DBMS Interview Questions for Beginners - Complete Guide

1. What is DBMS? What are its advantages?

Answer: DBMS (Database Management System) is software that manages databases and provides an interface between the database and users/applications.

Advantages:

- Data redundancy control
- Data consistency
- Data security
- Data integrity
- Concurrent access
- Backup and recovery

2. What is the difference between DBMS and RDBMS?

DBMS	RDBMS
Stores data in files	Stores data in tables
No relationships between data	Relationships exist between tables
No ACID properties	Follows ACID properties
Example: File systems	Example: MySQL, Oracle

3. What are the different types of databases?

Answer:

- **Hierarchical Database:** Tree-like structure
- **Network Database:** Graph structure with multiple parent-child relationships
- **Relational Database:** Data stored in tables with relationships
- **Object-Oriented Database:** Data stored as objects
- **NoSQL Database:** Non-relational databases (MongoDB, Cassandra)

4. What is a Primary Key? Write SQL to create one.

Answer: Primary Key uniquely identifies each record in a table. It cannot be NULL and must be unique.

```sql
-- Creating table with primary key
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(100)
);

-- Adding primary key to existing table
ALTER TABLE Students ADD PRIMARY KEY (student_id);
```

5. What is a Foreign Key? Provide an example.

Answer: Foreign Key is a field that refers to the Primary Key of another table, establishing relationships between tables.

```sql
-- Parent table
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);

-- Child table with foreign key
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
);
```

6. What are the different types of keys in DBMS?

Answer:

- **Primary Key:** Uniquely identifies records
- **Foreign Key:** References primary key of another table
- **Candidate Key:** Attributes that can become primary key
- **Super Key:** Set of attributes that uniquely identifies records
- **Composite Key:** Primary key made of multiple columns
- **Unique Key:** Ensures uniqueness but allows one NULL value

```sql
-- Example of composite key
CREATE TABLE OrderDetails (
    order_id INT,
    product_id INT,
    quantity INT,
    PRIMARY KEY (order_id, product_id)
);
```

7. What is Normalization? Explain 1NF, 2NF, and 3NF.

Answer: Normalization is the process of organizing data in a database to reduce redundancy.

1NF (First Normal Form):

- Each column contains atomic values
- No repeating groups

2NF (Second Normal Form):

- Must be in 1NF
- No partial dependencies on primary key

3NF (Third Normal Form):

- Must be in 2NF
- No transitive dependencies

```sql
-- Unnormalized table
CREATE TABLE StudentCourses (
    student_id INT,
    student_name VARCHAR(50),
    course1 VARCHAR(50),
    course2 VARCHAR(50)
);

-- Normalized tables (1NF, 2NF, 3NF)
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50)
);

CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50)
);

CREATE TABLE Enrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
```

---

8. What are ACID properties?

Answer:

- **Atomicity:** Transaction is all-or-nothing
- **Consistency:** Database remains in valid state
- **Isolation:** Concurrent transactions don't interfere
- **Durability:** Committed changes are permanent

```
-- Example of 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;
COMMIT;
```

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

Command	Purpose	Rollback	Speed
DELETE	Remove specific rows	Yes	Slow
DROP	Remove entire table	No	Fast
TRUNCATE	Remove all rows	No	Fast

```
-- DELETE - removes specific rows
DELETE FROM Employees WHERE dept_id = 10;

-- TRUNCATE - removes all rows
TRUNCATE TABLE Employees;

-- DROP - removes entire table
DROP TABLE Employees;
```

10. What are Joins? Explain different types.

Answer: Joins combine rows from multiple tables based on related columns.

```
-- Sample tables
CREATE TABLE Employees (
    emp_id INT,
    emp_name VARCHAR(50),
    dept_id INT
);

CREATE TABLE Departments (
    dept_id INT,
    dept_name VARCHAR(50)
);
```

```sql
-- INNER JOIN - matching records from both tables
SELECT e.emp_name, d.dept_name
FROM Employees e
INNER JOIN Departments d ON e.dept_id = d.dept_id;

-- LEFT JOIN - all records from left table
SELECT e.emp_name, d.dept_name
FROM Employees e
LEFT JOIN Departments d ON e.dept_id = d.dept_id;

-- RIGHT JOIN - all records from right table
SELECT e.emp_name, d.dept_name
FROM Employees e
RIGHT JOIN Departments d ON e.dept_id = d.dept_id;

-- FULL OUTER JOIN - all records from both tables
SELECT e.emp_name, d.dept_name
FROM Employees e
FULL OUTER JOIN Departments d ON e.dept_id = d.dept_id;
```

11. What is a View? How to create one?

Answer: A View is a virtual table based on the result of an SQL statement.

```sql
-- Creating a view
CREATE VIEW EmployeeView AS
SELECT emp_id, emp_name, dept_name
FROM Employees e
JOIN Departments d ON e.dept_id = d.dept_id;

-- Using the view
SELECT * FROM EmployeeView;

-- Dropping a view
DROP VIEW EmployeeView;
```

12. What is an Index? Types of indexes?

Answer: Index is a database object that improves query performance.

Types:

- **Clustered Index:** Physically reorders data
- **Non-Clustered Index:** Logical ordering with pointers
- **Unique Index:** Ensures uniqueness
- **Composite Index:** Multiple columns

```sql
-- Creating indexes
CREATE INDEX idx_emp_name ON Employees(emp_name);
CREATE UNIQUE INDEX idx_emp_email ON Employees(email);
CREATE INDEX idx_emp_dept ON Employees(emp_id, dept_id);

-- Dropping index
DROP INDEX idx_emp_name;
```

---

13. What are Aggregate Functions?

Answer: Functions that perform calculations on multiple rows and return single value.

```sql
-- Common aggregate functions
SELECT
    COUNT(*) as total_employees,
    SUM(salary) as total_salary,
    AVG(salary) as average_salary,
    MAX(salary) as highest_salary,
    MIN(salary) as lowest_salary
FROM Employees;

-- GROUP BY with aggregate functions
SELECT dept_id, COUNT(*) as emp_count, AVG(salary) as avg_salary
FROM Employees
GROUP BY dept_id
HAVING COUNT(*) > 5;
```

---

14. What is the difference between WHERE and HAVING?

WHERE	HAVING
Filters rows before grouping	Filters groups after grouping
Cannot use aggregate functions	Can use aggregate functions
Used with SELECT, UPDATE, DELETE	Used with GROUP BY

```sql
-- WHERE clause example
SELECT * FROM Employees WHERE salary > 50000;

-- HAVING clause example
SELECT dept_id, AVG(salary)
FROM Employees
GROUP BY dept_id
HAVING AVG(salary) > 60000;
```

15. What are Subqueries? Types of subqueries?

Answer: Query within another query.

Types:

- **Single Row Subquery:** Returns one row
- **Multiple Row Subquery:** Returns multiple rows
- **Correlated Subquery:** References outer query
- **Non-Correlated Subquery:** Independent of outer query

```sql
-- Single row subquery
SELECT * FROM Employees
WHERE salary > (SELECT AVG(salary) FROM Employees);

-- Multiple row subquery
SELECT * FROM Employees
WHERE dept_id IN (SELECT dept_id FROM Departments WHERE dept_name LIKE 'IT%');

-- Correlated subquery
SELECT * FROM Employees e1
WHERE salary > (SELECT AVG(salary) FROM Employees e2 WHERE e1.dept_id =
e2.dept_id);
```

16. What is a Transaction? Transaction states?

Answer: Transaction is a unit of work performed against a database.

Transaction States:

- **Active:** Transaction is being executed
- **Partially Committed:** After final statement executed
- **Committed:** Transaction completed successfully
- **Failed:** Transaction cannot proceed
- **Aborted:** Transaction cancelled and rolled back

```sql
-- Transaction example
BEGIN TRANSACTION;
INSERT INTO Employees VALUES (101, 'John', 1, 50000);
UPDATE Departments SET emp_count = emp_count + 1 WHERE dept_id = 1;
COMMIT;

-- Transaction with error handling
BEGIN TRANSACTION;
INSERT INTO Employees VALUES (102, 'Jane', 2, 55000);
IF @@ERROR <> 0
    ROLLBACK;
```

```
ELSE
    COMMIT;
```

17. What are Stored Procedures? How to create them?

Answer: Pre-compiled SQL code stored in database for reuse.

```sql
-- Creating a stored procedure
DELIMITER //
CREATE PROCEDURE GetEmployeesByDept(IN dept_id INT)
BEGIN
    SELECT * FROM Employees WHERE dept_id = dept_id;
END //
DELIMITER ;

-- Calling stored procedure
CALL GetEmployeesByDept(1);

-- Stored procedure with parameters
DELIMITER //
CREATE PROCEDURE UpdateEmployeeSalary(
    IN emp_id INT,
    IN new_salary DECIMAL(10,2),
    OUT result VARCHAR(50)
)
BEGIN
    DECLARE emp_count INT;
    SELECT COUNT(*) INTO emp_count FROM Employees WHERE emp_id = emp_id;

    IF emp_count > 0 THEN
        UPDATE Employees SET salary = new_salary WHERE emp_id = emp_id;
        SET result = 'Success';
    ELSE
        SET result = 'Employee not found';
    END IF;
END //
DELIMITER ;
```

18. What are Triggers? Types of triggers?

Answer: Special stored procedures that automatically execute in response to database events.

Types:

- **BEFORE Triggers:** Execute before the triggering event
- **AFTER Triggers:** Execute after the triggering event
- **INSTEAD OF Triggers:** Replace the triggering event

```sql
-- BEFORE INSERT trigger
DELIMITER //
CREATE TRIGGER before_employee_insert
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
    SET NEW.created_date = NOW();
    SET NEW.emp_code = CONCAT('EMP', NEW.emp_id);
END //
DELIMITER ;

-- AFTER UPDATE trigger
DELIMITER //
CREATE TRIGGER after_salary_update
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO SalaryAudit (emp_id, old_salary, new_salary, change_date)
    VALUES (NEW.emp_id, OLD.salary, NEW.salary, NOW());
END //
DELIMITER ;
```

19. What is Deadlock? How to prevent it?

Answer: Deadlock occurs when two or more transactions wait for each other to release locks.

Prevention methods:

- Lock timeout
- Deadlock detection and resolution
- Proper lock ordering
- Minimize transaction time

```sql
-- Example that can cause deadlock
-- Transaction 1
BEGIN TRANSACTION;
UPDATE Employees SET salary = 60000 WHERE emp_id = 1;
UPDATE Departments SET budget = 100000 WHERE dept_id = 1;
COMMIT;

-- Transaction 2 (running simultaneously)
BEGIN TRANSACTION;
UPDATE Departments SET budget = 200000 WHERE dept_id = 1;
UPDATE Employees SET salary = 65000 WHERE emp_id = 1;
COMMIT;

-- Solution: Use consistent lock ordering
-- Both transactions should lock tables in same order
```

20. What is Concurrency Control?

Answer: Mechanism to ensure correct execution of concurrent transactions.

Techniques:

- **Locking:** Prevent conflicts using locks
- **Timestamping:** Use timestamps to order transactions
- **Optimistic Concurrency Control:** Allow conflicts, resolve later
- **Multiversion Concurrency Control:** Multiple versions of data

```sql
-- Locking example
BEGIN TRANSACTION;
SELECT * FROM Employees WHERE emp_id = 1 FOR UPDATE; -- Exclusive lock
UPDATE Employees SET salary = 55000 WHERE emp_id = 1;
COMMIT;
```

21. What are Constraints? Types of constraints?

Answer: Rules enforced on data columns to maintain data integrity.

```sql
-- Different types of constraints
CREATE TABLE Products (
    product_id INT PRIMARY KEY,                    -- Primary Key
    product_name VARCHAR(100) NOT NULL,            -- Not Null
    price DECIMAL(10,2) CHECK (price > 0),        -- Check
    category_id INT,
    email VARCHAR(100) UNIQUE,                     -- Unique
    created_date DATE DEFAULT CURRENT_DATE,       -- Default
    FOREIGN KEY (category_id) REFERENCES Categories(category_id)  -- Foreign Key
);

-- Adding constraints to existing table
ALTER TABLE Products ADD CONSTRAINT chk_price CHECK (price BETWEEN 1 AND 10000);
ALTER TABLE Products ADD CONSTRAINT uk_product_name UNIQUE (product_name);
```

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

Clustered Index	Non-Clustered Index
Physically reorders data	Logical ordering with pointers
One per table	Multiple per table

Clustered Index	Non-Clustered Index
Faster for range queries	Faster for exact matches
Larger storage overhead	Smaller storage overhead

```sql
-- Clustered index (usually on primary key)
CREATE CLUSTERED INDEX idx_emp_id ON Employees(emp_id);

-- Non-clustered index
CREATE NONCLUSTERED INDEX idx_emp_name ON Employees(emp_name);
CREATE NONCLUSTERED INDEX idx_emp_dept_salary ON Employees(dept_id, salary);
```

---

23. What is Database Schema?

Answer: Logical structure that defines how data is organized in a database.

Types:

- **Physical Schema:** How data is stored physically
- **Logical Schema:** Logical structure of database
- **View Schema:** How data appears to users

```sql
-- Creating schema
CREATE SCHEMA company_schema;

-- Creating table in schema
CREATE TABLE company_schema.employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50)
);

-- Using schema
SELECT * FROM company_schema.employees;
```

---

24. What are Window Functions?

Answer: Functions that perform calculations across a set of rows related to the current row.

```sql
-- Common window functions
SELECT
    emp_id,
    emp_name,
    salary,
    dept_id,
    -- Ranking functions
    ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) as row_num,
    RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) as rank_num,
    DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) as dense_rank,

    -- Aggregate functions
    SUM(salary) OVER (PARTITION BY dept_id) as dept_total_salary,
    AVG(salary) OVER (PARTITION BY dept_id) as dept_avg_salary,
    COUNT(*) OVER (PARTITION BY dept_id) as dept_emp_count,

    -- Value functions
    LAG(salary, 1) OVER (ORDER BY emp_id) as prev_salary,
    LEAD(salary, 1) OVER (ORDER BY emp_id) as next_salary
FROM Employees;
```

25. What is the difference between UNION and UNION ALL?

UNION	UNION ALL
Removes duplicates	Keeps duplicates
Slower performance	Faster performance
Implicit DISTINCT	No DISTINCT

```
-- UNION - removes duplicates
SELECT emp_id, emp_name FROM Employees WHERE dept_id = 1
UNION
SELECT emp_id, emp_name FROM Employees WHERE salary > 50000;

-- UNION ALL - keeps duplicates
SELECT emp_id, emp_name FROM Employees WHERE dept_id = 1
UNION ALL
SELECT emp_id, emp_name FROM Employees WHERE salary > 50000;
```

26. What is Cursor? How to use it?

Answer: Database object used to retrieve and manipulate data row by row.

```
-- Declaring and using cursor
DELIMITER //
CREATE PROCEDURE ProcessEmployees()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE emp_id INT;
    DECLARE emp_name VARCHAR(50);
    DECLARE emp_salary DECIMAL(10,2);
    -- Declare cursor
    DECLARE emp_cursor CURSOR FOR
        SELECT emp_id, emp_name, salary FROM Employees;

    -- Declare continue handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Open cursor
    OPEN emp_cursor;

    -- Loop through cursor
    read_loop: LOOP
        FETCH emp_cursor INTO emp_id, emp_name, emp_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Process each row
        IF emp_salary < 50000 THEN
            UPDATE Employees SET salary = salary * 1.1 WHERE emp_id = emp_id;
        END IF;
    END LOOP;

    -- Close cursor
    CLOSE emp_cursor;
END //
DELIMITER ;
```

27. What are CTE (Common Table Expressions)?

Answer: Temporary named result sets that exist within the scope of a single SQL statement.

```
-- Simple CTE
WITH DepartmentSalaries AS (
    SELECT dept_id, AVG(salary) as avg_salary
    FROM Employees
    GROUP BY dept_id
)
SELECT e.emp_name, e.salary, ds.avg_salary
FROM Employees e
JOIN DepartmentSalaries ds ON e.dept_id = ds.dept_id
WHERE e.salary > ds.avg_salary;

-- Recursive CTE - Employee hierarchy
WITH RECURSIVE EmployeeHierarchy AS (
    -- Anchor member (managers)
    SELECT emp_id, emp_name, manager_id, 1 as level
    FROM Employees
    WHERE manager_id IS NULL
    UNION ALL

    -- Recursive member
    SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
    FROM Employees e
    JOIN EmployeeHierarchy eh ON e.manager_id = eh.emp_id
)
SELECT * FROM EmployeeHierarchy ORDER BY level, emp_name;
```

28. What is the difference between CHAR and VARCHAR?

CHAR	VARCHAR
Fixed length	Variable length
Padded with spaces	No padding
Faster access	More storage efficient
Up to 255 characters	Up to 65,535 characters

```sql
-- CHAR vs VARCHAR example
CREATE TABLE DataTypes (
    id INT PRIMARY KEY,
    fixed_code CHAR(5),        -- Always uses 5 bytes
    variable_name VARCHAR(50)   -- Uses only needed bytes + 1-2 bytes overhead
);

INSERT INTO DataTypes VALUES (1, 'ABC', 'John');
-- fixed_code stores 'ABC  ' (with 2 trailing spaces)
-- variable_name stores 'John' (4 bytes + overhead)
```

29. What are Date and Time functions?

Answer: Functions to manipulate date and time values.

```sql
-- Common date/time functions
SELECT
    NOW() as current_datetime,
    CURDATE() as current_date,
    CURTIME() as current_time,

    -- Date formatting
    DATE_FORMAT(NOW(), '%Y-%m-%d %H:%i:%s') as formatted_date,

    -- Date arithmetic
    DATE_ADD(CURDATE(), INTERVAL 30 DAY) as thirty_days_later,
    DATE_SUB(CURDATE(), INTERVAL 1 MONTH) as one_month_ago,
    DATEDIFF('2024-12-31', CURDATE()) as days_until_new_year,

    -- Date parts
    YEAR(NOW()) as current_year,
    MONTH(NOW()) as current_month,
    DAY(NOW()) as current_day,
    HOUR(NOW()) as current_hour,

    -- Day of week/year
    DAYOFWEEK(CURDATE()) as day_of_week,
    DAYOFYEAR(CURDATE()) as day_of_year,
    WEEKOFYEAR(CURDATE()) as week_of_year;

-- Using date functions in queries
SELECT emp_name, hire_date,
    DATEDIFF(CURDATE(), hire_date) as days_employed,
    YEAR(CURDATE()) - YEAR(hire_date) as years_employed
FROM Employees
WHERE MONTH(hire_date) = MONTH(CURDATE());
```

---

30. What is the difference between INNER JOIN and OUTER JOIN?

Answer: INNER JOIN returns only matching records, while OUTER JOIN returns all records from one or both tables.

```
-- Sample data for demonstration
INSERT INTO Employees VALUES
(1, 'John', 10), (2, 'Jane', 20), (3, 'Bob', NULL);

INSERT INTO Departments VALUES
(10, 'IT'), (20, 'HR'), (30, 'Finance');

-- INNER JOIN - only matching records (2 results)
SELECT e.emp_name, d.dept_name
FROM Employees e
INNER JOIN Departments d ON e.dept_id = d.dept_id;

-- LEFT OUTER JOIN - all employees (3 results)
SELECT e.emp_name, d.dept_name
FROM Employees e
LEFT JOIN Departments d ON e.dept_id = d.dept_id;

-- RIGHT OUTER JOIN - all departments (4 results)
SELECT e.emp_name, d.dept_name
FROM Employees e
RIGHT JOIN Departments d ON e.dept_id = d.dept_id;

-- FULL OUTER JOIN - all records (5 results)
SELECT e.emp_name, d.dept_name
```

```sql
FROM Employees e
FULL OUTER JOIN Departments d ON e.dept_id = d.dept_id;
```

31. What are String Functions in SQL?

Answer: Functions to manipulate string/text data.

```sql
-- Common string functions
SELECT
    -- Length and case functions
    LENGTH('Hello World') as str_length,
    UPPER('hello world') as uppercase,
    LOWER('HELLO WORLD') as lowercase,

    -- Substring functions
    SUBSTRING('Hello World', 1, 5) as substring_result,
    LEFT('Hello World', 5) as left_chars,
    RIGHT('Hello World', 5) as right_chars,

    -- Search and replace
    LOCATE('World', 'Hello World') as position,
    REPLACE('Hello World', 'World', 'SQL') as replaced,

    -- Trimming
    LTRIM('  Hello World') as left_trim,
    RTRIM('Hello World  ') as right_trim,
    TRIM('  Hello World  ') as both_trim,

    -- Concatenation
    CONCAT('Hello', ' ', 'World') as concatenated,
    CONCAT_WS('-', 'Hello', 'World', 'SQL') as concat_with_separator;

-- Using string functions in queries
SELECT emp_name,
    UPPER(emp_name) as name_upper,
    CONCAT(emp_name, ' (ID: ', emp_id, ')') as formatted_name,
    LENGTH(emp_name) as name_length
FROM Employees
WHERE emp_name LIKE '%John%';
```

32. What is a Composite Key? Provide an example.

Answer: A composite key is a primary key composed of two or more columns.

```sql
-- Example: Order details where combination of order_id and product_id is unique
CREATE TABLE OrderDetails (
    order_id INT,
    product_id INT,
    quantity INT,
    unit_price DECIMAL(10,2),
    discount DECIMAL(5,2),

    -- Composite primary key
    PRIMARY KEY (order_id, product_id),

    -- Foreign keys
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

-- Another example: Student course enrollment
CREATE TABLE StudentCourses (
    student_id INT,
    course_id INT,
    semester VARCHAR(10),
    grade CHAR(2),

    -- Composite primary key
    PRIMARY KEY (student_id, course_id, semester)
);

-- Inserting data
INSERT INTO OrderDetails VALUES (1, 101, 5, 25.00, 0.10);
INSERT INTO OrderDetails VALUES (1, 102, 3, 15.00, 0.05);
INSERT INTO OrderDetails VALUES (2, 101, 2, 25.00, 0.00);
```

---

33. What is Database Backup and Recovery?

Answer: Process of creating copies of database and restoring them when needed.

Types of Backup:

- **Full Backup:** Complete database backup
- **Incremental Backup:** Only changes since last backup
- **Differential Backup:** Changes since last full backup
- **Transaction Log Backup:** Log of all transactions

```
-- Creating backup (SQL Server syntax)
BACKUP DATABASE CompanyDB
TO DISK = 'C:\Backups\CompanyDB_Full.bak'
WITH FORMAT, COMPRESSION;

-- Creating transaction log backup
BACKUP LOG CompanyDB
TO DISK = 'C:\Backups\CompanyDB_Log.trn';

-- Restoring database
```

```
RESTORE DATABASE CompanyDB
FROM DISK = 'C:\Backups\CompanyDB_Full.bak'
WITH REPLACE;

-- Point-in-time recovery
RESTORE DATABASE CompanyDB
FROM DISK = 'C:\Backups\CompanyDB_Full.bak'
WITH NORECOVERY;

RESTORE LOG CompanyDB
FROM DISK = 'C:\Backups\CompanyDB_Log.trn'
WITH STOPAT = '2024-06-27 14:30:00';
```

---

34. What are Numeric Functions in SQL?

Answer: Functions to perform mathematical operations on numeric data.

```
-- Common numeric functions
SELECT
    -- Basic math functions
    ABS(-15) as absolute_value,
    CEILING(15.3) as ceiling_value,
    FLOOR(15.8) as floor_value,
    ROUND(15.567, 2) as rounded_value,

    -- Power and root functions
    POWER(2, 3) as power_result,
    SQRT(16) as square_root,

    -- Trigonometric functions
    SIN(PI()/2) as sine_90_degrees,
    COS(0) as cosine_0_degrees,
    TAN(PI()/4) as tangent_45_degrees,

    -- Random and sign functions
    RAND() as random_number,
    SIGN(-5) as sign_negative,
    SIGN(5) as sign_positive,

    -- Modulo operation
    MOD(10, 3) as modulo_result;

-- Using numeric functions with employee data
SELECT emp_name, salary,
    ROUND(salary * 1.1, 2) as salary_with_raise,
    CEILING(salary / 12) as monthly_salary_ceiling,
    FLOOR(salary * 0.12) as annual_tax_floor
FROM Employees
WHERE ABS(salary - 50000) < 10000;
```

35. What is the difference between SQL and NoSQL databases?

SQL Databases	NoSQL Databases
Structured data (tables)	Unstructured/semi-structured
ACID properties	Eventually consistent
Vertical scaling	Horizontal scaling
Complex queries	Simple queries
Schema-based	Schema-less
Examples: MySQL, Oracle	Examples: MongoDB, Cassandra

```sql
-- SQL Database example
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100),
    profile JSON  -- Even SQL databases now support JSON
);

INSERT INTO Users VALUES
(1, 'john_doe', 'john@email.com', '{"age": 30, "city": "New York"}');

-- Querying JSON data in SQL
SELECT username,
    JSON_EXTRACT(profile, '$.age') as age,
    JSON_EXTRACT(profile, '$.city') as city
FROM Users
WHERE JSON_EXTRACT(profile, '$.age') > 25;
```

36. What are the different types of SQL statements?

Answer: SQL statements are categorized into different types:

DDL (Data Definition Language):

- CREATE, ALTER, DROP, TRUNCATE

DML (Data Manipulation Language):

- SELECT, INSERT, UPDATE, DELETE

DCL (Data Control Language):

- GRANT, REVOKE

TCL (Transaction Control Language):

- COMMIT, ROLLBACK, SAVEPOINT

```sql
-- DDL Examples
CREATE TABLE Products (product_id INT, product_name VARCHAR(50));
ALTER TABLE Products ADD price DECIMAL(10,2);
DROP TABLE Products;

-- DML Examples
INSERT INTO Products VALUES (1, 'Laptop', 999.99);
UPDATE Products SET price = 899.99 WHERE product_id = 1;
DELETE FROM Products WHERE product_id = 1;
SELECT * FROM Products;

-- DCL Examples
GRANT SELECT, INSERT ON Products TO user1;
REVOKE INSERT ON Products FROM user1;

-- TCL Examples
BEGIN TRANSACTION;
INSERT INTO Products VALUES (2, 'Mouse', 29.99);
SAVEPOINT sp1;
UPDATE Products SET price = 25.99 WHERE product_id = 2;
ROLLBACK TO sp1;  -- Rollback to savepoint
COMMIT;  -- Commit the transaction
```

---

37. What is the CASE statement? Provide examples.

Answer: CASE statement provides conditional logic in SQL queries.

```sql
-- Simple CASE statement
SELECT emp_name, salary,
    CASE
        WHEN salary < 30000 THEN 'Low'
        WHEN salary BETWEEN 30000 AND 60000 THEN 'Medium'
        WHEN salary > 60000 THEN 'High'
        ELSE 'Unknown'
    END as salary_category
FROM Employees;

-- CASE with aggregate functions
SELECT dept_id,
    COUNT(*) as total_employees,
    SUM(CASE WHEN salary > 50000 THEN 1 ELSE 0 END) as high_salary_count,
    AVG(CASE WHEN gender = 'M' THEN salary END) as avg_male_salary,
    AVG(CASE WHEN gender = 'F' THEN salary END) as avg_female_salary
FROM Employees
GROUP BY dept_id;

-- CASE in UPDATE statement
```

```sql
UPDATE Employees
SET salary = CASE
    WHEN performance_rating = 'Excellent' THEN salary * 1.15
    WHEN performance_rating = 'Good' THEN salary * 1.10
    WHEN performance_rating = 'Average' THEN salary * 1.05
    ELSE salary
END;

-- Searched CASE vs Simple CASE
-- Simple CASE
SELECT emp_name,
    CASE dept_id
        WHEN 1 THEN 'IT Department'
        WHEN 2 THEN 'HR Department'
        WHEN 3 THEN 'Finance Department'
        ELSE 'Other Department'
    END as department_name
FROM Employees;
```

---

38. What is Data Integrity? Types of Data Integrity?

Answer: Data Integrity ensures accuracy, consistency, and reliability of data.

Types:

- **Entity Integrity:** Primary key constraints
- **Referential Integrity:** Foreign key constraints
- **Domain Integrity:** Data type and check constraints
- **User-Defined Integrity:** Business rules and triggers

```sql
-- Entity Integrity - Primary Key
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,  -- Cannot be NULL or duplicate
    customer_name VARCHAR(100) NOT NULL
);

-- Referential Integrity - Foreign Key
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);

-- Domain Integrity - Check Constraints
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10,2) CHECK (price > 0),
    category VARCHAR(50) CHECK (category IN ('Electronics', 'Clothing', 'Books')),
    stock_quantity INT CHECK (stock_quantity >= 0)
);

-- User-Defined Integrity - Trigger
DELIMITER //
CREATE TRIGGER check_order_total
BEFORE INSERT ON OrderDetails
FOR EACH ROW
BEGIN
    DECLARE total_amount DECIMAL(10,2);
    SELECT SUM(quantity * unit_price) INTO total_amount
    FROM OrderDetails WHERE order_id = NEW.order_id;

    IF (total_amount + (NEW.quantity * NEW.unit_price)) > 10000 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Order total cannot exceed
$10,000';
    END IF;
END //
DELIMITER ;
```

39. What is the difference between RANK(), DENSE_RANK(), and ROW_NUMBER()?

Answer: All are window functions for ranking, but handle ties differently.

Function	Ties Handling	Next Rank
ROW_NUMBER()	Assigns unique numbers	Sequential
RANK()	Same rank for ties	Skips numbers
DENSE_RANK()	Same rank for ties	No gaps

```
-- Sample data with salary ties
CREATE TABLE SalaryExample (
    emp_id INT,
    emp_name VARCHAR(50),
    salary DECIMAL(10,2)
);

INSERT INTO SalaryExample VALUES
(1, 'Alice', 70000),
(2, 'Bob', 65000),
(3, 'Charlie', 70000),  -- Tie with Alice
(4, 'David', 60000),
(5, 'Eve', 65000);      -- Tie with Bob

-- Comparing ranking functions
SELECT emp_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 SalaryExample
ORDER BY salary DESC;

-- Results:
-- Alice    70000  1  1  1
-- Charlie  70000  2  1  1  (same dense_rank, different row_number)
-- Bob      65000  3  3  2  (rank skips 2, dense_rank doesn't)
-- Eve      65000  4  3  2
-- David    60000  5  5  3

-- Practical example: Top 3 highest paid employees per department
SELECT dept_id, emp_name, salary, dense_rank_num
FROM (
    SELECT dept_id, emp_name, salary,
        DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) as dense_rank_num
    FROM Employees
) ranked
WHERE dense_rank_num <= 3;
```

40. What are Set Operations in SQL?

Answer: Set operations combine results from multiple SELECT statements.

Operations:

- **UNION:** Combines results, removes duplicates
- **UNION ALL:** Combines results, keeps duplicates
- **INTERSECT:** Returns common records
- **EXCEPT/MINUS:** Returns records from first query not in second

```
-- Sample tables for demonstration
CREATE TABLE CurrentEmployees (emp_id INT, emp_name VARCHAR(50));
CREATE TABLE FormerEmployees (emp_id INT, emp_name VARCHAR(50));

INSERT INTO CurrentEmployees VALUES (1, 'John'), (2, 'Jane'), (3, 'Bob');
INSERT INTO FormerEmployees VALUES (2, 'Jane'), (4, 'Alice'), (5, 'Charlie');

-- UNION - All unique employees (current and former)
SELECT emp_id, emp_name, 'Current' as status FROM CurrentEmployees
UNION
SELECT emp_id, emp_name, 'Former' as status FROM FormerEmployees;

-- UNION ALL - All employees including duplicates
SELECT emp_id, emp_name FROM CurrentEmployees
UNION ALL
SELECT emp_id, emp_name FROM FormerEmployees;
```

```sql
-- INTERSECT - Employees who are both current and former (rehired)
SELECT emp_id, emp_name FROM CurrentEmployees
INTERSECT
SELECT emp_id, emp_name FROM FormerEmployees;

-- EXCEPT/MINUS - Current employees who were never former employees
SELECT emp_id, emp_name FROM CurrentEmployees
EXCEPT
SELECT emp_id, emp_name FROM FormerEmployees;

-- Complex example: Department-wise employee analysis
SELECT dept_id, 'High Performer' as category, COUNT(*) as count
FROM Employees
WHERE performance_rating >= 4
GROUP BY dept_id

UNION ALL

SELECT dept_id, 'Average Performer' as category, COUNT(*) as count
FROM Employees
WHERE performance_rating = 3
GROUP BY dept_id

UNION ALL

SELECT dept_id, 'Low Performer' as category, COUNT(*) as count
FROM Employees
WHERE performance_rating < 3
GROUP BY dept_id

ORDER BY dept_id, category;
```

Bonus Questions (41-45)

41. What is Database Partitioning?

Answer: Dividing large tables into smaller, manageable pieces while maintaining logical unity.

Types:

- **Horizontal Partitioning:** Split rows (Range, Hash, List)
- **Vertical Partitioning:** Split columns
- **Functional Partitioning:** Split by feature/module

```sql
-- Range Partitioning Example (MySQL)
CREATE TABLE Sales (
    sale_id INT,
    sale_date DATE,
    amount DECIMAL(10,2),
    customer_id INT
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025),
    PARTITION p_future VALUES LESS THAN MAXVALUE
);

-- Hash Partitioning
CREATE TABLE Customers (
    customer_id INT,
    customer_name VARCHAR(100),
    email VARCHAR(100)
)
PARTITION BY HASH(customer_id)
PARTITIONS 4;

-- List Partitioning
CREATE TABLE Employees (
    emp_id INT,
    emp_name VARCHAR(50),
    department VARCHAR(50)
)
PARTITION BY LIST COLUMNS(department) (
    PARTITION p_tech VALUES IN ('IT', 'Engineering', 'QA'),
    PARTITION p_business VALUES IN ('Sales', 'Marketing', 'HR'),
    PARTITION p_ops VALUES IN ('Operations', 'Finance', 'Admin')
);
```

42. What are Materialized Views?

Answer: Physical copies of query results stored as tables, updated periodically.

Differences from Regular Views:

- Materialized views store data physically
- Better performance for complex queries
- Need to be refreshed to update data
- Consume storage space

```
-- Creating Materialized View (Oracle syntax)
CREATE MATERIALIZED VIEW mv_department_summary
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
AS
SELECT
    d.dept_id,
    d.dept_name,
    COUNT(e.emp_id) as employee_count,
    AVG(e.salary) as avg_salary,
    SUM(e.salary) as total_salary,
    MAX(e.hire_date) as latest_hire_date
FROM Departments d
LEFT JOIN Employees e ON d.dept_id = e.dept_id
GROUP BY d.dept_id, d.dept_name;

-- Refreshing Materialized View
EXEC DBMS_MVIEW.REFRESH('mv_department_summary', 'C');

-- Using Materialized View
SELECT * FROM mv_department_summary WHERE employee_count > 10;

-- PostgreSQL Materialized View
CREATE MATERIALIZED VIEW mv_monthly_sales AS
SELECT
    DATE_TRUNC('month', order_date) as month,
    COUNT(*) as order_count,
    SUM(total_amount) as total_sales
FROM Orders
WHERE order_date >= '2024-01-01'
GROUP BY DATE_TRUNC('month', order_date);

-- Refresh PostgreSQL Materialized View
REFRESH MATERIALIZED VIEW mv_monthly_sales;
```

---

43. What is Database Sharding?

Answer: Horizontal partitioning across multiple database servers/instances.

Types:

- **Range-based Sharding:** Based on value ranges
- **Hash-based Sharding:** Based on hash function
- **Directory-based Sharding:** Lookup service determines shard

```
-- Example: User data sharding by user_id
-- Shard 1: user_id 1-1000000
CREATE TABLE users_shard1 (
    user_id INT PRIMARY KEY CHECK (user_id BETWEEN 1 AND 1000000),
    username VARCHAR(50),
    email VARCHAR(100),
    created_date DATE
);

-- Shard 2: user_id 1000001-2000000
CREATE TABLE users_shard2 (
    user_id INT PRIMARY KEY CHECK (user_id BETWEEN 1000001 AND 2000000),
    username VARCHAR(50),
    email VARCHAR(100),
    created_date DATE
);

-- Application logic for sharding
-- Function to determine shard based on user_id
/*
function getShardForUser(user_id) {
    if (user_id <= 1000000) return 'shard1';
    else if (user_id <= 2000000) return 'shard2';
    // ... more shards
}
*/

-- Geographic sharding example
CREATE TABLE orders_us (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    region VARCHAR(10) DEFAULT 'US'
);

CREATE TABLE orders_eu (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    region VARCHAR(10) DEFAULT 'EU'
);
```

44. What are Database Design Patterns?

Answer: Common solutions to recurring database design problems.

Common Patterns:

- **One-to-One:** User and Profile
- **One-to-Many:** Department and Employees
- **Many-to-Many:** Students and Courses
- **Self-Referencing:** Employee and Manager
- **Inheritance:** Table per hierarchy, Table per type

```
-- One-to-One Pattern
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100)
);

CREATE TABLE UserProfiles (
    user_id INT PRIMARY KEY,
    first_name VARCHAR(50),
```

```sql
    last_name VARCHAR(50),
    bio TEXT,
    avatar_url VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
);

-- One-to-Many Pattern
CREATE TABLE Categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(50)
);

CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category_id INT,
    FOREIGN KEY (category_id) REFERENCES Categories(category_id)
);

-- Many-to-Many Pattern
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100)
);

CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100)
);

CREATE TABLE StudentCourses (
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    grade CHAR(2),
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);

-- Self-Referencing Pattern (Employee-Manager)
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES Employees(emp_id)
);

-- Inheritance Pattern - Table Per Type
CREATE TABLE Vehicles (
    vehicle_id INT PRIMARY KEY,
    make VARCHAR(50),
    model VARCHAR(50),
    year INT
```

```sql
);

CREATE TABLE Cars (
    vehicle_id INT PRIMARY KEY,
    doors INT,
    fuel_type VARCHAR(20),
    FOREIGN KEY (vehicle_id) REFERENCES Vehicles(vehicle_id)
);

CREATE TABLE Motorcycles (
    vehicle_id INT PRIMARY KEY,
    engine_size INT,
    has_sidecar BOOLEAN,
    FOREIGN KEY (vehicle_id) REFERENCES Vehicles(vehicle_id)
);
```

---

45. What are Performance Optimization Techniques?

Answer: Methods to improve database query performance and overall system efficiency.

Techniques:

- **Indexing:** Create appropriate indexes
- **Query Optimization:** Write efficient queries
- **Partitioning:** Split large tables
- **Caching:** Store frequently accessed data
- **Database Design:** Proper normalization/denormalization

```sql
-- Index Optimization
-- Create composite index for common query patterns
CREATE INDEX idx_emp_dept_salary ON Employees(dept_id, salary);
CREATE INDEX idx_order_date_customer ON Orders(order_date, customer_id);

-- Query Optimization Examples

-- INEFFICIENT: Using functions in WHERE clause
SELECT * FROM Employees WHERE YEAR(hire_date) = 2023;

-- EFFICIENT: Using date ranges
SELECT * FROM Employees
WHERE hire_date >= '2023-01-01' AND hire_date < '2024-01-01';

-- INEFFICIENT: SELECT *
SELECT * FROM Employees WHERE dept_id = 10;

-- EFFICIENT: Select only needed columns
SELECT emp_id, emp_name, salary FROM Employees WHERE dept_id = 10;

-- INEFFICIENT: Correlated subquery
SELECT emp_name FROM Employees e1
```

```sql
WHERE salary > (SELECT AVG(salary) FROM Employees e2 WHERE e1.dept_id =
e2.dept_id);

-- EFFICIENT: Window function
SELECT emp_name FROM (
    SELECT emp_name, salary,
           AVG(salary) OVER (PARTITION BY dept_id) as avg_dept_salary
    FROM Employees
) t WHERE salary > avg_dept_salary;

-- Query Execution Plan Analysis
EXPLAIN SELECT e.emp_name, d.dept_name
FROM Employees e
JOIN Departments d ON e.dept_id = d.dept_id
WHERE e.salary > 50000;

-- Optimization with proper indexing
CREATE INDEX idx_emp_salary ON Employees(salary);
CREATE INDEX idx_emp_dept_join ON Employees(dept_id);

-- Partitioning for large tables
CREATE TABLE LogEntries (
    log_id BIGINT AUTO_INCREMENT,
    log_date DATE,
    log_level VARCHAR(10),
    message TEXT,
    PRIMARY KEY (log_id, log_date)
)
PARTITION BY RANGE (TO_DAYS(log_date)) (
    PARTITION p_2024_01 VALUES LESS THAN (TO_DAYS('2024-02-01')),
    PARTITION p_2024_02 VALUES LESS THAN (TO_DAYS('2024-03-01')),
    PARTITION p_2024_03 VALUES LESS THAN (TO_DAYS('2024-04-01'))
);

-- Query optimization with LIMIT
-- Use LIMIT for pagination instead of loading all data
SELECT emp_id, emp_name, salary
FROM Employees
ORDER BY salary DESC
LIMIT 10 OFFSET 20;  -- Page 3, 10 records per page
```

Summary of Key Interview Topics:

1. **Database Fundamentals:** DBMS vs RDBMS, Types of databases
2. **Keys and Constraints:** Primary, Foreign, Unique keys, Check constraints
3. **Normalization:** 1NF, 2NF, 3NF and their importance
4. **ACID Properties:** Atomicity, Consistency, Isolation, Durability
5. **SQL Operations:** DDL, DML, DCL, TCL commands
6. **Joins:** Inner, Outer, Left, Right, Full joins
7. **Advanced SQL:** Subqueries, Window functions, CTEs

8. **Indexing:** Types of indexes and their performance impact
9. **Transactions:** Transaction states, Concurrency control
10. **Database Objects:** Views, Stored procedures, Triggers
11. **Performance:** Query optimization, Partitioning, Sharding
12. **Functions:** String, Numeric, Date/Time functions

Tips for Interview Success:

- Practice writing SQL queries by hand
- Understand the theoretical concepts behind each topic
- Be able to explain trade-offs (e.g., normalization vs performance)
- Know when to use different types of joins and indexes
- Understand real-world scenarios where these concepts apply