triggers call stored procedures, and are there any limitations?

- Yes, but avoid transactional conflicts, recursion, and long-running logic.

Transactions & Concurrency Control

Transactions & Concurrency

1. What are the ACID properties of a database transaction?

- **Atomicity:** All or nothing.
- **Consistency:** Preserves rules.
- **Isolation:** Transactions don't interfere.
- **Durability:** Changes are permanent after commit.

---

2. What are the different SQL isolation levels and what phenomena do they prevent?

Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
READ UNCOMMITTED	Possible	Possible	Possible

Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
READ COMMITTED	Prevented	Possible	Possible
REPEATABLE READ	Prevented	Prevented	Possible
SERIALIZABLE	Prevented	Prevented	Prevented

---

3. What is a deadlock and how can you prevent or resolve deadlocks?

- **Deadlock:** Two or more transactions block each other.
- **Prevention:** Access tables in same order, keep transactions short.
- **Resolution:** DB aborts one transaction (victim).

---

4. How do you control transactions in SQL? Give an example.

```
BEGIN TRANSACTION;
        UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
        UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
IF @@ERROR <> 0
        ROLLBACK;
ELSE
        COMMIT;
```

---

5. If you run a long SELECT query while another transaction is updating, will you see old or new data?

- **Default (READ COMMITTED):** You see only committed data at the time each row is read.

---

6. What is the difference between pessimistic and optimistic locking?

- **Pessimistic:** Locks data when read; use for high-conflict.
- **Optimistic:** No locks when reading; checks for changes before writing.

---

7. What is a savepoint in a transaction, and how do you use it?

```
BEGIN TRANSACTION;
        UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
        SAVE TRANSACTION Save1;
        UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
        -- If needed:
        ROLLBACK TRANSACTION Save1;
COMMIT;
```

---

8. How do two-phase commit protocols work in distributed transactions?

1. **Prepare phase:** Coordinator asks all nodes if they can commit.
2. **Commit phase:** If all agree, coordinator tells all to commit; else, roll back.

---

9. How can you identify and terminate a blocking or long-running transaction?

- **Identify:** Use system views (e.g., `sys.dm_exec_requests`, `sp_who2`).
- **Terminate:** Use `KILL session_id` (SQL Server) or `ALTER SYSTEM KILL SESSION` (Oracle).

---

10. What is deadlock detection, and how does the database engine choose a deadlock victim?

- The DB detects deadlocks and rolls back the "victim" (usually least costly transaction).

Performance Tuning & Query Optimization

Performance Tuning & Query Optimization

1. What is a query execution plan and how do you use it to improve performance?

Answer:

A **query execution plan** is a roadmap showing how the database will execute a query, including operations, access methods, and estimated costs.

- **How to View:** Use `EXPLAIN`, `EXPLAIN PLAN`, or graphical tools.
- **Optimization:** Identify expensive operations, add indexes, rewrite queries.

Example:

```
EXPLAIN SELECT * FROM Employees WHERE DepartmentID = 10;
```

---

2. How would you optimize a slow SQL query in production?

- Analyze execution plan
- Add indexes on WHERE/JOIN/ORDER BY columns
- Rewrite queries for efficiency
- Select only needed columns
- Partition large tables
- Update statistics & rebuild indexes

---

3. What is the difference between UNION and UNION ALL in SQL?

Operator	Duplicates Removed?	Performance
UNION	Yes	Slower
UNION ALL	No	Faster

Example:

```
SELECT Name FROM Employees
UNION
SELECT Name FROM Managers;

SELECT Name FROM Employees
UNION ALL
SELECT Name FROM Managers;
```

---

4. How can you find duplicate rows in a table using SQL?

```
SELECT column1, column2, COUNT(*) AS duplicate_count
FROM table_name
GROUP BY column1, column2
HAVING COUNT(*) > 1;
```

---

5. Write a SQL query to find the 10th highest salary in an Employee table.

Approach 1 (MySQL/PostgreSQL):

```
SELECT MIN(Salary) AS TenthHighestSalary
FROM (
        SELECT DISTINCT Salary
        FROM Employee
        ORDER BY Salary DESC
        LIMIT 10
) AS Top10;
```

Approach 2 (Window Function):

```
SELECT Salary AS TenthHighestSalary
FROM (
        SELECT Salary, DENSE_RANK() OVER (ORDER BY Salary DESC) AS rnk
        FROM Employee
) t
WHERE rnk = 10;
```

6. How would you retrieve the last 5 records (by date or ID) from a table?

```
SELECT *
FROM table_name
ORDER BY date_column DESC
LIMIT 5;
```

7. Write a SQL query to exclude specific values (e.g., select all rows except those where ID is X or Y).

```
SELECT *
FROM Student
WHERE ID NOT IN (101, 102);
```

8. How do you retrieve the Nth record (e.g., the 3rd record) from a table?

MySQL/PostgreSQL:

```
SELECT *
FROM table_name
ORDER BY ordering_column
LIMIT 1 OFFSET 2; -- 3rd record (OFFSET is zero-based)
```

SQL Server:

```
SELECT *
FROM (
        SELECT *, ROW_NUMBER() OVER (ORDER BY ordering_column) AS rn
        FROM table_name
) t
WHERE rn = 3;
```

9. How do you obtain the CREATE TABLE DDL for an existing table in SQL?

Database	Command/Tool
MySQL	`SHOW CREATE TABLE employees;`
PostgreSQL	`pg_dump -U username -d dbname -t employees --schema-only`

Database	Command/Tool
SQL Server	SSMS: Script Table as → CREATE To
Oracle	`SELECT DBMS_METADATA.GET_DDL('TABLE', 'EMPLOYEES') FROM DUAL;`
SQLite	`SELECT sql FROM sqlite_master WHERE type='table' AND name='employees';`

10. Explain the difference between the RANK() and DENSE_RANK() window functions.

Function	Ranking Behavior	Example Output (for ties)
RANK()	Skips ranks after ties (gaps)	1, 2, 2, 4
DENSE_RANK()	No gaps after ties (consecutive)	1, 2, 2, 3

11. When would you use ROW_NUMBER(), RANK(), or DENSE_RANK() in a query? Give a use case.

- **ROW_NUMBER():** Pagination, deduplication, selecting Nth row.
- **RANK():** Competition ranking with gaps after ties.
- **DENSE_RANK():** Leaderboard/reporting with consecutive ranks.

12. Write a SQL query to compute the median number of searches made by users.

Standard SQL:

```
SELECT AVG(search_count) AS median_searches
FROM (
        SELECT
                search_count,
                ROW_NUMBER() OVER (ORDER BY search_count) AS rn,
                COUNT(*) OVER () AS total
        FROM user_search_summary
) t
WHERE
        rn = (total + 1) / 2
        OR (total % 2 = 0 AND rn = (total / 2) + 1);
```

If supported:

```
SELECT
        PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY search_count) AS
median_searches
FROM user_search_summary;
```

13. Write a SQL query to calculate the sum of odd-numbered and even-numbered measurements separately for each day.

```
SELECT
        day,
        SUM(CASE WHEN MOD(measurement_number, 2) = 1 THEN value ELSE 0 END) AS
odd_sum,
        SUM(CASE WHEN MOD(measurement_number, 2) = 0 THEN value ELSE 0 END) AS
even_sum
FROM measurements
GROUP BY day;
```

---

14. Write a SQL query to get the average review rating for each product for each month.

PostgreSQL:

```
SELECT
        product_id,
        DATE_TRUNC('month', review_date) AS review_month,
        AVG(rating) AS avg_rating
FROM reviews
GROUP BY product_id, DATE_TRUNC('month', review_date)
ORDER BY product_id, review_month;
```

MySQL:

```
SELECT
        product_id,
        DATE_FORMAT(review_date, '%Y-%m') AS review_month,
        AVG(rating) AS avg_rating
FROM reviews
GROUP BY product_id, review_month
ORDER BY product_id, review_month;
```

Prepared by GitHub Copilot — Your AI SQL Interview Coach