SQL Interview Preparation Guide

Complete Beginner to Advanced SQL Reference

This comprehensive guide covers everything you need to know about SQL for technical interviews:

- **Database fundamentals and concepts**
- **All SQL commands from basic to advanced**
- **JOIN operations and subqueries**
- **Window functions and complex queries**
- **Performance optimization techniques**
- **100+ practical examples and solutions**
- **Common interview questions and patterns**

Perfect for beginners who know nothing about SQL and want to master it for interviews!

Table of Contents

23. [Practice Problems](#practice-problems)

SQL Basics

What is SQL?

SQL (Structured Query Language) is a programming language designed for managing and manipulating relational databases. It's used to:

- Query data **`FROM`** databases

- **`INSERT`**, **`UPDATE`**, and **`DELETE`** records

- **`CREATE`** and modify **`DATABASE`** structures

- Control access to data

Key SQL Commands Categories

1. **DDL (Data Definition Language)**: **`CREATE`**, **`ALTER`**, **`DROP`**

2. **DML (Data Manipulation Language)**: **`SELECT`**, **`INSERT`**, **`UPDATE`**, **`DELETE`**

3. **DCL (Data Control Language)**: GRANT, REVOKE

4. **TCL (Transaction Control Language)**: COMMIT, ROLLBACK, SAVEPOINT

Database Fundamentals

Understanding database concepts including tables, relationships, keys, and normalization principles.

What is a Database?

A **DATABASE** is an organized collection of structured data stored electronically in a computer system.

What is a Table?

A **TABLE** is a collection of related data entries consisting of columns and rows.

Key Terms

- **Row/Record**: A single entry in a **TABLE**

- **Column/Field**: A vertical entity in a **TABLE** containing all information associated with a specific field

- **Primary Key**: A unique identifier for each record in a **TABLE**

- **Foreign Key**: A field that links to the primary key of another **TABLE**

- **Schema**: The structure of a **DATABASE** (tables, columns, relationships)

Sample Database Structure

```
-- Employees Table
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100),
    department_id INT,
    salary DECIMAL(10,2),
    hire_date DATE
);
```

```
-- Departments Table
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100),
    manager_id INT
);
```

Data Types

SQL supports various data types including numeric, string, date/time, and boolean types.

Common SQL Data Types

Numeric Types

```
INT             -- Integer numbers (-2,147,483,648 to 2,147,483,647)
BIGINT          -- Large integer numbers
DECIMAL(p,s)    -- Fixed-point numbers (p=precision, s=scale)
NUMERIC(p,s)    -- Same as DECIMAL
FLOAT           -- Floating-point numbers
REAL            -- Single precision floating-point
```

String Types

```
CHAR(n)         -- Fixed-length string (n characters)
VARCHAR(n)      -- Variable-length string (up to n characters)
TEXT            -- Large variable-length string
NCHAR(n)        -- Fixed-length Unicode string
NVARCHAR(n)     -- Variable-length Unicode string
```

Date and Time Types

```
DATE            -- Date (YYYY-MM-DD)
TIME            -- Time (HH:MM:SS)
DATETIME        -- Date and time (YYYY-MM-DD HH:MM:SS)
TIMESTAMP       -- Date and time with timezone
```

```
YEAR            -- Year (YYYY)
```

Boolean Type

```
BOOLEAN         -- TRUE/FALSE values
BIT             -- 0 or 1
```

SELECT Statement

The most fundamental SQL command for retrieving data from database tables.

Basic SELECT Syntax

```
SELECT column1, column2, ...
FROM table_name;
```

Examples

```
-- Select all columns
SELECT * FROM employees;

-- Select specific columns
SELECT first_name, last_name, salary FROM employees;

-- Select with alias
SELECT first_name AS "First Name",
       last_name AS "Last Name",
       salary AS "Annual Salary"
FROM employees;

-- Select distinct values
SELECT DISTINCT department_id FROM employees;

-- Select with calculations
SELECT first_name,
       last_name,
```

```
       salary,
       salary * 12 AS annual_salary
FROM employees;
```

Filtering Data

Using WHERE clauses and conditions to filter and retrieve specific data.

WHERE Clause

```
-- Basic WHERE
SELECT * FROM employees WHERE department_id = 10;

-- Multiple conditions with AND
SELECT * FROM employees
WHERE department_id = 10 AND salary > 50000;

-- Multiple conditions with OR
SELECT * FROM employees
WHERE department_id = 10 OR department_id = 20;

-- NOT operator
SELECT * FROM employees
WHERE NOT department_id = 10;
```

Comparison Operators

```
-- Equal
SELECT * FROM employees WHERE salary = 50000;

-- Not equal
SELECT * FROM employees WHERE salary != 50000;
SELECT * FROM employees WHERE salary <> 50000;

-- Greater than
SELECT * FROM employees WHERE salary > 50000;

-- Less than
SELECT * FROM employees WHERE salary < 50000;
```

```
-- Greater than or equal
SELECT * FROM employees WHERE salary >= 50000;

-- Less than or equal
SELECT * FROM employees WHERE salary <= 50000;
```

BETWEEN Operator

```
-- Between (inclusive)
SELECT * FROM employees WHERE salary BETWEEN 40000 AND 60000;

-- Not between
SELECT * FROM employees WHERE salary NOT BETWEEN 40000 AND 60000;
```

IN Operator

```
-- IN operator
SELECT * FROM employees WHERE department_id IN (10, 20, 30);

-- NOT IN operator
SELECT * FROM employees WHERE department_id NOT IN (10, 20, 30);
```

LIKE Operator (Pattern Matching)

```
-- Starts with 'J'
SELECT * FROM employees WHERE first_name LIKE 'J%';

-- Ends with 'son'
SELECT * FROM employees WHERE last_name LIKE '%son';

-- Contains 'an'
SELECT * FROM employees WHERE first_name LIKE '%an%';

-- Second character is 'o'
SELECT * FROM employees WHERE first_name LIKE '_o%';

-- Exactly 5 characters
SELECT * FROM employees WHERE first_name LIKE '_____';
```

IS NULL and IS NOT NULL

```
-- Check for NULL values
SELECT * FROM employees WHERE manager_id IS NULL;

-- Check for NOT NULL values
SELECT * FROM employees WHERE manager_id IS NOT NULL;
```

Sorting and Grouping

ORDER BY Clause

```
-- Sort ascending (default)
SELECT * FROM employees ORDER BY salary;

-- Sort descending
SELECT * FROM employees ORDER BY salary DESC;

-- Sort by multiple columns
SELECT * FROM employees ORDER BY department_id, salary DESC;

-- Sort by column position
SELECT first_name, last_name, salary
FROM employees
ORDER BY 3 DESC; -- Sort by 3rd column (salary)
```

GROUP BY Clause

```
-- Group by single column
SELECT department_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id;

-- Group by multiple columns
SELECT department_id,
       EXTRACT(YEAR FROM hire_date) as hire_year,
       COUNT(*) as employee_count
FROM employees
```

```
GROUP BY department_id, EXTRACT(YEAR FROM hire_date);
```

HAVING Clause

```
-- HAVING (filter groups)
SELECT department_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;

-- HAVING with aggregate functions
SELECT department_id, AVG(salary) as avg_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 50000;
```

Joins

Types of Joins

1. **INNER JOIN**: Returns records that have matching values in both tables

2. **LEFT JOIN**: Returns all records FROM left TABLE, matched records FROM right TABLE

3. **RIGHT JOIN**: Returns all records FROM right TABLE, matched records FROM left TABLE

4. **FULL OUTER JOIN**: Returns all records when there's a match in either TABLE

5. **CROSS JOIN**: Returns Cartesian product of both tables

INNER JOIN

```
-- Basic INNER JOIN
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
```

```
INNER JOIN departments d ON e.department_id = d.department_id;

-- Multiple table join
SELECT e.first_name, e.last_name, d.department_name, p.project_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id
INNER JOIN projects p ON e.employee_id = p.employee_id;
```

LEFT JOIN

```
-- LEFT JOIN (all employees, even without departments)
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;
```

RIGHT JOIN

```
-- RIGHT JOIN (all departments, even without employees)
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;
```

FULL OUTER JOIN

```
-- FULL OUTER JOIN (all records from both tables)
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
FULL OUTER JOIN departments d ON e.department_id = d.department_id;
```

CROSS JOIN

```
-- CROSS JOIN (Cartesian product)
SELECT e.first_name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

Self Join

```
-- Self join (employees and their managers)
SELECT e.first_name AS employee, m.first_name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

Subqueries

Nested queries that provide powerful data retrieval and filtering capabilities.

What is a Subquery?

A **subquery** is a query nested inside another query. It can be used in **`SELECT`**, **`INSERT`**, **`UPDATE`**, or **`DELETE`** statements.

Types of Subqueries

1. **Single-row subquery**: Returns one row

2. **Multi-row subquery**: Returns multiple rows

3. **Correlated subquery**: References columns **`FROM`** outer query

Single-row Subquery

```
-- Find employees with salary higher than average
SELECT first_name, last_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- Find employee with highest salary
SELECT first_name, last_name, salary
FROM employees
```

```
WHERE salary = (SELECT MAX(salary) FROM employees);
```

Multi-row Subquery

```
-- Find employees in departments with more than 5 employees
SELECT first_name, last_name, department_id
FROM employees
WHERE department_id IN (
    SELECT department_id
    FROM employees
    GROUP BY department_id
    HAVING COUNT(*) > 5
);

-- Using ANY
SELECT first_name, last_name, salary
FROM employees
WHERE salary > ANY (
    SELECT salary
    FROM employees
    WHERE department_id = 10
);

-- Using ALL
SELECT first_name, last_name, salary
FROM employees
WHERE salary > ALL (
    SELECT salary
    FROM employees
    WHERE department_id = 10
);
```

Correlated Subquery

```
-- Find employees earning more than average in their department
SELECT first_name, last_name, salary, department_id
FROM employees e1
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
    WHERE e1.department_id = e2.department_id
);
```

EXISTS and NOT EXISTS

```
-- Find employees who have at least one project
SELECT first_name, last_name
FROM employees e
WHERE EXISTS (
    SELECT 1
    FROM projects p
    WHERE p.employee_id = e.employee_id
);

-- Find employees who have no projects
SELECT first_name, last_name
FROM employees e
WHERE NOT EXISTS (
    SELECT 1
    FROM projects p
    WHERE p.employee_id = e.employee_id
);
```

INSERT, UPDATE, DELETE

INSERT Statement

```
-- Insert single row
INSERT INTO employees (employee_id, first_name, last_name, email, salary, hire_da
VALUES (101, 'John', 'Doe', 'john.doe@company.com', 50000, '2023-01-15');

-- Insert multiple rows
INSERT INTO employees (employee_id, first_name, last_name, email, salary, hire_da
VALUES
    (102, 'Jane', 'Smith', 'jane.smith@company.com', 55000, '2023-02-01'),
    (103, 'Bob', 'Johnson', 'bob.johnson@company.com', 52000, '2023-02-15');

-- Insert from another table
INSERT INTO employees_backup
SELECT * FROM employees WHERE department_id = 10;

-- Insert with subquery
INSERT INTO high_earners (employee_id, first_name, last_name, salary)
SELECT employee_id, first_name, last_name, salary
FROM employees
```

```
WHERE salary > (SELECT AVG(salary) FROM employees);
```

UPDATE Statement

```
-- Update single column
UPDATE employees
SET salary = 55000
WHERE employee_id = 101;

-- Update multiple columns
UPDATE employees
SET salary = 60000, department_id = 20
WHERE employee_id = 101;

-- Update with subquery
UPDATE employees
SET salary = salary * 1.1
WHERE department_id = (
    SELECT department_id
    FROM departments
    WHERE department_name = 'Sales'
);

-- Update with JOIN
UPDATE employees e
SET salary = salary * 1.05
FROM departments d
WHERE e.department_id = d.department_id
AND d.department_name = 'IT';
```

DELETE Statement

```
-- Delete specific records
DELETE FROM employees
WHERE employee_id = 101;

-- Delete with condition
DELETE FROM employees
WHERE department_id = 30 AND salary < 40000;

-- Delete with subquery
DELETE FROM employees
WHERE department_id IN (
    SELECT department_id
    FROM departments
```

```
    WHERE department_name = 'Temp'
);

-- Delete all records (keep table structure)
DELETE FROM employees;
```

CREATE and ALTER

CREATE TABLE

```
-- Basic CREATE TABLE
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    department_id INT,
    salary DECIMAL(10,2),
    hire_date DATE,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);

-- CREATE TABLE with constraints
CREATE TABLE products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(8,2) CHECK (price > 0),
    category_id INT,
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_category (category_id)
);

-- CREATE TABLE from another table
CREATE TABLE employees_backup AS
SELECT * FROM employees;

-- CREATE TABLE with specific columns
CREATE TABLE employee_summary AS
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE department_id = 10;
```

ALTER TABLE

```
-- Add column
ALTER TABLE employees
ADD COLUMN phone VARCHAR(20);

-- Drop column
ALTER TABLE employees
DROP COLUMN phone;

-- Modify column
ALTER TABLE employees
MODIFY COLUMN salary DECIMAL(12,2);

-- Rename column
ALTER TABLE employees
RENAME COLUMN first_name TO fname;

-- Add constraint
ALTER TABLE employees
ADD CONSTRAINT fk_department
FOREIGN KEY (department_id) REFERENCES departments(department_id);

-- Drop constraint
ALTER TABLE employees
DROP CONSTRAINT fk_department;

-- Rename table
ALTER TABLE employees
RENAME TO staff;
```

DROP TABLE

```
-- Drop table
DROP TABLE employees_backup;

-- Drop table if exists
DROP TABLE IF EXISTS employees_backup;
```

Aggregate Functions

Built-in functions like COUNT, SUM, AVG for data analysis and calculations.

Common Aggregate Functions

```
-- COUNT: Count number of rows
SELECT COUNT(*) FROM employees;
SELECT COUNT(DISTINCT department_id) FROM employees;

-- SUM: Sum of values
SELECT SUM(salary) FROM employees;
SELECT SUM(salary) FROM employees WHERE department_id = 10;

-- AVG: Average of values
SELECT AVG(salary) FROM employees;
SELECT AVG(salary) FROM employees WHERE department_id = 10;

-- MIN: Minimum value
SELECT MIN(salary) FROM employees;
SELECT MIN(hire_date) FROM employees;

-- MAX: Maximum value
SELECT MAX(salary) FROM employees;
SELECT MAX(hire_date) FROM employees;
```

Aggregate Functions with GROUP BY

```
-- Count employees by department
SELECT department_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id;

-- Average salary by department
SELECT department_id, AVG(salary) as avg_salary
FROM employees
GROUP BY department_id;

-- Multiple aggregates
SELECT department_id,
       COUNT(*) as employee_count,
       AVG(salary) as avg_salary,
       MIN(salary) as min_salary,
       MAX(salary) as max_salary,
       SUM(salary) as total_salary
FROM employees
GROUP BY department_id;
```

Window Functions

Advanced functions for performing calculations across related rows.

What are Window Functions?

Window functions perform calculations across a set of rows related to the current row, without collapsing the rows into a single result.

ROW_NUMBER()

```
-- Add row numbers
SELECT employee_id, first_name, last_name, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) as row_num
FROM employees;

-- Row numbers within each department
SELECT employee_id, first_name, last_name, department_id, salary,
       ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) as dep
FROM employees;
```

RANK() and DENSE_RANK()

```
-- RANK (with gaps)
SELECT employee_id, first_name, last_name, salary,
       RANK() OVER (ORDER BY salary DESC) as salary_rank
FROM employees;

-- DENSE_RANK (no gaps)
SELECT employee_id, first_name, last_name, salary,
       DENSE_RANK() OVER (ORDER BY salary DESC) as salary_dense_rank
FROM employees;
```

LAG() and LEAD()

```
-- LAG (previous row value)
SELECT employee_id, first_name, salary,
       LAG(salary, 1) OVER (ORDER BY salary) as prev_salary
FROM employees;

-- LEAD (next row value)
SELECT employee_id, first_name, salary,
       LEAD(salary, 1) OVER (ORDER BY salary) as next_salary
FROM employees;
```

FIRST_VALUE() and LAST_VALUE()

```
-- FIRST_VALUE and LAST_VALUE
SELECT employee_id, first_name, salary,
       FIRST_VALUE(salary) OVER (ORDER BY salary DESC) as highest_salary,
       LAST_VALUE(salary) OVER (ORDER BY salary DESC
                                ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLL
FROM employees;
```

SUM() and AVG() as Window Functions

```
-- Running total
SELECT employee_id, first_name, salary,
       SUM(salary) OVER (ORDER BY employee_id) as running_total
FROM employees;

-- Moving average
SELECT employee_id, first_name, salary,
       AVG(salary) OVER (ORDER BY employee_id ROWS BETWEEN 2 PRECEDING AND CURREN
FROM employees;
```

String Functions

Functions for manipulating and processing text data in SQL.

Common String Functions

```
-- UPPER: Convert to uppercase
SELECT UPPER(first_name) FROM employees;

-- LOWER: Convert to lowercase
SELECT LOWER(first_name) FROM employees;

-- LENGTH: Get string length
SELECT first_name, LENGTH(first_name) as name_length FROM employees;

-- SUBSTRING: Extract substring
SELECT SUBSTRING(first_name, 1, 3) as first_three FROM employees;

-- CONCAT: Concatenate strings
SELECT CONCAT(first_name, ' ', last_name) as full_name FROM employees;

-- TRIM: Remove leading/trailing spaces
SELECT TRIM(first_name) FROM employees;

-- LTRIM and RTRIM: Remove left/right spaces
SELECT LTRIM(first_name), RTRIM(last_name) FROM employees;

-- REPLACE: Replace substring
SELECT REPLACE(email, '@company.com', '@newcompany.com') FROM employees;

-- LEFT and RIGHT: Extract from left/right
SELECT LEFT(first_name, 3), RIGHT(last_name, 3) FROM employees;

-- CHARINDEX/POSITION: Find position of substring
SELECT CHARINDEX('@', email) as at_position FROM employees;
```

String Functions Examples

```
-- Extract domain from email
SELECT email,
       SUBSTRING(email, CHARINDEX('@', email) + 1, LEN(email)) as domain
FROM employees;

-- Create initials
SELECT CONCAT(LEFT(first_name, 1), '.', LEFT(last_name, 1), '.') as initials
FROM employees;

-- Check if email is valid
SELECT email,
       CASE WHEN email LIKE '%@%.%' THEN 'Valid' ELSE 'Invalid' END as email_stat
FROM employees;
```

Date and Time Functions

Functions for working with date and time data types.

Current Date/Time Functions

```
-- Current date and time
SELECT GETDATE();        -- SQL Server
SELECT NOW();           -- MySQL
SELECT SYSDATE;         -- Oracle
SELECT CURRENT_TIMESTAMP; -- Standard SQL

-- Current date only
SELECT CAST(GETDATE() AS DATE);  -- SQL Server
SELECT CURDATE();               -- MySQL
SELECT CURRENT_DATE;            -- Standard SQL

-- Current time only
SELECT CAST(GETDATE() AS TIME);  -- SQL Server
SELECT CURTIME();               -- MySQL
SELECT CURRENT_TIME;            -- Standard SQL
```

Date Arithmetic

```
-- Add days
SELECT DATEADD(DAY, 30, hire_date) as thirty_days_later FROM employees;

-- Subtract days
SELECT DATEADD(DAY, -30, hire_date) as thirty_days_earlier FROM employees;

-- Add months
SELECT DATEADD(MONTH, 6, hire_date) as six_months_later FROM employees;

-- Add years
SELECT DATEADD(YEAR, 1, hire_date) as one_year_later FROM employees;
```

Date Difference

```
-- Difference in days
SELECT first_name, hire_date,
       DATEDIFF(DAY, hire_date, GETDATE()) as days_employed
FROM employees;

-- Difference in months
SELECT first_name, hire_date,
       DATEDIFF(MONTH, hire_date, GETDATE()) as months_employed
FROM employees;

-- Difference in years
SELECT first_name, hire_date,
       DATEDIFF(YEAR, hire_date, GETDATE()) as years_employed
FROM employees;
```

Extract Date Parts

```
-- Extract year, month, day
SELECT hire_date,
       YEAR(hire_date) as hire_year,
       MONTH(hire_date) as hire_month,
       DAY(hire_date) as hire_day
FROM employees;

-- Extract weekday
SELECT hire_date,
       DATENAME(WEEKDAY, hire_date) as hire_weekday
FROM employees;

-- Extract quarter
SELECT hire_date,
       DATEPART(QUARTER, hire_date) as hire_quarter
FROM employees;
```

Constraints

Rules and restrictions to ensure data integrity and consistency.

Types of Constraints

1. **NOT NULL**: Ensures column cannot have NULL values

2. **UNIQUE**: Ensures all values in column are unique

3. **PRIMARY KEY**: Combination of NOT NULL and UNIQUE

4. **FOREIGN KEY**: Links to primary key of another **TABLE**

5. **CHECK**: Ensures values meet specific condition

6. **DEFAULT**: Sets default value for column

Constraint Examples

```
-- Create table with constraints
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    department_id INT,
    salary DECIMAL(10,2) CHECK (salary > 0),
    hire_date DATE DEFAULT CURRENT_DATE,
    status VARCHAR(20) DEFAULT 'Active' CHECK (status IN ('Active', 'Inactive')),
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);

-- Add constraints to existing table
ALTER TABLE employees
ADD CONSTRAINT chk_salary CHECK (salary >= 0);

ALTER TABLE employees
ADD CONSTRAINT fk_department
FOREIGN KEY (department_id) REFERENCES departments(department_id);

-- Drop constraints
ALTER TABLE employees
DROP CONSTRAINT chk_salary;
```

Indexes

Database structures that improve query performance and data retrieval speed.

What are Indexes?

Indexes are **DATABASE** objects that improve the speed of data retrieval operations on a **TABLE**.

Types of Indexes

1. **Clustered INDEX**: Determines physical order of data

2. **Non-Clustered INDEX**: Separate structure pointing to data

3. **Unique INDEX**: Ensures uniqueness and improves performance

4. **Composite INDEX**: **INDEX** on multiple columns

Creating Indexes

```
-- Create simple index
CREATE INDEX idx_employee_lastname ON employees(last_name);

-- Create composite index
CREATE INDEX idx_employee_dept_salary ON employees(department_id, salary);

-- Create unique index
CREATE UNIQUE INDEX idx_employee_email ON employees(email);

-- Create clustered index
CREATE CLUSTERED INDEX idx_employee_id ON employees(employee_id);
```

Managing Indexes

```
-- View indexes
```

```
SELECT * FROM sys.indexes WHERE object_id = OBJECT_ID('employees');

-- Drop index
DROP INDEX idx_employee_lastname ON employees;

-- Rebuild index
ALTER INDEX idx_employee_lastname ON employees REBUILD;
```

Views

Virtual tables that provide customized data presentation and security.

What are Views?

A **VIEW** is a virtual **TABLE** based on the result of a SQL statement. It contains rows and columns like a real **TABLE**.

Creating Views

```
-- Simple view
CREATE VIEW employee_details AS
SELECT e.employee_id, e.first_name, e.last_name, d.department_name, e.salary
FROM employees e
JOIN departments d ON e.department_id = d.department_id;

-- View with calculations
CREATE VIEW employee_summary AS
SELECT department_id,
       COUNT(*) as employee_count,
       AVG(salary) as avg_salary,
       MAX(salary) as max_salary,
       MIN(salary) as min_salary
FROM employees
GROUP BY department_id;

-- View with filtering
CREATE VIEW high_earners AS
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary > 75000;
```

Using Views

```
-- Query from view
SELECT * FROM employee_details;

-- Filter view results
SELECT * FROM employee_details WHERE department_name = 'Sales';

-- Join views
SELECT ed.first_name, ed.last_name, es.avg_salary
FROM employee_details ed
JOIN employee_summary es ON ed.department_id = es.department_id;
```

Managing Views

```
-- Modify view
ALTER VIEW employee_details AS
SELECT e.employee_id, e.first_name, e.last_name, d.department_name, e.salary, e.h
FROM employees e
JOIN departments d ON e.department_id = d.department_id;

-- Drop view
DROP VIEW employee_details;
```

Stored Procedures

Reusable database programs that encapsulate complex operations.

What are Stored Procedures?

A **stored PROCEDURE** is a prepared SQL code that can be saved and reused.

Creating Stored Procedures

```
-- Simple stored procedure
CREATE PROCEDURE GetEmployeeCount
AS
BEGIN
    SELECT COUNT(*) as total_employees FROM employees;
END;

-- Stored procedure with parameters
CREATE PROCEDURE GetEmployeesByDepartment
    @DepartmentId INT
AS
BEGIN
    SELECT employee_id, first_name, last_name, salary
    FROM employees
    WHERE department_id = @DepartmentId;
END;

-- Stored procedure with output parameter
CREATE PROCEDURE GetEmployeeCountByDept
    @DepartmentId INT,
    @EmployeeCount INT OUTPUT
AS
BEGIN
    SELECT @EmployeeCount = COUNT(*)
    FROM employees
    WHERE department_id = @DepartmentId;
END;
```

Executing Stored Procedures

```
-- Execute simple procedure
EXEC GetEmployeeCount;

-- Execute with parameters
EXEC GetEmployeesByDepartment @DepartmentId = 10;

-- Execute with output parameter
DECLARE @Count INT;
EXEC GetEmployeeCountByDept @DepartmentId = 10, @EmployeeCount = @Count OUTPUT;
SELECT @Count as EmployeeCount;
```

Transactions

Mechanisms for ensuring data consistency and handling concurrent operations.

What are Transactions?

A **transaction** is a sequence of SQL operations treated as a single unit of work.

ACID Properties

1. **Atomicity**: All operations succeed or all fail

2. **Consistency**: **DATABASE** remains in valid state

3. **Isolation**: Concurrent transactions don't interfere

4. **Durability**: Committed changes persist

Transaction Control

```
-- Basic transaction
BEGIN TRANSACTION;
    INSERT INTO employees VALUES (104, 'Alice', 'Wilson', 'alice@company.com', 10
    UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;
COMMIT;

-- Transaction with rollback
BEGIN TRANSACTION;
    DELETE FROM employees WHERE employee_id = 104;
    -- Check if we accidentally deleted too many records
    IF @@ROWCOUNT > 1
    BEGIN
        ROLLBACK;
        PRINT 'Transaction rolled back - too many records affected';
    END
    ELSE
    BEGIN
        COMMIT;
        PRINT 'Transaction committed successfully';
    END;
```

```
-- Using savepoints
BEGIN TRANSACTION;
    INSERT INTO employees VALUES (105, 'Bob', 'Brown', 'bob@company.com', 20, 550
    SAVE TRANSACTION SavePoint1;

    UPDATE employees SET salary = 70000 WHERE employee_id = 105;

    -- Something went wrong, rollback to savepoint
    ROLLBACK TRANSACTION SavePoint1;

    -- Continue with original transaction
    UPDATE employees SET salary = 58000 WHERE employee_id = 105;
COMMIT;
```

Advanced SQL Concepts

Complex topics including CTEs, recursive queries, and advanced optimization.

Common Table Expressions (CTEs)

```
-- Simple CTE
WITH DepartmentStats AS (
    SELECT department_id,
           COUNT(*) as employee_count,
           AVG(salary) as avg_salary
    FROM employees
    GROUP BY department_id
)
SELECT d.department_name, ds.employee_count, ds.avg_salary
FROM DepartmentStats ds
JOIN departments d ON ds.department_id = d.department_id;

-- Recursive CTE (Employee hierarchy)
WITH EmployeeHierarchy AS (
    -- Anchor member (top-level managers)
    SELECT employee_id, first_name, last_name, manager_id, 0 as level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive member
    SELECT e.employee_id, e.first_name, e.last_name, e.manager_id, eh.level + 1
```

```
    FROM employees e
    JOIN EmployeeHierarchy eh ON e.manager_id = eh.employee_id
)
SELECT * FROM EmployeeHierarchy ORDER BY level, employee_id;
```

PIVOT and UNPIVOT

```
-- PIVOT example
SELECT *
FROM (
    SELECT department_name, YEAR(hire_date) as hire_year
    FROM employees e
    JOIN departments d ON e.department_id = d.department_id
) as SourceTable
PIVOT (
    COUNT(hire_year)
    FOR hire_year IN ([2020], [2021], [2022], [2023])
) as PivotTable;

-- UNPIVOT example
SELECT department_name, hire_year, employee_count
FROM (
    SELECT department_name, [2020], [2021], [2022], [2023]
    FROM PivotTable
) as SourceTable
UNPIVOT (
    employee_count FOR hire_year IN ([2020], [2021], [2022], [2023])
) as UnpivotTable;
```

CASE Statements

```
-- Simple CASE
SELECT first_name, last_name, salary,
       CASE
           WHEN salary < 40000 THEN 'Low'
           WHEN salary BETWEEN 40000 AND 70000 THEN 'Medium'
           ELSE 'High'
       END as salary_grade
FROM employees;

-- CASE with multiple conditions
SELECT first_name, last_name, department_id, salary,
       CASE
           WHEN department_id = 10 AND salary > 60000 THEN 'Senior IT'
           WHEN department_id = 10 THEN 'IT'
```

```
            WHEN department_id = 20 AND salary > 50000 THEN 'Senior Sales'
            WHEN department_id = 20 THEN 'Sales'
            ELSE 'Other'
        END as employee_category
FROM employees;
```

Performance Optimization

Techniques for improving query performance and database efficiency.

Query Optimization Tips

1. **Use indexes appropriately**

2. **Avoid `SELECT` *** - specify needed columns

3. **Use `WHERE` clauses to filter early**

4. **Use JOINs instead of subqueries when possible**

5. **Avoid functions in `WHERE` clauses**

6. **Use LIMIT/TOP for large datasets**

Execution Plan Analysis

```
-- View execution plan (SQL Server)
SET SHOWPLAN_ALL ON;
SELECT * FROM employees WHERE department_id = 10;
SET SHOWPLAN_ALL OFF;

-- Include actual execution plan
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
SELECT * FROM employees WHERE department_id = 10;
SET STATISTICS IO OFF;
SET STATISTICS TIME OFF;
```

Optimization Examples

```
-- Bad: Function in WHERE clause
SELECT * FROM employees WHERE YEAR(hire_date) = 2023;

-- Good: Range condition
SELECT * FROM employees WHERE hire_date >= '2023-01-01' AND hire_date < '2024-01-

-- Bad: SELECT *
SELECT * FROM employees WHERE department_id = 10;

-- Good: Specific columns
SELECT employee_id, first_name, last_name, salary
FROM employees WHERE department_id = 10;

-- Bad: Correlated subquery
SELECT e1.first_name, e1.last_name
FROM employees e1
WHERE e1.salary > (SELECT AVG(e2.salary) FROM employees e2 WHERE e2.department_id

-- Good: Window function
SELECT first_name, last_name
FROM (
    SELECT first_name, last_name, salary,
           AVG(salary) OVER (PARTITION BY department_id) as dept_avg
    FROM employees
) t
WHERE salary > dept_avg;
```

Common Interview Questions

Frequently asked SQL questions and their solutions for technical interviews.

1. Find Second Highest Salary

```
-- Method 1: Using window function
SELECT salary
FROM (
    SELECT salary, ROW_NUMBER() OVER (ORDER BY salary DESC) as rn
```

```
    FROM employees
) t
WHERE rn = 2;

-- Method 2: Using subquery
SELECT MAX(salary) as second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);

-- Method 3: Using LIMIT/TOP
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;  -- MySQL
-- OR
SELECT TOP 1 salary
FROM (
    SELECT DISTINCT TOP 2 salary
    FROM employees
    ORDER BY salary DESC
) t
ORDER BY salary ASC;  -- SQL Server
```

2. Find Nth Highest Salary

```
-- Using window function
SELECT salary
FROM (
    SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) as rank
    FROM employees
) t
WHERE rank = N;

-- Using subquery (for 3rd highest)
SELECT salary
FROM employees e1
WHERE 3 = (
    SELECT COUNT(DISTINCT salary)
    FROM employees e2
    WHERE e2.salary >= e1.salary
);
```

3. Find Employees with Same Salary

```
SELECT e1.first_name, e1.last_name, e1.salary
```

```
FROM employees e1
JOIN employees e2 ON e1.salary = e2.salary AND e1.employee_id != e2.employee_id;

-- Or using window function
SELECT first_name, last_name, salary
FROM (
    SELECT first_name, last_name, salary,
           COUNT(*) OVER (PARTITION BY salary) as salary_count
    FROM employees
) t
WHERE salary_count > 1;
```

4. Find Employees with No Manager

```
SELECT first_name, last_name
FROM employees
WHERE manager_id IS NULL;
```

5. Find Department with Highest Average Salary

```
SELECT d.department_name, AVG(e.salary) as avg_salary
FROM employees e
JOIN departments d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
ORDER BY avg_salary DESC
LIMIT 1;
```

6. Find Employees Hired in Last 30 Days

```
SELECT first_name, last_name, hire_date
FROM employees
WHERE hire_date >= DATEADD(DAY, -30, GETDATE());  -- SQL Server
-- OR
WHERE hire_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY);  -- MySQL
```

7. Find Duplicate Records

```
-- Find duplicate emails
```

```sql
SELECT email, COUNT(*) as count
FROM employees
GROUP BY email
HAVING COUNT(*) > 1;

-- Find all duplicate records
SELECT *
FROM employees
WHERE email IN (
    SELECT email
    FROM employees
    GROUP BY email
    HAVING COUNT(*) > 1
);
```

8. Delete Duplicate Records

```sql
-- Using window function
WITH DuplicateRecords AS (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY email ORDER BY employee_id) as rn
    FROM employees
)
DELETE FROM DuplicateRecords
WHERE rn > 1;
```

9. Find Running Total

```sql
SELECT employee_id, first_name, salary,
       SUM(salary) OVER (ORDER BY employee_id) as running_total
FROM employees;
```

10. Find Top 3 Employees by Salary in Each Department

```sql
SELECT department_id, first_name, last_name, salary
FROM (
    SELECT department_id, first_name, last_name, salary,
           ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) as
    FROM employees
) t
WHERE rn <= 3;
```

Practice Problems

Easy Level

1. **List all employees with their department names**
2. **Find total number of employees in each department**
3. **Find employees with salary greater than 50000**
4. **List employees hired in 2023**
5. **Find the average salary of all employees**

Medium Level

1. **Find employees earning more than their department average**
2. **List departments with more than 5 employees**
3. **Find employees who have been with company for more than 2 years**
4. **Calculate the percentage of total salary each employee represents**
5. **Find employees with same first name**

Hard Level

1. **Find the 3rd highest salary without using TOP/LIMIT**
2. **`CREATE` a hierarchical query showing employee-manager relationships**
3. **Find departments `WHERE` average salary is above company average**

4. **Calculate running totals and moving averages**

5. **Find employees whose salary is above 80th percentile**

Solutions to Practice Problems

Easy Level Solutions

```
-- 1. List all employees with their department names
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;

-- 2. Find total number of employees in each department
SELECT d.department_name, COUNT(*) as employee_count
FROM employees e
JOIN departments d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name;

-- 3. Find employees with salary greater than 50000
SELECT first_name, last_name, salary
FROM employees
WHERE salary > 50000;

-- 4. List employees hired in 2023
SELECT first_name, last_name, hire_date
FROM employees
WHERE YEAR(hire_date) = 2023;

-- 5. Find the average salary of all employees
SELECT AVG(salary) as average_salary
FROM employees;
```

Medium Level Solutions

```
-- 1. Find employees earning more than their department average
SELECT e.first_name, e.last_name, e.salary, e.department_id
FROM employees e
WHERE e.salary > (
    SELECT AVG(e2.salary)
    FROM employees e2
    WHERE e2.department_id = e.department_id
);
```

```sql
-- 2. List departments with more than 5 employees
SELECT d.department_name, COUNT(*) as employee_count
FROM employees e
JOIN departments d ON e.department_id = d.department_id
GROUP BY d.department_id, d.department_name
HAVING COUNT(*) > 5;

-- 3. Find employees who have been with company for more than 2 years
SELECT first_name, last_name, hire_date,
       DATEDIFF(YEAR, hire_date, GETDATE()) as years_employed
FROM employees
WHERE DATEDIFF(YEAR, hire_date, GETDATE()) > 2;

-- 4. Calculate the percentage of total salary each employee represents
SELECT first_name, last_name, salary,
       (salary * 100.0 / (SELECT SUM(salary) FROM employees)) as salary_percentag
FROM employees;

-- 5. Find employees with same first name
SELECT first_name, COUNT(*) as count
FROM employees
GROUP BY first_name
HAVING COUNT(*) > 1;
```

SQL Best Practices

1. Naming Conventions

- Use lowercase with underscores for **TABLE** and column names

- Use descriptive names

- Prefix foreign keys with **TABLE** name

- Use singular nouns for **TABLE** names

2. Query Writing

- Use proper indentation and formatting

- Use **TABLE** aliases for readability

- Always specify column names in **INSERT** statements

- Use EXISTS instead of IN for better performance

3. Performance Tips

- **CREATE** indexes on frequently queried columns

- Use appropriate data types

- Avoid **SELECT** * in production code

- Use LIMIT/TOP for large result sets

- Use connection pooling

4. Security

- Use parameterized queries to prevent SQL injection

- Grant minimum necessary permissions

- Use stored procedures for complex operations

- Regularly backup your **DATABASE**

Conclusion

This SQL guide covers everything **FROM** basic concepts to advanced techniques needed for technical interviews. Practice these concepts regularly and work on real-world problems to become proficient in SQL.

Key Takeaways

1. **Master the basics**: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `JOIN`

2. **Understand data relationships**: Primary keys, foreign keys, normalization

3. **Practice complex queries**: Subqueries, window functions, CTEs

4. **Learn optimization**: Indexes, execution plans, performance tuning

5. **Solve problems**: Work through common interview questions

Next Steps

1. Practice on online platforms (LeetCode, HackerRank, SQLBolt)

2. Work with real databases

3. Learn `DATABASE`-specific features (MySQL, PostgreSQL, SQL Server)

4. Study `DATABASE` design and normalization

5. Explore advanced topics (triggers, functions, data warehousing)

Good luck with your SQL interviews! ■