# SQL Commands: Complete Reference Guide

## DDL, DML, DCL, and DQL with Use Cases & Examples

---

## TABLE OF CONTENTS

---

## INTRODUCTION

SQL (Structured Query Language) commands are categorized into five main groups based on their function:

| Category | Purpose | Commands |
|---|---|---|
| **DDL** | Define database structure | CREATE, ALTER, DROP, TRUNCATE |
| **DML** | Manipulate data | INSERT, UPDATE, DELETE |
| **DQL** | Query/retrieve data | SELECT |
| **DCL** | Control access permissions | GRANT, REVOKE |
| **TCL** | Manage transactions | COMMIT, ROLLBACK, SAVEPOINT |

---

## DDL - DATA DEFINITION LANGUAGE

DDL commands define, modify, and delete database structures (tables, schemas, indexes, etc.). Changes are **auto-committed**.

### 1. CREATE

**Purpose**: Create new database objects (tables, databases, indexes, views).

#### CREATE DATABASE

```
-- Create a new database
CREATE DATABASE company_db;

-- Create database with specific character set
CREATE DATABASE company_db CHARACTER SET utf8mb4;

-- Check if database exists before creating (avoid error)
CREATE DATABASE IF NOT EXISTS company_db;
```

**Use Cases**:

- Initialize new application database

- Set up separate databases for dev, test, staging, production
- Create isolated databases for different projects

---

### CREATE TABLE

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

-- Create table with multiple constraints
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    roll_number VARCHAR(20) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    gpa DECIMAL(3, 2) CHECK (gpa >= 0 AND gpa <= 4.0),
    enrollment_date DATE DEFAULT CURRENT_DATE,
    age INT CHECK (age >= 18)
);

-- Create temporary table (auto-deleted after session)
CREATE TEMPORARY TABLE temp_sales AS
SELECT product_id, SUM(amount) as total
FROM orders
GROUP BY product_id;
```

**Use Cases**:

- Set up database schema for new application
- Define relationships between entities (foreign keys)
- Establish data validation rules (constraints)
- Create temporary working tables for ETL processes

**Key Constraints**:

- `PRIMARY KEY`: Unique identifier, no NULLs
- `FOREIGN KEY`: References another table's primary key
- `UNIQUE`: No duplicate values allowed
- `NOT NULL`: Value must be provided
- `CHECK`: Value must meet condition
- `DEFAULT`: Default value if not specified

---

### CREATE INDEX

```
-- Simple index on single column
CREATE INDEX idx_email ON employees(email);

-- Index on multiple columns
CREATE INDEX idx_name ON employees(first_name, last_name);

-- Unique index (prevents duplicates)
CREATE UNIQUE INDEX idx_phone ON customers(phone_number);

-- Index with condition (partial index)
```

```
CREATE INDEX idx_active_orders ON orders(customer_id)
WHERE status = 'active';
```

**Use Cases**:

- Speed up SELECT queries on frequently searched columns
- Enforce uniqueness at column level
- Optimize WHERE, JOIN, and ORDER BY clauses
- Filter large tables (partial indexes)

---

### CREATE VIEW

```
-- Basic view
CREATE VIEW employee_summary AS
SELECT
    employee_id,
    CONCAT(first_name, ' ', last_name) AS full_name,
    email,
    salary
FROM employees
WHERE status = 'active';

-- Use the view
SELECT * FROM employee_summary WHERE salary > 50000;

-- Create view with aggregation
CREATE VIEW sales_by_region AS
SELECT
    region,
    COUNT(order_id) AS total_orders,
    SUM(amount) AS total_sales,
    AVG(amount) AS avg_order
FROM orders
GROUP BY region;
```

**Use Cases**:

- Simplify complex queries for end users
- Provide consistent data aggregation
- Hide sensitive columns (security)
- Create virtual tables without storing redundant data

---

## 2. ALTER

**Purpose**: Modify existing database structures.

### ALTER TABLE - Add Column

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

-- Add column with constraints
ALTER TABLE employees
ADD COLUMN manager_id INT,
ADD CONSTRAINT fk_manager
FOREIGN KEY (manager_id) REFERENCES employees(employee_id);

-- Add multiple columns
ALTER TABLE employees
ADD (
    birth_date DATE,
    ssn VARCHAR(20) UNIQUE,
    department VARCHAR(50)
);
```

**Use Cases**:

- Add new attributes as business requirements evolve
- Add calculated or derived columns
- Extend table schema post-production

---

### ALTER TABLE - Modify Column

```
-- Change data type
ALTER TABLE employees
MODIFY COLUMN salary DECIMAL(12, 2);

-- Change column constraint
ALTER TABLE employees
MODIFY COLUMN email VARCHAR(150) NOT NULL;

-- Rename column
ALTER TABLE employees
RENAME COLUMN phone_number TO contact_phone;

-- Set default value
ALTER TABLE employees
ALTER COLUMN hire_date SET DEFAULT CURRENT_DATE;
```

**Use Cases**:

- Adjust data types for precision/performance
- Update constraints after deployment
- Accommodate new business logic
- Rename columns for clarity

---

### ALTER TABLE - Drop Column

```
-- Remove single column
ALTER TABLE employees
DROP COLUMN phone_number;

-- Remove multiple columns
ALTER TABLE employees
DROP COLUMN birth_date,
DROP COLUMN ssn;

-- Drop constraint
ALTER TABLE employees
DROP CONSTRAINT fk_manager;
```

**Use Cases**:

- Remove obsolete columns
- Clean up schema after features removed
- Reduce storage footprint

---

### ALTER TABLE - Rename Table

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

-- In MySQL
RENAME TABLE employees TO staff;
```

---

## 3. DROP

**Purpose**: Delete database objects permanently (non-recoverable from backups).

```
-- Drop table (removes structure and all data)
DROP TABLE employees;

-- Drop table only if it exists
DROP TABLE IF EXISTS temp_employees;

-- Drop database
DROP DATABASE company_db;

-- Drop index
DROP INDEX idx_email ON employees;

-- Drop view
DROP VIEW employee_summary;

-- Drop with cascade (remove dependent objects)
DROP TABLE employees CASCADE;
```

⚠ **WARNING**:

- DROP is **irreversible** without backups
- All data and structure are permanently deleted
- Check twice before executing!

**Use Cases**:

- Remove obsolete tables from schema
- Clean up test databases
- Retire unused indexes
- Delete temporary objects

---

## 4. TRUNCATE

**Purpose**: Remove all rows from table but keep structure (faster than DELETE).

```
-- Remove all data from table
TRUNCATE TABLE employees;

-- Truncate resets identity/auto-increment
TRUNCATE TABLE orders;

-- Some databases require:
TRUNCATE TABLE employees CASCADE;
```

**TRUNCATE vs DELETE**:

| Aspect | TRUNCATE | DELETE |
|---|---|---|
| **Speed** | Faster (deallocates pages) | Slower (row-by-row) |
| **Rollback** | Yes (in transaction) | Yes (in transaction) |

| Aspect | TRUNCATE | DELETE |
|---|---|---|
| **Identity** | Resets to seed | Continues sequence |
| **Triggers** | Not fired | Triggers fired |
| **WHERE** | Not supported | Supports WHERE clause |
| **Space** | Deallocates | Keeps space allocated |

**Use Cases**:

- Clear test data before new test run
- Archive and reset operational tables
- Quickly empty staging tables
- Fast data cleanup (bulk remove)

---

# DML - DATA MANIPULATION LANGUAGE

DML commands modify data within tables. Changes require explicit COMMIT in transactions.

## 1. INSERT

**Purpose**: Add new rows to table.

### Basic INSERT

```
-- Insert single row with all columns
INSERT INTO employees
VALUES (1, 'John', 'Doe', 'john@company.com', 75000, '2024-01-15', 10);

-- Insert with specific columns (other columns get NULL or DEFAULT)
INSERT INTO employees (employee_id, first_name, last_name, email, salary)
VALUES (2, 'Jane', 'Smith', 'jane@company.com', 80000);

-- Insert multiple rows at once
INSERT INTO employees (employee_id, first_name, last_name, email, salary)
VALUES
    (3, 'Bob', 'Johnson', 'bob@company.com', 65000),
    (4, 'Alice', 'Williams', 'alice@company.com', 72000),
    (5, 'Charlie', 'Brown', 'charlie@company.com', 68000);
```

### INSERT FROM SELECT

```
-- Insert data from another table
INSERT INTO archived_employees
SELECT * FROM employees WHERE status = 'inactive';

-- Insert with transformation
INSERT INTO employee_salary_history (employee_id, salary, date)
SELECT employee_id, salary, CURRENT_DATE
FROM employees;

-- Insert with filtering
INSERT INTO high_earners (employee_id, name, salary)
SELECT employee_id, CONCAT(first_name, ' ', last_name), salary
FROM employees
WHERE salary > 100000;
```

**Use Cases**:

- Add new customer orders

- Record new employee information
- Bulk import from external sources
- Archive historical data
- Create backups/snapshots

---

## 2. UPDATE

**Purpose**: Modify existing data in table.

### Basic UPDATE

```
-- Update single row
UPDATE employees
SET salary = 85000
WHERE employee_id = 1;

-- Update multiple columns
UPDATE employees
SET salary = 90000, department_id = 5
WHERE employee_id = 2;

-- Update multiple rows with condition
UPDATE employees
SET salary = salary * 1.1  -- 10% raise
WHERE hire_date < '2020-01-01';  -- employees hired before 2020

-- Update all rows (no WHERE = all affected!)
UPDATE products
SET price = price * 1.05;  -- 5% price increase
```

### UPDATE with JOIN

```
-- Update based on related table data
UPDATE employees e
JOIN departments d ON e.department_id = d.department_id
SET e.salary = e.salary * 1.15
WHERE d.dept_name = 'Sales';

-- Update using subquery
UPDATE employees
SET manager_id = (SELECT employee_id FROM employees WHERE last_name = 'Smith')
WHERE department_id = 5;
```

⚠ **Safety Tip**:

- Always use WHERE clause
- Test SELECT first: `SELECT * FROM employees WHERE ...`
- Run in transaction: `BEGIN; UPDATE ...; ROLLBACK;` to test

**Use Cases**:

- Apply salary increases
- Change order status (pending → shipped)
- Correct data entry errors
- Update prices after inflation adjustment
- Change customer contact information

---

## 3. DELETE

**Purpose**: Remove specific rows from table.

```
-- Delete single row
DELETE FROM employees
WHERE employee_id = 1;

-- Delete multiple rows
DELETE FROM orders
WHERE status = 'cancelled' AND order_date < '2023-01-01';

-- Delete with join condition
DELETE e FROM employees e
WHERE e.employee_id IN (
    SELECT manager_id FROM departments
    WHERE budget = 0
);

-- Delete all rows (slower than TRUNCATE)
DELETE FROM temp_table;
```

**DELETE vs TRUNCATE vs DROP**:

| Operation | Type | Data | Structure | Rollback | Speed |
|---|---|---|---|---|---|
| **DELETE** | DML | Removed | Kept | Yes | Slow |
| **TRUNCATE** | DDL | Removed | Kept | Yes* | Fast |
| **DROP** | DDL | Removed | Removed | No | Fast |

*Rollback only if in transaction

**Use Cases**:

- Remove canceled orders
- Delete inactive customer records
- Clean up duplicate data
- Prune old log entries
- Remove test records

---

# DQL - DATA QUERY LANGUAGE

DQL commands retrieve and display data without modifying it.

## SELECT - The Foundation

**Purpose**: Query and retrieve data from one or more tables.

### Basic SELECT

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

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

-- Select with aliases
SELECT
    employee_id AS emp_id,
    CONCAT(first_name, ' ', last_name) AS full_name,
    salary AS annual_salary
FROM employees;
```

---

### SELECT with WHERE

```
-- Filter by condition
SELECT * FROM employees
WHERE salary > 75000;

-- Multiple conditions
SELECT * FROM orders
WHERE status = 'shipped' AND order_date >= '2024-01-01';

-- OR condition
SELECT * FROM customers
WHERE country = 'USA' OR country = 'Canada';

-- IN clause
SELECT * FROM employees
WHERE department_id IN (1, 3, 5);

-- BETWEEN
SELECT * FROM orders
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31';

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

-- IS NULL
SELECT * FROM employees
WHERE manager_id IS NULL;
```

---

### SELECT with JOIN

**INNER JOIN**: Returns matching rows from both tables

```
SELECT
    e.employee_id,
    e.first_name,
    e.last_name,
    d.dept_name,
    d.location
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;
```

**LEFT JOIN**: All from left + matching from right

```
SELECT
    e.first_name,
    e.last_name,
    d.dept_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;
-- Shows all employees, even those with no department assigned
```

**RIGHT JOIN**: All from right + matching from left

```
SELECT
    e.first_name,
    d.dept_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;
-- Shows all departments, even those with no employees
```

**FULL OUTER JOIN**: All rows from both tables

```
SELECT
    e.first_name,
    d.dept_name
FROM employees e
FULL OUTER JOIN departments d ON e.department_id = d.department_id;
-- Shows all employees and all departments
```

**CROSS JOIN**: Cartesian product (all combinations)

```
SELECT * FROM colors
CROSS JOIN sizes;
-- If 5 colors × 3 sizes = 15 combinations
```

**SELF JOIN**: Join table to itself

```
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;
```

---

### SELECT with GROUP BY & HAVING

```
-- Count employees per department
SELECT
    department_id,
    COUNT(*) AS emp_count,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;

-- Group with HAVING (filter groups)
SELECT
    department_id,
    COUNT(*) AS emp_count,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5 AND AVG(salary) > 70000;
-- Shows only departments with 5+ employees AND avg salary > 70k

-- Multiple aggregate functions
SELECT
    product_category,
    COUNT(order_id) AS total_orders,
    SUM(amount) AS total_revenue,
    AVG(amount) AS avg_order,
    MIN(amount) AS lowest_order,
    MAX(amount) AS highest_order
FROM orders
GROUP BY product_category
ORDER BY total_revenue DESC;
```

**Aggregate Functions**:

- `COUNT()`: Number of rows
- `SUM()`: Total of numeric column
- `AVG()`: Average of numeric column
- `MIN()`: Lowest value
- `MAX()`: Highest value
- `COUNT(DISTINCT)`: Unique values

---

### SELECT with ORDER BY

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

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

-- Multiple columns
SELECT * FROM orders
ORDER BY customer_id ASC, order_date DESC;
-- Sort by customer first, then by date within each customer

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

---

### SELECT with LIMIT/OFFSET

```
-- Get first 5 rows
SELECT * FROM employees
LIMIT 5;

-- Pagination: skip 10, get next 5
SELECT * FROM employees
LIMIT 5 OFFSET 10;

-- In MySQL, alternative syntax
SELECT * FROM employees
LIMIT 10, 5;  -- Skip 10, get 5

-- Top N with ORDER BY
SELECT * FROM employees
ORDER BY salary DESC
LIMIT 10;  -- Top 10 highest paid
```

---

### SELECT with DISTINCT

```
-- Get unique values
SELECT DISTINCT department_id FROM employees;

-- Unique combinations
SELECT DISTINCT country, city FROM customers;

-- Count distinct
SELECT COUNT(DISTINCT department_id) FROM employees;
```

---

### Advanced: Window Functions

```
-- Row numbering per group
SELECT
    employee_id,
    salary,
    department_id,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank
FROM employees;
-- Shows top earners per department

-- Running total
SELECT
```

```
    order_id,
    amount,
    SUM(amount) OVER (ORDER BY order_date) AS running_total
FROM orders;

-- Percentile ranking
SELECT
    employee_id,
    salary,
    PERCENT_RANK() OVER (ORDER BY salary) AS salary_percentile
FROM employees;
```

---

### SELECT with CTE (Common Table Expression)

```
-- Single CTE
WITH high_earners AS (
    SELECT employee_id, first_name, salary
    FROM employees
    WHERE salary > 100000
)
SELECT * FROM high_earners WHERE department_id = 5;

-- Multiple CTEs
WITH
recent_orders AS (
    SELECT customer_id, SUM(amount) as total
    FROM orders
    WHERE order_date >= DATEADD(MONTH, -3, CURRENT_DATE)
    GROUP BY customer_id
),
loyal_customers AS (
    SELECT customer_id, COUNT(*) as order_count
    FROM orders
    GROUP BY customer_id
    HAVING COUNT(*) > 10
)
SELECT
    c.customer_id,
    ro.total as recent_spending,
    lc.order_count as total_orders
FROM loyal_customers lc
JOIN recent_orders ro ON lc.customer_id = ro.customer_id;
```

---

### SELECT with UNION

```
-- Combine results from multiple queries
SELECT first_name, last_name FROM employees
UNION
SELECT first_name, last_name FROM contractors;
-- Returns unique combinations only

-- UNION ALL includes duplicates
SELECT first_name, last_name FROM employees
UNION ALL
SELECT first_name, last_name FROM contractors;
```

---

### SELECT with CASE

```
-- Conditional column
SELECT
    employee_id,
    first_name,
    salary,
    CASE
        WHEN salary >= 100000 THEN 'Senior'
        WHEN salary >= 75000 THEN 'Mid-level'
        WHEN salary >= 50000 THEN 'Junior'
        ELSE 'Intern'
    END AS salary_bracket
FROM employees;

-- Simple CASE
SELECT
    order_id,
    status,
    CASE status
        WHEN 'shipped' THEN 'Ready'
        WHEN 'pending' THEN 'Processing'
        WHEN 'cancelled' THEN 'Cancelled'
    END AS status_label
FROM orders;
```

---

# DCL - DATA CONTROL LANGUAGE

DCL commands manage user permissions and access control.

## GRANT

**Purpose**: Give user specific permissions on objects.

```
-- Grant all permissions on table
GRANT ALL PRIVILEGES ON employees TO 'john_user'@'localhost';

-- Grant specific permissions
GRANT SELECT, INSERT, UPDATE ON products TO 'sales_user'@'%';

-- Grant on specific columns
GRANT SELECT (employee_id, first_name, salary) ON employees TO 'hr_user'@'localhost';

-- Grant permission to grant to others
GRANT SELECT ON orders TO 'analyst'@'localhost' WITH GRANT OPTION;

-- Grant database permissions
GRANT ALL PRIVILEGES ON company_db.* TO 'admin'@'localhost';

-- Grant administrative privileges
GRANT CREATE, DROP, ALTER ON company_db.* TO 'dba'@'localhost';
```

**Common Privileges**:

- `SELECT`: Read data
- `INSERT`: Add new rows
- `UPDATE`: Modify data
- `DELETE`: Remove rows
- `CREATE`: Create tables/databases
- `DROP`: Delete objects
- `ALTER`: Modify structure
- `ALL PRIVILEGES`: All permissions

**Use Cases**:

- Create application user with SELECT-only access
- Give HR staff UPDATE permission for employee records
- Restrict analysts to specific tables
- Create read-only reports user

- Implement principle of least privilege

---

### REVOKE

**Purpose**: Remove user permissions.

```
-- Revoke specific permission
REVOKE SELECT ON employees FROM 'john_user'@'localhost';

-- Revoke all permissions
REVOKE ALL PRIVILEGES ON company_db.* FROM 'temp_user'@'localhost';

-- Revoke INSERT and UPDATE
REVOKE INSERT, UPDATE ON products FROM 'view_only_user'@'localhost';

-- Revoke GRANT OPTION
REVOKE GRANT OPTION ON employees FROM 'analyst'@'localhost';
```

**Use Cases**:

- Remove access when employee leaves
- Reduce permissions after employee role change
- Revoke emergency access granted temporarily
- Clean up unused user accounts

---

## TCL - TRANSACTION CONTROL LANGUAGE

TCL commands manage transactions (group of SQL statements as one atomic unit).

### COMMIT

**Purpose**: Save all changes permanently.

```
-- Implicit commit (auto-commit enabled)
INSERT INTO employees VALUES (...);  -- Automatically saved

-- Explicit commit in transaction
START TRANSACTION;
INSERT INTO employees VALUES (...);
UPDATE departments SET budget = budget - 50000 WHERE id = 5;
COMMIT;  -- Both changes saved together
```

---

### ROLLBACK

**Purpose**: Undo changes not yet committed.

```
-- Undo all changes in transaction
START TRANSACTION;
INSERT INTO employees VALUES (...);
DELETE FROM orders WHERE id = 100;
ROLLBACK;  -- Both operations undone

-- Rollback to savepoint
START TRANSACTION;
INSERT INTO table1 VALUES (...);
SAVEPOINT sp1;
INSERT INTO table2 VALUES (...);
ROLLBACK TO sp1;  -- Undo table2 insert, keep table1 insert
COMMIT;
```

## SAVEPOINT

**Purpose**: Create checkpoint within transaction.

```
START TRANSACTION;

INSERT INTO employees VALUES (1, 'John', ...);
SAVEPOINT after_emp_insert;

INSERT INTO salaries VALUES (1, 75000);
SAVEPOINT after_salary_insert;

INSERT INTO benefits VALUES (1, 'health');
-- Error occurs here
ROLLBACK TO after_salary_insert;  -- Undo only benefit insert

COMMIT;  -- Save employee and salary inserts
```

**Use Cases**:

- Multi-step processes with error recovery
- Partial rollback of complex operations
- Testing scenarios without full data loss

---

## Transaction Properties (ACID)

| Property | Definition | Example |
|---|---|---|
| **Atomicity** | All or nothing | Transfer money: debit AND credit succeed or both fail |
| **Consistency** | Valid state maintained | Account balances stay accurate |
| **Isolation** | Independent execution | Two users can't corrupt each other's data |
| **Durability** | Permanent after commit | Power loss doesn't lose committed data |

---

# COMMAND QUICK REFERENCE

## DDL Commands Summary

| Command | Purpose | Rollback |
|---|---|---|
| `CREATE` | Create objects | No |
| `ALTER` | Modify objects | No |
| `DROP` | Delete objects | No |
| `TRUNCATE` | Remove all rows | Yes* |
| `RENAME` | Rename object | No |

*If within transaction

## DML Commands Summary

| Command | Purpose | Rollback |
|---|---|---|
| `INSERT` | Add rows | Yes |
| `UPDATE` | Modify rows | Yes |
| `DELETE` | Remove rows | Yes |

### DQL Commands Summary

| Command | Purpose |
|---|---|
| `SELECT` | Retrieve data |

### DCL Commands Summary

| Command | Purpose |
|---|---|
| `GRANT` | Give permissions |
| `REVOKE` | Remove permissions |

### TCL Commands Summary

| Command | Purpose |
|---|---|
| `COMMIT` | Save changes |
| `ROLLBACK` | Undo changes |
| `SAVEPOINT` | Create checkpoint |

---

## REAL-WORLD PRACTICE SCENARIOS

### Scenario 1: New Hire Onboarding

```
-- 1. Create necessary records
START TRANSACTION;

-- Insert employee
INSERT INTO employees (first_name, last_name, email, hire_date, department_id)
VALUES ('Sarah', 'Johnson', 'sarah@company.com', CURRENT_DATE, 3);

-- Get inserted employee ID
SET @emp_id = LAST_INSERT_ID();

-- Insert salary record
INSERT INTO salaries (employee_id, salary, effective_date)
VALUES (@emp_id, 65000, CURRENT_DATE);

-- Insert benefits
INSERT INTO benefits (employee_id, health_insurance, retirement_plan)
VALUES (@emp_id, 'premium', '401k');

-- Update department headcount
UPDATE departments
SET employee_count = employee_count + 1
WHERE department_id = 3;

COMMIT;
```

---

### Scenario 2: Order Processing

```
-- 1. Retrieve customer info and create order
START TRANSACTION;

-- Check customer credit limit
SELECT credit_limit, current_balance FROM customers
WHERE customer_id = 100;

-- Create order
```

```
INSERT INTO orders (customer_id, order_date, total_amount)
VALUES (100, CURRENT_DATE, 5000);

SET @order_id = LAST_INSERT_ID();

-- Add order items (from cart)
INSERT INTO order_items (order_id, product_id, quantity, unit_price)
SELECT @order_id, product_id, quantity, price
FROM shopping_cart
WHERE customer_id = 100;

-- Update inventory
UPDATE products
SET stock = stock - (SELECT quantity FROM order_items WHERE order_id = @order_id);

-- Update customer balance
UPDATE customers
SET current_balance = current_balance + 5000
WHERE customer_id = 100;

-- Clear shopping cart
DELETE FROM shopping_cart WHERE customer_id = 100;

COMMIT;
```

---

## Scenario 3: Data Migration with Error Handling

```
-- Archive and delete old records
START TRANSACTION;

-- Archive to history table
INSERT INTO employee_history
SELECT * FROM employees
WHERE status = 'inactive'
  AND termination_date < DATE_SUB(CURRENT_DATE, INTERVAL 1 YEAR);

SAVEPOINT after_archive;

-- Delete from active table
DELETE FROM employees
WHERE status = 'inactive'
  AND termination_date < DATE_SUB(CURRENT_DATE, INTERVAL 1 YEAR);

-- If error occurs, rollback only the delete
-- ROLLBACK TO after_archive;

COMMIT;
```

---

## Scenario 4: Salary Review & Bonus Distribution

```
-- Distribute year-end bonuses based on performance
START TRANSACTION;

-- Create bonus records
INSERT INTO bonuses (employee_id, amount, distribution_date)
SELECT
    e.employee_id,
    CASE
        WHEN p.rating = 'A' THEN e.salary * 0.20
        WHEN p.rating = 'B' THEN e.salary * 0.15
        WHEN p.rating = 'C' THEN e.salary * 0.10
        ELSE 0
    END,
    CURRENT_DATE
FROM employees e
```

```
JOIN performance_ratings p ON e.employee_id = p.employee_id
WHERE YEAR(p.review_date) = YEAR(CURRENT_DATE);

-- Grant salary increases to high performers
UPDATE employees e
JOIN performance_ratings p ON e.employee_id = p.employee_id
SET e.salary = e.salary * 1.10
WHERE p.rating = 'A' AND YEAR(p.review_date) = YEAR(CURRENT_DATE);

-- Update budget allocations
UPDATE departments
SET budget = budget - (
    SELECT COALESCE(SUM(amount), 0)
    FROM bonuses
    WHERE department_id = d.department_id
)
FROM departments d;

COMMIT;
```

---

### Scenario 5: User Access Management

```
-- Grant permissions for new analyst
GRANT SELECT ON sales_db.* TO 'analyst_new'@'localhost';
GRANT SELECT ON sales_db.orders TO 'analyst_new'@'localhost';
GRANT SELECT ON sales_db.customers TO 'analyst_new'@'localhost';

-- Remove permissions for departing employee
REVOKE ALL PRIVILEGES ON sales_db.* FROM 'temp_intern'@'localhost';

-- Create read-only reporting user
CREATE USER 'report_user'@'localhost' IDENTIFIED BY 'secure_password';
GRANT SELECT ON analytics_db.* TO 'report_user'@'localhost';

-- Create admin user with full privileges
CREATE USER 'dba_user'@'localhost' IDENTIFIED BY 'secure_password';
GRANT ALL PRIVILEGES ON *.* TO 'dba_user'@'localhost' WITH GRANT OPTION;
```

---

## INTERVIEW PREPARATION TIPS

### Common Questions & Answers

**Q: What's the difference between DELETE and TRUNCATE?** A: DELETE removes rows one-by-one (slow, triggers fired, WHERE supported), TRUNCATE deallocates pages (fast, no triggers). Both keep structure.

**Q: Can you rollback DDL commands?** A: No, DDL commands auto-commit. They're non-transactional (except in some databases with special settings).

**Q: What's the difference between DROP and TRUNCATE?** A: DROP removes structure + data (irreversible), TRUNCATE removes data only, keeping structure.

**Q: What are the ACID properties?** A: Atomicity (all/nothing), Consistency (valid state), Isolation (independent), Durability (permanent).

**Q: When should you use SELECT into vs INSERT INTO SELECT?** A: Both copy data; SELECT INTO creates new table, INSERT INTO adds to existing table.

**Q: What's a savepoint and when do you use it?** A: Checkpoint within transaction allowing partial rollback. Use for complex multi-step processes with error recovery.

---

## BEST PRACTICES

1. **Always backup before DDL/DML**: DDL commands don't rollback
2. **Use transactions**: Wrap related operations to ensure atomicity
3. **Test with SELECT first**: Before UPDATE/DELETE, test with SELECT using same WHERE
4. **Principle of least privilege**: Grant minimum permissions needed
5. **Use indexes wisely**: Speed up SELECT/WHERE, slow down INSERT/UPDATE/DELETE
6. **Document schema**: Add comments explaining tables and relationships
7. **Version control**: Keep database schema in version control
8. **Monitor transactions**: Long-running transactions lock resources
9. **Use constraints**: Primary keys, foreign keys ensure data integrity
10. **Audit changes**: Log who made what changes and when

---

**Last Updated:** January 2026 **Relevant For:** MySQL, PostgreSQL, SQL Server, Oracle **Difficulty Level:** Beginner to Intermediate

# SQL Comments: Complete Interview Reference Guide

## From Basics to Advanced Concepts

---

## TABLE OF CONTENTS

---

## LEVEL 1: BASIC COMMENTS

### 1.1 Single-Line Comments

**Syntax:**

```
-- This is a single-line comment
SELECT * FROM customers;
```

**Key Points:**

- Starts with two consecutive hyphens (`--`)
- Everything after `--` until the end of the line is ignored
- Used for brief explanations
- Cannot span multiple lines

**Examples:**

```
-- Select all customers
SELECT * FROM customers;
SELECT customer_id, -- unique customer identifier
       customer_name  -- person's name
FROM customers;

-- WHERE city = 'New York'; -- This entire line is commented out
```

**Interview Question:**

> Q: What is the difference between `--` and `//` for comments in SQL? A: `--` is the standard SQL comment syntax. `//` is not valid in standard SQL (though supported in some database systems). Always use `--` for portability.

---

## 1.2 Multi-Line Comments

**Syntax:**

```
/* This is a multi-line comment
   that spans multiple lines */
SELECT * FROM customers;
```

**Key Points:**

- Starts with `/*` and ends with `*/`
- Can span multiple lines
- Cannot be nested in most SQL databases
- Useful for longer explanations

**Examples:**

```
/* Select all customers from the database
   This query retrieves the complete customer list
   for reporting purposes */
SELECT * FROM customers;

/* Temporary comment - remove this section after testing
SELECT * FROM test_table; */

SELECT customer_id, customer_name
/* , customer_email */ -- Temporarily disabled email column
FROM customers;
```

**Interview Question:**

> Q: Can you nest multi-line comments in SQL? A: No, most SQL databases don't support nested comments. You cannot have `/* comment1 /* comment2 */ comment1 */`. Some modern databases like PostgreSQL support nested comments, but it's not standard SQL.

---

# LEVEL 2: INTERMEDIATE COMMENT TECHNIQUES

## 2.1 Inline Comments

**Concept:** Placing comments on the same line as code for quick clarifications.

**Examples:**

```
SELECT
    customer_id,           -- Unique customer identifier
    customer_name,         -- Customer's full name
    email,                  -- Contact email address
    order_count             -- Total number of orders placed
FROM customers
WHERE status = 'active';    -- Only active customers

-- Alternative: Using /* */ inline
SELECT customer_id /* PK */, order_total /* Amount in USD */
FROM orders;
```

**When to Use:**

- Column explanations in SELECT clauses
- Clarifying filter conditions
- Brief context for aggregations

---

## 2.2 Comment Blocks for Query Sections

**Pattern:** Organize long queries with comment blocks separating logical sections.

```
-- ========================================
-- QUERY: Monthly Sales Report
-- Date: January 2024
-- ========================================

-- 1. CTE to calculate monthly totals
WITH monthly_sales AS (
    SELECT
        DATE_TRUNC('month', order_date) AS month,
        SUM(order_amount) AS total_sales
    FROM orders
    WHERE YEAR(order_date) = 2024
    GROUP BY DATE_TRUNC('month', order_date)
),

-- 2. CTE to rank months by sales
ranked_months AS (
    SELECT
        month,
        total_sales,
        ROW_NUMBER() OVER (ORDER BY total_sales DESC) AS sales_rank
    FROM monthly_sales
)

-- 3. Final selection with filters
SELECT
    month,
    total_sales,
    sales_rank
FROM ranked_months
WHERE sales_rank <= 5;  -- Top 5 months only
```

---

## 2.3 Commenting Out Code for Testing

**Purpose:** Temporarily disable queries without deleting them.

```
-- Testing new filters
/*
SELECT customer_id, customer_name, order_count
FROM customers
WHERE order_count > 100;
*/

-- Using existing filter instead
```

```
SELECT customer_id, customer_name, order_count
FROM customers
WHERE order_count > 50;
```

---

### 2.4 Documentation Comments

**Pattern:** Include metadata about the query at the top.

```
/*
=============================================================================
QUERY NAME: Customer Lifetime Value Report
PURPOSE: Calculate total spending per customer for analysis
AUTHOR: Data Analytics Team
DATE CREATED: 2024-01-15
LAST MODIFIED: 2024-01-20
MODIFIED BY: John Doe
DATABASE: main_analytics
FREQUENCY: Weekly (Tuesday 8 AM)
OWNER: Finance Department
NOTES: Used for customer segmentation and targeting
=============================================================================
*/

SELECT
    customer_id,
    customer_name,
    SUM(order_total) AS lifetime_value,
    COUNT(order_id) AS total_orders,
    AVG(order_total) AS avg_order_value
FROM orders
GROUP BY customer_id, customer_name
ORDER BY lifetime_value DESC;
```

---

# LEVEL 3: ADVANCED COMMENT PATTERNS

### 3.1 Explaining Complex Business Logic

**Purpose:** Document non-obvious decisions and business rules.

```
-- Complex scenario: Calculate average order value excluding outliers
-- Business rule: Exclude top 1% of orders (likely bulk/special deals)
-- and bottom 1% (likely discounted/promotional orders)
-- This gives us a more representative average for standard customers

SELECT
    customer_id,
    -- Using PERCENTILE_CONT for statistical accuracy
    -- Note: Some databases use different function names
    AVG(order_total) AS normalized_avg_order_value
FROM orders
WHERE order_total NOT IN (
    -- Exclude top 1% (high-value outliers)
    SELECT PERCENTILE_CONT(0.99) WITHIN GROUP (ORDER BY order_total)
    FROM orders
)
AND order_total NOT IN (
    -- Exclude bottom 1% (discount/promo outliers)
    SELECT PERCENTILE_CONT(0.01) WITHIN GROUP (ORDER BY order_total)
    FROM orders
)
GROUP BY customer_id;
```

---

## 3.2 Performance-Related Comments

**Purpose:** Explain optimization decisions.

```
/*
PERFORMANCE NOTE:
- Using INNER JOIN instead of subquery for better execution plan
- Original approach with subquery took 4.2 seconds on 10M rows
- Refactored join version takes 0.8 seconds
- Index: orders(customer_id, order_date) on line 3 query is crucial
*/

SELECT
    c.customer_id,
    c.customer_name,
    COUNT(o.order_id) AS order_count,
    SUM(o.order_total) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
    -- Using INNER JOIN as LEFT JOIN was 5x slower due to
    -- aggregate function behavior with NULL handling
WHERE o.order_date >= DATEADD(YEAR, -1, GETDATE())
GROUP BY c.customer_id, c.customer_name
HAVING COUNT(o.order_id) > 5  -- Customers with 5+ orders only
ORDER BY total_spent DESC;
```

---

## 3.3 Data Quality & Edge Case Comments

**Purpose:** Flag known data issues and how they're handled.

```
/*
DATA QUALITY NOTES:
1. NULL customer_email values: ~2% of records
   - Handling: Excluded from email notifications via COALESCE check
2. Duplicate order entries: Some orders appear twice in raw data
   - Handling: Using ROW_NUMBER() window function to deduplicate
3. Future-dated orders: ~50 records with dates > today
   - Handling: Explicitly filtering WHERE order_date <= CURRENT_DATE
4. Negative amounts: ~0.5% of records (refunds/adjustments)
   - Handling: Including in calculations as legitimate transactions
*/

SELECT
    customer_id,
    customer_name,
    COALESCE(email, 'no_email@unknown.com') AS customer_email,
    total_spent
FROM (
    SELECT DISTINCT ON (customer_id)  -- PostgreSQL syntax, use ROW_NUMBER for others
        c.customer_id,
        c.customer_name,
        c.email,
        SUM(o.order_total) AS total_spent,
        ROW_NUMBER() OVER (PARTITION BY c.customer_id ORDER BY o.order_date DESC) AS rn
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.order_date <= CURRENT_DATE
) deduped
WHERE rn = 1;
```

---

## 3.4 Predicate Pushdown & Optimization Comments

**Purpose:** Explain filter placement strategy for distributed systems.

```
/*
QUERY OPTIMIZATION: PREDICATE PUSHDOWN STRATEGY
Modern distributed SQL engines (Spark, Presto) benefit from early filtering.
This query is designed with filters positioned to:
1. Filter in source tables first (WHERE clauses on base tables)
2. Minimize data shuffled between nodes
3. Apply aggregate filters AFTER GROUP BY (HAVING)

Execution order optimized for systems like Spark SQL:
- Source filtering (TableScan + Filter) → Most selective
- Early projections (Select specific columns) → Reduces memory
- Join predicates (ON conditions) → Applied before join
- Post-aggregation filters (HAVING) → Applied last
*/

-- Step 1: Filter BEFORE aggregation (most efficient)
SELECT
    product_category,
    SUM(sales_amount) AS total_sales,
    COUNT(order_id) AS order_count
FROM sales_transactions
WHERE -- Applied at source (pushed down to storage)
    transaction_date >= DATE '2024-01-01'
    AND region_id IN (1, 2, 3)  -- Partition-aware filter
    AND sales_amount > 0  -- Exclude zero/negative values early
GROUP BY product_category
HAVING -- Applied after aggregation
    COUNT(order_id) >= 100  -- Minimum transaction threshold
ORDER BY total_sales DESC;
```

---

### 3.5 Version History & Migration Comments

**Purpose:** Track changes over time for maintenance.

```
/*
VERSION HISTORY:
v3.0 (2024-01-20) - John Doe
  - Changed INNER JOIN to LEFT JOIN for historical analysis
  - Added date range parameters for flexibility

v2.5 (2024-01-10) - Jane Smith
  - Optimized aggregation using window functions
  - Reduced query time from 12s to 3s
  - Added comments for business logic

v2.0 (2023-12-15) - Initial Production Release
  - Replaced subqueries with CTEs
  - Added indexing recommendations

DEPRECATED: Old version uses SELECT * - REMOVE after migration complete

MIGRATION NOTE:
This query replaces stored procedure sp_customer_analysis
To be phased out by Q1 2025
*/

SELECT ...
```

---

## LEVEL 4: BEST PRACTICES & PRODUCTION CODE

### 4.1 Professional Commenting Standards

**□ DO:**

- Explain the "why" behind decisions, not just the "what"
- Comment complex business logic and non-obvious joins
- Include data quality notes and edge cases handled
- Document performance-critical sections
- Use consistent formatting and indentation
- Keep comments updated when code changes

□ **DON'T:**

- Comment obvious code (e.g., `SELECT * FROM customers -- select all customers`)
- Over-comment simple, self-explanatory statements
- Include secrets, passwords, or sensitive information in comments
- Forget to update comments when code logic changes
- Write misleading or outdated comments
- Use cryptic abbreviations without explanation

---

## 4.2 Production-Grade Comment Template

```
/*
================================================================================
QUERY METADATA
================================================================================
NAME:           Customer Churn Analysis Report
PURPOSE:        Identify high-risk customers likely to churn in next quarter
OWNER:          Customer Success Team
CREATED:        2024-01-15 by Analytics Team
LAST UPDATED:   2024-01-20 by John Doe
FREQUENCY:      Weekly (Every Monday 6 AM)
RUNTIME:        ~2.5 minutes on 100M row dataset
AFFECTED TEAMS: Sales, Customer Success, Finance
DEPENDENCIES:   orders, customers, payments, returns tables
ALERT OWNERS:   manager@company.com, analyst@company.com
================================================================================

BUSINESS LOGIC:
Churned customers = no purchases for 90+ days + high return rate (>30%)
Rationale: Customers without recent activity + quality concerns are highest risk

DATA QUALITY ISSUES HANDLED:
1. NULL payment dates: Treated as no transaction (counted as dormant)
2. Duplicate orders: Deduplicated using latest order date
3. Negative amounts: Included (refunds/returns tracked separately)
4. Future dates: Excluded (data entry errors)

PERFORMANCE NOTES:
- Indexed columns: customer_id, order_date, payment_status
- Join strategy: INNER JOINs used (removed NULL edge cases)
- Predicate pushdown: Filters applied on source tables first
- Previous version (subqueries): 8.2 sec → Current (CTEs): 2.5 sec

KNOWN LIMITATIONS:
- Does not account for seasonal variations
- Churned customers dataset lags by 2-3 days (batch job delay)
- Missing data for 15% of international customers
================================================================================
*/

WITH customer_activity AS (
    -- Calculate days since last purchase and order metrics
    SELECT
        c.customer_id,
        c.customer_name,
        c.signup_date,
        MAX(o.order_date) AS last_purchase_date,
        DATEDIFF(DAY, MAX(o.order_date), CURRENT_DATE) AS days_since_purchase,
        COUNT(o.order_id) AS total_orders,
        SUM(o.order_total) AS lifetime_value
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    WHERE c.signup_date < DATEADD(YEAR, -1, CURRENT_DATE)  -- Active for 1+ year
    GROUP BY c.customer_id, c.customer_name, c.signup_date
),

customer_returns AS (
    -- Calculate return rate (high returns indicate dissatisfaction)
    SELECT
        c.customer_id,
        COUNT(DISTINCT r.order_id) AS return_count,
        COUNT(DISTINCT o.order_id) AS order_count,
        CAST(COUNT(DISTINCT r.order_id) AS FLOAT)
            / NULLIF(COUNT(DISTINCT o.order_id), 0) AS return_rate
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    LEFT JOIN returns r ON o.order_id = r.order_id
    WHERE o.order_date >= DATEADD(YEAR, -2, CURRENT_DATE)
    GROUP BY c.customer_id
)

-- Final churn risk assessment
SELECT
    ca.customer_id,
    ca.customer_name,
    ca.days_since_purchase,
    ca.last_purchase_date,
    ca.total_orders,
    ca.lifetime_value,
    cr.return_rate,
    CASE
        WHEN ca.days_since_purchase >= 90 AND cr.return_rate > 0.30
            THEN 'HIGH_RISK'
        WHEN ca.days_since_purchase >= 60 AND cr.return_rate > 0.20
            THEN 'MEDIUM_RISK'
        WHEN ca.days_since_purchase >= 30
            THEN 'LOW_RISK'
        ELSE 'ACTIVE'
    END AS churn_risk_category
FROM customer_activity ca
LEFT JOIN customer_returns cr ON ca.customer_id = cr.customer_id
WHERE ca.days_since_purchase >= 30  -- Only customers inactive 30+ days
ORDER BY ca.days_since_purchase DESC;
```

---

## 4.3 Database-Specific Comment Features

**MySQL:**

```
-- Single-line comment
# Alternative single-line comment
/* Multi-line comment */
```

**PostgreSQL:**

```
-- Standard single-line
/* Standard multi-line */
/* Nested comments supported: /* inner */ outer */
```

**SQL Server:**

```
-- Standard single-line
/* Standard multi-line
```

```
   spanning multiple lines */
-- No nested comment support
```

**Oracle:**

```
-- Single-line
/* Multi-line */
-- No nested comments
```

---

## INTERVIEW Q&A SECTION

### Q1: What are the two main types of SQL comments?

**Answer:**

1. **Single-line comments**: Use `--` syntax, extend to end of line
2. **Multi-line comments**: Use `/* ... */` syntax, can span multiple lines

```
-- Example single-line
SELECT * FROM customers;

/* Example
   multi-line */
SELECT * FROM orders;
```

---

### Q2: Can you nest multi-line comments in SQL?

**Answer:** Depends on the database:

- **Most databases (MySQL, SQL Server, PostgreSQL)**: NO, nesting is NOT supported
- **PostgreSQL (with special flag)**: YES, nesting IS supported

```
-- This will cause an error in most databases:
/* Outer comment
   /* Inner comment */  -- This closes the outer comment early!
   Rest is NOT commented
*/

-- Workaround: Use different syntax or separate comments
/* Outer comment */
/* Inner comment */
```

---

### Q3: How should you comment complex business logic in production?

**Answer:** Focus on **WHY**, not WHAT. Include:

- Purpose of the calculation
- Business rules being applied
- Any edge cases or data quality issues handled
- Performance considerations

```
--  BAD: Just restates the code
SELECT COUNT(*) FROM customers; -- Count customers

--  GOOD: Explains the business logic
-- Customers with 2+ orders in last 90 days represent
-- "engaged" segment for targeted marketing campaign
-- Excludes test/demo accounts (email like '%@internal.%')
SELECT COUNT(*)
```

```
FROM customers
WHERE order_count >= 2
  AND last_order_date >= DATEADD(DAY, -90, CURRENT_DATE)
  AND email NOT LIKE '%@internal.%';
```

---

## Q4: What's the best practice for commenting code you temporarily disable?

**Answer:** Use multi-line comments and include WHY it's disabled:

```
/*
TEMPORARILY DISABLED (2024-01-20 by John Doe)
Reason: Awaiting data pipeline fix - email column has 40% NULLs
Expected re-enable: 2024-01-25
Alternative: Using customer_phone instead

SELECT customer_id, email
FROM customers;
*/

-- Using alternative until email data is clean
SELECT customer_id, customer_phone
FROM customers;
```

---

## Q5: How do comments help with query optimization discussion?

**Answer:** Comments document optimization decisions and their impact:

```
/*
PERFORMANCE OPTIMIZATION:
- Original subquery approach: 8.2 seconds
- Refactored with JOIN: 2.5 seconds (69% faster)
- Reason: Avoids repeated table scans from subquery execution

Index requirements:
- CREATE INDEX idx_orders_customer ON orders(customer_id, order_date);
*/

SELECT c.customer_id, COUNT(o.order_id)
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id  -- Optimized approach
WHERE o.order_date > DATEADD(YEAR, -1, CURRENT_DATE)
GROUP BY c.customer_id;
```

---

## Q6: What information should be in a query header comment?

**Answer:**

- Query name/purpose
- Owner/creator and last modified by
- Frequency of execution
- Any dependencies or related queries
- Known data quality issues
- Performance characteristics

```
/*
QUERY: Monthly Revenue Report
PURPOSE: Track monthly revenue trends for executive dashboard
OWNER: Finance Department (finance@company.com)
CREATED: 2024-01-10
LAST UPDATED: 2024-01-20
FREQUENCY: Daily at 6 AM
```

```
DEPENDENCIES: sales, orders, products tables
RUNTIME: ~45 seconds on production (100M rows)
KNOWN ISSUES:
  - International sales missing 5% due to conversion delays
  - Refunds processed with 2-day lag
*/
```

---

## Q7: How do you handle comments for different SQL database systems?

**Answer:** The two standard syntaxes work across all major databases:

- `--` single-line (most portable)
- `/* */` multi-line (most portable)

```
-- STANDARD (works everywhere): Use these

-- Single-line comment works in all databases
SELECT * FROM customers;

/* Multi-line comment
   works in all databases */
SELECT * FROM orders;

-- Database-specific (avoid for portability):
# MySQL only
/* PostgreSQL allows nesting with special flag */
```

---

## Q8: What's the difference between documenting code and over-commenting?

**Answer:**

□ **Good documentation:**

- Explains non-obvious business rules
- Clarifies complex JOIN conditions
- Notes data quality issues handled
- Documents optimization decisions
- Why a specific approach was chosen

□ **Over-commenting:**

- Restating obvious code
- Excessive comments on simple statements
- Irrelevant information
- Comments that go out of sync with code

```
-- □ Over-commenting (obvious code)
SELECT customer_id, -- This is customer id
       customer_name -- This is customer name
FROM customers;    -- From customers table

-- □ Good commenting (informative)
-- BUSINESS RULE: "Active" customers = purchased in last 90 days
-- This segment receives priority customer support
SELECT customer_id, customer_name
FROM customers
WHERE last_purchase_date >= DATEADD(DAY, -90, CURRENT_DATE);
```

---

## Q9: How should comments handle deprecated code?

**Answer:** Include deprecation notice with timeline and replacement information:

```
/*
⚠ DEPRECATED: This procedure is being phased out
Replacement: Use get_customer_summary_v2 instead
Timeline: Will be removed 2024-03-31
Migration guide: See confluence link [insert URL]
Owner: Data Engineering Team
*/

-- Old version using cursor (slow and outdated)
CREATE PROCEDURE old_customer_analysis
AS BEGIN
    -- ... cursor-based implementation ...
END;

-- NEW VERSION: Use this instead
SELECT customer_id, summary_data
FROM customer_summary_v2
WHERE active = 1;
```

---

## Q10: What's the best comment style for collaborative teams?

**Answer:** Consistent, standardized format with clear ownership:

```
/*
================================================================================
QUERY NAME:     Quarterly Sales Analysis
OWNER:          @john_doe (john@company.com)
MODIFIED:       2024-01-20 by @jane_smith
REVIEWED:       2024-01-20 by @manager
APPROVER:       @executive_sponsor
================================================================================

PURPOSE:
Generate quarterly sales metrics for board reporting

BUSINESS LOGIC:
- Sales include all completed transactions (payment_status = 'COMPLETED')
- Excludes test/internal transactions (source = 'INTERNAL')
- Refunds shown as negative amounts (not excluded)

ASSUMPTIONS:
- Data refreshed nightly from source system
- All dates in UTC timezone
- Currency conversions based on spot rates

KNOWN LIMITATIONS:
- Lag of 2-3 days for international transactions
- 15% of legacy data missing category information
- Sales tax calculations incomplete for 8% of US orders

DATA QUALITY NOTES:
See data quality dashboard: [insert URL]
Last validation: 2024-01-20

PERFORMANCE:
Runtime: 2.3 minutes (100M row dataset)
Last optimization: 2024-01-15 (CTE refactor)
Recommended frequency: Daily

DEPENDENCIES:
- Table: sales (updated nightly 1 AM UTC)
- Table: customers (real-time sync)
- View: reporting_dates (maintained by data team)
```

```
RELATED QUERIES:
- annual_sales_analysis.sql
- sales_by_region.sql
- customer_lifetime_value.sql

NEXT REVIEW DATE: 2024-02-20
================================================================================
*/

SELECT ...
```

---

## QUICK REFERENCE CHEAT SHEET

| Aspect | Single-Line (`--`) | Multi-Line (`/* */`) |
|---|---|---|
| **Syntax** | `-- comment` | `/* comment */` |
| **Length** | One line only | Multiple lines |
| **Nesting** | N/A | Not supported (usually) |
| **Use Case** | Quick notes, inline | Long explanations, blocks |
| **Portability** | All databases | All databases |
| **Best For** | Inline clarifications | Documentation headers |

---

## KEY INTERVIEW TAKEAWAYS

1. **Commenting is about communication** - Explain WHY decisions were made, not WHAT the code does
2. **Two types**: `--` for single-line, `/* */` for multi-line - both universal
3. **Quality > Quantity** - Better to comment complex logic than everything
4. **Production professionalism** - Include metadata, owner, dependencies, known issues
5. **Data quality matters** - Document edge cases and how they're handled
6. **Performance perspective** - Explain optimization decisions and their impact
7. **Consistency wins** - Standardized format across team is more valuable than perfect comments
8. **Keep it current** - Update comments when code changes (more critical than initial creation)
9. **Avoid pitfalls** - No misleading, outdated, or over-obvious comments
10. **Database awareness** - Know your system (MySQL vs PostgreSQL vs SQL Server differences)

---

**Last Updated:** January 2024 **Relevant For:** All SQL Database Systems (MySQL, PostgreSQL, SQL Server, Oracle) **Interview Confidence Level:** Beginner to Advanced Professional