01 SQL Interview Questions & Answers

SQL Interview Questions & Answers

Basic SQL Questions

DDL & DML

What is the difference between DDL and DML? Provide examples of each.

DDL (Data Definition Language) is used to define and modify database structure:

- **Purpose:** Create, modify, or delete database objects
- **Examples:** CREATE, ALTER, DROP, TRUNCATE
- **Auto-commit:** DDL statements are automatically committed

DML (Data Manipulation Language) is used to manipulate data within tables:

- **Purpose:** Insert, update, delete, or retrieve data
- **Examples:** INSERT, UPDATE, DELETE, SELECT
- **Transaction control:** Can be rolled back using ROLLBACK

```sql
-- DDL Examples
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    salary DECIMAL(10,2)
);

ALTER TABLE employees ADD COLUMN department VARCHAR(30);
DROP TABLE temp_table;

-- DML Examples
INSERT INTO employees VALUES (1, 'John Doe', 50000);
UPDATE employees SET salary = 55000 WHERE emp_id = 1;
DELETE FROM employees WHERE emp_id = 1;
SELECT * FROM employees;
```

How do you rename a table in SQL?

```sql
-- Standard SQL (MySQL, PostgreSQL)
ALTER TABLE old_table_name RENAME TO new_table_name;

-- SQL Server
EXEC sp_rename 'old_table_name', 'new_table_name';
```

```sql
-- Oracle
ALTER TABLE old_table_name RENAME TO new_table_name;
```

Write a query to rename a column in a table.

```sql
-- MySQL
ALTER TABLE employees CHANGE old_column_name new_column_name VARCHAR(50);

-- PostgreSQL
ALTER TABLE employees RENAME COLUMN old_column_name TO new_column_name;

-- SQL Server
EXEC sp_rename 'employees.old_column_name', 'new_column_name', 'COLUMN';

-- Oracle
ALTER TABLE employees RENAME COLUMN old_column_name TO new_column_name;
```

What are the different types of indexes in SQL?

1. **Clustered Index**
   - Physical ordering of data matches index order
   - One per table (usually on primary key)
   - Faster for range queries
2. **Non-Clustered Index**
   - Separate structure pointing to data rows
   - Multiple allowed per table
   - Faster for specific lookups
3. **Unique Index**
   - Ensures uniqueness of indexed columns
   - Automatically created for PRIMARY KEY and UNIQUE constraints
4. **Composite Index**
   - Index on multiple columns
   - Order of columns matters for query optimization
5. **Partial Index**
   - Index on subset of rows (with WHERE condition)
   - Saves space and improves performance

```sql
-- Creating different types of indexes
CREATE INDEX idx_employee_name ON employees(name);
CREATE UNIQUE INDEX idx_employee_email ON employees(email);
CREATE INDEX idx_emp_dept_salary ON employees(department, salary);
CREATE INDEX idx_high_salary ON employees(salary) WHERE salary > 50000;
```

Data Types & Constraints

What is the difference between CHAR and VARCHAR?

Aspect	CHAR	VARCHAR
Storage	Fixed-length	Variable-length
Padding	Right-padded with spaces	No padding
Storage Efficiency	Less efficient for short strings	More efficient
Performance	Slightly faster for fixed-size data	Slightly slower due to length calculation
Use Case	Status codes, country codes	Names, descriptions

```
-- CHAR example - always uses 10 bytes
name CHAR(10) -- 'John' stored as 'John      '

-- VARCHAR example - uses only needed bytes + length info
name VARCHAR(10) -- 'John' stored as 'John' (4 bytes + length)
```

What is a PRIMARY KEY? What is an ALTERNATIVE KEY?

PRIMARY KEY:

- Uniquely identifies each record in a table
- Cannot contain NULL values
- Only one per table
- Automatically creates a unique index
- Cannot be changed once defined

ALTERNATIVE KEY (Candidate Key):

- Could serve as primary key but wasn't chosen
- Must be unique and not null
- Multiple alternative keys possible per table
- Often implemented using UNIQUE constraint

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,            -- Primary Key
    ssn VARCHAR(11) UNIQUE NOT NULL,   -- Alternative Key
    email VARCHAR(100) UNIQUE NOT NULL, -- Alternative Key
    name VARCHAR(50)
);
```

How much storage does CHAR(10) take versus VARCHAR(10)?

CHAR(10):

- Always uses exactly 10 bytes
- 'Hello' → 'Hello ' (10 bytes)

VARCHAR(10):

- Uses 1-2 bytes for length + actual data length
- 'Hello' → 1-2 bytes (length) + 5 bytes (data) = 6-7 bytes
- Maximum: 1-2 bytes + 10 bytes = 11-12 bytes

What are the different constraints you can apply to a column?

1. **NOT NULL:** Prevents null values
2. **UNIQUE:** Ensures all values are unique
3. **PRIMARY KEY:** Combination of NOT NULL and UNIQUE
4. **FOREIGN KEY:** Links to another table's primary key
5. **CHECK:** Validates data against a condition
6. **DEFAULT:** Provides default value when none specified

```sql
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    age INT CHECK (age >= 18 AND age <= 65),
    department VARCHAR(30) DEFAULT 'General',
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employees(emp_id)
);
```

Normalization

Explain the different normal forms in database normalization.

1NF (First Normal Form):

- Each column contains atomic (indivisible) values
- No repeating groups or arrays
- Each record is unique

2NF (Second Normal Form):

- Must be in 1NF
- All non-key attributes fully depend on the primary key
- Eliminates partial dependencies

3NF (Third Normal Form):

- Must be in 2NF
- No transitive dependencies

- Non-key attributes depend only on primary key

BCNF (Boyce-Codd Normal Form):

- Stricter version of 3NF
- Every determinant must be a candidate key

Normalization Issues Example

Problem Table:

```
Orders(order_id, customer_name, customer_address, product1, price1, product2,
price2)
```

Issues:

1. **1NF Violation:** Multiple products in single row
2. **Data Redundancy:** Customer info repeated
3. **Update Anomalies:** Changing customer address requires multiple updates
4. **Insert Anomalies:** Can't add customer without order

Solution:

```
-- Normalized structure
Customers(customer_id, customer_name, customer_address)
Products(product_id, product_name, price)
Orders(order_id, customer_id, order_date)
Order_Items(order_id, product_id, quantity)
```

Query-Based Questions

Basic Queries

Find employees in HR department with salary above 50,000

```sql
SELECT e.emp_id, e.name, e.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE d.dept_name = 'HR'
  AND e.salary > 50000;
```

Aggregation & Grouping

Find the number of employees in each department

```sql
SELECT d.dept_name, COUNT(e.emp_id) as employee_count
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
```

```sql
GROUP BY d.dept_id, d.dept_name
ORDER BY employee_count DESC;
```

Find average salary excluding employees above company average

```sql
WITH company_avg AS (
    SELECT AVG(salary) as avg_salary
    FROM employees
)
SELECT d.dept_name, AVG(e.salary) as dept_avg_salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
CROSS JOIN company_avg ca
WHERE e.salary ≤ ca.avg_salary
GROUP BY d.dept_id, d.dept_name;
```

Join Operations

Find all employees and their managers

```sql
SELECT
    e.emp_id,
    e.name as employee_name,
    COALESCE(m.name, 'No Manager') as manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.emp_id
ORDER BY e.emp_id;
```

Highest Salary Queries

Find the 3rd highest salary

```sql
-- Method 1: Using DENSE_RANK()
WITH salary_ranks AS (
    SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) as rank
    FROM employees
)
SELECT DISTINCT salary
FROM salary_ranks
WHERE rank = 3;

-- Method 2: Using LIMIT/TOP with subquery
SELECT DISTINCT salary
FROM employees
```

```sql
ORDER BY salary DESC
LIMIT 1 OFFSET 2;  -- PostgreSQL/MySQL

-- Method 3: Using correlated subquery
SELECT DISTINCT salary
FROM employees e1
WHERE 2 = (
    SELECT COUNT(DISTINCT salary)
    FROM employees e2
    WHERE e2.salary > e1.salary
);
```

Find department with highest average salary

```sql
WITH dept_avg AS (
    SELECT
        d.dept_id,
        d.name as dept_name,
        AVG(e.salary) as avg_salary
    FROM departments d
    JOIN employees e ON d.dept_id = e.dept_id
    GROUP BY d.dept_id, d.name
)
SELECT dept_name, avg_salary
FROM dept_avg
WHERE avg_salary = (SELECT MAX(avg_salary) FROM dept_avg);
```

Top N Records

Find top 5 customers with highest total order amounts

```sql
SELECT
    customer_id,
    SUM(order_amount) as total_amount
FROM orders
GROUP BY customer_id
ORDER BY total_amount DESC
LIMIT 5;
```

Find top 5 selling products in last 3 months

```sql
SELECT
    product_id,
    SUM(quantity) as total_sold
```

```sql
FROM sales
WHERE sale_date ≥ CURRENT_DATE - INTERVAL '3 months'
GROUP BY product_id
ORDER BY total_sold DESC
LIMIT 5;
```

Advanced SQL Questions

Window Functions

Rank employees by salary within each department

```sql
SELECT
    emp_id,
    name,
    dept_id,
    salary,
    RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) as salary_rank
FROM employees
ORDER BY dept_id, salary_rank;
```

Difference between RANK(), DENSE_RANK(), and ROW_NUMBER()

```sql
SELECT
    name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) as row_num,
    RANK() OVER (ORDER BY salary DESC) as rank_num,
    DENSE_RANK() OVER (ORDER BY salary DESC) as dense_rank_num
FROM employees
ORDER BY salary DESC;

/*
Example Output:
name     salary  row_num  rank_num  dense_rank_num
Alice    100000     1        1            1
Bob      100000     2        1            1   -- Same salary
Charlie   90000     3        3            2   -- RANK skips 2, DENSE_RANK doesn't
David     90000     4        3            2
Eve       80000     5        5            3   -- RANK skips 4
*/
```

Key Differences:

- **ROW_NUMBER():** Always unique, sequential numbers

- **RANK():** Same rank for ties, skips next rank(s)
- **DENSE_RANK():** Same rank for ties, no gaps in ranking

Compare current month's sales with previous month using LAG/LEAD

```sql
SELECT
    month,
    sales_amount,
    LAG(sales_amount) OVER (ORDER BY month) as prev_month_sales,
    LEAD(sales_amount) OVER (ORDER BY month) as next_month_sales,
    sales_amount - LAG(sales_amount) OVER (ORDER BY month) as
month_over_month_change,
    ROUND(
        (sales_amount - LAG(sales_amount) OVER (ORDER BY month)) * 100.0 /
        LAG(sales_amount) OVER (ORDER BY month), 2
    ) as percent_change
FROM monthly_sales
ORDER BY month;
```

Duplicate Handling

Find duplicate records

```sql
-- Method 1: Using GROUP BY and HAVING
SELECT
    name, email, COUNT(*) as duplicate_count
FROM employees
GROUP BY name, email
HAVING COUNT(*) > 1;

-- Method 2: Using window functions
SELECT *
FROM (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY name, email ORDER BY emp_id) as row_num
    FROM employees
) t
WHERE row_num > 1;
```

Delete duplicate records keeping only one copy

```sql
-- Using CTE and ROW_NUMBER()
WITH duplicate_cte AS (
    SELECT *,
```

```
            ROW_NUMBER() OVER (
                PARTITION BY name, email
                ORDER BY emp_id  -- Keep the one with smallest emp_id
            ) as row_num
    FROM employees
)
DELETE FROM employees
WHERE emp_id IN (
    SELECT emp_id
    FROM duplicate_cte
    WHERE row_num > 1
);

-- Alternative method using self-join
DELETE e1 FROM employees e1
INNER JOIN employees e2
WHERE e1.emp_id > e2.emp_id
  AND e1.name = e2.name
  AND e1.email = e2.email;
```

Correlated Subqueries

Find employees who earn more than their department's average salary

```
SELECT emp_id, name, dept_id, salary
FROM employees e1
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
    WHERE e2.dept_id = e1.dept_id
);

-- Alternative using window functions (more efficient)
WITH dept_avg AS (
    SELECT *,
           AVG(salary) OVER (PARTITION BY dept_id) as dept_avg_salary
    FROM employees
)
SELECT emp_id, name, dept_id, salary
FROM dept_avg
WHERE salary > dept_avg_salary;
```

Query Performance Optimization

How would you optimize this query?

```sql
-- Original query
SELECT *
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= '2024-01-01'
  AND o.status = 'completed'
  AND c.country = 'USA';
```

Optimization strategies:

1. **Add indexes:**

```sql
CREATE INDEX idx_orders_date_status ON orders(order_date, status);
CREATE INDEX idx_customers_country ON customers(country);
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
```

2. **Select only needed columns:**

```sql
SELECT o.order_id, o.order_date, o.order_amount, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= '2024-01-01'
  AND o.status = 'completed'
  AND c.country = 'USA';
```

3. **Filter early and use covering indexes:**

```sql
-- Create covering index
CREATE INDEX idx_orders_covering ON orders(order_date, status, customer_id,
order_id, order_amount);

-- Rewrite query to filter orders first
SELECT o.order_id, o.order_date, o.order_amount, c.customer_name
FROM (
    SELECT customer_id, order_id, order_date, order_amount
    FROM orders
    WHERE order_date >= '2024-01-01' AND status = 'completed'
) o
JOIN customers c ON o.customer_id = c.customer_id
WHERE c.country = 'USA';
```

What will you do if your query is running slow?

1. **Analyze Execution Plan:**

```
EXPLAIN ANALYZE SELECT ... -- PostgreSQL
EXPLAIN EXECUTION PLAN SELECT ... -- Oracle
SET SHOWPLAN_ALL ON; SELECT ... -- SQL Server
```

2. **Check for missing indexes:**
   - Look for table scans in execution plan
   - Add indexes on WHERE, JOIN, and ORDER BY columns
3. **Optimize WHERE clauses:**
   - Put most selective conditions first
   - Avoid functions on indexed columns
   - Use EXISTS instead of IN for large subqueries
4. **Review JOIN operations:**
   - Ensure JOIN conditions use indexed columns
   - Consider JOIN order for multiple tables
5. **Update table statistics:**

```
ANALYZE TABLE table_name; -- MySQL
UPDATE STATISTICS table_name; -- SQL Server
```

6. **Consider query rewriting:**
   - Replace correlated subqueries with JOINs
   - Use window functions instead of multiple GROUP BYs
   - Break complex queries into simpler parts

Conditional Aggregation

Count employees in each department who joined in 2024

```
-- With GROUP BY
SELECT
    d.dept_name,
    COUNT(CASE WHEN YEAR(e.hire_date) = 2024 THEN 1 END) as new_hires_2024,
    COUNT(e.emp_id) as total_employees
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
GROUP BY d.dept_id, d.dept_name;

-- Without GROUP BY (using window functions)
SELECT DISTINCT
    d.dept_name,
    COUNT(CASE WHEN YEAR(e.hire_date) = 2024 THEN 1 END)
        OVER (PARTITION BY d.dept_id) as new_hires_2024,
```

```sql
    COUNT(e.emp_id) OVER (PARTITION BY d.dept_id) as total_employees
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id;
```

Common Table Expressions (CTE)

Rewrite subquery using CTE

```sql
-- Original with subquery
SELECT dept_name,
       (SELECT COUNT(*)
        FROM employees e
        WHERE e.dept_id = d.dept_id) as emp_count
FROM departments d;

-- Rewritten with CTE
WITH dept_employee_count AS (
    SELECT
        dept_id,
        COUNT(*) as emp_count
    FROM employees
    GROUP BY dept_id
)
SELECT
    d.dept_name,
    COALESCE(dec.emp_count, 0) as emp_count
FROM departments d
LEFT JOIN dept_employee_count dec ON d.dept_id = dec.dept_id;
```

Date-Based Queries

Find all orders from the last 3 months with rouge date handling

```sql
SELECT *
FROM orders
WHERE order_date >= CURRENT_DATE - INTERVAL '3 months'
  AND order_date <= CURRENT_DATE  -- Handle future dates
  AND order_date >= '1900-01-01'  -- Handle unrealistic past dates
  AND order_date IS NOT NULL      -- Handle NULL dates
ORDER BY order_date DESC;

-- More robust version with CASE statement
SELECT *,
    CASE
        WHEN order_date IS NULL THEN 'Missing Date'
```

```sql
        WHEN order_date > CURRENT_DATE THEN 'Future Date'
        WHEN order_date < '1900-01-01' THEN 'Invalid Past Date'
        ELSE 'Valid Date'
    END as date_status
FROM orders
WHERE order_date ≥ CURRENT_DATE - INTERVAL '3 months'
  AND order_date ≤ CURRENT_DATE
  AND order_date ≥ '1900-01-01'
  AND order_date IS NOT NULL;
```

Transaction Management

What are ACID properties?

ACID ensures database transactions are processed reliably:

1. **Atomicity**: Transaction is all-or-nothing
    - Either all operations succeed or all fail
    - No partial transactions
2. **Consistency**: Database remains in valid state
    - All constraints and rules are maintained
    - Data integrity is preserved
3. **Isolation:** Concurrent transactions don't interfere
    - Transactions appear to run sequentially
    - Prevents dirty reads, phantom reads
4. **Durability:** Committed changes are permanent
    - Survive system crashes
    - Changes are written to persistent storage

```sql
-- Example transaction demonstrating ACID
BEGIN TRANSACTION;
    UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

    -- Check if both accounts have sufficient balance
    IF (SELECT balance FROM accounts WHERE account_id = 1) < 0
        ROLLBACK;  -- Atomicity: undo all changes
    ELSE
        COMMIT;    -- Durability: make changes permanent
END TRANSACTION;
```

NULL Handling

Handle null values in salary calculation

```sql
-- Different approaches to handle NULLs
SELECT
    emp_id,
    name,
    salary,

    -- Replace NULL with 0
    COALESCE(salary, 0) as salary_with_zero,

    -- Replace NULL with average salary
    COALESCE(salary, (SELECT AVG(salary) FROM employees WHERE salary IS NOT NULL))
as salary_with_avg,

    -- Use CASE statement
    CASE
        WHEN salary IS NULL THEN 0
        ELSE salary
    END as salary_case,

    -- Calculate bonus (10% of salary, 0 if NULL)
    COALESCE(salary * 0.10, 0) as bonus,

    -- Total compensation
    COALESCE(salary, 0) + COALESCE(bonus_amount, 0) as total_compensation
FROM employees;

-- Aggregate functions with NULL handling
SELECT
    dept_id,
    COUNT(*) as total_employees,
    COUNT(salary) as employees_with_salary,  -- Excludes NULLs
    AVG(salary) as avg_salary,               -- Ignores NULLs
    AVG(COALESCE(salary, 0)) as avg_salary_with_zeros
FROM employees
GROUP BY dept_id;
```

Data Cleaning Queries

Handle duplicate values and standardize data

```sql
-- Clean and standardize customer data
WITH cleaned_customers AS (
    SELECT
        customer_id,
        -- Standardize names
```

```
        INITCAP(TRIM(LOWER(customer_name))) as clean_name,

        -- Clean phone numbers
        REGEXP_REPLACE(phone, '[^0-9]', '') as clean_phone,

        -- Standardize email
        LOWER(TRIM(email)) as clean_email,

        -- Clean addresses
        INITCAP(TRIM(REGEXP_REPLACE(address, '\s+', ' '))) as clean_address,

        -- Handle NULL values
        COALESCE(city, 'Unknown') as city,
        UPPER(COALESCE(state, 'XX')) as state
    FROM customers
    WHERE email IS NOT NULL
      AND email LIKE '%@%'  -- Basic email validation
)
SELECT * FROM cleaned_customers;

-- Remove duplicates based on business rules
WITH ranked_customers AS (
    SELECT *,
           ROW_NUMBER() OVER (
               PARTITION BY clean_email
               ORDER BY
                   CASE WHEN phone IS NOT NULL THEN 1 ELSE 2 END,
                   customer_id
           ) as rn
    FROM cleaned_customers
)
SELECT * FROM ranked_customers WHERE rn = 1;
```

Query Performance Best Practices

Indexing Strategy

1. **Primary Key Index:** Automatically created
2. **Foreign Key Indexes:** Always index foreign key columns
3. **Covering Indexes:** Include all columns needed by query
4. **Composite Indexes:** Order columns by selectivity (most selective first)

```
-- Good composite index order
CREATE INDEX idx_orders_date_status_customer
ON orders(order_date, status, customer_id);
```

```sql
-- Include frequently selected columns
CREATE INDEX idx_orders_covering
ON orders(order_date, status)
INCLUDE (customer_id, order_amount);
```

Query Writing Best Practices

1. **Select only needed columns**
2. **Use appropriate JOINs**
3. **Filter early in subqueries**
4. **Avoid functions on indexed columns in WHERE clause**
5. **Use EXISTS instead of IN for large result sets**
6. **Consider using UNION ALL instead of UNION when duplicates are acceptable**

```sql
-- Bad: Function on indexed column
SELECT * FROM orders WHERE YEAR(order_date) = 2024;

-- Good: Range condition
SELECT * FROM orders
WHERE order_date >= '2024-01-01'
  AND order_date < '2025-01-01';
```