-- Validate

IF @balance >= 100 THEN

-- Deduct from source

UPDATE accounts SET balance = balance - 100 WHERE id = 1;

```
-- Add to destination
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

-- Record transaction
INSERT INTO transfers (from_account, to_account, amount, created_at)
VALUES (1, 2, 100, NOW());

COMMIT;
```

ELSE

ROLLBACK;

SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Insufficient funds';

END IF;

Savepoints

Create checkpoints within a transaction.
`````sql
START TRANSACTION;

INSERT INTO logs (message) VALUES ('Start operation');
SAVEPOINT after_log;

UPDATE products SET price = price * 1.1 WHERE category = 'Electronics';
SAVEPOINT after_electronics;

UPDATE products SET price = price * 1.2 WHERE category = 'Clothing';
SAVEPOINT after_clothing;

-- Oops, rollback clothing changes but keep electronics
ROLLBACK TO SAVEPOINT after_electronics;

-- Or start over
ROLLBACK TO SAVEPOINT after_log;

COMMIT;

-- Release savepoint (optional cleanup)
RELEASE SAVEPOINT after_electronics;
`````

✅ Transaction Best Practices:

1. **Keep transactions short** - Minimize lock duration
2. **Don't include user interaction** in transactions
3. **Acquire locks in consistent order** to prevent deadlocks
4. **Use appropriate isolation level**
5. **Always handle errors** with proper rollback
6. **Avoid DDL in transactions** (CREATE, ALTER, DROP trigger implicit commit)
7. **Test rollback scenarios**
8. **Monitor long-running transactions**
`````sql
-- Bad: User interaction in transaction
START TRANSACTION;
UPDATE inventory SET qty = qty - 1 WHERE product_id = 101;
-- Wait for user to confirm payment (locks held!)
COMMIT;

-- Good: Complete transaction quickly
`````

```
-- 1. Reserve inventory
START TRANSACTION;
UPDATE inventory SET qty = qty - 1, reserved = reserved + 1 WHERE product_id = 101;
COMMIT;

-- 2. User confirms payment

-- 3. Complete order
START TRANSACTION;
UPDATE inventory SET reserved = reserved - 1 WHERE product_id = 101;
INSERT INTO orders ...;
COMMIT;
```

6.2 Isolation Levels

Control how concurrent transactions interact and trade off consistency vs performance.

Four Isolation Levels

Level	Dirty Read	Non-Repeatable Read	Phantom Read	Performance
READ UNCOMMITTED	✓	✓	✓	Fastest
READ COMMITTED	✗	✓	✓	Fast
REPEATABLE READ (default)	✗	✗	✓*	Medium
SERIALIZABLE	✗	✗	✗	Slowest

*InnoDB prevents phantom reads even at REPEATABLE READ

Set Isolation Level

```sql
-- Session level (current connection)
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Global level (all new connections)
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- For next transaction only
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Check current level
SELECT @@transaction_isolation;  -- MySQL 8.0+
SELECT @@tx_isolation;           -- MySQL 5.7
```

READ UNCOMMITTED

Lowest isolation. Transactions can see uncommitted changes from other transactions.

```sql
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
-- Not committed yet

-- Transaction 2 (READ UNCOMMITTED)
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1;  -- Sees 1000 (dirty read!)

-- Transaction 1
ROLLBACK;  -- Transaction 2 saw data that was rolled back!
```

Use case: Rarely used. Only for analytics on non-critical data where performance is critical.

READ COMMITTED

Default in PostgreSQL/Oracle. Prevents dirty reads.

```sql
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1;  -- Returns 500

-- Transaction 2
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
COMMIT;

-- Transaction 1 (still open)
SELECT balance FROM accounts WHERE id = 1;  -- Returns 1000 (non-repeatable read!)
-- Same query, different result!
COMMIT;
```

Use case: Good for reducing locks, acceptable when reads don't need to be repeatable.

REPEATABLE READ

MySQL's default. Prevents dirty and non-repeatable reads.

```sql
-- Transaction 1
```

```sql
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1;  -- Returns 500

-- Transaction 2
UPDATE accounts SET balance = 1000 WHERE id = 1;
COMMIT;

-- Transaction 1 (still open)
SELECT balance FROM accounts WHERE id = 1;  -- Still returns 500 (consistent read!)
COMMIT;
```

InnoDB Special: Uses MVCC (Multi-Version Concurrency Control) to prevent phantom reads too.

```sql
-- Transaction 1
START TRANSACTION;
SELECT COUNT(*) FROM orders WHERE status = 'pending';  -- Returns 10

-- Transaction 2
INSERT INTO orders (status) VALUES ('pending');
COMMIT;

-- Transaction 1
SELECT COUNT(*) FROM orders WHERE status = 'pending';  -- Still returns 10 in InnoDB!
```

Use case: Default choice. Good balance of consistency and performance.

SERIALIZABLE

Highest isolation. Complete isolation, as if transactions ran serially.

```sql
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT * FROM inventory WHERE product_id = 101;
-- Holds shared locks on all rows read

-- Transaction 2
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
UPDATE inventory SET qty = qty - 1 WHERE product_id = 101;
-- BLOCKED! Waiting for Transaction 1 to release locks

-- Transaction 1
COMMIT;  -- Transaction 2 can now proceed
```

```

**Use case**: Financial systems, when absolute consistency required. Expect reduced concurrency.

#### Choosing Isolation Level
```sql
-- READ UNCOMMITTED
-- ❌ Almost never use
-- When: Bulk analytics, non-critical reports

-- READ COMMITTED
-- ✅ Good for reducing locks
-- When: Most OLTP workloads, acceptable non-repeatable reads

-- REPEATABLE READ
-- ✅ MySQL default, good choice
-- When: Need consistent reads within transaction

-- SERIALIZABLE
-- ⚠️ Use sparingly
-- When: Financial transactions, strict consistency required
```

6.3 Deadlocks

Deadlock occurs when two+ transactions wait for each other to release locks.

Deadlock Example
```sql
-- Transaction 1
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;  -- Locks account 1
-- ... doing some work ...
UPDATE accounts SET balance = balance + 100 WHERE id = 2;  -- Waits for lock on account 2

-- Transaction 2 (running concurrently)
START TRANSACTION;
UPDATE accounts SET balance = balance - 50 WHERE id = 2;   -- Locks account 2
-- ... doing some work ...
UPDATE accounts SET balance = balance + 50 WHERE id = 1;   -- Waits for lock on account 1

-- DEADLOCK! Each waiting for the other
-- MySQL detects deadlock and rolls back one transaction
```

Detecting Deadlocks
```sql
```
```

```
-- Show recent deadlock information
SHOW ENGINE INNODB STATUS;
-- Look for "LATEST DETECTED DEADLOCK" section

-- Enable deadlock logging (MySQL 8.0+)
SET GLOBAL innodb_print_all_deadlocks = ON;
-- Deadlocks logged to error log
```

Preventing Deadlocks

1. Access objects in same order

```sql
-- Bad: Different order
-- Transaction 1: Lock A, then B
-- Transaction 2: Lock B, then A  -- DEADLOCK RISK

-- Good: Same order
-- Transaction 1: Lock A, then B
-- Transaction 2: Lock A, then B  -- NO DEADLOCK
```

```sql
-- Always lock accounts in ID order
START TRANSACTION;
SELECT @id1 := LEAST(from_account, to_account);
SELECT @id2 := GREATEST(from_account, to_account);
SELECT * FROM accounts WHERE id = @id1 FOR UPDATE;
SELECT * FROM accounts WHERE id = @id2 FOR UPDATE;
-- Now safe to update
```

2. Keep transactions short

```sql
-- Bad: Long transaction
START TRANSACTION;
UPDATE inventory SET qty = qty - 1 WHERE id = 101;
-- ... complex business logic (10 seconds) ...
-- ... external API call (30 seconds) ...
COMMIT;

-- Good: Quick transaction
-- Do prep work outside transaction
START TRANSACTION;
UPDATE inventory SET qty = qty - 1 WHERE id = 101;
COMMIT;
-- Then do other work
```

3. Use lower isolation level

```sql
-- SERIALIZABLE: More locks = more deadlock risk
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- Fewer locks held
```

4. Use timeouts

```sql
-- Set lock wait timeout (seconds)
SET SESSION innodb_lock_wait_timeout = 5;

-- Transaction will error after 5 seconds waiting for lock
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- If lock not acquired in 5s: ERROR 1205: Lock wait timeout exceeded
```

5. Retry deadlock victims

```python
# Python example
def transfer_money(from_id, to_id, amount):
    max_retries = 3
    for attempt in range(max_retries):
        try:
            cursor.execute("START TRANSACTION")
            cursor.execute("UPDATE accounts SET balance = balance - %s WHERE id = %s",
                           (amount, from_id))
            cursor.execute("UPDATE accounts SET balance = balance + %s WHERE id = %s",
                           (amount, to_id))
            cursor.execute("COMMIT")
            return True
        except DeadlockError:
            cursor.execute("ROLLBACK")
            if attempt == max_retries - 1:
                raise
            time.sleep(0.1 * (attempt + 1))  # Exponential backoff
    return False
```

6. Use SELECT ... FOR UPDATE carefully

```sql
-- Acquire exclusive locks explicitly
START TRANSACTION;
SELECT * FROM inventory WHERE product_id = 101 FOR UPDATE;
-- Now safe to update
```

```
UPDATE inventory SET qty = qty - 1 WHERE product_id = 101;
COMMIT;
```

7. Avoid hot spots
```sql
-- Bad: Everyone updating same counter
UPDATE global_stats SET page_views = page_views + 1;

-- Better: Use separate counters, aggregate later
INSERT INTO page_view_log (timestamp) VALUES (NOW());
-- Aggregate periodically
```

Locking Reads
```sql
-- Shared lock (SELECT ... FOR SHARE)
-- Others can read, not write
START TRANSACTION;
SELECT * FROM products WHERE id = 101 FOR SHARE;
-- Other transactions can also read, but not update
COMMIT;

-- Exclusive lock (SELECT ... FOR UPDATE)
-- Others can't read or write
START TRANSACTION;
SELECT * FROM products WHERE id = 101 FOR UPDATE;
-- Other transactions blocked from reading/writing
UPDATE products SET stock = stock - 1 WHERE id = 101;
COMMIT;

-- Skip locked rows (MySQL 8.0+)
-- Useful for job queues
START TRANSACTION;
SELECT * FROM jobs
WHERE status = 'pending'
ORDER BY created_at
LIMIT 1
FOR UPDATE SKIP LOCKED;
-- Gets first available job, skips if locked by another worker

-- NOWAIT (MySQL 8.0+)
-- Error immediately instead of waiting
START TRANSACTION;
SELECT * FROM accounts WHERE id = 1 FOR UPDATE NOWAIT;
-- Errors immediately if locked, doesn't wait
```

✅ Deadlock Best Practices:

1. **Always lock in same order** (e.g., ascending ID)
2. **Keep transactions as short as possible**
3. **Avoid user interaction in transactions**
4. **Use appropriate isolation level**
5. **Implement retry logic** for deadlock victims
6. **Monitor deadlock frequency**
7. **Index foreign keys** to reduce lock escalation
8. **Consider application-level locking** for complex scenarios

---

Chapter 7: Security Best Practices

7.1 User Management

Creating Users
```sql
-- Create user
CREATE USER 'appuser'@'localhost' IDENTIFIED BY 'strong_password_123!';

-- Create user with host wildcard
CREATE USER 'appuser'@'%' IDENTIFIED BY 'strong_password_123!';
-- '%' allows connection from any host

-- Create user for specific IP range
CREATE USER 'appuser'@'192.168.1.%' IDENTIFIED BY 'strong_password_123!';

-- Create user with authentication plugin
CREATE USER 'appuser'@'localhost'
IDENTIFIED WITH caching_sha2_password BY 'password';

-- Create user if not exists (MySQL 8.0+)
CREATE USER IF NOT EXISTS 'appuser'@'localhost' IDENTIFIED BY 'password';
```

Password Management
```sql
-- Change password
ALTER USER 'appuser'@'localhost' IDENTIFIED BY 'new_password';

-- Set password expiration
ALTER USER 'appuser'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;

-- Force password change on next login
```

```
ALTER USER 'appuser'@'localhost' PASSWORD EXPIRE;

-- Password history (prevent reuse)
SET GLOBAL password_history = 5;  -- Can't reuse last 5 passwords
ALTER USER 'appuser'@'localhost' PASSWORD HISTORY 5;

-- Password complexity requirements
SET GLOBAL validate_password.policy = STRONG;
SET GLOBAL validate_password.length = 12;
SET GLOBAL validate_password.mixed_case_count = 1;
SET GLOBAL validate_password.number_count = 1;
SET GLOBAL validate_password.special_char_count = 1;
```

Grant Privileges

```sql
-- Grant all privileges on database
GRANT ALL PRIVILEGES ON mydb.* TO 'appuser'@'localhost';

-- Grant specific privileges
GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.* TO 'appuser'@'localhost';

-- Grant on specific table
GRANT SELECT, UPDATE ON mydb.users TO 'appuser'@'localhost';

-- Grant specific columns
GRANT UPDATE (email, phone) ON mydb.users TO 'appuser'@'localhost';

-- Grant with grant option (user can grant to others)
GRANT SELECT ON mydb.* TO 'appuser'@'localhost' WITH GRANT OPTION;

-- Common privilege sets
-- Read-only user
GRANT SELECT ON mydb.* TO 'readonly'@'localhost';

-- Application user
GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.* TO 'appuser'@'localhost';

-- Admin user (not root)
GRANT ALL PRIVILEGES ON mydb.* TO 'admin'@'localhost';

-- Backup user
GRANT SELECT, LOCK TABLES, SHOW VIEW, RELOAD ON *.* TO 'backup'@'localhost';

-- Apply changes
FLUSH PRIVILEGES;
```

Privilege Levels
```sql
-- Global privileges
GRANT SELECT ON *.* TO 'user'@'host';

-- Database privileges
GRANT SELECT ON mydb.* TO 'user'@'host';

-- Table privileges
GRANT SELECT ON mydb.users TO 'user'@'host';

-- Column privileges
GRANT SELECT (id, email) ON mydb.users TO 'user'@'host';

-- Stored routine privileges
GRANT EXECUTE ON PROCEDURE mydb.my_procedure TO 'user'@'host';
```

Revoke Privileges
```sql
-- Revoke specific privilege
REVOKE DELETE ON mydb.* FROM 'appuser'@'localhost';

-- Revoke all privileges
REVOKE ALL PRIVILEGES ON mydb.* FROM 'appuser'@'localhost';

-- Remove grant option
REVOKE GRANT OPTION ON mydb.* FROM 'appuser'@'localhost';
```

View Privileges
```sql
-- Show current user privileges
SHOW GRANTS;

-- Show specific user privileges
SHOW GRANTS FOR 'appuser'@'localhost';

-- Query privilege tables
SELECT * FROM mysql.user WHERE user = 'appuser';
SELECT * FROM mysql.db WHERE user = 'appuser';
SELECT * FROM mysql.tables_priv WHERE user = 'appuser';
```

Delete Users
```sql
```

```
-- Drop user
DROP USER 'appuser'@'localhost';

-- Drop if exists
DROP USER IF EXISTS 'appuser'@'localhost';

-- Drop multiple users
DROP USER 'user1'@'localhost', 'user2'@'localhost';
```

Roles (MySQL 8.0+)

```sql
-- Create role
CREATE ROLE 'app_read', 'app_write', 'app_admin';

-- Grant privileges to role
GRANT SELECT ON mydb.* TO 'app_read';
GRANT INSERT, UPDATE, DELETE ON mydb.* TO 'app_write';
GRANT ALL PRIVILEGES ON mydb.* TO 'app_admin';

-- Assign role to user
GRANT 'app_read', 'app_write' TO 'appuser'@'localhost';

-- Set default role
SET DEFAULT ROLE 'app_read' TO 'appuser'@'localhost';

-- Activate role in session
SET ROLE 'app_write';

-- Show current roles
SELECT CURRENT_ROLE();

-- Revoke role
REVOKE 'app_write' FROM 'appuser'@'localhost';

-- Drop role
DROP ROLE 'app_read';
```

✅ User Management Best Practices:

1. **Never use root for applications**
2. **Principle of least privilege** - Grant minimum needed
3. **Use strong passwords** (12+ characters, mixed case, numbers, symbols)
4. **Restrict host access** (not '%' if possible)
5. **Use roles** for easier management (MySQL 8.0+)
6. **Rotate passwords regularly**

7. **Monitor failed login attempts**
8. **Remove unused accounts**
9. **Use separate users** for different applications
10. **Audit user activity**

7.2 SQL Injection Prevention

SQL injection is one of the most dangerous vulnerabilities. **Never trust user input!**

Vulnerable Code Examples
```python
# Python - VULNERABLE!
username = request.form['username']
query = f"SELECT * FROM users WHERE username = '{username}'"
cursor.execute(query)

# Attacker input: ' OR '1'='1
# Resulting query: SELECT * FROM users WHERE username = '' OR '1'='1'
# Returns ALL users!

# PHP - VULNERABLE!
$username = $_POST['username'];
$query = "SELECT * FROM users WHERE username = '$username'";
mysqli_query($conn, $query);

# JavaScript - VULNERABLE!
const username = req.body.username;
const query = `SELECT * FROM users WHERE username = '${username}'`;
connection.query(query);
```

Safe Code: Prepared Statements

Python (MySQL Connector)
```python
# SAFE: Using parameterized query
username = request.form['username']
query = "SELECT * FROM users WHERE username = %s"
cursor.execute(query, (username,))

# Multiple parameters
email = request.form['email']
password = request.form['password']
query = "INSERT INTO users (username, email, password_hash) VALUES (%s, %s, %s)"
cursor.execute(query, (username, email, hash_password(password)))
```

PHP (MySQLi)

```php
// SAFE: Prepared statement
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ?");
$stmt->bind_param("s", $username);  // "s" = string
$username = $_POST['username'];
$stmt->execute();

// Multiple parameters
$stmt = $conn->prepare("INSERT INTO users (username, email) VALUES (?, ?)");
$stmt->bind_param("ss", $username, $email);
$username = $_POST['username'];
$email = $_POST['email'];
$stmt->execute();
```

PHP (PDO)

```php
// SAFE: PDO prepared statement
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");
$stmt->execute(['username' => $_POST['username']]);

// Named parameters
$stmt = $pdo->prepare("INSERT INTO users (username, email) VALUES (:username, :email)");
$stmt->execute([
    'username' => $_POST['username'],
    'email' => $_POST['email']
]);
```

JavaScript (Node.js + MySQL2)

```javascript
// SAFE: Parameterized query
const username = req.body.username;
connection.execute(
    'SELECT * FROM users WHERE username = ?',
    [username],
    (err, results) => { ... }
);

// Multiple parameters
connection.execute(
    'INSERT INTO users (username, email) VALUES (?, ?)',
    [username, email],
    (err, results) => { ... }
);
```

Why Prepared Statements Work
```sql
-- Prepared statement process:
-- 1. Send query template to database
PREPARE stmt FROM 'SELECT * FROM users WHERE username = ?';

-- 2. Database parses and compiles query

-- 3. Send parameters separately (never interpreted as SQL)
SET @username = 'malicious\' OR \'1\'=\'1';

-- 4. Execute with parameters
EXECUTE stmt USING @username;

-- Parameters treated as data, not SQL code!
-- Query: SELECT * FROM users WHERE username = 'malicious" OR "1"="1'
-- Looks for literal username "malicious' OR '1'='1" (no injection!)
```

Additional Protection Layers

1. Input Validation
```python
# Validate input format
import re

def validate_username(username):
    # Only alphanumeric and underscore, 3-20 chars
    if not re.match(r'^[a-zA-Z0-9_]{3,20}# Complete MySQL Guide: From Basics to Industry Expert

**A Comprehensive Resource for Backend Developers**

Version 1.0 | 2025 Edition

---

## Table of Contents
```

Introduction

Welcome to the Complete MySQL Guide, designed specifically for backend developers who want to master MySQL from fundamental concepts to industry-level expertise. This comprehensive resource covers everything you need to know to become proficient in database design, query optimization, security, and production-ready MySQL deployments.

Who This Book Is For

This book is designed for:

- Backend developers wanting to strengthen their database skills
- Software engineers preparing for senior-level positions
- Database administrators looking to deepen their MySQL knowledge
- Anyone seeking to understand production-grade database management

What You'll Learn

By the end of this guide, you will:

- Master SQL syntax and advanced query techniques
- Design efficient and scalable database schemas
- Implement robust security measures
- Optimize queries for maximum performance
- Handle transactions and maintain data integrity
- Set up replication and high-availability systems
- Monitor, troubleshoot, and maintain production databases

Chapter 1: Getting Started with MySQL

1.1 What is MySQL?

MySQL is an open-source relational database management system (RDBMS) that uses Structured Query Language (SQL). It's one of the most popular databases in the world, powering everything from small websites to large-scale enterprise applications.

Key Features

- **Open Source**: Free to use with a large community
- **Cross-Platform**: Runs on Windows, Linux, macOS
- **ACID Compliant**: Ensures data integrity
- **Scalable**: Handles millions of queries per day
- **Secure**: Robust security features built-in
- **Reliable**: Proven track record in production environments
- **Fast**: Optimized for performance
- **Flexible**: Supports various storage engines

1.2 Installation and Setup

Ubuntu/Debian
````bash
# Update package index
sudo apt update

# Install MySQL Server
sudo apt install mysql-server

# Secure installation
sudo mysql_secure_installation

# Start MySQL service
sudo systemctl start mysql

# Enable on boot
sudo systemctl enable mysql

# Check status
sudo systemctl status mysql
````

macOS (using Homebrew)
````bash
# Install MySQL
brew install mysql

# Start MySQL service
brew services start mysql

# Secure installation
mysql_secure_installation
````

Windows

1. Download MySQL Installer from mysql.com

2. Run the installer
3. Choose "Developer Default" setup type
4. Complete the configuration wizard
5. Set root password

First Login
````bash
# Login as root
sudo mysql -u root -p

# Or without password on first install
sudo mysql
````

Initial Configuration
````sql
-- Check MySQL version
SELECT VERSION();

-- Show current user
SELECT USER();

-- Show databases
SHOW DATABASES;

-- Show current database
SELECT DATABASE();
````

1.3 MySQL Architecture

Understanding MySQL's architecture is crucial for optimization and troubleshooting.

Architecture Layers

1. **Connection Layer**
   - Handles client connections
   - Authentication and security
   - Thread management
   - Connection pooling

2. **SQL Layer**
   - Query parsing and validation
   - Query optimization
   - Query cache (removed in MySQL 8.0)
   - Access control

3. **Storage Engine Layer**
   - Data storage and retrieval
   - Indexing mechanisms
   - Transaction management
   - Lock management

Storage Engines

Engine	Transactions	Locking	Foreign Keys	Use Case
InnoDB	Yes	Row-level	Yes	Default, ACID-compliant, general purpose
MyISAM	No	Table-level	No	Legacy, read-heavy workloads
MEMORY	No	Table-level	No	Temporary data, caching
CSV	No	Table-level	No	Data exchange
ARCHIVE	No	Row-level	No	Historical data, logs

✅ Best Practice: Always use InnoDB for production applications. It provides ACID compliance, crash recovery, and better concurrency through row-level locking.

Check Storage Engine
````sql
-- Show default storage engine
SHOW VARIABLES LIKE 'default_storage_engine';

-- Show table storage engine
SHOW TABLE STATUS WHERE Name = 'users';

-- Change storage engine
ALTER TABLE users ENGINE = InnoDB;
````

1.4 MySQL Configuration

Important Configuration Variables
````sql
-- Show all variables
SHOW VARIABLES;

-- Show specific variable
SHOW VARIABLES LIKE 'max_connections';

-- Set session variable
SET SESSION sql_mode = 'STRICT_TRANS_TABLES';

-- Set global variable (persists until restart)
SET GLOBAL max_connections = 500;
````

my.cnf Configuration File
````ini
[mysqld]
# Basic Settings
port = 3306
datadir = /var/lib/mysql
socket = /var/lib/mysql/mysql.sock

# Character Set
character-set-server = utf8mb4
collation-server = utf8mb4_unicode_ci

# InnoDB Settings
innodb_buffer_pool_size = 1G
innodb_log_file_size = 256M
innodb_flush_log_at_trx_commit = 1
innodb_file_per_table = 1

# Connection Settings
max_connections = 500
max_connect_errors = 100
connect_timeout = 10
wait_timeout = 600

# Query Cache (MySQL 5.7 and earlier)
query_cache_type = 0
query_cache_size = 0

# Logging
slow_query_log = 1
slow_query_log_file = /var/log/mysql/slow-query.log
long_query_time = 2
log_error = /var/log/mysql/error.log
````

Chapter 2: Database Fundamentals

2.1 Creating Databases

Basic Database Operations
````sql
-- Create database
CREATE DATABASE ecommerce;
````

```sql
-- Create with character set and collation
CREATE DATABASE ecommerce
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;

-- Create if not exists
CREATE DATABASE IF NOT EXISTS ecommerce;

-- List all databases
SHOW DATABASES;

-- Select database for use
USE ecommerce;

-- Show current database
SELECT DATABASE();

-- Delete database (CAREFUL!)
DROP DATABASE ecommerce;

-- Drop if exists
DROP DATABASE IF EXISTS ecommerce;
```

✅ Best Practice: Always use `utf8mb4` character set (not `utf8`). It supports full Unicode including emojis and special characters. The `utf8mb4_unicode_ci` collation provides accurate sorting for international characters.

Database Information

```sql
-- Show database creation statement
SHOW CREATE DATABASE ecommerce;

-- Show database size
SELECT
    table_schema AS 'Database',
    ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS 'Size (MB)'
FROM information_schema.tables
GROUP BY table_schema;

-- Show tables in database
SHOW TABLES;

-- Show tables with details
SHOW TABLE STATUS;
```

2.2 Data Types

Choosing the right data type is critical for performance, storage efficiency, and data integrity.

Numeric Types

Type	Storage	Range (Signed)	Range (Unsigned)	Use Case
TINYINT	1 byte	-128 to 127	0 to 255	Boolean, small numbers
SMALLINT	2 bytes	-32,768 to 32,767	0 to 65,535	Small integers
MEDIUMINT	3 bytes	-8M to 8M	0 to 16M	Medium integers
INT	4 bytes	-2B to 2B	0 to 4B	Standard integers, IDs
BIGINT	8 bytes	-9 quintillion	0 to 18 quintillion	Large numbers, timestamps
DECIMAL(p,s)	Variable	Exact precision	Exact precision	Money, precise calculations
FLOAT	4 bytes	Approximate	Approximate	Scientific data
DOUBLE	8 bytes	Approximate	Approximate	Scientific data (avoid for money)

Examples:
````sql
CREATE TABLE numeric_examples (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    age TINYINT UNSIGNED,                -- 0-255
    quantity SMALLINT,                   -- -32,768 to 32,767
    views INT UNSIGNED,                  -- 0 to 4 billion
    user_id BIGINT UNSIGNED,             -- Large IDs
    price DECIMAL(10,2),                 -- Max 99,999,999.99
    rating DECIMAL(3,2),                 -- Max 9.99
    weight FLOAT,                        -- Approximate
    latitude DOUBLE,                     -- High precision
    is_active BOOLEAN                    -- Stored as TINYINT(1)
);
````

String Types

Type	Max Size	Storage	Use Case
CHAR(n)	255	Fixed-length	Country codes, fixed-length strings
VARCHAR(n)	65,535	Variable-length + 1-2 bytes	Most common string type
TINYTEXT	255	Variable + 1 byte	Very short text
TEXT	65,535	Variable + 2 bytes	Large text blocks
MEDIUMTEXT	16MB	Variable + 3 bytes	Articles, long content
LONGTEXT	4GB	Variable + 4 bytes	Very large text (rarely needed)
ENUM	65,535 values	1-2 bytes	Fixed set of values
SET	64 values	1-8 bytes	Multiple selections

Examples:
````sql
````

```
CREATE TABLE string_examples (
    id INT PRIMARY KEY,
    country_code CHAR(2),                  -- 'US', 'UK'
    username VARCHAR(50) NOT NULL,         -- Variable length
    email VARCHAR(100) NOT NULL,
    bio TEXT,                              -- Long description
    content MEDIUMTEXT,                    -- Article content
    status ENUM('draft', 'published', 'archived') DEFAULT 'draft',
    permissions SET('read', 'write', 'delete', 'admin')
);
````

✅ Best Practices for String Types:
- Use VARCHAR for most cases (more flexible than CHAR)
- CHAR is only better for truly fixed-length data (rare)
- Don't use TEXT types unless necessary (can't have default values, full indexing limited)
- ENUM is faster than VARCHAR for fixed sets but harder to modify
- Limit VARCHAR length appropriately (impacts memory allocation)

Date and Time Types

Type	Format	Range	Storage	Use Case
DATE	YYYY-MM-DD	1000-01-01 to 9999-12-31	3 bytes	Birth dates, deadlines
DATETIME	YYYY-MM-DD HH:MM:SS	1000 to 9999	8 bytes	Timestamps without timezone
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1970 to 2038	4 bytes	Auto-updating timestamps
TIME	HH:MM:SS	-838:59:59 to 838:59:59	3 bytes	Duration, time of day
YEAR	YYYY	1901 to 2155	1 byte	Year only

Examples:
````sql
CREATE TABLE datetime_examples (
    id INT PRIMARY KEY,
    birth_date DATE,                       -- '1990-05-15'
    appointment_time TIME,                 -- '14:30:00'
    event_datetime DATETIME,               -- '2024-01-15 14:30:00'
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    year_manufactured YEAR                 -- 2024
);

-- Date/Time functions
SELECT
    NOW(),                                 -- Current datetime
    CURDATE(),                             -- Current date
    CURTIME(),                             -- Current time
    DATE_FORMAT(NOW(), '%Y-%m-%d'),        -- Formatted date
```

```
    DATE_ADD(NOW(), INTERVAL 7 DAY),       -- Add days
    DATEDIFF(NOW(), '2024-01-01'),         -- Days between
    YEAR(NOW()),                           -- Extract year
    MONTH(NOW()),                          -- Extract month
    DAY(NOW());                            -- Extract day
````

TIMESTAMP vs DATETIME:
- TIMESTAMP: Stores as Unix timestamp, converts to timezone, auto-updates
- DATETIME: Stores as is, no timezone conversion
- Use TIMESTAMP for "when did this happen" fields
- Use DATETIME for "when should this happen" fields

Binary Types

Type	Max Size	Use Case
BINARY(n)	255	Fixed binary data
VARBINARY(n)	65,535	Variable binary data
TINYBLOB	255	Small binary objects
BLOB	65KB	Binary objects
MEDIUMBLOB	16MB	Images, files
LONGBLOB	4GB	Large files

````sql
CREATE TABLE binary_examples (
    id INT PRIMARY KEY,
    file_hash BINARY(32),                  -- SHA-256 hash
    thumbnail BLOB,                        -- Small image
    document MEDIUMBLOB                    -- PDF file
);
````

⚠️ Warning: Storing files in database is generally NOT recommended. Use file storage (S3, local filesystem) and store file paths in database instead.

JSON Type (MySQL 5.7+)

````sql
CREATE TABLE json_examples (
    id INT PRIMARY KEY,
    metadata JSON,
    settings JSON
);

-- Insert JSON
INSERT INTO json_examples (id, metadata)
VALUES (1, '{"name": "John", "age": 30, "tags": ["developer", "mysql"]}');
````

```
-- Query JSON
SELECT
    id,
    JSON_EXTRACT(metadata, '$.name') as name,
    metadata->'$.age' as age,
    metadata->>'$.age' as age_text
FROM json_examples;

-- JSON functions
SELECT
    JSON_KEYS(metadata),
    JSON_LENGTH(metadata),
    JSON_CONTAINS(metadata, '"developer"', '$.tags')
FROM json_examples;
````

2.3 Creating Tables

Basic Table Creation
````sql
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    phone VARCHAR(20),
    date_of_birth DATE,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

    INDEX idx_email (email),
    INDEX idx_username (username),
    INDEX idx_name (last_name, first_name)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
````

Constraints
````sql
CREATE TABLE products (
    id INT UNSIGNED AUTO_INCREMENT,
    sku VARCHAR(50) NOT NULL,
    name VARCHAR(200) NOT NULL,
    description TEXT,
    price DECIMAL(10,2) NOT NULL,
```

```
    stock_quantity INT NOT NULL DEFAULT 0,
    category_id INT UNSIGNED,

    -- Primary Key
    PRIMARY KEY (id),

    -- Unique Constraint
    UNIQUE KEY unique_sku (sku),

    -- Check Constraint (MySQL 8.0.16+)
    CHECK (price >= 0),
    CHECK (stock_quantity >= 0),

    -- Foreign Key
    FOREIGN KEY (category_id)
        REFERENCES categories(id)
        ON DELETE SET NULL
        ON UPDATE CASCADE,

    -- Indexes
    INDEX idx_category (category_id),
    INDEX idx_name (name),
    FULLTEXT INDEX idx_description (description)
) ENGINE=InnoDB;
````

Foreign Key Actions

Action	Description
CASCADE	Delete/update parent = delete/update children
SET NULL	Delete/update parent = set child FK to NULL
RESTRICT	Prevent delete/update if children exist (default)
NO ACTION	Same as RESTRICT
SET DEFAULT	Set to default value (not supported in InnoDB)

````sql
CREATE TABLE orders (
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    user_id INT UNSIGNED NOT NULL,
    order_number VARCHAR(20) NOT NULL UNIQUE,
    total_amount DECIMAL(10,2) NOT NULL,
    status ENUM('pending', 'processing', 'shipped', 'delivered', 'cancelled')
        DEFAULT 'pending',
    notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
```

```
    -- Foreign key with CASCADE
    FOREIGN KEY (user_id)
        REFERENCES users(id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE,

    -- Composite index for user's orders sorted by date
    INDEX idx_user_orders (user_id, created_at),
    INDEX idx_status (status),
    INDEX idx_order_number (order_number)
) ENGINE=InnoDB;

CREATE TABLE order_items (
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    order_id BIGINT UNSIGNED NOT NULL,
    product_id INT UNSIGNED NOT NULL,
    quantity INT UNSIGNED NOT NULL,
    price DECIMAL(10,2) NOT NULL,

    -- Cascade delete when order is deleted
    FOREIGN KEY (order_id)
        REFERENCES orders(id)
        ON DELETE CASCADE,

    -- Restrict if product is in orders
    FOREIGN KEY (product_id)
        REFERENCES products(id)
        ON DELETE RESTRICT,

    INDEX idx_order (order_id),
    INDEX idx_product (product_id)
) ENGINE=InnoDB;
````

Table Alteration
````sql
-- Add column
ALTER TABLE users ADD COLUMN middle_name VARCHAR(50) AFTER first_name;

-- Add column with constraint
ALTER TABLE users ADD COLUMN age TINYINT UNSIGNED CHECK (age >= 18);

-- Modify column type
ALTER TABLE users MODIFY COLUMN phone VARCHAR(25);

-- Change column name and type
ALTER TABLE users CHANGE COLUMN phone phone_number VARCHAR(25);
```

```sql
-- Drop column
ALTER TABLE users DROP COLUMN middle_name;

-- Add index
ALTER TABLE users ADD INDEX idx_last_name (last_name);

-- Add unique constraint
ALTER TABLE users ADD UNIQUE KEY unique_phone (phone_number);

-- Add foreign key
ALTER TABLE orders
ADD CONSTRAINT fk_user
FOREIGN KEY (user_id) REFERENCES users(id);

-- Drop foreign key
ALTER TABLE orders DROP FOREIGN KEY fk_user;

-- Drop index
ALTER TABLE users DROP INDEX idx_last_name;

-- Rename table
ALTER TABLE users RENAME TO customers;
RENAME TABLE customers TO users;

-- Change storage engine
ALTER TABLE users ENGINE = InnoDB;

-- Add AUTO_INCREMENT
ALTER TABLE users MODIFY id INT UNSIGNED AUTO_INCREMENT;

-- Reset AUTO_INCREMENT
ALTER TABLE users AUTO_INCREMENT = 1000;
````

⚠️ Note: ALTER TABLE operations can lock the table on large datasets. For production databases with large tables, use tools like:
- `pt-online-schema-change` (Percona Toolkit)
- `gh-ost` (GitHub's online schema migration)
- MySQL 8.0's instant ADD COLUMN feature

Creating Tables from Queries
````sql
-- Create table from SELECT
CREATE TABLE active_users AS
SELECT id, username, email, created_at
FROM users
````

```
WHERE is_active = TRUE;

-- Create table structure only (no data)
CREATE TABLE users_backup LIKE users;

-- Copy data to backup table
INSERT INTO users_backup SELECT * FROM users;
```

Temporary Tables
```sql
-- Create temporary table (auto-deleted when session ends)
CREATE TEMPORARY TABLE temp_results (
    id INT,
    total DECIMAL(10,2)
);

-- Use temporary table
INSERT INTO temp_results
SELECT user_id, SUM(total_amount)
FROM orders
GROUP BY user_id;

SELECT * FROM temp_results;
-- Table automatically dropped when connection closes
```

Partitioning
```sql
-- Range partitioning by date
CREATE TABLE orders_partitioned (
    id BIGINT UNSIGNED AUTO_INCREMENT,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10,2),
    PRIMARY KEY (id, order_date)
) PARTITION BY RANGE (YEAR(order_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
);

-- List partitioning
CREATE TABLE users_region (
    id INT,
    name VARCHAR(50),
    region VARCHAR(20),
```

```
    PRIMARY KEY (id, region)
) PARTITION BY LIST COLUMNS(region) (
    PARTITION p_north VALUES IN('North', 'Northeast', 'Northwest'),
    PARTITION p_south VALUES IN('South', 'Southeast', 'Southwest'),
    PARTITION p_east VALUES IN('East'),
    PARTITION p_west VALUES IN('West')
);

-- Hash partitioning
CREATE TABLE logs (
    id BIGINT AUTO_INCREMENT,
    log_data TEXT,
    PRIMARY KEY (id)
) PARTITION BY HASH(id)
PARTITIONS 4;
````

Chapter 3: CRUD Operations

3.1 INSERT Operations

Basic INSERT
```sql
-- Single row insert
INSERT INTO users (username, email, password_hash, first_name, last_name)
VALUES ('johndoe', 'john@example.com', 'hashed_password', 'John', 'Doe');

-- Get last inserted ID
SELECT LAST_INSERT_ID();

-- Multiple rows insert (much more efficient)
INSERT INTO users (username, email, password_hash)
VALUES
    ('alice', 'alice@example.com', 'hash1'),
    ('bob', 'bob@example.com', 'hash2'),
    ('charlie', 'charlie@example.com', 'hash3');
````

✅ Best Practice: Use batch inserts instead of individual inserts. Inserting 1000 rows in one statement is typically 10-100x faster than 1000 separate INSERT statements.

INSERT with All Columns
```sql
-- If providing values for all columns in order
INSERT INTO users VALUES
```

```
    (NULL, 'dave', 'dave@example.com', 'hash4', 'Dave', 'Smith', '555-0100', '1990-01-01', TRUE, NOW(), NOW());
````

⚠️ Warning: This method is fragile. If table structure changes, INSERT breaks. Always specify column names explicitly.

INSERT from SELECT
````sql
-- Copy data from another table
INSERT INTO archived_users (id, username, email, created_at)
SELECT id, username, email, created_at
FROM users
WHERE created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR);

-- Insert with transformation
INSERT INTO user_summary (user_id, total_orders, total_spent)
SELECT
    user_id,
    COUNT(*) as order_count,
    SUM(total_amount) as total
FROM orders
GROUP BY user_id;
````

INSERT IGNORE
````sql
-- Ignore errors (like duplicate keys)
INSERT IGNORE INTO users (username, email, password_hash)
VALUES ('johndoe', 'john@example.com', 'hash');
-- If username 'johndoe' exists, this silently fails

-- Check affected rows
SELECT ROW_COUNT();  -- Returns 0 if ignored, 1 if inserted
````

⚠️ Caution: INSERT IGNORE suppresses ALL errors, not just duplicates. Use carefully.

INSERT ... ON DUPLICATE KEY UPDATE (Upsert)
````sql
-- Update if duplicate key found
INSERT INTO product_inventory (product_id, quantity)
VALUES (101, 50)
ON DUPLICATE KEY UPDATE
    quantity = quantity + VALUES(quantity),
    updated_at = CURRENT_TIMESTAMP;

-- More complex upsert
```

````
INSERT INTO user_stats (user_id, login_count, last_login)
VALUES (1, 1, NOW())
ON DUPLICATE KEY UPDATE
    login_count = login_count + 1,
    last_login = NOW();
````

REPLACE (Delete and Insert)
````sql
-- Delete existing row and insert new one
REPLACE INTO settings (key_name, value)
VALUES ('theme', 'dark');

-- Equivalent to:
-- DELETE FROM settings WHERE key_name = 'theme';
-- INSERT INTO settings (key_name, value) VALUES ('theme', 'dark');
````

⚠️ Warning: REPLACE deletes the old row (if exists) and inserts new one. This can affect:
- Foreign key relationships
- AUTO_INCREMENT values
- Triggers

INSERT with DEFAULT VALUES
````sql
-- Use default values explicitly
INSERT INTO users (username, email, password_hash, is_active)
VALUES ('newuser', 'new@example.com', 'hash', DEFAULT);

-- Insert row with all defaults
INSERT INTO log_entries () VALUES ();
````

3.2 SELECT Queries

Basic SELECT
````sql
-- Select specific columns
SELECT id, username, email FROM users;

-- Select all columns (avoid in production)
SELECT * FROM users;

-- With WHERE clause
SELECT * FROM users
WHERE is_active = TRUE;
````

```sql
-- Multiple conditions
SELECT * FROM users
WHERE is_active = TRUE
  AND created_at > '2024-01-01'
  AND email LIKE '%@gmail.com';

-- OR condition
SELECT * FROM users
WHERE city = 'New York'
  OR city = 'Los Angeles';

-- IN operator (cleaner than multiple ORs)
SELECT * FROM users
WHERE city IN ('New York', 'Los Angeles', 'Chicago');

-- NOT IN
SELECT * FROM users
WHERE status NOT IN ('banned', 'deleted');

-- BETWEEN
SELECT * FROM orders
WHERE total_amount BETWEEN 100 AND 500;

-- IS NULL / IS NOT NULL
SELECT * FROM users
WHERE phone IS NULL;

SELECT * FROM users
WHERE phone IS NOT NULL;
````

Pattern Matching
````sql
-- LIKE operator (% = any characters, _ = single character)
SELECT * FROM users
WHERE email LIKE '%@gmail.com';

SELECT * FROM products
WHERE name LIKE 'iPhone%';

SELECT * FROM users
WHERE phone LIKE '555-____';

-- Case-insensitive by default in MySQL
SELECT * FROM products
WHERE name LIKE '%laptop%';
```

```sql
-- Case-sensitive LIKE using BINARY
SELECT * FROM products
WHERE BINARY name LIKE '%Laptop%';

-- REGEXP/RLIKE for complex patterns
SELECT * FROM users
WHERE email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$';

SELECT * FROM users
WHERE phone REGEXP '^[0-9]{3}-[0-9]{3}-[0-9]{4}$';

-- NOT LIKE
SELECT * FROM users
WHERE email NOT LIKE '%@tempmail.com';
````

Performance Note: LIKE with leading wildcard (`%value`) cannot use index. Prefer `value%` when possible.

Sorting
````sql
-- ORDER BY ascending
SELECT * FROM products
ORDER BY price ASC;

-- ORDER BY descending
SELECT * FROM products
ORDER BY price DESC;

-- Multiple sort columns
SELECT * FROM users
ORDER BY last_name ASC, first_name ASC;

-- Sort by expression
SELECT * FROM products
ORDER BY (price * quantity) DESC;

-- Custom sort order with CASE
SELECT * FROM orders
ORDER BY
  CASE status
    WHEN 'urgent' THEN 1
    WHEN 'processing' THEN 2
    WHEN 'pending' THEN 3
    ELSE 4
  END;

-- NULL values first or last
```

```sql
SELECT * FROM users
ORDER BY phone IS NULL, phone;  -- NULLs first

SELECT * FROM users
ORDER BY phone IS NULL DESC, phone;  -- NULLs last

-- Random order
SELECT * FROM products
ORDER BY RAND()
LIMIT 10;
````

⚠️ Warning: `ORDER BY RAND()` is slow on large tables. Use alternative methods for random selection.

Limiting Results

````sql
-- LIMIT
SELECT * FROM products
ORDER BY created_at DESC
LIMIT 10;

-- LIMIT with OFFSET (pagination)
SELECT * FROM products
ORDER BY created_at DESC
LIMIT 20 OFFSET 40;  -- Page 3, 20 items per page

-- Alternative syntax
SELECT * FROM products
ORDER BY created_at DESC
LIMIT 40, 20;  -- LIMIT offset, count

-- Get top N per category (MySQL 8.0+)
SELECT * FROM (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY price DESC) as rn
    FROM products
) t
WHERE rn <= 5;
````

Pagination Best Practice: For large offsets, use cursor-based pagination:

````sql
-- Instead of OFFSET 10000 (slow)
SELECT * FROM products
WHERE id > 10000
ORDER BY id
LIMIT 20;
```

````

DISTINCT
````sql
-- Get unique values
SELECT DISTINCT city FROM users;

-- Count unique values
SELECT COUNT(DISTINCT city) FROM users;

-- Multiple columns
SELECT DISTINCT city, state FROM users;

-- DISTINCT with aggregates
SELECT
    city,
    COUNT(DISTINCT email) as unique_emails
FROM users
GROUP BY city;
````

Column Aliases
````sql
-- AS keyword (optional but recommended)
SELECT
    first_name AS fname,
    last_name AS lname,
    CONCAT(first_name, ' ', last_name) AS full_name,
    YEAR(CURDATE()) - YEAR(date_of_birth) AS age
FROM users;

-- Without AS (works but less clear)
SELECT
    first_name fname,
    last_name lname
FROM users;

-- Table aliases
SELECT
    u.id,
    u.username,
    o.order_number
FROM users u
INNER JOIN orders o ON u.id = o.user_id;
````

Calculated Fields
````

````sql
SELECT
    product_name,
    price,
    tax_rate,
    price * tax_rate AS tax_amount,
    price + (price * tax_rate) AS total_price,
    ROUND(price * 0.9, 2) AS discounted_price
FROM products;

SELECT
    first_name,
    last_name,
    CONCAT(first_name, ' ', last_name) AS full_name,
    UPPER(email) AS email_upper,
    CHAR_LENGTH(username) AS username_length,
    DATE_FORMAT(created_at, '%M %d, %Y') AS formatted_date
FROM users;
````

CASE Statements
````sql
-- Simple CASE
SELECT
    product_name,
    stock_quantity,
    CASE
        WHEN stock_quantity = 0 THEN 'Out of Stock'
        WHEN stock_quantity < 10 THEN 'Low Stock'
        WHEN stock_quantity < 50 THEN 'In Stock'
        ELSE 'Well Stocked'
    END AS stock_status
FROM products;

-- CASE in calculations
SELECT
    order_id,
    total_amount,
    CASE
        WHEN total_amount > 1000 THEN total_amount * 0.9
        WHEN total_amount > 500 THEN total_amount * 0.95
        ELSE total_amount
    END AS final_price
FROM orders;

-- Searched CASE
SELECT
````

```
    username,
    CASE
        WHEN login_count > 100 THEN 'Power User'
        WHEN login_count > 50 THEN 'Regular User'
        WHEN login_count > 10 THEN 'Casual User'
        ELSE 'New User'
    END AS user_type
FROM user_stats;
````

IF Function
````sql
-- Simple IF
SELECT
    username,
    IF(is_active, 'Active', 'Inactive') AS status
FROM users;

-- Nested IF
SELECT
    product_name,
    price,
    IF(price > 100,
        'Premium',
        IF(price > 50, 'Standard', 'Budget')
    ) AS price_tier
FROM products;
````

COALESCE and IFNULL
````sql
-- Return first non-NULL value
SELECT
    username,
    COALESCE(phone, email, 'No contact info') AS contact
FROM users;

-- IFNULL (only 2 arguments)
SELECT
    product_name,
    IFNULL(discount_price, price) AS final_price
FROM products;

-- NULLIF (returns NULL if equal)
SELECT
    username,
    NULLIF(bio, '') AS bio  -- Returns NULL if bio is empty string
```

````
FROM users;
````

3.3 UPDATE Statements

Basic UPDATE
````sql
-- Update single column
UPDATE users
SET email = 'newemail@example.com'
WHERE id = 1;

-- Update multiple columns
UPDATE users
SET
    first_name = 'John',
    last_name = 'Smith',
    updated_at = CURRENT_TIMESTAMP
WHERE id = 1;

-- Update based on condition
UPDATE products
SET is_featured = TRUE
WHERE rating > 4.5 AND stock_quantity > 0;
````

🚨 CRITICAL WARNING: Always use WHERE clause with UPDATE! Without it, ALL rows will be updated!
````sql
-- Enable safe updates mode (recommended for development)
SET sql_safe_updates = 1;

-- This will now error if you forget WHERE
UPDATE users SET email = 'test@example.com';  -- ERROR!
````

UPDATE with Calculations
````sql
-- Increment value
UPDATE product_inventory
SET quantity = quantity + 10
WHERE product_id = 101;

-- Decrement
UPDATE users
SET login_count = login_count - 1
WHERE id = 5;
````

```sql
-- Apply percentage
UPDATE products
SET price = price * 1.1  -- 10% increase
WHERE category = 'Electronics';

-- Apply discount with ROUND
UPDATE products
SET price = ROUND(price * 0.9, 2)
WHERE category = 'Clearance';
````

UPDATE with CASE

````sql
-- Conditional updates
UPDATE products
SET status = CASE
    WHEN stock_quantity = 0 THEN 'out_of_stock'
    WHEN stock_quantity < 10 THEN 'low_stock'
    ELSE 'in_stock'
END;

-- Multiple columns with CASE
UPDATE employees
SET
    salary = CASE
        WHEN performance_rating = 5 THEN salary * 1.15
        WHEN performance_rating = 4 THEN salary * 1.10
        WHEN performance_rating = 3 THEN salary * 1.05
        ELSE salary
    END,
    bonus = CASE
        WHEN performance_rating >= 4 THEN salary * 0.2
        ELSE 0
    END
WHERE review_date = CURDATE();
````

UPDATE with JOIN

````sql
-- Update based on related table
UPDATE orders o
INNER JOIN users u ON o.user_id = u.id
SET o.customer_name = CONCAT(u.first_name, ' ', u.last_name)
WHERE o.customer_name IS NULL;

-- Update with multiple joins
UPDATE products p
```

```
INNER JOIN categories c ON p.category_id = c.id
INNER JOIN suppliers s ON p.supplier_id = s.id
SET p.full_description = CONCAT(c.name, ' - ', p.name, ' by ', s.name);

-- Update with LEFT JOIN (update all from left table)
UPDATE users u
LEFT JOIN orders o ON u.id = o.user_id AND o.status = 'completed'
SET u.has_completed_order = IF(o.id IS NOT NULL, TRUE, FALSE);
````

UPDATE with Subquery
````sql
-- Update based on aggregate from another table
UPDATE users u
SET u.total_spent = (
    SELECT COALESCE(SUM(total_amount), 0)
    FROM orders
    WHERE user_id = u.id AND status = 'completed'
);

-- Update using subquery condition
UPDATE products
SET is_bestseller = TRUE
WHERE id IN (
    SELECT product_id
    FROM order_items
    GROUP BY product_id
    HAVING SUM(quantity) > 1000
);

-- Update to average value
UPDATE products
SET price = (
    SELECT AVG(price)
    FROM products p2
    WHERE p2.category_id = products.category_id
)
WHERE price IS NULL;
````

UPDATE with LIMIT
````sql
-- Update only first N rows
UPDATE products
SET is_promoted = TRUE
WHERE is_promoted = FALSE
ORDER BY rating DESC
```

````
LIMIT 10;

-- Useful for batch updates
UPDATE large_table
SET processed = TRUE
WHERE processed = FALSE
LIMIT 1000;
````

UPDATE with ORDER BY
````sql
-- Update in specific order (useful with LIMIT)
UPDATE users
SET membership_tier = 'gold'
WHERE membership_tier = 'silver'
ORDER BY total_spent DESC
LIMIT 100;
````

3.4 DELETE Operations

Basic DELETE
````sql
-- Delete specific rows
DELETE FROM users
WHERE id = 1;

-- Delete with multiple conditions
DELETE FROM logs
WHERE created_at < DATE_SUB(NOW(), INTERVAL 90 DAY)
  AND status = 'archived';

-- Delete all rows (keep table structure)
DELETE FROM temp_data;
````

🚨 CRITICAL WARNING: Always use WHERE with DELETE! Enable safe updates:
````sql
SET sql_safe_updates = 1;
````

DELETE with JOIN
````sql
-- Delete from single table based on join
DELETE o
FROM orders o
INNER JOIN users u ON o.user_id = u.id
````

```
WHERE u.status = 'deleted';

-- Delete from multiple tables
DELETE o, oi
FROM orders o
INNER JOIN order_items oi ON o.id = oi.order_id
WHERE o.status = 'cancelled'
  AND o.created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR);

-- Delete using LEFT JOIN (orphaned records)
DELETE oi
FROM order_items oi
LEFT JOIN orders o ON oi.order_id = o.id
WHERE o.id IS NULL;
````

DELETE with Subquery
````sql
-- Delete based on subquery
DELETE FROM products
WHERE id IN (
  SELECT product_id
  FROM discontinued_items
);

-- Delete based on aggregate condition
DELETE FROM users
WHERE id IN (
  SELECT user_id
  FROM orders
  GROUP BY user_id
  HAVING COUNT(*) = 0
) AND created_at < DATE_SUB(NOW(), INTERVAL 2 YEAR);
````

DELETE with LIMIT
````sql
-- Delete in batches (safer for large tables)
DELETE FROM logs
WHERE created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR)
LIMIT 10000;

-- Can run multiple times until all deleted
-- Check affected rows
SELECT ROW_COUNT();
````

TRUNCATE vs DELETE

````sql
-- DELETE: Slower, logged, can have WHERE, can ROLLBACK
DELETE FROM temp_data WHERE id < 1000;

-- TRUNCATE: Fast, minimal logging, removes ALL rows, resets AUTO_INCREMENT
TRUNCATE TABLE temp_data;
````

TRUNCATE vs DELETE comparison:

Feature	DELETE	TRUNCATE
WHERE clause	✅ Yes	❌ No
Speed	Slower	Faster
Rollback	✅ Yes	❌ No (in most cases)
Triggers	✅ Fires	❌ Doesn't fire
Reset AUTO_INCREMENT	❌ No	✅ Yes
Use case	Selective deletion	Clear entire table

Soft Deletes (Best Practice)

Instead of physically deleting data, use soft deletes:

````sql
-- Add deleted_at column
ALTER TABLE users
ADD COLUMN deleted_at TIMESTAMP NULL DEFAULT NULL;

-- Add index for performance
CREATE INDEX idx_deleted_at ON users(deleted_at);

-- Soft delete
UPDATE users
SET deleted_at = CURRENT_TIMESTAMP
WHERE id = 1;

-- Query only active records
SELECT * FROM users
WHERE deleted_at IS NULL;

-- Include deleted records
SELECT * FROM users;

-- Restore soft-deleted record
UPDATE users
SET deleted_at = NULL
WHERE id = 1;
````

```
-- Permanently delete (hard delete)
DELETE FROM users
WHERE deleted_at IS NOT NULL
    AND deleted_at < DATE_SUB(NOW(), INTERVAL 30 DAY);
````

✅ Benefits of Soft Deletes:
- Data recovery possible
- Audit trail maintained
- Foreign key relationships preserved
- Can analyze deleted data
- Gradual cleanup possible

❌ Considerations:
- Queries must filter `deleted_at IS NULL`
- Indexes include deleted rows
- Storage continues to grow
- Unique constraints need adjustment
````sql
-- Create view for active records
CREATE VIEW active_users AS
SELECT * FROM users
WHERE deleted_at IS NULL;

-- Use view in queries
SELECT * FROM active_users WHERE city = 'New York';
````

Chapter 4: Advanced Queries

4.1 JOINs Mastery

JOINs combine rows from two or more tables based on related columns.

INNER JOIN

Returns only rows that have matching values in both tables.
````sql
-- Basic INNER JOIN
SELECT
    u.id,
    u.username,
    o.order_number,
    o.total_amount,
````

```
    o.created_at AS order_date
FROM users u
INNER JOIN orders o ON u.id = o.user_id
WHERE o.status = 'delivered'
ORDER BY o.created_at DESC;

-- Multiple conditions in JOIN
SELECT
    u.username,
    o.order_number
FROM users u
INNER JOIN orders o
    ON u.id = o.user_id
    AND o.created_at >= '2024-01-01'
    AND o.status != 'cancelled';

-- Using USING (when column names match)
SELECT
    u.username,
    o.order_number
FROM users u
INNER JOIN orders o USING(user_id);
````

LEFT JOIN (LEFT OUTER JOIN)

Returns all rows from left table and matched rows from right. NULL for non-matches.

````sql
-- Find all users and their order count (including users with no orders)
SELECT
    u.id,
    u.username,
    COUNT(o.id) AS order_count,
    COALESCE(SUM(o.total_amount), 0) AS total_spent
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.username;

-- Find users with NO orders
SELECT
    u.id,
    u.username,
    u.email
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE o.id IS NULL;
```

```sql
-- Find products never ordered
SELECT p.id, p.product_name
FROM products p
LEFT JOIN order_items oi ON p.id = oi.product_id
WHERE oi.id IS NULL;
```

#### RIGHT JOIN (RIGHT OUTER JOIN)

Returns all rows from right table and matched rows from left.

```sql
-- All orders with user info (even if user deleted)
SELECT
    o.order_number,
    o.total_amount,
    u.username,
    u.email
FROM users u
RIGHT JOIN orders o ON u.id = o.user_id;

-- Note: RIGHT JOIN is less common; can always be rewritten as LEFT JOIN
```

#### FULL OUTER JOIN

MySQL doesn't support FULL OUTER JOIN directly. Use UNION:

```sql
-- Simulate FULL OUTER JOIN
SELECT u.id, u.username, o.order_number
FROM users u
LEFT JOIN orders o ON u.id = o.user_id

UNION

SELECT u.id, u.username, o.order_number
FROM users u
RIGHT JOIN orders o ON u.id = o.user_id;
```

#### CROSS JOIN

Cartesian product - every row from table1 paired with every row from table2.

```sql
-- Generate all combinations
SELECT
    c.color_name,
    s.size_name,
```

```
    CONCAT(c.color_name, ' - ', s.size_name) AS variant
FROM colors c
CROSS JOIN sizes s;

-- Useful for generating test data
SELECT
    CONCAT('user', n1.num, n2.num) AS username
FROM
    (SELECT 0 AS num UNION SELECT 1 UNION SELECT 2) n1
CROSS JOIN
    (SELECT 0 AS num UNION SELECT 1 UNION SELECT 2) n2;
````

SELF JOIN

Join table to itself.

````sql
-- Employee hierarchy
SELECT
    e.name AS employee,
    m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.id;

-- Find users from same city
SELECT
    u1.username AS user1,
    u2.username AS user2,
    u1.city
FROM users u1
INNER JOIN users u2
    ON u1.city = u2.city
    AND u1.id < u2.id  -- Avoid duplicates
ORDER BY u1.city;

-- Find products in same price range
SELECT
    p1.product_name AS product1,
    p2.product_name AS product2,
    p1.price
FROM products p1
INNER JOIN products p2
    ON ABS(p1.price - p2.price) < 10
    AND p1.id < p2.id;
````

Multiple JOINs

````sql
-- Complex query with multiple joins
SELECT
    o.order_number,
    u.username,
    u.email,
    p.product_name,
    c.category_name,
    oi.quantity,
    oi.price,
    (oi.quantity * oi.price) AS line_total
FROM orders o
INNER JOIN users u ON o.user_id = u.id
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
INNER JOIN categories c ON p.category_id = c.id
WHERE o.status = 'delivered'
    AND o.created_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
ORDER BY o.created_at DESC;

-- Mix of LEFT and INNER JOINs
SELECT
    u.username,
    u.email,
    COUNT(DISTINCT o.id) AS order_count,
    COUNT(DISTINCT r.id) AS review_count,
    AVG(r.rating) AS avg_rating
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
LEFT JOIN reviews r ON u.id = r.user_id
GROUP BY u.id, u.username, u.email
HAVING order_count > 0 OR review_count > 0;
````

Natural JOIN (Use with Caution)
````sql
-- Automatically joins on all columns with same name
SELECT *
FROM users
NATURAL JOIN orders;

-- ⚠️ Dangerous! Can produce unexpected results if schema changes
-- Better to be explicit with JOIN conditions
````

✅ JOIN Best Practices:

1. **Always use explicit JOIN syntax** (not WHERE clause joins)
2. **Index foreign key columns** for better performance
3. **Use table aliases** for readability
4. **Consider query execution order**: WHERE → JOIN → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
5. **Use EXPLAIN** to understand join strategies
6. **Avoid joining too many tables** (consider denormalization or summary tables)
7. **Filter early**: Put conditions in WHERE, not SELECT

````sql
-- Bad: Filtering in SELECT
SELECT
    u.username,
    IF(o.status = 'completed', o.total_amount, 0) AS completed_amount
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;

-- Good: Filter in JOIN condition
SELECT
    u.username,
    o.total_amount
FROM users u
LEFT JOIN orders o
    ON u.id = o.user_id
    AND o.status = 'completed';
````

4.2 Subqueries

A query nested inside another query.

Subquery in WHERE Clause

````sql
-- Find users who placed orders above average
SELECT username, email
FROM users
WHERE id IN (
    SELECT DISTINCT user_id
    FROM orders
    WHERE total_amount > (
        SELECT AVG(total_amount) FROM orders
    )
);

-- Find products more expensive than average in their category
SELECT
    product_name,
    price,
````

```
    category_id
FROM products p
WHERE price > (
    SELECT AVG(price)
    FROM products
    WHERE category_id = p.category_id
);
````

Subquery in SELECT Clause
````sql
-- Add calculated columns with subqueries
SELECT
    u.id,
    u.username,
    (SELECT COUNT(*) FROM orders WHERE user_id = u.id) AS total_orders,
    (SELECT SUM(total_amount) FROM orders WHERE user_id = u.id AND status = 'completed') AS lifetime_value,
    (SELECT MAX(created_at) FROM orders WHERE user_id = u.id) AS last_order_date
FROM users u
WHERE u.is_active = TRUE;
````

⚠️ Performance Warning: Scalar subqueries in SELECT execute once per row. Use JOINs for better performance:
````sql
-- Better performance with JOIN and GROUP BY
SELECT
    u.id,
    u.username,
    COUNT(o.id) AS total_orders,
    SUM(CASE WHEN o.status = 'completed' THEN o.total_amount ELSE 0 END) AS lifetime_value,
    MAX(o.created_at) AS last_order_date
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.is_active = TRUE
GROUP BY u.id, u.username;
````

Subquery in FROM Clause (Derived Table)
````sql
-- Use subquery as temporary table
SELECT
    category,
    AVG(product_count) AS avg_products_per_supplier
FROM (
    SELECT
        category_id AS category,
        supplier_id,
```

```
        COUNT(*) AS product_count
    FROM products
    GROUP BY category_id, supplier_id
) AS category_supplier_stats
GROUP BY category;

-- Find top customers by month
SELECT
    order_month,
    username,
    total_spent
FROM (
    SELECT
        DATE_FORMAT(o.created_at, '%Y-%m') AS order_month,
        u.username,
        SUM(o.total_amount) AS total_spent,
        RANK() OVER (
            PARTITION BY DATE_FORMAT(o.created_at, '%Y-%m')
            ORDER BY SUM(o.total_amount) DESC
        ) AS monthly_rank
    FROM orders o
    INNER JOIN users u ON o.user_id = u.id
    GROUP BY DATE_FORMAT(o.created_at, '%Y-%m'), u.id, u.username
) AS monthly_rankings
WHERE monthly_rank <= 10;
````

Correlated Subquery

Subquery references columns from outer query (executes once per outer row).

````sql
-- Find products priced above their category average
SELECT
    p.product_name,
    p.price,
    (SELECT AVG(price) FROM products WHERE category_id = p.category_id) AS category_avg
FROM products p
WHERE p.price > (
    SELECT AVG(price)
    FROM products
    WHERE category_id = p.category_id
);

-- Find users with more than 5 orders
SELECT username
FROM users u
WHERE (
```

```
    SELECT COUNT(*)
    FROM orders
    WHERE user_id = u.id
) > 5;
````

EXISTS and NOT EXISTS

More efficient than IN for checking existence.

````sql
-- EXISTS: Stops at first match
SELECT username
FROM users u
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.id
        AND o.total_amount > 1000
);

-- NOT EXISTS: Find users without orders
SELECT username
FROM users u
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.id
);

-- Complex EXISTS example
SELECT p.product_name
FROM products p
WHERE EXISTS (
    SELECT 1
    FROM order_items oi
    INNER JOIN orders o ON oi.order_id = o.id
    WHERE oi.product_id = p.id
        AND o.created_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
        AND o.status = 'delivered'
);
````

IN vs EXISTS

````sql
-- IN: Evaluates subquery completely, stores results
SELECT username
FROM users
```

```
WHERE id IN (
    SELECT user_id
    FROM orders
    WHERE total_amount > 1000
);

-- EXISTS: Short-circuits at first match (usually faster)
SELECT username
FROM users u
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.id
        AND o.total_amount > 1000
);
````

Performance Guideline:
- Use EXISTS when subquery returns many rows
- Use IN when subquery returns few rows or constant list
- EXISTS stops at first match (faster for correlated queries)
- IN must evaluate all results before comparison

ALL, ANY, SOME
```sql
-- ANY/SOME: True if comparison is true for any value
SELECT product_name, price
FROM products
WHERE price > ANY (
    SELECT price
    FROM products
    WHERE category_id = 5
);

-- ALL: True if comparison is true for all values
SELECT product_name, price
FROM products
WHERE price > ALL (
    SELECT price
    FROM products
    WHERE category_id = 5
);

-- Equivalent to
SELECT product_name, price
FROM products
WHERE price > (
```

```
    SELECT MAX(price)
    FROM products
    WHERE category_id = 5
);
````

✅ Subquery Best Practices:

1. **Use JOINs instead of subqueries when possible** (usually faster)
2. **Use EXISTS instead of IN** for correlated subqueries
3. **Avoid subqueries in SELECT clause** (executes per row)
4. **Index columns used in subquery conditions**
5. **Use EXPLAIN to compare performance** between subqueries and JOINs

4.3 Aggregate Functions

Perform calculations on sets of rows.

Basic Aggregates
```sql
-- COUNT
SELECT COUNT(*) AS total_users FROM users;
SELECT COUNT(DISTINCT city) AS unique_cities FROM users;
SELECT COUNT(phone) AS users_with_phone FROM users;  -- Excludes NULLs

-- SUM
SELECT SUM(total_amount) AS total_revenue FROM orders;
SELECT SUM(quantity * price) AS total_value FROM order_items;

-- AVG
SELECT AVG(price) AS average_price FROM products;
SELECT AVG(rating) AS average_rating FROM reviews WHERE rating IS NOT NULL;

-- MIN and MAX
SELECT
    MIN(price) AS cheapest,
    MAX(price) AS most_expensive,
    MAX(price) - MIN(price) AS price_range
FROM products;

-- Multiple aggregates
SELECT
    COUNT(*) AS total_orders,
    SUM(total_amount) AS revenue,
    AVG(total_amount) AS avg_order_value,
    MIN(total_amount) AS smallest_order,
    MAX(total_amount) AS largest_order
```

```
FROM orders
WHERE status = 'completed';
```

GROUP BY

Group rows with same values into summary rows.

```sql
-- Basic GROUP BY
SELECT
    status,
    COUNT(*) AS order_count,
    SUM(total_amount) AS total_revenue
FROM orders
GROUP BY status;

-- Multiple columns
SELECT
    DATE(created_at) AS order_date,
    status,
    COUNT(*) AS order_count,
    AVG(total_amount) AS avg_order_value
FROM orders
GROUP BY DATE(created_at), status
ORDER BY order_date DESC, status;

-- With JOIN
SELECT
    c.category_name,
    COUNT(p.id) AS product_count,
    AVG(p.price) AS avg_price,
    SUM(p.stock_quantity) AS total_stock
FROM categories c
LEFT JOIN products p ON c.id = p.category_id
GROUP BY c.id, c.category_name
ORDER BY product_count DESC;
```

Important: All non-aggregated columns in SELECT must be in GROUP BY (unless using ONLY_FULL_GROUP_BY mode is disabled).

```sql
-- Modern MySQL (ONLY_FULL_GROUP_BY enabled)
SELECT
    category_id,
    category_name,  -- Must be in GROUP BY
    COUNT(*) AS product_count
FROM products
```

````
GROUP BY category_id, category_name;

-- Check SQL mode
SELECT @@sql_mode;
````

HAVING Clause

Filter groups (use HAVING for aggregates, WHERE for rows).

````sql
-- Find users with more than 5 orders
SELECT
    user_id,
    COUNT(*) AS order_count,
    SUM(total_amount) AS total_spent
FROM orders
WHERE status = 'completed'
GROUP BY user_id
HAVING order_count > 5
    AND total_spent > 1000
ORDER BY total_spent DESC;

-- Categories with average price above threshold
SELECT
    category_id,
    COUNT(*) AS product_count,
    AVG(price) AS avg_price
FROM products
WHERE is_active = TRUE
GROUP BY category_id
HAVING avg_price > 100
    AND product_count >= 5;
````

WHERE vs HAVING:
- **WHERE**: Filters rows BEFORE grouping
- **HAVING**: Filters groups AFTER grouping
- Use WHERE when possible (better performance)

````sql
-- Good: Filter before grouping
SELECT category_id, COUNT(*) AS active_products
FROM products
WHERE is_active = TRUE
GROUP BY category_id;

-- Bad: Filter after grouping (slower)
SELECT category_id, COUNT(*) AS product_count
````

```
FROM products
GROUP BY category_id
HAVING SUM(is_active) = COUNT(*);
````

GROUP_CONCAT

Concatenate values from multiple rows into single string.

````sql
-- Basic GROUP_CONCAT
SELECT
    category_id,
    GROUP_CONCAT(product_name) AS products
FROM products
GROUP BY category_id;

-- With separator
SELECT
    category_id,
    GROUP_CONCAT(product_name SEPARATOR ' | ') AS products
FROM products
GROUP BY category_id;

-- With ORDER BY
SELECT
    category_id,
    GROUP_CONCAT(
        product_name
        ORDER BY price DESC
        SEPARATOR ', '
    ) AS products_by_price
FROM products
GROUP BY category_id;

-- With DISTINCT
SELECT
    order_id,
    GROUP_CONCAT(DISTINCT product_name ORDER BY product_name) AS unique_products
FROM order_items oi
INNER JOIN products p ON oi.product_id = p.id
GROUP BY order_id;

-- Control max length
SET GROUP_CONCAT_MAX_LEN = 10000;
````

WITH ROLLUP

Add subtotals and grand totals to GROUP BY.

````sql
-- Basic ROLLUP
SELECT
    category,
    subcategory,
    SUM(quantity) AS total_quantity
FROM inventory
GROUP BY category, subcategory WITH ROLLUP;

-- Results:
-- Electronics  Phones      100
-- Electronics  Laptops     50
-- Electronics  NULL        150  (subtotal for Electronics)
-- Clothing     Shirts      200
-- Clothing     Pants       150
-- Clothing     NULL        350  (subtotal for Clothing)
-- NULL         NULL        500  (grand total)

-- Make NULL labels meaningful
SELECT
    COALESCE(category, 'TOTAL') AS category,
    COALESCE(subcategory, 'Subtotal') AS subcategory,
    SUM(quantity) AS total_quantity
FROM inventory
GROUP BY category, subcategory WITH ROLLUP;
````

Statistical Functions

````sql
-- Standard deviation
SELECT
    category_id,
    AVG(price) AS avg_price,
    STDDEV(price) AS std_dev,
    VARIANCE(price) AS variance
FROM products
GROUP BY category_id;

-- Bit operations
SELECT
    BIT_AND(permissions) AS common_permissions,
    BIT_OR(permissions) AS any_permissions,
    BIT_XOR(permissions) AS xor_permissions
FROM user_roles
WHERE group_id = 1;
````

````

### 4.4 Window Functions

Perform calculations across a set of rows related to the current row (MySQL 8.0+).

#### ROW_NUMBER

Assigns unique sequential number to each row.

```sql
-- Basic row numbering
SELECT
    product_name,
    price,
    ROW_NUMBER() OVER (ORDER BY price DESC) AS price_rank
FROM products;

-- Numbered within groups
SELECT
    category_id,
    product_name,
    price,
    ROW_NUMBER() OVER (
        PARTITION BY category_id
        ORDER BY price DESC
    ) AS category_rank
FROM products;
````

RANK and DENSE_RANK

```sql
-- RANK: Gaps in sequence for ties (1, 2, 2, 4)
-- DENSE_RANK: No gaps (1, 2, 2, 3)
SELECT
    product_name,
    price,
    RANK() OVER (ORDER BY price DESC) AS rank_with_gaps,
    DENSE_RANK() OVER (ORDER BY price DESC) AS rank_no_gaps,
    ROW_NUMBER() OVER (ORDER BY price DESC) AS row_num
FROM products;

-- Example results:
-- iPhone 15    $999    1    1    1
-- Galaxy S23   $999    1    1    2  (same price, tied rank)
-- Pixel 8      $899    3    2    3  (RANK skips 2, DENSE_RANK doesn't)
````
````

PARTITION BY

Divide result set into partitions and apply window function to each.

````sql
-- Rank products within each category
SELECT
    category_name,
    product_name,
    price,
    ROW_NUMBER() OVER (
        PARTITION BY category_name
        ORDER BY price DESC
    ) AS category_rank
FROM products p
INNER JOIN categories c ON p.category_id = c.id;

-- Top 3 products per category
SELECT *
FROM (
    SELECT
        category_id,
        product_name,
        price,
        ROW_NUMBER() OVER (
            PARTITION BY category_id
            ORDER BY price DESC
        ) AS rn
    FROM products
) ranked
WHERE rn <= 3;
````

Running Totals (Cumulative Sum)

````sql
-- Basic running total
SELECT
    order_date,
    daily_revenue,
    SUM(daily_revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_revenue
FROM daily_sales;

-- Shorter syntax (same result)
SELECT
    order_date,
````

```
    daily_revenue,
    SUM(daily_revenue) OVER (ORDER BY order_date) AS cumulative_revenue
FROM daily_sales;

-- Running total by category
SELECT
    category_id,
    order_date,
    daily_revenue,
    SUM(daily_revenue) OVER (
        PARTITION BY category_id
        ORDER BY order_date
    ) AS category_cumulative
FROM daily_sales;
````

Moving Averages
````sql
-- 7-day moving average
SELECT
    order_date,
    revenue,
    AVG(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS moving_avg_7_days
FROM daily_revenue;

-- 3-day moving average (previous, current, next)
SELECT
    order_date,
    revenue,
    AVG(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS moving_avg_3_days
FROM daily_revenue;

-- Moving sum
SELECT
    order_date,
    revenue,
    SUM(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
    ) AS rolling_30_day_revenue
FROM daily_revenue;
```

````

LAG and LEAD

Access previous and next rows without self-join.

````sql
-- Compare with previous day
SELECT
    order_date,
    revenue,
    LAG(revenue, 1) OVER (ORDER BY order_date) AS previous_day_revenue,
    revenue - LAG(revenue, 1) OVER (ORDER BY order_date) AS day_over_day_change,
    ROUND(
        (revenue - LAG(revenue, 1) OVER (ORDER BY order_date)) /
        LAG(revenue, 1) OVER (ORDER BY order_date) * 100,
        2
    ) AS percent_change
FROM daily_revenue;

-- Compare with next day
SELECT
    order_date,
    revenue,
    LEAD(revenue, 1) OVER (ORDER BY order_date) AS next_day_revenue
FROM daily_revenue;

-- Default value for missing rows
SELECT
    order_date,
    revenue,
    LAG(revenue, 1, 0) OVER (ORDER BY order_date) AS previous_day_revenue
FROM daily_revenue;

-- Look back 7 days
SELECT
    order_date,
    revenue,
    LAG(revenue, 7) OVER (ORDER BY order_date) AS week_ago_revenue
FROM daily_revenue;
````

FIRST_VALUE and LAST_VALUE

````sql
-- Compare each product to cheapest/most expensive in category
SELECT
    category_id,
    product_name,
````
````

```
    price,
    FIRST_VALUE(price) OVER (
        PARTITION BY category_id
        ORDER BY price
    ) AS cheapest_in_category,
    LAST_VALUE(price) OVER (
        PARTITION BY category_id
        ORDER BY price
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS most_expensive_in_category
FROM products;

-- Get first and last order date for each user
SELECT DISTINCT
    user_id,
    FIRST_VALUE(order_date) OVER (
        PARTITION BY user_id
        ORDER BY order_date
    ) AS first_order,
    LAST_VALUE(order_date) OVER (
        PARTITION BY user_id
        ORDER BY order_date
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_order
FROM orders;
````

NTH_VALUE

````sql
-- Get 2nd highest price in each category
SELECT DISTINCT
    category_id,
    NTH_VALUE(price, 2) OVER (
        PARTITION BY category_id
        ORDER BY price DESC
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS second_highest_price
FROM products;
````

NTILE

Divide rows into N buckets.

````sql
-- Divide products into 4 price quartiles
SELECT
    product_name,
```

```
    price,
    NTILE(4) OVER (ORDER BY price) AS price_quartile
FROM products;

-- Results:
-- Quartile 1: Bottom 25% (cheapest)
-- Quartile 2: 25-50%
-- Quartile 3: 50-75%
-- Quartile 4: Top 25% (most expensive)

-- Customer segmentation by spending
SELECT
    user_id,
    total_spent,
    NTILE(3) OVER (ORDER BY total_spent DESC) AS customer_tier,
    CASE NTILE(3) OVER (ORDER BY total_spent DESC)
        WHEN 1 THEN 'High Value'
        WHEN 2 THEN 'Medium Value'
        WHEN 3 THEN 'Low Value'
    END AS tier_label
FROM (
    SELECT user_id, SUM(total_amount) AS total_spent
    FROM orders
    GROUP BY user_id
) user_spending;
````

PERCENT_RANK and CUME_DIST

````sql
-- Percentile rank (0 to 1)
SELECT
    product_name,
    price,
    PERCENT_RANK() OVER (ORDER BY price) AS percentile_rank,
    ROUND(PERCENT_RANK() OVER (ORDER BY price) * 100, 2) AS percentile
FROM products;

-- Cumulative distribution
SELECT
    product_name,
    price,
    CUME_DIST() OVER (ORDER BY price) AS cumulative_dist,
    ROUND(CUME_DIST() OVER (ORDER BY price) * 100, 2) AS percent_below_or_equal
FROM products;
````

Frame Specifications

Control which rows are included in window function calculation.

````sql
-- ROWS: Physical rows
-- RANGE: Logical range (all rows with same ORDER BY value)

-- Current row only
SELECT
    date, value,
    SUM(value) OVER (
        ORDER BY date
        ROWS BETWEEN CURRENT ROW AND CURRENT ROW
    ) AS current_only
FROM data;

-- All preceding rows
SELECT
    date, value,
    SUM(value) OVER (
        ORDER BY date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM data;

-- 3 rows before and after
SELECT
    date, value,
    AVG(value) OVER (
        ORDER BY date
        ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING
    ) AS moving_avg_7
FROM data;

-- All rows in partition
SELECT
    category_id, value,
    SUM(value) OVER (
        PARTITION BY category_id
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS category_total
FROM data;
````

✅ Window Function Best Practices:

1. **Window functions don't reduce rows** (unlike GROUP BY)
2. **Executed after WHERE, GROUP BY, and HAVING**

3. **Can't be used in WHERE clause** (use subquery or CTE)
4. **More efficient than self-joins** for many use cases
5. **Use PARTITION BY to reset calculation per group**
6. **Named windows can reduce code duplication**

````sql
-- Named window
SELECT
    product_name,
    price,
    AVG(price) OVER w AS avg_price,
    MAX(price) OVER w AS max_price,
    MIN(price) OVER w AS min_price
FROM products
WINDOW w AS (PARTITION BY category_id);
````

Chapter 5: Indexing & Performance

5.1 Understanding Indexes

Indexes are data structures that improve query performance by allowing the database to find rows faster.

How Indexes Work

Think of an index like a book's index:
- **Without index**: Read every page to find a topic (full table scan)
- **With index**: Look up topic in index, jump directly to the right page

MySQL uses **B-Tree** indexes by default (balanced tree structure).

Performance Impact

````sql
-- Without index: Full table scan
SELECT * FROM users WHERE email = 'john@example.com';
-- Scans all 1,000,000 rows

-- With index on email: Index seek
CREATE INDEX idx_email ON users(email);
SELECT * FROM users WHERE email = 'john@example.com';
-- Reads only a few rows
````

When to Use Indexes

✅ Index these columns:

- Primary keys (auto-indexed)
- Foreign keys
- Columns in WHERE clauses
- Columns in JOIN conditions
- Columns in ORDER BY
- Columns in GROUP BY
- Columns with high cardinality (many unique values)

❌ Don't index:
- Small tables (< 1000 rows)
- Columns with low cardinality (few unique values like boolean, gender)
- Columns that are frequently updated
- Tables with heavy write operations (indexes slow down INSERT/UPDATE/DELETE)

Index Trade-offs

Pros:
- Faster SELECT queries
- Faster JOIN operations
- Enforces uniqueness (UNIQUE indexes)

Cons:
- Slower INSERT/UPDATE/DELETE (must update indexes)
- Uses disk space
- Too many indexes confuse optimizer

5.2 Index Types

Primary Key Index
````sql
-- Automatically created with PRIMARY KEY
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL
);

-- Composite primary key
CREATE TABLE order_items (
    order_id INT UNSIGNED,
    product_id INT UNSIGNED,
    quantity INT,
    PRIMARY KEY (order_id, product_id)
);
````

InnoDB: Primary key is the clustered index (table data is stored in primary key order).

Unique Index

````sql
-- Ensures column values are unique
CREATE UNIQUE INDEX idx_email ON users(email);

-- Or during table creation
CREATE TABLE users (
    id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    username VARCHAR(50),
    UNIQUE KEY unique_username (username)
);

-- Composite unique index
CREATE UNIQUE INDEX idx_user_product ON favorites(user_id, product_id);
````

Regular Index (B-Tree)

````sql
-- Single column index
CREATE INDEX idx_last_name ON users(last_name);

-- Composite index (multiple columns)
CREATE INDEX idx_name ON users(last_name, first_name);

-- The order matters!
-- This index helps:
-- ✅ WHERE last_name = 'Smith'
-- ✅ WHERE last_name = 'Smith' AND first_name = 'John'
-- ✅ ORDER BY last_name, first_name
--
-- Does NOT help:
-- ❌ WHERE first_name = 'John' (alone)
-- ❌ ORDER BY first_name

-- Add index to existing table
ALTER TABLE users ADD INDEX idx_city (city);

-- Index with sort order (MySQL 8.0+)
CREATE INDEX idx_created_desc ON posts(created_at DESC);
````

Prefix Index

For long VARCHAR or TEXT columns, index only the first N characters.

````sql
-- Index first 10 characters
````

```
CREATE INDEX idx_url_prefix ON pages(url(10));

-- Find optimal prefix length
SELECT
    COUNT(DISTINCT url) AS total_unique,
    COUNT(DISTINCT LEFT(url, 5)) AS prefix_5,
    COUNT(DISTINCT LEFT(url, 10)) AS prefix_10,
    COUNT(DISTINCT LEFT(url, 15)) AS prefix_15,
    COUNT(DISTINCT LEFT(url, 20)) AS prefix_20
FROM pages;

-- Choose prefix length where prefix_N ≈ total_unique
````

Trade-off: Saves space but can't be used for ORDER BY or covering index.

Covering Index

Index that includes all columns needed by query (no table lookup required).
````sql
-- Query
SELECT id, username, email FROM users WHERE username = 'johndoe';

-- Covering index includes all SELECT columns
CREATE INDEX idx_user_profile ON users(username, email, id);

-- MySQL can answer query using only the index (very fast!)
````

Check with EXPLAIN:
````sql
EXPLAIN SELECT id, username, email FROM users WHERE username = 'johndoe';
-- Extra: "Using index" = covering index used
````

Full-Text Index

For text searching (better than LIKE).
````sql
-- Create full-text index
CREATE FULLTEXT INDEX idx_content ON articles(title, body);

-- Search
SELECT * FROM articles
WHERE MATCH(title, body)
AGAINST('mysql performance' IN NATURAL LANGUAGE MODE);
```

```
-- Boolean mode (AND, OR, NOT)
SELECT * FROM articles
WHERE MATCH(title, body)
AGAINST('+mysql -mongodb' IN BOOLEAN MODE);
-- + means must include
-- - means must not include

-- Query expansion
SELECT * FROM articles
WHERE MATCH(title, body)
AGAINST('database' WITH QUERY EXPANSION);
````

Limitations:
- Only for CHAR, VARCHAR, TEXT columns
- Minimum word length: 4 characters (by default)
- Ignores common words (stopwords)
- InnoDB and MyISAM only

Spatial Index

For geographic data (GEOMETRY columns).
````sql
CREATE TABLE stores (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    location POINT NOT NULL,
    SPATIAL INDEX idx_location (location)
);

-- Find nearby stores
SELECT id, name,
    ST_Distance_Sphere(
        location,
        ST_GeomFromText('POINT(40.7128 -74.0060)')
    ) AS distance_meters
FROM stores
WHERE ST_Distance_Sphere(
    location,
    ST_GeomFromText('POINT(40.7128 -74.0060)')
) < 5000
ORDER BY distance_meters;
````

Descending Index (MySQL 8.0+)
````sql
-- Mixed sort orders
```

```
CREATE INDEX idx_score_date ON leaderboard(score DESC, created_at ASC);

-- Helps query:
SELECT * FROM leaderboard
ORDER BY score DESC, created_at ASC;
````

Invisible Index (MySQL 8.0+)

Test removing an index without actually dropping it.
````sql
-- Make index invisible (not used by optimizer)
ALTER TABLE users ALTER INDEX idx_city INVISIBLE;

-- Test queries
-- ...

-- Make visible again
ALTER TABLE users ALTER INDEX idx_city VISIBLE;

-- Or drop if not needed
DROP INDEX idx_city ON users;
````

Index Management
````sql
-- Show indexes on table
SHOW INDEX FROM users;

-- Show index usage (MySQL 8.0+)
SELECT * FROM sys.schema_unused_indexes WHERE object_schema = 'mydb';

-- Drop index
DROP INDEX idx_last_name ON users;

-- Rename index (MySQL 5.7+)
ALTER TABLE users RENAME INDEX old_name TO new_name;

-- Disable/enable keys (MyISAM only)
ALTER TABLE mytable DISABLE KEYS;
-- ... bulk insert
ALTER TABLE mytable ENABLE KEYS;

-- Rebuild index
ALTER TABLE users DROP INDEX idx_name, ADD INDEX idx_name(last_name, first_name);

-- Analyze table to update index statistics
```

````
ANALYZE TABLE users;
````

5.3 Query Optimization

Optimization Principles

1. **Select only needed columns**
2. **Use appropriate indexes**
3. **Filter early with WHERE**
4. **Limit result set**
5. **Avoid functions on indexed columns**
6. **Use proper join types**
7. **Cache results when appropriate**

Optimization Techniques

1. Select Only Needed Columns
````sql
-- Bad: Fetches all columns
SELECT * FROM users WHERE id = 1;

-- Good: Fetch only what you need
SELECT id, username, email FROM users WHERE id = 1;
````

Why it matters:
- Less data transferred
- Less memory used
- Enables covering indexes
- Faster network transmission

2. Use LIMIT
````sql
-- Without LIMIT: Reads all matching rows
SELECT * FROM users WHERE city = 'New York';

-- With LIMIT: Stops scanning after N rows found
SELECT * FROM users WHERE city = 'New York' LIMIT 100;

-- Existence check
SELECT 1 FROM orders WHERE user_id = 123 LIMIT 1;
-- Much faster than COUNT(*)
````

3. Avoid Functions on Indexed Columns
````sql
````

```sql
-- Bad: Can't use index on created_at
SELECT * FROM orders
WHERE YEAR(created_at) = 2024;

-- Good: Can use index
SELECT * FROM orders
WHERE created_at >= '2024-01-01'
  AND created_at < '2025-01-01';

-- Bad: Can't use index
SELECT * FROM users
WHERE LOWER(email) = 'john@example.com';

-- Good: Store email in lowercase, or use case-insensitive collation
SELECT * FROM users
WHERE email = 'john@example.com';
```

4. Use EXISTS Instead of COUNT for Existence Check

```sql
-- Bad: Counts all matching rows (slow)
SELECT IF(COUNT(*) > 0, 'exists', 'not exists')
FROM orders
WHERE user_id = 1;

-- Good: Stops at first match (fast)
SELECT IF(EXISTS(
  SELECT 1 FROM orders WHERE user_id = 1
), 'exists', 'not exists');
```

5. Optimize LIKE Queries

```sql
-- Bad: Leading wildcard can't use index
SELECT * FROM users WHERE email LIKE '%@gmail.com';

-- Good: Can use index
SELECT * FROM users WHERE email LIKE 'john%';

-- For full-text search, use FULLTEXT index
SELECT * FROM articles
WHERE MATCH(content) AGAINST('optimization');

-- Alternative for ends-with: reverse string index
ALTER TABLE users ADD COLUMN email_reversed VARCHAR(100)
  AS (REVERSE(email));
CREATE INDEX idx_email_reversed ON users(email_reversed);
```

```sql
SELECT * FROM users
WHERE email_reversed LIKE REVERSE('%@gmail.com');
```

6. Avoid OR with Different Columns
```sql
-- Bad: May not use indexes efficiently
SELECT * FROM users
WHERE first_name = 'John' OR last_name = 'Smith';

-- Better: Use UNION (can use separate indexes)
SELECT * FROM users WHERE first_name = 'John'
UNION
SELECT * FROM users WHERE last_name = 'Smith';

-- Or create a composite index
CREATE INDEX idx_names ON users(first_name, last_name);
```

7. Use JOIN Instead of Subquery (Usually)
```sql
-- Often slower: Subquery
SELECT * FROM users
WHERE id IN (
    SELECT user_id FROM orders WHERE total > 1000
);

-- Usually faster: JOIN
SELECT DISTINCT u.*
FROM users u
INNER JOIN orders o ON u.id = o.user_id
WHERE o.total > 1000;

-- But EXISTS can be faster than both for large sets
SELECT * FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.user_id = u.id AND o.total > 1000
);
```

8. Optimize Pagination
```sql
-- Bad: OFFSET becomes very slow for large offsets
SELECT * FROM products
ORDER BY id
```

```
LIMIT 1000 OFFSET 100000;  -- Reads and discards 100,000 rows!

-- Good: Cursor-based pagination (seek method)
SELECT * FROM products
WHERE id > 100000
ORDER BY id
LIMIT 1000;

-- For complex sorting, use bookmark
SELECT * FROM products
WHERE (created_at, id) > ('2024-01-15 10:00:00', 12345)
ORDER BY created_at, id
LIMIT 1000;
````

9. Use Proper Data Types
````sql
-- Bad: Storing numbers as strings
CREATE TABLE bad (
    id VARCHAR(20),  -- Should be INT
    price VARCHAR(10)  -- Should be DECIMAL
);

-- Good: Use appropriate types
CREATE TABLE good (
    id INT UNSIGNED,
    price DECIMAL(10,2)
);
````

Why it matters:
- Smaller storage = faster queries
- Proper sorting/comparison
- Index efficiency
- Type safety

10. Avoid SELECT DISTINCT When Possible
````sql
-- Bad: DISTINCT on large result set (slow)
SELECT DISTINCT user_id FROM orders;

-- Better: Use GROUP BY with index
SELECT user_id FROM orders GROUP BY user_id;

-- Or if you need other columns:
SELECT user_id, MIN(created_at) as first_order
FROM orders
```

````
GROUP BY user_id;
````

11. Use UNION ALL Instead of UNION
````sql
-- UNION: Removes duplicates (slower)
SELECT id FROM table1
UNION
SELECT id FROM table2;

-- UNION ALL: Keeps duplicates (faster)
SELECT id FROM table1
UNION ALL
SELECT id FROM table2;
````

Use UNION ALL when you know there are no duplicates or don't need to remove them.

12. Optimize JOIN Order
````sql
-- MySQL optimizer usually handles this, but you can help:

-- Start with smallest/most filtered table
SELECT o.*, p.*
FROM orders o  -- Smaller table after WHERE
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
WHERE o.created_at > '2024-01-01'  -- Filters orders significantly
````

13. Use Index Hints (Rarely Needed)
````sql
-- Force use of specific index
SELECT * FROM users
FORCE INDEX (idx_email)
WHERE email = 'john@example.com';

-- Suggest index
SELECT * FROM users
USE INDEX (idx_city)
WHERE city = 'New York';

-- Ignore index
SELECT * FROM users
IGNORE INDEX (idx_name)
WHERE last_name = 'Smith';
````

⚠️ Warning: Index hints override the optimizer. Only use when you know better than the optimizer (rare).

14. Avoid Implicit Type Conversion

```sql
-- Bad: id is INT, but comparing to string (can't use index)
SELECT * FROM users WHERE id = '123';

-- Good: Use correct type
SELECT * FROM users WHERE id = 123;
```

15. Use Batch Operations

```sql
-- Bad: Individual inserts
INSERT INTO logs VALUES (1, 'msg1');
INSERT INTO logs VALUES (2, 'msg2');
-- ... 1000 times

-- Good: Batch insert (100x faster)
INSERT INTO logs VALUES
  (1, 'msg1'),
  (2, 'msg2'),
  -- ... 1000 rows
  (1000, 'msg1000');

-- Bad: Individual updates
UPDATE users SET status = 'active' WHERE id = 1;
UPDATE users SET status = 'active' WHERE id = 2;

-- Good: Single update
UPDATE users SET status = 'active' WHERE id IN (1, 2, 3, ...);
```

5.4 EXPLAIN Plans

EXPLAIN shows how MySQL executes a query. Essential for optimization.

Basic EXPLAIN

```sql
EXPLAIN SELECT * FROM users WHERE email = 'john@example.com';
```

EXPLAIN Output Columns

Column	Meaning

id	SELECT identifier
select_type	Type of SELECT (SIMPLE, SUBQUERY, etc.)
table	Table being accessed
partitions	Partitions matched
type	**Join type (most important!)**
possible_keys	Indexes that could be used
key	**Index actually used**
key_len	Length of key used
ref	Columns compared to index
rows	**Estimated rows to examine**
filtered	% of rows filtered by condition
Extra	**Additional information**

Join Types (Best to Worst)

1. **system**: Table has only one row
2. **const**: At most one matching row (PRIMARY KEY or UNIQUE lookup)
3. **eq_ref**: One row per combination (JOINs on PRIMARY KEY)
4. **ref**: Multiple rows with matching index value
5. **fulltext**: FULLTEXT index used
6. **ref_or_null**: Like ref, but searches for NULL too
7. **index_merge**: Multiple indexes used
8. **unique_subquery**: Subquery with unique index
9. **index_subquery**: Subquery with non-unique index
10. **range**: Index range scan (BETWEEN, >, <, IN)
11. **index**: Full index scan (better than ALL)
12. **ALL**: **Full table scan (AVOID!)**

```sql
-- type: const (best)
EXPLAIN SELECT * FROM users WHERE id = 1;

-- type: ref (good)
EXPLAIN SELECT * FROM users WHERE city = 'New York';

-- type: range (decent)
EXPLAIN SELECT * FROM orders WHERE created_at > '2024-01-01';

-- type: ALL (bad - full table scan!)
EXPLAIN SELECT * FROM users WHERE YEAR(created_at) = 2024;
```

Important Extra Values

Extra	Meaning	Good/Bad
Using index	Covering index (no table lookup)	✅ Excellent
Using where	WHERE clause filtering	✅ Normal

Using index condition	Index Condition Pushdown	✅ Good
Using temporary	Temporary table created	⚠️ Slow
Using filesort	External sort needed	⚠️ Slow
Using join buffer	Join buffer used (no index)	⚠️ Bad
Impossible WHERE	WHERE always false	✅ Optimized away
Select tables optimized away	Aggregates computed at optimization	✅ Excellent

````sql
-- Using index (best)
EXPLAIN SELECT id, email FROM users WHERE email = 'test@test.com';

-- Using filesort (add index on ORDER BY column)
EXPLAIN SELECT * FROM products ORDER BY price;

-- Using temporary (avoid GROUP BY on non-indexed columns)
EXPLAIN SELECT city, COUNT(*) FROM users GROUP BY city;
````

EXPLAIN Formats

````sql
-- Traditional format
EXPLAIN SELECT * FROM users WHERE id = 1;

-- Extended information (MySQL 5.7+)
EXPLAIN EXTENDED SELECT * FROM users WHERE id = 1;
SHOW WARNINGS;  -- Shows optimized query

-- JSON format (detailed)
EXPLAIN FORMAT=JSON
SELECT * FROM users u
INNER JOIN orders o ON u.id = o.user_id;

-- Tree format (MySQL 8.0.16+)
EXPLAIN FORMAT=TREE
SELECT * FROM users u
INNER JOIN orders o ON u.id = o.user_id;
````

EXPLAIN ANALYZE (MySQL 8.0.18+)

Shows actual execution time and row counts (not just estimates).

````sql
EXPLAIN ANALYZE
SELECT u.username, COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id;
````

```
-- Output includes:
-- - Actual time taken
-- - Actual rows processed
-- - Number of loops
```

Reading EXPLAIN

```sql
EXPLAIN
SELECT o.order_number, u.username, p.product_name
FROM orders o
INNER JOIN users u ON o.user_id = u.id
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
WHERE o.created_at > '2024-01-01';
```

What to look for:

1. **type: ALL** = Full table scan (add index!)
2. **key: NULL** = No index used (add index!)
3. **rows: Large number** = Too many rows examined (filter earlier, add index)
4. **Extra: Using filesort** = Slow sort (add index on ORDER BY columns)
5. **Extra: Using temporary** = Temp table created (optimize GROUP BY)
6. **Extra: Using index** = Great! Covering index

Optimization Workflow

1. **Run EXPLAIN on slow query**
2. **Check type column** - Should not be ALL
3. **Check key column** - Should use an index
4. **Check rows** - Should be reasonable number
5. **Check Extra** - Look for warnings (filesort, temporary)
6. **Add/modify indexes** as needed
7. **Run EXPLAIN again** - Verify improvement
8. **Test with real data** - EXPLAIN uses estimates

```sql
-- Before optimization
EXPLAIN SELECT * FROM orders WHERE YEAR(created_at) = 2024;
-- type: ALL, rows: 1000000, Extra: Using where

-- Add index
CREATE INDEX idx_created ON orders(created_at);

-- Rewrite query
EXPLAIN SELECT * FROM orders
WHERE created_at >= '2024-01-01' AND created_at < '2025-01-01';
```

-- type: range, key: idx_created, rows: 50000
````

---

## Chapter 6: Transactions & ACID

### 6.1 Transaction Basics

A transaction is a sequence of SQL operations treated as a single unit of work. All operations succeed together or fail together.

#### ACID Properties

**Atomicity**: All or nothing
````sql
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;  -- Both updates succeed or both fail
````

Consistency: Database moves from one valid state to another
````sql
-- Total balance before = Total balance after
-- Constraints are enforced
-- Foreign keys maintained
````

Isolation: Concurrent transactions don't interfere
````sql
-- Transaction 1 doesn't see uncommitted changes from Transaction 2
````

Durability: Committed changes are permanent
````sql
COMMIT;  -- Changes survive server crash, power failure
````

Basic Transaction Syntax
````sql
-- Start transaction
START TRANSACTION;
-- or
BEGIN;

-- Your SQL operations

```
INSERT INTO orders (user_id, total) VALUES (1, 100);
INSERT INTO order_items (order_id, product_id, quantity)
VALUES (LAST_INSERT_ID(), 101, 2);
UPDATE products SET stock = stock - 2 WHERE id = 101;

-- Commit changes (make permanent)
COMMIT;

-- Or rollback if error
ROLLBACK;
````

Autocommit Mode
````sql
-- Check autocommit status
SELECT @@autocommit;

-- Disable autocommit
SET autocommit = 0;

-- Now each statement starts implicit transaction
INSERT INTO users VALUES (...);  -- Transaction started
UPDATE users SET ...;            -- Same transaction
COMMIT;                          -- Commit explicitly

-- Enable autocommit (default)
SET autocommit = 1;
-- Each statement auto-commits
````

Transaction Example: Money Transfer
````sql
START TRANSACTION;

-- Check source account balance
SELECT @balance := balance FROM accounts WHERE id = 1 FOR UPDATE;

-- Validate, username):
        raise ValueError("Invalid username format")
    return username

# Use in query
username = validate_username(request.form['username'])
cursor.execute("SELECT * FROM users WHERE username = %s", (username,))
````

2. Whitelist Validation for Dynamic Parts

````python
# When query structure must be dynamic
allowed_columns = ['name', 'email', 'created_at']
sort_column = request.args.get('sort', 'created_at')

if sort_column not in allowed_columns:
    sort_column = 'created_at'

# Safe to use in query now
query = f"SELECT * FROM users ORDER BY {sort_column}"
cursor.execute(query)
````

3. Escape Special Characters (Last Resort)
````python
# Use only when prepared statements impossible
from mysql.connector import connect

def escape_string(value):
    return conn.escape_string(str(value))

# Still use prepared statements when possible!
````

4. Principle of Least Privilege
````sql
-- Application user should not have DROP, ALTER, etc.
GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.* TO 'appuser'@'localhost';

-- Even if SQL injection occurs, damage is limited
````

5. Stored Procedures
````sql
-- Create stored procedure
DELIMITER //
CREATE PROCEDURE GetUserByUsername(IN p_username VARCHAR(50))
BEGIN
    SELECT * FROM users WHERE username = p_username;
END //
DELIMITER ;

-- Call from application
CALL GetUserByUsername('johndoe');
````
````python
# Python
````

```
cursor.callproc('GetUserByUsername', (username,))
```

✅ SQL Injection Prevention Checklist:

1. ✅ **Always use prepared statements/parameterized queries**
2. ✅ **Never concatenate user input** into SQL strings
3. ✅ **Validate and sanitize all input**
4. ✅ **Use whitelist validation** for dynamic query parts
5. ✅ **Apply principle of least privilege** to database users
6. ✅ **Use stored procedures** when appropriate
7. ✅ **Enable SQL error logging** (but don't expose errors to users)
8. ✅ **Regular security audits** of codebase
9. ✅ **Use WAF (Web Application Firewall)** for additional protection
10. ✅ **Keep database and drivers updated**

7.3 Data Encryption

Encryption at Rest

1. Table Encryption (InnoDB)
```sql
-- Enable encryption
SET GLOBAL innodb_encryption_threads = 4;

-- Create encrypted table
CREATE TABLE sensitive_data (
    id INT PRIMARY KEY,
    ssn VARCHAR(11),
    credit_card VARCHAR(16)
) ENCRYPTION='Y';

-- Encrypt existing table
ALTER TABLE users ENCRYPTION='Y';

-- Check encryption status
SELECT
    TABLE_SCHEMA,
    TABLE_NAME,
    CREATE_OPTIONS
FROM information_schema.TABLES
WHERE CREATE_OPTIONS LIKE '%ENCRYPTION%';
```

2. Tablespace Encryption
```sql
-- Enable in my.cnf
```

```
[mysqld]
early-plugin-load=keyring_file.so
keyring_file_data=/var/lib/mysql-keyring/keyring

-- Create encrypted tablespace
CREATE TABLESPACE encrypted_ts ADD DATAFILE 'encrypted.ibd' ENCRYPTION='Y';

-- Use tablespace
CREATE TABLE users (
    id INT PRIMARY KEY,
    email VARCHAR(100)
) TABLESPACE=encrypted_ts;
````

3. Binary Log Encryption
````ini
# my.cnf
[mysqld]
binlog_encryption = ON
````

Encryption in Transit (SSL/TLS)

1. Enable SSL
````sql
-- Check SSL status
SHOW VARIABLES LIKE '%ssl%';

-- Require SSL for user
ALTER USER 'appuser'@'%' REQUIRE SSL;

-- Require specific SSL options
ALTER USER 'appuser'@'%'
REQUIRE X509;  -- Client must present valid certificate

ALTER USER 'appuser'@'%'
REQUIRE ISSUER '/CN=MySQL_CA';  -- Specific CA

ALTER USER 'appuser'@'%'
REQUIRE SUBJECT '/CN=appuser';  -- Specific subject
````

2. Configure SSL (my.cnf)
````ini
[mysqld]
ssl-ca=/path/to/ca.pem
ssl-cert=/path/to/server-cert.pem
```

````
ssl-key=/path/to/server-key.pem

# Require SSL for all connections
require_secure_transport = ON
````

3. Connect with SSL
````python
# Python
import mysql.connector

conn = mysql.connector.connect(
    host='db.example.com',
    user='appuser',
    password='password',
    database='mydb',
    ssl_ca='/path/to/ca.pem',
    ssl_verify_cert=True
)
````
````php
// PHP
$conn = new mysqli();
$conn->ssl_set(null, null, '/path/to/ca.pem', null, null);
$conn->real_connect('db.example.com', 'appuser', 'password', 'mydb', 3306, null, MYSQLI_CLIENT_SSL);
````

Application-Level Encryption

For sensitive data like credit cards, SSN:
````python
# Python example using cryptography library
from cryptography.fernet import Fernet

# Generate key (store securely!)
key = Fernet.generate_key()
cipher = Fernet(key)

# Encrypt
ssn = "123-45-6789"
encrypted_ssn = cipher.encrypt(ssn.encode())

# Store encrypted data
cursor.execute(
    "INSERT INTO users (name, ssn_encrypted) VALUES (%s, %s)",
    (name, encrypted_ssn)
)
````

```
# Decrypt when needed
cursor.execute("SELECT ssn_encrypted FROM users WHERE id = %s", (user_id,))
encrypted = cursor.fetchone()[0]
decrypted_ssn = cipher.decrypt(encrypted).decode()
```

Best Practices:
- Store encryption keys separately (not in database!)
- Use key management service (AWS KMS, Azure Key Vault, HashiCorp Vault)
- Rotate keys regularly
- Use strong encryption algorithms (AES-256)

Hashing Passwords

Never store plain text passwords!
```python
# Python - bcrypt (recommended)
import bcrypt

# Hash password
password = "user_password_123"
hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

# Store hash
cursor.execute(
    "INSERT INTO users (username, password_hash) VALUES (%s, %s)",
    (username, hashed)
)

# Verify password
stored_hash = cursor.fetchone()[0]
if bcrypt.checkpw(password.encode(), stored_hash):
    print("Password correct!")
```
```php
// PHP - password_hash (built-in)
// Hash
$hash = password_hash($password, PASSWORD_BCRYPT);

// Store
$stmt->execute([$username, $hash]);

// Verify
if (password_verify($input_password, $stored_hash)) {
    echo "Password correct!";
}
```

````

**✅ Password Hashing Best Practices:**

1. Use bcrypt, Argon2, or PBKDF2 (not MD5 or SHA1!)
2. Use salt (automatic with bcrypt)
3. Use high work factor/cost
4. Never decrypt passwords (one-way hash only)
5. Implement rate limiting on login attempts
6. Consider adding pepper (server-side secret)

### 7.4 Audit Logging

Track who did what and when for security and compliance.

#### Enable Audit Plugin
````sql
-- Install audit plugin (MySQL Enterprise or MariaDB)
INSTALL PLUGIN audit_log SONAME 'audit_log.so';

-- Configure in my.cnf
[mysqld]
audit_log = FORCE_PLUS_PERMANENT
audit_log_file = /var/log/mysql/audit.log
audit_log_format = JSON
audit_log_policy = ALL  -- Log all events
````

General Query Log (Development Only!)
````sql
-- Enable (logs ALL queries - performance impact!)
SET GLOBAL general_log = 'ON';
SET GLOBAL general_log_file = '/var/log/mysql/general.log';

-- Check status
SHOW VARIABLES LIKE 'general_log%';

-- Disable in production!
SET GLOBAL general_log = 'OFF';
````

⚠️ Warning: General log logs everything including passwords in plain text! Use only for debugging.

Application-Level Audit Trail
````sql
-- Create audit table
CREATE TABLE audit_log (

```
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    user_id INT UNSIGNED,
    action VARCHAR(50) NOT NULL,
    table_name VARCHAR(64),
    record_id BIGINT,
    old_values JSON,
    new_values JSON,
    ip_address VARCHAR(45),
    user_agent TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_user (user_id, created_at),
    INDEX idx_table (table_name, record_id),
    INDEX idx_action (action, created_at)
) ENGINE=InnoDB;

-- Log changes
INSERT INTO audit_log (user_id, action, table_name, record_id, new_values, ip_address)
VALUES (
    123,
    'UPDATE',
    'users',
    456,
    '{"email": "new@example.com"}',
    '192.168.1.100'
);
````

Triggers for Automatic Audit
```sql
-- Audit trigger for updates
DELIMITER //
CREATE TRIGGER users_audit_update
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (
        user_id,
        action,
        table_name,
        record_i# Complete MySQL Guide: From Basics to Industry Expert

**A Comprehensive Resource for Backend Developers**

Version 1.0 | 2025 Edition

---
```

Table of Contents

Introduction

Welcome to the Complete MySQL Guide, designed specifically for backend developers who want to master MySQL from fundamental concepts to industry-level expertise. This comprehensive resource covers everything you need to know to become proficient in database design, query optimization, security, and production-ready MySQL deployments.

Who This Book Is For

This book is designed for:

- Backend developers wanting to strengthen their database skills
- Software engineers preparing for senior-level positions
- Database administrators looking to deepen their MySQL knowledge
- Anyone seeking to understand production-grade database management

What You'll Learn

By the end of this guide, you will:

- Master SQL syntax and advanced query techniques
- Design efficient and scalable database schemas
- Implement robust security measures
- Optimize queries for maximum performance
- Handle transactions and maintain data integrity
- Set up replication and high-availability systems
- Monitor, troubleshoot, and maintain production databases

Chapter 1: Getting Started with MySQL

1.1 What is MySQL?

MySQL is an open-source relational database management system (RDBMS) that uses Structured Query Language (SQL). It's one of the most popular databases in the world, powering everything from small websites to large-scale enterprise applications.

Key Features

- **Open Source**: Free to use with a large community
- **Cross-Platform**: Runs on Windows, Linux, macOS
- **ACID Compliant**: Ensures data integrity
- **Scalable**: Handles millions of queries per day
- **Secure**: Robust security features built-in
- **Reliable**: Proven track record in production environments
- **Fast**: Optimized for performance
- **Flexible**: Supports various storage engines

1.2 Installation and Setup

Ubuntu/Debian
```bash
# Update package index
sudo apt update

# Install MySQL Server
sudo apt install mysql-server

# Secure installation
sudo mysql_secure_installation

# Start MySQL service
sudo systemctl start mysql

# Enable on boot
sudo systemctl enable mysql

# Check status
sudo systemctl status mysql
```

macOS (using Homebrew)
```bash
# Install MySQL
brew install mysql
```

```
# Start MySQL service
brew services start mysql

# Secure installation
mysql_secure_installation
```

Windows

1. Download MySQL Installer from mysql.com
2. Run the installer
3. Choose "Developer Default" setup type
4. Complete the configuration wizard
5. Set root password

First Login

```bash
# Login as root
sudo mysql -u root -p

# Or without password on first install
sudo mysql
```

Initial Configuration

```sql
-- Check MySQL version
SELECT VERSION();

-- Show current user
SELECT USER();

-- Show databases
SHOW DATABASES;

-- Show current database
SELECT DATABASE();
```

1.3 MySQL Architecture

Understanding MySQL's architecture is crucial for optimization and troubleshooting.

Architecture Layers

1. **Connection Layer**
   - Handles client connections

- Authentication and security
- Thread management
- Connection pooling

2. **SQL Layer**
- Query parsing and validation
- Query optimization
- Query cache (removed in MySQL 8.0)
- Access control

3. **Storage Engine Layer**
- Data storage and retrieval
- Indexing mechanisms
- Transaction management
- Lock management

Storage Engines

Engine	Transactions	Locking	Foreign Keys	Use Case
InnoDB	Yes	Row-level	Yes	Default, ACID-compliant, general purpose
MyISAM	No	Table-level	No	Legacy, read-heavy workloads
MEMORY	No	Table-level	No	Temporary data, caching
CSV	No	Table-level	No	Data exchange
ARCHIVE	No	Row-level	No	Historical data, logs

✅ Best Practice: Always use InnoDB for production applications. It provides ACID compliance, crash recovery, and better concurrency through row-level locking.

Check Storage Engine
```sql
-- Show default storage engine
SHOW VARIABLES LIKE 'default_storage_engine';

-- Show table storage engine
SHOW TABLE STATUS WHERE Name = 'users';

-- Change storage engine
ALTER TABLE users ENGINE = InnoDB;
```

1.4 MySQL Configuration

Important Configuration Variables
```sql
-- Show all variables
SHOW VARIABLES;
```

```
-- Show specific variable
SHOW VARIABLES LIKE 'max_connections';

-- Set session variable
SET SESSION sql_mode = 'STRICT_TRANS_TABLES';

-- Set global variable (persists until restart)
SET GLOBAL max_connections = 500;
```

my.cnf Configuration File
```ini
[mysqld]
# Basic Settings
port = 3306
datadir = /var/lib/mysql
socket = /var/lib/mysql/mysql.sock

# Character Set
character-set-server = utf8mb4
collation-server = utf8mb4_unicode_ci

# InnoDB Settings
innodb_buffer_pool_size = 1G
innodb_log_file_size = 256M
innodb_flush_log_at_trx_commit = 1
innodb_file_per_table = 1

# Connection Settings
max_connections = 500
max_connect_errors = 100
connect_timeout = 10
wait_timeout = 600

# Query Cache (MySQL 5.7 and earlier)
query_cache_type = 0
query_cache_size = 0

# Logging
slow_query_log = 1
slow_query_log_file = /var/log/mysql/slow-query.log
long_query_time = 2
log_error = /var/log/mysql/error.log
```

Chapter 2: Database Fundamentals

2.1 Creating Databases

Basic Database Operations
```sql
-- Create database
CREATE DATABASE ecommerce;

-- Create with character set and collation
CREATE DATABASE ecommerce
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;

-- Create if not exists
CREATE DATABASE IF NOT EXISTS ecommerce;

-- List all databases
SHOW DATABASES;

-- Select database for use
USE ecommerce;

-- Show current database
SELECT DATABASE();

-- Delete database (CAREFUL!)
DROP DATABASE ecommerce;

-- Drop if exists
DROP DATABASE IF EXISTS ecommerce;
```

✅ Best Practice: Always use `utf8mb4` character set (not `utf8`). It supports full Unicode including emojis and special characters. The `utf8mb4_unicode_ci` collation provides accurate sorting for international characters.

Database Information
```sql
-- Show database creation statement
SHOW CREATE DATABASE ecommerce;

-- Show database size
SELECT
    table_schema AS 'Database',
    ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS 'Size (MB)'
FROM information_schema.tables
```

```
GROUP BY table_schema;

-- Show tables in database
SHOW TABLES;

-- Show tables with details
SHOW TABLE STATUS;
```

2.2 Data Types

Choosing the right data type is critical for performance, storage efficiency, and data integrity.

Numeric Types

Type	Storage	Range (Signed)	Range (Unsigned)	Use Case
TINYINT	1 byte	-128 to 127	0 to 255	Boolean, small numbers
SMALLINT	2 bytes	-32,768 to 32,767	0 to 65,535	Small integers
MEDIUMINT	3 bytes	-8M to 8M	0 to 16M	Medium integers
INT	4 bytes	-2B to 2B	0 to 4B	Standard integers, IDs
BIGINT	8 bytes	-9 quintillion	0 to 18 quintillion	Large numbers, timestamps
DECIMAL(p,s)	Variable	Exact precision	Exact precision	Money, precise calculations
FLOAT	4 bytes	Approximate	Approximate	Scientific data
DOUBLE	8 bytes	Approximate	Approximate	Scientific data (avoid for money)

Examples:
```sql
CREATE TABLE numeric_examples (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    age TINYINT UNSIGNED,                -- 0-255
    quantity SMALLINT,                   -- -32,768 to 32,767
    views INT UNSIGNED,                  -- 0 to 4 billion
    user_id BIGINT UNSIGNED,             -- Large IDs
    price DECIMAL(10,2),                 -- Max 99,999,999.99
    rating DECIMAL(3,2),                 -- Max 9.99
    weight FLOAT,                        -- Approximate
    latitude DOUBLE,                     -- High precision
    is_active BOOLEAN                    -- Stored as TINYINT(1)
);
```

String Types

Type	Max Size	Storage	Use Case
CHAR(n)	255	Fixed-length	Country codes, fixed-length strings

VARCHAR(n)	65,535	Variable-length + 1-2 bytes	Most common string type
TINYTEXT	255	Variable + 1 byte	Very short text
TEXT	65,535	Variable + 2 bytes	Large text blocks
MEDIUMTEXT	16MB	Variable + 3 bytes	Articles, long content
LONGTEXT	4GB	Variable + 4 bytes	Very large text (rarely needed)
ENUM	65,535 values	1-2 bytes	Fixed set of values
SET	64 values	1-8 bytes	Multiple selections

Examples:
```sql
CREATE TABLE string_examples (
    id INT PRIMARY KEY,
    country_code CHAR(2),                    -- 'US', 'UK'
    username VARCHAR(50) NOT NULL,           -- Variable length
    email VARCHAR(100) NOT NULL,
    bio TEXT,                                -- Long description
    content MEDIUMTEXT,                      -- Article content
    status ENUM('draft', 'published', 'archived') DEFAULT 'draft',
    permissions SET('read', 'write', 'delete', 'admin')
);
```

✅ Best Practices for String Types:
- Use VARCHAR for most cases (more flexible than CHAR)
- CHAR is only better for truly fixed-length data (rare)
- Don't use TEXT types unless necessary (can't have default values, full indexing limited)
- ENUM is faster than VARCHAR for fixed sets but harder to modify
- Limit VARCHAR length appropriately (impacts memory allocation)

Date and Time Types

Type	Format	Range	Storage	Use Case
DATE	YYYY-MM-DD	1000-01-01 to 9999-12-31	3 bytes	Birth dates, deadlines
DATETIME	YYYY-MM-DD HH:MM:SS	1000 to 9999	8 bytes	Timestamps without timezone
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1970 to 2038	4 bytes	Auto-updating timestamps
TIME	HH:MM:SS	-838:59:59 to 838:59:59	3 bytes	Duration, time of day
YEAR	YYYY	1901 to 2155	1 byte	Year only

Examples:
```sql
CREATE TABLE datetime_examples (
    id INT PRIMARY KEY,
    birth_date DATE,                         -- '1990-05-15'
    appointment_time TIME,                   -- '14:30:00'
    event_datetime DATETIME,                 -- '2024-01-15 14:30:00'
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    year_manufactured YEAR                  -- 2024
);

-- Date/Time functions
SELECT
    NOW(),                              -- Current datetime
    CURDATE(),                          -- Current date
    CURTIME(),                          -- Current time
    DATE_FORMAT(NOW(), '%Y-%m-%d'),         -- Formatted date
    DATE_ADD(NOW(), INTERVAL 7 DAY),        -- Add days
    DATEDIFF(NOW(), '2024-01-01'),          -- Days between
    YEAR(NOW()),                        -- Extract year
    MONTH(NOW()),                       -- Extract month
    DAY(NOW());                         -- Extract day
```

TIMESTAMP vs DATETIME:
- TIMESTAMP: Stores as Unix timestamp, converts to timezone, auto-updates
- DATETIME: Stores as is, no timezone conversion
- Use TIMESTAMP for "when did this happen" fields
- Use DATETIME for "when should this happen" fields

Binary Types

Type	Max Size	Use Case
BINARY(n)	255	Fixed binary data
VARBINARY(n)	65,535	Variable binary data
TINYBLOB	255	Small binary objects
BLOB	65KB	Binary objects
MEDIUMBLOB	16MB	Images, files
LONGBLOB	4GB	Large files

```sql
CREATE TABLE binary_examples (
    id INT PRIMARY KEY,
    file_hash BINARY(32),               -- SHA-256 hash
    thumbnail BLOB,                     -- Small image
    document MEDIUMBLOB                 -- PDF file
);
```

⚠️ Warning: Storing files in database is generally NOT recommended. Use file storage (S3, local filesystem) and store file paths in database instead.

JSON Type (MySQL 5.7+)

```sql
```

```
CREATE TABLE json_examples (
    id INT PRIMARY KEY,
    metadata JSON,
    settings JSON
);

-- Insert JSON
INSERT INTO json_examples (id, metadata)
VALUES (1, '{"name": "John", "age": 30, "tags": ["developer", "mysql"]}');

-- Query JSON
SELECT
    id,
    JSON_EXTRACT(metadata, '$.name') as name,
    metadata->'$.age' as age,
    metadata->>'$.age' as age_text
FROM json_examples;

-- JSON functions
SELECT
    JSON_KEYS(metadata),
    JSON_LENGTH(metadata),
    JSON_CONTAINS(metadata, '"developer"', '$.tags')
FROM json_examples;
```

2.3 Creating Tables

Basic Table Creation

```sql
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    phone VARCHAR(20),
    date_of_birth DATE,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

    INDEX idx_email (email),
    INDEX idx_username (username),
    INDEX idx_name (last_name, first_name)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

```

#### Constraints
```sql
CREATE TABLE products (
    id INT UNSIGNED AUTO_INCREMENT,
    sku VARCHAR(50) NOT NULL,
    name VARCHAR(200) NOT NULL,
    description TEXT,
    price DECIMAL(10,2) NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    category_id INT UNSIGNED,

    -- Primary Key
    PRIMARY KEY (id),

    -- Unique Constraint
    UNIQUE KEY unique_sku (sku),

    -- Check Constraint (MySQL 8.0.16+)
    CHECK (price >= 0),
    CHECK (stock_quantity >= 0),

    -- Foreign Key
    FOREIGN KEY (category_id)
        REFERENCES categories(id)
        ON DELETE SET NULL
        ON UPDATE CASCADE,

    -- Indexes
    INDEX idx_category (category_id),
    INDEX idx_name (name),
    FULLTEXT INDEX idx_description (description)
) ENGINE=InnoDB;
```

Foreign Key Actions

Action	Description
CASCADE	Delete/update parent = delete/update children
SET NULL	Delete/update parent = set child FK to NULL
RESTRICT	Prevent delete/update if children exist (default)
NO ACTION	Same as RESTRICT
SET DEFAULT	Set to default value (not supported in InnoDB)

```sql
CREATE TABLE orders (

```
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    user_id INT UNSIGNED NOT NULL,
    order_number VARCHAR(20) NOT NULL UNIQUE,
    total_amount DECIMAL(10,2) NOT NULL,
    status ENUM('pending', 'processing', 'shipped', 'delivered', 'cancelled')
        DEFAULT 'pending',
    notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

    -- Foreign key with CASCADE
    FOREIGN KEY (user_id)
        REFERENCES users(id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE,

    -- Composite index for user's orders sorted by date
    INDEX idx_user_orders (user_id, created_at),
    INDEX idx_status (status),
    INDEX idx_order_number (order_number)
) ENGINE=InnoDB;

CREATE TABLE order_items (
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    order_id BIGINT UNSIGNED NOT NULL,
    product_id INT UNSIGNED NOT NULL,
    quantity INT UNSIGNED NOT NULL,
    price DECIMAL(10,2) NOT NULL,

    -- Cascade delete when order is deleted
    FOREIGN KEY (order_id)
        REFERENCES orders(id)
        ON DELETE CASCADE,

    -- Restrict if product is in orders
    FOREIGN KEY (product_id)
        REFERENCES products(id)
        ON DELETE RESTRICT,

    INDEX idx_order (order_id),
    INDEX idx_product (product_id)
) ENGINE=InnoDB;
```

Table Alteration
```sql
-- Add column
```

```
ALTER TABLE users ADD COLUMN middle_name VARCHAR(50) AFTER first_name;

-- Add column with constraint
ALTER TABLE users ADD COLUMN age TINYINT UNSIGNED CHECK (age >= 18);

-- Modify column type
ALTER TABLE users MODIFY COLUMN phone VARCHAR(25);

-- Change column name and type
ALTER TABLE users CHANGE COLUMN phone phone_number VARCHAR(25);

-- Drop column
ALTER TABLE users DROP COLUMN middle_name;

-- Add index
ALTER TABLE users ADD INDEX idx_last_name (last_name);

-- Add unique constraint
ALTER TABLE users ADD UNIQUE KEY unique_phone (phone_number);

-- Add foreign key
ALTER TABLE orders
ADD CONSTRAINT fk_user
FOREIGN KEY (user_id) REFERENCES users(id);

-- Drop foreign key
ALTER TABLE orders DROP FOREIGN KEY fk_user;

-- Drop index
ALTER TABLE users DROP INDEX idx_last_name;

-- Rename table
ALTER TABLE users RENAME TO customers;
RENAME TABLE customers TO users;

-- Change storage engine
ALTER TABLE users ENGINE = InnoDB;

-- Add AUTO_INCREMENT
ALTER TABLE users MODIFY id INT UNSIGNED AUTO_INCREMENT;

-- Reset AUTO_INCREMENT
ALTER TABLE users AUTO_INCREMENT = 1000;
```

⚠️ Note: ALTER TABLE operations can lock the table on large datasets. For production databases with large tables, use tools like:

- `pt-online-schema-change` (Percona Toolkit)
- `gh-ost` (GitHub's online schema migration)
- MySQL 8.0's instant ADD COLUMN feature

Creating Tables from Queries
```sql
-- Create table from SELECT
CREATE TABLE active_users AS
SELECT id, username, email, created_at
FROM users
WHERE is_active = TRUE;

-- Create table structure only (no data)
CREATE TABLE users_backup LIKE users;

-- Copy data to backup table
INSERT INTO users_backup SELECT * FROM users;
```

Temporary Tables
```sql
-- Create temporary table (auto-deleted when session ends)
CREATE TEMPORARY TABLE temp_results (
    id INT,
    total DECIMAL(10,2)
);

-- Use temporary table
INSERT INTO temp_results
SELECT user_id, SUM(total_amount)
FROM orders
GROUP BY user_id;

SELECT * FROM temp_results;
-- Table automatically dropped when connection closes
```

Partitioning
```sql
-- Range partitioning by date
CREATE TABLE orders_partitioned (
    id BIGINT UNSIGNED AUTO_INCREMENT,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10,2),
    PRIMARY KEY (id, order_date)
) PARTITION BY RANGE (YEAR(order_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
);

-- List partitioning
CREATE TABLE users_region (
    id INT,
    name VARCHAR(50),
    region VARCHAR(20),
    PRIMARY KEY (id, region)
) PARTITION BY LIST COLUMNS(region) (
    PARTITION p_north VALUES IN('North', 'Northeast', 'Northwest'),
    PARTITION p_south VALUES IN('South', 'Southeast', 'Southwest'),
    PARTITION p_east VALUES IN('East'),
    PARTITION p_west VALUES IN('West')
);

-- Hash partitioning
CREATE TABLE logs (
    id BIGINT AUTO_INCREMENT,
    log_data TEXT,
    PRIMARY KEY (id)
) PARTITION BY HASH(id)
PARTITIONS 4;
```

Chapter 3: CRUD Operations

3.1 INSERT Operations

Basic INSERT

```sql
-- Single row insert
INSERT INTO users (username, email, password_hash, first_name, last_name)
VALUES ('johndoe', 'john@example.com', 'hashed_password', 'John', 'Doe');

-- Get last inserted ID
SELECT LAST_INSERT_ID();

-- Multiple rows insert (much more efficient)
INSERT INTO users (username, email, password_hash)
VALUES
    ('alice', 'alice@example.com', 'hash1'),
    ('bob', 'bob@example.com', 'hash2'),
```

```
    ('charlie', 'charlie@example.com', 'hash3');
```

✅ Best Practice: Use batch inserts instead of individual inserts. Inserting 1000 rows in one statement is typically 10-100x faster than 1000 separate INSERT statements.

INSERT with All Columns
```sql
-- If providing values for all columns in order
INSERT INTO users VALUES
    (NULL, 'dave', 'dave@example.com', 'hash4', 'Dave', 'Smith', '555-0100', '1990-01-01', TRUE, NOW(), NOW());
```

⚠️ Warning: This method is fragile. If table structure changes, INSERT breaks. Always specify column names explicitly.

INSERT from SELECT
```sql
-- Copy data from another table
INSERT INTO archived_users (id, username, email, created_at)
SELECT id, username, email, created_at
FROM users
WHERE created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR);

-- Insert with transformation
INSERT INTO user_summary (user_id, total_orders, total_spent)
SELECT
    user_id,
    COUNT(*) as order_count,
    SUM(total_amount) as total
FROM orders
GROUP BY user_id;
```

INSERT IGNORE
```sql
-- Ignore errors (like duplicate keys)
INSERT IGNORE INTO users (username, email, password_hash)
VALUES ('johndoe', 'john@example.com', 'hash');
-- If username 'johndoe' exists, this silently fails

-- Check affected rows
SELECT ROW_COUNT();  -- Returns 0 if ignored, 1 if inserted
```

⚠️ Caution: INSERT IGNORE suppresses ALL errors, not just duplicates. Use carefully.

INSERT ... ON DUPLICATE KEY UPDATE (Upsert)
```sql
-- Update if duplicate key found
INSERT INTO product_inventory (product_id, quantity)
VALUES (101, 50)
ON DUPLICATE KEY UPDATE
    quantity = quantity + VALUES(quantity),
    updated_at = CURRENT_TIMESTAMP;

-- More complex upsert
INSERT INTO user_stats (user_id, login_count, last_login)
VALUES (1, 1, NOW())
ON DUPLICATE KEY UPDATE
    login_count = login_count + 1,
    last_login = NOW();
```

REPLACE (Delete and Insert)
```sql
-- Delete existing row and insert new one
REPLACE INTO settings (key_name, value)
VALUES ('theme', 'dark');

-- Equivalent to:
-- DELETE FROM settings WHERE key_name = 'theme';
-- INSERT INTO settings (key_name, value) VALUES ('theme', 'dark');
```

⚠️ Warning: REPLACE deletes the old row (if exists) and inserts new one. This can affect:
- Foreign key relationships
- AUTO_INCREMENT values
- Triggers

INSERT with DEFAULT VALUES
```sql
-- Use default values explicitly
INSERT INTO users (username, email, password_hash, is_active)
VALUES ('newuser', 'new@example.com', 'hash', DEFAULT);

-- Insert row with all defaults
INSERT INTO log_entries () VALUES ();
```

3.2 SELECT Queries

Basic SELECT
```sql
```

```sql
-- Select specific columns
SELECT id, username, email FROM users;

-- Select all columns (avoid in production)
SELECT * FROM users;

-- With WHERE clause
SELECT * FROM users
WHERE is_active = TRUE;

-- Multiple conditions
SELECT * FROM users
WHERE is_active = TRUE
  AND created_at > '2024-01-01'
  AND email LIKE '%@gmail.com';

-- OR condition
SELECT * FROM users
WHERE city = 'New York'
  OR city = 'Los Angeles';

-- IN operator (cleaner than multiple ORs)
SELECT * FROM users
WHERE city IN ('New York', 'Los Angeles', 'Chicago');

-- NOT IN
SELECT * FROM users
WHERE status NOT IN ('banned', 'deleted');

-- BETWEEN
SELECT * FROM orders
WHERE total_amount BETWEEN 100 AND 500;

-- IS NULL / IS NOT NULL
SELECT * FROM users
WHERE phone IS NULL;

SELECT * FROM users
WHERE phone IS NOT NULL;
```

Pattern Matching

```sql
-- LIKE operator (% = any characters, _ = single character)
SELECT * FROM users
WHERE email LIKE '%@gmail.com';
```

```sql
SELECT * FROM products
WHERE name LIKE 'iPhone%';

SELECT * FROM users
WHERE phone LIKE '555-____';

-- Case-insensitive by default in MySQL
SELECT * FROM products
WHERE name LIKE '%laptop%';

-- Case-sensitive LIKE using BINARY
SELECT * FROM products
WHERE BINARY name LIKE '%Laptop%';

-- REGEXP/RLIKE for complex patterns
SELECT * FROM users
WHERE email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$';

SELECT * FROM users
WHERE phone REGEXP '^[0-9]{3}-[0-9]{3}-[0-9]{4}$';

-- NOT LIKE
SELECT * FROM users
WHERE email NOT LIKE '%@tempmail.com';
```

Performance Note: LIKE with leading wildcard (`%value`) cannot use index. Prefer `value%` when possible.

Sorting
```sql
-- ORDER BY ascending
SELECT * FROM products
ORDER BY price ASC;

-- ORDER BY descending
SELECT * FROM products
ORDER BY price DESC;

-- Multiple sort columns
SELECT * FROM users
ORDER BY last_name ASC, first_name ASC;

-- Sort by expression
SELECT * FROM products
ORDER BY (price * quantity) DESC;

-- Custom sort order with CASE
```

```sql
SELECT * FROM orders
ORDER BY
  CASE status
    WHEN 'urgent' THEN 1
    WHEN 'processing' THEN 2
    WHEN 'pending' THEN 3
    ELSE 4
  END;

-- NULL values first or last
SELECT * FROM users
ORDER BY phone IS NULL, phone;  -- NULLs first

SELECT * FROM users
ORDER BY phone IS NULL DESC, phone;  -- NULLs last

-- Random order
SELECT * FROM products
ORDER BY RAND()
LIMIT 10;
```

⚠️ Warning: `ORDER BY RAND()` is slow on large tables. Use alternative methods for random selection.

Limiting Results

```sql
-- LIMIT
SELECT * FROM products
ORDER BY created_at DESC
LIMIT 10;

-- LIMIT with OFFSET (pagination)
SELECT * FROM products
ORDER BY created_at DESC
LIMIT 20 OFFSET 40;  -- Page 3, 20 items per page

-- Alternative syntax
SELECT * FROM products
ORDER BY created_at DESC
LIMIT 40, 20;  -- LIMIT offset, count

-- Get top N per category (MySQL 8.0+)
SELECT * FROM (
  SELECT *,
    ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY price DESC) as rn
  FROM products
) t
```

```
WHERE rn <= 5;
```

Pagination Best Practice: For large offsets, use cursor-based pagination:
```sql
-- Instead of OFFSET 10000 (slow)
SELECT * FROM products
WHERE id > 10000
ORDER BY id
LIMIT 20;
```

DISTINCT
```sql
-- Get unique values
SELECT DISTINCT city FROM users;

-- Count unique values
SELECT COUNT(DISTINCT city) FROM users;

-- Multiple columns
SELECT DISTINCT city, state FROM users;

-- DISTINCT with aggregates
SELECT
    city,
    COUNT(DISTINCT email) as unique_emails
FROM users
GROUP BY city;
```

Column Aliases
```sql
-- AS keyword (optional but recommended)
SELECT
    first_name AS fname,
    last_name AS lname,
    CONCAT(first_name, ' ', last_name) AS full_name,
    YEAR(CURDATE()) - YEAR(date_of_birth) AS age
FROM users;

-- Without AS (works but less clear)
SELECT
    first_name fname,
    last_name lname
FROM users;
```

```sql
-- Table aliases
SELECT
    u.id,
    u.username,
    o.order_number
FROM users u
INNER JOIN orders o ON u.id = o.user_id;
```

Calculated Fields
```sql
SELECT
    product_name,
    price,
    tax_rate,
    price * tax_rate AS tax_amount,
    price + (price * tax_rate) AS total_price,
    ROUND(price * 0.9, 2) AS discounted_price
FROM products;

SELECT
    first_name,
    last_name,
    CONCAT(first_name, ' ', last_name) AS full_name,
    UPPER(email) AS email_upper,
    CHAR_LENGTH(username) AS username_length,
    DATE_FORMAT(created_at, '%M %d, %Y') AS formatted_date
FROM users;
```

CASE Statements
```sql
-- Simple CASE
SELECT
    product_name,
    stock_quantity,
    CASE
        WHEN stock_quantity = 0 THEN 'Out of Stock'
        WHEN stock_quantity < 10 THEN 'Low Stock'
        WHEN stock_quantity < 50 THEN 'In Stock'
        ELSE 'Well Stocked'
    END AS stock_status
FROM products;

-- CASE in calculations
SELECT
    order_id,
```

```
    total_amount,
    CASE
        WHEN total_amount > 1000 THEN total_amount * 0.9
        WHEN total_amount > 500 THEN total_amount * 0.95
        ELSE total_amount
    END AS final_price
FROM orders;

-- Searched CASE
SELECT
    username,
    CASE
        WHEN login_count > 100 THEN 'Power User'
        WHEN login_count > 50 THEN 'Regular User'
        WHEN login_count > 10 THEN 'Casual User'
        ELSE 'New User'
    END AS user_type
FROM user_stats;
```

IF Function
```sql
-- Simple IF
SELECT
    username,
    IF(is_active, 'Active', 'Inactive') AS status
FROM users;

-- Nested IF
SELECT
    product_name,
    price,
    IF(price > 100,
        'Premium',
        IF(price > 50, 'Standard', 'Budget')
    ) AS price_tier
FROM products;
```

COALESCE and IFNULL
```sql
-- Return first non-NULL value
SELECT
    username,
    COALESCE(phone, email, 'No contact info') AS contact
FROM users;
```

```sql
-- IFNULL (only 2 arguments)
SELECT
    product_name,
    IFNULL(discount_price, price) AS final_price
FROM products;

-- NULLIF (returns NULL if equal)
SELECT
    username,
    NULLIF(bio, '') AS bio  -- Returns NULL if bio is empty string
FROM users;
```

3.3 UPDATE Statements

Basic UPDATE

```sql
-- Update single column
UPDATE users
SET email = 'newemail@example.com'
WHERE id = 1;

-- Update multiple columns
UPDATE users
SET
    first_name = 'John',
    last_name = 'Smith',
    updated_at = CURRENT_TIMESTAMP
WHERE id = 1;

-- Update based on condition
UPDATE products
SET is_featured = TRUE
WHERE rating > 4.5 AND stock_quantity > 0;
```

🚨 CRITICAL WARNING: Always use WHERE clause with UPDATE! Without it, ALL rows will be updated!

```sql
-- Enable safe updates mode (recommended for development)
SET sql_safe_updates = 1;

-- This will now error if you forget WHERE
UPDATE users SET email = 'test@example.com';  -- ERROR!
```

UPDATE with Calculations

```sql
```

```sql
-- Increment value
UPDATE product_inventory
SET quantity = quantity + 10
WHERE product_id = 101;

-- Decrement
UPDATE users
SET login_count = login_count - 1
WHERE id = 5;

-- Apply percentage
UPDATE products
SET price = price * 1.1  -- 10% increase
WHERE category = 'Electronics';

-- Apply discount with ROUND
UPDATE products
SET price = ROUND(price * 0.9, 2)
WHERE category = 'Clearance';
```

UPDATE with CASE
```sql
-- Conditional updates
UPDATE products
SET status = CASE
    WHEN stock_quantity = 0 THEN 'out_of_stock'
    WHEN stock_quantity < 10 THEN 'low_stock'
    ELSE 'in_stock'
END;

-- Multiple columns with CASE
UPDATE employees
SET
    salary = CASE
        WHEN performance_rating = 5 THEN salary * 1.15
        WHEN performance_rating = 4 THEN salary * 1.10
        WHEN performance_rating = 3 THEN salary * 1.05
        ELSE salary
    END,
    bonus = CASE
        WHEN performance_rating >= 4 THEN salary * 0.2
        ELSE 0
    END
WHERE review_date = CURDATE();
```

UPDATE with JOIN

```sql
-- Update based on related table
UPDATE orders o
INNER JOIN users u ON o.user_id = u.id
SET o.customer_name = CONCAT(u.first_name, ' ', u.last_name)
WHERE o.customer_name IS NULL;

-- Update with multiple joins
UPDATE products p
INNER JOIN categories c ON p.category_id = c.id
INNER JOIN suppliers s ON p.supplier_id = s.id
SET p.full_description = CONCAT(c.name, ' - ', p.name, ' by ', s.name);

-- Update with LEFT JOIN (update all from left table)
UPDATE users u
LEFT JOIN orders o ON u.id = o.user_id AND o.status = 'completed'
SET u.has_completed_order = IF(o.id IS NOT NULL, TRUE, FALSE);
```

UPDATE with Subquery

```sql
-- Update based on aggregate from another table
UPDATE users u
SET u.total_spent = (
    SELECT COALESCE(SUM(total_amount), 0)
    FROM orders
    WHERE user_id = u.id AND status = 'completed'
);

-- Update using subquery condition
UPDATE products
SET is_bestseller = TRUE
WHERE id IN (
    SELECT product_id
    FROM order_items
    GROUP BY product_id
    HAVING SUM(quantity) > 1000
);

-- Update to average value
UPDATE products
SET price = (
    SELECT AVG(price)
    FROM products p2
    WHERE p2.category_id = products.category_id
)
```

```
WHERE price IS NULL;
```

UPDATE with LIMIT
```sql
-- Update only first N rows
UPDATE products
SET is_promoted = TRUE
WHERE is_promoted = FALSE
ORDER BY rating DESC
LIMIT 10;

-- Useful for batch updates
UPDATE large_table
SET processed = TRUE
WHERE processed = FALSE
LIMIT 1000;
```

UPDATE with ORDER BY
```sql
-- Update in specific order (useful with LIMIT)
UPDATE users
SET membership_tier = 'gold'
WHERE membership_tier = 'silver'
ORDER BY total_spent DESC
LIMIT 100;
```

3.4 DELETE Operations

Basic DELETE
```sql
-- Delete specific rows
DELETE FROM users
WHERE id = 1;

-- Delete with multiple conditions
DELETE FROM logs
WHERE created_at < DATE_SUB(NOW(), INTERVAL 90 DAY)
    AND status = 'archived';

-- Delete all rows (keep table structure)
DELETE FROM temp_data;
```

🚨 CRITICAL WARNING: Always use WHERE with DELETE! Enable safe updates:

```sql
SET sql_safe_updates = 1;
```

DELETE with JOIN
```sql
-- Delete from single table based on join
DELETE o
FROM orders o
INNER JOIN users u ON o.user_id = u.id
WHERE u.status = 'deleted';

-- Delete from multiple tables
DELETE o, oi
FROM orders o
INNER JOIN order_items oi ON o.id = oi.order_id
WHERE o.status = 'cancelled'
  AND o.created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR);

-- Delete using LEFT JOIN (orphaned records)
DELETE oi
FROM order_items oi
LEFT JOIN orders o ON oi.order_id = o.id
WHERE o.id IS NULL;
```

DELETE with Subquery
```sql
-- Delete based on subquery
DELETE FROM products
WHERE id IN (
  SELECT product_id
  FROM discontinued_items
);

-- Delete based on aggregate condition
DELETE FROM users
WHERE id IN (
  SELECT user_id
  FROM orders
  GROUP BY user_id
  HAVING COUNT(*) = 0
) AND created_at < DATE_SUB(NOW(), INTERVAL 2 YEAR);
```

DELETE with LIMIT
```sql
```

```sql
-- Delete in batches (safer for large tables)
DELETE FROM logs
WHERE created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR)
LIMIT 10000;

-- Can run multiple times until all deleted
-- Check affected rows
SELECT ROW_COUNT();
```

TRUNCATE vs DELETE

```sql
-- DELETE: Slower, logged, can have WHERE, can ROLLBACK
DELETE FROM temp_data WHERE id < 1000;

-- TRUNCATE: Fast, minimal logging, removes ALL rows, resets AUTO_INCREMENT
TRUNCATE TABLE temp_data;
```

TRUNCATE vs DELETE comparison:

Feature	DELETE	TRUNCATE
WHERE clause	✅ Yes	❌ No
Speed	Slower	Faster
Rollback	✅ Yes	❌ No (in most cases)
Triggers	✅ Fires	❌ Doesn't fire
Reset AUTO_INCREMENT	❌ No	✅ Yes
Use case	Selective deletion	Clear entire table

Soft Deletes (Best Practice)

Instead of physically deleting data, use soft deletes:

```sql
-- Add deleted_at column
ALTER TABLE users
ADD COLUMN deleted_at TIMESTAMP NULL DEFAULT NULL;

-- Add index for performance
CREATE INDEX idx_deleted_at ON users(deleted_at);

-- Soft delete
UPDATE users
SET deleted_at = CURRENT_TIMESTAMP
WHERE id = 1;

-- Query only active records
```

```
SELECT * FROM users
WHERE deleted_at IS NULL;

-- Include deleted records
SELECT * FROM users;

-- Restore soft-deleted record
UPDATE users
SET deleted_at = NULL
WHERE id = 1;

-- Permanently delete (hard delete)
DELETE FROM users
WHERE deleted_at IS NOT NULL
    AND deleted_at < DATE_SUB(NOW(), INTERVAL 30 DAY);
```

✅ Benefits of Soft Deletes:
- Data recovery possible
- Audit trail maintained
- Foreign key relationships preserved
- Can analyze deleted data
- Gradual cleanup possible

❌ Considerations:
- Queries must filter `deleted_at IS NULL`
- Indexes include deleted rows
- Storage continues to grow
- Unique constraints need adjustment

```sql
-- Create view for active records
CREATE VIEW active_users AS
SELECT * FROM users
WHERE deleted_at IS NULL;

-- Use view in queries
SELECT * FROM active_users WHERE city = 'New York';
```

Chapter 4: Advanced Queries

4.1 JOINs Mastery

JOINs combine rows from two or more tables based on related columns.

INNER JOIN

Returns only rows that have matching values in both tables.

```sql
-- Basic INNER JOIN
SELECT
    u.id,
    u.username,
    o.order_number,
    o.total_amount,
    o.created_at AS order_date
FROM users u
INNER JOIN orders o ON u.id = o.user_id
WHERE o.status = 'delivered'
ORDER BY o.created_at DESC;

-- Multiple conditions in JOIN
SELECT
    u.username,
    o.order_number
FROM users u
INNER JOIN orders o
    ON u.id = o.user_id
    AND o.created_at >= '2024-01-01'
    AND o.status != 'cancelled';

-- Using USING (when column names match)
SELECT
    u.username,
    o.order_number
FROM users u
INNER JOIN orders o USING(user_id);
```

LEFT JOIN (LEFT OUTER JOIN)

Returns all rows from left table and matched rows from right. NULL for non-matches.

```sql
-- Find all users and their order count (including users with no orders)
SELECT
    u.id,
    u.username,
    COUNT(o.id) AS order_count,
    COALESCE(SUM(o.total_amount), 0) AS total_spent
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.username;
```

```sql
-- Find users with NO orders
SELECT
    u.id,
    u.username,
    u.email
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE o.id IS NULL;

-- Find products never ordered
SELECT p.id, p.product_name
FROM products p
LEFT JOIN order_items oi ON p.id = oi.product_id
WHERE oi.id IS NULL;
```

RIGHT JOIN (RIGHT OUTER JOIN)

Returns all rows from right table and matched rows from left.

```sql
-- All orders with user info (even if user deleted)
SELECT
    o.order_number,
    o.total_amount,
    u.username,
    u.email
FROM users u
RIGHT JOIN orders o ON u.id = o.user_id;

-- Note: RIGHT JOIN is less common; can always be rewritten as LEFT JOIN
```

FULL OUTER JOIN

MySQL doesn't support FULL OUTER JOIN directly. Use UNION:

```sql
-- Simulate FULL OUTER JOIN
SELECT u.id, u.username, o.order_number
FROM users u
LEFT JOIN orders o ON u.id = o.user_id

UNION

SELECT u.id, u.username, o.order_number
FROM users u
RIGHT JOIN orders o ON u.id = o.user_id;
```

```

#### CROSS JOIN

Cartesian product - every row from table1 paired with every row from table2.

```sql
-- Generate all combinations
SELECT
    c.color_name,
    s.size_name,
    CONCAT(c.color_name, ' - ', s.size_name) AS variant
FROM colors c
CROSS JOIN sizes s;

-- Useful for generating test data
SELECT
    CONCAT('user', n1.num, n2.num) AS username
FROM
    (SELECT 0 AS num UNION SELECT 1 UNION SELECT 2) n1
CROSS JOIN
    (SELECT 0 AS num UNION SELECT 1 UNION SELECT 2) n2;
```

SELF JOIN

Join table to itself.

```sql
-- Employee hierarchy
SELECT
    e.name AS employee,
    m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.id;

-- Find users from same city
SELECT
    u1.username AS user1,
    u2.username AS user2,
    u1.city
FROM users u1
INNER JOIN users u2
    ON u1.city = u2.city
    AND u1.id < u2.id  -- Avoid duplicates
ORDER BY u1.city;

-- Find products in same price range
SELECT
```
```

```
    p1.product_name AS product1,
    p2.product_name AS product2,
    p1.price
FROM products p1
INNER JOIN products p2
    ON ABS(p1.price - p2.price) < 10
    AND p1.id < p2.id;
```

Multiple JOINs
```sql
-- Complex query with multiple joins
SELECT
    o.order_number,
    u.username,
    u.email,
    p.product_name,
    c.category_name,
    oi.quantity,
    oi.price,
    (oi.quantity * oi.price) AS line_total
FROM orders o
INNER JOIN users u ON o.user_id = u.id
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
INNER JOIN categories c ON p.category_id = c.id
WHERE o.status = 'delivered'
    AND o.created_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
ORDER BY o.created_at DESC;

-- Mix of LEFT and INNER JOINs
SELECT
    u.username,
    u.email,
    COUNT(DISTINCT o.id) AS order_count,
    COUNT(DISTINCT r.id) AS review_count,
    AVG(r.rating) AS avg_rating
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
LEFT JOIN reviews r ON u.id = r.user_id
GROUP BY u.id, u.username, u.email
HAVING order_count > 0 OR review_count > 0;
```

Natural JOIN (Use with Caution)
```sql
-- Automatically joins on all columns with same name
```

```
SELECT *
FROM users
NATURAL JOIN orders;

-- ⚠️ Dangerous! Can produce unexpected results if schema changes
-- Better to be explicit with JOIN conditions
```

✅ JOIN Best Practices:

1. **Always use explicit JOIN syntax** (not WHERE clause joins)
2. **Index foreign key columns** for better performance
3. **Use table aliases** for readability
4. **Consider query execution order**: WHERE → JOIN → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
5. **Use EXPLAIN** to understand join strategies
6. **Avoid joining too many tables** (consider denormalization or summary tables)
7. **Filter early**: Put conditions in WHERE, not SELECT

```sql
-- Bad: Filtering in SELECT
SELECT
    u.username,
    IF(o.status = 'completed', o.total_amount, 0) AS completed_amount
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;

-- Good: Filter in JOIN condition
SELECT
    u.username,
    o.total_amount
FROM users u
LEFT JOIN orders o
    ON u.id = o.user_id
    AND o.status = 'completed';
```

4.2 Subqueries

A query nested inside another query.

Subquery in WHERE Clause

```sql
-- Find users who placed orders above average
SELECT username, email
FROM users
WHERE id IN (
    SELECT DISTINCT user_id
```

```
    FROM orders
    WHERE total_amount > (
        SELECT AVG(total_amount) FROM orders
    )
);

-- Find products more expensive than average in their category
SELECT
    product_name,
    price,
    category_id
FROM products p
WHERE price > (
    SELECT AVG(price)
    FROM products
    WHERE category_id = p.category_id
);
```

Subquery in SELECT Clause

```sql
-- Add calculated columns with subqueries
SELECT
    u.id,
    u.username,
    (SELECT COUNT(*) FROM orders WHERE user_id = u.id) AS total_orders,
    (SELECT SUM(total_amount) FROM orders WHERE user_id = u.id AND status = 'completed') AS lifetime_value,
    (SELECT MAX(created_at) FROM orders WHERE user_id = u.id) AS last_order_date
FROM users u
WHERE u.is_active = TRUE;
```

⚠️ Performance Warning: Scalar subqueries in SELECT execute once per row. Use JOINs for better performance:

```sql
-- Better performance with JOIN and GROUP BY
SELECT
    u.id,
    u.username,
    COUNT(o.id) AS total_orders,
    SUM(CASE WHEN o.status = 'completed' THEN o.total_amount ELSE 0 END) AS lifetime_value,
    MAX(o.created_at) AS last_order_date
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.is_active = TRUE
GROUP BY u.id, u.username;
```

Subquery in FROM Clause (Derived Table)

```sql
-- Use subquery as temporary table
SELECT
    category,
    AVG(product_count) AS avg_products_per_supplier
FROM (
    SELECT
        category_id AS category,
        supplier_id,
        COUNT(*) AS product_count
    FROM products
    GROUP BY category_id, supplier_id
) AS category_supplier_stats
GROUP BY category;

-- Find top customers by month
SELECT
    order_month,
    username,
    total_spent
FROM (
    SELECT
        DATE_FORMAT(o.created_at, '%Y-%m') AS order_month,
        u.username,
        SUM(o.total_amount) AS total_spent,
        RANK() OVER (
            PARTITION BY DATE_FORMAT(o.created_at, '%Y-%m')
            ORDER BY SUM(o.total_amount) DESC
        ) AS monthly_rank
    FROM orders o
    INNER JOIN users u ON o.user_id = u.id
    GROUP BY DATE_FORMAT(o.created_at, '%Y-%m'), u.id, u.username
) AS monthly_rankings
WHERE monthly_rank <= 10;
```

Correlated Subquery

Subquery references columns from outer query (executes once per outer row).

```sql
-- Find products priced above their category average
SELECT
    p.product_name,
    p.price,
    (SELECT AVG(price) FROM products WHERE category_id = p.category_id) AS category_avg
FROM products p
```

```sql
WHERE p.price > (
    SELECT AVG(price)
    FROM products
    WHERE category_id = p.category_id
);

-- Find users with more than 5 orders
SELECT username
FROM users u
WHERE (
    SELECT COUNT(*)
    FROM orders
    WHERE user_id = u.id
) > 5;
```

EXISTS and NOT EXISTS

More efficient than IN for checking existence.

```sql
-- EXISTS: Stops at first match
SELECT username
FROM users u
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.id
        AND o.total_amount > 1000
);

-- NOT EXISTS: Find users without orders
SELECT username
FROM users u
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.id
);

-- Complex EXISTS example
SELECT p.product_name
FROM products p
WHERE EXISTS (
    SELECT 1
    FROM order_items oi
    INNER JOIN orders o ON oi.order_id = o.id
    WHERE oi.product_id = p.id
```

```
        AND o.created_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
        AND o.status = 'delivered'
);
```

IN vs EXISTS
```sql
-- IN: Evaluates subquery completely, stores results
SELECT username
FROM users
WHERE id IN (
    SELECT user_id
    FROM orders
    WHERE total_amount > 1000
);

-- EXISTS: Short-circuits at first match (usually faster)
SELECT username
FROM users u
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.user_id = u.id
        AND o.total_amount > 1000
);
```

Performance Guideline:
- Use EXISTS when subquery returns many rows
- Use IN when subquery returns few rows or constant list
- EXISTS stops at first match (faster for correlated queries)
- IN must evaluate all results before comparison

ALL, ANY, SOME
```sql
-- ANY/SOME: True if comparison is true for any value
SELECT product_name, price
FROM products
WHERE price > ANY (
    SELECT price
    FROM products
    WHERE category_id = 5
);

-- ALL: True if comparison is true for all values
SELECT product_name, price
FROM products
```

```
WHERE price > ALL (
    SELECT price
    FROM products
    WHERE category_id = 5
);

-- Equivalent to
SELECT product_name, price
FROM products
WHERE price > (
    SELECT MAX(price)
    FROM products
    WHERE category_id = 5
);
```

✅ Subquery Best Practices:

1. **Use JOINs instead of subqueries when possible** (usually faster)
2. **Use EXISTS instead of IN** for correlated subqueries
3. **Avoid subqueries in SELECT clause** (executes per row)
4. **Index columns used in subquery conditions**
5. **Use EXPLAIN to compare performance** between subqueries and JOINs

4.3 Aggregate Functions

Perform calculations on sets of rows.

Basic Aggregates

```sql
-- COUNT
SELECT COUNT(*) AS total_users FROM users;
SELECT COUNT(DISTINCT city) AS unique_cities FROM users;
SELECT COUNT(phone) AS users_with_phone FROM users;  -- Excludes NULLs

-- SUM
SELECT SUM(total_amount) AS total_revenue FROM orders;
SELECT SUM(quantity * price) AS total_value FROM order_items;

-- AVG
SELECT AVG(price) AS average_price FROM products;
SELECT AVG(rating) AS average_rating FROM reviews WHERE rating IS NOT NULL;

-- MIN and MAX
SELECT
    MIN(price) AS cheapest,
    MAX(price) AS most_expensive,
```

```
    MAX(price) - MIN(price) AS price_range
FROM products;

-- Multiple aggregates
SELECT
    COUNT(*) AS total_orders,
    SUM(total_amount) AS revenue,
    AVG(total_amount) AS avg_order_value,
    MIN(total_amount) AS smallest_order,
    MAX(total_amount) AS largest_order
FROM orders
WHERE status = 'completed';
```

GROUP BY

Group rows with same values into summary rows.

```sql
-- Basic GROUP BY
SELECT
    status,
    COUNT(*) AS order_count,
    SUM(total_amount) AS total_revenue
FROM orders
GROUP BY status;

-- Multiple columns
SELECT
    DATE(created_at) AS order_date,
    status,
    COUNT(*) AS order_count,
    AVG(total_amount) AS avg_order_value
FROM orders
GROUP BY DATE(created_at), status
ORDER BY order_date DESC, status;

-- With JOIN
SELECT
    c.category_name,
    COUNT(p.id) AS product_count,
    AVG(p.price) AS avg_price,
    SUM(p.stock_quantity) AS total_stock
FROM categories c
LEFT JOIN products p ON c.id = p.category_id
GROUP BY c.id, c.category_name
ORDER BY product_count DESC;
```

Important: All non-aggregated columns in SELECT must be in GROUP BY (unless using ONLY_FULL_GROUP_BY mode is disabled).

```sql
-- Modern MySQL (ONLY_FULL_GROUP_BY enabled)
SELECT
    category_id,
    category_name,  -- Must be in GROUP BY
    COUNT(*) AS product_count
FROM products
GROUP BY category_id, category_name;

-- Check SQL mode
SELECT @@sql_mode;
```

HAVING Clause

Filter groups (use HAVING for aggregates, WHERE for rows).

```sql
-- Find users with more than 5 orders
SELECT
    user_id,
    COUNT(*) AS order_count,
    SUM(total_amount) AS total_spent
FROM orders
WHERE status = 'completed'
GROUP BY user_id
HAVING order_count > 5
    AND total_spent > 1000
ORDER BY total_spent DESC;

-- Categories with average price above threshold
SELECT
    category_id,
    COUNT(*) AS product_count,
    AVG(price) AS avg_price
FROM products
WHERE is_active = TRUE
GROUP BY category_id
HAVING avg_price > 100
    AND product_count >= 5;
```

WHERE vs HAVING:
- **WHERE**: Filters rows BEFORE grouping
- **HAVING**: Filters groups AFTER grouping

- Use WHERE when possible (better performance)

```sql
-- Good: Filter before grouping
SELECT category_id, COUNT(*) AS active_products
FROM products
WHERE is_active = TRUE
GROUP BY category_id;

-- Bad: Filter after grouping (slower)
SELECT category_id, COUNT(*) AS product_count
FROM products
GROUP BY category_id
HAVING SUM(is_active) = COUNT(*);
```

GROUP_CONCAT

Concatenate values from multiple rows into single string.

```sql
-- Basic GROUP_CONCAT
SELECT
    category_id,
    GROUP_CONCAT(product_name) AS products
FROM products
GROUP BY category_id;

-- With separator
SELECT
    category_id,
    GROUP_CONCAT(product_name SEPARATOR ' | ') AS products
FROM products
GROUP BY category_id;

-- With ORDER BY
SELECT
    category_id,
    GROUP_CONCAT(
        product_name
        ORDER BY price DESC
        SEPARATOR ', '
    ) AS products_by_price
FROM products
GROUP BY category_id;

-- With DISTINCT
SELECT
    order_id,
```

```
    GROUP_CONCAT(DISTINCT product_name ORDER BY product_name) AS unique_products
FROM order_items oi
INNER JOIN products p ON oi.product_id = p.id
GROUP BY order_id;

-- Control max length
SET GROUP_CONCAT_MAX_LEN = 10000;
```

WITH ROLLUP

Add subtotals and grand totals to GROUP BY.

```sql
-- Basic ROLLUP
SELECT
    category,
    subcategory,
    SUM(quantity) AS total_quantity
FROM inventory
GROUP BY category, subcategory WITH ROLLUP;

-- Results:
-- Electronics  Phones      100
-- Electronics  Laptops     50
-- Electronics  NULL        150  (subtotal for Electronics)
-- Clothing     Shirts      200
-- Clothing     Pants       150
-- Clothing     NULL        350  (subtotal for Clothing)
-- NULL         NULL        500  (grand total)

-- Make NULL labels meaningful
SELECT
    COALESCE(category, 'TOTAL') AS category,
    COALESCE(subcategory, 'Subtotal') AS subcategory,
    SUM(quantity) AS total_quantity
FROM inventory
GROUP BY category, subcategory WITH ROLLUP;
```

Statistical Functions

```sql
-- Standard deviation
SELECT
    category_id,
    AVG(price) AS avg_price,
    STDDEV(price) AS std_dev,
    VARIANCE(price) AS variance
```

```
FROM products
GROUP BY category_id;

-- Bit operations
SELECT
    BIT_AND(permissions) AS common_permissions,
    BIT_OR(permissions) AS any_permissions,
    BIT_XOR(permissions) AS xor_permissions
FROM user_roles
WHERE group_id = 1;
```

4.4 Window Functions

Perform calculations across a set of rows related to the current row (MySQL 8.0+).

ROW_NUMBER

Assigns unique sequential number to each row.

```sql
-- Basic row numbering
SELECT
    product_name,
    price,
    ROW_NUMBER() OVER (ORDER BY price DESC) AS price_rank
FROM products;

-- Numbered within groups
SELECT
    category_id,
    product_name,
    price,
    ROW_NUMBER() OVER (
        PARTITION BY category_id
        ORDER BY price DESC
    ) AS category_rank
FROM products;
```

RANK and DENSE_RANK

```sql
-- RANK: Gaps in sequence for ties (1, 2, 2, 4)
-- DENSE_RANK: No gaps (1, 2, 2, 3)
SELECT
    product_name,
    price,
    RANK() OVER (ORDER BY price DESC) AS rank_with_gaps,
```

```
    DENSE_RANK() OVER (ORDER BY price DESC) AS rank_no_gaps,
    ROW_NUMBER() OVER (ORDER BY price DESC) AS row_num
FROM products;

-- Example results:
-- iPhone 15    $999    1    1    1
-- Galaxy S23   $999    1    1    2  (same price, tied rank)
-- Pixel 8      $899    3    2    3  (RANK skips 2, DENSE_RANK doesn't)
```

PARTITION BY

Divide result set into partitions and apply window function to each.

```sql
-- Rank products within each category
SELECT
    category_name,
    product_name,
    price,
    ROW_NUMBER() OVER (
        PARTITION BY category_name
        ORDER BY price DESC
    ) AS category_rank
FROM products p
INNER JOIN categories c ON p.category_id = c.id;

-- Top 3 products per category
SELECT *
FROM (
    SELECT
        category_id,
        product_name,
        price,
        ROW_NUMBER() OVER (
            PARTITION BY category_id
            ORDER BY price DESC
        ) AS rn
    FROM products
) ranked
WHERE rn <= 3;
```

Running Totals (Cumulative Sum)

```sql
-- Basic running total
SELECT
    order_date,
```

```sql
    daily_revenue,
    SUM(daily_revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_revenue
FROM daily_sales;

-- Shorter syntax (same result)
SELECT
    order_date,
    daily_revenue,
    SUM(daily_revenue) OVER (ORDER BY order_date) AS cumulative_revenue
FROM daily_sales;

-- Running total by category
SELECT
    category_id,
    order_date,
    daily_revenue,
    SUM(daily_revenue) OVER (
        PARTITION BY category_id
        ORDER BY order_date
    ) AS category_cumulative
FROM daily_sales;
```

Moving Averages

```sql
-- 7-day moving average
SELECT
    order_date,
    revenue,
    AVG(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS moving_avg_7_days
FROM daily_revenue;

-- 3-day moving average (previous, current, next)
SELECT
    order_date,
    revenue,
    AVG(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS moving_avg_3_days
FROM daily_revenue;
```

```sql
-- Moving sum
SELECT
    order_date,
    revenue,
    SUM(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
    ) AS rolling_30_day_revenue
FROM daily_revenue;
```

LAG and LEAD

Access previous and next rows without self-join.

```sql
-- Compare with previous day
SELECT
    order_date,
    revenue,
    LAG(revenue, 1) OVER (ORDER BY order_date) AS previous_day_revenue,
    revenue - LAG(revenue, 1) OVER (ORDER BY order_date) AS day_over_day_change,
    ROUND(
        (revenue - LAG(revenue, 1) OVER (ORDER BY order_date)) /
        LAG(revenue, 1) OVER (ORDER BY order_date) * 100,
        2
    ) AS percent_change
FROM daily_revenue;

-- Compare with next day
SELECT
    order_date,
    revenue,
    LEAD(revenue, 1) OVER (ORDER BY order_date) AS next_day_revenue
FROM daily_revenue;

-- Default value for missing rows
SELECT
    order_date,
    revenue,
    LAG(revenue, 1, 0) OVER (ORDER BY order_date) AS previous_day_revenue
FROM daily_revenue;

-- Look back 7 days
SELECT
    order_date,
    revenue,
```

```
    LAG(revenue, 7) OVER (ORDER BY order_date) AS week_ago_revenue
FROM daily_revenue;
```

FIRST_VALUE and LAST_VALUE

```sql
-- Compare each product to cheapest/most expensive in category
SELECT
    category_id,
    product_name,
    price,
    FIRST_VALUE(price) OVER (
        PARTITION BY category_id
        ORDER BY price
    ) AS cheapest_in_category,
    LAST_VALUE(price) OVER (
        PARTITION BY category_id
        ORDER BY price
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS most_expensive_in_category
FROM products;

-- Get first and last order date for each user
SELECT DISTINCT
    user_id,
    FIRST_VALUE(order_date) OVER (
        PARTITION BY user_id
        ORDER BY order_date
    ) AS first_order,
    LAST_VALUE(order_date) OVER (
        PARTITION BY user_id
        ORDER BY order_date
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_order
FROM orders;
```

NTH_VALUE

```sql
-- Get 2nd highest price in each category
SELECT DISTINCT
    category_id,
    NTH_VALUE(price, 2) OVER (
        PARTITION BY category_id
        ORDER BY price DESC
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS second_highest_price
```

```
FROM products;
```

NTILE

Divide rows into N buckets.

```sql
-- Divide products into 4 price quartiles
SELECT
    product_name,
    price,
    NTILE(4) OVER (ORDER BY price) AS price_quartile
FROM products;

-- Results:
-- Quartile 1: Bottom 25% (cheapest)
-- Quartile 2: 25-50%
-- Quartile 3: 50-75%
-- Quartile 4: Top 25% (most expensive)

-- Customer segmentation by spending
SELECT
    user_id,
    total_spent,
    NTILE(3) OVER (ORDER BY total_spent DESC) AS customer_tier,
    CASE NTILE(3) OVER (ORDER BY total_spent DESC)
        WHEN 1 THEN 'High Value'
        WHEN 2 THEN 'Medium Value'
        WHEN 3 THEN 'Low Value'
    END AS tier_label
FROM (
    SELECT user_id, SUM(total_amount) AS total_spent
    FROM orders
    GROUP BY user_id
) user_spending;
```

PERCENT_RANK and CUME_DIST

```sql
-- Percentile rank (0 to 1)
SELECT
    product_name,
    price,
    PERCENT_RANK() OVER (ORDER BY price) AS percentile_rank,
    ROUND(PERCENT_RANK() OVER (ORDER BY price) * 100, 2) AS percentile
FROM products;
```

```sql
-- Cumulative distribution
SELECT
    product_name,
    price,
    CUME_DIST() OVER (ORDER BY price) AS cumulative_dist,
    ROUND(CUME_DIST() OVER (ORDER BY price) * 100, 2) AS percent_below_or_equal
FROM products;
```

Frame Specifications

Control which rows are included in window function calculation.

```sql
-- ROWS: Physical rows
-- RANGE: Logical range (all rows with same ORDER BY value)

-- Current row only
SELECT
    date, value,
    SUM(value) OVER (
        ORDER BY date
        ROWS BETWEEN CURRENT ROW AND CURRENT ROW
    ) AS current_only
FROM data;

-- All preceding rows
SELECT
    date, value,
    SUM(value) OVER (
        ORDER BY date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM data;

-- 3 rows before and after
SELECT
    date, value,
    AVG(value) OVER (
        ORDER BY date
        ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING
    ) AS moving_avg_7
FROM data;

-- All rows in partition
SELECT
    category_id, value,
    SUM(value) OVER (
```

```
        PARTITION BY category_id
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS category_total
FROM data;
```

✅ Window Function Best Practices:

1. **Window functions don't reduce rows** (unlike GROUP BY)
2. **Executed after WHERE, GROUP BY, and HAVING**
3. **Can't be used in WHERE clause** (use subquery or CTE)
4. **More efficient than self-joins** for many use cases
5. **Use PARTITION BY to reset calculation per group**
6. **Named windows can reduce code duplication**

```sql
-- Named window
SELECT
    product_name,
    price,
    AVG(price) OVER w AS avg_price,
    MAX(price) OVER w AS max_price,
    MIN(price) OVER w AS min_price
FROM products
WINDOW w AS (PARTITION BY category_id);
```

Chapter 5: Indexing & Performance

5.1 Understanding Indexes

Indexes are data structures that improve query performance by allowing the database to find rows faster.

How Indexes Work

Think of an index like a book's index:
- **Without index**: Read every page to find a topic (full table scan)
- **With index**: Look up topic in index, jump directly to the right page

MySQL uses **B-Tree** indexes by default (balanced tree structure).

Performance Impact

```sql
-- Without index: Full table scan
SELECT * FROM users WHERE email = 'john@example.com';
-- Scans all 1,000,000 rows
```

```sql
-- With index on email: Index seek
CREATE INDEX idx_email ON users(email);
SELECT * FROM users WHERE email = 'john@example.com';
-- Reads only a few rows
```

When to Use Indexes

✅ Index these columns:
- Primary keys (auto-indexed)
- Foreign keys
- Columns in WHERE clauses
- Columns in JOIN conditions
- Columns in ORDER BY
- Columns in GROUP BY
- Columns with high cardinality (many unique values)

❌ Don't index:
- Small tables (< 1000 rows)
- Columns with low cardinality (few unique values like boolean, gender)
- Columns that are frequently updated
- Tables with heavy write operations (indexes slow down INSERT/UPDATE/DELETE)

Index Trade-offs

Pros:
- Faster SELECT queries
- Faster JOIN operations
- Enforces uniqueness (UNIQUE indexes)

Cons:
- Slower INSERT/UPDATE/DELETE (must update indexes)
- Uses disk space
- Too many indexes confuse optimizer

5.2 Index Types

Primary Key Index
```sql
-- Automatically created with PRIMARY KEY
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL
);

-- Composite primary key
```

```
CREATE TABLE order_items (
    order_id INT UNSIGNED,
    product_id INT UNSIGNED,
    quantity INT,
    PRIMARY KEY (order_id, product_id)
);
```

InnoDB: Primary key is the clustered index (table data is stored in primary key order).

Unique Index
```sql
-- Ensures column values are unique
CREATE UNIQUE INDEX idx_email ON users(email);

-- Or during table creation
CREATE TABLE users (
    id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    username VARCHAR(50),
    UNIQUE KEY unique_username (username)
);

-- Composite unique index
CREATE UNIQUE INDEX idx_user_product ON favorites(user_id, product_id);
```

Regular Index (B-Tree)
```sql
-- Single column index
CREATE INDEX idx_last_name ON users(last_name);

-- Composite index (multiple columns)
CREATE INDEX idx_name ON users(last_name, first_name);

-- The order matters!
-- This index helps:
-- ✅ WHERE last_name = 'Smith'
-- ✅ WHERE last_name = 'Smith' AND first_name = 'John'
-- ✅ ORDER BY last_name, first_name
--
-- Does NOT help:
-- ❌ WHERE first_name = 'John' (alone)
-- ❌ ORDER BY first_name

-- Add index to existing table
ALTER TABLE users ADD INDEX idx_city (city);
```

```sql
-- Index with sort order (MySQL 8.0+)
CREATE INDEX idx_created_desc ON posts(created_at DESC);
```

Prefix Index

For long VARCHAR or TEXT columns, index only the first N characters.

```sql
-- Index first 10 characters
CREATE INDEX idx_url_prefix ON pages(url(10));

-- Find optimal prefix length
SELECT
    COUNT(DISTINCT url) AS total_unique,
    COUNT(DISTINCT LEFT(url, 5)) AS prefix_5,
    COUNT(DISTINCT LEFT(url, 10)) AS prefix_10,
    COUNT(DISTINCT LEFT(url, 15)) AS prefix_15,
    COUNT(DISTINCT LEFT(url, 20)) AS prefix_20
FROM pages;

-- Choose prefix length where prefix_N ≈ total_unique
```

Trade-off: Saves space but can't be used for ORDER BY or covering index.

Covering Index

Index that includes all columns needed by query (no table lookup required).

```sql
-- Query
SELECT id, username, email FROM users WHERE username = 'johndoe';

-- Covering index includes all SELECT columns
CREATE INDEX idx_user_profile ON users(username, email, id);

-- MySQL can answer query using only the index (very fast!)
```

Check with EXPLAIN:

```sql
EXPLAIN SELECT id, username, email FROM users WHERE username = 'johndoe';
-- Extra: "Using index" = covering index used
```

Full-Text Index

For text searching (better than LIKE).

```sql
-- Create full-text index
CREATE FULLTEXT INDEX idx_content ON articles(title, body);

-- Search
SELECT * FROM articles
WHERE MATCH(title, body)
AGAINST('mysql performance' IN NATURAL LANGUAGE MODE);

-- Boolean mode (AND, OR, NOT)
SELECT * FROM articles
WHERE MATCH(title, body)
AGAINST('+mysql -mongodb' IN BOOLEAN MODE);
-- + means must include
-- - means must not include

-- Query expansion
SELECT * FROM articles
WHERE MATCH(title, body)
AGAINST('database' WITH QUERY EXPANSION);
```

Limitations:
- Only for CHAR, VARCHAR, TEXT columns
- Minimum word length: 4 characters (by default)
- Ignores common words (stopwords)
- InnoDB and MyISAM only

Spatial Index

For geographic data (GEOMETRY columns).

```sql
CREATE TABLE stores (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    location POINT NOT NULL,
    SPATIAL INDEX idx_location (location)
);

-- Find nearby stores
SELECT id, name,
    ST_Distance_Sphere(
        location,
        ST_GeomFromText('POINT(40.7128 -74.0060)')
    ) AS distance_meters
FROM stores
```

```
WHERE ST_Distance_Sphere(
    location,
    ST_GeomFromText('POINT(40.7128 -74.0060)')
) < 5000
ORDER BY distance_meters;
```

Descending Index (MySQL 8.0+)
```sql
-- Mixed sort orders
CREATE INDEX idx_score_date ON leaderboard(score DESC, created_at ASC);

-- Helps query:
SELECT * FROM leaderboard
ORDER BY score DESC, created_at ASC;
```

Invisible Index (MySQL 8.0+)

Test removing an index without actually dropping it.
```sql
-- Make index invisible (not used by optimizer)
ALTER TABLE users ALTER INDEX idx_city INVISIBLE;

-- Test queries
-- ...

-- Make visible again
ALTER TABLE users ALTER INDEX idx_city VISIBLE;

-- Or drop if not needed
DROP INDEX idx_city ON users;
```

Index Management
```sql
-- Show indexes on table
SHOW INDEX FROM users;

-- Show index usage (MySQL 8.0+)
SELECT * FROM sys.schema_unused_indexes WHERE object_schema = 'mydb';

-- Drop index
DROP INDEX idx_last_name ON users;

-- Rename index (MySQL 5.7+)
ALTER TABLE users RENAME INDEX old_name TO new_name;
```

```
-- Disable/enable keys (MyISAM only)
ALTER TABLE mytable DISABLE KEYS;
-- ... bulk insert
ALTER TABLE mytable ENABLE KEYS;

-- Rebuild index
ALTER TABLE users DROP INDEX idx_name, ADD INDEX idx_name(last_name, first_name);

-- Analyze table to update index statistics
ANALYZE TABLE users;
```

5.3 Query Optimization

Optimization Principles

1. **Select only needed columns**
2. **Use appropriate indexes**
3. **Filter early with WHERE**
4. **Limit result set**
5. **Avoid functions on indexed columns**
6. **Use proper join types**
7. **Cache results when appropriate**

Optimization Techniques

1. Select Only Needed Columns

```sql
-- Bad: Fetches all columns
SELECT * FROM users WHERE id = 1;

-- Good: Fetch only what you need
SELECT id, username, email FROM users WHERE id = 1;
```

Why it matters:
- Less data transferred
- Less memory used
- Enables covering indexes
- Faster network transmission

2. Use LIMIT

```sql
-- Without LIMIT: Reads all matching rows
SELECT * FROM users WHERE city = 'New York';
```

```sql
-- With LIMIT: Stops scanning after N rows found
SELECT * FROM users WHERE city = 'New York' LIMIT 100;

-- Existence check
SELECT 1 FROM orders WHERE user_id = 123 LIMIT 1;
-- Much faster than COUNT(*)
```

3. Avoid Functions on Indexed Columns
```sql
-- Bad: Can't use index on created_at
SELECT * FROM orders
WHERE YEAR(created_at) = 2024;

-- Good: Can use index
SELECT * FROM orders
WHERE created_at >= '2024-01-01'
  AND created_at < '2025-01-01';

-- Bad: Can't use index
SELECT * FROM users
WHERE LOWER(email) = 'john@example.com';

-- Good: Store email in lowercase, or use case-insensitive collation
SELECT * FROM users
WHERE email = 'john@example.com';
```

4. Use EXISTS Instead of COUNT for Existence Check
```sql
-- Bad: Counts all matching rows (slow)
SELECT IF(COUNT(*) > 0, 'exists', 'not exists')
FROM orders
WHERE user_id = 1;

-- Good: Stops at first match (fast)
SELECT IF(EXISTS(
  SELECT 1 FROM orders WHERE user_id = 1
), 'exists', 'not exists');
```

5. Optimize LIKE Queries
```sql
-- Bad: Leading wildcard can't use index
SELECT * FROM users WHERE email LIKE '%@gmail.com';

-- Good: Can use index
```

```sql
SELECT * FROM users WHERE email LIKE 'john%';

-- For full-text search, use FULLTEXT index
SELECT * FROM articles
WHERE MATCH(content) AGAINST('optimization');

-- Alternative for ends-with: reverse string index
ALTER TABLE users ADD COLUMN email_reversed VARCHAR(100)
    AS (REVERSE(email));
CREATE INDEX idx_email_reversed ON users(email_reversed);

SELECT * FROM users
WHERE email_reversed LIKE REVERSE('%@gmail.com');
```

6. Avoid OR with Different Columns
```sql
-- Bad: May not use indexes efficiently
SELECT * FROM users
WHERE first_name = 'John' OR last_name = 'Smith';

-- Better: Use UNION (can use separate indexes)
SELECT * FROM users WHERE first_name = 'John'
UNION
SELECT * FROM users WHERE last_name = 'Smith';

-- Or create a composite index
CREATE INDEX idx_names ON users(first_name, last_name);
```

7. Use JOIN Instead of Subquery (Usually)
```sql
-- Often slower: Subquery
SELECT * FROM users
WHERE id IN (
    SELECT user_id FROM orders WHERE total > 1000
);

-- Usually faster: JOIN
SELECT DISTINCT u.*
FROM users u
INNER JOIN orders o ON u.id = o.user_id
WHERE o.total > 1000;

-- But EXISTS can be faster than both for large sets
SELECT * FROM users u
WHERE EXISTS (
```

```
    SELECT 1 FROM orders o
    WHERE o.user_id = u.id AND o.total > 1000
);
```

8. Optimize Pagination
```sql
-- Bad: OFFSET becomes very slow for large offsets
SELECT * FROM products
ORDER BY id
LIMIT 1000 OFFSET 100000;  -- Reads and discards 100,000 rows!

-- Good: Cursor-based pagination (seek method)
SELECT * FROM products
WHERE id > 100000
ORDER BY id
LIMIT 1000;

-- For complex sorting, use bookmark
SELECT * FROM products
WHERE (created_at, id) > ('2024-01-15 10:00:00', 12345)
ORDER BY created_at, id
LIMIT 1000;
```

9. Use Proper Data Types
```sql
-- Bad: Storing numbers as strings
CREATE TABLE bad (
    id VARCHAR(20),  -- Should be INT
    price VARCHAR(10)  -- Should be DECIMAL
);

-- Good: Use appropriate types
CREATE TABLE good (
    id INT UNSIGNED,
    price DECIMAL(10,2)
);
```

Why it matters:
- Smaller storage = faster queries
- Proper sorting/comparison
- Index efficiency
- Type safety

10. Avoid SELECT DISTINCT When Possible

```sql
-- Bad: DISTINCT on large result set (slow)
SELECT DISTINCT user_id FROM orders;

-- Better: Use GROUP BY with index
SELECT user_id FROM orders GROUP BY user_id;

-- Or if you need other columns:
SELECT user_id, MIN(created_at) as first_order
FROM orders
GROUP BY user_id;
```

11. Use UNION ALL Instead of UNION
```sql
-- UNION: Removes duplicates (slower)
SELECT id FROM table1
UNION
SELECT id FROM table2;

-- UNION ALL: Keeps duplicates (faster)
SELECT id FROM table1
UNION ALL
SELECT id FROM table2;
```

Use UNION ALL when you know there are no duplicates or don't need to remove them.

12. Optimize JOIN Order
```sql
-- MySQL optimizer usually handles this, but you can help:

-- Start with smallest/most filtered table
SELECT o.*, p.*
FROM orders o  -- Smaller table after WHERE
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
WHERE o.created_at > '2024-01-01'  -- Filters orders significantly
```

13. Use Index Hints (Rarely Needed)
```sql
-- Force use of specific index
SELECT * FROM users
FORCE INDEX (idx_email)
WHERE email = 'john@example.com';
```

```sql
-- Suggest index
SELECT * FROM users
USE INDEX (idx_city)
WHERE city = 'New York';

-- Ignore index
SELECT * FROM users
IGNORE INDEX (idx_name)
WHERE last_name = 'Smith';
```

⚠️ Warning: Index hints override the optimizer. Only use when you know better than the optimizer (rare).

14. Avoid Implicit Type Conversion
```sql
-- Bad: id is INT, but comparing to string (can't use index)
SELECT * FROM users WHERE id = '123';

-- Good: Use correct type
SELECT * FROM users WHERE id = 123;
```

15. Use Batch Operations
```sql
-- Bad: Individual inserts
INSERT INTO logs VALUES (1, 'msg1');
INSERT INTO logs VALUES (2, 'msg2');
-- ... 1000 times

-- Good: Batch insert (100x faster)
INSERT INTO logs VALUES
  (1, 'msg1'),
  (2, 'msg2'),
  -- ... 1000 rows
  (1000, 'msg1000');

-- Bad: Individual updates
UPDATE users SET status = 'active' WHERE id = 1;
UPDATE users SET status = 'active' WHERE id = 2;

-- Good: Single update
UPDATE users SET status = 'active' WHERE id IN (1, 2, 3, ...);
```

5.4 EXPLAIN Plans

EXPLAIN shows how MySQL executes a query. Essential for optimization.

Basic EXPLAIN

```sql
EXPLAIN SELECT * FROM users WHERE email = 'john@example.com';
```

EXPLAIN Output Columns

Column	Meaning
id	SELECT identifier
select_type	Type of SELECT (SIMPLE, SUBQUERY, etc.)
table	Table being accessed
partitions	Partitions matched
type	**Join type (most important!)**
possible_keys	Indexes that could be used
key	**Index actually used**
key_len	Length of key used
ref	Columns compared to index
rows	**Estimated rows to examine**
filtered	% of rows filtered by condition
Extra	**Additional information**

Join Types (Best to Worst)

1. **system**: Table has only one row
2. **const**: At most one matching row (PRIMARY KEY or UNIQUE lookup)
3. **eq_ref**: One row per combination (JOINs on PRIMARY KEY)
4. **ref**: Multiple rows with matching index value
5. **fulltext**: FULLTEXT index used
6. **ref_or_null**: Like ref, but searches for NULL too
7. **index_merge**: Multiple indexes used
8. **unique_subquery**: Subquery with unique index
9. **index_subquery**: Subquery with non-unique index
10. **range**: Index range scan (BETWEEN, >, <, IN)
11. **index**: Full index scan (better than ALL)
12. **ALL**: **Full table scan (AVOID!)**

```sql
-- type: const (best)
EXPLAIN SELECT * FROM users WHERE id = 1;

-- type: ref (good)
EXPLAIN SELECT * FROM users WHERE city = 'New York';

-- type: range (decent)
EXPLAIN SELECT * FROM orders WHERE created_at > '2024-01-01';
```

```sql
-- type: ALL (bad - full table scan!)
EXPLAIN SELECT * FROM users WHERE YEAR(created_at) = 2024;
```

Important Extra Values

Extra	Meaning	Good/Bad
Using index	Covering index (no table lookup)	✅ Excellent
Using where	WHERE clause filtering	✅ Normal
Using index condition	Index Condition Pushdown	✅ Good
Using temporary	Temporary table created	⚠️ Slow
Using filesort	External sort needed	⚠️ Slow
Using join buffer	Join buffer used (no index)	⚠️ Bad
Impossible WHERE	WHERE always false	✅ Optimized away
Select tables optimized away	Aggregates computed at optimization	✅ Excellent

```sql
-- Using index (best)
EXPLAIN SELECT id, email FROM users WHERE email = 'test@test.com';

-- Using filesort (add index on ORDER BY column)
EXPLAIN SELECT * FROM products ORDER BY price;

-- Using temporary (avoid GROUP BY on non-indexed columns)
EXPLAIN SELECT city, COUNT(*) FROM users GROUP BY city;
```

EXPLAIN Formats

```sql
-- Traditional format
EXPLAIN SELECT * FROM users WHERE id = 1;

-- Extended information (MySQL 5.7+)
EXPLAIN EXTENDED SELECT * FROM users WHERE id = 1;
SHOW WARNINGS;  -- Shows optimized query

-- JSON format (detailed)
EXPLAIN FORMAT=JSON
SELECT * FROM users u
INNER JOIN orders o ON u.id = o.user_id;

-- Tree format (MySQL 8.0.16+)
EXPLAIN FORMAT=TREE
SELECT * FROM users u
INNER JOIN orders o ON u.id = o.user_id;
```

EXPLAIN ANALYZE (MySQL 8.0.18+)

Shows actual execution time and row counts (not just estimates).

```sql
EXPLAIN ANALYZE
SELECT u.username, COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id;

-- Output includes:
-- - Actual time taken
-- - Actual rows processed
-- - Number of loops
```

Reading EXPLAIN

```sql
EXPLAIN
SELECT o.order_number, u.username, p.product_name
FROM orders o
INNER JOIN users u ON o.user_id = u.id
INNER JOIN order_items oi ON o.id = oi.order_id
INNER JOIN products p ON oi.product_id = p.id
WHERE o.created_at > '2024-01-01';
```

What to look for:

1. **type: ALL** = Full table scan (add index!)
2. **key: NULL** = No index used (add index!)
3. **rows: Large number** = Too many rows examined (filter earlier, add index)
4. **Extra: Using filesort** = Slow sort (add index on ORDER BY columns)
5. **Extra: Using temporary** = Temp table created (optimize GROUP BY)
6. **Extra: Using index** = Great! Covering index

Optimization Workflow

1. **Run EXPLAIN on slow query**
2. **Check type column** - Should not be ALL
3. **Check key column** - Should use an index
4. **Check rows** - Should be reasonable number
5. **Check Extra** - Look for warnings (filesort, temporary)
6. **Add/modify indexes** as needed
7. **Run EXPLAIN again** - Verify improvement
8. **Test with real data** - EXPLAIN uses estimates

```sql
```

```
-- Before optimization
EXPLAIN SELECT * FROM orders WHERE YEAR(created_at) = 2024;
-- type: ALL, rows: 1000000, Extra: Using where

-- Add index
CREATE INDEX idx_created ON orders(created_at);

-- Rewrite query
EXPLAIN SELECT * FROM orders
WHERE created_at >= '2024-01-01' AND created_at < '2025-01-01';
-- type: range, key: idx_created, rows: 50000
```

Chapter 6: Transactions & ACID

6.1 Transaction Basics

A transaction is a sequence of SQL operations treated as a single unit of work. All operations succeed together or fail together.

ACID Properties

Atomicity: All or nothing
```sql
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;  -- Both updates succeed or both fail
```

Consistency: Database moves from one valid state to another
```sql
-- Total balance before = Total balance after
-- Constraints are enforced
-- Foreign keys maintained
```

Isolation: Concurrent transactions don't interfere
```sql
-- Transaction 1 doesn't see uncommitted changes from Transaction 2
```

Durability: Committed changes are permanent
```sql
COMMIT;  -- Changes survive server crash, power failure
```

```

#### Basic Transaction Syntax
```sql
-- Start transaction
START TRANSACTION;
-- or
BEGIN;

-- Your SQL operations
INSERT INTO orders (user_id, total) VALUES (1, 100);
INSERT INTO order_items (order_id, product_id, quantity)
VALUES (LAST_INSERT_ID(), 101, 2);
UPDATE products SET stock = stock - 2 WHERE id = 101;

-- Commit changes (make permanent)
COMMIT;

-- Or rollback if error
ROLLBACK;
```

Autocommit Mode
```sql
-- Check autocommit status
SELECT @@autocommit;

-- Disable autocommit
SET autocommit = 0;

-- Now each statement starts implicit transaction
INSERT INTO users VALUES (...);  -- Transaction started
UPDATE users SET ...;            -- Same transaction
COMMIT;                          -- Commit explicitly

-- Enable autocommit (default)
SET autocommit = 1;
-- Each statement auto-commits
```

Transaction Example: Money Transfer
```sql
START TRANSACTION;

-- Check source account balance
SELECT @balance := balance FROM accounts WHERE id = 1 FOR UPDATE;
```
```

```
-- Validate
```