SQL Learning Index & Navigation Guide

> **Complete SQL Mastery Path**: From beginner to advanced database professional

Learning Path Overview

This comprehensive SQL course covers **MySQL** and **PostgreSQL** with real-world examples, performance optimization, and interview preparation. Follow the chapters in order for structured learning, or jump to specific topics as needed.

Progress Tracking

- [ ] **Basics Complete** (Chapters 1-5)
- [ ] **Intermediate Complete** (Chapters 6-9)
- [ ] **Advanced Complete** (Chapters 10-17)
- [ ] **Enterprise & Modern SQL Complete** (Chapters 18-23)
- [ ] **Ready for Production**

BASICS (Chapters 1-5)

Foundation concepts every SQL developer needs

Chapter 1: Introduction to SQL & RDBMS

Learn: Database fundamentals, MySQL vs PostgreSQL, ACID properties
Time: 45 minutes
Hands-on: Setting up databases, basic commands

Chapter 2: Data Types

Learn: Numeric, string, date types across databases
Time: 60 minutes
Hands-on: Choosing optimal data types, storage efficiency

Chapter 3: Creating Databases & Tables

Learn: DDL statements, table design, naming conventions
Time: 75 minutes
Hands-on: Building e-commerce database schema

Chapter 4: CRUD Operations

Learn: SELECT, INSERT, UPDATE, DELETE mastery
Time: 90 minutes
Hands-on: Data manipulation, bulk operations

Chapter 5: Constraints

🎯 **Learn:** PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK constraints
⏱️ **Time:** 60 minutes
🔧 **Hands-on:** Data integrity, relationship enforcement

📓 INTERMEDIATE (Chapters 6-9)

Building complex queries and relationships

Chapter 6: WHERE, GROUP BY, HAVING, ORDER BY

🎯 **Learn:** Filtering, grouping, sorting data
⏱️ **Time:** 75 minutes
🔧 **Hands-on:** Complex filtering conditions, data aggregation

Chapter 7: Joins

🎯 **Learn:** INNER, LEFT, RIGHT, FULL OUTER joins
⏱️ **Time:** 90 minutes
🔧 **Hands-on:** Multi-table queries, relationship navigation

Chapter 8: Aggregate Functions

🎯 **Learn:** COUNT, SUM, AVG, MIN, MAX, statistical functions
⏱️ **Time:** 60 minutes
🔧 **Hands-on:** Business analytics, reporting queries

Chapter 9: Subqueries & Nested Queries

🎯 **Learn:** Correlated subqueries, EXISTS, IN, scalar subqueries
⏱️ **Time:** 90 minutes
🔧 **Hands-on:** Complex data retrieval, conditional logic

📓 ADVANCED (Chapters 10-17)

Professional-level database skills

Chapter 10: Window Functions

🎯 **Learn:** ROW_NUMBER, RANK, LAG, LEAD, analytical functions
⏱️ **Time:** 120 minutes
🔧 **Hands-on:** Advanced analytics, running totals, comparisons

Chapter 11: Indexes & Query Optimization

🎯 **Learn:** Index types, EXPLAIN plans, performance tuning
⏱️ **Time:** 90 minutes
🔧 **Hands-on:** Query optimization, performance analysis

Chapter 12: Stored Procedures & Functions

🎯 **Learn:** Reusable code, parameters, control structures
⏱️ **Time:** 75 minutes
🔧 **Hands-on:** Business logic implementation, code organization

Chapter 13: Triggers & Events

🎯 **Learn:** Automated responses, data validation, scheduling
⏱️ **Time:** 75 minutes
🔧 **Hands-on:** Audit trails, data synchronization

Chapter 14: Views & CTEs

🎯 **Learn:** Virtual tables, Common Table Expressions, recursive queries
⏱️ **Time:** 90 minutes
🔧 **Hands-on:** Data abstraction, complex hierarchical queries

Chapter 15: Transactions & Concurrency

🎯 **Learn:** ACID properties, isolation levels, deadlock handling
⏱️ **Time:** 105 minutes
🔧 **Hands-on:** Multi-user scenarios, data consistency

Chapter 16: Database Security & User Management

🎯 **Learn:** Authentication, authorization, encryption, auditing
⏱️ **Time:** 90 minutes
🔧 **Hands-on:** Security implementation, compliance

Chapter 17: Backup and Recovery

🎯 **Learn:** Backup strategies, disaster recovery, high availability
⏱️ **Time:** 120 minutes
🔧 **Hands-on:** Recovery procedures, replication setup

🏢 ENTERPRISE & MODERN SQL (Chapters 18-23)

Enterprise-level database architecture and modern practices

Chapter 18: Database Replication & High Availability

🎯 **Learn:** Master-slave replication, clustering, failover strategies
⏱️ **Time:** 135 minutes
🔧 **Hands-on:** Setting up replication, testing failover scenarios

Chapter 19: Database Proxies & Connection Pooling

🎯 **Learn:** Connection management, load balancing solutions
⏱️ **Time:** 90 minutes
🔧 **Hands-on:** Configuring database proxies, implementing connection pooling

Chapter 20: Partitioning & Sharding

🎯 **Learn:** Scale databases through partitioning and sharding strategies
⏱️ **Time:** 120 minutes
🔧 **Hands-on:** Implementing partitioning schemes, designing shard architectures

Chapter 21: ETL & Data Warehousing

🎯 **Learn:** ETL pipelines, data warehousing concepts
⏱️ **Time:** 150 minutes
🔧 **Hands-on:** Creating ETL processes, dimensional modeling, data quality checks

Chapter 22: Cloud Databases & DBaaS

🎯 **Learn:** Deploy and manage databases in cloud environments
⏱️ **Time:** 105 minutes
🔧 **Hands-on:** Setting up cloud databases, migration strategies, cost optimization

Chapter 23: Database DevOps & CI/CD

🎯 **Learn:** Database DevOps practices and automated deployments
⏱️ **Time:** 120 minutes
🔧 **Hands-on:** Database version control, CI/CD pipelines, infrastructure as code

---

🎯 Learning Strategies

📚 **Sequential Learning** (Recommended for beginners)

1. Start with Chapter 1 and progress linearly
2. Complete all hands-on exercises
3. Review interview questions at the end of each chapter
4. Build the practice projects as you learn

🎯 **Topic-Focused Learning** (For experienced developers)

- **Performance Focus:** Chapters 11, 15, 10, 17
- **Analytics Focus:** Chapters 8, 9, 10, 14
- **Security Focus:** Chapters 16, 15, 5, 17
- **Architecture Focus:** Chapters 1, 3, 11, 14, 17

🔄 **Review & Practice**

- **Week 1-2:** Basics (Chapters 1-5)
- **Week 3-4:** Intermediate (Chapters 6-9)
- **Week 5-7:** Advanced (Chapters 10-17)
- **Week 8:** Review, practice projects, interview prep

🛠️ Practical Projects

🏢 **E-commerce Database** (Chapters 1-9)

Build a complete online store database with:

- Customer management
- Product catalog
- Order processing
- Inventory tracking

📊 **Analytics Dashboard** (Chapters 10-14)

Create business intelligence queries for:

- Sales reporting
- Customer segmentation
- Performance metrics
- Trend analysis

🔒 **Enterprise Security** (Chapters 15-17)

Implement production-ready:

- User access control
- Data encryption
- Audit logging
- Backup strategies
- Disaster recovery

🏗️ **Scalable Multi-Tenant SaaS Platform** (Chapters 18-20)

Build enterprise-grade infrastructure with:

- Database replication and high availability
- Connection pooling and load balancing
- Horizontal partitioning and sharding
- Multi-tenant data isolation

🔄 **Real-time Data Pipeline & Warehouse** (Chapter 21)

Create comprehensive data processing system:

- ETL pipeline development
- Data transformation and validation
- Dimensional modeling
- Data quality monitoring

☁️ **Cloud-Native Database Infrastructure** (Chapters 22-23)

Implement modern cloud database operations:

- Cloud database deployment and migration

- Database DevOps and CI/CD pipelines
- Infrastructure as code
- Automated monitoring and scaling

Interview Preparation

Common SQL Interview Topics

- **Joins & Subqueries** (Chapters 7, 9)
- **Window Functions** (Chapter 10)
- **Performance Optimization** (Chapter 11)
- **Database Design** (Chapters 3, 5)
- **Transactions** (Chapter 15)
- **Backup & Recovery** (Chapter 17)
- **Database Replication & High Availability** (Chapter 18)
- **Partitioning & Sharding** (Chapter 20)
- **ETL & Data Warehousing** (Chapter 21)
- **Cloud Databases** (Chapter 22)
- **Database DevOps** (Chapter 23)

Practice Questions by Level

- **Junior:** Basic CRUD, simple joins, aggregate functions
- **Mid-level:** Complex joins, subqueries, window functions
- **Senior:** Performance tuning, architecture, security, disaster recovery
- **Database Architect:** Replication strategies, sharding design, ETL pipelines
- **Enterprise Engineer:** Cloud migration, DevOps automation, high availability

---

Quick Reference Links

Syntax References

- MySQL Documentation
- PostgreSQL Documentation

Tools & Setup

- MySQL Workbench
- pgAdmin (PostgreSQL)
- DBeaver (Universal)
- VS Code with SQL extensions

Additional Resources

- SQL practice platforms
- Database design tools
- Performance monitoring tools

✅ Completion Checklist

📓 Basics Mastery

- ☐ Can create and modify database schemas
- ☐ Comfortable with all CRUD operations
- ☐ Understands data types and constraints
- ☐ Can write basic queries with filtering and sorting

📓 Intermediate Mastery

- ☐ Masters all types of joins
- ☐ Can write complex subqueries
- ☐ Comfortable with aggregate functions
- ☐ Understands query execution flow

📓 Advanced Mastery

- ☐ Uses window functions for analytics
- ☐ Can optimize query performance
- ☐ Implements stored procedures and triggers
- ☐ Understands transactions and concurrency
- ☐ Can design secure database systems
- ☐ Masters backup and recovery procedures

🏢 Enterprise Mastery

- ☐ Sets up database replication and high availability
- ☐ Configures database proxies and connection pooling
- ☐ Implements partitioning and sharding strategies
- ☐ Builds ETL pipelines and data warehouses
- ☐ Deploys and manages cloud databases
- ☐ Implements database DevOps and CI/CD practices

🚀 Production Ready

- ☐ Can design scalable database schemas
- ☐ Implements proper security measures
- ☐ Monitors and optimizes performance
- ☐ Handles backup and recovery
- ☐ Plans for disaster recovery
- ☐ Ready for senior database roles

> 🎯 **Goal**: Master SQL to work cross-database, troubleshoot queries, optimize performance, design clean relational schemas, and ensure data protection. If you understand how transactions behave in PostgreSQL vs MySQL, when to use window functions, how to model efficient schemas, and how to recover from disasters — you've nailed it! 🚀

📓 Chapter 1: Introduction to SQL & RDBMS

🎯 What You'll Learn

- What SQL is and why it matters
- Understanding Relational Database Management Systems (RDBMS)
- Key differences between MySQL and PostgreSQL
- Setting up your first database

🕮 Concept Explanation

SQL (Structured Query Language) is a standardized language for managing and manipulating relational databases. Think of it as the "universal language" that lets you communicate with databases to store, retrieve, update, and delete data.

RDBMS (Relational Database Management System) is software that manages relational databases. Popular examples include:

- **MySQL**: Fast, reliable, widely used in web applications
- **PostgreSQL**: Feature-rich, standards-compliant, excellent for complex queries
- **SQLite**: Lightweight, file-based, perfect for small applications
- **Microsoft SQL Server**: Enterprise-grade, Windows-focused
- **MariaDB**: MySQL fork with additional features

Why Relational Databases?

- **Structure**: Data is organized in tables with rows and columns
- **Relationships**: Tables can be linked together (hence "relational")
- **ACID Properties**: Atomicity, Consistency, Isolation, Durability
- **Scalability**: Can handle millions of records efficiently

---

🔧 Syntax Comparison

Creating a Database

MySQL:

```
CREATE DATABASE company_db;
USE company_db;
```

PostgreSQL:

```
CREATE DATABASE company_db;
\c company_db  -- Connect to database in psql
```

Basic Database Operations

Show Databases:

```
-- MySQL
SHOW DATABASES;

-- PostgreSQL
\l  -- In psql command line
-- OR
SELECT datname FROM pg_database;
```

Show Tables:

```
-- MySQL
SHOW TABLES;

-- PostgreSQL
\dt  -- In psql
-- OR
SELECT table_name FROM information_schema.tables
WHERE table_schema = 'public';
```

Real-World Examples

Example 1: E-commerce Database Structure

```
-- Creating a simple e-commerce database
CREATE DATABASE ecommerce_db;

-- MySQL/PostgreSQL (similar syntax)
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,  -- MySQL
    -- customer_id SERIAL PRIMARY KEY,          -- PostgreSQL alternative
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE products (
    product_id INT PRIMARY KEY AUTO_INCREMENT,   -- MySQL
    -- product_id SERIAL PRIMARY KEY,           -- PostgreSQL alternative
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    stock_quantity INT DEFAULT 0,
```

```
    category VARCHAR(50)
);
```

Example 2: Blog Database

```
CREATE TABLE users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE posts (
    post_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(200) NOT NULL,
    content TEXT,
    author_id INT,
    published_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (author_id) REFERENCES users(user_id)
);
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between SQL and NoSQL?"**
    - SQL: Structured, ACID compliant, good for complex relationships
    - NoSQL: Flexible schema, horizontally scalable, good for big data
2. **"Why choose PostgreSQL over MySQL?"**
    - PostgreSQL: Better standards compliance, advanced features (JSON, arrays, window functions)
    - MySQL: Faster for simple queries, easier to set up, more hosting options
3. **"What are ACID properties?"**
    - **Atomicity**: All or nothing transactions
    - **Consistency**: Data integrity maintained
    - **Isolation**: Concurrent transactions don't interfere
    - **Durability**: Committed data survives system failures

Real-World Use Cases:

- **E-commerce**: Product catalogs, order management, inventory tracking
- **Social Media**: User profiles, posts, comments, relationships
- **Banking**: Account management, transactions, audit trails

- **Healthcare**: Patient records, appointments, medical history

---

⚠ Things to Watch Out For

1. **Case Sensitivity Differences**

```
-- MySQL: Generally case-insensitive (depends on OS)
SELECT * FROM Users;  -- Works
SELECT * FROM users;  -- Also works

-- PostgreSQL: Case-sensitive
SELECT * FROM Users;  -- Error if table is 'users'
SELECT * FROM users;  -- Correct
```

2. **Auto-Increment Syntax**

```
-- MySQL
id INT PRIMARY KEY AUTO_INCREMENT

-- PostgreSQL
id SERIAL PRIMARY KEY
-- OR
id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY
```

3. **String Concatenation**

```
-- MySQL
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM users;

-- PostgreSQL
SELECT first_name || ' ' || last_name AS full_name FROM users;
-- OR
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM users;
```

4. **Limit Syntax**

```
-- MySQL
SELECT * FROM products LIMIT 10;

-- PostgreSQL
SELECT * FROM products LIMIT 10;
-- Both support LIMIT, but PostgreSQL also supports:
SELECT * FROM products LIMIT 10 OFFSET 20;
```

5. **Date/Time Functions**

```
-- MySQL
SELECT NOW(), CURDATE(), CURTIME();

-- PostgreSQL
SELECT NOW(), CURRENT_DATE, CURRENT_TIME;
```

🚀 Next Steps

In the next chapter, we'll dive into **Data Types** and explore how MySQL and PostgreSQL handle different kinds of data. You'll learn when to use VARCHAR vs TEXT, INT vs BIGINT, and how to work with dates, JSON, and more.

📝 Quick Practice

Try creating a simple database for a library system with tables for:

- `books` (id, title, author, isbn, published_year)
- `members` (id, name, email, join_date)
- `loans` (id, book_id, member_id, loan_date, return_date)

Think about what data types and constraints you'd use for each field!

📓 Chapter 2: Data Types (MySQL vs PostgreSQL)

🎯 What You'll Learn

- Core data types in MySQL and PostgreSQL
- When to use each data type
- Cross-database compatibility considerations
- Performance implications of data type choices

📖 Concept Explanation

Data types define what kind of data can be stored in each column of a table. Choosing the right data type is crucial for:

- **Storage efficiency**: Using the smallest appropriate type saves space
- **Performance**: Proper types enable faster queries and indexing
- **Data integrity**: Types prevent invalid data from being stored
- **Application compatibility**: Ensures your app can handle the data correctly

Key Principles:

1. **Use the most restrictive type that fits your data**
2. **Consider future growth** (but don't over-engineer)
3. **Think about queries** you'll run on the data
4. **Plan for data validation** at the database level

🔧 Data Types Comparison

📊 Numeric Types

Purpose	MySQL	PostgreSQL	Notes
Small integers	`TINYINT` (-128 to 127)	`SMALLINT` (-32,768 to 32,767)	PostgreSQL doesn't have TINYINT
Regular integers	`INT` (-2B to 2B)	`INTEGER` (same range)	Most common for IDs
Large integers	`BIGINT`	`BIGINT`	For very large numbers
Auto-increment	`AUTO_INCREMENT`	`SERIAL` or `IDENTITY`	Different syntax, same concept
Decimals	`DECIMAL(10,2)`	`NUMERIC(10,2)`	Exact precision
Floating point	`FLOAT`, `DOUBLE`	`REAL`, `DOUBLE PRECISION`	Approximate values

Examples:

```
-- MySQL
CREATE TABLE products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    price DECIMAL(10,2),  -- $99,999,999.99 max
    weight FLOAT,         -- Approximate weight
    stock_count SMALLINT UNSIGNED  -- 0 to 65,535
);

-- PostgreSQL
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    price NUMERIC(10,2),  -- Same precision as MySQL DECIMAL
    weight REAL,          -- Approximate weight
    stock_count SMALLINT CHECK (stock_count >= 0)
);
```

📝 String Types

Purpose	MySQL	PostgreSQL	Max Length
Fixed length	`CHAR(n)`	`CHAR(n)`	n characters

Purpose	MySQL	PostgreSQL	Max Length
Variable length	`VARCHAR(n)`	`VARCHAR(n)`	n characters
Large text	`TEXT`	`TEXT`	~65KB (MySQL), ~1GB (PostgreSQL)
Huge text	`LONGTEXT`	`TEXT`	~4GB (MySQL), unlimited (PostgreSQL)
Binary data	`BLOB`	`BYTEA`	For files, images

Examples:

```
-- User profile table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL,      -- Short, indexed field
    email VARCHAR(255) NOT NULL,        -- Email addresses
    bio TEXT,                           -- Long description
    avatar_data BYTEA,                  -- Profile picture (PostgreSQL)
    -- avatar_data BLOB,                -- Profile picture (MySQL)
    country_code CHAR(2)                -- Fixed: 'US', 'CA', 'UK'
);
```

Date and Time Types

Purpose	MySQL	PostgreSQL	Notes
Date only	`DATE`	`DATE`	YYYY-MM-DD
Time only	`TIME`	`TIME`	HH:MM:SS
Date + Time	`DATETIME`	`TIMESTAMP`	Different default behaviors
With timezone	`TIMESTAMP`	`TIMESTAMPTZ`	PostgreSQL handles timezones better
Year only	`YEAR`	Use `INTEGER`	MySQL-specific

Examples:

```
-- Event scheduling table
CREATE TABLE events (
    id SERIAL PRIMARY KEY,
    event_name VARCHAR(100),
    event_date DATE,                    -- Just the date
    start_time TIME,                    -- Just the time
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- PostgreSQL also supports:
    scheduled_at TIMESTAMPTZ,           -- With timezone info
    duration INTERVAL                   -- PostgreSQL: '2 hours 30 minutes'
);
```

🔢 Boolean and Special Types

Boolean:

```
-- MySQL (no native boolean)
CREATE TABLE settings (
    id INT PRIMARY KEY,
    is_active TINYINT(1),  -- 0 or 1
    -- OR
    is_public BOOLEAN      -- Actually TINYINT(1)
);

-- PostgreSQL (true boolean)
CREATE TABLE settings (
    id SERIAL PRIMARY KEY,
    is_active BOOLEAN,     -- TRUE/FALSE/NULL
    is_public BOOL         -- Alias for BOOLEAN
);
```

JSON (Modern databases):

```
-- MySQL 5.7+
CREATE TABLE products (
    id INT PRIMARY KEY,
    attributes JSON,
    metadata JSON
);

-- PostgreSQL (superior JSON support)
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    attributes JSON,       -- Basic JSON
    metadata JSONB,       -- Binary JSON (faster, indexable)
    tags TEXT[]           -- Array of strings (PostgreSQL only)
);
```

💡 Real-World Examples

Example 1: E-commerce Product Catalog

```
-- Optimized for performance and storage
CREATE TABLE products (
    -- Primary key: INT is sufficient for most catalogs
    product_id INT AUTO_INCREMENT PRIMARY KEY,

    -- Product info: VARCHAR for known limits
    sku VARCHAR(50) UNIQUE NOT NULL,
```

```
    name VARCHAR(200) NOT NULL,

    -- Price: DECIMAL for exact money calculations
    price DECIMAL(10,2) NOT NULL,
    sale_price DECIMAL(10,2),

    -- Inventory: SMALLINT saves space
    stock_quantity SMALLINT UNSIGNED DEFAULT 0,

    -- Descriptions: TEXT for variable length
    short_description VARCHAR(500),
    full_description TEXT,

    -- Flags: BOOLEAN/TINYINT for true/false
    is_active BOOLEAN DEFAULT TRUE,
    is_featured BOOLEAN DEFAULT FALSE,

    -- Tracking: TIMESTAMP for audit trail
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

    -- Categories: INT for foreign key
    category_id INT,

    -- Metadata: JSON for flexible attributes
    attributes JSON,  -- {"color": "red", "size": "large"}

    FOREIGN KEY (category_id) REFERENCES categories(category_id)
);
```

Example 2: User Management System

```
CREATE TABLE users (
    user_id BIGINT AUTO_INCREMENT PRIMARY KEY,  -- BIGINT for large user base

    -- Authentication
    email VARCHAR(255) UNIQUE NOT NULL,
    username VARCHAR(50) UNIQUE NOT NULL,
    password_hash CHAR(60),  -- bcrypt hash is always 60 chars

    -- Profile
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE,

    -- Settings
    email_verified BOOLEAN DEFAULT FALSE,
    is_admin BOOLEAN DEFAULT FALSE,
    timezone VARCHAR(50) DEFAULT 'UTC',

    -- Tracking
```

```
    last_login TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Flexible data
    preferences JSON,  -- User settings, theme, etc.

    -- Constraints
    CHECK (email LIKE '%@%'),  -- Basic email validation
    CHECK (date_of_birth < CURRENT_DATE)
);
```

Example 3: Financial Transactions

```
CREATE TABLE transactions (
    transaction_id BIGINT AUTO_INCREMENT PRIMARY KEY,

    -- Money: Always use DECIMAL for currency
    amount DECIMAL(15,2) NOT NULL,  -- Up to $999,999,999,999.99
    currency CHAR(3) DEFAULT 'USD', -- ISO currency codes

    -- References
    from_account_id BIGINT NOT NULL,
    to_account_id BIGINT,

    -- Transaction details
    transaction_type ENUM('deposit', 'withdrawal', 'transfer', 'fee'),
    status ENUM('pending', 'completed', 'failed', 'cancelled') DEFAULT 'pending',

    -- Audit trail
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    processed_at TIMESTAMP,

    -- Additional info
    description VARCHAR(255),
    reference_number VARCHAR(50) UNIQUE,

    -- Constraints
    CHECK (amount > 0),
    CHECK (from_account_id != to_account_id)
);
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"When would you use VARCHAR vs TEXT?"**
    - VARCHAR: When you know the approximate max length (names, emails)
    - TEXT: For variable-length content (blog posts, comments)

- Performance: VARCHAR can be indexed more efficiently

2. **"Why use DECIMAL instead of FLOAT for money?"**
   - DECIMAL: Exact precision, no rounding errors
   - FLOAT: Approximate, can cause rounding issues with currency
   - Example: 0.1 + 0.2 might not equal 0.3 in floating point

3. **"What's the difference between DATETIME and TIMESTAMP?"**
   - DATETIME: Fixed time, no timezone conversion
   - TIMESTAMP: Converts to UTC, affected by timezone changes
   - TIMESTAMP has smaller range but better for global apps

Performance Tips:

1. **Choose the smallest appropriate type**

```
-- Bad: Wastes 3 bytes per row
age INT;

-- Good: TINYINT is sufficient (0-255)
age TINYINT UNSIGNED;
```

2. **Use fixed-length types when possible**

```
-- Good for indexing and joins
country_code CHAR(2);
phone_country_code CHAR(3);
```

3. **Consider indexing implications**

```
-- Shorter keys = faster indexes
email VARCHAR(255),  -- Good
bio TEXT,           -- Don't index the full column
```

⚠ Things to Watch Out For

1. **MySQL vs PostgreSQL Auto-Increment**

```
-- MySQL
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY
);
```

```
-- PostgreSQL
CREATE TABLE users (
    id SERIAL PRIMARY KEY
    -- OR (PostgreSQL 10+)
    -- id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY
);
```

2. **Boolean Handling**

```
-- MySQL: No true boolean
SELECT * FROM users WHERE is_active = 1;  -- Use 1/0

-- PostgreSQL: True boolean
SELECT * FROM users WHERE is_active = TRUE;  -- Use TRUE/FALSE
SELECT * FROM users WHERE is_active;         -- Implicit TRUE check
```

3. **String Length Limits**

```
-- MySQL: VARCHAR max is 65,535 characters
CREATE TABLE posts (content VARCHAR(65535));  -- Max in MySQL

-- PostgreSQL: VARCHAR can be much larger
CREATE TABLE posts (content VARCHAR(1000000)); -- Works in PostgreSQL
```

4. **JSON Support Differences**

```
-- MySQL: Basic JSON
SELECT JSON_EXTRACT(data, '$.name') FROM products;

-- PostgreSQL: Rich JSON operators
SELECT data->>'name' FROM products;           -- Get as text
SELECT data->'attributes'->'color' FROM products; -- Nested access
```

5. **Date/Time Timezone Handling**

```
-- MySQL: TIMESTAMP affected by timezone setting
SET time_zone = '+00:00';
INSERT INTO events (created_at) VALUES (NOW());

-- PostgreSQL: TIMESTAMPTZ stores timezone info
INSERT INTO events (created_at) VALUES (NOW()); -- Includes timezone
```

6. **Enum vs Check Constraints**

```
-- MySQL: ENUM type
CREATE TABLE orders (
    status ENUM('pending', 'shipped', 'delivered')
);

-- PostgreSQL: Can use ENUM or CHECK
CREATE TYPE order_status AS ENUM ('pending', 'shipped', 'delivered');
CREATE TABLE orders (
    status order_status
);
-- OR
CREATE TABLE orders (
    status VARCHAR(20) CHECK (status IN ('pending', 'shipped', 'delivered'))
);
```

🚀 Next Steps

In the next chapter, we'll learn how to **Create Databases & Tables** with proper constraints, indexes, and relationships. You'll see how to put these data types to work in real table designs.

📝 Quick Practice

Design data types for a social media platform:

- User profiles (username, bio, follower count)
- Posts (content, likes, shares, timestamp)
- Comments (text, reply threading)
- Media uploads (file data, metadata)

Think about:

- What's the maximum reasonable length for each field?
- Which fields need exact vs approximate numbers?
- How will you handle user-generated content of varying lengths?

📓 Chapter 3: Creating Databases & Tables

🎯 What You'll Learn

- How to create and manage databases
- Table creation with proper structure
- Primary keys, foreign keys, and constraints
- Indexes for performance
- Best practices for schema design

📖 Concept Explanation

Database creation is the foundation of any data-driven application. A well-designed database schema is like a blueprint for a house - it determines how efficiently your application will run and how easily you can maintain it.

Key Concepts:

- **Database**: A container that holds related tables
- **Schema**: The structure and organization of your database
- **Table**: A collection of related data organized in rows and columns
- **Constraints**: Rules that ensure data integrity
- **Indexes**: Data structures that speed up queries

Design Principles:

1. **Normalization**: Reduce data redundancy
2. **Consistency**: Use consistent naming conventions
3. **Scalability**: Design for future growth
4. **Performance**: Consider query patterns
5. **Integrity**: Enforce data quality through constraints

🔧 Syntax Comparison

Creating Databases

MySQL:

```
-- Create database
CREATE DATABASE ecommerce_db
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;

-- Use database
USE ecommerce_db;

-- Show current database
SELECT DATABASE();

-- Drop database (careful!)
DROP DATABASE IF EXISTS ecommerce_db;
```

PostgreSQL:

```
-- Create database
CREATE DATABASE ecommerce_db
```

```
WITH ENCODING 'UTF8'
LC_COLLATE = 'en_US.UTF-8'
LC_CTYPE = 'en_US.UTF-8';

-- Connect to database (in psql)
\c ecommerce_db

-- Show current database
SELECT current_database();

-- Drop database (careful!)
DROP DATABASE IF EXISTS ecommerce_db;
```

Basic Table Creation

Simple Table:

```
-- Works in both MySQL and PostgreSQL
CREATE TABLE categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(100) NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Auto-Increment Differences:

```
-- MySQL
CREATE TABLE products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(200) NOT NULL,
    price DECIMAL(10,2) NOT NULL
);

-- PostgreSQL
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(200) NOT NULL,
    price NUMERIC(10,2) NOT NULL
);
```

Real-World Examples

Example 1: Complete E-commerce Database

```

```

```sql
-- 1. Categories table
CREATE TABLE categories (
    category_id SERIAL PRIMARY KEY,
    category_name VARCHAR(100) UNIQUE NOT NULL,
    parent_category_id INT,
    description TEXT,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Self-referencing foreign key for subcategories
    FOREIGN KEY (parent_category_id) REFERENCES categories(category_id)
);

-- 2. Products table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    sku VARCHAR(50) UNIQUE NOT NULL,
    product_name VARCHAR(200) NOT NULL,
    description TEXT,
    price NUMERIC(10,2) NOT NULL,
    cost_price NUMERIC(10,2),
    stock_quantity INT DEFAULT 0,
    min_stock_level INT DEFAULT 0,
    category_id INT NOT NULL,
    brand VARCHAR(100),
    weight NUMERIC(8,3),
    dimensions VARCHAR(50), -- "10x5x3 cm"
    is_active BOOLEAN DEFAULT TRUE,
    is_featured BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Constraints
    FOREIGN KEY (category_id) REFERENCES categories(category_id),
    CHECK (price > 0),
    CHECK (stock_quantity >= 0),
    CHECK (cost_price IS NULL OR cost_price >= 0)
);

-- 3. Customers table
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    phone VARCHAR(20),
    date_of_birth DATE,
    gender CHAR(1) CHECK (gender IN ('M', 'F', 'O')),
    is_active BOOLEAN DEFAULT TRUE,
    email_verified BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    last_login TIMESTAMP,

    -- Constraints
    CHECK (email LIKE '%@%'),
    CHECK (date_of_birth IS NULL OR date_of_birth < CURRENT_DATE)
);

-- 4. Addresses table (one-to-many with customers)
CREATE TABLE addresses (
    address_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    address_type VARCHAR(20) DEFAULT 'shipping', -- 'shipping', 'billing'
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    company VARCHAR(100),
    address_line1 VARCHAR(255) NOT NULL,
    address_line2 VARCHAR(255),
    city VARCHAR(100) NOT NULL,
    state_province VARCHAR(100),
    postal_code VARCHAR(20) NOT NULL,
    country VARCHAR(100) NOT NULL,
    phone VARCHAR(20),
    is_default BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE CASCADE,
    CHECK (address_type IN ('shipping', 'billing'))
);

-- 5. Orders table
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    order_number VARCHAR(50) UNIQUE NOT NULL,
    customer_id INT NOT NULL,
    order_status VARCHAR(20) DEFAULT 'pending',
    payment_status VARCHAR(20) DEFAULT 'pending',
    subtotal NUMERIC(10,2) NOT NULL,
    tax_amount NUMERIC(10,2) DEFAULT 0,
    shipping_amount NUMERIC(10,2) DEFAULT 0,
    discount_amount NUMERIC(10,2) DEFAULT 0,
    total_amount NUMERIC(10,2) NOT NULL,
    currency CHAR(3) DEFAULT 'USD',

    -- Address snapshots (data at time of order)
    shipping_address_id INT,
    billing_address_id INT,

    -- Timestamps
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    shipped_date TIMESTAMP,
    delivered_date TIMESTAMP,

    -- Notes
    customer_notes TEXT,
```

```
    admin_notes TEXT,

    FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
    FOREIGN KEY (shipping_address_id) REFERENCES addresses(address_id),
    FOREIGN KEY (billing_address_id) REFERENCES addresses(address_id),

    CHECK (order_status IN ('pending', 'processing', 'shipped', 'delivered',
'cancelled')),
    CHECK (payment_status IN ('pending', 'paid', 'failed', 'refunded')),
    CHECK (subtotal >= 0),
    CHECK (total_amount >= 0)
);

-- 6. Order items table (many-to-many between orders and products)
CREATE TABLE order_items (
    order_item_id SERIAL PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    unit_price NUMERIC(10,2) NOT NULL,
    total_price NUMERIC(10,2) NOT NULL,

    -- Product snapshot at time of order
    product_name VARCHAR(200) NOT NULL,
    product_sku VARCHAR(50) NOT NULL,

    FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(product_id),

    CHECK (quantity > 0),
    CHECK (unit_price >= 0),
    CHECK (total_price >= 0),

    -- Ensure total_price = quantity * unit_price
    CHECK (total_price = quantity * unit_price)
);
```

Example 2: Blog/CMS Database

```
-- 1. Users table
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    bio TEXT,
    avatar_url VARCHAR(500),
    role VARCHAR(20) DEFAULT 'user',
    is_active BOOLEAN DEFAULT TRUE,
```

```
    email_verified BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,

    CHECK (role IN ('admin', 'editor', 'author', 'user')),
    CHECK (email LIKE '%@%')
);

-- 2. Categories table
CREATE TABLE post_categories (
    category_id SERIAL PRIMARY KEY,
    category_name VARCHAR(100) UNIQUE NOT NULL,
    slug VARCHAR(100) UNIQUE NOT NULL,
    description TEXT,
    parent_category_id INT,
    sort_order INT DEFAULT 0,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (parent_category_id) REFERENCES post_categories(category_id)
);

-- 3. Posts table
CREATE TABLE posts (
    post_id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(255) UNIQUE NOT NULL,
    excerpt TEXT,
    content TEXT NOT NULL,
    featured_image_url VARCHAR(500),
    author_id INT NOT NULL,
    category_id INT,
    status VARCHAR(20) DEFAULT 'draft',
    is_featured BOOLEAN DEFAULT FALSE,
    view_count INT DEFAULT 0,
    comment_count INT DEFAULT 0,
    like_count INT DEFAULT 0,

    -- SEO fields
    meta_title VARCHAR(255),
    meta_description VARCHAR(500),
    meta_keywords VARCHAR(500),

    -- Timestamps
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    published_at TIMESTAMP,

    FOREIGN KEY (author_id) REFERENCES users(user_id),
    FOREIGN KEY (category_id) REFERENCES post_categories(category_id),

    CHECK (status IN ('draft', 'published', 'archived')),
    CHECK (view_count >= 0),
```

```
    CHECK (comment_count >= 0),
    CHECK (like_count >= 0)
);

-- 4. Comments table
CREATE TABLE comments (
    comment_id SERIAL PRIMARY KEY,
    post_id INT NOT NULL,
    parent_comment_id INT, -- For nested comments
    author_id INT, -- NULL for guest comments
    author_name VARCHAR(100), -- For guest comments
    author_email VARCHAR(255), -- For guest comments
    content TEXT NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    ip_address INET, -- PostgreSQL specific
    user_agent TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    FOREIGN KEY (parent_comment_id) REFERENCES comments(comment_id),
    FOREIGN KEY (author_id) REFERENCES users(user_id),

    CHECK (status IN ('pending', 'approved', 'spam', 'trash')),
    -- Either registered user OR guest info required
    CHECK (
        (author_id IS NOT NULL) OR
        (author_name IS NOT NULL AND author_email IS NOT NULL)
    )
);
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"How do you design a database schema?"**
    - Start with entities and relationships
    - Normalize to reduce redundancy
    - Add constraints for data integrity
    - Consider performance and indexing
    - Plan for scalability
2. **"What's the difference between PRIMARY KEY and UNIQUE?"**
    - PRIMARY KEY: Unique + NOT NULL + one per table
    - UNIQUE: Can have NULLs + multiple per table
    - PRIMARY KEY creates clustered index (usually)
3. **"When would you use CASCADE vs RESTRICT for foreign keys?"**

- CASCADE: Delete related records (order_items when order deleted)
- RESTRICT: Prevent deletion if references exist
- SET NULL: Set foreign key to NULL when parent deleted

Schema Design Best Practices:

1. **Naming Conventions**

```
-- Good: Consistent, descriptive names
CREATE TABLE user_profiles (
    user_id INT,
    profile_id INT,
    created_at TIMESTAMP
);

-- Bad: Inconsistent, unclear
CREATE TABLE UserProf (
    uid INT,
    PID INT,
    dt TIMESTAMP
);
```

2. **Use Appropriate Constraints**

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    price NUMERIC(10,2) NOT NULL CHECK (price > 0),
    stock_quantity INT DEFAULT 0 CHECK (stock_quantity >= 0),
    category_id INT NOT NULL,
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
);
```

3. **Plan for Soft Deletes**

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    is_deleted BOOLEAN DEFAULT FALSE,
    deleted_at TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

⚠ Things to Watch Out For

1. **Foreign Key Constraint Differences**

```sql
-- MySQL: Foreign keys optional by default
CREATE TABLE orders (
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
) ENGINE=InnoDB; -- Must specify InnoDB for FK support

-- PostgreSQL: Foreign keys always enforced
CREATE TABLE orders (
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
); -- Always enforced
```

2. **Auto-Increment Reset Behavior**

```sql
-- MySQL: AUTO_INCREMENT continues after DELETE
DELETE FROM products WHERE product_id = 5;
INSERT INTO products (name) VALUES ('New Product'); -- Gets ID 6, not 5

-- PostgreSQL: SERIAL continues after DELETE
DELETE FROM products WHERE product_id = 5;
INSERT INTO products (name) VALUES ('New Product'); -- Gets ID 6, not 5

-- To reset (be careful!):
-- MySQL
ALTER TABLE products AUTO_INCREMENT = 1;
-- PostgreSQL
ALTER SEQUENCE products_product_id_seq RESTART WITH 1;
```

3. **Case Sensitivity in Names**

```sql
-- MySQL: Generally case-insensitive
CREATE TABLE Users (id INT); -- Creates 'users' table
SELECT * FROM users; -- Works

-- PostgreSQL: Case-sensitive
CREATE TABLE Users (id INT); -- Creates 'Users' table
SELECT * FROM users; -- Error! Table is 'Users'
SELECT * FROM "Users"; -- Correct
```

4. **Default Value Differences**

```sql
-- MySQL: Can use functions in defaults
CREATE TABLE logs (
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
```

```
);

-- PostgreSQL: Limited function defaults
CREATE TABLE logs (
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    -- No automatic ON UPDATE equivalent
);
```

5. **Storage Engine Considerations (MySQL)**

```
-- MySQL: Different storage engines
CREATE TABLE products (
    id INT PRIMARY KEY
) ENGINE=InnoDB; -- Supports transactions, foreign keys

CREATE TABLE search_cache (
    keyword VARCHAR(100)
) ENGINE=MEMORY; -- Faster, but data lost on restart
```

6. **Index Creation Timing**

```
-- Create indexes after table creation for better performance
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(200) NOT NULL,
    category_id INT,
    price NUMERIC(10,2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Add indexes separately
CREATE INDEX idx_products_category ON products(category_id);
CREATE INDEX idx_products_price ON products(price);
CREATE INDEX idx_products_name ON products(name);
CREATE INDEX idx_products_created_at ON products(created_at);

-- Composite index for common query patterns
CREATE INDEX idx_products_category_price ON products(category_id, price);
```

🚀 Next Steps

In the next chapter, we'll dive into **CRUD Operations** (Create, Read, Update, Delete) - the fundamental operations you'll use to manipulate data in your tables. You'll learn how to insert, select, update, and delete data effectively.

📝 Quick Practice

Design a database schema for a library management system:

Required entities:

- Books (title, author, ISBN, publication year, copies available)
- Members (name, email, phone, membership date)
- Loans (which book, which member, loan date, due date, return date)
- Authors (name, biography, birth date)
- Categories (fiction, non-fiction, science, etc.)

Consider:

- What relationships exist between entities?
- What constraints would ensure data integrity?
- What indexes would improve query performance?
- How would you handle a book with multiple authors?

📓 Chapter 4: CRUD Operations (SELECT, INSERT, UPDATE, DELETE)

🎯 What You'll Learn

- Master the four fundamental database operations
- Write efficient SELECT queries with filtering and sorting
- Insert single and multiple records safely
- Update data with precision and safety
- Delete data responsibly with proper safeguards

🕮 Concept Explanation

CRUD stands for the four basic operations you can perform on data:

- **C**reate (INSERT) - Add new data
- **R**ead (SELECT) - Retrieve existing data
- **U**pdate (UPDATE) - Modify existing data
- **D**elete (DELETE) - Remove data

These operations form the foundation of all database interactions. Whether you're building a web app, mobile app, or data analysis tool, you'll use these operations constantly.

Key Principles:

1. **Safety First**: Always use WHERE clauses with UPDATE/DELETE
2. **Performance**: Write efficient queries that use indexes
3. **Data Integrity**: Respect constraints and relationships
4. **Transactions**: Group related operations together

5. **Testing**: Test queries on small datasets first

🔧 CREATE (INSERT) Operations

Basic INSERT Syntax

```
-- Insert single record
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);

-- Insert multiple records
INSERT INTO table_name (column1, column2, column3)
VALUES
    (value1a, value2a, value3a),
    (value1b, value2b, value3b),
    (value1c, value2c, value3c);
```

Real-World Examples

1. Adding New Customers:

```
-- Single customer
INSERT INTO customers (first_name, last_name, email, phone)
VALUES ('John', 'Doe', 'john.doe@email.com', '+1-555-0123');

-- Multiple customers at once (more efficient)
INSERT INTO customers (first_name, last_name, email, phone, created_at)
VALUES
    ('Alice', 'Smith', 'alice@email.com', '+1-555-0124', CURRENT_TIMESTAMP),
    ('Bob', 'Johnson', 'bob@email.com', '+1-555-0125', CURRENT_TIMESTAMP),
    ('Carol', 'Williams', 'carol@email.com', '+1-555-0126', CURRENT_TIMESTAMP);
```

2. Adding Products with Categories:

```
-- First, ensure category exists
INSERT INTO categories (category_name, description)
VALUES ('Electronics', 'Electronic devices and accessories');

-- Then add products
INSERT INTO products (sku, product_name, price, category_id, stock_quantity)
VALUES
    ('LAPTOP001', 'Gaming Laptop Pro', 1299.99, 1, 15),
    ('MOUSE001', 'Wireless Gaming Mouse', 79.99, 1, 50),
    ('KEYBOARD001', 'Mechanical Keyboard RGB', 149.99, 1, 30);
```

3. Handling Auto-Increment and Defaults:

```sql
-- Let database generate ID and timestamp
INSERT INTO orders (customer_id, total_amount, order_status)
VALUES (1, 1529.97, 'pending');

-- Get the generated ID (MySQL)
SELECT LAST_INSERT_ID();

-- Get the generated ID (PostgreSQL)
INSERT INTO orders (customer_id, total_amount, order_status)
VALUES (1, 1529.97, 'pending')
RETURNING order_id;
```

4. INSERT with SELECT (Copy Data):

```sql
-- Copy products from one category to another
INSERT INTO product_archive (product_name, price, archived_date)
SELECT product_name, price, CURRENT_TIMESTAMP
FROM products
WHERE category_id = 5 AND is_active = FALSE;
```

INSERT Safety and Error Handling

```sql
-- Handle duplicate key errors gracefully

-- MySQL: INSERT IGNORE (skips duplicates)
INSERT IGNORE INTO customers (email, first_name, last_name)
VALUES ('existing@email.com', 'John', 'Doe');

-- MySQL: ON DUPLICATE KEY UPDATE
INSERT INTO customers (email, first_name, last_name, updated_at)
VALUES ('john@email.com', 'John', 'Doe', CURRENT_TIMESTAMP)
ON DUPLICATE KEY UPDATE
    first_name = VALUES(first_name),
    last_name = VALUES(last_name),
    updated_at = CURRENT_TIMESTAMP;

-- PostgreSQL: ON CONFLICT
INSERT INTO customers (email, first_name, last_name, updated_at)
VALUES ('john@email.com', 'John', 'Doe', CURRENT_TIMESTAMP)
ON CONFLICT (email) DO UPDATE SET
    first_name = EXCLUDED.first_name,
    last_name = EXCLUDED.last_name,
    updated_at = CURRENT_TIMESTAMP;
```

🔍 READ (SELECT) Operations

Basic SELECT Syntax

```
-- Basic structure
SELECT column1, column2, column3
FROM table_name
WHERE condition
ORDER BY column1 ASC/DESC
LIMIT number;
```

Essential SELECT Patterns

1. Basic Filtering:

```
-- Select specific columns
SELECT first_name, last_name, email
FROM customers
WHERE is_active = TRUE;

-- Multiple conditions
SELECT product_name, price
FROM products
WHERE price BETWEEN 50 AND 200
  AND category_id IN (1, 2, 3)
  AND stock_quantity > 0;

-- Pattern matching
SELECT *
FROM customers
WHERE email LIKE '%@gmail.com'
  AND first_name ILIKE 'john%';  -- ILIKE is case-insensitive (PostgreSQL)
```

2. Sorting and Limiting:

```
-- Order by multiple columns
SELECT product_name, price, stock_quantity
FROM products
ORDER BY category_id ASC, price DESC;

-- Top 10 most expensive products
SELECT product_name, price
FROM products
ORDER BY price DESC
LIMIT 10;

-- Pagination (skip first 20, get next 10)
SELECT product_name, price
FROM products
```

```
ORDER BY product_name
LIMIT 10 OFFSET 20;
```

3. Aggregate Functions:

```
-- Count records
SELECT COUNT(*) as total_customers
FROM customers
WHERE created_at >= '2024-01-01';

-- Sum, average, min, max
SELECT
    COUNT(*) as total_products,
    AVG(price) as average_price,
    MIN(price) as cheapest_price,
    MAX(price) as most_expensive,
    SUM(stock_quantity * price) as total_inventory_value
FROM products
WHERE is_active = TRUE;
```

4. Grouping Data:

```
-- Sales by category
SELECT
    c.category_name,
    COUNT(p.product_id) as product_count,
    AVG(p.price) as average_price,
    SUM(p.stock_quantity) as total_stock
FROM categories c
LEFT JOIN products p ON c.category_id = p.category_id
GROUP BY c.category_id, c.category_name
HAVING COUNT(p.product_id) > 0
ORDER BY product_count DESC;
```

5. Advanced Filtering:

```
-- Subqueries
SELECT customer_id, first_name, last_name
FROM customers
WHERE customer_id IN (
    SELECT DISTINCT customer_id
    FROM orders
    WHERE order_date >= '2024-01-01'
);

-- EXISTS clause
SELECT c.customer_id, c.first_name, c.last_name
```

```
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.total_amount > 1000
);

-- CASE statements
SELECT
    product_name,
    price,
    CASE
        WHEN price < 50 THEN 'Budget'
        WHEN price BETWEEN 50 AND 200 THEN 'Mid-range'
        WHEN price > 200 THEN 'Premium'
        ELSE 'Unknown'
    END as price_category
FROM products;
```

UPDATE Operations

Basic UPDATE Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

Safe UPDATE Practices

1. Always Use WHERE Clause:

```
-- DANGEROUS: Updates ALL records
UPDATE products SET price = price * 1.1;

-- SAFE: Updates specific records
UPDATE products
SET price = price * 1.1
WHERE category_id = 1 AND is_active = TRUE;
```

2. Update Single Records:

```
-- Update customer information
UPDATE customers
SET
    first_name = 'John',
```

```
    last_name = 'Smith',
    phone = '+1-555-9999',
    updated_at = CURRENT_TIMESTAMP
WHERE customer_id = 123;

-- Verify the update
SELECT customer_id, first_name, last_name, phone, updated_at
FROM customers
WHERE customer_id = 123;
```

3. Conditional Updates:

```
-- Update stock after sale
UPDATE products
SET
    stock_quantity = stock_quantity - 5,
    updated_at = CURRENT_TIMESTAMP
WHERE product_id = 456
  AND stock_quantity >= 5;  -- Only if enough stock

-- Update order status
UPDATE orders
SET
    order_status = 'shipped',
    shipped_date = CURRENT_TIMESTAMP
WHERE order_id = 789
  AND order_status = 'processing';
```

4. Bulk Updates:

```
-- Deactivate old products
UPDATE products
SET
    is_active = FALSE,
    updated_at = CURRENT_TIMESTAMP
WHERE created_at < '2020-01-01'
  AND stock_quantity = 0;

-- Apply discount to category
UPDATE products
SET
    price = price * 0.9,  -- 10% discount
    updated_at = CURRENT_TIMESTAMP
WHERE category_id = 5
  AND is_active = TRUE;
```

5. UPDATE with JOIN (MySQL):

```sql
-- Update product prices based on category
UPDATE products p
JOIN categories c ON p.category_id = c.category_id
SET p.price = p.price * 1.15
WHERE c.category_name = 'Electronics';
```

6. UPDATE with FROM (PostgreSQL):

```sql
-- Update product prices based on category
UPDATE products
SET price = price * 1.15
FROM categories
WHERE products.category_id = categories.category_id
  AND categories.category_name = 'Electronics';
```

🗑️ DELETE Operations

Basic DELETE Syntax

```sql
DELETE FROM table_name
WHERE condition;
```

Safe DELETE Practices

1. Always Test with SELECT First:

```sql
-- STEP 1: Test what will be deleted
SELECT customer_id, first_name, last_name, created_at
FROM customers
WHERE is_active = FALSE
  AND created_at < '2020-01-01'
  AND customer_id NOT IN (
      SELECT DISTINCT customer_id
      FROM orders
      WHERE customer_id IS NOT NULL
  );

-- STEP 2: If results look correct, then delete
DELETE FROM customers
WHERE is_active = FALSE
  AND created_at < '2020-01-01'
  AND customer_id NOT IN (
      SELECT DISTINCT customer_id
      FROM orders
```

```
      WHERE customer_id IS NOT NULL
  );
```

2. Soft Delete vs Hard Delete:

```
-- Soft delete (recommended for important data)
UPDATE customers
SET
    is_deleted = TRUE,
    deleted_at = CURRENT_TIMESTAMP
WHERE customer_id = 123;

-- Hard delete (permanent removal)
DELETE FROM customers
WHERE customer_id = 123;
```

3. Delete with Foreign Key Considerations:

```
-- Delete order and related items (with CASCADE)
DELETE FROM orders WHERE order_id = 456;
-- This automatically deletes related order_items if CASCADE is set

-- Manual cleanup approach
BEGIN TRANSACTION;

-- Delete related records first
DELETE FROM order_items WHERE order_id = 456;

-- Then delete main record
DELETE FROM orders WHERE order_id = 456;

COMMIT;
```

4. Conditional Deletes:

```
-- Delete expired sessions
DELETE FROM user_sessions
WHERE expires_at < CURRENT_TIMESTAMP;

-- Delete duplicate records (keep newest)
DELETE FROM customers c1
WHERE EXISTS (
    SELECT 1
    FROM customers c2
    WHERE c2.email = c1.email
      AND c2.customer_id > c1.customer_id
);
```

Real-World CRUD Examples

Example 1: E-commerce Order Processing

```sql
-- 1. CREATE: New customer places order
BEGIN TRANSACTION;

-- Insert customer if not exists
INSERT INTO customers (email, first_name, last_name)
VALUES ('newcustomer@email.com', 'Jane', 'Doe')
ON CONFLICT (email) DO NOTHING;

-- Get customer ID
SET @customer_id = (
    SELECT customer_id
    FROM customers
    WHERE email = 'newcustomer@email.com'
);

-- Create order
INSERT INTO orders (customer_id, order_status, total_amount)
VALUES (@customer_id, 'pending', 299.98);

SET @order_id = LAST_INSERT_ID();

-- Add order items
INSERT INTO order_items (order_id, product_id, quantity, unit_price, total_price)
VALUES
    (@order_id, 101, 2, 99.99, 199.98),
    (@order_id, 102, 1, 99.99, 99.99);

-- Update product stock
UPDATE products
SET stock_quantity = stock_quantity - 2
WHERE product_id = 101 AND stock_quantity >= 2;

UPDATE products
SET stock_quantity = stock_quantity - 1
WHERE product_id = 102 AND stock_quantity >= 1;

COMMIT;
```

Example 2: Blog Content Management

```sql
-- 1. CREATE: Publish new blog post
INSERT INTO posts (title, slug, content, author_id, category_id, status)
VALUES (
```

```
    'Getting Started with SQL',
    'getting-started-with-sql',
    'SQL is the standard language for managing relational databases...',
    1,
    2,
    'published'
);

-- 2. READ: Get published posts with author info
SELECT
    p.title,
    p.slug,
    p.excerpt,
    p.published_at,
    u.first_name || ' ' || u.last_name as author_name,
    c.category_name,
    p.view_count,
    p.comment_count
FROM posts p
JOIN users u ON p.author_id = u.user_id
JOIN post_categories c ON p.category_id = c.category_id
WHERE p.status = 'published'
ORDER BY p.published_at DESC
LIMIT 10;

-- 3. UPDATE: Increment view count
UPDATE posts
SET view_count = view_count + 1
WHERE slug = 'getting-started-with-sql';

-- 4. DELETE: Remove spam comments
DELETE FROM comments
WHERE status = 'spam'
  AND created_at < CURRENT_TIMESTAMP - INTERVAL '30 days';
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"How do you prevent accidental data loss?"**
    - Always use WHERE clauses with UPDATE/DELETE
    - Test with SELECT first
    - Use transactions for multi-step operations
    - Implement soft deletes for critical data
    - Regular backups
2. **"What's the difference between DELETE and TRUNCATE?"**
    - DELETE: Removes rows based on WHERE clause, can be rolled back
    - TRUNCATE: Removes all rows, faster, resets auto-increment, limited rollback

- DROP: Removes entire table structure

3. **"How do you handle duplicate data during INSERT?"**
   - MySQL: INSERT IGNORE, ON DUPLICATE KEY UPDATE
   - PostgreSQL: ON CONFLICT DO UPDATE/NOTHING
   - Check for existence before inserting

Performance Tips:

1. **Use Indexes for WHERE Clauses:**

```
-- Slow without index on email
SELECT * FROM customers WHERE email = 'john@email.com';

-- Create index for faster lookups
CREATE INDEX idx_customers_email ON customers(email);
```

2. **Batch Operations:**

```
-- Slow: Multiple single inserts
INSERT INTO products (name, price) VALUES ('Product 1', 10.00);
INSERT INTO products (name, price) VALUES ('Product 2', 15.00);

-- Fast: Batch insert
INSERT INTO products (name, price) VALUES
    ('Product 1', 10.00),
    ('Product 2', 15.00);
```

3. **Limit Large Operations:**

```
-- Process in chunks to avoid locking
UPDATE products
SET is_active = FALSE
WHERE created_at < '2020-01-01'
LIMIT 1000;
```

⚠ Things to Watch Out For

1. **Missing WHERE Clauses**

```
-- DISASTER: Updates all records
UPDATE customers SET email = 'test@email.com';
```

```
-- SAFE: Updates specific record
UPDATE customers SET email = 'test@email.com' WHERE customer_id = 1;
```

2. **Foreign Key Violations**

```
-- Error: Cannot delete customer with orders
DELETE FROM customers WHERE customer_id = 1;

-- Solution: Delete orders first, or use CASCADE
DELETE FROM order_items WHERE order_id IN (
    SELECT order_id FROM orders WHERE customer_id = 1
);
DELETE FROM orders WHERE customer_id = 1;
DELETE FROM customers WHERE customer_id = 1;
```

3. **Data Type Mismatches**

```
-- Error: String in numeric field
INSERT INTO products (price) VALUES ('not a number');

-- Correct: Proper data type
INSERT INTO products (price) VALUES (29.99);
```

4. **Transaction Management**

```
-- Always wrap related operations in transactions
BEGIN TRANSACTION;

UPDATE products SET stock_quantity = stock_quantity - 1 WHERE product_id = 1;
INSERT INTO order_items (product_id, quantity) VALUES (1, 1);

-- Check if both operations succeeded
IF @@ERROR = 0
    COMMIT;
ELSE
    ROLLBACK;
```

5. **NULL Handling**

```
-- Wrong: NULL comparison
SELECT * FROM customers WHERE phone = NULL;

-- Correct: IS NULL
SELECT * FROM customers WHERE phone IS NULL;
```

```
-- Handle NULLs in calculations
SELECT
    product_name,
    COALESCE(discount_price, price) as final_price
FROM products;
```

🚀 Next Steps

In the next chapter, we'll explore **Constraints** - the rules that ensure your data stays clean and consistent. You'll learn about primary keys, foreign keys, unique constraints, check constraints, and how they protect your database integrity.

📝 Quick Practice

Using the e-commerce database from previous chapters:

1. **INSERT**: Add 5 new products in the "Books" category
2. **SELECT**: Find all customers who have placed orders over $500
3. **UPDATE**: Apply a 15% discount to all products in the "Clothing" category
4. **DELETE**: Remove all products that have 0 stock and haven't been ordered in the last year

Remember to:

- Test your SELECT queries first
- Use transactions for multi-step operations
- Verify your results after each operation

📓 Chapter 5: Constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, etc.)

🎯 What You'll Learn

- Understanding database constraints and their importance
- Primary keys, foreign keys, and referential integrity
- Unique constraints and check constraints
- NOT NULL constraints and default values
- How constraints prevent data corruption and maintain quality

---

📖 Concept Explanation

Database constraints are rules enforced by the database to ensure data integrity, consistency, and validity. Think of them as "guardrails" that prevent bad data from entering your database and maintain relationships between tables.

Why Constraints Matter:

1. **Data Integrity**: Prevent invalid or inconsistent data
2. **Referential Integrity**: Maintain relationships between tables
3. **Business Rules**: Enforce application logic at the database level
4. **Performance**: Some constraints create indexes automatically
5. **Documentation**: Constraints serve as living documentation of your data rules

Types of Constraints:

- **PRIMARY KEY**: Unique identifier for each row
- **FOREIGN KEY**: Links to another table's primary key
- **UNIQUE**: Ensures column values are unique
- **NOT NULL**: Prevents empty values
- **CHECK**: Custom validation rules
- **DEFAULT**: Provides default values

🔧 PRIMARY KEY Constraints

What is a Primary Key?

A primary key uniquely identifies each row in a table. Every table should have one (and only one) primary key.

Rules:

- Must be unique across all rows
- Cannot be NULL
- Should be immutable (rarely changed)
- Automatically creates a unique index

Primary Key Examples

1. Single Column Primary Key:

```
-- Auto-increment primary key (most common)
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,  -- PostgreSQL
    -- customer_id INT AUTO_INCREMENT PRIMARY KEY,  -- MySQL
    email VARCHAR(255) NOT NULL,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL
);

-- Natural primary key (use with caution)
CREATE TABLE countries (
    country_code CHAR(2) PRIMARY KEY,  -- 'US', 'CA', 'UK'
    country_name VARCHAR(100) NOT NULL,
    continent VARCHAR(50)
);
```

2. Composite Primary Key:

```sql
-- Multiple columns form the primary key
CREATE TABLE order_items (
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,

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

    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);

-- User roles assignment
CREATE TABLE user_roles (
    user_id INT NOT NULL,
    role_id INT NOT NULL,
    assigned_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    assigned_by INT,

    PRIMARY KEY (user_id, role_id),

    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (role_id) REFERENCES roles(role_id),
    FOREIGN KEY (assigned_by) REFERENCES users(user_id)
);
```

3. Adding Primary Key to Existing Table:

```sql
-- Add primary key constraint
ALTER TABLE existing_table
ADD CONSTRAINT pk_existing_table PRIMARY KEY (id);

-- Drop primary key (MySQL)
ALTER TABLE existing_table DROP PRIMARY KEY;

-- Drop primary key (PostgreSQL)
ALTER TABLE existing_table DROP CONSTRAINT existing_table_pkey;
```

🔗 FOREIGN KEY Constraints

What is a Foreign Key?

A foreign key creates a link between two tables, ensuring referential integrity. It must match a value in the referenced table's primary key or be NULL.

Foreign Key Examples

1. Basic Foreign Key:

```
-- Orders table references customers
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,

    -- Foreign key constraint
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Alternative syntax
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL REFERENCES customers(customer_id),
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL
);
```

2. Foreign Key with Actions:

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,

    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
        ON DELETE RESTRICT    -- Prevent deletion if orders exist
        ON UPDATE CASCADE     -- Update order.customer_id if customer.customer_id
changes
);

-- Different action options:
CREATE TABLE order_items (
    order_item_id SERIAL PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,

    FOREIGN KEY (order_id) REFERENCES orders(order_id)
        ON DELETE CASCADE,    -- Delete order_items when order is deleted
```

```
    FOREIGN KEY (product_id) REFERENCES products(product_id)
        ON DELETE SET NULL    -- Set product_id to NULL if product is deleted
        ON UPDATE CASCADE
);
```

3. Self-Referencing Foreign Key:

```
-- Employee hierarchy
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    manager_id INT,
    department VARCHAR(100),

    -- Self-referencing foreign key
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
        ON DELETE SET NULL
);

-- Category hierarchy
CREATE TABLE categories (
    category_id SERIAL PRIMARY KEY,
    category_name VARCHAR(100) NOT NULL,
    parent_category_id INT,

    FOREIGN KEY (parent_category_id) REFERENCES categories(category_id)
        ON DELETE CASCADE
);
```

4. Multiple Foreign Keys:

```
CREATE TABLE product_reviews (
    review_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    customer_id INT NOT NULL,
    rating INT NOT NULL CHECK (rating BETWEEN 1 AND 5),
    review_text TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Multiple foreign keys
    FOREIGN KEY (product_id) REFERENCES products(product_id)
        ON DELETE CASCADE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
        ON DELETE CASCADE,

    -- Ensure one review per customer per product
    UNIQUE (product_id, customer_id)
);
```

Foreign Key Actions Explained

Action	Description	Use Case
`RESTRICT`	Prevent parent deletion/update	Default, safest option
`CASCADE`	Delete/update child records too	Order items when order deleted
`SET NULL`	Set foreign key to NULL	Optional relationships
`SET DEFAULT`	Set to default value	Rarely used
`NO ACTION`	Same as RESTRICT	PostgreSQL default

🔒 UNIQUE Constraints

What is a UNIQUE Constraint?

Ensures that all values in a column (or combination of columns) are unique across the table.

UNIQUE Examples

1. Single Column UNIQUE:

```
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,  -- Inline unique constraint
    username VARCHAR(50),
    phone VARCHAR(20),

    -- Named unique constraints
    CONSTRAINT uk_customers_username UNIQUE (username),
    CONSTRAINT uk_customers_phone UNIQUE (phone)
);
```

2. Composite UNIQUE:

```
-- Ensure unique combination of first_name + last_name + date_of_birth
CREATE TABLE persons (
    person_id SERIAL PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    date_of_birth DATE NOT NULL,
    email VARCHAR(255),

    UNIQUE (first_name, last_name, date_of_birth)
);

-- Product SKU must be unique per supplier
```

```sql
CREATE TABLE supplier_products (
    id SERIAL PRIMARY KEY,
    supplier_id INT NOT NULL,
    product_sku VARCHAR(50) NOT NULL,
    product_name VARCHAR(200) NOT NULL,
    price DECIMAL(10,2),

    FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id),
    UNIQUE (supplier_id, product_sku)
);
```

3. Adding/Dropping UNIQUE Constraints:

```sql
-- Add unique constraint to existing table
ALTER TABLE customers
ADD CONSTRAINT uk_customers_email UNIQUE (email);

-- Drop unique constraint
ALTER TABLE customers
DROP CONSTRAINT uk_customers_email;

-- MySQL syntax for dropping
ALTER TABLE customers
DROP INDEX uk_customers_email;
```

✅ CHECK Constraints

What is a CHECK Constraint?

Enforces custom business rules by validating data before it's inserted or updated.

CHECK Examples

1. Range Validation:

```sql
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(200) NOT NULL,
    price DECIMAL(10,2) NOT NULL CHECK (price > 0),
    discount_percentage INT CHECK (discount_percentage BETWEEN 0 AND 100),
    stock_quantity INT CHECK (stock_quantity >= 0),
    weight DECIMAL(8,3) CHECK (weight > 0),

    -- Named check constraints
    CONSTRAINT chk_products_price_range CHECK (price BETWEEN 0.01 AND 999999.99)
);
```

2. Enum-like Validation:

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    order_status VARCHAR(20) NOT NULL
        CHECK (order_status IN ('pending', 'processing', 'shipped', 'delivered',
'cancelled')),
    payment_status VARCHAR(20) NOT NULL
        CHECK (payment_status IN ('pending', 'paid', 'failed', 'refunded')),
    priority VARCHAR(10) DEFAULT 'normal'
        CHECK (priority IN ('low', 'normal', 'high', 'urgent')),

    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

3. Complex Business Rules:

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    email VARCHAR(255) NOT NULL,
    hire_date DATE NOT NULL,
    birth_date DATE,
    salary DECIMAL(10,2),
    department VARCHAR(100),

    -- Business rule validations
    CHECK (hire_date >= '1900-01-01'),
    CHECK (birth_date IS NULL OR birth_date < hire_date),
    CHECK (salary IS NULL OR salary > 0),
    CHECK (email LIKE '%@%'),

    -- Age must be at least 16 at hire date
    CHECK (birth_date IS NULL OR
           (hire_date - birth_date) >= INTERVAL '16 years')
);
```

4. Cross-Column Validation:

```
CREATE TABLE promotions (
    promotion_id SERIAL PRIMARY KEY,
    promotion_name VARCHAR(200) NOT NULL,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    discount_percentage INT,
    discount_amount DECIMAL(10,2),
```

```
    -- Date validation
    CHECK (end_date > start_date),

    -- Either percentage OR amount discount, not both
    CHECK (
        (discount_percentage IS NOT NULL AND discount_amount IS NULL) OR
        (discount_percentage IS NULL AND discount_amount IS NOT NULL)
    ),

    -- Percentage must be reasonable
    CHECK (discount_percentage IS NULL OR
           discount_percentage BETWEEN 1 AND 100),

    -- Amount must be positive
    CHECK (discount_amount IS NULL OR discount_amount > 0)
);
```

🚫 NOT NULL Constraints

What is NOT NULL?

Prevents empty (NULL) values in a column, ensuring required data is always present.

NOT NULL Examples

1. Required Fields:

```
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,        -- Required
    first_name VARCHAR(100) NOT NULL,   -- Required
    last_name VARCHAR(100) NOT NULL,    -- Required
    phone VARCHAR(20),                  -- Optional
    date_of_birth DATE,                 -- Optional
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

2. Adding/Removing NOT NULL:

```
-- Add NOT NULL to existing column (ensure no NULLs exist first)
UPDATE customers SET phone = 'Unknown' WHERE phone IS NULL;
ALTER TABLE customers ALTER COLUMN phone SET NOT NULL;

-- Remove NOT NULL constraint
ALTER TABLE customers ALTER COLUMN phone DROP NOT NULL;
```

```
-- MySQL syntax
ALTER TABLE customers MODIFY phone VARCHAR(20) NULL;
```

🎯 DEFAULT Constraints

What is DEFAULT?

Provides automatic values when no value is specified during INSERT.

DEFAULT Examples

1. Common Defaults:

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    order_status VARCHAR(20) DEFAULT 'pending',
    payment_status VARCHAR(20) DEFAULT 'pending',
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,
    currency CHAR(3) DEFAULT 'USD',
    is_gift BOOLEAN DEFAULT FALSE,

    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

2. Function-based Defaults:

```
CREATE TABLE audit_log (
    log_id SERIAL PRIMARY KEY,
    table_name VARCHAR(100) NOT NULL,
    action VARCHAR(20) NOT NULL,
    user_id INT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    date_only DATE DEFAULT CURRENT_DATE,

    -- PostgreSQL: Generate UUID
    session_id UUID DEFAULT gen_random_uuid(),

    -- MySQL: Generate UUID
    -- session_id CHAR(36) DEFAULT (UUID())
);
```

3. Computed Defaults:

```
-- PostgreSQL: More advanced defaults
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(200) NOT NULL,
    sku VARCHAR(50) DEFAULT ('SKU-' || nextval('sku_sequence')),
    price DECIMAL(10,2) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Generate slug from name (would need trigger for this)
    slug VARCHAR(255)
);
```

Real-World Constraint Examples

Example 1: E-commerce Product Catalog

```
CREATE TABLE products (
    -- Primary key
    product_id SERIAL PRIMARY KEY,

    -- Required fields with constraints
    sku VARCHAR(50) NOT NULL UNIQUE,
    product_name VARCHAR(200) NOT NULL,

    -- Price validation
    price DECIMAL(10,2) NOT NULL CHECK (price > 0),
    cost_price DECIMAL(10,2) CHECK (cost_price IS NULL OR cost_price >= 0),

    -- Stock management
    stock_quantity INT NOT NULL DEFAULT 0 CHECK (stock_quantity >= 0),
    min_stock_level INT DEFAULT 0 CHECK (min_stock_level >= 0),
    max_stock_level INT CHECK (max_stock_level IS NULL OR max_stock_level >=
min_stock_level),

    -- Category relationship
    category_id INT NOT NULL,
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
        ON DELETE RESTRICT,

    -- Product attributes
    weight DECIMAL(8,3) CHECK (weight IS NULL OR weight > 0),
    dimensions VARCHAR(50),

    -- Status and flags
    status VARCHAR(20) DEFAULT 'active'
        CHECK (status IN ('active', 'inactive', 'discontinued')),
    is_featured BOOLEAN DEFAULT FALSE,
    is_digital BOOLEAN DEFAULT FALSE,
```

```
    -- Audit fields
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    created_by INT,
    FOREIGN KEY (created_by) REFERENCES users(user_id),

    -- Business rules
    CHECK (cost_price IS NULL OR cost_price <= price),
    CHECK (NOT (is_digital = TRUE AND weight IS NOT NULL))
);
```

Example 2: User Management System

```
CREATE TABLE users (
    -- Primary key
    user_id SERIAL PRIMARY KEY,

    -- Authentication (required, unique)
    email VARCHAR(255) NOT NULL UNIQUE
        CHECK (email ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'),
    username VARCHAR(50) NOT NULL UNIQUE
        CHECK (LENGTH(username) >= 3 AND username ~ '^[a-zA-Z0-9_]+$'),
    password_hash VARCHAR(255) NOT NULL,

    -- Profile information
    first_name VARCHAR(100) NOT NULL CHECK (LENGTH(TRIM(first_name)) > 0),
    last_name VARCHAR(100) NOT NULL CHECK (LENGTH(TRIM(last_name)) > 0),
    date_of_birth DATE CHECK (date_of_birth < CURRENT_DATE),
    phone VARCHAR(20) CHECK (phone ~ '^\+?[1-9]\d{1,14}$'),

    -- Account settings
    role VARCHAR(20) DEFAULT 'user'
        CHECK (role IN ('admin', 'moderator', 'user', 'guest')),
    status VARCHAR(20) DEFAULT 'active'
        CHECK (status IN ('active', 'inactive', 'suspended', 'deleted')),

    -- Verification and security
    email_verified BOOLEAN DEFAULT FALSE,
    phone_verified BOOLEAN DEFAULT FALSE,
    two_factor_enabled BOOLEAN DEFAULT FALSE,

    -- Audit trail
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,
    login_count INT DEFAULT 0 CHECK (login_count >= 0),

    -- Age validation (must be at least 13)
    CHECK (date_of_birth IS NULL OR
           (CURRENT_DATE - date_of_birth) >= INTERVAL '13 years')
);
```

Example 3: Financial Transactions

```
CREATE TABLE transactions (
    -- Primary key
    transaction_id SERIAL PRIMARY KEY,

    -- Transaction reference
    reference_number VARCHAR(50) NOT NULL UNIQUE,

    -- Account relationships
    from_account_id BIGINT,
    to_account_id BIGINT,
    FOREIGN KEY (from_account_id) REFERENCES accounts(account_id),
    FOREIGN KEY (to_account_id) REFERENCES accounts(account_id),

    -- Amount and currency
    amount DECIMAL(15,2) NOT NULL CHECK (amount > 0),
    currency CHAR(3) NOT NULL DEFAULT 'USD'
        CHECK (currency ~ '^[A-Z]{3}$'),

    -- Transaction details
    transaction_type VARCHAR(20) NOT NULL
        CHECK (transaction_type IN ('deposit', 'withdrawal', 'transfer', 'fee',
'interest')),
    status VARCHAR(20) DEFAULT 'pending'
        CHECK (status IN ('pending', 'processing', 'completed', 'failed',
'cancelled')),

    -- Descriptions and metadata
    description VARCHAR(255),
    internal_notes TEXT,

    -- Timestamps
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    processed_at TIMESTAMP,

    -- Business rules
    CHECK (
        -- Transfer must have both accounts
        (transaction_type = 'transfer' AND from_account_id IS NOT NULL AND
to_account_id IS NOT NULL) OR
        -- Deposit must have to_account
        (transaction_type = 'deposit' AND to_account_id IS NOT NULL) OR
        -- Withdrawal must have from_account
        (transaction_type = 'withdrawal' AND from_account_id IS NOT NULL) OR
        -- Fee can have either or both
        (transaction_type IN ('fee', 'interest'))
    ),

    -- Cannot transfer to same account
    CHECK (from_account_id IS NULL OR to_account_id IS NULL OR from_account_id !=
```

```
to_account_id),

    -- Processed timestamp only for completed transactions
    CHECK (status != 'completed' OR processed_at IS NOT NULL)
);
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between PRIMARY KEY and UNIQUE?"**
   - PRIMARY KEY: One per table, cannot be NULL, creates clustered index
   - UNIQUE: Multiple per table, can have one NULL, creates non-clustered index
   - Both ensure uniqueness, but PRIMARY KEY has additional restrictions

2. **"When would you use CASCADE vs RESTRICT for foreign keys?"**
   - CASCADE: When child records should be deleted with parent (order_items with orders)
   - RESTRICT: When you want to prevent accidental deletion (customers with orders)
   - SET NULL: When relationship is optional (manager_id when manager is deleted)

3. **"How do you enforce business rules in the database?"**
   - CHECK constraints for validation rules
   - FOREIGN KEY constraints for referential integrity
   - UNIQUE constraints for business uniqueness
   - NOT NULL for required fields
   - Triggers for complex logic

Best Practices:

1. **Name Your Constraints:**

```
-- Good: Named constraints are easier to manage
CONSTRAINT pk_customers PRIMARY KEY (customer_id),
CONSTRAINT fk_orders_customer FOREIGN KEY (customer_id) REFERENCES
customers(customer_id),
CONSTRAINT uk_customers_email UNIQUE (email),
CONSTRAINT chk_products_price CHECK (price > 0)
```

2. **Use Appropriate Data Types:**

```
-- Good: Specific types with constraints
email VARCHAR(255) NOT NULL CHECK (email LIKE '%@%'),
status VARCHAR(20) CHECK (status IN ('active', 'inactive')),
```

```
-- Avoid: Generic types without validation
email TEXT,
status TEXT
```

3. **Document Business Rules:**

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    total_amount DECIMAL(10,2) NOT NULL,

    -- Business rule: Orders must be at least $1
    CONSTRAINT chk_orders_min_amount CHECK (total_amount >= 1.00)
);
```

⚠ Things to Watch Out For

1. **Constraint Naming Conflicts**

```
-- Bad: Generic constraint names
CREATE TABLE users (
    id INT PRIMARY KEY,  -- Creates generic name
    email VARCHAR(255) UNIQUE  -- Creates generic name
);

-- Good: Explicit constraint names
CREATE TABLE users (
    id INT,
    email VARCHAR(255),
    CONSTRAINT pk_users PRIMARY KEY (id),
    CONSTRAINT uk_users_email UNIQUE (email)
);
```

2. **Foreign Key Performance**

```
-- Ensure foreign key columns are indexed
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Add index for better performance
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
```

3. **CHECK Constraint Limitations**

```
-- MySQL: CHECK constraints ignored in older versions
-- Always test your constraints!

-- PostgreSQL: Cannot reference other tables in CHECK
-- This won't work:
CHECK (category_id IN (SELECT category_id FROM categories))

-- Use foreign key instead:
FOREIGN KEY (category_id) REFERENCES categories(category_id)
```

4. **Circular Foreign Key References**

```
-- Problem: Circular dependency
CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);

CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Solution: Create tables first, add constraints later
CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    manager_id INT
);

CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    dept_id INT
);

-- Add foreign keys after both tables exist
ALTER TABLE departments
ADD FOREIGN KEY (manager_id) REFERENCES employees(employee_id);

ALTER TABLE employees
ADD FOREIGN KEY (dept_id) REFERENCES departments(dept_id);
```

5. **Constraint Violation Handling**

```
-- Handle constraint violations gracefully in application code

-- PostgreSQL: Use ON CONFLICT
INSERT INTO customers (email, first_name, last_name)
VALUES ('john@email.com', 'John', 'Doe')
ON CONFLICT (email) DO UPDATE SET
    first_name = EXCLUDED.first_name,
    last_name = EXCLUDED.last_name;

-- MySQL: Use ON DUPLICATE KEY UPDATE
INSERT INTO customers (email, first_name, last_name)
VALUES ('john@email.com', 'John', 'Doe')
ON DUPLICATE KEY UPDATE
    first_name = VALUES(first_name),
    last_name = VALUES(last_name);
```

🚀 Next Steps

In the next chapter, we'll explore **Joins** - one of the most powerful features of SQL that allows you to combine data from multiple tables. You'll learn about INNER, LEFT, RIGHT, and FULL joins, and how to write complex queries that span multiple tables.

📝 Quick Practice

Design constraints for a social media platform:

1. **Users table**: Email must be unique and valid format, username 3-50 characters, age must be 13+
2. **Posts table**: Content cannot be empty, status must be 'draft', 'published', or 'deleted'
3. **Followers table**: User cannot follow themselves, combination of follower_id + following_id must be unique
4. **Comments table**: Must reference valid post and user, content length 1-1000 characters

Consider:

- What happens when a user is deleted?
- How do you prevent spam or invalid data?
- What business rules need to be enforced?

📓 Chapter 6: WHERE, GROUP BY, HAVING, ORDER BY

🎯 What You'll Learn

- Filtering data with WHERE clauses
- Grouping data with GROUP BY
- Filtering groups with HAVING
- Sorting results with ORDER BY

- Combining these clauses effectively
- Performance considerations and best practices

📖 Concept Explanation

These four clauses are the backbone of data retrieval in SQL. They work together to filter, group, aggregate, and sort your data:

- **WHERE**: Filters individual rows before grouping
- **GROUP BY**: Groups rows that share common values
- **HAVING**: Filters groups after aggregation
- **ORDER BY**: Sorts the final result set

SQL Query Execution Order:

```
1. FROM       - Choose tables
2. WHERE      - Filter rows
3. GROUP BY   - Group rows
4. HAVING     - Filter groups
5. SELECT     - Choose columns
6. ORDER BY   - Sort results
7. LIMIT      - Limit results
```

Understanding this order is crucial for writing effective queries!

🔍 WHERE Clause - Filtering Rows

Basic WHERE Syntax

```sql
SELECT columns
FROM table
WHERE condition;
```

Comparison Operators

1. Basic Comparisons:

```sql
-- Equality and inequality
SELECT * FROM products WHERE price = 29.99;
SELECT * FROM products WHERE price != 29.99;  -- or <>
SELECT * FROM products WHERE price > 50;
SELECT * FROM products WHERE price <= 100;
SELECT * FROM products WHERE stock_quantity >= 10;

-- Date comparisons
```

```sql
SELECT * FROM orders WHERE order_date = '2024-01-15';
SELECT * FROM orders WHERE order_date > '2024-01-01';
SELECT * FROM employees WHERE hire_date < '2020-01-01';
```

2. Range Conditions:

```sql
-- BETWEEN (inclusive)
SELECT * FROM products
WHERE price BETWEEN 10 AND 50;

-- Equivalent to:
SELECT * FROM products
WHERE price >= 10 AND price <= 50;

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

-- NOT BETWEEN
SELECT * FROM products
WHERE price NOT BETWEEN 10 AND 50;
```

3. List Conditions:

```sql
-- IN operator
SELECT * FROM products
WHERE category_id IN (1, 3, 5, 7);

SELECT * FROM orders
WHERE status IN ('pending', 'processing', 'shipped');

-- NOT IN
SELECT * FROM products
WHERE category_id NOT IN (2, 4, 6);

-- IN with subquery
SELECT * FROM products
WHERE category_id IN (
    SELECT category_id
    FROM categories
    WHERE category_name LIKE '%Electronics%'
);
```

4. Pattern Matching:

```sql
-- LIKE with wildcards
SELECT * FROM customers
```

```sql
WHERE first_name LIKE 'John%';     -- Starts with 'John'

SELECT * FROM customers
WHERE email LIKE '%@gmail.com';    -- Ends with '@gmail.com'

SELECT * FROM products
WHERE product_name LIKE '%phone%'; -- Contains 'phone'

SELECT * FROM customers
WHERE phone LIKE '555-____';       -- _ matches single character

-- Case-insensitive search (PostgreSQL)
SELECT * FROM customers
WHERE first_name ILIKE 'john%';

-- Regular expressions (PostgreSQL)
SELECT * FROM customers
WHERE email ~ '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$';

-- MySQL regular expressions
SELECT * FROM customers
WHERE email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$';
```

5. NULL Handling:

```sql
-- Check for NULL values
SELECT * FROM customers WHERE phone IS NULL;
SELECT * FROM customers WHERE phone IS NOT NULL;

-- NULL in comparisons (always returns NULL/false)
SELECT * FROM products WHERE description = NULL;     -- Wrong! Returns no rows
SELECT * FROM products WHERE description IS NULL;    -- Correct!

-- COALESCE for NULL handling
SELECT customer_id,
       COALESCE(phone, 'No phone') as contact_phone
FROM customers;
```

Logical Operators

1. AND, OR, NOT:

```sql
-- AND: All conditions must be true
SELECT * FROM products
WHERE price > 10 AND price < 100 AND stock_quantity > 0;

-- OR: At least one condition must be true
SELECT * FROM orders
WHERE status = 'shipped' OR status = 'delivered';
```

```sql
-- NOT: Negates the condition
SELECT * FROM customers
WHERE NOT (first_name = 'John' AND last_name = 'Doe');

-- Complex combinations with parentheses
SELECT * FROM products
WHERE (category_id = 1 OR category_id = 2)
  AND price > 50
  AND stock_quantity > 0;
```

2. Operator Precedence:

```sql
-- Without parentheses (AND has higher precedence)
SELECT * FROM products
WHERE category_id = 1 OR category_id = 2 AND price > 50;
-- Equivalent to: category_id = 1 OR (category_id = 2 AND price > 50)

-- With parentheses (clearer intent)
SELECT * FROM products
WHERE (category_id = 1 OR category_id = 2) AND price > 50;
```

Real-World WHERE Examples

1. E-commerce Product Search:

```sql
-- Find available products in specific categories with good ratings
SELECT product_id, product_name, price, average_rating
FROM products
WHERE category_id IN (1, 3, 5)           -- Electronics, Books, Clothing
  AND stock_quantity > 0                  -- In stock
  AND price BETWEEN 10 AND 500           -- Price range
  AND average_rating >= 4.0              -- Good ratings
  AND status = 'active'                  -- Active products
  AND (discount_percentage > 0 OR is_featured = true); -- On sale or featured
```

2. Customer Segmentation:

```sql
-- Find high-value customers for marketing campaign
SELECT customer_id, email, first_name, last_name, total_spent
FROM customers
WHERE total_spent > 1000                 -- High spenders
  AND last_order_date > CURRENT_DATE - INTERVAL '90 days'  -- Recent activity
  AND email_verified = true              -- Verified email
  AND marketing_consent = true           -- Opted in for marketing
  AND status = 'active'                  -- Active account
  AND country IN ('US', 'CA', 'UK');     -- Specific regions
```

3. Order Analysis:

```sql
-- Find problematic orders that need attention
SELECT order_id, customer_id, order_date, total_amount, status
FROM orders
WHERE (
    -- Old pending orders
    (status = 'pending' AND order_date < CURRENT_DATE - INTERVAL '7 days')
    OR
    -- High-value failed orders
    (status = 'failed' AND total_amount > 500)
    OR
    -- Shipped orders without tracking
    (status = 'shipped' AND tracking_number IS NULL)
  )
  AND order_date >= CURRENT_DATE - INTERVAL '30 days'; -- Last 30 days only
```

📊 GROUP BY Clause - Grouping Data

Basic GROUP BY Syntax

```sql
SELECT column1, aggregate_function(column2)
FROM table
WHERE condition
GROUP BY column1;
```

Simple Grouping

1. Basic Grouping:

```sql
-- Count customers by country
SELECT country, COUNT(*) as customer_count
FROM customers
GROUP BY country;

-- Average order value by customer
SELECT customer_id, AVG(total_amount) as avg_order_value
FROM orders
GROUP BY customer_id;

-- Total sales by product category
SELECT c.category_name, SUM(oi.quantity * oi.unit_price) as total_sales
FROM categories c
JOIN products p ON c.category_id = p.category_id
```

```
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY c.category_id, c.category_name;
```

2. Multiple Column Grouping:

```
-- Sales by category and month
SELECT
    c.category_name,
    EXTRACT(YEAR FROM o.order_date) as year,
    EXTRACT(MONTH FROM o.order_date) as month,
    SUM(oi.quantity * oi.unit_price) as monthly_sales,
    COUNT(DISTINCT o.order_id) as order_count
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
GROUP BY c.category_name, EXTRACT(YEAR FROM o.order_date), EXTRACT(MONTH FROM
o.order_date)
ORDER BY year, month, category_name;

-- Customer behavior by country and age group
SELECT
    country,
    CASE
        WHEN EXTRACT(YEAR FROM age(date_of_birth)) < 25 THEN '18-24'
        WHEN EXTRACT(YEAR FROM age(date_of_birth)) < 35 THEN '25-34'
        WHEN EXTRACT(YEAR FROM age(date_of_birth)) < 45 THEN '35-44'
        WHEN EXTRACT(YEAR FROM age(date_of_birth)) < 55 THEN '45-54'
        ELSE '55+'
    END as age_group,
    COUNT(*) as customer_count,
    AVG(total_spent) as avg_spent
FROM customers
WHERE date_of_birth IS NOT NULL
GROUP BY country, age_group
ORDER BY country, age_group;
```

Advanced Grouping

1. Grouping with Expressions:

```
-- Sales by quarter
SELECT
    EXTRACT(YEAR FROM order_date) as year,
    EXTRACT(QUARTER FROM order_date) as quarter,
    COUNT(*) as order_count,
    SUM(total_amount) as total_sales,
    AVG(total_amount) as avg_order_value
FROM orders
```

```sql
WHERE order_date >= '2024-01-01'
GROUP BY EXTRACT(YEAR FROM order_date), EXTRACT(QUARTER FROM order_date)
ORDER BY year, quarter;

-- Product performance by price range
SELECT
    CASE
        WHEN price < 25 THEN 'Budget ($0-$24)'
        WHEN price < 100 THEN 'Mid-range ($25-$99)'
        WHEN price < 500 THEN 'Premium ($100-$499)'
        ELSE 'Luxury ($500+)'
    END as price_category,
    COUNT(*) as product_count,
    AVG(average_rating) as avg_rating,
    SUM(units_sold) as total_units_sold
FROM products
GROUP BY price_category
ORDER BY MIN(price);
```

2. Grouping Sets (PostgreSQL):

```sql
-- Multiple grouping levels in one query
SELECT
    category_id,
    EXTRACT(YEAR FROM order_date) as year,
    SUM(total_amount) as total_sales
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
GROUP BY GROUPING SETS (
    (category_id, EXTRACT(YEAR FROM order_date)),  -- By category and year
    (category_id),                                  -- By category only
    (EXTRACT(YEAR FROM order_date)),               -- By year only
    ()                                             -- Grand total
)
ORDER BY category_id, year;
```

3. ROLLUP and CUBE (PostgreSQL):

```sql
-- ROLLUP: Hierarchical subtotals
SELECT
    country,
    state,
    city,
    COUNT(*) as customer_count
FROM customers
GROUP BY ROLLUP(country, state, city)
ORDER BY country, state, city;
```

```sql
-- CUBE: All possible combinations
SELECT
    category_id,
    EXTRACT(YEAR FROM order_date) as year,
    EXTRACT(QUARTER FROM order_date) as quarter,
    SUM(total_amount) as total_sales
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
GROUP BY CUBE(category_id, year, quarter)
ORDER BY category_id, year, quarter;
```

🎯 HAVING Clause - Filtering Groups

Basic HAVING Syntax

```sql
SELECT column1, aggregate_function(column2)
FROM table
WHERE condition
GROUP BY column1
HAVING aggregate_condition;
```

HAVING vs WHERE

- **WHERE**: Filters rows before grouping
- **HAVING**: Filters groups after aggregation

HAVING Examples

1. Basic HAVING:

```sql
-- Customers with more than 5 orders
SELECT customer_id, COUNT(*) as order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 5;

-- Categories with average product price > $50
SELECT category_id, AVG(price) as avg_price
FROM products
GROUP BY category_id
HAVING AVG(price) > 50;

-- Countries with total sales > $10,000
SELECT
    c.country,
    SUM(o.total_amount) as total_sales,
    COUNT(DISTINCT c.customer_id) as customer_count
```

```
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.country
HAVING SUM(o.total_amount) > 10000;
```

2. Complex HAVING Conditions:

```
-- High-value customer segments
SELECT
    customer_id,
    COUNT(*) as order_count,
    SUM(total_amount) as total_spent,
    AVG(total_amount) as avg_order_value
FROM orders
WHERE order_date >= CURRENT_DATE - INTERVAL '1 year'
GROUP BY customer_id
HAVING COUNT(*) >= 3                    -- At least 3 orders
   AND SUM(total_amount) > 500          -- Spent more than $500
   AND AVG(total_amount) > 50;          -- Average order > $50

-- Product categories with consistent sales
SELECT
    c.category_name,
    COUNT(DISTINCT EXTRACT(MONTH FROM o.order_date)) as active_months,
    SUM(oi.quantity * oi.unit_price) as total_sales,
    STDDEV(monthly_sales.sales) as sales_variance
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
JOIN (
    SELECT
        p2.category_id,
        EXTRACT(MONTH FROM o2.order_date) as month,
        SUM(oi2.quantity * oi2.unit_price) as sales
    FROM products p2
    JOIN order_items oi2 ON p2.product_id = oi2.product_id
    JOIN orders o2 ON oi2.order_id = o2.order_id
    GROUP BY p2.category_id, EXTRACT(MONTH FROM o2.order_date)
) monthly_sales ON c.category_id = monthly_sales.category_id
WHERE o.order_date >= CURRENT_DATE - INTERVAL '1 year'
GROUP BY c.category_id, c.category_name
HAVING COUNT(DISTINCT EXTRACT(MONTH FROM o.order_date)) >= 6  -- Active in 6+
months
   AND STDDEV(monthly_sales.sales) < 1000;                    -- Low variance
```

3. HAVING with Subqueries:

```sql
-- Categories performing above average
SELECT
    c.category_name,
    SUM(oi.quantity * oi.unit_price) as category_sales
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY c.category_id, c.category_name
HAVING SUM(oi.quantity * oi.unit_price) > (
    SELECT AVG(category_total)
    FROM (
        SELECT SUM(oi2.quantity * oi2.unit_price) as category_total
        FROM categories c2
        JOIN products p2 ON c2.category_id = p2.category_id
        JOIN order_items oi2 ON p2.product_id = oi2.product_id
        GROUP BY c2.category_id
    ) avg_calc
);
```

☑ ORDER BY Clause - Sorting Results

Basic ORDER BY Syntax

```sql
SELECT columns
FROM table
WHERE condition
GROUP BY columns
HAVING condition
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

Sorting Options

1. Basic Sorting:

```sql
-- Single column sorting
SELECT * FROM products ORDER BY price;           -- Ascending (default)
SELECT * FROM products ORDER BY price ASC;      -- Explicit ascending
SELECT * FROM products ORDER BY price DESC;     -- Descending

-- Multiple column sorting
SELECT * FROM customers
ORDER BY country ASC, last_name ASC, first_name ASC;

-- Mixed sorting directions
SELECT product_name, price, average_rating
FROM products
ORDER BY average_rating DESC, price ASC;
```

2. Sorting by Expressions:

```
-- Sort by calculated values
SELECT
    product_name,
    price,
    stock_quantity,
    (price * stock_quantity) as inventory_value
FROM products
ORDER BY (price * stock_quantity) DESC;

-- Sort by string functions
SELECT first_name, last_name, email
FROM customers
ORDER BY LENGTH(email) DESC, email ASC;

-- Sort by date parts
SELECT order_id, order_date, total_amount
FROM orders
ORDER BY EXTRACT(MONTH FROM order_date), EXTRACT(DAY FROM order_date);
```

3. Conditional Sorting:

```
-- Custom sort order with CASE
SELECT order_id, status, order_date
FROM orders
ORDER BY
    CASE status
        WHEN 'urgent' THEN 1
        WHEN 'processing' THEN 2
        WHEN 'pending' THEN 3
        WHEN 'shipped' THEN 4
        WHEN 'delivered' THEN 5
        ELSE 6
    END,
    order_date DESC;

-- Sort products by availability, then by rating
SELECT product_name, stock_quantity, average_rating
FROM products
ORDER BY
    CASE WHEN stock_quantity > 0 THEN 0 ELSE 1 END,  -- In stock first
    average_rating DESC,
    product_name ASC;
```

4. NULL Handling in Sorting:

```sql
-- PostgreSQL: Control NULL position
SELECT customer_id, phone
FROM customers
ORDER BY phone NULLS LAST;   -- NULLs at the end

SELECT customer_id, phone
FROM customers
ORDER BY phone NULLS FIRST;  -- NULLs at the beginning

-- MySQL: Use ISNULL() or COALESCE()
SELECT customer_id, phone
FROM customers
ORDER BY ISNULL(phone), phone;  -- NULLs first

SELECT customer_id, phone
FROM customers
ORDER BY COALESCE(phone, 'ZZZZZ');  -- Treat NULL as 'ZZZZZ'
```

Advanced Sorting

1. Sorting Aggregated Results:

```sql
-- Top customers by total spending
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count,
    SUM(o.total_amount) as total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_spent DESC NULLS LAST, order_count DESC;

-- Best selling products by category
SELECT
    c.category_name,
    p.product_name,
    SUM(oi.quantity) as total_sold,
    SUM(oi.quantity * oi.unit_price) as total_revenue
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY c.category_id, c.category_name, p.product_id, p.product_name
ORDER BY c.category_name, total_revenue DESC;
```

2. Window Function Sorting:

```sql
-- Rank products within each category by sales
SELECT
    c.category_name,
    p.product_name,
    SUM(oi.quantity * oi.unit_price) as revenue,
    RANK() OVER (
        PARTITION BY c.category_id
        ORDER BY SUM(oi.quantity * oi.unit_price) DESC
    ) as category_rank
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY c.category_id, c.category_name, p.product_id, p.product_name
ORDER BY c.category_name, category_rank;
```

🔄 Combining All Clauses

Complete Query Structure

```sql
SELECT
    -- What to show
    c.category_name,
    COUNT(DISTINCT p.product_id) as product_count,
    COUNT(DISTINCT o.order_id) as order_count,
    SUM(oi.quantity) as total_units_sold,
    SUM(oi.quantity * oi.unit_price) as total_revenue,
    AVG(oi.unit_price) as avg_unit_price,
    MAX(o.order_date) as last_order_date
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE
    -- Filter individual rows
    o.order_date >= CURRENT_DATE - INTERVAL '1 year'  -- Last year only
    AND o.status IN ('completed', 'shipped', 'delivered')  -- Successful orders
    AND p.status = 'active'                            -- Active products
GROUP BY
    -- Group by category
    c.category_id, c.category_name
HAVING
    -- Filter groups
    COUNT(DISTINCT o.order_id) >= 10                   -- At least 10 orders
    AND SUM(oi.quantity * oi.unit_price) > 1000      -- Revenue > $1000
ORDER BY
    -- Sort results
    total_revenue DESC,                                -- Highest revenue first
    product_count DESC;                                -- Then by product count
```

Real-World Complex Examples

1. Sales Performance Dashboard:

```sql
-- Monthly sales performance with growth metrics
SELECT
    EXTRACT(YEAR FROM o.order_date) as year,
    EXTRACT(MONTH FROM o.order_date) as month,
    COUNT(DISTINCT o.order_id) as order_count,
    COUNT(DISTINCT o.customer_id) as unique_customers,
    SUM(o.total_amount) as total_revenue,
    AVG(o.total_amount) as avg_order_value,
    SUM(CASE WHEN c.created_at >= o.order_date - INTERVAL '30 days'
             THEN o.total_amount ELSE 0 END) as new_customer_revenue,
    COUNT(CASE WHEN o.total_amount > 100 THEN 1 END) as high_value_orders
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE
    o.order_date >= CURRENT_DATE - INTERVAL '2 years'
    AND o.status IN ('completed', 'shipped', 'delivered')
    AND o.total_amount > 0
GROUP BY
    EXTRACT(YEAR FROM o.order_date),
    EXTRACT(MONTH FROM o.order_date)
HAVING
    COUNT(DISTINCT o.order_id) >= 5  -- Months with at least 5 orders
ORDER BY
    year DESC, month DESC;
```

2. Customer Segmentation Analysis:

```sql
-- RFM Analysis: Recency, Frequency, Monetary
SELECT
    customer_id,
    -- Recency: Days since last order
    CURRENT_DATE - MAX(order_date) as days_since_last_order,

    -- Frequency: Number of orders
    COUNT(*) as order_frequency,

    -- Monetary: Total spent
    SUM(total_amount) as total_monetary_value,
    AVG(total_amount) as avg_order_value,

    -- Customer segment based on behavior
    CASE
        WHEN COUNT(*) >= 10 AND SUM(total_amount) >= 1000
             AND CURRENT_DATE - MAX(order_date) <= 30
        THEN 'VIP'
```

```
        WHEN COUNT(*) >= 5 AND SUM(total_amount) >= 500
             AND CURRENT_DATE - MAX(order_date) <= 90
        THEN 'Loyal'

        WHEN COUNT(*) >= 3 AND CURRENT_DATE - MAX(order_date) <= 180
        THEN 'Regular'

        WHEN CURRENT_DATE - MAX(order_date) <= 365
        THEN 'Occasional'

        ELSE 'Inactive'
    END as customer_segment
FROM orders
WHERE
    order_date >= CURRENT_DATE - INTERVAL '2 years'
    AND status IN ('completed', 'delivered')
GROUP BY customer_id
HAVING
    COUNT(*) >= 1  -- At least one order
ORDER BY
    total_monetary_value DESC,
    order_frequency DESC,
    days_since_last_order ASC;
```

3. Product Performance Analysis:

```
-- Product performance with seasonal trends
SELECT
    p.product_id,
    p.product_name,
    c.category_name,

    -- Overall metrics
    COUNT(DISTINCT oi.order_id) as order_count,
    SUM(oi.quantity) as total_units_sold,
    SUM(oi.quantity * oi.unit_price) as total_revenue,
    AVG(oi.unit_price) as avg_selling_price,

    -- Seasonal performance
    SUM(CASE WHEN EXTRACT(QUARTER FROM o.order_date) = 1
             THEN oi.quantity ELSE 0 END) as q1_units,
    SUM(CASE WHEN EXTRACT(QUARTER FROM o.order_date) = 2
             THEN oi.quantity ELSE 0 END) as q2_units,
    SUM(CASE WHEN EXTRACT(QUARTER FROM o.order_date) = 3
             THEN oi.quantity ELSE 0 END) as q3_units,
    SUM(CASE WHEN EXTRACT(QUARTER FROM o.order_date) = 4
             THEN oi.quantity ELSE 0 END) as q4_units,

    -- Performance indicators
    CASE
        WHEN SUM(oi.quantity * oi.unit_price) > 10000 THEN 'Top Performer'
        WHEN SUM(oi.quantity * oi.unit_price) > 5000 THEN 'Good Performer'
```

```
        WHEN SUM(oi.quantity * oi.unit_price) > 1000 THEN 'Average Performer'
        ELSE 'Poor Performer'
    END as performance_category
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE
    o.order_date >= CURRENT_DATE - INTERVAL '1 year'
    AND o.status IN ('completed', 'delivered')
    AND p.status = 'active'
GROUP BY
    p.product_id, p.product_name, c.category_name
HAVING
    COUNT(DISTINCT oi.order_id) >= 5      -- At least 5 orders
    AND SUM(oi.quantity) >= 10            -- At least 10 units sold
ORDER BY
    total_revenue DESC,
    total_units_sold DESC;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between WHERE and HAVING?"**
   - WHERE filters rows before grouping
   - HAVING filters groups after aggregation
   - WHERE cannot use aggregate functions, HAVING can
   - WHERE is processed before GROUP BY, HAVING after

2. **"Explain the SQL execution order"**
   - FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
   - Understanding this helps write correct queries and debug issues

3. **"How do you optimize queries with these clauses?"**
   - Use indexes on WHERE clause columns
   - Use indexes on GROUP BY columns
   - Use indexes on ORDER BY columns
   - Filter early with WHERE to reduce data for grouping
   - Use LIMIT when you don't need all results

Performance Best Practices:

1. **Index Strategy:**

```
-- Create composite indexes for common query patterns
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
```

```
CREATE INDEX idx_products_category_price ON products(category_id, price);
CREATE INDEX idx_orders_status_date ON orders(status, order_date);
```

2. **Query Optimization:**

```
-- Good: Filter early with WHERE
SELECT category_id, COUNT(*)
FROM products
WHERE status = 'active'  -- Filter first
GROUP BY category_id;

-- Avoid: Filtering after grouping when possible
SELECT category_id, COUNT(*)
FROM products
GROUP BY category_id
HAVING SUM(CASE WHEN status = 'active' THEN 1 ELSE 0 END) > 0;
```

3. **Efficient Aggregation:**

```
-- Use specific aggregate functions
SELECT COUNT(*) FROM orders;              -- Count all rows
SELECT COUNT(customer_id) FROM orders;    -- Count non-NULL values
SELECT COUNT(DISTINCT customer_id) FROM orders;  -- Count unique values
```

⚠ Things to Watch Out For

1. **GROUP BY Rules**

```
-- Error: Non-aggregated column in SELECT without GROUP BY
SELECT customer_id, order_date, COUNT(*)
FROM orders
GROUP BY customer_id;  -- order_date not in GROUP BY!

-- Correct: Include all non-aggregated columns in GROUP BY
SELECT customer_id, order_date, COUNT(*)
FROM orders
GROUP BY customer_id, order_date;

-- Or use aggregation for the column
SELECT customer_id, MAX(order_date) as latest_order, COUNT(*)
FROM orders
GROUP BY customer_id;
```

2. **NULL Handling in Aggregates**

```sql
-- COUNT(*) counts all rows, COUNT(column) ignores NULLs
SELECT
    COUNT(*) as total_customers,           -- All customers
    COUNT(phone) as customers_with_phone,  -- Only non-NULL phones
    COUNT(DISTINCT country) as unique_countries
FROM customers;

-- SUM, AVG ignore NULLs
SELECT
    AVG(rating) as avg_rating,                    -- Ignores NULL ratings
    AVG(COALESCE(rating, 0)) as avg_rating_with_zero  -- Treats NULL as 0
FROM product_reviews;
```

3. **Date/Time Grouping Pitfalls**

```sql
-- Problem: Grouping by full timestamp
SELECT order_date, COUNT(*)
FROM orders
GROUP BY order_date;  -- Groups by exact timestamp!

-- Solution: Extract date part
SELECT DATE(order_date) as order_day, COUNT(*)
FROM orders
GROUP BY DATE(order_date);

-- Or use date truncation (PostgreSQL)
SELECT DATE_TRUNC('day', order_date) as order_day, COUNT(*)
FROM orders
GROUP BY DATE_TRUNC('day', order_date);
```

4. **Performance Issues**

```sql
-- Slow: No indexes on filtered/grouped columns
SELECT category_id, COUNT(*)
FROM products
WHERE price > 100
GROUP BY category_id
ORDER BY COUNT(*) DESC;

-- Fast: With proper indexes
CREATE INDEX idx_products_price_category ON products(price, category_id);
```

5. **Logical Operator Precedence**

```sql
-- Confusing: AND has higher precedence than OR
SELECT * FROM products
WHERE category_id = 1 OR category_id = 2 AND price > 50;
-- Equivalent to: category_id = 1 OR (category_id = 2 AND price > 50)

-- Clear: Use parentheses
SELECT * FROM products
WHERE (category_id = 1 OR category_id = 2) AND price > 50;
```

🚀 Next Steps

In the next chapter, we'll explore **Joins** - the powerful feature that allows you to combine data from multiple tables. You'll learn about INNER, LEFT, RIGHT, and FULL joins, and how to write complex queries that span multiple related tables.

📝 Quick Practice

Try these exercises with a sample e-commerce database:

1. **Basic Filtering**: Find all orders from the last 30 days with a total amount greater than $100
2. **Grouping**: Calculate total sales by month for the current year
3. **Having**: Find customers who have placed more than 3 orders and spent more than $500 total
4. **Complex Query**: Create a report showing the top 5 product categories by revenue, including the number of products and average price per category
5. **Performance**: Identify which indexes would improve the performance of your queries

Consider:

- How do you handle NULL values in your calculations?
- What happens when you group by date columns?
- How do you ensure your queries are efficient?
- What business insights can you derive from the grouped data?

📓 Chapter 7: Joins (INNER, LEFT, RIGHT, FULL)

🎯 What You'll Learn

- Understanding table relationships and join concepts
- INNER JOIN for matching records
- LEFT JOIN for preserving left table records
- RIGHT JOIN for preserving right table records
- FULL OUTER JOIN for all records
- CROSS JOIN for cartesian products
- Self-joins and complex join scenarios
- Join performance optimization

🕮 Concept Explanation

Joins are the heart of relational databases. They allow you to combine data from multiple tables based on relationships between them. Think of joins as "connecting the dots" between related information stored in different tables.

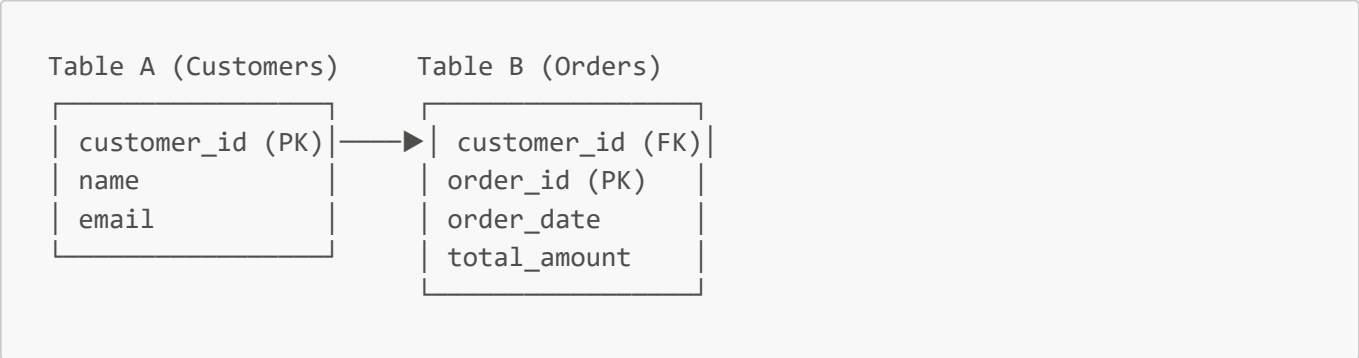
Why Use Joins?

1. **Normalization**: Data is stored in separate tables to avoid redundancy
2. **Relationships**: Tables are connected through foreign keys
3. **Comprehensive Queries**: Get complete information by combining related data
4. **Data Integrity**: Maintain consistency across related tables

Types of Joins:

- **INNER JOIN**: Only matching records from both tables
- **LEFT JOIN**: All records from left table + matching from right
- **RIGHT JOIN**: All records from right table + matching from left
- **FULL OUTER JOIN**: All records from both tables
- **CROSS JOIN**: Cartesian product (every row with every row)
- **SELF JOIN**: Join a table with itself

Visual Join Representation:



🔗 INNER JOIN - Matching Records Only

Basic INNER JOIN Syntax

```sql
SELECT columns
FROM table1
INNER JOIN table2 ON table1.column = table2.column;

-- Alternative syntax (implicit join)
SELECT columns
FROM table1, table2
WHERE table1.column = table2.column;
```

Simple INNER JOIN Examples

1. Customers with Orders:

```sql
-- Get customers who have placed orders
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    o.order_id,
    o.order_date,
    o.total_amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
ORDER BY c.last_name, c.first_name, o.order_date;

-- Count orders per customer (only customers with orders)
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count,
    SUM(o.total_amount) as total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_spent DESC;
```

2. Products with Categories:

```sql
-- Get products with their category information
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name,
    c.description as category_description
FROM products p
INNER JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
ORDER BY c.category_name, p.product_name;

-- Average price by category (only categories with products)
SELECT
    c.category_name,
    COUNT(p.product_id) as product_count,
    AVG(p.price) as avg_price,
    MIN(p.price) as min_price,
    MAX(p.price) as max_price
FROM categories c
INNER JOIN products p ON c.category_id = p.category_id
```

```
WHERE p.status = 'active'
GROUP BY c.category_id, c.category_name
ORDER BY avg_price DESC;
```

Multiple INNER JOINs

1. Three Table Join:

```
-- Get order details with customer and product information
SELECT
    c.first_name,
    c.last_name,
    o.order_id,
    o.order_date,
    p.product_name,
    oi.quantity,
    oi.unit_price,
    (oi.quantity * oi.unit_price) as line_total
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
INNER JOIN order_items oi ON o.order_id = oi.order_id
INNER JOIN products p ON oi.product_id = p.product_id
WHERE o.order_date >= CURRENT_DATE - INTERVAL '30 days'
ORDER BY o.order_date DESC, o.order_id, oi.order_item_id;
```

2. Complex Business Query:

```
-- Sales report: Customer, Order, Product, and Category information
SELECT
    cat.category_name,
    p.product_name,
    c.first_name || ' ' || c.last_name as customer_name,
    c.email,
    o.order_date,
    oi.quantity,
    oi.unit_price,
    (oi.quantity * oi.unit_price) as revenue,

    -- Calculate discount if original price is available
    CASE
        WHEN p.price > oi.unit_price
        THEN ROUND(((p.price - oi.unit_price) / p.price * 100), 2)
        ELSE 0
    END as discount_percentage
FROM categories cat
INNER JOIN products p ON cat.category_id = p.category_id
INNER JOIN order_items oi ON p.product_id = oi.product_id
INNER JOIN orders o ON oi.order_id = o.order_id
INNER JOIN customers c ON o.customer_id = c.customer_id
```

```
WHERE
    o.order_date BETWEEN '2024-01-01' AND '2024-12-31'
    AND o.status IN ('completed', 'shipped', 'delivered')
ORDER BY
    cat.category_name,
    revenue DESC;
```

← LEFT JOIN - Preserve Left Table

Basic LEFT JOIN Syntax

```
SELECT columns
FROM table1
LEFT JOIN table2 ON table1.column = table2.column;

-- Alternative syntax
SELECT columns
FROM table1
LEFT OUTER JOIN table2 ON table1.column = table2.column;
```

LEFT JOIN Examples

1. All Customers (with or without orders):

```
-- Get all customers and their order information (if any)
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    o.order_id,
    o.order_date,
    o.total_amount,

    -- Handle NULL values from right table
    CASE
        WHEN o.order_id IS NULL THEN 'No orders'
        ELSE 'Has orders'
    END as order_status
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
ORDER BY c.last_name, c.first_name, o.order_date;

-- Find customers who have never placed an order
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
```

```
    c.email,
    c.created_at
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.customer_id IS NULL  -- No matching orders
ORDER BY c.created_at DESC;
```

2. Customer Order Summary:

```
-- All customers with their order statistics
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,

    -- Aggregate functions handle NULLs appropriately
    COUNT(o.order_id) as order_count,
    COALESCE(SUM(o.total_amount), 0) as total_spent,
    COALESCE(AVG(o.total_amount), 0) as avg_order_value,
    MAX(o.order_date) as last_order_date,

    -- Customer segmentation
    CASE
        WHEN COUNT(o.order_id) = 0 THEN 'No Orders'
        WHEN COUNT(o.order_id) = 1 THEN 'One-time Customer'
        WHEN COUNT(o.order_id) BETWEEN 2 AND 5 THEN 'Regular Customer'
        WHEN COUNT(o.order_id) > 5 THEN 'VIP Customer'
    END as customer_segment
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
ORDER BY total_spent DESC;
```

3. Product Performance Analysis:

```
-- All products with their sales performance (including unsold products)
SELECT
    p.product_id,
    p.product_name,
    p.price,
    p.stock_quantity,
    c.category_name,

    -- Sales metrics (NULL-safe)
    COALESCE(SUM(oi.quantity), 0) as total_units_sold,
    COALESCE(SUM(oi.quantity * oi.unit_price), 0) as total_revenue,
    COUNT(DISTINCT oi.order_id) as order_count,
```

```
    -- Performance indicators
    CASE
        WHEN SUM(oi.quantity) IS NULL THEN 'Never Sold'
        WHEN SUM(oi.quantity) < 10 THEN 'Low Sales'
        WHEN SUM(oi.quantity) < 50 THEN 'Moderate Sales'
        ELSE 'High Sales'
    END as sales_performance
FROM products p
LEFT JOIN categories c ON p.category_id = c.category_id
LEFT JOIN order_items oi ON p.product_id = oi.product_id
LEFT JOIN orders o ON oi.order_id = o.order_id
    AND o.status IN ('completed', 'shipped', 'delivered')
WHERE p.status = 'active'
GROUP BY
    p.product_id, p.product_name, p.price, p.stock_quantity, c.category_name
ORDER BY total_revenue DESC;
```

⮕ RIGHT JOIN - Preserve Right Table

Basic RIGHT JOIN Syntax

```
SELECT columns
FROM table1
RIGHT JOIN table2 ON table1.column = table2.column;

-- Alternative syntax
SELECT columns
FROM table1
RIGHT OUTER JOIN table2 ON table1.column = table2.column;
```

RIGHT JOIN Examples

1. All Orders (with customer info if available):

```
-- Get all orders, even if customer data is missing
SELECT
    o.order_id,
    o.order_date,
    o.total_amount,
    o.status,

    -- Customer info (might be NULL)
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,

    -- Handle missing customer data
```

```
    CASE
        WHEN c.customer_id IS NULL THEN 'Customer data missing'
        ELSE 'Customer found'
    END as customer_status
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id
ORDER BY o.order_date DESC;

-- Find orders with missing customer references (data integrity check)
SELECT
    o.order_id,
    o.customer_id as orphaned_customer_id,
    o.order_date,
    o.total_amount
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id
WHERE c.customer_id IS NULL;
```

Note: RIGHT JOIN is less commonly used than LEFT JOIN. Most developers prefer to rewrite RIGHT JOINs as LEFT JOINs by switching table order:

```
-- These queries are equivalent:

-- RIGHT JOIN version
SELECT *
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;

-- LEFT JOIN version (preferred)
SELECT *
FROM orders o
LEFT JOIN customers c ON o.customer_id = c.customer_id;
```

🔄 FULL OUTER JOIN - All Records

Basic FULL OUTER JOIN Syntax

```
SELECT columns
FROM table1
FULL OUTER JOIN table2 ON table1.column = table2.column;

-- Alternative syntax
SELECT columns
FROM table1
FULL JOIN table2 ON table1.column = table2.column;
```

Note: MySQL doesn't support FULL OUTER JOIN directly. You can simulate it using UNION:

```
-- MySQL: Simulate FULL OUTER JOIN with UNION
SELECT columns FROM table1 LEFT JOIN table2 ON condition
UNION
SELECT columns FROM table1 RIGHT JOIN table2 ON condition;
```

FULL OUTER JOIN Examples (PostgreSQL)

1. Complete Customer-Order Relationship:

```
-- Get all customers and all orders, showing relationships where they exist
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    o.order_id,
    o.order_date,
    o.total_amount,

    -- Categorize the relationship
    CASE
        WHEN c.customer_id IS NULL THEN 'Order without customer'
        WHEN o.order_id IS NULL THEN 'Customer without orders'
        ELSE 'Customer with orders'
    END as relationship_status
FROM customers c
FULL OUTER JOIN orders o ON c.customer_id = o.customer_id
ORDER BY
    c.customer_id NULLS LAST,
    o.order_date NULLS LAST;
```

2. Data Integrity Analysis:

```
-- Find all mismatches between customers and orders tables
SELECT
    COALESCE(c.customer_id, o.customer_id) as customer_id,
    c.first_name,
    c.last_name,
    o.order_id,
    o.order_date,

    -- Identify data issues
    CASE
        WHEN c.customer_id IS NULL THEN 'ORPHANED ORDER: No customer record'
        WHEN o.order_id IS NULL THEN 'CUSTOMER WITHOUT ORDERS: Potential marketing
target'
        ELSE 'VALID RELATIONSHIP'
    END as data_status
FROM customers c
```

```
FULL OUTER JOIN orders o ON c.customer_id = o.customer_id
WHERE
    c.customer_id IS NULL OR o.order_id IS NULL  -- Only show mismatches
ORDER BY data_status, customer_id;
```

❌ CROSS JOIN - Cartesian Product

Basic CROSS JOIN Syntax

```
SELECT columns
FROM table1
CROSS JOIN table2;

-- Alternative syntax
SELECT columns
FROM table1, table2;
```

CROSS JOIN Examples

1. Product-Size Combinations:

```
-- Generate all possible product-size combinations
SELECT
    p.product_name,
    s.size_name,
    p.base_price,
    s.price_modifier,
    (p.base_price + s.price_modifier) as final_price
FROM products p
CROSS JOIN sizes s
WHERE p.category_id = 1  -- Only for clothing category
ORDER BY p.product_name, s.sort_order;

-- Create a size availability matrix
SELECT
    p.product_name,
    COUNT(CASE WHEN s.size_name = 'XS' THEN 1 END) as xs_available,
    COUNT(CASE WHEN s.size_name = 'S' THEN 1 END) as s_available,
    COUNT(CASE WHEN s.size_name = 'M' THEN 1 END) as m_available,
    COUNT(CASE WHEN s.size_name = 'L' THEN 1 END) as l_available,
    COUNT(CASE WHEN s.size_name = 'XL' THEN 1 END) as xl_available
FROM products p
CROSS JOIN sizes s
WHERE p.category_id = 1
GROUP BY p.product_id, p.product_name;
```

2. Date Range Generation:

```sql
-- Generate sales report template for all months and categories
WITH months AS (
    SELECT generate_series(
        DATE_TRUNC('month', CURRENT_DATE - INTERVAL '11 months'),
        DATE_TRUNC('month', CURRENT_DATE),
        INTERVAL '1 month'
    )::date as month_start
)
SELECT
    m.month_start,
    TO_CHAR(m.month_start, 'YYYY-MM') as month_label,
    c.category_id,
    c.category_name,

    -- Placeholder for actual sales data
    0 as planned_sales,
    NULL as actual_sales
FROM months m
CROSS JOIN categories c
ORDER BY m.month_start, c.category_name;
```

⚠ **Warning**: CROSS JOIN can produce very large result sets. A table with 1,000 rows crossed with another 1,000-row table produces 1,000,000 rows!

🔄 SELF JOIN - Join Table with Itself

Self JOIN Concept

A self join is when you join a table with itself, typically to find relationships within the same table (like hierarchies, comparisons, or sequences).

Self JOIN Examples

1. Employee Hierarchy:

```sql
-- Find employees and their managers
SELECT
    e.employee_id,
    e.first_name || ' ' || e.last_name as employee_name,
    e.job_title,
    m.employee_id as manager_id,
    m.first_name || ' ' || m.last_name as manager_name,
    m.job_title as manager_title
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id
ORDER BY m.employee_id NULLS FIRST, e.last_name;

-- Find all employees who report to a specific manager
SELECT
```

```
    m.first_name || ' ' || m.last_name as manager_name,
    COUNT(e.employee_id) as direct_reports,
    STRING_AGG(e.first_name || ' ' || e.last_name, ', ') as team_members
FROM employees m
LEFT JOIN employees e ON m.employee_id = e.manager_id
GROUP BY m.employee_id, m.first_name, m.last_name
HAVING COUNT(e.employee_id) > 0
ORDER BY direct_reports DESC;
```

2. Product Comparisons:

```
-- Find products in the same category with similar prices
SELECT
    p1.product_name as product_1,
    p1.price as price_1,
    p2.product_name as product_2,
    p2.price as price_2,
    ABS(p1.price - p2.price) as price_difference
FROM products p1
JOIN products p2 ON p1.category_id = p2.category_id
    AND p1.product_id < p2.product_id  -- Avoid duplicates and self-comparison
    AND ABS(p1.price - p2.price) < 10  -- Similar prices (within $10)
WHERE p1.status = 'active' AND p2.status = 'active'
ORDER BY p1.category_id, price_difference;
```

3. Sequential Data Analysis:

```
-- Compare consecutive orders from the same customer
SELECT
    c.first_name || ' ' || c.last_name as customer_name,
    o1.order_id as first_order,
    o1.order_date as first_date,
    o1.total_amount as first_amount,
    o2.order_id as next_order,
    o2.order_date as next_date,
    o2.total_amount as next_amount,

    -- Calculate time between orders
    (o2.order_date - o1.order_date) as days_between,

    -- Calculate spending change
    (o2.total_amount - o1.total_amount) as amount_change,
    ROUND(
        ((o2.total_amount - o1.total_amount) / o1.total_amount * 100), 2
    ) as percent_change
FROM orders o1
JOIN orders o2 ON o1.customer_id = o2.customer_id
    AND o2.order_date > o1.order_date
JOIN customers c ON o1.customer_id = c.customer_id
```

```
WHERE NOT EXISTS (
    -- Ensure o2 is the immediate next order
    SELECT 1 FROM orders o3
    WHERE o3.customer_id = o1.customer_id
    AND o3.order_date > o1.order_date
    AND o3.order_date < o2.order_date
)
ORDER BY c.customer_id, o1.order_date;
```

🔧 Advanced Join Techniques

1. Multiple Join Conditions

```
-- Join with multiple conditions
SELECT
    c.customer_name,
    o.order_date,
    p.product_name,
    oi.quantity
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
    AND o.order_date >= '2024-01-01'  -- Additional condition
JOIN order_items oi ON o.order_id = oi.order_id
    AND oi.quantity > 1  -- Additional condition
JOIN products p ON oi.product_id = p.product_id
    AND p.status = 'active';  -- Additional condition
```

2. Conditional Joins

```
-- Join based on conditions
SELECT
    p.product_name,
    p.price,
    d.discount_percentage,
    CASE
        WHEN d.discount_id IS NOT NULL
        THEN p.price * (1 - d.discount_percentage / 100)
        ELSE p.price
    END as final_price
FROM products p
LEFT JOIN discounts d ON p.product_id = d.product_id
    AND d.start_date <= CURRENT_DATE
    AND d.end_date >= CURRENT_DATE
    AND d.is_active = true;
```

3. Subquery Joins

```sql
-- Join with subquery results
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    recent_orders.order_count,
    recent_orders.total_spent
FROM customers c
JOIN (
    SELECT
        customer_id,
        COUNT(*) as order_count,
        SUM(total_amount) as total_spent
    FROM orders
    WHERE order_date >= CURRENT_DATE - INTERVAL '90 days'
    GROUP BY customer_id
    HAVING COUNT(*) >= 2
) recent_orders ON c.customer_id = recent_orders.customer_id;
```

4. Window Functions with Joins

```sql
-- Rank customers by spending within each country
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.country,
    SUM(o.total_amount) as total_spent,
    RANK() OVER (
        PARTITION BY c.country
        ORDER BY SUM(o.total_amount) DESC
    ) as country_rank,
    PERCENT_RANK() OVER (
        PARTITION BY c.country
        ORDER BY SUM(o.total_amount)
    ) as percentile_rank
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status IN ('completed', 'delivered')
GROUP BY c.customer_id, c.first_name, c.last_name, c.country
ORDER BY c.country, country_rank;
```

Real-World Join Examples

Example 1: E-commerce Analytics Dashboard

```sql
-- Comprehensive sales analytics with multiple joins
SELECT
    -- Time dimensions
    DATE_TRUNC('month', o.order_date) as month,

    -- Product dimensions
    cat.category_name,
    p.product_name,

    -- Customer dimensions
    c.country,
    CASE
        WHEN EXTRACT(YEAR FROM age(c.date_of_birth)) < 25 THEN '18-24'
        WHEN EXTRACT(YEAR FROM age(c.date_of_birth)) < 35 THEN '25-34'
        WHEN EXTRACT(YEAR FROM age(c.date_of_birth)) < 45 THEN '35-44'
        WHEN EXTRACT(YEAR FROM age(c.date_of_birth)) < 55 THEN '45-54'
        ELSE '55+'
    END as age_group,

    -- Metrics
    COUNT(DISTINCT o.order_id) as order_count,
    COUNT(DISTINCT c.customer_id) as unique_customers,
    SUM(oi.quantity) as units_sold,
    SUM(oi.quantity * oi.unit_price) as revenue,
    AVG(oi.unit_price) as avg_unit_price,

    -- Advanced metrics
    SUM(oi.quantity * oi.unit_price) / COUNT(DISTINCT o.order_id) as
avg_order_value,
    SUM(oi.quantity * oi.unit_price) / COUNT(DISTINCT c.customer_id) as
revenue_per_customer
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
JOIN categories cat ON p.category_id = cat.category_id
LEFT JOIN discounts d ON p.product_id = d.product_id
    AND o.order_date BETWEEN d.start_date AND d.end_date
WHERE
    o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    AND o.status IN ('completed', 'shipped', 'delivered')
    AND c.date_of_birth IS NOT NULL
GROUP BY
    DATE_TRUNC('month', o.order_date),
    cat.category_name,
    p.product_name,
    c.country,
    age_group
HAVING
    COUNT(DISTINCT o.order_id) >= 5  -- Significant volume
ORDER BY
    month DESC,
    revenue DESC;
```

Example 2: Customer Lifetime Value Analysis

```sql
-- Calculate customer lifetime value with cohort analysis
WITH customer_cohorts AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
        c.created_at as registration_date,
        DATE_TRUNC('month', c.created_at) as cohort_month,
        MIN(o.order_date) as first_order_date,
        MAX(o.order_date) as last_order_date
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
        AND o.status IN ('completed', 'delivered')
    GROUP BY c.customer_id, c.first_name, c.last_name, c.email, c.created_at
),
customer_metrics AS (
    SELECT
        cc.customer_id,
        cc.first_name,
        cc.last_name,
        cc.cohort_month,
        cc.first_order_date,
        cc.last_order_date,

        -- Order metrics
        COUNT(o.order_id) as total_orders,
        SUM(o.total_amount) as total_spent,
        AVG(o.total_amount) as avg_order_value,

        -- Time metrics
        CASE
            WHEN cc.first_order_date IS NOT NULL
            THEN cc.last_order_date - cc.first_order_date
            ELSE NULL
        END as customer_lifespan_days,

        -- Product diversity
        COUNT(DISTINCT oi.product_id) as unique_products_purchased,
        COUNT(DISTINCT cat.category_id) as unique_categories_purchased
    FROM customer_cohorts cc
    LEFT JOIN orders o ON cc.customer_id = o.customer_id
        AND o.status IN ('completed', 'delivered')
    LEFT JOIN order_items oi ON o.order_id = oi.order_id
    LEFT JOIN products p ON oi.product_id = p.product_id
    LEFT JOIN categories cat ON p.category_id = cat.category_id
    GROUP BY
        cc.customer_id, cc.first_name, cc.last_name,
        cc.cohort_month, cc.first_order_date, cc.last_order_date
```

```
)
SELECT
    cm.*,

    -- Customer segmentation
    CASE
        WHEN cm.total_orders = 0 THEN 'Never Purchased'
        WHEN cm.total_orders = 1 THEN 'One-time Buyer'
        WHEN cm.total_orders BETWEEN 2 AND 5 THEN 'Occasional Buyer'
        WHEN cm.total_orders BETWEEN 6 AND 15 THEN 'Regular Customer'
        ELSE 'VIP Customer'
    END as customer_segment,

    -- Value segmentation
    CASE
        WHEN cm.total_spent = 0 THEN 'No Value'
        WHEN cm.total_spent < 100 THEN 'Low Value'
        WHEN cm.total_spent < 500 THEN 'Medium Value'
        WHEN cm.total_spent < 2000 THEN 'High Value'
        ELSE 'Premium Value'
    END as value_segment,

    -- Engagement metrics
    CASE
        WHEN cm.last_order_date IS NULL THEN 'Never Ordered'
        WHEN cm.last_order_date >= CURRENT_DATE - INTERVAL '30 days' THEN 'Active'
        WHEN cm.last_order_date >= CURRENT_DATE - INTERVAL '90 days' THEN 'Recent'
        WHEN cm.last_order_date >= CURRENT_DATE - INTERVAL '365 days' THEN
'Dormant'
        ELSE 'Inactive'
    END as engagement_status
FROM customer_metrics cm
ORDER BY cm.total_spent DESC, cm.total_orders DESC;
```

Example 3: Inventory and Sales Correlation

```
-- Analyze relationship between inventory levels and sales performance
SELECT
    p.product_id,
    p.product_name,
    cat.category_name,
    p.price,
    p.stock_quantity as current_stock,

    -- Sales metrics (last 90 days)
    COALESCE(sales.units_sold, 0) as units_sold_90d,
    COALESCE(sales.revenue, 0) as revenue_90d,
    COALESCE(sales.order_count, 0) as order_count_90d,

    -- Inventory metrics
    CASE
```

```
        WHEN COALESCE(sales.units_sold, 0) = 0 THEN NULL
        ELSE p.stock_quantity::float / (sales.units_sold / 90.0)  -- Days of
inventory
    END as days_of_inventory,

    -- Performance indicators
    CASE
        WHEN p.stock_quantity = 0 THEN 'Out of Stock'
        WHEN p.stock_quantity < 10 THEN 'Low Stock'
        WHEN COALESCE(sales.units_sold, 0) = 0 THEN 'No Recent Sales'
        WHEN p.stock_quantity::float / (sales.units_sold / 90.0) < 30 THEN 'Fast
Moving'
        WHEN p.stock_quantity::float / (sales.units_sold / 90.0) > 180 THEN 'Slow
Moving'
        ELSE 'Normal'
    END as inventory_status,

    -- Supplier information
    s.supplier_name,
    s.lead_time_days,

    -- Reorder recommendations
    CASE
        WHEN p.stock_quantity = 0 THEN 'URGENT: Reorder immediately'
        WHEN p.stock_quantity < 10 AND COALESCE(sales.units_sold, 0) > 0 THEN
'HIGH: Reorder soon'
        WHEN p.stock_quantity::float / (sales.units_sold / 90.0) < 30 THEN
'MEDIUM: Monitor closely'
        WHEN COALESCE(sales.units_sold, 0) = 0 AND p.stock_quantity > 50 THEN
'LOW: Consider discontinuing'
        ELSE 'NORMAL: No action needed'
    END as reorder_priority
FROM products p
JOIN categories cat ON p.category_id = cat.category_id
LEFT JOIN suppliers s ON p.supplier_id = s.supplier_id
LEFT JOIN (
    SELECT
        oi.product_id,
        SUM(oi.quantity) as units_sold,
        SUM(oi.quantity * oi.unit_price) as revenue,
        COUNT(DISTINCT oi.order_id) as order_count
    FROM order_items oi
    JOIN orders o ON oi.order_id = o.order_id
    WHERE
        o.order_date >= CURRENT_DATE - INTERVAL '90 days'
        AND o.status IN ('completed', 'shipped', 'delivered')
    GROUP BY oi.product_id
) sales ON p.product_id = sales.product_id
WHERE p.status = 'active'
ORDER BY
    CASE
        WHEN p.stock_quantity = 0 THEN 1
        WHEN p.stock_quantity < 10 AND COALESCE(sales.units_sold, 0) > 0 THEN 2
        ELSE 3
```

```
    END,
    revenue_90d DESC;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"Explain the difference between INNER JOIN and LEFT JOIN"**
    - INNER JOIN: Only returns rows where there's a match in both tables
    - LEFT JOIN: Returns all rows from left table, plus matching rows from right table
    - Use INNER JOIN when you need data that exists in both tables
    - Use LEFT JOIN when you want to preserve all records from one table
2. **"When would you use a self-join?"**
    - Employee-manager relationships
    - Hierarchical data (categories, organizational charts)
    - Comparing records within the same table
    - Finding duplicates or similar records
3. **"How do you optimize join performance?"**
    - Create indexes on join columns
    - Use appropriate join types
    - Filter early with WHERE clauses
    - Consider query execution order
    - Use EXPLAIN to analyze query plans

Best Practices:

1. **Always Use Table Aliases:**

```
-- Good: Clear and readable
SELECT c.name, o.total
FROM customers c
JOIN orders o ON c.id = o.customer_id;

-- Avoid: Ambiguous and verbose
SELECT customers.name, orders.total
FROM customers
JOIN orders ON customers.id = orders.customer_id;
```

2. **Be Explicit with JOIN Types:**

```
-- Good: Explicit intent
SELECT *
```

```
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;

-- Avoid: Implicit join (harder to read)
SELECT *
FROM customers c, orders o
WHERE c.customer_id = o.customer_id;
```

3. **Handle NULL Values Appropriately:**

```
-- Use COALESCE for NULL handling
SELECT
    c.name,
    COALESCE(SUM(o.total), 0) as total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name;
```

⚠ Things to Watch Out For

1. **Cartesian Products (Accidental CROSS JOINs)**

```
-- Dangerous: Missing join condition
SELECT *
FROM customers c
JOIN orders o;  -- Missing ON clause!

-- This creates a cartesian product: every customer with every order
-- 1,000 customers × 10,000 orders = 10,000,000 rows!

-- Correct: Always include join conditions
SELECT *
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

2. **NULL Handling in Joins**

```
-- NULLs don't match in joins
SELECT *
FROM table1 t1
JOIN table2 t2 ON t1.column = t2.column;
-- Rows with NULL in either column won't match

-- Use IS NULL checks when needed
SELECT *
FROM customers c
```

```
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.customer_id IS NULL;  -- Customers without orders
```

3. **Performance Issues with Large Joins**

```
-- Slow: No indexes on join columns
SELECT *
FROM large_table1 t1
JOIN large_table2 t2 ON t1.unindexed_column = t2.unindexed_column;

-- Fast: Proper indexes
CREATE INDEX idx_table1_join_col ON large_table1(join_column);
CREATE INDEX idx_table2_join_col ON large_table2(join_column);
```

4. **Ambiguous Column Names**

```
-- Error: Ambiguous column reference
SELECT id, name  -- Which table's id and name?
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;

-- Correct: Use table aliases
SELECT c.customer_id, c.name, o.order_id
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

5. **Incorrect Join Logic**

```
-- Wrong: Using WHERE instead of ON for join conditions
SELECT *
FROM customers c
LEFT JOIN orders o
WHERE c.customer_id = o.customer_id;  -- This becomes an INNER JOIN!

-- Correct: Use ON for join conditions
SELECT *
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

🚀 Next Steps

In the next chapter, we'll explore **Aggregate Functions** (COUNT, SUM, AVG, etc.) in detail. You'll learn how to use these functions effectively with joins and grouping to create powerful analytical queries.

📝 Quick Practice

Using a sample e-commerce database, try these join exercises:

1. **Basic Joins**: Find all customers who have placed orders in the last 30 days
2. **LEFT JOIN**: List all products and their total sales (including products with no sales)
3. **Multiple Joins**: Create a report showing customer name, order date, product name, and category for all orders
4. **Self Join**: Find all employees and their managers in an employee hierarchy
5. **Complex Analysis**: Identify customers who bought products from multiple categories

Consider:

- Which join type is most appropriate for each scenario?
- How do you handle NULL values in your results?
- What indexes would improve performance?
- How do you avoid cartesian products?

📓 Chapter 8: Aggregate Functions (COUNT, SUM, AVG, etc.)

🎯 What You'll Learn

- Understanding aggregate functions and their purpose
- COUNT for counting records and non-NULL values
- SUM for calculating totals
- AVG for calculating averages
- MIN and MAX for finding extremes
- Advanced aggregate functions (STDDEV, VARIANCE, etc.)
- Using aggregates with GROUP BY and HAVING
- Window functions vs aggregate functions
- Performance considerations

📖 Concept Explanation

Aggregate functions perform calculations on a set of rows and return a single result. They're essential for data analysis, reporting, and business intelligence. Think of them as "summarizing" your data - turning many rows into meaningful insights.

Key Characteristics:

1. **Input**: Multiple rows
2. **Output**: Single value
3. **NULL Handling**: Most aggregates ignore NULL values
4. **Grouping**: Work with GROUP BY to create subtotals
5. **Filtering**: Use HAVING to filter aggregated results

Common Aggregate Functions:

- **COUNT()**: Count rows or non-NULL values
- **SUM()**: Add up numeric values
- **AVG()**: Calculate average of numeric values
- **MIN()**: Find minimum value
- **MAX()**: Find maximum value
- **STDDEV()**: Calculate standard deviation
- **VARIANCE()**: Calculate variance

🔢 COUNT Function

COUNT Variations

```
COUNT(*)           -- Count all rows (including NULLs)
COUNT(column)      -- Count non-NULL values in column
COUNT(DISTINCT column)  -- Count unique non-NULL values
```

Basic COUNT Examples

1. Simple Counting:

```sql
-- Total number of customers
SELECT COUNT(*) as total_customers
FROM customers;

-- Customers with phone numbers
SELECT COUNT(phone) as customers_with_phone
FROM customers;

-- Customers with verified emails
SELECT COUNT(*) as verified_customers
FROM customers
WHERE email_verified = true;

-- Unique countries
SELECT COUNT(DISTINCT country) as unique_countries
FROM customers;
```

2. COUNT with Conditions:

```sql
-- Count orders by status
SELECT
    COUNT(*) as total_orders,
    COUNT(CASE WHEN status = 'completed' THEN 1 END) as completed_orders,
    COUNT(CASE WHEN status = 'pending' THEN 1 END) as pending_orders,
```

```
    COUNT(CASE WHEN status = 'cancelled' THEN 1 END) as cancelled_orders,

    -- Calculate percentages
    ROUND(
        COUNT(CASE WHEN status = 'completed' THEN 1 END) * 100.0 / COUNT(*), 2
    ) as completion_rate
FROM orders;

-- Count customers by registration year
SELECT
    EXTRACT(YEAR FROM created_at) as registration_year,
    COUNT(*) as new_customers,
    COUNT(CASE WHEN email_verified = true THEN 1 END) as verified_customers
FROM customers
GROUP BY EXTRACT(YEAR FROM created_at)
ORDER BY registration_year;
```

3. Advanced COUNT Patterns:

```
-- Count distinct customers who made purchases in different time periods
SELECT
    COUNT(DISTINCT CASE
        WHEN order_date >= CURRENT_DATE - INTERVAL '30 days'
        THEN customer_id
    END) as active_last_30_days,

    COUNT(DISTINCT CASE
        WHEN order_date >= CURRENT_DATE - INTERVAL '90 days'
        THEN customer_id
    END) as active_last_90_days,

    COUNT(DISTINCT customer_id) as total_customers_with_orders
FROM orders
WHERE status IN ('completed', 'shipped', 'delivered');

-- Count products by availability and price range
SELECT
    COUNT(*) as total_products,
    COUNT(CASE WHEN stock_quantity > 0 THEN 1 END) as in_stock_products,
    COUNT(CASE WHEN price < 50 THEN 1 END) as budget_products,
    COUNT(CASE WHEN price BETWEEN 50 AND 200 THEN 1 END) as mid_range_products,
    COUNT(CASE WHEN price > 200 THEN 1 END) as premium_products,

    -- Count products with reviews
    COUNT(CASE WHEN average_rating IS NOT NULL THEN 1 END) as
products_with_reviews
FROM products
WHERE status = 'active';
```

➕ SUM Function

Basic SUM Examples

1. Simple Totals:

```sql
-- Total revenue
SELECT SUM(total_amount) as total_revenue
FROM orders
WHERE status IN ('completed', 'delivered');

-- Total inventory value
SELECT SUM(price * stock_quantity) as total_inventory_value
FROM products
WHERE status = 'active';

-- Total units sold
SELECT SUM(quantity) as total_units_sold
FROM order_items oi
JOIN orders o ON oi.order_id = o.order_id
WHERE o.status IN ('completed', 'delivered');
```

2. Conditional SUM:

```sql
-- Revenue by payment method
SELECT
    SUM(CASE WHEN payment_method = 'credit_card' THEN total_amount ELSE 0 END) as
credit_card_revenue,
    SUM(CASE WHEN payment_method = 'paypal' THEN total_amount ELSE 0 END) as
paypal_revenue,
    SUM(CASE WHEN payment_method = 'bank_transfer' THEN total_amount ELSE 0 END)
as bank_transfer_revenue,
    SUM(total_amount) as total_revenue
FROM orders
WHERE status = 'completed';

-- Monthly revenue comparison
SELECT
    EXTRACT(MONTH FROM order_date) as month,
    SUM(total_amount) as monthly_revenue,
    SUM(CASE WHEN EXTRACT(YEAR FROM order_date) = 2024 THEN total_amount ELSE 0
END) as revenue_2024,
    SUM(CASE WHEN EXTRACT(YEAR FROM order_date) = 2023 THEN total_amount ELSE 0
END) as revenue_2023
FROM orders
WHERE status = 'completed'
  AND order_date >= '2023-01-01'
GROUP BY EXTRACT(MONTH FROM order_date)
ORDER BY month;
```

3. SUM with Joins:

```sql
-- Revenue by product category
SELECT
    c.category_name,
    SUM(oi.quantity * oi.unit_price) as category_revenue,
    SUM(oi.quantity) as units_sold,
    COUNT(DISTINCT o.order_id) as order_count,
    COUNT(DISTINCT o.customer_id) as unique_customers
FROM categories c
JOIN products p ON c.category_id = p.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE o.status IN ('completed', 'delivered')
  AND o.order_date >= CURRENT_DATE - INTERVAL '1 year'
GROUP BY c.category_id, c.category_name
ORDER BY category_revenue DESC;

-- Customer lifetime value
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    COUNT(o.order_id) as total_orders,
    SUM(o.total_amount) as lifetime_value,
    AVG(o.total_amount) as avg_order_value,
    MIN(o.order_date) as first_order_date,
    MAX(o.order_date) as last_order_date
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status IN ('completed', 'delivered')
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
HAVING SUM(o.total_amount) > 500  -- High-value customers only
ORDER BY lifetime_value DESC;
```

📊 AVG Function

Basic AVG Examples

1. Simple Averages:

```sql
-- Average order value
SELECT AVG(total_amount) as avg_order_value
FROM orders
WHERE status = 'completed';

-- Average product price by category
SELECT
```

```
    c.category_name,
    AVG(p.price) as avg_price,
    COUNT(p.product_id) as product_count
FROM categories c
JOIN products p ON c.category_id = p.category_id
WHERE p.status = 'active'
GROUP BY c.category_id, c.category_name
ORDER BY avg_price DESC;

-- Average customer age
SELECT
    AVG(EXTRACT(YEAR FROM age(date_of_birth))) as avg_age
FROM customers
WHERE date_of_birth IS NOT NULL;
```

2. Weighted Averages:

```
-- Weighted average product rating (by number of reviews)
SELECT
    p.product_id,
    p.product_name,
    COUNT(r.review_id) as review_count,
    AVG(r.rating) as avg_rating,

    -- Weighted average considering review volume
    CASE
        WHEN COUNT(r.review_id) >= 10 THEN AVG(r.rating)
        WHEN COUNT(r.review_id) >= 5 THEN AVG(r.rating) * 0.9
        WHEN COUNT(r.review_id) >= 1 THEN AVG(r.rating) * 0.8
        ELSE NULL
    END as weighted_rating
FROM products p
LEFT JOIN reviews r ON p.product_id = r.product_id
WHERE p.status = 'active'
GROUP BY p.product_id, p.product_name
HAVING COUNT(r.review_id) > 0
ORDER BY weighted_rating DESC;

-- Average order value by customer segment
SELECT
    CASE
        WHEN customer_orders.order_count = 1 THEN 'One-time'
        WHEN customer_orders.order_count BETWEEN 2 AND 5 THEN 'Occasional'
        WHEN customer_orders.order_count BETWEEN 6 AND 15 THEN 'Regular'
        ELSE 'VIP'
    END as customer_segment,
    COUNT(*) as customers_in_segment,
    AVG(customer_orders.avg_order_value) as segment_avg_order_value,
    AVG(customer_orders.total_spent) as segment_avg_lifetime_value
FROM (
    SELECT
        c.customer_id,
```

```
        COUNT(o.order_id) as order_count,
        AVG(o.total_amount) as avg_order_value,
        SUM(o.total_amount) as total_spent
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id
) customer_orders
GROUP BY customer_segment
ORDER BY segment_avg_lifetime_value DESC;
```

3. Moving Averages:

```
-- 7-day moving average of daily sales
SELECT
    order_date,
    daily_revenue,
    AVG(daily_revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as moving_avg_7_days
FROM (
    SELECT
        DATE(order_date) as order_date,
        SUM(total_amount) as daily_revenue
    FROM orders
    WHERE status = 'completed'
      AND order_date >= CURRENT_DATE - INTERVAL '90 days'
    GROUP BY DATE(order_date)
) daily_sales
ORDER BY order_date;
```

🔍 MIN and MAX Functions

Basic MIN/MAX Examples

1. Simple Extremes:

```
-- Price range analysis
SELECT
    MIN(price) as cheapest_product,
    MAX(price) as most_expensive_product,
    AVG(price) as average_price,
    MAX(price) - MIN(price) as price_range
FROM products
WHERE status = 'active';

-- Order date range
```

```sql
SELECT
    MIN(order_date) as first_order,
    MAX(order_date) as latest_order,
    MAX(order_date) - MIN(order_date) as business_duration_days
FROM orders;

-- Customer age demographics
SELECT
    MIN(EXTRACT(YEAR FROM age(date_of_birth))) as youngest_customer,
    MAX(EXTRACT(YEAR FROM age(date_of_birth))) as oldest_customer,
    AVG(EXTRACT(YEAR FROM age(date_of_birth))) as average_age
FROM customers
WHERE date_of_birth IS NOT NULL;
```

2. MIN/MAX with GROUP BY:

```sql
-- Price range by category
SELECT
    c.category_name,
    COUNT(p.product_id) as product_count,
    MIN(p.price) as min_price,
    MAX(p.price) as max_price,
    AVG(p.price) as avg_price,
    MAX(p.price) - MIN(p.price) as price_range
FROM categories c
JOIN products p ON c.category_id = p.category_id
WHERE p.status = 'active'
GROUP BY c.category_id, c.category_name
ORDER BY price_range DESC;

-- Customer order patterns
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as total_orders,
    MIN(o.order_date) as first_order_date,
    MAX(o.order_date) as last_order_date,
    MIN(o.total_amount) as smallest_order,
    MAX(o.total_amount) as largest_order,

    -- Calculate customer lifecycle metrics
    MAX(o.order_date) - MIN(o.order_date) as customer_lifespan_days,
    CURRENT_DATE - MAX(o.order_date) as days_since_last_order
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(o.order_id) > 1  -- Multi-order customers only
ORDER BY customer_lifespan_days DESC;
```

3. Finding Records with MIN/MAX Values:

```
-- Most expensive product in each category
SELECT
    c.category_name,
    p.product_name,
    p.price
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.price = (
    SELECT MAX(p2.price)
    FROM products p2
    WHERE p2.category_id = p.category_id
      AND p2.status = 'active'
)
AND p.status = 'active'
ORDER BY p.price DESC;

-- Latest order for each customer
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    o.order_id,
    o.order_date,
    o.total_amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_date = (
    SELECT MAX(o2.order_date)
    FROM orders o2
    WHERE o2.customer_id = c.customer_id
      AND o2.status = 'completed'
)
AND o.status = 'completed'
ORDER BY o.order_date DESC;
```

☑ Advanced Aggregate Functions

Statistical Functions

1. Standard Deviation and Variance:

```
-- Price distribution analysis
SELECT
    c.category_name,
    COUNT(p.product_id) as product_count,
    AVG(p.price) as avg_price,
    STDDEV(p.price) as price_stddev,
```

```
    VARIANCE(p.price) as price_variance,

    -- Coefficient of variation (relative variability)
    CASE
        WHEN AVG(p.price) > 0
        THEN STDDEV(p.price) / AVG(p.price)
        ELSE NULL
    END as coefficient_of_variation
FROM categories c
JOIN products p ON c.category_id = p.category_id
WHERE p.status = 'active'
GROUP BY c.category_id, c.category_name
HAVING COUNT(p.product_id) >= 5  -- Categories with enough products
ORDER BY coefficient_of_variation DESC;

-- Order value consistency by customer
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count,
    AVG(o.total_amount) as avg_order_value,
    STDDEV(o.total_amount) as order_value_stddev,

    -- Customer spending consistency
    CASE
        WHEN STDDEV(o.total_amount) < AVG(o.total_amount) * 0.3 THEN 'Consistent'
        WHEN STDDEV(o.total_amount) < AVG(o.total_amount) * 0.7 THEN 'Moderate'
        ELSE 'Variable'
    END as spending_pattern
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(o.order_id) >= 5  -- Customers with enough orders
ORDER BY order_value_stddev DESC;
```

2. Percentiles (PostgreSQL):

```
-- Order value percentiles
SELECT
    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY total_amount) as
q1_25th_percentile,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY total_amount) as
median_50th_percentile,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY total_amount) as
q3_75th_percentile,
    PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY total_amount) as
p90_90th_percentile,
    PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY total_amount) as
p95_95th_percentile
FROM orders
```

```
WHERE status = 'completed'
  AND order_date >= CURRENT_DATE - INTERVAL '1 year';

-- Customer spending distribution
SELECT
    customer_segment,
    COUNT(*) as customer_count,
    AVG(total_spent) as avg_spent,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY total_spent) as median_spent
FROM (
    SELECT
        c.customer_id,
        SUM(o.total_amount) as total_spent,
        CASE
            WHEN SUM(o.total_amount) < 100 THEN 'Low Value'
            WHEN SUM(o.total_amount) < 500 THEN 'Medium Value'
            WHEN SUM(o.total_amount) < 2000 THEN 'High Value'
            ELSE 'Premium Value'
        END as customer_segment
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id
) customer_spending
GROUP BY customer_segment
ORDER BY avg_spent DESC;
```

String Aggregation

1. STRING_AGG (PostgreSQL) / GROUP_CONCAT (MySQL):

```
-- PostgreSQL: Concatenate product names by category
SELECT
    c.category_name,
    COUNT(p.product_id) as product_count,
    STRING_AGG(p.product_name, ', ' ORDER BY p.product_name) as product_list
FROM categories c
JOIN products p ON c.category_id = p.category_id
WHERE p.status = 'active'
GROUP BY c.category_id, c.category_name
ORDER BY product_count DESC;

-- MySQL: Same functionality with GROUP_CONCAT
SELECT
    c.category_name,
    COUNT(p.product_id) as product_count,
    GROUP_CONCAT(p.product_name ORDER BY p.product_name SEPARATOR ', ') as
product_list
FROM categories c
JOIN products p ON c.category_id = p.category_id
WHERE p.status = 'active'
```

```
GROUP BY c.category_id, c.category_name
ORDER BY product_count DESC;

-- Customer order history summary
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count,
    SUM(o.total_amount) as total_spent,
    STRING_AGG(
        o.order_id::text || ' ($' || o.total_amount || ')',
        ', '
        ORDER BY o.order_date DESC
    ) as order_history
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(o.order_id) <= 5  -- Customers with few orders
ORDER BY total_spent DESC;
```

🔄 Aggregate Functions with Window Functions

Running Totals and Cumulative Aggregates

1. Running Totals:

```
-- Daily sales with running total
SELECT
    order_date,
    daily_revenue,
    SUM(daily_revenue) OVER (
        ORDER BY order_date
        ROWS UNBOUNDED PRECEDING
    ) as running_total,

    -- Running average
    AVG(daily_revenue) OVER (
        ORDER BY order_date
        ROWS UNBOUNDED PRECEDING
    ) as running_average
FROM (
    SELECT
        DATE(order_date) as order_date,
        SUM(total_amount) as daily_revenue
    FROM orders
    WHERE status = 'completed'
      AND order_date >= CURRENT_DATE - INTERVAL '90 days'
    GROUP BY DATE(order_date)
```

```
) daily_sales
ORDER BY order_date;
```

2. Ranking with Aggregates:

```
-- Top customers by spending with rankings
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(o.total_amount) as total_spent,
    COUNT(o.order_id) as order_count,
    AVG(o.total_amount) as avg_order_value,

    -- Rankings
    RANK() OVER (ORDER BY SUM(o.total_amount) DESC) as spending_rank,
    RANK() OVER (ORDER BY COUNT(o.order_id) DESC) as frequency_rank,
    RANK() OVER (ORDER BY AVG(o.total_amount) DESC) as avg_value_rank,

    -- Percentile rankings
    PERCENT_RANK() OVER (ORDER BY SUM(o.total_amount)) as spending_percentile
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_spent DESC;
```

Real-World Aggregate Examples

Example 1: Sales Performance Dashboard

```
-- Comprehensive sales dashboard with multiple aggregates
WITH monthly_sales AS (
    SELECT
        DATE_TRUNC('month', o.order_date) as month,
        COUNT(DISTINCT o.order_id) as order_count,
        COUNT(DISTINCT o.customer_id) as unique_customers,
        SUM(o.total_amount) as revenue,
        AVG(o.total_amount) as avg_order_value,
        MIN(o.total_amount) as min_order_value,
        MAX(o.total_amount) as max_order_value,
        STDDEV(o.total_amount) as order_value_stddev
    FROM orders o
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY DATE_TRUNC('month', o.order_date)
),
product_performance AS (
```

```
    SELECT
        DATE_TRUNC('month', o.order_date) as month,
        COUNT(DISTINCT oi.product_id) as unique_products_sold,
        SUM(oi.quantity) as total_units_sold,
        AVG(oi.unit_price) as avg_unit_price
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY DATE_TRUNC('month', o.order_date)
)
SELECT
    ms.month,
    TO_CHAR(ms.month, 'YYYY-MM') as month_label,

    -- Order metrics
    ms.order_count,
    ms.unique_customers,
    ms.revenue,
    ms.avg_order_value,
    ms.order_value_stddev,

    -- Product metrics
    pp.unique_products_sold,
    pp.total_units_sold,
    pp.avg_unit_price,

    -- Calculated metrics
    ROUND(ms.revenue / ms.order_count, 2) as revenue_per_order,
    ROUND(ms.revenue / ms.unique_customers, 2) as revenue_per_customer,
    ROUND(pp.total_units_sold::float / ms.order_count, 2) as units_per_order,

    -- Growth metrics (compared to previous month)
    LAG(ms.revenue) OVER (ORDER BY ms.month) as prev_month_revenue,
    ROUND(
        (ms.revenue - LAG(ms.revenue) OVER (ORDER BY ms.month)) /
        LAG(ms.revenue) OVER (ORDER BY ms.month) * 100, 2
    ) as revenue_growth_percent
FROM monthly_sales ms
JOIN product_performance pp ON ms.month = pp.month
ORDER BY ms.month DESC;
```

Example 2: Customer Segmentation Analysis

```
-- RFM Analysis using aggregates
WITH customer_rfm AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
```

```sql
        -- Recency: Days since last order
        CURRENT_DATE - MAX(o.order_date) as days_since_last_order,

        -- Frequency: Number of orders
        COUNT(o.order_id) as order_frequency,

        -- Monetary: Total amount spent
        SUM(o.total_amount) as total_monetary_value,
        AVG(o.total_amount) as avg_order_value,

        -- Additional metrics
        MIN(o.order_date) as first_order_date,
        MAX(o.order_date) as last_order_date,
        MAX(o.order_date) - MIN(o.order_date) as customer_lifespan_days
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status IN ('completed', 'delivered')
    GROUP BY c.customer_id, c.first_name, c.last_name, c.email
),
rfm_scores AS (
    SELECT
        *,
        -- Recency score (lower days = higher score)
        CASE
            WHEN days_since_last_order <= 30 THEN 5
            WHEN days_since_last_order <= 60 THEN 4
            WHEN days_since_last_order <= 90 THEN 3
            WHEN days_since_last_order <= 180 THEN 2
            ELSE 1
        END as recency_score,

        -- Frequency score
        CASE
            WHEN order_frequency >= 20 THEN 5
            WHEN order_frequency >= 10 THEN 4
            WHEN order_frequency >= 5 THEN 3
            WHEN order_frequency >= 2 THEN 2
            ELSE 1
        END as frequency_score,

        -- Monetary score
        CASE
            WHEN total_monetary_value >= 2000 THEN 5
            WHEN total_monetary_value >= 1000 THEN 4
            WHEN total_monetary_value >= 500 THEN 3
            WHEN total_monetary_value >= 100 THEN 2
            ELSE 1
        END as monetary_score
    FROM customer_rfm
)
SELECT
    -- Customer segments based on RFM scores
    CASE
```

```
        WHEN recency_score >= 4 AND frequency_score >= 4 AND monetary_score >= 4
THEN 'Champions'
        WHEN recency_score >= 3 AND frequency_score >= 3 AND monetary_score >= 3
THEN 'Loyal Customers'
        WHEN recency_score >= 4 AND frequency_score <= 2 THEN 'New Customers'
        WHEN recency_score <= 2 AND frequency_score >= 3 AND monetary_score >= 3
THEN 'At Risk'
        WHEN recency_score <= 2 AND frequency_score <= 2 THEN 'Lost Customers'
        WHEN frequency_score >= 3 AND monetary_score <= 2 THEN 'Price Sensitive'
        ELSE 'Others'
    END as customer_segment,

    COUNT(*) as customer_count,
    AVG(total_monetary_value) as avg_monetary_value,
    AVG(order_frequency) as avg_frequency,
    AVG(days_since_last_order) as avg_recency,
    SUM(total_monetary_value) as segment_total_value,

    -- Segment percentages
    ROUND(
        COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (), 2
    ) as segment_percentage
FROM rfm_scores
GROUP BY customer_segment
ORDER BY segment_total_value DESC;
```

Example 3: Product Performance Analysis

```
-- Comprehensive product analysis with multiple aggregates
SELECT
    p.product_id,
    p.product_name,
    c.category_name,
    p.price as current_price,
    p.stock_quantity,

    -- Sales metrics (last 12 months)
    COALESCE(sales.units_sold, 0) as units_sold_12m,
    COALESCE(sales.revenue, 0) as revenue_12m,
    COALESCE(sales.order_count, 0) as order_count_12m,
    COALESCE(sales.unique_customers, 0) as unique_customers_12m,
    COALESCE(sales.avg_unit_price, p.price) as avg_selling_price,

    -- Review metrics
    COALESCE(reviews.review_count, 0) as review_count,
    COALESCE(reviews.avg_rating, 0) as avg_rating,
    COALESCE(reviews.rating_stddev, 0) as rating_stddev,

    -- Performance calculations
    CASE
        WHEN sales.units_sold > 0 THEN
```

```
            ROUND(sales.revenue / sales.units_sold, 2)
        ELSE p.price
    END as actual_avg_price,

    CASE
        WHEN sales.units_sold > 0 THEN
            ROUND((p.price - sales.avg_unit_price) / p.price * 100, 2)
        ELSE 0
    END as avg_discount_percent,

    -- Inventory turnover (annual rate)
    CASE
        WHEN p.stock_quantity > 0 AND sales.units_sold > 0 THEN
            ROUND(sales.units_sold::float / p.stock_quantity, 2)
        ELSE 0
    END as inventory_turnover_rate,

    -- Performance categories
    CASE
        WHEN COALESCE(sales.revenue, 0) = 0 THEN 'No Sales'
        WHEN sales.revenue > 10000 THEN 'Top Performer'
        WHEN sales.revenue > 5000 THEN 'Good Performer'
        WHEN sales.revenue > 1000 THEN 'Average Performer'
        ELSE 'Poor Performer'
    END as performance_category,

    -- Stock status
    CASE
        WHEN p.stock_quantity = 0 THEN 'Out of Stock'
        WHEN p.stock_quantity < 10 THEN 'Low Stock'
        WHEN sales.units_sold > 0 AND
             (p.stock_quantity::float / (sales.units_sold / 12.0)) < 30 THEN 'Fast
Moving'
        WHEN sales.units_sold > 0 AND
             (p.stock_quantity::float / (sales.units_sold / 12.0)) > 180 THEN
'Slow Moving'
        ELSE 'Normal Stock'
    END as stock_status
FROM products p
JOIN categories c ON p.category_id = c.category_id
LEFT JOIN (
    SELECT
        oi.product_id,
        SUM(oi.quantity) as units_sold,
        SUM(oi.quantity * oi.unit_price) as revenue,
        COUNT(DISTINCT oi.order_id) as order_count,
        COUNT(DISTINCT o.customer_id) as unique_customers,
        AVG(oi.unit_price) as avg_unit_price
    FROM order_items oi
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY oi.product_id
) sales ON p.product_id = sales.product_id
```

```
LEFT JOIN (
    SELECT
        product_id,
        COUNT(*) as review_count,
        AVG(rating) as avg_rating,
        STDDEV(rating) as rating_stddev
    FROM reviews
    WHERE created_at >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY product_id
) reviews ON p.product_id = reviews.product_id
WHERE p.status = 'active'
ORDER BY revenue_12m DESC;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between COUNT(*) and COUNT(column)?"**
    - COUNT(*): Counts all rows, including those with NULL values
    - COUNT(column): Counts only non-NULL values in the specified column
    - COUNT(DISTINCT column): Counts unique non-NULL values
2. **"How do aggregate functions handle NULL values?"**
    - Most aggregates (SUM, AVG, MIN, MAX) ignore NULL values
    - COUNT(*) includes rows with NULLs, COUNT(column) excludes them
    - If all values are NULL, SUM returns NULL, COUNT returns 0
3. **"When would you use HAVING vs WHERE?"**
    - WHERE: Filters rows before grouping and aggregation
    - HAVING: Filters groups after aggregation
    - HAVING can use aggregate functions, WHERE cannot

Performance Best Practices:

1. **Index Strategy for Aggregates:**

```
-- Create indexes for GROUP BY columns
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
CREATE INDEX idx_order_items_product ON order_items(product_id);

-- Covering indexes for common aggregate queries
CREATE INDEX idx_orders_status_date_amount ON orders(status, order_date,
total_amount);
```

2. **Efficient Aggregate Queries:**

```sql
-- Good: Filter before aggregating
SELECT customer_id, SUM(total_amount)
FROM orders
WHERE status = 'completed'  -- Filter first
GROUP BY customer_id;

-- Avoid: Aggregating then filtering
SELECT customer_id, total_spent
FROM (
    SELECT customer_id, SUM(total_amount) as total_spent
    FROM orders
    GROUP BY customer_id
) t
WHERE total_spent > 1000;  -- Better to use HAVING
```

3. **Use Appropriate Data Types:**

```sql
-- Use DECIMAL for monetary calculations
SELECT SUM(price::DECIMAL(10,2)) FROM products;

-- Be careful with integer division
SELECT AVG(price::FLOAT) FROM products;  -- Avoid integer truncation
```

⚠ Things to Watch Out For

1. **NULL Handling Gotchas**

```sql
-- Problem: Unexpected results with NULLs
SELECT AVG(rating) FROM reviews;  -- Ignores NULL ratings

-- Solution: Handle NULLs explicitly
SELECT
    AVG(rating) as avg_rating_excluding_nulls,
    AVG(COALESCE(rating, 0)) as avg_rating_including_nulls_as_zero,
    COUNT(*) as total_reviews,
    COUNT(rating) as reviews_with_rating
FROM reviews;
```

2. **Integer Division Issues**

```sql
-- Problem: Integer division truncates
SELECT 5 / 2;  -- Returns 2, not 2.5

-- Solution: Cast to decimal/float
SELECT 5.0 / 2;  -- Returns 2.5
```

```
SELECT 5::FLOAT / 2;  -- PostgreSQL
SELECT CAST(5 AS FLOAT) / 2;  -- Standard SQL
```

3. **GROUP BY Requirements**

```
-- Error: Non-aggregated column not in GROUP BY
SELECT customer_id, order_date, COUNT(*)
FROM orders
GROUP BY customer_id;  -- order_date must be in GROUP BY or aggregated

-- Correct: Include all non-aggregated columns
SELECT customer_id, order_date, COUNT(*)
FROM orders
GROUP BY customer_id, order_date;

-- Or aggregate the column
SELECT customer_id, MAX(order_date), COUNT(*)
FROM orders
GROUP BY customer_id;
```

4. **Performance with Large Datasets**

```
-- Slow: Aggregating without proper indexes
SELECT category_id, AVG(price)
FROM products
GROUP BY category_id;  -- Needs index on category_id

-- Fast: With proper index
CREATE INDEX idx_products_category ON products(category_id);
```

5. **Precision Issues with Averages**

```
-- Problem: Precision loss
SELECT AVG(price) FROM products;  -- May lose precision

-- Solution: Use appropriate precision
SELECT ROUND(AVG(price), 2) FROM products;
SELECT AVG(price::DECIMAL(10,2)) FROM products;
```

🚀 Next Steps

In the next chapter, we'll explore **Subqueries and Nested Queries** - powerful techniques for creating complex queries that use the results of one query within another. You'll learn about correlated subqueries, EXISTS clauses, and how to break down complex problems into manageable parts.

Quick Practice

Try these aggregate function exercises:

1. **Basic Aggregates**: Calculate total sales, average order value, and customer count for the last quarter
2. **Conditional Aggregates**: Count orders by status and calculate completion rate
3. **Grouped Aggregates**: Find the top 5 product categories by revenue
4. **Statistical Analysis**: Calculate price distribution (mean, median, standard deviation) by category
5. **Customer Analysis**: Identify customers in the top 10% by spending using percentiles

Consider:

- How do you handle NULL values in your calculations?
- What indexes would improve performance?
- How do you ensure precision in monetary calculations?
- What business insights can you derive from the aggregated data?

Chapter 9: Subqueries and Nested Queries

What You'll Learn

- Understanding subqueries and their types
- Scalar subqueries (single value)
- Row subqueries (single row, multiple columns)
- Table subqueries (multiple rows and columns)
- Correlated vs non-correlated subqueries
- EXISTS and NOT EXISTS clauses
- IN and NOT IN with subqueries
- ANY, ALL, and SOME operators
- Common Table Expressions (CTEs)
- Performance considerations and optimization

Concept Explanation

Subqueries (also called nested queries or inner queries) are queries embedded within another query. They allow you to break complex problems into smaller, manageable parts and perform operations that would be difficult or impossible with a single query.

Key Characteristics:

1. **Nested Structure**: A query inside another query
2. **Execution Order**: Inner query executes first (usually)
3. **Scope**: Inner query can reference outer query columns (correlated)
4. **Return Types**: Can return single value, single row, or multiple rows
5. **Usage**: Can be used in SELECT, FROM, WHERE, HAVING clauses

Types of Subqueries:

- **Scalar Subquery**: Returns single value (one row, one column)
- **Row Subquery**: Returns single row with multiple columns
- **Table Subquery**: Returns multiple rows and columns
- **Correlated Subquery**: References outer query columns
- **Non-correlated Subquery**: Independent of outer query

🔍 Scalar Subqueries (Single Value)

Basic Scalar Subqueries

1. Simple Scalar Examples:

```
-- Find customers who spent more than the average
SELECT
    customer_id,
    first_name,
    last_name,
    total_spent
FROM (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        SUM(o.total_amount) as total_spent
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id, c.first_name, c.last_name
) customer_spending
WHERE total_spent > (
    SELECT AVG(customer_total)
    FROM (
        SELECT SUM(o.total_amount) as customer_total
        FROM orders o
        WHERE o.status = 'completed'
        GROUP BY o.customer_id
    ) avg_calculation
);

-- Products priced above category average
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name,
    (
        SELECT AVG(p2.price)
        FROM products p2
        WHERE p2.category_id = p.category_id
          AND p2.status = 'active'
    ) as category_avg_price
```

```
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND p.price > (
      SELECT AVG(p2.price)
      FROM products p2
      WHERE p2.category_id = p.category_id
        AND p2.status = 'active'
  )
ORDER BY p.price DESC;

-- Orders with above-average value
SELECT
    order_id,
    customer_id,
    order_date,
    total_amount,
    (
        SELECT AVG(total_amount)
        FROM orders
        WHERE status = 'completed'
    ) as overall_avg_order_value,

    total_amount - (
        SELECT AVG(total_amount)
        FROM orders
        WHERE status = 'completed'
    ) as difference_from_avg
FROM orders
WHERE status = 'completed'
  AND total_amount > (
      SELECT AVG(total_amount)
      FROM orders
      WHERE status = 'completed'
  )
ORDER BY total_amount DESC;
```

2. Scalar Subqueries in SELECT Clause:

```
-- Customer summary with comparative metrics
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,

    -- Customer's order count
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
```

```
    ) as order_count,

    -- Customer's total spending
    (
        SELECT COALESCE(SUM(o.total_amount), 0)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as total_spent,

    -- Customer's average order value
    (
        SELECT COALESCE(AVG(o.total_amount), 0)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as avg_order_value,

    -- Days since last order
    (
        SELECT COALESCE(
            CURRENT_DATE - MAX(o.order_date),
            CURRENT_DATE - c.created_at
        )
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as days_since_last_order,

    -- Compare to overall average
    (
        SELECT AVG(total_amount)
        FROM orders
        WHERE status = 'completed'
    ) as overall_avg_order_value
FROM customers c
WHERE c.status = 'active'
ORDER BY total_spent DESC;

-- Product performance with market context
SELECT
    p.product_id,
    p.product_name,
    p.price,
    p.stock_quantity,
    c.category_name,

    -- Units sold (last 12 months)
    (
        SELECT COALESCE(SUM(oi.quantity), 0)
        FROM order_items oi
        JOIN orders o ON oi.order_id = o.order_id
        WHERE oi.product_id = p.product_id
          AND o.status IN ('completed', 'delivered')
```

```
            AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    ) as units_sold_12m,

    -- Revenue generated (last 12 months)
    (
        SELECT COALESCE(SUM(oi.quantity * oi.unit_price), 0)
        FROM order_items oi
        JOIN orders o ON oi.order_id = o.order_id
        WHERE oi.product_id = p.product_id
          AND o.status IN ('completed', 'delivered')
          AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    ) as revenue_12m,

    -- Category average price
    (
        SELECT AVG(p2.price)
        FROM products p2
        WHERE p2.category_id = p.category_id
          AND p2.status = 'active'
    ) as category_avg_price,

    -- Market share within category (by revenue)
    (
        SELECT
            CASE
                WHEN SUM(oi.quantity * oi.unit_price) > 0 THEN
                    ROUND(
                        (
                            SELECT SUM(oi2.quantity * oi2.unit_price)
                            FROM order_items oi2
                            JOIN orders o2 ON oi2.order_id = o2.order_id
                            WHERE oi2.product_id = p.product_id
                              AND o2.status IN ('completed', 'delivered')
                              AND o2.order_date >= CURRENT_DATE - INTERVAL '12
months'
                        ) * 100.0 / SUM(oi.quantity * oi.unit_price), 2
                    )
                ELSE 0
            END
        FROM order_items oi
        JOIN orders o ON oi.order_id = o.order_id
        JOIN products p2 ON oi.product_id = p2.product_id
        WHERE p2.category_id = p.category_id
          AND o.status IN ('completed', 'delivered')
          AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    ) as category_market_share_percent
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
ORDER BY revenue_12m DESC;
```

📊 Row and Table Subqueries

Row Subqueries (Single Row, Multiple Columns)

1. Row Comparisons:

```sql
-- Find customers with the same registration date and city as customer ID 1
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    city,
    created_at
FROM customers
WHERE (city, DATE(created_at)) = (
    SELECT city, DATE(created_at)
    FROM customers
    WHERE customer_id = 1
)
AND customer_id != 1;

-- Products with the same price and category as the most expensive product
SELECT
    p1.product_id,
    p1.product_name,
    p1.price,
    c.category_name
FROM products p1
JOIN categories c ON p1.category_id = c.category_id
WHERE (p1.price, p1.category_id) = (
    SELECT price, category_id
    FROM products
    WHERE status = 'active'
    ORDER BY price DESC
    LIMIT 1
)
AND p1.status = 'active';

-- Orders placed on the same date with the same total as a specific order
SELECT
    o1.order_id,
    o1.customer_id,
    o1.order_date,
    o1.total_amount,
    c.first_name,
    c.last_name
FROM orders o1
JOIN customers c ON o1.customer_id = c.customer_id
WHERE (DATE(o1.order_date), o1.total_amount) = (
    SELECT DATE(order_date), total_amount
    FROM orders
```

```
    WHERE order_id = 12345  -- Specific order to match
)
AND o1.order_id != 12345
AND o1.status = 'completed';
```

Table Subqueries (Multiple Rows and Columns)

1. Subqueries in FROM Clause:

```
-- Customer spending analysis using derived table
SELECT
    spending_tier,
    COUNT(*) as customer_count,
    AVG(total_spent) as avg_spending,
    MIN(total_spent) as min_spending,
    MAX(total_spent) as max_spending
FROM (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        SUM(o.total_amount) as total_spent,
        CASE
            WHEN SUM(o.total_amount) < 100 THEN 'Low Spender'
            WHEN SUM(o.total_amount) < 500 THEN 'Medium Spender'
            WHEN SUM(o.total_amount) < 2000 THEN 'High Spender'
            ELSE 'VIP Spender'
        END as spending_tier
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id, c.first_name, c.last_name
) customer_spending
GROUP BY spending_tier
ORDER BY avg_spending DESC;

-- Monthly sales trends with year-over-year comparison
SELECT
    month_name,
    current_year_sales,
    previous_year_sales,
    CASE
        WHEN previous_year_sales > 0 THEN
            ROUND(
                (current_year_sales - previous_year_sales) / previous_year_sales *
100, 2
            )
        ELSE NULL
    END as growth_percentage
FROM (
    SELECT
```

```
        TO_CHAR(order_date, 'Month') as month_name,
        EXTRACT(MONTH FROM order_date) as month_num,
        SUM(CASE WHEN EXTRACT(YEAR FROM order_date) = 2024 THEN total_amount ELSE
0 END) as current_year_sales,
        SUM(CASE WHEN EXTRACT(YEAR FROM order_date) = 2023 THEN total_amount ELSE
0 END) as previous_year_sales
    FROM orders
    WHERE status = 'completed'
      AND EXTRACT(YEAR FROM order_date) IN (2023, 2024)
    GROUP BY EXTRACT(MONTH FROM order_date), TO_CHAR(order_date, 'Month')
) monthly_comparison
ORDER BY month_num;

-- Product performance ranking within categories
SELECT
    category_name,
    product_name,
    revenue_12m,
    category_rank,
    category_total_revenue,
    ROUND(revenue_12m / category_total_revenue * 100, 2) as category_revenue_share
FROM (
    SELECT
        c.category_name,
        p.product_name,
        COALESCE(sales.revenue, 0) as revenue_12m,
        ROW_NUMBER() OVER (
            PARTITION BY c.category_id
            ORDER BY COALESCE(sales.revenue, 0) DESC
        ) as category_rank,
        SUM(COALESCE(sales.revenue, 0)) OVER (
            PARTITION BY c.category_id
        ) as category_total_revenue
    FROM products p
    JOIN categories c ON p.category_id = c.category_id
    LEFT JOIN (
        SELECT
            oi.product_id,
            SUM(oi.quantity * oi.unit_price) as revenue
        FROM order_items oi
        JOIN orders o ON oi.order_id = o.order_id
        WHERE o.status IN ('completed', 'delivered')
          AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
        GROUP BY oi.product_id
    ) sales ON p.product_id = sales.product_id
    WHERE p.status = 'active'
) ranked_products
WHERE category_rank <= 5  -- Top 5 products per category
ORDER BY category_name, category_rank;
```

Correlated Subqueries

Understanding Correlated Subqueries

1. Basic Correlated Examples:

```sql
-- Customers who have placed orders above their personal average
SELECT
    o.order_id,
    o.customer_id,
    c.first_name,
    c.last_name,
    o.order_date,
    o.total_amount,
    (
        SELECT AVG(o2.total_amount)
        FROM orders o2
        WHERE o2.customer_id = o.customer_id
          AND o2.status = 'completed'
    ) as customer_avg_order_value
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.status = 'completed'
  AND o.total_amount > (
      SELECT AVG(o2.total_amount)
      FROM orders o2
      WHERE o2.customer_id = o.customer_id
        AND o2.status = 'completed'
  )
ORDER BY o.total_amount DESC;

-- Products that are the most expensive in their category
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND p.price = (
      SELECT MAX(p2.price)
      FROM products p2
      WHERE p2.category_id = p.category_id
        AND p2.status = 'active'
  )
ORDER BY p.price DESC;

-- Customers with their latest order information
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
```

```
    (
        SELECT o.order_date
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
        ORDER BY o.order_date DESC
        LIMIT 1
    ) as last_order_date,
    (
        SELECT o.total_amount
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
        ORDER BY o.order_date DESC
        LIMIT 1
    ) as last_order_amount,
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as total_orders
FROM customers c
WHERE c.status = 'active'
  AND EXISTS (
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
        AND o.status = 'completed'
  )
ORDER BY last_order_date DESC;
```

2. Advanced Correlated Patterns:

```
-- Products with above-average sales in their category
SELECT
    p.product_id,
    p.product_name,
    c.category_name,
    p.price,
    (
        SELECT COALESCE(SUM(oi.quantity), 0)
        FROM order_items oi
        JOIN orders o ON oi.order_id = o.order_id
        WHERE oi.product_id = p.product_id
          AND o.status IN ('completed', 'delivered')
          AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    ) as units_sold_12m,
    (
        SELECT AVG(category_sales.units_sold)
        FROM (
            SELECT
```

```
                p2.product_id,
                COALESCE(SUM(oi2.quantity), 0) as units_sold
            FROM products p2
            LEFT JOIN order_items oi2 ON p2.product_id = oi2.product_id
            LEFT JOIN orders o2 ON oi2.order_id = o2.order_id
                AND o2.status IN ('completed', 'delivered')
                AND o2.order_date >= CURRENT_DATE - INTERVAL '12 months'
            WHERE p2.category_id = p.category_id
              AND p2.status = 'active'
            GROUP BY p2.product_id
        ) category_sales
    ) as category_avg_units_sold
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND (
      SELECT COALESCE(SUM(oi.quantity), 0)
      FROM order_items oi
      JOIN orders o ON oi.order_id = o.order_id
      WHERE oi.product_id = p.product_id
        AND o.status IN ('completed', 'delivered')
        AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
  ) > (
      SELECT AVG(category_sales.units_sold)
      FROM (
          SELECT
              p2.product_id,
              COALESCE(SUM(oi2.quantity), 0) as units_sold
          FROM products p2
          LEFT JOIN order_items oi2 ON p2.product_id = oi2.product_id
          LEFT JOIN orders o2 ON oi2.order_id = o2.order_id
              AND o2.status IN ('completed', 'delivered')
              AND o2.order_date >= CURRENT_DATE - INTERVAL '12 months'
          WHERE p2.category_id = p.category_id
            AND p2.status = 'active'
          GROUP BY p2.product_id
      ) category_sales
  )
ORDER BY units_sold_12m DESC;

-- Customers who haven't ordered in longer than their typical interval
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    (
        SELECT MAX(o.order_date)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as last_order_date,
    (
        SELECT
```

```
            AVG(EXTRACT(EPOCH FROM (o2.order_date - o1.order_date)) / 86400)
        FROM orders o1
        JOIN orders o2 ON o1.customer_id = o2.customer_id
        WHERE o1.customer_id = c.customer_id
          AND o1.status = 'completed'
          AND o2.status = 'completed'
          AND o2.order_date > o1.order_date
          AND NOT EXISTS (
              SELECT 1
              FROM orders o3
              WHERE o3.customer_id = c.customer_id
                AND o3.status = 'completed'
                AND o3.order_date > o1.order_date
                AND o3.order_date < o2.order_date
          )
    ) as avg_days_between_orders,
    CURRENT_DATE - (
        SELECT MAX(o.order_date)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as days_since_last_order
FROM customers c
WHERE c.status = 'active'
  AND EXISTS (
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
        AND o.status = 'completed'
      HAVING COUNT(*) >= 3  -- At least 3 orders to calculate average interval
  )
  AND (
      CURRENT_DATE - (
          SELECT MAX(o.order_date)
          FROM orders o
          WHERE o.customer_id = c.customer_id
            AND o.status = 'completed'
      )
  ) > (
      SELECT
          AVG(EXTRACT(EPOCH FROM (o2.order_date - o1.order_date)) / 86400) * 1.5
      FROM orders o1
      JOIN orders o2 ON o1.customer_id = o2.customer_id
      WHERE o1.customer_id = c.customer_id
        AND o1.status = 'completed'
        AND o2.status = 'completed'
        AND o2.order_date > o1.order_date
        AND NOT EXISTS (
            SELECT 1
            FROM orders o3
            WHERE o3.customer_id = c.customer_id
              AND o3.status = 'completed'
              AND o3.order_date > o1.order_date
              AND o3.order_date < o2.order_date
```

```
        )
  )
ORDER BY days_since_last_order DESC;
```

✅ EXISTS and NOT EXISTS

EXISTS Clause

1. Basic EXISTS Examples:

```
-- Customers who have placed at least one order
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    c.created_at
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.status = 'completed'
)
ORDER BY c.created_at DESC;

-- Products that have been ordered in the last 30 days
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND EXISTS (
      SELECT 1
      FROM order_items oi
      JOIN orders o ON oi.order_id = o.order_id
      WHERE oi.product_id = p.product_id
        AND o.status IN ('completed', 'delivered')
        AND o.order_date >= CURRENT_DATE - INTERVAL '30 days'
  )
ORDER BY p.product_name;

-- Categories that have products with reviews
SELECT
    c.category_id,
    c.category_name,
    c.description
```

```
FROM categories c
WHERE EXISTS (
    SELECT 1
    FROM products p
    WHERE p.category_id = c.category_id
      AND p.status = 'active'
      AND EXISTS (
          SELECT 1
          FROM reviews r
          WHERE r.product_id = p.product_id
      )
)
ORDER BY c.category_name;
```

2. Complex EXISTS Patterns:

```
-- Customers who have ordered from multiple categories
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    (
        SELECT COUNT(DISTINCT p.category_id)
        FROM orders o
        JOIN order_items oi ON o.order_id = oi.order_id
        JOIN products p ON oi.product_id = p.product_id
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as categories_purchased_from
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM (
        SELECT DISTINCT p.category_id
        FROM orders o
        JOIN order_items oi ON o.order_id = oi.order_id
        JOIN products p ON oi.product_id = p.product_id
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) customer_categories
    HAVING COUNT(*) >= 3  -- At least 3 different categories
)
ORDER BY categories_purchased_from DESC;

-- Products that are frequently bought together
SELECT
    p1.product_id as product_a_id,
    p1.product_name as product_a_name,
    p2.product_id as product_b_id,
    p2.product_name as product_b_name,
    COUNT(*) as times_bought_together
```

```sql
FROM products p1
JOIN order_items oi1 ON p1.product_id = oi1.product_id
JOIN order_items oi2 ON oi1.order_id = oi2.order_id
JOIN products p2 ON oi2.product_id = p2.product_id
JOIN orders o ON oi1.order_id = o.order_id
WHERE p1.product_id < p2.product_id  -- Avoid duplicates
  AND o.status = 'completed'
  AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
  AND EXISTS (
      SELECT 1
      FROM order_items oi_check
      JOIN orders o_check ON oi_check.order_id = o_check.order_id
      WHERE oi_check.product_id = p1.product_id
        AND o_check.status = 'completed'
        AND o_check.order_date >= CURRENT_DATE - INTERVAL '12 months'
      HAVING COUNT(*) >= 10  -- Product A sold at least 10 times
  )
  AND EXISTS (
      SELECT 1
      FROM order_items oi_check
      JOIN orders o_check ON oi_check.order_id = o_check.order_id
      WHERE oi_check.product_id = p2.product_id
        AND o_check.status = 'completed'
        AND o_check.order_date >= CURRENT_DATE - INTERVAL '12 months'
      HAVING COUNT(*) >= 10  -- Product B sold at least 10 times
  )
GROUP BY p1.product_id, p1.product_name, p2.product_id, p2.product_name
HAVING COUNT(*) >= 5  -- Bought together at least 5 times
ORDER BY times_bought_together DESC;
```

NOT EXISTS Clause

1. Basic NOT EXISTS Examples:

```sql
-- Customers who have never placed an order
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    c.created_at,
    CURRENT_DATE - c.created_at as days_since_registration
FROM customers c
WHERE c.status = 'active'
  AND NOT EXISTS (
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
  )
ORDER BY c.created_at;
```

```sql
-- Products that have never been ordered
SELECT
    p.product_id,
    p.product_name,
    p.price,
    p.stock_quantity,
    c.category_name,
    p.created_at
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND NOT EXISTS (
      SELECT 1
      FROM order_items oi
      WHERE oi.product_id = p.product_id
  )
ORDER BY p.created_at DESC;

-- Categories without any active products
SELECT
    c.category_id,
    c.category_name,
    c.description
FROM categories c
WHERE NOT EXISTS (
    SELECT 1
    FROM products p
    WHERE p.category_id = c.category_id
      AND p.status = 'active'
)
ORDER BY c.category_name;
```

2. Advanced NOT EXISTS Patterns:

```sql
-- Customers who haven't ordered in the last 6 months but were active before
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    (
        SELECT MAX(o.order_date)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as last_order_date,
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as total_orders,
```

```
    (
        SELECT SUM(o.total_amount)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as lifetime_value
FROM customers c
WHERE c.status = 'active'
  AND EXISTS (
      -- Has placed orders before
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
        AND o.status = 'completed'
  )
  AND NOT EXISTS (
      -- But not in the last 6 months
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
        AND o.status = 'completed'
        AND o.order_date >= CURRENT_DATE - INTERVAL '6 months'
  )
ORDER BY lifetime_value DESC;

-- Products available in some categories but missing in others
SELECT
    c.category_id,
    c.category_name,
    missing_products.product_name,
    missing_products.avg_price_in_other_categories
FROM categories c
CROSS JOIN (
    SELECT DISTINCT
        p.product_name,
        AVG(p.price) as avg_price_in_other_categories
    FROM products p
    WHERE p.status = 'active'
    GROUP BY p.product_name
    HAVING COUNT(DISTINCT p.category_id) >= 2  -- Available in at least 2
categories
) missing_products
WHERE NOT EXISTS (
    SELECT 1
    FROM products p
    WHERE p.category_id = c.category_id
      AND p.product_name = missing_products.product_name
      AND p.status = 'active'
)
ORDER BY c.category_name, missing_products.product_name;

-- Identify gaps in customer purchase patterns
SELECT
    c.customer_id,
```

```
    c.first_name,
    c.last_name,
    popular_products.product_name,
    popular_products.times_ordered,
    popular_products.avg_price
FROM customers c
CROSS JOIN (
    SELECT
        p.product_id,
        p.product_name,
        COUNT(*) as times_ordered,
        AVG(oi.unit_price) as avg_price
    FROM products p
    JOIN order_items oi ON p.product_id = oi.product_id
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status = 'completed'
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY p.product_id, p.product_name
    HAVING COUNT(*) >= 50  -- Popular products (ordered 50+ times)
) popular_products
WHERE EXISTS (
    -- Customer has placed orders
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.status = 'completed'
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
)
AND NOT EXISTS (
    -- But hasn't ordered this popular product
    SELECT 1
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    WHERE o.customer_id = c.customer_id
      AND oi.product_id = popular_products.product_id
      AND o.status = 'completed'
)
AND (
    SELECT COUNT(*)
    FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.status = 'completed'
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
) >= 3  -- Active customers (3+ orders in last year)
ORDER BY c.customer_id, popular_products.times_ordered DESC;
```

📋 IN and NOT IN with Subqueries

IN Clause with Subqueries

1. Basic IN Examples:

```sql
-- Customers who have ordered specific high-value products
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email
FROM customers c
WHERE c.customer_id IN (
    SELECT DISTINCT o.customer_id
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    WHERE oi.product_id IN (
        SELECT product_id
        FROM products
        WHERE price > 500
          AND status = 'active'
    )
    AND o.status = 'completed'
)
ORDER BY c.last_name, c.first_name;

-- Products in categories that have high average ratings
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND p.category_id IN (
      SELECT p2.category_id
      FROM products p2
      JOIN reviews r ON p2.product_id = r.product_id
      WHERE p2.status = 'active'
      GROUP BY p2.category_id
      HAVING AVG(r.rating) >= 4.0
        AND COUNT(r.review_id) >= 20  -- At least 20 reviews
  )
ORDER BY c.category_name, p.product_name;

-- Orders placed by customers from specific cities
SELECT
    o.order_id,
    o.customer_id,
    o.order_date,
    o.total_amount,
    c.city
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.status = 'completed'
  AND c.city IN (
```

```
        SELECT city
        FROM customers
        WHERE status = 'active'
        GROUP BY city
        HAVING COUNT(*) >= 100  -- Cities with at least 100 customers
    )
ORDER BY o.order_date DESC;
```

2. Complex IN Patterns:

```
-- Products that are top sellers in their respective categories
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name,
    sales_data.units_sold,
    sales_data.revenue
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN (
    SELECT
        oi.product_id,
        SUM(oi.quantity) as units_sold,
        SUM(oi.quantity * oi.unit_price) as revenue
    FROM order_items oi
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY oi.product_id
) sales_data ON p.product_id = sales_data.product_id
WHERE p.status = 'active'
  AND p.product_id IN (
      -- Top 3 products by revenue in each category
      SELECT ranked_products.product_id
      FROM (
          SELECT
              p2.product_id,
              p2.category_id,
              SUM(oi2.quantity * oi2.unit_price) as revenue,
              ROW_NUMBER() OVER (
                  PARTITION BY p2.category_id
                  ORDER BY SUM(oi2.quantity * oi2.unit_price) DESC
              ) as category_rank
          FROM products p2
          JOIN order_items oi2 ON p2.product_id = oi2.product_id
          JOIN orders o2 ON oi2.order_id = o2.order_id
          WHERE p2.status = 'active'
            AND o2.status IN ('completed', 'delivered')
            AND o2.order_date >= CURRENT_DATE - INTERVAL '12 months'
          GROUP BY p2.product_id, p2.category_id
      ) ranked_products
```

```
        WHERE ranked_products.category_rank <= 3
    )
ORDER BY c.category_name, sales_data.revenue DESC;

-- Customers who have purchased from the same categories as VIP customers
SELECT DISTINCT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    customer_stats.total_spent,
    customer_stats.order_count
FROM customers c
JOIN (
    SELECT
        c2.customer_id,
        SUM(o2.total_amount) as total_spent,
        COUNT(o2.order_id) as order_count
    FROM customers c2
    JOIN orders o2 ON c2.customer_id = o2.customer_id
    WHERE o2.status = 'completed'
    GROUP BY c2.customer_id
) customer_stats ON c.customer_id = customer_stats.customer_id
WHERE customer_stats.total_spent < 2000  -- Non-VIP customers
  AND c.customer_id IN (
      SELECT DISTINCT o.customer_id
      FROM orders o
      JOIN order_items oi ON o.order_id = oi.order_id
      JOIN products p ON oi.product_id = p.product_id
      WHERE o.status = 'completed'
        AND p.category_id IN (
            -- Categories purchased by VIP customers
            SELECT DISTINCT p2.category_id
            FROM orders o2
            JOIN order_items oi2 ON o2.order_id = oi2.order_id
            JOIN products p2 ON oi2.product_id = p2.product_id
            JOIN (
                SELECT customer_id
                FROM orders
                WHERE status = 'completed'
                GROUP BY customer_id
                HAVING SUM(total_amount) >= 2000  -- VIP threshold
            ) vip_customers ON o2.customer_id = vip_customers.customer_id
            WHERE o2.status = 'completed'
        )
  )
ORDER BY customer_stats.total_spent DESC;
```

NOT IN Clause with Subqueries

1. Basic NOT IN Examples:

```sql
-- Customers who have never ordered expensive products
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as total_orders,
    (
        SELECT COALESCE(SUM(o.total_amount), 0)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as total_spent
FROM customers c
WHERE c.status = 'active'
  AND c.customer_id NOT IN (
      SELECT DISTINCT o.customer_id
      FROM orders o
      JOIN order_items oi ON o.order_id = oi.order_id
      JOIN products p ON oi.product_id = p.product_id
      WHERE o.status = 'completed'
        AND p.price > 200  -- Expensive products
        AND o.customer_id IS NOT NULL  -- Important for NOT IN
  )
  AND EXISTS (
      -- But have placed at least one order
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
        AND o.status = 'completed'
  )
ORDER BY total_spent DESC;

-- Products not ordered in the last 6 months
SELECT
    p.product_id,
    p.product_name,
    p.price,
    p.stock_quantity,
    c.category_name,
    (
        SELECT MAX(o.order_date)
        FROM order_items oi
        JOIN orders o ON oi.order_id = o.order_id
        WHERE oi.product_id = p.product_id
          AND o.status IN ('completed', 'delivered')
    ) as last_ordered_date
FROM products p
```

```sql
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND p.product_id NOT IN (
      SELECT DISTINCT oi.product_id
      FROM order_items oi
      JOIN orders o ON oi.order_id = o.order_id
      WHERE o.status IN ('completed', 'delivered')
        AND o.order_date >= CURRENT_DATE - INTERVAL '6 months'
        AND oi.product_id IS NOT NULL  -- Important for NOT IN
  )
ORDER BY last_ordered_date DESC NULLS LAST;

-- Categories without products in a specific price range
SELECT
    c.category_id,
    c.category_name,
    (
        SELECT COUNT(*)
        FROM products p
        WHERE p.category_id = c.category_id
          AND p.status = 'active'
    ) as total_products,
    (
        SELECT MIN(p.price)
        FROM products p
        WHERE p.category_id = c.category_id
          AND p.status = 'active'
    ) as min_price,
    (
        SELECT MAX(p.price)
        FROM products p
        WHERE p.category_id = c.category_id
          AND p.status = 'active'
    ) as max_price
FROM categories c
WHERE c.category_id NOT IN (
    SELECT DISTINCT p.category_id
    FROM products p
    WHERE p.status = 'active'
      AND p.price BETWEEN 50 AND 150  -- Mid-range products
      AND p.category_id IS NOT NULL  -- Important for NOT IN
)
AND EXISTS (
    -- But have at least one active product
    SELECT 1
    FROM products p
    WHERE p.category_id = c.category_id
      AND p.status = 'active'
)
ORDER BY c.category_name;
```

2. NOT IN with NULL Handling:

```sql
-- Safe NOT IN pattern (handling NULLs)
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email
FROM customers c
WHERE c.status = 'active'
  AND c.customer_id NOT IN (
      SELECT o.customer_id
      FROM orders o
      WHERE o.status = 'cancelled'
        AND o.customer_id IS NOT NULL  -- Explicit NULL check
  )
ORDER BY c.last_name, c.first_name;

-- Alternative using NOT EXISTS (safer than NOT IN)
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email
FROM customers c
WHERE c.status = 'active'
  AND NOT EXISTS (
      SELECT 1
      FROM orders o
      WHERE o.customer_id = c.customer_id
        AND o.status = 'cancelled'
  )
ORDER BY c.last_name, c.first_name;
```

🔄 ANY, ALL, and SOME Operators

ANY and SOME Operators (equivalent)

1. Basic ANY/SOME Examples:

```sql
-- Products more expensive than ANY product in the 'Electronics' category
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND p.price > ANY (
      SELECT p2.price
```

```
        FROM products p2
        JOIN categories c2 ON p2.category_id = c2.category_id
        WHERE c2.category_name = 'Electronics'
          AND p2.status = 'active'
    )
ORDER BY p.price DESC;

-- Customers who have spent more than ANY customer from 'New York'
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.city,
    customer_spending.total_spent
FROM customers c
JOIN (
    SELECT
        customer_id,
        SUM(total_amount) as total_spent
    FROM orders
    WHERE status = 'completed'
    GROUP BY customer_id
) customer_spending ON c.customer_id = customer_spending.customer_id
WHERE c.status = 'active'
  AND customer_spending.total_spent > ANY (
      SELECT SUM(o.total_amount)
      FROM customers c2
      JOIN orders o ON c2.customer_id = o.customer_id
      WHERE c2.city = 'New York'
        AND c2.status = 'active'
        AND o.status = 'completed'
      GROUP BY c2.customer_id
  )
  AND c.city != 'New York'  -- Exclude New York customers
ORDER BY customer_spending.total_spent DESC;

-- Orders with amounts greater than ANY order from the previous month
SELECT
    o.order_id,
    o.customer_id,
    o.order_date,
    o.total_amount,
    c.first_name,
    c.last_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.status = 'completed'
  AND EXTRACT(MONTH FROM o.order_date) = EXTRACT(MONTH FROM CURRENT_DATE)
  AND EXTRACT(YEAR FROM o.order_date) = EXTRACT(YEAR FROM CURRENT_DATE)
  AND o.total_amount > ANY (
      SELECT o2.total_amount
      FROM orders o2
      WHERE o2.status = 'completed'
        AND EXTRACT(MONTH FROM o2.order_date) = EXTRACT(MONTH FROM CURRENT_DATE) -
```

```
1
        AND EXTRACT(YEAR FROM o2.order_date) = EXTRACT(YEAR FROM CURRENT_DATE)
  )
ORDER BY o.total_amount DESC;
```

ALL Operator

1. Basic ALL Examples:

```
-- Products more expensive than ALL products in the 'Books' category
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active'
  AND p.price > ALL (
      SELECT p2.price
      FROM products p2
      JOIN categories c2 ON p2.category_id = c2.category_id
      WHERE c2.category_name = 'Books'
        AND p2.status = 'active'
  )
ORDER BY p.price;

-- Customers who have spent more than ALL customers from 'Small Town'
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.city,
    customer_spending.total_spent
FROM customers c
JOIN (
    SELECT
        customer_id,
        SUM(total_amount) as total_spent
    FROM orders
    WHERE status = 'completed'
    GROUP BY customer_id
) customer_spending ON c.customer_id = customer_spending.customer_id
WHERE c.status = 'active'
  AND customer_spending.total_spent > ALL (
      SELECT COALESCE(SUM(o.total_amount), 0)
      FROM customers c2
      LEFT JOIN orders o ON c2.customer_id = o.customer_id
          AND o.status = 'completed'
      WHERE c2.city = 'Small Town'
        AND c2.status = 'active'
```

```
      GROUP BY c2.customer_id
  )
  AND c.city != 'Small Town'  -- Exclude Small Town customers
ORDER BY customer_spending.total_spent DESC;

-- Products with ratings higher than ALL products in competing categories
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name,
    product_ratings.avg_rating,
    product_ratings.review_count
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN (
    SELECT
        product_id,
        AVG(rating) as avg_rating,
        COUNT(*) as review_count
    FROM reviews
    GROUP BY product_id
    HAVING COUNT(*) >= 5  -- At least 5 reviews
) product_ratings ON p.product_id = product_ratings.product_id
WHERE p.status = 'active'
  AND product_ratings.avg_rating > ALL (
      SELECT AVG(r2.rating)
      FROM products p2
      JOIN reviews r2 ON p2.product_id = r2.product_id
      JOIN categories c2 ON p2.category_id = c2.category_id
      WHERE c2.category_name IN ('Competing Category 1', 'Competing Category 2')
        AND p2.status = 'active'
      GROUP BY p2.product_id
      HAVING COUNT(r2.review_id) >= 5
  )
ORDER BY product_ratings.avg_rating DESC;
```

2. Complex ANY/ALL Patterns:

```
-- Products that outperform ALL products in lower price categories
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name,
    sales_metrics.units_sold_12m,
    sales_metrics.revenue_12m
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN (
    SELECT
        oi.product_id,
```

```
        SUM(oi.quantity) as units_sold_12m,
        SUM(oi.quantity * oi.unit_price) as revenue_12m
    FROM order_items oi
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY oi.product_id
) sales_metrics ON p.product_id = sales_metrics.product_id
WHERE p.status = 'active'
  AND p.price >= 100  -- Focus on higher-priced products
  AND sales_metrics.units_sold_12m > ALL (
      -- Compare to all products in lower price ranges
      SELECT COALESCE(SUM(oi2.quantity), 0)
      FROM products p2
      LEFT JOIN order_items oi2 ON p2.product_id = oi2.product_id
      LEFT JOIN orders o2 ON oi2.order_id = o2.order_id
          AND o2.status IN ('completed', 'delivered')
          AND o2.order_date >= CURRENT_DATE - INTERVAL '12 months'
      WHERE p2.status = 'active'
        AND p2.price < p.price * 0.7  -- Products priced at least 30% lower
      GROUP BY p2.product_id
  )
ORDER BY sales_metrics.units_sold_12m DESC;

-- Customers whose average order value exceeds ANY VIP customer's average
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    customer_metrics.order_count,
    customer_metrics.total_spent,
    customer_metrics.avg_order_value
FROM customers c
JOIN (
    SELECT
        customer_id,
        COUNT(*) as order_count,
        SUM(total_amount) as total_spent,
        AVG(total_amount) as avg_order_value
    FROM orders
    WHERE status = 'completed'
    GROUP BY customer_id
    HAVING COUNT(*) >= 3  -- At least 3 orders for reliable average
) customer_metrics ON c.customer_id = customer_metrics.customer_id
WHERE c.status = 'active'
  AND customer_metrics.total_spent < 2000  -- Non-VIP customers
  AND customer_metrics.avg_order_value > ANY (
      -- Compare to VIP customers' average order values
      SELECT AVG(o.total_amount)
      FROM orders o
      JOIN (
          SELECT customer_id
          FROM orders
```

```
            WHERE status = 'completed'
            GROUP BY customer_id
            HAVING SUM(total_amount) >= 2000  -- VIP threshold
        ) vip_customers ON o.customer_id = vip_customers.customer_id
        WHERE o.status = 'completed'
        GROUP BY o.customer_id
        HAVING COUNT(*) >= 3
    )
ORDER BY customer_metrics.avg_order_value DESC;
```

Common Table Expressions (CTEs)

Basic CTEs

1. Simple CTE Examples:

```
-- Customer spending analysis with CTE
WITH customer_spending AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
        COUNT(o.order_id) as order_count,
        SUM(o.total_amount) as total_spent,
        AVG(o.total_amount) as avg_order_value,
        MIN(o.order_date) as first_order_date,
        MAX(o.order_date) as last_order_date
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id, c.first_name, c.last_name, c.email
)
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    order_count,
    total_spent,
    avg_order_value,
    first_order_date,
    last_order_date,
    CURRENT_DATE - last_order_date as days_since_last_order,
    CASE
        WHEN total_spent >= 2000 THEN 'VIP'
        WHEN total_spent >= 1000 THEN 'Premium'
        WHEN total_spent >= 500 THEN 'Regular'
        ELSE 'Basic'
    END as customer_tier
```

```sql
FROM customer_spending
WHERE order_count >= 2
ORDER BY total_spent DESC;

-- Product performance analysis with CTE
WITH product_sales AS (
    SELECT
        p.product_id,
        p.product_name,
        p.price,
        c.category_name,
        COALESCE(SUM(oi.quantity), 0) as units_sold,
        COALESCE(SUM(oi.quantity * oi.unit_price), 0) as revenue,
        COUNT(DISTINCT o.order_id) as order_count
    FROM products p
    JOIN categories c ON p.category_id = c.category_id
    LEFT JOIN order_items oi ON p.product_id = oi.product_id
    LEFT JOIN orders o ON oi.order_id = o.order_id
        AND o.status IN ('completed', 'delivered')
        AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    WHERE p.status = 'active'
    GROUP BY p.product_id, p.product_name, p.price, c.category_name
),
category_averages AS (
    SELECT
        category_name,
        AVG(units_sold) as avg_category_units,
        AVG(revenue) as avg_category_revenue
    FROM product_sales
    GROUP BY category_name
)
SELECT
    ps.product_id,
    ps.product_name,
    ps.price,
    ps.category_name,
    ps.units_sold,
    ps.revenue,
    ps.order_count,
    ca.avg_category_units,
    ca.avg_category_revenue,
    CASE
        WHEN ps.units_sold > ca.avg_category_units THEN 'Above Average'
        WHEN ps.units_sold = ca.avg_category_units THEN 'Average'
        ELSE 'Below Average'
    END as performance_vs_category
FROM product_sales ps
JOIN category_averages ca ON ps.category_name = ca.category_name
ORDER BY ps.revenue DESC;
```

Multiple CTEs

1. Complex Analysis with Multiple CTEs:

```
-- Comprehensive customer analysis with multiple CTEs
WITH customer_orders AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
        c.city,
        c.created_at as registration_date,
        COUNT(o.order_id) as order_count,
        SUM(o.total_amount) as total_spent,
        AVG(o.total_amount) as avg_order_value,
        MIN(o.order_date) as first_order_date,
        MAX(o.order_date) as last_order_date
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
        AND o.status = 'completed'
    WHERE c.status = 'active'
    GROUP BY c.customer_id, c.first_name, c.last_name, c.email, c.city,
c.created_at
),
customer_segments AS (
    SELECT
        *,
        CASE
            WHEN total_spent >= 2000 THEN 'VIP'
            WHEN total_spent >= 1000 THEN 'Premium'
            WHEN total_spent >= 500 THEN 'Regular'
            WHEN total_spent > 0 THEN 'Basic'
            ELSE 'No Orders'
        END as spending_tier,
        CASE
            WHEN order_count = 0 THEN 'Never Ordered'
            WHEN last_order_date >= CURRENT_DATE - INTERVAL '30 days' THEN
'Active'
            WHEN last_order_date >= CURRENT_DATE - INTERVAL '90 days' THEN
'Recent'
            WHEN last_order_date >= CURRENT_DATE - INTERVAL '180 days' THEN
'Dormant'
            ELSE 'Inactive'
        END as activity_status
    FROM customer_orders
),
segment_stats AS (
    SELECT
        spending_tier,
        activity_status,
        COUNT(*) as customer_count,
        AVG(total_spent) as avg_spending,
        AVG(order_count) as avg_orders,
        AVG(avg_order_value) as avg_order_value
```

```
    FROM customer_segments
    GROUP BY spending_tier, activity_status
)
SELECT
    cs.customer_id,
    cs.first_name,
    cs.last_name,
    cs.email,
    cs.city,
    cs.spending_tier,
    cs.activity_status,
    cs.order_count,
    cs.total_spent,
    cs.avg_order_value,
    cs.last_order_date,
    ss.customer_count as peers_in_segment,
    ss.avg_spending as segment_avg_spending,
    CASE
        WHEN cs.total_spent > ss.avg_spending THEN 'Above Segment Average'
        ELSE 'Below Segment Average'
    END as performance_vs_peers
FROM customer_segments cs
JOIN segment_stats ss ON cs.spending_tier = ss.spending_tier
    AND cs.activity_status = ss.activity_status
WHERE cs.order_count > 0
ORDER BY cs.total_spent DESC;
```

2. Recursive CTEs (PostgreSQL):

```
-- Hierarchical category structure (if categories have parent-child relationships)
WITH RECURSIVE category_hierarchy AS (
    -- Base case: top-level categories
    SELECT
        category_id,
        category_name,
        parent_category_id,
        0 as level,
        category_name as path
    FROM categories
    WHERE parent_category_id IS NULL

    UNION ALL

    -- Recursive case: child categories
    SELECT
        c.category_id,
        c.category_name,
        c.parent_category_id,
        ch.level + 1,
        ch.path || ' > ' || c.category_name
    FROM categories c
    JOIN category_hierarchy ch ON c.parent_category_id = ch.category_id
```

```
)
SELECT
    ch.category_id,
    ch.category_name,
    ch.level,
    ch.path,
    COUNT(p.product_id) as product_count,
    COALESCE(SUM(sales.revenue), 0) as total_revenue
FROM category_hierarchy ch
LEFT JOIN products p ON ch.category_id = p.category_id
    AND p.status = 'active'
LEFT JOIN (
    SELECT
        p.category_id,
        SUM(oi.quantity * oi.unit_price) as revenue
    FROM products p
    JOIN order_items oi ON p.product_id = oi.product_id
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY p.category_id
) sales ON ch.category_id = sales.category_id
GROUP BY ch.category_id, ch.category_name, ch.level, ch.path
ORDER BY ch.level, ch.path;
```

⚡ Performance Considerations

Optimization Strategies

1. Index Usage:

```
-- Ensure proper indexes for subquery performance
CREATE INDEX idx_orders_customer_status ON orders(customer_id, status);
CREATE INDEX idx_orders_date_status ON orders(order_date, status);
CREATE INDEX idx_order_items_product ON order_items(product_id);
CREATE INDEX idx_products_category_status ON products(category_id, status);
CREATE INDEX idx_reviews_product_rating ON reviews(product_id, rating);

-- Covering indexes for common subquery patterns
CREATE INDEX idx_orders_customer_date_amount ON orders(customer_id, order_date,
total_amount)
WHERE status = 'completed';
```

2. Subquery vs JOIN Performance:

```
-- Slower: Correlated subquery
SELECT
    c.customer_id,
```

```
    c.first_name,
    c.last_name,
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.customer_id = c.customer_id
          AND o.status = 'completed'
    ) as order_count
FROM customers c;

-- Faster: LEFT JOIN with GROUP BY
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
    AND o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name;
```

3. EXISTS vs IN Performance:

```
-- Generally faster: EXISTS
SELECT c.*
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.status = 'completed'
);

-- Can be slower: IN
SELECT c.*
FROM customers c
WHERE c.customer_id IN (
    SELECT o.customer_id
    FROM orders o
    WHERE o.status = 'completed'
);
```

💡 Real-World Examples

Example 1: Customer Churn Analysis

```
-- Identify customers at risk of churning
WITH customer_activity AS (
```

```
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
        COUNT(o.order_id) as total_orders,
        MAX(o.order_date) as last_order_date,
        AVG(EXTRACT(EPOCH FROM (o2.order_date - o1.order_date)) / 86400) as
avg_days_between_orders
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
        AND o.status = 'completed'
    LEFT JOIN orders o1 ON c.customer_id = o1.customer_id
        AND o1.status = 'completed'
    LEFT JOIN orders o2 ON c.customer_id = o2.customer_id
        AND o2.status = 'completed'
        AND o2.order_date > o1.order_date
        AND NOT EXISTS (
            SELECT 1
            FROM orders o3
            WHERE o3.customer_id = c.customer_id
              AND o3.status = 'completed'
              AND o3.order_date > o1.order_date
              AND o3.order_date < o2.order_date
        )
    WHERE c.status = 'active'
    GROUP BY c.customer_id, c.first_name, c.last_name, c.email
    HAVING COUNT(o.order_id) >= 2
)
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    total_orders,
    last_order_date,
    CURRENT_DATE - last_order_date as days_since_last_order,
    avg_days_between_orders,
    CASE
        WHEN (CURRENT_DATE - last_order_date) > (avg_days_between_orders * 2) THEN
'High Risk'
        WHEN (CURRENT_DATE - last_order_date) > (avg_days_between_orders * 1.5)
THEN 'Medium Risk'
        ELSE 'Low Risk'
    END as churn_risk
FROM customer_activity
WHERE avg_days_between_orders IS NOT NULL
ORDER BY days_since_last_order DESC;
```

Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between correlated and non-correlated subqueries?"**
    - Non-correlated: Independent, executes once
    - Correlated: References outer query, executes for each outer row
    - Performance implications and use cases
2. **"When would you use EXISTS vs IN?"**
    - EXISTS: Better for checking existence, handles NULLs safely
    - IN: Good for exact matches, but watch out for NULL issues
    - Performance considerations
3. **"How do you handle NULL values in NOT IN subqueries?"**
    - NOT IN with NULLs returns no results
    - Use NOT EXISTS or explicit NULL checks
    - Always filter out NULLs in subquery

Performance Best Practices:

1. **Prefer EXISTS over IN for existence checks**
2. **Use proper indexes on subquery columns**
3. **Consider JOINs instead of correlated subqueries**
4. **Use CTEs for complex, reusable logic**
5. **Limit subquery result sets when possible**

⚠ Things to Watch Out For

1. **NULL Handling in NOT IN**

```
-- Problem: Returns no results if subquery contains NULL
SELECT * FROM customers
WHERE customer_id NOT IN (
    SELECT customer_id FROM orders  -- If any customer_id is NULL
);

-- Solution: Filter NULLs or use NOT EXISTS
SELECT * FROM customers
WHERE customer_id NOT IN (
    SELECT customer_id FROM orders WHERE customer_id IS NOT NULL
);
```

2. **Correlated Subquery Performance**

```
-- Slow: Executes subquery for each row
SELECT c.*, (
```

```
    SELECT COUNT(*) FROM orders o
    WHERE o.customer_id = c.customer_id
) FROM customers c;

-- Better: Use JOIN
SELECT c.*, COUNT(o.order_id)
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id;
```

3. **Subquery Return Type Mismatch**

```
-- Error: Scalar subquery returns multiple rows
SELECT customer_id, (
    SELECT order_date FROM orders  -- Missing WHERE clause
) FROM customers;

-- Fix: Ensure single value
SELECT customer_id, (
    SELECT MAX(order_date) FROM orders
    WHERE customer_id = customers.customer_id
) FROM customers;
```

🚀 Next Steps

In the next chapter, we'll explore **Window Functions** - advanced analytical functions that allow you to perform calculations across related rows without grouping. You'll learn about ROW_NUMBER(), RANK(), LAG(), LEAD(), and how to create powerful analytical queries.

📝 Quick Practice

Try these subquery exercises:

1. **Scalar Subqueries**: Find products priced above their category average
2. **EXISTS**: Find customers who have ordered in multiple categories
3. **NOT EXISTS**: Find products never ordered by VIP customers
4. **Correlated**: Find each customer's most expensive order
5. **CTEs**: Create a customer segmentation analysis

Consider:

- When to use subqueries vs JOINs
- How to handle NULL values safely
- Performance implications of different approaches
- How to break complex problems into manageable parts

Chapter 10: Window Functions

What You'll Learn

Window functions are powerful analytical tools that allow you to perform calculations across a set of rows related to the current row, without collapsing the result set like aggregate functions do. They're essential for advanced data analysis, ranking, running totals, and comparative analytics.

Learning Objectives

By the end of this chapter, you will:

- Understand what window functions are and how they differ from aggregate functions
- Master ranking functions (ROW_NUMBER, RANK, DENSE_RANK)
- Use analytical functions (LAG, LEAD, FIRST_VALUE, LAST_VALUE)
- Create running totals and moving averages
- Apply window functions for business analytics
- Optimize window function performance

Concept Explanation

What are Window Functions?

Window functions perform calculations across a set of rows that are somehow related to the current row. Unlike aggregate functions that return a single value for a group of rows, window functions return a value for each row while maintaining the detail level of the original query.

Key Components:

- **OVER clause**: Defines the window of rows
- **PARTITION BY**: Divides rows into groups (like GROUP BY but doesn't collapse rows)
- **ORDER BY**: Defines the order within each partition
- **Frame specification**: Defines which rows within the partition to include

Basic Syntax:

```
function_name() OVER (
    [PARTITION BY column1, column2, ...]
    [ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...]
    [ROWS|RANGE frame_specification]
)
```

Syntax Comparison

MySQL vs PostgreSQL Window Functions

Both MySQL (8.0+) and PostgreSQL support comprehensive window functions with similar syntax:

Feature	MySQL 8.0+	PostgreSQL
Basic Window Functions	✅	✅
Ranking Functions	✅	✅
Analytical Functions	✅	✅
Frame Specifications	✅	✅
Named Windows	✅	✅
Advanced Frames	Limited	Full Support

Note: MySQL versions before 8.0 don't support window functions.

📊 Ranking Functions

ROW_NUMBER()

Assigns a unique sequential number to each row within a partition.

```sql
-- Basic row numbering
SELECT
    customer_id,
    first_name,
    last_name,
    order_date,
    total_amount,
    ROW_NUMBER() OVER (ORDER BY order_date) as order_sequence
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.status = 'completed'
ORDER BY order_date;

-- Row numbering within partitions
SELECT
    customer_id,
    first_name,
    last_name,
    order_date,
    total_amount,
    ROW_NUMBER() OVER (
        PARTITION BY customer_id
        ORDER BY order_date
    ) as customer_order_number
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.status = 'completed'
ORDER BY customer_id, order_date;
```

RANK() and DENSE_RANK()

RANK(): Assigns ranks with gaps for ties **DENSE_RANK()**: Assigns ranks without gaps for ties

```
-- Product ranking by sales within categories
SELECT
    p.product_id,
    p.product_name,
    c.category_name,
    sales.total_revenue,
    sales.units_sold,
    RANK() OVER (
        PARTITION BY c.category_name
        ORDER BY sales.total_revenue DESC
    ) as revenue_rank,
    DENSE_RANK() OVER (
        PARTITION BY c.category_name
        ORDER BY sales.total_revenue DESC
    ) as revenue_dense_rank,
    ROW_NUMBER() OVER (
        PARTITION BY c.category_name
        ORDER BY sales.total_revenue DESC
    ) as revenue_row_number
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN (
    SELECT
        oi.product_id,
        SUM(oi.quantity * oi.unit_price) as total_revenue,
        SUM(oi.quantity) as units_sold
    FROM order_items oi
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status IN ('completed', 'delivered')
      AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY oi.product_id
) sales ON p.product_id = sales.product_id
WHERE p.status = 'active'
ORDER BY c.category_name, sales.total_revenue DESC;
```

PERCENT_RANK() and CUME_DIST()

```
-- Customer spending percentiles
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    customer_stats.total_spent,
    customer_stats.order_count,
    PERCENT_RANK() OVER (ORDER BY customer_stats.total_spent) as
```

```
    spending_percentile,
        CUME_DIST() OVER (ORDER BY customer_stats.total_spent) as
    cumulative_distribution,
        CASE
            WHEN PERCENT_RANK() OVER (ORDER BY customer_stats.total_spent) >= 0.9 THEN
    'Top 10%'
            WHEN PERCENT_RANK() OVER (ORDER BY customer_stats.total_spent) >= 0.75
    THEN 'Top 25%'
            WHEN PERCENT_RANK() OVER (ORDER BY customer_stats.total_spent) >= 0.5 THEN
    'Top 50%'
            ELSE 'Bottom 50%'
        END as spending_tier
    FROM customers c
    JOIN (
        SELECT
            customer_id,
            COUNT(order_id) as order_count,
            SUM(total_amount) as total_spent
        FROM orders
        WHERE status = 'completed'
        GROUP BY customer_id
        HAVING COUNT(order_id) >= 1
    ) customer_stats ON c.customer_id = customer_stats.customer_id
    WHERE c.status = 'active'
    ORDER BY customer_stats.total_spent DESC;
```

☑ Analytical Functions

LAG() and LEAD()

Access data from previous or next rows within the same result set.

```
    -- Monthly sales comparison
    WITH monthly_sales AS (
        SELECT
            DATE_TRUNC('month', order_date) as month,
            COUNT(order_id) as order_count,
            SUM(total_amount) as total_revenue,
            AVG(total_amount) as avg_order_value
        FROM orders
        WHERE status = 'completed'
          AND order_date >= CURRENT_DATE - INTERVAL '24 months'
        GROUP BY DATE_TRUNC('month', order_date)
        ORDER BY month
    )
    SELECT
        month,
        order_count,
        total_revenue,
        avg_order_value,
```

```
    LAG(total_revenue, 1) OVER (ORDER BY month) as prev_month_revenue,
    LEAD(total_revenue, 1) OVER (ORDER BY month) as next_month_revenue,
    total_revenue - LAG(total_revenue, 1) OVER (ORDER BY month) as revenue_change,
    ROUND(
        ((total_revenue - LAG(total_revenue, 1) OVER (ORDER BY month)) /
         NULLIF(LAG(total_revenue, 1) OVER (ORDER BY month), 0)) * 100, 2
    ) as revenue_change_percent,
    LAG(total_revenue, 12) OVER (ORDER BY month) as same_month_last_year,
    CASE
        WHEN LAG(total_revenue, 12) OVER (ORDER BY month) IS NOT NULL THEN
            ROUND(
                ((total_revenue - LAG(total_revenue, 12) OVER (ORDER BY month)) /
                 LAG(total_revenue, 12) OVER (ORDER BY month)) * 100, 2
            )
        ELSE NULL
    END as yoy_growth_percent
FROM monthly_sales
ORDER BY month;
```

FIRST_VALUE() and LAST_VALUE()

```
-- Customer order analysis with first and last orders
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    o.order_id,
    o.order_date,
    o.total_amount,
    FIRST_VALUE(o.order_date) OVER (
        PARTITION BY c.customer_id
        ORDER BY o.order_date
        ROWS UNBOUNDED PRECEDING
    ) as first_order_date,
    FIRST_VALUE(o.total_amount) OVER (
        PARTITION BY c.customer_id
        ORDER BY o.order_date
        ROWS UNBOUNDED PRECEDING
    ) as first_order_amount,
    LAST_VALUE(o.order_date) OVER (
        PARTITION BY c.customer_id
        ORDER BY o.order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as last_order_date,
    LAST_VALUE(o.total_amount) OVER (
        PARTITION BY c.customer_id
        ORDER BY o.order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as last_order_amount,
    ROW_NUMBER() OVER (
        PARTITION BY c.customer_id
```

```
        ORDER BY o.order_date
    ) as order_sequence
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.status = 'completed'
  AND c.status = 'active'
ORDER BY c.customer_id, o.order_date;
```

🔢 Running Totals and Moving Averages

Running Totals

```
-- Daily sales with running totals
WITH daily_sales AS (
    SELECT
        DATE(order_date) as sale_date,
        COUNT(order_id) as daily_orders,
        SUM(total_amount) as daily_revenue
    FROM orders
    WHERE status = 'completed'
      AND order_date >= CURRENT_DATE - INTERVAL '90 days'
    GROUP BY DATE(order_date)
    ORDER BY sale_date
)
SELECT
    sale_date,
    daily_orders,
    daily_revenue,
    SUM(daily_orders) OVER (
        ORDER BY sale_date
        ROWS UNBOUNDED PRECEDING
    ) as running_total_orders,
    SUM(daily_revenue) OVER (
        ORDER BY sale_date
        ROWS UNBOUNDED PRECEDING
    ) as running_total_revenue,
    AVG(daily_revenue) OVER (
        ORDER BY sale_date
        ROWS UNBOUNDED PRECEDING
    ) as cumulative_avg_revenue
FROM daily_sales
ORDER BY sale_date;
```

Moving Averages

```
-- 7-day and 30-day moving averages
WITH daily_sales AS (
    SELECT
```

```
        DATE(order_date) as sale_date,
        COUNT(order_id) as daily_orders,
        SUM(total_amount) as daily_revenue
    FROM orders
    WHERE status = 'completed'
      AND order_date >= CURRENT_DATE - INTERVAL '90 days'
    GROUP BY DATE(order_date)
    ORDER BY sale_date
)
SELECT
    sale_date,
    daily_orders,
    daily_revenue,
    AVG(daily_revenue) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as seven_day_avg,
    AVG(daily_revenue) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
    ) as thirty_day_avg,
    SUM(daily_revenue) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as seven_day_total,
    COUNT(*) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as days_in_7day_window
FROM daily_sales
ORDER BY sale_date;
```

Frame Specifications

Understanding Window Frames

Window frames define which rows within the partition to include in the calculation:

```
-- Different frame specifications
SELECT
    order_date,
    total_amount,
    -- Default frame: RANGE UNBOUNDED PRECEDING
    SUM(total_amount) OVER (ORDER BY order_date) as running_total_default,

    -- Explicit ROWS frame
    SUM(total_amount) OVER (
        ORDER BY order_date
        ROWS UNBOUNDED PRECEDING
    ) as running_total_rows,
```

```
    -- Moving window: last 3 rows including current
    SUM(total_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) as last_3_orders_total,

    -- Centered window: 1 before, current, 1 after
    AVG(total_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) as centered_avg,

    -- Future window: current and next 2 rows
    MAX(total_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING
    ) as max_next_3_orders
FROM orders
WHERE status = 'completed'
  AND customer_id = 1
ORDER BY order_date;
```

ROWS vs RANGE

```
-- Demonstrating ROWS vs RANGE with duplicate dates
SELECT
    order_date,
    total_amount,
    order_id,
    -- ROWS: Physical row-based window
    SUM(total_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) as rows_sum,

    -- RANGE: Logical value-based window (includes all rows with same order_date)
    SUM(total_amount) OVER (
        ORDER BY order_date
        RANGE BETWEEN INTERVAL '1 day' PRECEDING AND INTERVAL '1 day' FOLLOWING
    ) as range_sum
FROM orders
WHERE status = 'completed'
  AND order_date >= '2024-01-01'
  AND order_date <= '2024-01-10'
ORDER BY order_date, order_id;
```

🏢 Real-World Business Examples

Example 1: Sales Performance Dashboard

```sql
-- Comprehensive sales performance analysis
WITH sales_data AS (
    SELECT
        DATE_TRUNC('month', o.order_date) as month,
        c.city,
        c.state,
        COUNT(o.order_id) as orders,
        SUM(o.total_amount) as revenue,
        COUNT(DISTINCT o.customer_id) as unique_customers,
        AVG(o.total_amount) as avg_order_value
    FROM orders o
    JOIN customers c ON o.customer_id = c.customer_id
    WHERE o.status = 'completed'
      AND o.order_date >= CURRENT_DATE - INTERVAL '24 months'
    GROUP BY DATE_TRUNC('month', o.order_date), c.city, c.state
),
ranked_cities AS (
    SELECT
        month,
        city,
        state,
        orders,
        revenue,
        unique_customers,
        avg_order_value,
        -- Ranking within each month
        RANK() OVER (
            PARTITION BY month
            ORDER BY revenue DESC
        ) as monthly_revenue_rank,

        -- Running totals for each city
        SUM(revenue) OVER (
            PARTITION BY city, state
            ORDER BY month
            ROWS UNBOUNDED PRECEDING
        ) as cumulative_revenue,

        -- Month-over-month comparison
        LAG(revenue, 1) OVER (
            PARTITION BY city, state
            ORDER BY month
        ) as prev_month_revenue,

        -- 3-month moving average
        AVG(revenue) OVER (
            PARTITION BY city, state
            ORDER BY month
            ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
        ) as three_month_avg,
```

```
        -- Year-over-year comparison
        LAG(revenue, 12) OVER (
            PARTITION BY city, state
            ORDER BY month
        ) as same_month_last_year
    FROM sales_data
)
SELECT
    month,
    city,
    state,
    orders,
    revenue,
    unique_customers,
    avg_order_value,
    monthly_revenue_rank,
    cumulative_revenue,
    prev_month_revenue,
    revenue - prev_month_revenue as mom_change,
    CASE
        WHEN prev_month_revenue > 0 THEN
            ROUND(((revenue - prev_month_revenue) / prev_month_revenue) * 100, 2)
        ELSE NULL
    END as mom_change_percent,
    three_month_avg,
    same_month_last_year,
    CASE
        WHEN same_month_last_year > 0 THEN
            ROUND(((revenue - same_month_last_year) / same_month_last_year) * 100,
2)
        ELSE NULL
    END as yoy_change_percent
FROM ranked_cities
WHERE monthly_revenue_rank <= 10  -- Top 10 cities by revenue each month
ORDER BY month DESC, monthly_revenue_rank;
```

Example 2: Customer Lifecycle Analysis

```
-- Customer journey and lifecycle analysis
WITH customer_orders AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
        c.created_at as registration_date,
        o.order_id,
        o.order_date,
        o.total_amount,
        ROW_NUMBER() OVER (
```

```
            PARTITION BY c.customer_id
            ORDER BY o.order_date
        ) as order_number
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
      AND c.status = 'active'
),
customer_metrics AS (
    SELECT
        customer_id,
        first_name,
        last_name,
        email,
        registration_date,
        order_id,
        order_date,
        total_amount,
        order_number,

        -- First order details
        FIRST_VALUE(order_date) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
            ROWS UNBOUNDED PRECEDING
        ) as first_order_date,

        FIRST_VALUE(total_amount) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
            ROWS UNBOUNDED PRECEDING
        ) as first_order_amount,

        -- Running customer totals
        SUM(total_amount) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
            ROWS UNBOUNDED PRECEDING
        ) as cumulative_spent,

        -- Average order value up to this point
        AVG(total_amount) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
            ROWS UNBOUNDED PRECEDING
        ) as avg_order_value_to_date,

        -- Days between orders
        LAG(order_date, 1) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
        ) as prev_order_date,

        -- Time to next order
```

```
        LEAD(order_date, 1) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
        ) as next_order_date,

        -- Last order in sequence
        LAST_VALUE(order_date) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
        ) as last_order_date,

        -- Total orders for customer
        COUNT(*) OVER (
            PARTITION BY customer_id
        ) as total_orders
    FROM customer_orders
)
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    registration_date,
    order_number,
    order_date,
    total_amount,
    first_order_date,
    first_order_amount,
    cumulative_spent,
    avg_order_value_to_date,

    -- Days since registration to first order
    CASE
        WHEN order_number = 1 THEN
            order_date - registration_date
        ELSE NULL
    END as days_to_first_order,

    -- Days between consecutive orders
    CASE
        WHEN prev_order_date IS NOT NULL THEN
            order_date - prev_order_date
        ELSE NULL
    END as days_since_last_order,

    -- Customer lifecycle stage
    CASE
        WHEN order_number = 1 THEN 'New Customer'
        WHEN order_number = 2 THEN 'Repeat Customer'
        WHEN order_number >= 3 AND order_number <= 5 THEN 'Regular Customer'
        WHEN order_number > 5 THEN 'Loyal Customer'
    END as lifecycle_stage,
```

```
    -- Customer value tier based on cumulative spending
    CASE
        WHEN cumulative_spent >= 2000 THEN 'VIP'
        WHEN cumulative_spent >= 1000 THEN 'Premium'
        WHEN cumulative_spent >= 500 THEN 'Regular'
        ELSE 'Basic'
    END as value_tier,

    total_orders,
    last_order_date,
    CURRENT_DATE - last_order_date as days_since_last_order_overall
FROM customer_metrics
ORDER BY customer_id, order_number;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between RANK() and DENSE_RANK()?"**
    - RANK() leaves gaps after ties (1, 2, 2, 4)
    - DENSE_RANK() doesn't leave gaps (1, 2, 2, 3)
    - ROW_NUMBER() assigns unique numbers regardless of ties
2. **"How do you calculate a running total?"**
    - Use SUM() with ROWS UNBOUNDED PRECEDING
    - Explain frame specifications
    - Discuss performance considerations
3. **"When would you use LAG() vs LEAD()?"**
    - LAG(): Compare with previous values (month-over-month growth)
    - LEAD(): Look ahead to future values (forecasting)
    - Both useful for time series analysis
4. **"How do you handle NULL values in window functions?"**
    - Most window functions ignore NULLs by default
    - Use COALESCE() or CASE statements for explicit handling
    - Consider IGNORE NULLS clause where supported

Performance Best Practices:

1. **Use appropriate indexes for PARTITION BY and ORDER BY columns**
2. **Limit the window frame size when possible**
3. **Consider using named windows for repeated window specifications**
4. **Be careful with LAST_VALUE() - specify proper frame bounds**
5. **Use LIMIT with window functions for large datasets**

⚠ Things to Watch Out For

1. **LAST_VALUE() Default Frame**

```
-- Problem: Default frame only goes to current row
SELECT
    order_date,
    total_amount,
    LAST_VALUE(total_amount) OVER (ORDER BY order_date) as last_value_wrong
FROM orders;

-- Solution: Specify proper frame
SELECT
    order_date,
    total_amount,
    LAST_VALUE(total_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as last_value_correct
FROM orders;
```

2. **Performance with Large Datasets**

```
-- Slow: No indexes on window columns
SELECT
    customer_id,
    order_date,
    ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date)
FROM orders;  -- Millions of rows

-- Better: Create appropriate indexes
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
```

3. **NULL Handling in Analytical Functions**

```
-- Be aware of NULL behavior
SELECT
    order_date,
    total_amount,
    LAG(total_amount, 1) OVER (ORDER BY order_date) as prev_amount,
    LAG(total_amount, 1, 0) OVER (ORDER BY order_date) as prev_amount_default_0
FROM orders
WHERE customer_id = 1;
```

🚀 Next Steps

In the next chapter, we'll explore **Indexes and Query Optimization** - crucial topics for improving database performance. You'll learn about different index types, how to analyze query execution plans, and optimization strategies for complex queries.

📝 Quick Practice

Try these window function exercises:

1. **Ranking**: Find the top 3 products by sales in each category
2. **Running Totals**: Calculate cumulative sales by month
3. **Moving Averages**: Create 7-day moving average of daily orders
4. **Analytical Functions**: Find customers who increased their order frequency
5. **Complex Analysis**: Build a customer churn prediction model using window functions

Consider:

- When to use different ranking functions
- How frame specifications affect results
- Performance implications of window functions
- Real-world applications in business analytics

Chapter 11: Indexes and Query Optimization

📚 What You'll Learn

Indexes are database objects that improve query performance by creating shortcuts to data. Understanding how to create, use, and optimize indexes is crucial for building high-performance database applications. This chapter covers index types, query optimization techniques, and performance tuning strategies.

🎯 Learning Objectives

By the end of this chapter, you will:

- Understand different types of indexes and when to use them
- Know how to analyze query execution plans
- Master index creation and optimization strategies
- Learn query optimization techniques
- Understand the trade-offs between query performance and storage
- Apply performance tuning best practices

🔍 Concept Explanation

What are Indexes?

An index is a data structure that improves the speed of data retrieval operations on a database table. Think of it like an index in a book - instead of reading every page to find a topic, you can use the index to jump

directly to the relevant pages.

Key Benefits:

- Faster SELECT queries
- Faster WHERE clause filtering
- Faster ORDER BY operations
- Faster JOIN operations

Trade-offs:

- Additional storage space
- Slower INSERT/UPDATE/DELETE operations
- Maintenance overhead

🛠️ Index Types Comparison

MySQL vs PostgreSQL Index Types

Index Type	MySQL	PostgreSQL	Use Case
B-Tree	✅ (Default)	✅ (Default)	General purpose, range queries
Hash	✅ (Memory engine)	✅	Equality comparisons only
Full-Text	✅	✅	Text search
Spatial	✅	✅ (PostGIS)	Geographic data
Partial	❌	✅	Conditional indexing
Expression	❌	✅	Function-based indexing
GIN	❌	✅	Array, JSON, full-text
GiST	❌	✅	Complex data types
Covering	✅ (5.7+)	✅	Include non-key columns

📊 Basic Index Operations

Creating Indexes

MySQL:

```
-- Basic index creation
CREATE INDEX idx_customer_email ON customers(email);
CREATE INDEX idx_order_date ON orders(order_date);
CREATE INDEX idx_product_category ON products(category_id);

-- Composite index
CREATE INDEX idx_order_customer_date ON orders(customer_id, order_date);
```

```
-- Unique index
CREATE UNIQUE INDEX idx_customer_email_unique ON customers(email);

-- Covering index (MySQL 5.7+)
ALTER TABLE orders ADD INDEX idx_customer_covering (customer_id) INCLUDE
(order_date, total_amount);

-- Full-text index
CREATE FULLTEXT INDEX idx_product_description ON products(product_name,
description);
```

PostgreSQL:

```
-- Basic index creation
CREATE INDEX idx_customer_email ON customers(email);
CREATE INDEX idx_order_date ON orders(order_date);
CREATE INDEX idx_product_category ON products(category_id);

-- Composite index
CREATE INDEX idx_order_customer_date ON orders(customer_id, order_date);

-- Unique index
CREATE UNIQUE INDEX idx_customer_email_unique ON customers(email);

-- Partial index (PostgreSQL only)
CREATE INDEX idx_active_customers ON customers(email) WHERE status = 'active';

-- Expression index (PostgreSQL only)
CREATE INDEX idx_customer_lower_email ON customers(LOWER(email));

-- Covering index
CREATE INDEX idx_customer_covering ON orders(customer_id) INCLUDE (order_date,
total_amount);

-- GIN index for JSON
CREATE INDEX idx_product_attributes ON products USING GIN(attributes);

-- Full-text index
CREATE INDEX idx_product_search ON products USING GIN(to_tsvector('english',
product_name || ' ' || description));
```

Viewing and Managing Indexes

MySQL:

```
-- Show indexes for a table
SHOW INDEXES FROM orders;
```

```sql
-- Show index usage statistics
SELECT
    TABLE_SCHEMA,
    TABLE_NAME,
    INDEX_NAME,
    COLUMN_NAME,
    CARDINALITY
FROM INFORMATION_SCHEMA.STATISTICS
WHERE TABLE_SCHEMA = 'your_database'
  AND TABLE_NAME = 'orders';

-- Drop index
DROP INDEX idx_order_date ON orders;
```

PostgreSQL:

```sql
-- Show indexes for a table
\d+ orders

-- Query index information
SELECT
    schemaname,
    tablename,
    indexname,
    indexdef
FROM pg_indexes
WHERE tablename = 'orders';

-- Index usage statistics
SELECT
    schemaname,
    tablename,
    indexname,
    idx_tup_read,
    idx_tup_fetch
FROM pg_stat_user_indexes
WHERE tablename = 'orders';

-- Drop index
DROP INDEX idx_order_date;
```

🔍 Query Execution Plans

Understanding EXPLAIN

MySQL:

```sql
-- Basic EXPLAIN
EXPLAIN SELECT * FROM orders WHERE customer_id = 123;

-- Detailed execution plan
EXPLAIN FORMAT=JSON
SELECT
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count,
    SUM(o.total_amount) as total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE c.status = 'active'
  AND o.order_date >= '2024-01-01'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(o.order_id) > 5;

-- Analyze actual execution
EXPLAIN ANALYZE
SELECT * FROM orders
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31'
ORDER BY total_amount DESC
LIMIT 100;
```

PostgreSQL:

```sql
-- Basic EXPLAIN
EXPLAIN SELECT * FROM orders WHERE customer_id = 123;

-- Detailed execution plan
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
SELECT
    c.first_name,
    c.last_name,
    COUNT(o.order_id) as order_count,
    SUM(o.total_amount) as total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE c.status = 'active'
  AND o.order_date >= '2024-01-01'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(o.order_id) > 5;

-- Visual execution plan
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT * FROM orders
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31'
ORDER BY total_amount DESC
LIMIT 100;
```

Reading Execution Plans

Key Metrics to Watch:

- **Cost**: Estimated query cost (PostgreSQL)
- **Rows**: Number of rows processed
- **Time**: Actual execution time
- **Access Method**: How data is accessed (Index Scan, Seq Scan, etc.)
- **Join Type**: How tables are joined (Nested Loop, Hash Join, etc.)

```
-- Example: Analyzing a slow query
EXPLAIN ANALYZE
SELECT
    p.product_name,
    c.category_name,
    SUM(oi.quantity) as total_sold,
    SUM(oi.quantity * oi.unit_price) as revenue
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE o.order_date >= '2024-01-01'
  AND o.status = 'completed'
  AND p.status = 'active'
GROUP BY p.product_id, p.product_name, c.category_name
HAVING SUM(oi.quantity) > 100
ORDER BY revenue DESC;
```

⚡ Index Optimization Strategies

1. Composite Index Column Order

```
-- Rule: Most selective columns first, then by query patterns

-- Good: Selective customer_id first, then date range
CREATE INDEX idx_orders_customer_date_status ON orders(customer_id, order_date,
status);

-- Query that benefits from this index
SELECT * FROM orders
WHERE customer_id = 123
  AND order_date >= '2024-01-01'
  AND status = 'completed';

-- Also benefits from leftmost prefix
SELECT * FROM orders WHERE customer_id = 123;
SELECT * FROM orders WHERE customer_id = 123 AND order_date >= '2024-01-01';
```

```
-- Won't use the index efficiently
SELECT * FROM orders WHERE order_date >= '2024-01-01';  -- Skips customer_id
SELECT * FROM orders WHERE status = 'completed';        -- Skips customer_id and
order_date
```

2. Covering Indexes

```
-- Include frequently accessed columns in the index
CREATE INDEX idx_orders_covering ON orders(customer_id, order_date)
INCLUDE (total_amount, status);

-- This query can be satisfied entirely from the index
SELECT customer_id, order_date, total_amount, status
FROM orders
WHERE customer_id = 123
  AND order_date >= '2024-01-01';
```

3. Partial Indexes (PostgreSQL)

```
-- Index only active records
CREATE INDEX idx_active_customers_email ON customers(email)
WHERE status = 'active';

-- Index only recent orders
CREATE INDEX idx_recent_orders ON orders(customer_id, order_date)
WHERE order_date >= CURRENT_DATE - INTERVAL '1 year';

-- Index only high-value orders
CREATE INDEX idx_high_value_orders ON orders(customer_id, order_date)
WHERE total_amount >= 1000;
```

4. Expression Indexes (PostgreSQL)

```
-- Index on function results
CREATE INDEX idx_customer_lower_email ON customers(LOWER(email));

-- Query that uses the expression index
SELECT * FROM customers WHERE LOWER(email) = 'john@example.com';

-- Index on calculated values
CREATE INDEX idx_order_year_month ON orders(EXTRACT(YEAR FROM order_date),
EXTRACT(MONTH FROM order_date));

-- Query using the expression index
SELECT * FROM orders
```

```sql
WHERE EXTRACT(YEAR FROM order_date) = 2024
  AND EXTRACT(MONTH FROM order_date) = 6;
```

🚀 Query Optimization Techniques

1. WHERE Clause Optimization

```sql
-- Bad: Non-sargable queries (can't use indexes effectively)
SELECT * FROM orders WHERE YEAR(order_date) = 2024;
SELECT * FROM customers WHERE UPPER(first_name) = 'JOHN';
SELECT * FROM products WHERE price * 1.1 > 100;

-- Good: Sargable queries (can use indexes)
SELECT * FROM orders
WHERE order_date >= '2024-01-01'
  AND order_date < '2025-01-01';

SELECT * FROM customers WHERE first_name = 'John';
SELECT * FROM products WHERE price > 100 / 1.1;

-- Use EXISTS instead of IN for better performance
-- Bad: IN with subquery
SELECT * FROM customers
WHERE customer_id IN (
    SELECT customer_id FROM orders
    WHERE order_date >= '2024-01-01'
);

-- Good: EXISTS
SELECT * FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.order_date >= '2024-01-01'
);
```

2. JOIN Optimization

```sql
-- Ensure proper indexes on JOIN columns
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
CREATE INDEX idx_order_items_order_id ON order_items(order_id);
CREATE INDEX idx_order_items_product_id ON order_items(product_id);

-- Use appropriate JOIN types
-- INNER JOIN when you only want matching records
SELECT c.first_name, c.last_name, COUNT(o.order_id)
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
```

```sql
WHERE o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name;

-- LEFT JOIN when you want all records from left table
SELECT c.first_name, c.last_name, COALESCE(COUNT(o.order_id), 0)
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
  AND o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name;
```

3. LIMIT and Pagination Optimization

```sql
-- Bad: OFFSET becomes slow with large offsets
SELECT * FROM orders
ORDER BY order_date DESC
LIMIT 20 OFFSET 10000;  -- Slow for large offsets

-- Good: Cursor-based pagination
SELECT * FROM orders
WHERE order_date < '2024-06-15 10:30:00'  -- Last order_date from previous page
ORDER BY order_date DESC
LIMIT 20;

-- Even better: Use indexed column for cursor
SELECT * FROM orders
WHERE order_id < 12345  -- Last order_id from previous page
ORDER BY order_id DESC
LIMIT 20;
```

4. Subquery Optimization

```sql
-- Convert correlated subqueries to JOINs when possible
-- Bad: Correlated subquery
SELECT c.*, (
    SELECT COUNT(*)
    FROM orders o
    WHERE o.customer_id = c.customer_id
      AND o.status = 'completed'
) as order_count
FROM customers c;

-- Good: JOIN with GROUP BY
SELECT
    c.*,
    COALESCE(order_stats.order_count, 0) as order_count
FROM customers c
LEFT JOIN (
    SELECT
        customer_id,
```

```
        COUNT(*) as order_count
    FROM orders
    WHERE status = 'completed'
    GROUP BY customer_id
) order_stats ON c.customer_id = order_stats.customer_id;
```

📊 Performance Monitoring

MySQL Performance Monitoring

```
-- Enable slow query log
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL long_query_time = 2;  -- Log queries taking more than 2 seconds

-- Check index usage
SELECT
    OBJECT_SCHEMA,
    OBJECT_NAME,
    INDEX_NAME,
    COUNT_FETCH,
    COUNT_INSERT,
    COUNT_UPDATE,
    COUNT_DELETE
FROM performance_schema.table_io_waits_summary_by_index_usage
WHERE OBJECT_SCHEMA = 'your_database'
ORDER BY COUNT_FETCH DESC;

-- Find unused indexes
SELECT
    OBJECT_SCHEMA,
    OBJECT_NAME,
    INDEX_NAME
FROM performance_schema.table_io_waits_summary_by_index_usage
WHERE OBJECT_SCHEMA = 'your_database'
  AND INDEX_NAME IS NOT NULL
  AND COUNT_FETCH = 0
  AND COUNT_INSERT = 0
  AND COUNT_UPDATE = 0
  AND COUNT_DELETE = 0;

-- Query performance statistics
SELECT
    DIGEST_TEXT,
    COUNT_STAR,
    AVG_TIMER_WAIT/1000000000 as avg_time_seconds,
    MAX_TIMER_WAIT/1000000000 as max_time_seconds
FROM performance_schema.events_statements_summary_by_digest
WHERE DIGEST_TEXT IS NOT NULL
ORDER BY AVG_TIMER_WAIT DESC
LIMIT 10;
```

PostgreSQL Performance Monitoring

```
-- Enable query statistics
-- Add to postgresql.conf:
-- shared_preload_libraries = 'pg_stat_statements'
-- pg_stat_statements.track = all

-- Top slow queries
SELECT
    query,
    calls,
    total_time,
    mean_time,
    rows
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 10;

-- Index usage statistics
SELECT
    schemaname,
    tablename,
    indexname,
    idx_tup_read,
    idx_tup_fetch,
    idx_scan
FROM pg_stat_user_indexes
ORDER BY idx_scan DESC;

-- Unused indexes
SELECT
    schemaname,
    tablename,
    indexname,
    idx_scan,
    pg_size_pretty(pg_relation_size(indexrelid)) as size
FROM pg_stat_user_indexes
WHERE idx_scan = 0
  AND schemaname = 'public'
ORDER BY pg_relation_size(indexrelid) DESC;

-- Table and index sizes
SELECT
    tablename,
    pg_size_pretty(pg_total_relation_size(tablename::regclass)) as total_size,
    pg_size_pretty(pg_relation_size(tablename::regclass)) as table_size,
    pg_size_pretty(pg_total_relation_size(tablename::regclass) -
pg_relation_size(tablename::regclass)) as index_size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(tablename::regclass) DESC;
```

Real-World Optimization Examples

Example 1: E-commerce Product Search Optimization

```
-- Problem: Slow product search query
-- Original slow query
SELECT
    p.product_id,
    p.product_name,
    p.price,
    c.category_name,
    AVG(r.rating) as avg_rating,
    COUNT(r.review_id) as review_count
FROM products p
JOIN categories c ON p.category_id = c.category_id
LEFT JOIN reviews r ON p.product_id = r.product_id
WHERE p.product_name ILIKE '%laptop%'
  AND p.status = 'active'
  AND p.price BETWEEN 500 AND 2000
GROUP BY p.product_id, p.product_name, p.price, c.category_name
HAVING AVG(r.rating) >= 4.0
ORDER BY avg_rating DESC, review_count DESC
LIMIT 20;

-- Optimization steps:

-- 1. Create appropriate indexes
CREATE INDEX idx_products_status_price ON products(status, price);
CREATE INDEX idx_products_name_gin ON products USING GIN(to_tsvector('english',
product_name));  -- PostgreSQL
CREATE FULLTEXT INDEX idx_products_name_fulltext ON products(product_name);  --
MySQL
CREATE INDEX idx_reviews_product_rating ON reviews(product_id, rating);

-- 2. Optimized query with better structure
WITH product_search AS (
    SELECT
        p.product_id,
        p.product_name,
        p.price,
        c.category_name
    FROM products p
    JOIN categories c ON p.category_id = c.category_id
    WHERE p.status = 'active'
      AND p.price BETWEEN 500 AND 2000
      AND (
          -- PostgreSQL full-text search
          to_tsvector('english', p.product_name) @@ to_tsquery('english',
'laptop')
          -- OR MySQL full-text search
```

```
            -- MATCH(p.product_name) AGAINST('laptop' IN NATURAL LANGUAGE MODE)
        )
),
product_ratings AS (
    SELECT
        product_id,
        AVG(rating) as avg_rating,
        COUNT(*) as review_count
    FROM reviews
    WHERE product_id IN (SELECT product_id FROM product_search)
    GROUP BY product_id
    HAVING AVG(rating) >= 4.0
)
SELECT
    ps.product_id,
    ps.product_name,
    ps.price,
    ps.category_name,
    COALESCE(pr.avg_rating, 0) as avg_rating,
    COALESCE(pr.review_count, 0) as review_count
FROM product_search ps
LEFT JOIN product_ratings pr ON ps.product_id = pr.product_id
ORDER BY pr.avg_rating DESC NULLS LAST, pr.review_count DESC NULLS LAST
LIMIT 20;
```

Example 2: Customer Analytics Dashboard Optimization

```
-- Problem: Slow customer analytics query
-- Original query taking 30+ seconds
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    COUNT(o.order_id) as total_orders,
    SUM(o.total_amount) as total_spent,
    AVG(o.total_amount) as avg_order_value,
    MAX(o.order_date) as last_order_date,
    COUNT(DISTINCT EXTRACT(MONTH FROM o.order_date)) as active_months
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
  AND o.status = 'completed'
  AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
WHERE c.status = 'active'
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
ORDER BY total_spent DESC;

-- Optimization approach:

-- 1. Create materialized view or summary table (PostgreSQL)
CREATE MATERIALIZED VIEW customer_analytics_12m AS
```

```sql
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    c.status,
    COALESCE(order_stats.total_orders, 0) as total_orders,
    COALESCE(order_stats.total_spent, 0) as total_spent,
    COALESCE(order_stats.avg_order_value, 0) as avg_order_value,
    order_stats.last_order_date,
    COALESCE(order_stats.active_months, 0) as active_months,
    CURRENT_DATE as last_updated
FROM customers c
LEFT JOIN (
    SELECT
        customer_id,
        COUNT(order_id) as total_orders,
        SUM(total_amount) as total_spent,
        AVG(total_amount) as avg_order_value,
        MAX(order_date) as last_order_date,
        COUNT(DISTINCT EXTRACT(MONTH FROM order_date)) as active_months
    FROM orders
    WHERE status = 'completed'
      AND order_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY customer_id
) order_stats ON c.customer_id = order_stats.customer_id;

-- Create index on materialized view
CREATE INDEX idx_customer_analytics_spent ON customer_analytics_12m(total_spent
DESC);
CREATE INDEX idx_customer_analytics_orders ON customer_analytics_12m(total_orders
DESC);

-- Refresh materialized view (can be automated)
REFRESH MATERIALIZED VIEW customer_analytics_12m;

-- Fast query using materialized view
SELECT *
FROM customer_analytics_12m
WHERE status = 'active'
ORDER BY total_spent DESC
LIMIT 100;

-- 2. Alternative: Create summary table with triggers (MySQL/PostgreSQL)
CREATE TABLE customer_summary (
    customer_id INT PRIMARY KEY,
    total_orders INT DEFAULT 0,
    total_spent DECIMAL(10,2) DEFAULT 0,
    avg_order_value DECIMAL(10,2) DEFAULT 0,
    last_order_date DATE,
    active_months INT DEFAULT 0,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Create indexes
CREATE INDEX idx_customer_summary_spent ON customer_summary(total_spent DESC);
CREATE INDEX idx_customer_summary_updated ON customer_summary(last_updated);
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"How do you identify slow queries?"**
    - Enable slow query logs
    - Use performance monitoring tools
    - Analyze execution plans
    - Monitor database metrics
2. **"When would you use a composite index vs multiple single-column indexes?"**
    - Composite: When queries filter on multiple columns together
    - Single: When queries typically filter on one column at a time
    - Consider query patterns and selectivity
3. **"How do you decide which columns to include in a covering index?"**
    - Include frequently accessed columns in SELECT clause
    - Balance between query performance and index size
    - Consider update frequency of included columns
4. **"What's the difference between clustered and non-clustered indexes?"**
    - Clustered: Data pages stored in order of index key (one per table)
    - Non-clustered: Separate structure pointing to data pages
    - MySQL InnoDB: Primary key is clustered index

Performance Tuning Best Practices:

1. **Start with proper indexing strategy**
2. **Monitor and analyze query patterns**
3. **Use appropriate data types**
4. **Normalize appropriately (don't over-normalize)**
5. **Consider partitioning for very large tables**
6. **Regular maintenance (ANALYZE, VACUUM, etc.)**

⚠️ Things to Watch Out For

1. **Over-Indexing**

```
-- Problem: Too many indexes slow down writes
-- Don't create indexes for every possible query
```

```
-- Bad: Redundant indexes
CREATE INDEX idx1 ON orders(customer_id);
CREATE INDEX idx2 ON orders(customer_id, order_date);  -- idx1 is redundant

-- Good: Use composite index that serves multiple purposes
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
```

2. **Wrong Column Order in Composite Indexes**

```
-- Bad: Low selectivity column first
CREATE INDEX idx_bad ON orders(status, customer_id);  -- status has few values

-- Good: High selectivity column first
CREATE INDEX idx_good ON orders(customer_id, status);  -- customer_id is more
selective
```

3. **Ignoring Index Maintenance**

```
-- PostgreSQL: Regular maintenance
ANALYZE;  -- Update statistics
VACUUM;   -- Reclaim space
REINDEX;  -- Rebuild indexes if needed

-- MySQL: Regular maintenance
ANALYZE TABLE orders;
OPTIMIZE TABLE orders;  -- For MyISAM tables
```

4. **Not Considering Query Patterns**

```
-- If most queries are:
SELECT * FROM orders WHERE customer_id = ? AND status = 'completed';

-- Create index matching the query pattern:
CREATE INDEX idx_orders_customer_status ON orders(customer_id, status);

-- Not:
CREATE INDEX idx_orders_status_customer ON orders(status, customer_id);
```

🚀 Next Steps

In the next chapter, we'll explore **Stored Procedures and Functions** - reusable code blocks that can encapsulate business logic, improve performance, and provide better security. You'll learn how to create, manage, and optimize stored procedures and functions.

Quick Practice

Try these optimization exercises:

1. **Index Analysis**: Analyze a slow query and create appropriate indexes
2. **Execution Plans**: Compare execution plans before and after optimization
3. **Composite Indexes**: Design optimal composite indexes for complex queries
4. **Performance Monitoring**: Set up monitoring for index usage and query performance
5. **Real-world Scenario**: Optimize a dashboard query that aggregates large amounts of data

Consider:

- Query patterns and frequency
- Index maintenance overhead
- Storage space vs. performance trade-offs
- Different optimization strategies for OLTP vs. OLAP workloads

Chapter 12: Stored Procedures and Functions

What You'll Learn

Stored procedures and functions are reusable code blocks stored in the database that can encapsulate business logic, improve performance, and enhance security. This chapter covers how to create, manage, and optimize stored procedures and functions in both MySQL and PostgreSQL.

Learning Objectives

By the end of this chapter, you will:

- Understand the difference between stored procedures and functions
- Create and manage stored procedures and functions
- Use parameters, variables, and control structures
- Handle exceptions and errors
- Implement business logic in the database layer
- Understand security and performance implications

Concept Explanation

Stored Procedures vs Functions

Feature	Stored Procedure	Function
Return Value	Optional (via OUT parameters)	Must return a value
Usage	Called with CALL statement	Used in SELECT, WHERE, etc.
Side Effects	Can modify data	Should be read-only (best practice)

Feature	Stored Procedure	Function
Transaction Control	Can use COMMIT/ROLLBACK	Limited transaction control
Parameters	IN, OUT, INOUT	IN parameters only (usually)

Benefits:

- **Performance**: Compiled once, executed many times
- **Security**: Controlled data access, SQL injection prevention
- **Maintainability**: Centralized business logic
- **Network Traffic**: Reduced client-server communication
- **Consistency**: Standardized operations across applications

---

🛠️ Syntax Comparison

MySQL vs PostgreSQL Differences

Feature	MySQL	PostgreSQL
Language	SQL, limited procedural	SQL, PL/pgSQL, Python, etc.
Function Types	Limited	Scalar, Table, Aggregate
Exception Handling	DECLARE HANDLER	BEGIN/EXCEPTION blocks
Cursors	✅	✅
Triggers	✅	✅ (more advanced)
Packages	❌	✅ (Schemas)
Overloading	❌	✅

📝 Basic Stored Procedures

MySQL Stored Procedures

```
-- Simple procedure without parameters
DELIMITER //
CREATE PROCEDURE GetCustomerCount()
BEGIN
    SELECT COUNT(*) as total_customers FROM customers WHERE status = 'active';
END //
DELIMITER ;

-- Call the procedure
CALL GetCustomerCount();

-- Procedure with IN parameters
DELIMITER //
```

```sql
CREATE PROCEDURE GetCustomerOrders(
    IN customer_id_param INT,
    IN start_date DATE,
    IN end_date DATE
)
BEGIN
    SELECT
        o.order_id,
        o.order_date,
        o.total_amount,
        o.status
    FROM orders o
    WHERE o.customer_id = customer_id_param
      AND o.order_date BETWEEN start_date AND end_date
    ORDER BY o.order_date DESC;
END //
DELIMITER ;

-- Call with parameters
CALL GetCustomerOrders(123, '2024-01-01', '2024-12-31');

-- Procedure with OUT parameters
DELIMITER //
CREATE PROCEDURE GetCustomerStats(
    IN customer_id_param INT,
    OUT total_orders INT,
    OUT total_spent DECIMAL(10,2),
    OUT avg_order_value DECIMAL(10,2)
)
BEGIN
    SELECT
        COUNT(order_id),
        COALESCE(SUM(total_amount), 0),
        COALESCE(AVG(total_amount), 0)
    INTO total_orders, total_spent, avg_order_value
    FROM orders
    WHERE customer_id = customer_id_param
      AND status = 'completed';
END //
DELIMITER ;

-- Call and get output
CALL GetCustomerStats(123, @orders, @spent, @avg);
SELECT @orders as total_orders, @spent as total_spent, @avg as avg_order_value;
```

PostgreSQL Stored Procedures

```sql
-- Simple procedure (PostgreSQL 11+)
CREATE OR REPLACE PROCEDURE get_customer_count()
LANGUAGE plpgsql
AS $$
```

```
BEGIN
    RAISE NOTICE 'Total active customers: %', (SELECT COUNT(*) FROM customers
WHERE status = 'active');
END;
$$;

-- Call the procedure
CALL get_customer_count();

-- Procedure with parameters
CREATE OR REPLACE PROCEDURE get_customer_orders(
    customer_id_param INT,
    start_date DATE,
    end_date DATE
)
LANGUAGE plpgsql
AS $$
DECLARE
    order_record RECORD;
BEGIN
    FOR order_record IN
        SELECT
            o.order_id,
            o.order_date,
            o.total_amount,
            o.status
        FROM orders o
        WHERE o.customer_id = customer_id_param
          AND o.order_date BETWEEN start_date AND end_date
        ORDER BY o.order_date DESC
    LOOP
        RAISE NOTICE 'Order ID: %, Date: %, Amount: %',
            order_record.order_id,
            order_record.order_date,
            order_record.total_amount;
    END LOOP;
END;
$$;

-- Call with parameters
CALL get_customer_orders(123, '2024-01-01', '2024-12-31');

-- Procedure with INOUT parameters
CREATE OR REPLACE PROCEDURE get_customer_stats(
    INOUT customer_id_param INT,
    OUT total_orders INT,
    OUT total_spent DECIMAL(10,2),
    OUT avg_order_value DECIMAL(10,2)
)
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT
        COUNT(order_id),
```

```
        COALESCE(SUM(total_amount), 0),
        COALESCE(AVG(total_amount), 0)
    INTO total_orders, total_spent, avg_order_value
    FROM orders
    WHERE customer_id = customer_id_param
      AND status = 'completed';
END;
$$;

-- Call and get output
CALL get_customer_stats(123, NULL, NULL, NULL);
```

🔧 Functions

MySQL Functions

```
-- Scalar function
DELIMITER //
CREATE FUNCTION CalculateDiscount(
    order_amount DECIMAL(10,2),
    customer_tier VARCHAR(20)
) RETURNS DECIMAL(10,2)
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE discount_rate DECIMAL(5,4) DEFAULT 0;

    CASE customer_tier
        WHEN 'VIP' THEN SET discount_rate = 0.15;
        WHEN 'Premium' THEN SET discount_rate = 0.10;
        WHEN 'Regular' THEN SET discount_rate = 0.05;
        ELSE SET discount_rate = 0;
    END CASE;

    IF order_amount >= 1000 THEN
        SET discount_rate = discount_rate + 0.05;
    END IF;

    RETURN order_amount * discount_rate;
END //
DELIMITER ;

-- Use the function
SELECT
    order_id,
    total_amount,
    CalculateDiscount(total_amount, 'VIP') as discount_amount,
    total_amount - CalculateDiscount(total_amount, 'VIP') as final_amount
FROM orders
WHERE customer_id = 123;
```

```
-- Function with complex logic
DELIMITER //
CREATE FUNCTION GetCustomerTier(
    customer_id_param INT
) RETURNS VARCHAR(20)
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE total_spent DECIMAL(10,2) DEFAULT 0;
    DECLARE order_count INT DEFAULT 0;
    DECLARE tier VARCHAR(20) DEFAULT 'Basic';

    SELECT
        COALESCE(SUM(total_amount), 0),
        COUNT(order_id)
    INTO total_spent, order_count
    FROM orders
    WHERE customer_id = customer_id_param
      AND status = 'completed'
      AND order_date >= DATE_SUB(CURDATE(), INTERVAL 12 MONTH);

    IF total_spent >= 5000 OR order_count >= 20 THEN
        SET tier = 'VIP';
    ELSEIF total_spent >= 2000 OR order_count >= 10 THEN
        SET tier = 'Premium';
    ELSEIF total_spent >= 500 OR order_count >= 3 THEN
        SET tier = 'Regular';
    END IF;

    RETURN tier;
END //
DELIMITER ;

-- Use in queries
SELECT
    customer_id,
    first_name,
    last_name,
    GetCustomerTier(customer_id) as tier
FROM customers
WHERE status = 'active'
LIMIT 10;
```

PostgreSQL Functions

```
-- Scalar function
CREATE OR REPLACE FUNCTION calculate_discount(
    order_amount DECIMAL(10,2),
    customer_tier VARCHAR(20)
) RETURNS DECIMAL(10,2)
```

```sql
LANGUAGE plpgsql
IMMUTABLE
AS $$
DECLARE
    discount_rate DECIMAL(5,4) := 0;
BEGIN
    CASE customer_tier
        WHEN 'VIP' THEN discount_rate := 0.15;
        WHEN 'Premium' THEN discount_rate := 0.10;
        WHEN 'Regular' THEN discount_rate := 0.05;
        ELSE discount_rate := 0;
    END CASE;

    IF order_amount >= 1000 THEN
        discount_rate := discount_rate + 0.05;
    END IF;

    RETURN order_amount * discount_rate;
END;
$$;

-- Use the function
SELECT
    order_id,
    total_amount,
    calculate_discount(total_amount, 'VIP') as discount_amount,
    total_amount - calculate_discount(total_amount, 'VIP') as final_amount
FROM orders
WHERE customer_id = 123;

-- Table-valued function
CREATE OR REPLACE FUNCTION get_top_customers(
    limit_count INT DEFAULT 10
) RETURNS TABLE(
    customer_id INT,
    full_name TEXT,
    total_orders BIGINT,
    total_spent DECIMAL(10,2),
    avg_order_value DECIMAL(10,2)
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        c.customer_id,
        c.first_name || ' ' || c.last_name as full_name,
        COUNT(o.order_id) as total_orders,
        COALESCE(SUM(o.total_amount), 0) as total_spent,
        COALESCE(AVG(o.total_amount), 0) as avg_order_value
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
        AND o.status = 'completed'
        AND o.order_date >= CURRENT_DATE - INTERVAL '12 months'
```

```
    WHERE c.status = 'active'
    GROUP BY c.customer_id, c.first_name, c.last_name
    ORDER BY total_spent DESC
    LIMIT limit_count;
END;
$$;

-- Use table-valued function
SELECT * FROM get_top_customers(5);

-- Function with complex return type
CREATE TYPE customer_summary AS (
    customer_id INT,
    tier VARCHAR(20),
    total_spent DECIMAL(10,2),
    order_count INT,
    last_order_date DATE
);

CREATE OR REPLACE FUNCTION get_customer_summary(
    customer_id_param INT
) RETURNS customer_summary
LANGUAGE plpgsql
AS $$
DECLARE
    result customer_summary;
    total_spent DECIMAL(10,2);
    order_count INT;
BEGIN
    SELECT
        COALESCE(SUM(total_amount), 0),
        COUNT(order_id),
        MAX(order_date)
    INTO total_spent, order_count, result.last_order_date
    FROM orders
    WHERE customer_id = customer_id_param
      AND status = 'completed'
      AND order_date >= CURRENT_DATE - INTERVAL '12 months';

    result.customer_id := customer_id_param;
    result.total_spent := total_spent;
    result.order_count := order_count;

    -- Determine tier
    IF total_spent >= 5000 OR order_count >= 20 THEN
        result.tier := 'VIP';
    ELSIF total_spent >= 2000 OR order_count >= 10 THEN
        result.tier := 'Premium';
    ELSIF total_spent >= 500 OR order_count >= 3 THEN
        result.tier := 'Regular';
    ELSE
        result.tier := 'Basic';
    END IF;
```

```
    RETURN result;
END;
$$;

-- Use complex function
SELECT (get_customer_summary(123)).*;
```

Control Structures

Conditional Logic

MySQL:

```
DELIMITER //
CREATE PROCEDURE ProcessOrder(
    IN order_id_param INT,
    OUT result_message VARCHAR(255)
)
BEGIN
    DECLARE order_status VARCHAR(20);
    DECLARE order_amount DECIMAL(10,2);
    DECLARE customer_credit DECIMAL(10,2);

    -- Get order details
    SELECT status, total_amount INTO order_status, order_amount
    FROM orders
    WHERE order_id = order_id_param;

    -- Check if order exists
    IF order_status IS NULL THEN
        SET result_message = 'Order not found';
    ELSEIF order_status != 'pending' THEN
        SET result_message = 'Order already processed';
    ELSE
        -- Get customer credit
        SELECT credit_limit INTO customer_credit
        FROM customers c
        JOIN orders o ON c.customer_id = o.customer_id
        WHERE o.order_id = order_id_param;

        IF order_amount <= customer_credit THEN
            UPDATE orders SET status = 'approved' WHERE order_id = order_id_param;
            SET result_message = 'Order approved';
        ELSE
            UPDATE orders SET status = 'rejected' WHERE order_id = order_id_param;
            SET result_message = 'Order rejected - insufficient credit';
        END IF;
    END IF;
END //
DELIMITER ;
```

PostgreSQL:

```
CREATE OR REPLACE FUNCTION process_order(
    order_id_param INT
) RETURNS TEXT
LANGUAGE plpgsql
AS $$
DECLARE
    order_status VARCHAR(20);
    order_amount DECIMAL(10,2);
    customer_credit DECIMAL(10,2);
    result_message TEXT;
BEGIN
    -- Get order details
    SELECT status, total_amount INTO order_status, order_amount
    FROM orders
    WHERE order_id = order_id_param;

    -- Check if order exists
    IF NOT FOUND THEN
        RETURN 'Order not found';
    ELSIF order_status != 'pending' THEN
        RETURN 'Order already processed';
    ELSE
        -- Get customer credit
        SELECT c.credit_limit INTO customer_credit
        FROM customers c
        JOIN orders o ON c.customer_id = o.customer_id
        WHERE o.order_id = order_id_param;

        IF order_amount <= customer_credit THEN
            UPDATE orders SET status = 'approved' WHERE order_id = order_id_param;
            result_message := 'Order approved';
        ELSE
            UPDATE orders SET status = 'rejected' WHERE order_id = order_id_param;
            result_message := 'Order rejected - insufficient credit';
        END IF;
    END IF;

    RETURN result_message;
END;
$$;
```

Loops

MySQL:

```
DELIMITER //
CREATE PROCEDURE GenerateMonthlyReport(
```

```
    IN year_param INT
)
BEGIN
    DECLARE month_counter INT DEFAULT 1;
    DECLARE monthly_revenue DECIMAL(10,2);

    -- Create temporary table for results
    CREATE TEMPORARY TABLE IF NOT EXISTS monthly_report (
        month_num INT,
        month_name VARCHAR(20),
        revenue DECIMAL(10,2)
    );

    -- Clear previous data
    DELETE FROM monthly_report;

    WHILE month_counter <= 12 DO
        SELECT COALESCE(SUM(total_amount), 0) INTO monthly_revenue
        FROM orders
        WHERE YEAR(order_date) = year_param
          AND MONTH(order_date) = month_counter
          AND status = 'completed';

        INSERT INTO monthly_report VALUES (
            month_counter,
            MONTHNAME(STR_TO_DATE(month_counter, '%m')),
            monthly_revenue
        );

        SET month_counter = month_counter + 1;
    END WHILE;

    SELECT * FROM monthly_report ORDER BY month_num;
END //
DELIMITER ;
```

PostgreSQL:

```
CREATE OR REPLACE FUNCTION generate_monthly_report(
    year_param INT
) RETURNS TABLE(
    month_num INT,
    month_name TEXT,
    revenue DECIMAL(10,2)
)
LANGUAGE plpgsql
AS $$
DECLARE
    month_counter INT;
    monthly_revenue DECIMAL(10,2);
BEGIN
    FOR month_counter IN 1..12 LOOP
```

```
        SELECT COALESCE(SUM(total_amount), 0) INTO monthly_revenue
        FROM orders
        WHERE EXTRACT(YEAR FROM order_date) = year_param
          AND EXTRACT(MONTH FROM order_date) = month_counter
          AND status = 'completed';

        month_num := month_counter;
        month_name := TO_CHAR(TO_DATE(month_counter::TEXT, 'MM'), 'Month');
        revenue := monthly_revenue;

        RETURN NEXT;
    END LOOP;
END;
$$;

-- Use the function
SELECT * FROM generate_monthly_report(2024);
```

⚠ Error Handling

MySQL Error Handling

```
DELIMITER //
CREATE PROCEDURE SafeTransferFunds(
    IN from_account INT,
    IN to_account INT,
    IN amount DECIMAL(10,2),
    OUT result_message VARCHAR(255)
)
BEGIN
    DECLARE account_balance DECIMAL(10,2);
    DECLARE exit_handler_called BOOLEAN DEFAULT FALSE;

    -- Error handlers
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
        SET exit_handler_called = TRUE;
        ROLLBACK;
        SET result_message = 'Transaction failed due to error';
    END;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
    BEGIN
        SET exit_handler_called = TRUE;
        ROLLBACK;
        SET result_message = 'Account not found';
    END;

    START TRANSACTION;
```

```
    -- Check source account balance
    SELECT balance INTO account_balance
    FROM accounts
    WHERE account_id = from_account
    FOR UPDATE;

    IF exit_handler_called THEN
        LEAVE;
    END IF;

    IF account_balance < amount THEN
        ROLLBACK;
        SET result_message = 'Insufficient funds';
    ELSE
        -- Debit source account
        UPDATE accounts
        SET balance = balance - amount
        WHERE account_id = from_account;

        -- Credit destination account
        UPDATE accounts
        SET balance = balance + amount
        WHERE account_id = to_account;

        IF exit_handler_called THEN
            LEAVE;
        END IF;

        COMMIT;
        SET result_message = 'Transfer completed successfully';
    END IF;
END //
DELIMITER ;
```

PostgreSQL Error Handling

```
CREATE OR REPLACE FUNCTION safe_transfer_funds(
    from_account INT,
    to_account INT,
    amount DECIMAL(10,2)
) RETURNS TEXT
LANGUAGE plpgsql
AS $$
DECLARE
    account_balance DECIMAL(10,2);
BEGIN
    BEGIN
        -- Check source account balance
        SELECT balance INTO STRICT account_balance
        FROM accounts
        WHERE account_id = from_account
```

```
        FOR UPDATE;

        IF account_balance < amount THEN
            RETURN 'Insufficient funds';
        END IF;

        -- Debit source account
        UPDATE accounts
        SET balance = balance - amount
        WHERE account_id = from_account;

        -- Credit destination account
        UPDATE accounts
        SET balance = balance + amount
        WHERE account_id = to_account;

        IF NOT FOUND THEN
            RAISE EXCEPTION 'Destination account not found';
        END IF;

        RETURN 'Transfer completed successfully';

    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN 'Source account not found';
        WHEN TOO_MANY_ROWS THEN
            RETURN 'Multiple accounts found - data integrity issue';
        WHEN OTHERS THEN
            RETURN 'Transaction failed: ' || SQLERRM;
    END;
END;
$$;
```

Real-World Business Examples

Example 1: E-commerce Order Processing System

```
-- PostgreSQL comprehensive order processing
CREATE OR REPLACE FUNCTION process_ecommerce_order(
    customer_id_param INT,
    items JSONB,  -- [{"product_id": 1, "quantity": 2, "unit_price": 29.99}, ...]
    shipping_address JSONB,
    payment_method VARCHAR(50)
) RETURNS JSONB
LANGUAGE plpgsql
AS $$
DECLARE
    new_order_id INT;
    item JSONB;
    product_stock INT;
```

```
    order_total DECIMAL(10,2) := 0;
    tax_amount DECIMAL(10,2);
    shipping_cost DECIMAL(10,2) := 9.99;
    customer_tier VARCHAR(20);
    discount_amount DECIMAL(10,2) := 0;
    result JSONB;
BEGIN
    -- Validate customer
    SELECT get_customer_tier(customer_id_param) INTO customer_tier;

    IF customer_tier IS NULL THEN
        RETURN jsonb_build_object(
            'success', false,
            'error', 'Invalid customer ID'
        );
    END IF;

    BEGIN
        -- Create order
        INSERT INTO orders (customer_id, order_date, status, shipping_address,
payment_method)
        VALUES (customer_id_param, CURRENT_TIMESTAMP, 'pending', shipping_address,
payment_method)
        RETURNING order_id INTO new_order_id;

        -- Process each item
        FOR item IN SELECT * FROM jsonb_array_elements(items)
        LOOP
            -- Check stock
            SELECT stock_quantity INTO product_stock
            FROM products
            WHERE product_id = (item->>'product_id')::INT
              AND status = 'active'
            FOR UPDATE;

            IF NOT FOUND THEN
                RAISE EXCEPTION 'Product % not found or inactive', item-
>>'product_id';
            END IF;

            IF product_stock < (item->>'quantity')::INT THEN
                RAISE EXCEPTION 'Insufficient stock for product %. Available: %,
Requested: %',
                    item->>'product_id', product_stock, item->>'quantity';
            END IF;

            -- Add order item
            INSERT INTO order_items (order_id, product_id, quantity, unit_price)
            VALUES (
                new_order_id,
                (item->>'product_id')::INT,
                (item->>'quantity')::INT,
                (item->>'unit_price')::DECIMAL(10,2)
            );
```

```
            -- Update stock
            UPDATE products
            SET stock_quantity = stock_quantity - (item->>'quantity')::INT
            WHERE product_id = (item->>'product_id')::INT;

            -- Add to order total
            order_total := order_total + ((item->>'quantity')::INT * (item-
>>'unit_price')::DECIMAL(10,2));
        END LOOP;

        -- Calculate discount
        discount_amount := calculate_discount(order_total, customer_tier);

        -- Calculate tax (8.5%)
        tax_amount := (order_total - discount_amount) * 0.085;

        -- Free shipping for orders over $100 or VIP customers
        IF order_total >= 100 OR customer_tier = 'VIP' THEN
            shipping_cost := 0;
        END IF;

        -- Update order totals
        UPDATE orders
        SET
            subtotal = order_total,
            discount_amount = discount_amount,
            tax_amount = tax_amount,
            shipping_cost = shipping_cost,
            total_amount = order_total - discount_amount + tax_amount +
shipping_cost,
            status = 'confirmed'
        WHERE order_id = new_order_id;

        -- Return success result
        result := jsonb_build_object(
            'success', true,
            'order_id', new_order_id,
            'subtotal', order_total,
            'discount', discount_amount,
            'tax', tax_amount,
            'shipping', shipping_cost,
            'total', order_total - discount_amount + tax_amount + shipping_cost,
            'customer_tier', customer_tier
        );

        RETURN result;

    EXCEPTION
        WHEN OTHERS THEN
            -- Rollback is automatic in PostgreSQL functions
            RETURN jsonb_build_object(
                'success', false,
                'error', SQLERRM
```

```
            );
    END;
END;
$$;

-- Usage example
SELECT process_ecommerce_order(
    123,  -- customer_id
    '[{"product_id": 1, "quantity": 2, "unit_price": 29.99},
      {"product_id": 5, "quantity": 1, "unit_price": 149.99}]'::jsonb,
    '{"street": "123 Main St", "city": "Anytown", "state": "CA", "zip":
"12345"}'::jsonb,
    'credit_card'
);
```

Example 2: Customer Analytics and Segmentation

```
-- MySQL customer analytics procedure
DELIMITER //
CREATE PROCEDURE AnalyzeCustomerSegments(
    IN analysis_date DATE
)
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE customer_id_var INT;
    DECLARE total_spent DECIMAL(10,2);
    DECLARE order_count INT;
    DECLARE avg_order_value DECIMAL(10,2);
    DECLARE last_order_date DATE;
    DECLARE days_since_last_order INT;
    DECLARE customer_tier VARCHAR(20);
    DECLARE lifecycle_stage VARCHAR(20);
    DECLARE churn_risk VARCHAR(20);

    -- Cursor for active customers
    DECLARE customer_cursor CURSOR FOR
        SELECT customer_id FROM customers WHERE status = 'active';

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Create or clear analytics table
    CREATE TABLE IF NOT EXISTS customer_analytics (
        customer_id INT PRIMARY KEY,
        analysis_date DATE,
        total_spent DECIMAL(10,2),
        order_count INT,
        avg_order_value DECIMAL(10,2),
        last_order_date DATE,
        days_since_last_order INT,
        customer_tier VARCHAR(20),
        lifecycle_stage VARCHAR(20),
```

```
        churn_risk VARCHAR(20),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );

    DELETE FROM customer_analytics WHERE analysis_date = analysis_date;

    OPEN customer_cursor;

    customer_loop: LOOP
        FETCH customer_cursor INTO customer_id_var;

        IF done THEN
            LEAVE customer_loop;
        END IF;

        -- Calculate customer metrics
        SELECT
            COALESCE(SUM(total_amount), 0),
            COUNT(order_id),
            COALESCE(AVG(total_amount), 0),
            MAX(order_date)
        INTO total_spent, order_count, avg_order_value, last_order_date
        FROM orders
        WHERE customer_id = customer_id_var
          AND status = 'completed'
          AND order_date >= DATE_SUB(analysis_date, INTERVAL 12 MONTH);

        -- Calculate days since last order
        IF last_order_date IS NOT NULL THEN
            SET days_since_last_order = DATEDIFF(analysis_date, last_order_date);
        ELSE
            SET days_since_last_order = NULL;
        END IF;

        -- Determine customer tier
        IF total_spent >= 5000 OR order_count >= 20 THEN
            SET customer_tier = 'VIP';
        ELSEIF total_spent >= 2000 OR order_count >= 10 THEN
            SET customer_tier = 'Premium';
        ELSEIF total_spent >= 500 OR order_count >= 3 THEN
            SET customer_tier = 'Regular';
        ELSE
            SET customer_tier = 'Basic';
        END IF;

        -- Determine lifecycle stage
        IF order_count = 0 THEN
            SET lifecycle_stage = 'Prospect';
        ELSEIF order_count = 1 THEN
            SET lifecycle_stage = 'New Customer';
        ELSEIF order_count <= 5 THEN
            SET lifecycle_stage = 'Developing';
        ELSE
            SET lifecycle_stage = 'Established';
```

```
        END IF;

        -- Determine churn risk
        IF days_since_last_order IS NULL THEN
            SET churn_risk = 'Never Purchased';
        ELSEIF days_since_last_order <= 30 THEN
            SET churn_risk = 'Low';
        ELSEIF days_since_last_order <= 90 THEN
            SET churn_risk = 'Medium';
        ELSEIF days_since_last_order <= 180 THEN
            SET churn_risk = 'High';
        ELSE
            SET churn_risk = 'Very High';
        END IF;

        -- Insert analytics record
        INSERT INTO customer_analytics (
            customer_id, analysis_date, total_spent, order_count, avg_order_value,
            last_order_date, days_since_last_order, customer_tier,
lifecycle_stage, churn_risk
        ) VALUES (
            customer_id_var, analysis_date, total_spent, order_count,
avg_order_value,
            last_order_date, days_since_last_order, customer_tier,
lifecycle_stage, churn_risk
        );

    END LOOP;

    CLOSE customer_cursor;

    -- Return summary
    SELECT
        customer_tier,
        lifecycle_stage,
        churn_risk,
        COUNT(*) as customer_count,
        AVG(total_spent) as avg_spent,
        AVG(order_count) as avg_orders
    FROM customer_analytics
    WHERE analysis_date = analysis_date
    GROUP BY customer_tier, lifecycle_stage, churn_risk
    ORDER BY customer_tier, lifecycle_stage, churn_risk;
END //
DELIMITER ;

-- Run the analysis
CALL AnalyzeCustomerSegments(CURDATE());
```

Use Cases & Interview Tips

Common Interview Questions:

1. **"When would you use a stored procedure vs a function?"**
    - Procedures: Complex business logic, data modifications, transaction control
    - Functions: Calculations, data transformations, reusable logic in queries
    - Consider maintainability and where business logic should reside
2. **"How do you handle errors in stored procedures?"**
    - MySQL: DECLARE HANDLER statements
    - PostgreSQL: BEGIN/EXCEPTION blocks
    - Always consider transaction rollback and cleanup
3. **"What are the performance implications of stored procedures?"**
    - Pros: Compiled once, reduced network traffic, optimized execution plans
    - Cons: Database server load, harder to scale horizontally, debugging complexity
4. **"How do you secure stored procedures?"**
    - Grant specific EXECUTE permissions
    - Use DEFINER vs INVOKER rights
    - Validate all input parameters
    - Avoid dynamic SQL construction

Best Practices:

1. **Keep procedures focused and single-purpose**
2. **Use proper error handling and logging**
3. **Validate all input parameters**
4. **Use transactions appropriately**
5. **Document complex business logic**
6. **Consider version control for database objects**

⚠ Things to Watch Out For

1. **SQL Injection in Dynamic SQL**

```
-- Bad: Vulnerable to SQL injection
DELIMITER //
CREATE PROCEDURE BadSearch(IN search_term VARCHAR(255))
BEGIN
    SET @sql = CONCAT('SELECT * FROM products WHERE product_name LIKE "%',
search_term, '%"');
    PREPARE stmt FROM @sql;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
END //
DELIMITER ;
```

```
-- Good: Use parameters
DELIMITER //
CREATE PROCEDURE GoodSearch(IN search_term VARCHAR(255))
BEGIN
    SELECT * FROM products
    WHERE product_name LIKE CONCAT('%', search_term, '%');
END //
DELIMITER ;
```

2. **Transaction Management**

```
-- Be careful with transaction boundaries
CREATE OR REPLACE FUNCTION risky_function()
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    -- This function should not control transactions
    -- if it's called within another transaction
    INSERT INTO audit_log VALUES (...);

    -- Don't do this in a function:
    -- COMMIT;  -- This would commit the entire transaction
END;
$$;
```

3. **Performance Issues**

```
-- Avoid cursor loops when set-based operations work
-- Bad: Cursor-based processing
DELIMITER //
CREATE PROCEDURE SlowUpdate()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE order_id_var INT;
    DECLARE order_cursor CURSOR FOR SELECT order_id FROM orders WHERE status =
'pending';
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN order_cursor;
    order_loop: LOOP
        FETCH order_cursor INTO order_id_var;
        IF done THEN LEAVE order_loop; END IF;

        UPDATE orders SET processed_date = NOW() WHERE order_id = order_id_var;
    END LOOP;
    CLOSE order_cursor;
END //
```

```
DELIMITER ;

-- Good: Set-based operation
DELIMITER //
CREATE PROCEDURE FastUpdate()
BEGIN
    UPDATE orders SET processed_date = NOW() WHERE status = 'pending';
END //
DELIMITER ;
```

🚀 Next Steps

In the next chapter, we'll explore **Triggers and Events** - special stored procedures that automatically execute in response to database events. You'll learn how to create audit trails, enforce business rules, and automate database maintenance tasks.

📝 Quick Practice

Try these stored procedure exercises:

1. **Basic Procedure**: Create a procedure to calculate monthly sales totals
2. **Function**: Write a function to determine shipping cost based on weight and distance
3. **Error Handling**: Implement a safe money transfer procedure with proper error handling
4. **Business Logic**: Create a customer loyalty points calculation system
5. **Complex Processing**: Build an order fulfillment procedure that handles inventory, pricing, and notifications

Consider:

- When to use procedures vs functions
- Proper error handling strategies
- Transaction management
- Security implications
- Performance optimization techniques

Chapter 13: Triggers and Events

🗄 What You'll Learn

Triggers are special stored procedures that automatically execute ("fire") in response to specific database events. Events are scheduled tasks that run at predetermined times. This chapter covers how to create, manage, and optimize triggers and events for automation, auditing, and business rule enforcement.

🎯 Learning Objectives

By the end of this chapter, you will:

- Understand different types of triggers and their use cases
- Create BEFORE, AFTER, and INSTEAD OF triggers
- Implement audit trails and data validation
- Use events for scheduled database maintenance
- Handle trigger performance and avoid common pitfalls
- Implement complex business rules with triggers

🔍 Concept Explanation

What are Triggers?

Triggers are special stored procedures that automatically execute in response to specific database events such as INSERT, UPDATE, or DELETE operations. They cannot be called directly and run within the same transaction as the triggering statement.

Types of Triggers:

- **BEFORE**: Execute before the triggering event
- **AFTER**: Execute after the triggering event
- **INSTEAD OF**: Replace the triggering event (views only)

Common Use Cases:

- Audit trails and logging
- Data validation and business rules
- Automatic calculations and updates
- Maintaining derived data
- Security and access control

What are Events?

Events are scheduled tasks that run automatically at specified times or intervals. They're useful for database maintenance, reporting, and batch processing.

🛠️ Syntax Comparison

MySQL vs PostgreSQL Triggers

Feature	MySQL	PostgreSQL
Trigger Types	BEFORE, AFTER	BEFORE, AFTER, INSTEAD OF
Row vs Statement	Row-level only	Both row and statement level
Multiple Triggers	✅ (5.7+)	✅
Trigger Functions	Inline SQL	Separate functions
OLD/NEW References	OLD, NEW	OLD, NEW

Feature	MySQL	PostgreSQL
Conditional Logic	WHEN clause (5.7+)	WHEN clause
Events/Scheduling	Event Scheduler	pg_cron extension

Basic Triggers

MySQL Triggers

```
-- Simple audit trigger
DELIMITER //
CREATE TRIGGER audit_customer_changes
    AFTER UPDATE ON customers
    FOR EACH ROW
BEGIN
    INSERT INTO customer_audit (
        customer_id,
        action_type,
        old_email,
        new_email,
        old_status,
        new_status,
        changed_by,
        changed_at
    ) VALUES (
        NEW.customer_id,
        'UPDATE',
        OLD.email,
        NEW.email,
        OLD.status,
        NEW.status,
        USER(),
        NOW()
    );
END //
DELIMITER ;

-- BEFORE trigger for data validation
DELIMITER //
CREATE TRIGGER validate_order_before_insert
    BEFORE INSERT ON orders
    FOR EACH ROW
BEGIN
    -- Validate customer exists and is active
    IF NOT EXISTS (SELECT 1 FROM customers WHERE customer_id = NEW.customer_id AND
status = 'active') THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Customer does not exist or is
inactive';
    END IF;

    -- Set default values
```

```
    IF NEW.order_date IS NULL THEN
        SET NEW.order_date = NOW();
    END IF;

    IF NEW.status IS NULL THEN
        SET NEW.status = 'pending';
    END IF;

    -- Validate order amount
    IF NEW.total_amount < 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Order amount cannot be
negative';
    END IF;
END //
DELIMITER ;

-- AFTER trigger for automatic updates
DELIMITER //
CREATE TRIGGER update_customer_stats_after_order
    AFTER INSERT ON orders
    FOR EACH ROW
BEGIN
    -- Update customer statistics
    UPDATE customer_stats
    SET
        total_orders = total_orders + 1,
        total_spent = total_spent + NEW.total_amount,
        last_order_date = NEW.order_date
    WHERE customer_id = NEW.customer_id;

    -- Insert if customer stats don't exist
    IF ROW_COUNT() = 0 THEN
        INSERT INTO customer_stats (customer_id, total_orders, total_spent,
last_order_date)
        VALUES (NEW.customer_id, 1, NEW.total_amount, NEW.order_date);
    END IF;
END //
DELIMITER ;
```

PostgreSQL Triggers

```
-- Create trigger function first
CREATE OR REPLACE FUNCTION audit_customer_changes()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF TG_OP = 'UPDATE' THEN
        INSERT INTO customer_audit (
            customer_id,
            action_type,
```

```
            old_email,
            new_email,
            old_status,
            new_status,
            changed_by,
            changed_at
        ) VALUES (
            NEW.customer_id,
            'UPDATE',
            OLD.email,
            NEW.email,
            OLD.status,
            NEW.status,
            current_user,
            current_timestamp
        );
        RETURN NEW;
    ELSIF TG_OP = 'INSERT' THEN
        INSERT INTO customer_audit (
            customer_id,
            action_type,
            new_email,
            new_status,
            changed_by,
            changed_at
        ) VALUES (
            NEW.customer_id,
            'INSERT',
            NEW.email,
            NEW.status,
            current_user,
            current_timestamp
        );
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO customer_audit (
            customer_id,
            action_type,
            old_email,
            old_status,
            changed_by,
            changed_at
        ) VALUES (
            OLD.customer_id,
            'DELETE',
            OLD.email,
            OLD.status,
            current_user,
            current_timestamp
        );
        RETURN OLD;
    END IF;

    RETURN NULL;
```

```
END;
$$;

-- Create the trigger
CREATE TRIGGER audit_customer_changes
    AFTER INSERT OR UPDATE OR DELETE ON customers
    FOR EACH ROW
    EXECUTE FUNCTION audit_customer_changes();

-- BEFORE trigger for validation
CREATE OR REPLACE FUNCTION validate_order_before_insert()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    -- Validate customer exists and is active
    IF NOT EXISTS (SELECT 1 FROM customers WHERE customer_id = NEW.customer_id AND
status = 'active') THEN
        RAISE EXCEPTION 'Customer does not exist or is inactive';
    END IF;

    -- Set default values
    IF NEW.order_date IS NULL THEN
        NEW.order_date := current_timestamp;
    END IF;

    IF NEW.status IS NULL THEN
        NEW.status := 'pending';
    END IF;

    -- Validate order amount
    IF NEW.total_amount < 0 THEN
        RAISE EXCEPTION 'Order amount cannot be negative';
    END IF;

    RETURN NEW;
END;
$$;

CREATE TRIGGER validate_order_before_insert
    BEFORE INSERT ON orders
    FOR EACH ROW
    EXECUTE FUNCTION validate_order_before_insert();

-- Statement-level trigger (PostgreSQL only)
CREATE OR REPLACE FUNCTION log_bulk_operations()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO operation_log (table_name, operation, row_count, timestamp)
    VALUES (TG_TABLE_NAME, TG_OP, TG_LEVEL, current_timestamp);

    RETURN NULL;
```

```
END;
$$;

CREATE TRIGGER log_bulk_operations
    AFTER INSERT OR UPDATE OR DELETE ON orders
    FOR EACH STATEMENT
    EXECUTE FUNCTION log_bulk_operations();
```

📊 Advanced Trigger Examples

Inventory Management System

MySQL:

```
-- Create inventory tracking tables
CREATE TABLE inventory_movements (
    movement_id INT AUTO_INCREMENT PRIMARY KEY,
    product_id INT NOT NULL,
    movement_type ENUM('IN', 'OUT', 'ADJUSTMENT') NOT NULL,
    quantity INT NOT NULL,
    reference_type VARCHAR(50),
    reference_id INT,
    movement_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    notes TEXT
);

-- Trigger to track inventory movements
DELIMITER //
CREATE TRIGGER track_inventory_on_order_item_insert
    AFTER INSERT ON order_items
    FOR EACH ROW
BEGIN
    DECLARE current_stock INT;

    -- Check current stock
    SELECT stock_quantity INTO current_stock
    FROM products
    WHERE product_id = NEW.product_id;

    -- Validate sufficient stock
    IF current_stock < NEW.quantity THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = CONCAT('Insufficient stock. Available: ',
current_stock, ', Requested: ', NEW.quantity);
    END IF;

    -- Update product stock
    UPDATE products
    SET stock_quantity = stock_quantity - NEW.quantity
    WHERE product_id = NEW.product_id;
```

```
    -- Record inventory movement
    INSERT INTO inventory_movements (
        product_id,
        movement_type,
        quantity,
        reference_type,
        reference_id,
        notes
    ) VALUES (
        NEW.product_id,
        'OUT',
        NEW.quantity,
        'ORDER',
        NEW.order_id,
        CONCAT('Order item for order #', NEW.order_id)
    );
END //
DELIMITER ;

-- Trigger to handle order cancellations
DELIMITER //
CREATE TRIGGER restore_inventory_on_order_cancel
    AFTER UPDATE ON orders
    FOR EACH ROW
BEGIN
    -- Only process if order status changed to cancelled
    IF OLD.status != 'cancelled' AND NEW.status = 'cancelled' THEN
        -- Restore inventory for all order items
        INSERT INTO inventory_movements (product_id, movement_type, quantity,
reference_type, reference_id, notes)
        SELECT
            oi.product_id,
            'IN',
            oi.quantity,
            'ORDER_CANCEL',
            NEW.order_id,
            CONCAT('Inventory restored from cancelled order #', NEW.order_id)
        FROM order_items oi
        WHERE oi.order_id = NEW.order_id;

        -- Update product stock
        UPDATE products p
        JOIN order_items oi ON p.product_id = oi.product_id
        SET p.stock_quantity = p.stock_quantity + oi.quantity
        WHERE oi.order_id = NEW.order_id;
    END IF;
END //
DELIMITER ;
```

PostgreSQL:

```
-- Comprehensive inventory management trigger
CREATE OR REPLACE FUNCTION manage_inventory()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    current_stock INT;
    item_record RECORD;
BEGIN
    IF TG_TABLE_NAME = 'order_items' THEN
        IF TG_OP = 'INSERT' THEN
            -- Check and update stock for new order item
            SELECT stock_quantity INTO current_stock
            FROM products
            WHERE product_id = NEW.product_id
            FOR UPDATE;

            IF NOT FOUND THEN
                RAISE EXCEPTION 'Product % not found', NEW.product_id;
            END IF;

            IF current_stock < NEW.quantity THEN
                RAISE EXCEPTION 'Insufficient stock. Available: %, Requested: %',
current_stock, NEW.quantity;
            END IF;

            -- Update stock
            UPDATE products
            SET stock_quantity = stock_quantity - NEW.quantity
            WHERE product_id = NEW.product_id;

            -- Record movement
            INSERT INTO inventory_movements (
                product_id, movement_type, quantity, reference_type, reference_id,
notes
            ) VALUES (
                NEW.product_id, 'OUT', NEW.quantity, 'ORDER', NEW.order_id,
                'Order item for order #' || NEW.order_id
            );

            RETURN NEW;

        ELSIF TG_OP = 'DELETE' THEN
            -- Restore stock when order item is deleted
            UPDATE products
            SET stock_quantity = stock_quantity + OLD.quantity
            WHERE product_id = OLD.product_id;

            INSERT INTO inventory_movements (
                product_id, movement_type, quantity, reference_type, reference_id,
notes
            ) VALUES (
                OLD.product_id, 'IN', OLD.quantity, 'ORDER_ITEM_DELETE',
```

```
    OLD.order_id,
                    'Stock restored from deleted order item'
                );

                RETURN OLD;
            END IF;

        ELSIF TG_TABLE_NAME = 'orders' THEN
            IF TG_OP = 'UPDATE' AND OLD.status != 'cancelled' AND NEW.status =
'cancelled' THEN
                -- Restore inventory for cancelled orders
                FOR item_record IN
                    SELECT product_id, quantity FROM order_items WHERE order_id =
NEW.order_id
                LOOP
                    UPDATE products
                    SET stock_quantity = stock_quantity + item_record.quantity
                    WHERE product_id = item_record.product_id;

                    INSERT INTO inventory_movements (
                        product_id, movement_type, quantity, reference_type,
reference_id, notes
                    ) VALUES (
                        item_record.product_id, 'IN', item_record.quantity,
'ORDER_CANCEL', NEW.order_id,
                        'Inventory restored from cancelled order #' || NEW.order_id
                    );
                END LOOP;
            END IF;

            RETURN NEW;
        END IF;

        RETURN NULL;
    END;
    $$;

    -- Create triggers
    CREATE TRIGGER manage_inventory_order_items
        AFTER INSERT OR DELETE ON order_items
        FOR EACH ROW
        EXECUTE FUNCTION manage_inventory();

    CREATE TRIGGER manage_inventory_orders
        AFTER UPDATE ON orders
        FOR EACH ROW
        EXECUTE FUNCTION manage_inventory();
```

Comprehensive Audit System

PostgreSQL Advanced Audit:

```
-- Generic audit function
CREATE OR REPLACE FUNCTION audit_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    old_data JSONB;
    new_data JSONB;
    changed_fields JSONB;
BEGIN
    -- Convert row data to JSON
    IF TG_OP = 'DELETE' THEN
        old_data := to_jsonb(OLD);
        new_data := NULL;
    ELSIF TG_OP = 'INSERT' THEN
        old_data := NULL;
        new_data := to_jsonb(NEW);
    ELSIF TG_OP = 'UPDATE' THEN
        old_data := to_jsonb(OLD);
        new_data := to_jsonb(NEW);

        -- Calculate changed fields
        SELECT jsonb_object_agg(key, jsonb_build_object(
            'old', old_data->key,
            'new', new_data->key
        )) INTO changed_fields
        FROM jsonb_each(new_data)
        WHERE new_data->key IS DISTINCT FROM old_data->key;
    END IF;

    -- Insert audit record
    INSERT INTO audit_log (
        table_name,
        operation,
        record_id,
        old_data,
        new_data,
        changed_fields,
        user_name,
        timestamp,
        application_name,
        client_addr
    ) VALUES (
        TG_TABLE_NAME,
        TG_OP,
        COALESCE(NEW.id, OLD.id),  -- Assumes 'id' column exists
        old_data,
        new_data,
        changed_fields,
        current_user,
        current_timestamp,
        current_setting('application_name', true),
        inet_client_addr()
```

```
    );

    IF TG_OP = 'DELETE' THEN
        RETURN OLD;
    ELSE
        RETURN NEW;
    END IF;
END;
$$;

-- Apply audit trigger to multiple tables
CREATE TRIGGER audit_customers
    AFTER INSERT OR UPDATE OR DELETE ON customers
    FOR EACH ROW
    EXECUTE FUNCTION audit_trigger();

CREATE TRIGGER audit_orders
    AFTER INSERT OR UPDATE OR DELETE ON orders
    FOR EACH ROW
    EXECUTE FUNCTION audit_trigger();

CREATE TRIGGER audit_products
    AFTER INSERT OR UPDATE OR DELETE ON products
    FOR EACH ROW
    EXECUTE FUNCTION audit_trigger();
```

Events and Scheduled Tasks

MySQL Event Scheduler

```
-- Enable event scheduler
SET GLOBAL event_scheduler = ON;

-- Daily cleanup event
DELIMITER //
CREATE EVENT daily_cleanup
ON SCHEDULE EVERY 1 DAY
STARTS '2024-01-01 02:00:00'
DO
BEGIN
    -- Clean up old audit records (keep 1 year)
    DELETE FROM audit_log
    WHERE created_at < DATE_SUB(NOW(), INTERVAL 1 YEAR);

    -- Clean up expired sessions
    DELETE FROM user_sessions
    WHERE expires_at < NOW();

    -- Update customer statistics
    CALL UpdateCustomerStatistics();
```

```
    -- Log cleanup completion
    INSERT INTO maintenance_log (task_name, completion_time, records_affected)
    VALUES ('daily_cleanup', NOW(), ROW_COUNT());
END //
DELIMITER ;

-- Weekly report generation
DELIMITER //
CREATE EVENT weekly_sales_report
ON SCHEDULE EVERY 1 WEEK
STARTS '2024-01-01 06:00:00'
DO
BEGIN
    DECLARE report_start_date DATE;
    DECLARE report_end_date DATE;

    SET report_end_date = DATE_SUB(CURDATE(), INTERVAL 1 DAY);
    SET report_start_date = DATE_SUB(report_end_date, INTERVAL 6 DAY);

    -- Generate weekly sales summary
    INSERT INTO weekly_sales_reports (
        report_week,
        start_date,
        end_date,
        total_orders,
        total_revenue,
        avg_order_value,
        new_customers,
        generated_at
    )
    SELECT
        YEARWEEK(report_end_date),
        report_start_date,
        report_end_date,
        COUNT(o.order_id),
        SUM(o.total_amount),
        AVG(o.total_amount),
        COUNT(DISTINCT CASE WHEN c.created_at >= report_start_date THEN
c.customer_id END),
        NOW()
    FROM orders o
    JOIN customers c ON o.customer_id = c.customer_id
    WHERE o.order_date BETWEEN report_start_date AND report_end_date
      AND o.status = 'completed';
END //
DELIMITER ;

-- Monthly maintenance event
DELIMITER //
CREATE EVENT monthly_maintenance
ON SCHEDULE EVERY 1 MONTH
STARTS '2024-01-01 01:00:00'
DO
```

```
BEGIN
    -- Optimize tables
    OPTIMIZE TABLE customers, orders, order_items, products;

    -- Update table statistics
    ANALYZE TABLE customers, orders, order_items, products;

    -- Archive old data
    INSERT INTO orders_archive
    SELECT * FROM orders
    WHERE order_date < DATE_SUB(NOW(), INTERVAL 2 YEAR)
      AND status IN ('completed', 'cancelled');

    DELETE FROM orders
    WHERE order_date < DATE_SUB(NOW(), INTERVAL 2 YEAR)
      AND status IN ('completed', 'cancelled');

    -- Log maintenance completion
    INSERT INTO maintenance_log (task_name, completion_time, records_affected)
    VALUES ('monthly_maintenance', NOW(), ROW_COUNT());
END //
DELIMITER ;

-- View and manage events
SHOW EVENTS;

-- Disable an event
ALTER EVENT daily_cleanup DISABLE;

-- Enable an event
ALTER EVENT daily_cleanup ENABLE;

-- Drop an event
DROP EVENT IF EXISTS old_event;
```

PostgreSQL Scheduled Tasks (pg_cron)

```
-- Install pg_cron extension (requires superuser)
CREATE EXTENSION IF NOT EXISTS pg_cron;

-- Schedule daily cleanup (runs at 2 AM every day)
SELECT cron.schedule('daily-cleanup', '0 2 * * *', $$
    -- Clean up old audit records
    DELETE FROM audit_log WHERE timestamp < NOW() - INTERVAL '1 year';

    -- Clean up expired sessions
    DELETE FROM user_sessions WHERE expires_at < NOW();

    -- Update materialized views
    REFRESH MATERIALIZED VIEW customer_analytics_12m;
```

```
    -- Log completion
    INSERT INTO maintenance_log (task_name, completion_time)
    VALUES ('daily_cleanup', NOW());
$$);

-- Schedule weekly report (runs at 6 AM every Monday)
SELECT cron.schedule('weekly-report', '0 6 * * 1', $$
    INSERT INTO weekly_sales_reports (
        report_week,
        start_date,
        end_date,
        total_orders,
        total_revenue,
        avg_order_value,
        new_customers,
        generated_at
    )
    SELECT
        EXTRACT(WEEK FROM (CURRENT_DATE - INTERVAL '1 week')),
        CURRENT_DATE - INTERVAL '1 week',
        CURRENT_DATE - INTERVAL '1 day',
        COUNT(o.order_id),
        SUM(o.total_amount),
        AVG(o.total_amount),
        COUNT(DISTINCT CASE WHEN c.created_at >= CURRENT_DATE - INTERVAL '1 week'
THEN c.customer_id END),
        NOW()
    FROM orders o
    JOIN customers c ON o.customer_id = c.customer_id
    WHERE o.order_date >= CURRENT_DATE - INTERVAL '1 week'
      AND o.order_date < CURRENT_DATE
      AND o.status = 'completed';
$$);

-- Schedule monthly maintenance (runs at 1 AM on the 1st of each month)
SELECT cron.schedule('monthly-maintenance', '0 1 1 * *', $$
    -- Vacuum and analyze tables
    VACUUM ANALYZE customers;
    VACUUM ANALYZE orders;
    VACUUM ANALYZE order_items;
    VACUUM ANALYZE products;

    -- Reindex if needed
    REINDEX INDEX CONCURRENTLY idx_orders_customer_date;

    -- Archive old data
    WITH archived_orders AS (
        DELETE FROM orders
        WHERE order_date < NOW() - INTERVAL '2 years'
          AND status IN ('completed', 'cancelled')
        RETURNING *
    )
    INSERT INTO orders_archive SELECT * FROM archived_orders;
$$);
```

```
-- View scheduled jobs
SELECT * FROM cron.job;

-- Unschedule a job
SELECT cron.unschedule('daily-cleanup');
```

Real-World Business Examples

Example 1: E-commerce Price Change Tracking

```
-- PostgreSQL: Track price changes and notify stakeholders
CREATE TABLE price_history (
    price_history_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    old_price DECIMAL(10,2),
    new_price DECIMAL(10,2),
    price_change_percent DECIMAL(5,2),
    changed_by VARCHAR(100),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    reason TEXT
);

CREATE TABLE price_alerts (
    alert_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    alert_type VARCHAR(50),
    message TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    processed BOOLEAN DEFAULT FALSE
);

CREATE OR REPLACE FUNCTION track_price_changes()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    price_change_percent DECIMAL(5,2);
    alert_message TEXT;
BEGIN
    -- Only process if price actually changed
    IF OLD.price IS DISTINCT FROM NEW.price THEN
        -- Calculate percentage change
        IF OLD.price > 0 THEN
            price_change_percent := ((NEW.price - OLD.price) / OLD.price) * 100;
        ELSE
            price_change_percent := 100;  -- New product
        END IF;

        -- Record price history
```

```
        INSERT INTO price_history (
            product_id,
            old_price,
            new_price,
            price_change_percent,
            changed_by
        ) VALUES (
            NEW.product_id,
            OLD.price,
            NEW.price,
            price_change_percent,
            current_user
        );

        -- Generate alerts for significant changes
        IF ABS(price_change_percent) >= 10 THEN
            alert_message := format(
                'Significant price change for product %s (%s): %s → %s (%.1f%%
%s)',
                NEW.product_name,
                NEW.sku,
                OLD.price,
                NEW.price,
                ABS(price_change_percent),
                CASE WHEN price_change_percent > 0 THEN 'increase' ELSE 'decrease'
END
            );

            INSERT INTO price_alerts (product_id, alert_type, message)
            VALUES (NEW.product_id, 'SIGNIFICANT_CHANGE', alert_message);
        END IF;

        -- Alert for products going out of stock
        IF OLD.stock_quantity > 0 AND NEW.stock_quantity = 0 THEN
            INSERT INTO price_alerts (product_id, alert_type, message)
            VALUES (NEW.product_id, 'OUT_OF_STOCK',
                   format('Product %s (%s) is now out of stock', NEW.product_name,
NEW.sku));
        END IF;

        -- Alert for products coming back in stock
        IF OLD.stock_quantity = 0 AND NEW.stock_quantity > 0 THEN
            INSERT INTO price_alerts (product_id, alert_type, message)
            VALUES (NEW.product_id, 'BACK_IN_STOCK',
                   format('Product %s (%s) is back in stock (%s units)',
                          NEW.product_name, NEW.sku, NEW.stock_quantity));
        END IF;
    END IF;

    RETURN NEW;
END;
$$;

CREATE TRIGGER track_price_changes
```

```
    AFTER UPDATE ON products
    FOR EACH ROW
    EXECUTE FUNCTION track_price_changes();
```

Example 2: Customer Loyalty Points System

```
-- MySQL: Automatic loyalty points calculation
CREATE TABLE customer_loyalty (
    customer_id INT PRIMARY KEY,
    total_points INT DEFAULT 0,
    tier VARCHAR(20) DEFAULT 'Bronze',
    tier_start_date DATE,
    lifetime_points INT DEFAULT 0,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE points_transactions (
    transaction_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    points_earned INT DEFAULT 0,
    points_redeemed INT DEFAULT 0,
    transaction_type VARCHAR(50),
    reference_id INT,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

DELIMITER //
CREATE TRIGGER calculate_loyalty_points
    AFTER INSERT ON orders
    FOR EACH ROW
BEGIN
    DECLARE points_earned INT DEFAULT 0;
    DECLARE current_tier VARCHAR(20);
    DECLARE new_tier VARCHAR(20);
    DECLARE total_lifetime_points INT;

    -- Only process completed orders
    IF NEW.status = 'completed' THEN
        -- Calculate base points (1 point per dollar)
        SET points_earned = FLOOR(NEW.total_amount);

        -- Get current customer tier
        SELECT tier INTO current_tier
        FROM customer_loyalty
        WHERE customer_id = NEW.customer_id;

        -- Apply tier multipliers
        CASE current_tier
            WHEN 'Gold' THEN SET points_earned = points_earned * 2;
            WHEN 'Silver' THEN SET points_earned = FLOOR(points_earned * 1.5);
```

```
            WHEN 'Bronze' THEN SET points_earned = points_earned * 1;
        END CASE;

        -- Bonus points for large orders
        IF NEW.total_amount >= 500 THEN
            SET points_earned = points_earned + 100;
        ELSEIF NEW.total_amount >= 200 THEN
            SET points_earned = points_earned + 50;
        END IF;

        -- Update customer loyalty record
        INSERT INTO customer_loyalty (customer_id, total_points, lifetime_points,
tier, tier_start_date)
        VALUES (NEW.customer_id, points_earned, points_earned, 'Bronze',
CURDATE())
        ON DUPLICATE KEY UPDATE
            total_points = total_points + points_earned,
            lifetime_points = lifetime_points + points_earned;

        -- Check for tier upgrade
        SELECT lifetime_points INTO total_lifetime_points
        FROM customer_loyalty
        WHERE customer_id = NEW.customer_id;

        IF total_lifetime_points >= 10000 THEN
            SET new_tier = 'Gold';
        ELSEIF total_lifetime_points >= 5000 THEN
            SET new_tier = 'Silver';
        ELSE
            SET new_tier = 'Bronze';
        END IF;

        -- Update tier if changed
        IF new_tier != current_tier THEN
            UPDATE customer_loyalty
            SET tier = new_tier, tier_start_date = CURDATE()
            WHERE customer_id = NEW.customer_id;
        END IF;

        -- Record points transaction
        INSERT INTO points_transactions (
            customer_id,
            points_earned,
            transaction_type,
            reference_id,
            description
        ) VALUES (
            NEW.customer_id,
            points_earned,
            'ORDER_PURCHASE',
            NEW.order_id,
            CONCAT('Points earned from order #', NEW.order_id, ' ($',
NEW.total_amount, ')')
        );
```

```
    END IF;
END //
DELIMITER ;

-- Trigger for points redemption
DELIMITER //
CREATE TRIGGER handle_points_redemption
    AFTER INSERT ON points_redemptions
    FOR EACH ROW
BEGIN
    DECLARE current_points INT;

    -- Check if customer has enough points
    SELECT total_points INTO current_points
    FROM customer_loyalty
    WHERE customer_id = NEW.customer_id;

    IF current_points >= NEW.points_redeemed THEN
        -- Deduct points
        UPDATE customer_loyalty
        SET total_points = total_points - NEW.points_redeemed
        WHERE customer_id = NEW.customer_id;

        -- Record transaction
        INSERT INTO points_transactions (
            customer_id,
            points_redeemed,
            transaction_type,
            reference_id,
            description
        ) VALUES (
            NEW.customer_id,
            NEW.points_redeemed,
            'REDEMPTION',
            NEW.redemption_id,
            NEW.description
        );
    ELSE
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Insufficient loyalty points for redemption';
    END IF;
END //
DELIMITER ;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between BEFORE and AFTER triggers?"**
    - BEFORE: Can modify NEW values, prevent operations, validation

- AFTER: Cannot modify data, used for logging, cascading updates
- INSTEAD OF: Replace the operation (views only)

2. **"How do you handle recursive triggers?"**
   - Use conditional logic to prevent infinite loops
   - Set flags or use session variables
   - Consider using statement-level triggers
3. **"What are the performance implications of triggers?"**
   - Triggers add overhead to DML operations
   - Can slow down bulk operations
   - Consider batching and asynchronous processing
4. **"When would you use triggers vs stored procedures?"**
   - Triggers: Automatic, event-driven, data integrity
   - Procedures: Manual execution, complex business logic, better control

Best Practices:

1. **Keep triggers simple and fast**
2. **Avoid complex business logic in triggers**
3. **Use proper error handling**
4. **Document trigger behavior thoroughly**
5. **Test trigger interactions carefully**
6. **Consider alternatives for heavy processing**

⚠ Things to Watch Out For

1. **Recursive Triggers**

```
-- Problem: Infinite loop
CREATE TRIGGER update_timestamp
    BEFORE UPDATE ON orders
    FOR EACH ROW
    SET NEW.updated_at = NOW();

-- This trigger fires on every update, including its own!

-- Solution: Add condition
DELIMITER //
CREATE TRIGGER update_timestamp_safe
    BEFORE UPDATE ON orders
    FOR EACH ROW
BEGIN
    IF NEW.updated_at = OLD.updated_at THEN
        SET NEW.updated_at = NOW();
    END IF;
```

```
END //
DELIMITER ;
```

2. **Performance Issues with Bulk Operations**

```
-- Problem: Row-level trigger on bulk insert
-- This will fire once for each of 10,000 rows
INSERT INTO orders (customer_id, total_amount)
SELECT customer_id, 100
FROM customers
LIMIT 10000;

-- Solution: Use statement-level triggers (PostgreSQL)
CREATE TRIGGER bulk_operation_log
    AFTER INSERT ON orders
    FOR EACH STATEMENT
    EXECUTE FUNCTION log_bulk_insert();
```

3. **Transaction Rollback Issues**

```
-- Be careful with error handling in triggers
CREATE OR REPLACE FUNCTION risky_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    -- This could cause the entire transaction to rollback
    INSERT INTO external_api_log VALUES (...);

    -- Better: Use exception handling
    BEGIN
        INSERT INTO external_api_log VALUES (...);
    EXCEPTION
        WHEN OTHERS THEN
            -- Log error but don't fail the main operation
            INSERT INTO error_log VALUES (SQLERRM);
    END;

    RETURN NEW;
END;
$$;
```

4. **Trigger Order Dependencies**

```
-- Multiple triggers on same table can have order issues
-- MySQL: Use naming convention or FOLLOWS/PRECEDES (5.7+)
```

```
CREATE TRIGGER audit_first
    AFTER UPDATE ON customers
    FOR EACH ROW
    -- This should run first
    INSERT INTO audit_log VALUES (...);

CREATE TRIGGER update_stats_second
    AFTER UPDATE ON customers
    FOR EACH ROW
    FOLLOWS audit_first  -- MySQL 5.7+
    -- This runs after audit_first
    UPDATE customer_stats SET ...;
```

🚀 Next Steps

In the next chapter, we'll explore **Views and CTEs (Common Table Expressions)** - powerful tools for creating virtual tables, simplifying complex queries, and organizing data access. You'll learn how to create updatable views, recursive CTEs, and use them for security and abstraction.

📝 Quick Practice

Try these trigger and event exercises:

1. **Audit System**: Create a comprehensive audit trail for all table changes
2. **Inventory Management**: Build triggers to automatically manage product stock levels
3. **Business Rules**: Implement complex validation rules using triggers
4. **Scheduled Maintenance**: Create events for database cleanup and optimization
5. **Real-time Analytics**: Use triggers to maintain real-time summary tables

Consider:

- When to use triggers vs application logic
- Performance implications of trigger complexity
- Error handling and transaction management
- Testing strategies for trigger-dependent systems
- Alternatives like message queues for heavy processing

Chapter 14: Views and CTEs (Common Table Expressions)

🗄 What You'll Learn

Views and CTEs are powerful tools for creating virtual tables, simplifying complex queries, and organizing data access. Views provide a persistent virtual layer over your data, while CTEs offer temporary, query-scoped virtual tables. This chapter covers creating, managing, and optimizing both for better database design and query organization.

🎯 Learning Objectives

By the end of this chapter, you will:

- Understand the differences between views and CTEs
- Create simple and complex views with proper security
- Build recursive and non-recursive CTEs
- Implement updatable views and understand their limitations
- Use views for data abstraction and security
- Optimize view and CTE performance
- Handle view dependencies and maintenance

🔍 Concept Explanation

What are Views?

Views are virtual tables that don't store data themselves but provide a saved query that can be referenced like a table. They're useful for:

- **Data Abstraction**: Hide complex joins and calculations
- **Security**: Restrict access to specific columns/rows
- **Simplification**: Provide user-friendly interfaces to complex data
- **Consistency**: Ensure consistent data access patterns

Types of Views:

- **Simple Views**: Based on single table, usually updatable
- **Complex Views**: Multiple tables, aggregations, may not be updatable
- **Materialized Views**: Physically store results (PostgreSQL)
- **Indexed Views**: Materialized with indexes (SQL Server)

What are CTEs?

Common Table Expressions (CTEs) are temporary named result sets that exist only within the scope of a single query. They're excellent for:

- **Query Organization**: Break complex queries into readable parts
- **Recursive Operations**: Tree traversal, hierarchical data
- **Avoiding Repetition**: Reference the same subquery multiple times
- **Readability**: Make complex logic more understandable

Types of CTEs:

- **Non-Recursive**: Simple temporary result sets
- **Recursive**: Self-referencing for hierarchical data
- **Multiple CTEs**: Several CTEs in one query

🛠️ Syntax Comparison

MySQL vs PostgreSQL Views

Feature	MySQL	PostgreSQL
Basic Views	✅	✅
Updatable Views	✅ (limited)	✅ (more flexible)
Materialized Views	❌	✅
Recursive Views	❌	✅ (via recursive CTEs)
View Dependencies	Limited tracking	Full dependency tracking
Security Options	Basic	Advanced (RLS, policies)
WITH CHECK OPTION	✅	✅
CTEs	✅ (8.0+)	✅
Recursive CTEs	✅ (8.0+)	✅

👁 Basic Views

Creating Simple Views

MySQL:

```
-- Simple customer view with calculated fields
CREATE VIEW customer_summary AS
SELECT
    customer_id,
    CONCAT(first_name, ' ', last_name) AS full_name,
    email,
    phone,
    status,
    created_at,
    DATEDIFF(CURDATE(), created_at) AS days_since_registration
FROM customers
WHERE status = 'active';

-- Product catalog view with pricing
CREATE VIEW product_catalog AS
SELECT
    p.product_id,
    p.product_name,
    p.sku,
    c.category_name,
    p.price,
    p.stock_quantity,
    CASE
        WHEN p.stock_quantity = 0 THEN 'Out of Stock'
        WHEN p.stock_quantity < 10 THEN 'Low Stock'
        ELSE 'In Stock'
```

```
    END AS stock_status,
    p.created_at
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.status = 'active';

-- Order details view with customer info
CREATE VIEW order_details_view AS
SELECT
    o.order_id,
    o.order_date,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    c.email AS customer_email,
    o.status,
    o.total_amount,
    COUNT(oi.order_item_id) AS item_count,
    GROUP_CONCAT(p.product_name SEPARATOR ', ') AS products
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
GROUP BY o.order_id, o.order_date, c.first_name, c.last_name, c.email, o.status,
o.total_amount;

-- Using views
SELECT * FROM customer_summary WHERE days_since_registration > 365;

SELECT * FROM product_catalog WHERE stock_status = 'Low Stock';

SELECT * FROM order_details_view WHERE order_date >= '2024-01-01';
```

PostgreSQL:

```
-- Customer analytics view with window functions
CREATE VIEW customer_analytics AS
SELECT
    c.customer_id,
    c.first_name || ' ' || c.last_name AS full_name,
    c.email,
    c.status,
    c.created_at,
    EXTRACT(DAYS FROM (CURRENT_DATE - c.created_at)) AS days_since_registration,
    COUNT(o.order_id) AS total_orders,
    COALESCE(SUM(o.total_amount), 0) AS total_spent,
    COALESCE(AVG(o.total_amount), 0) AS avg_order_value,
    MAX(o.order_date) AS last_order_date,
    RANK() OVER (ORDER BY COALESCE(SUM(o.total_amount), 0) DESC) AS spending_rank
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id AND o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name, c.email, c.status,
c.created_at;
```

```sql
-- Product performance view
CREATE VIEW product_performance AS
SELECT
    p.product_id,
    p.product_name,
    p.sku,
    c.category_name,
    p.price,
    p.stock_quantity,
    COUNT(oi.order_item_id) AS times_ordered,
    SUM(oi.quantity) AS total_quantity_sold,
    SUM(oi.quantity * oi.unit_price) AS total_revenue,
    AVG(oi.unit_price) AS avg_selling_price,
    (p.price - AVG(oi.unit_price)) AS avg_discount,
    CASE
        WHEN COUNT(oi.order_item_id) = 0 THEN 'Never Ordered'
        WHEN COUNT(oi.order_item_id) < 5 THEN 'Low Demand'
        WHEN COUNT(oi.order_item_id) < 20 THEN 'Medium Demand'
        ELSE 'High Demand'
    END AS demand_category
FROM products p
JOIN categories c ON p.category_id = c.category_id
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name, p.sku, c.category_name, p.price,
p.stock_quantity;

-- Monthly sales summary view
CREATE VIEW monthly_sales_summary AS
SELECT
    DATE_TRUNC('month', o.order_date) AS month,
    COUNT(DISTINCT o.order_id) AS total_orders,
    COUNT(DISTINCT o.customer_id) AS unique_customers,
    SUM(o.total_amount) AS total_revenue,
    AVG(o.total_amount) AS avg_order_value,
    SUM(oi.quantity) AS total_items_sold,
    COUNT(DISTINCT oi.product_id) AS unique_products_sold
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
WHERE o.status = 'completed'
GROUP BY DATE_TRUNC('month', o.order_date)
ORDER BY month;
```

Updatable Views

MySQL:

```sql
-- Simple updatable view
CREATE VIEW active_customers AS
SELECT
    customer_id,
    first_name,
```

```
    last_name,
    email,
    phone,
    status
FROM customers
WHERE status = 'active'
WITH CHECK OPTION;

-- Update through view
UPDATE active_customers
SET phone = '555-0123'
WHERE customer_id = 1;

-- Insert through view (will enforce WHERE condition due to WITH CHECK OPTION)
INSERT INTO active_customers (first_name, last_name, email, status)
VALUES ('John', 'Doe', 'john.doe@email.com', 'active');

-- This would fail due to WITH CHECK OPTION
-- INSERT INTO active_customers (first_name, last_name, email, status)
-- VALUES ('Jane', 'Smith', 'jane@email.com', 'inactive');

-- View with calculated columns (not updatable)
CREATE VIEW customer_stats AS
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    (SELECT COUNT(*) FROM orders WHERE customer_id = c.customer_id) AS
order_count,
    (SELECT SUM(total_amount) FROM orders WHERE customer_id = c.customer_id) AS
total_spent
FROM customers c;

-- This view is not updatable due to subqueries
```

PostgreSQL:

```
-- Updatable view with rules
CREATE VIEW active_products AS
SELECT
    product_id,
    product_name,
    sku,
    price,
    stock_quantity,
    category_id
FROM products
WHERE status = 'active';

-- Create rules for complex update behavior
CREATE OR REPLACE RULE active_products_update AS
```

```
    ON UPDATE TO active_products
    DO INSTEAD
    UPDATE products
    SET
        product_name = NEW.product_name,
        sku = NEW.sku,
        price = NEW.price,
        stock_quantity = NEW.stock_quantity,
        category_id = NEW.category_id,
        updated_at = CURRENT_TIMESTAMP
    WHERE product_id = OLD.product_id
      AND status = 'active';

CREATE OR REPLACE RULE active_products_insert AS
    ON INSERT TO active_products
    DO INSTEAD
    INSERT INTO products (product_name, sku, price, stock_quantity, category_id,
status, created_at)
    VALUES (NEW.product_name, NEW.sku, NEW.price, NEW.stock_quantity,
NEW.category_id, 'active', CURRENT_TIMESTAMP);

CREATE OR REPLACE RULE active_products_delete AS
    ON DELETE TO active_products
    DO INSTEAD
    UPDATE products
    SET status = 'inactive', updated_at = CURRENT_TIMESTAMP
    WHERE product_id = OLD.product_id;

-- Using the updatable view
INSERT INTO active_products (product_name, sku, price, stock_quantity,
category_id)
VALUES ('New Product', 'NP001', 29.99, 100, 1);

UPDATE active_products
SET price = 24.99
WHERE sku = 'NP001';

-- "Delete" (actually sets status to inactive)
DELETE FROM active_products WHERE sku = 'NP001';
```

🔄 Common Table Expressions (CTEs)

Non-Recursive CTEs

MySQL 8.0+ and PostgreSQL:

```
-- Single CTE for customer segmentation
WITH customer_segments AS (
    SELECT
        c.customer_id,
```

```
        c.first_name,
        c.last_name,
        c.email,
        COUNT(o.order_id) AS order_count,
        SUM(o.total_amount) AS total_spent,
        CASE
            WHEN SUM(o.total_amount) >= 1000 THEN 'VIP'
            WHEN SUM(o.total_amount) >= 500 THEN 'Premium'
            WHEN SUM(o.total_amount) >= 100 THEN 'Regular'
            ELSE 'New'
        END AS segment
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id AND o.status = 'completed'
    GROUP BY c.customer_id, c.first_name, c.last_name, c.email
)
SELECT
    segment,
    COUNT(*) AS customer_count,
    AVG(total_spent) AS avg_spent,
    MIN(total_spent) AS min_spent,
    MAX(total_spent) AS max_spent
FROM customer_segments
GROUP BY segment
ORDER BY avg_spent DESC;

-- Multiple CTEs for complex analysis
WITH
-- CTE 1: Monthly sales
monthly_sales AS (
    SELECT
        DATE_FORMAT(order_date, '%Y-%m') AS month,
        SUM(total_amount) AS monthly_revenue,
        COUNT(*) AS monthly_orders
    FROM orders
    WHERE status = 'completed'
      AND order_date >= DATE_SUB(CURDATE(), INTERVAL 12 MONTH)
    GROUP BY DATE_FORMAT(order_date, '%Y-%m')
),
-- CTE 2: Product categories performance
category_performance AS (
    SELECT
        c.category_name,
        SUM(oi.quantity * oi.unit_price) AS category_revenue,
        SUM(oi.quantity) AS items_sold
    FROM categories c
    JOIN products p ON c.category_id = p.category_id
    JOIN order_items oi ON p.product_id = oi.product_id
    JOIN orders o ON oi.order_id = o.order_id
    WHERE o.status = 'completed'
      AND o.order_date >= DATE_SUB(CURDATE(), INTERVAL 12 MONTH)
    GROUP BY c.category_id, c.category_name
),
-- CTE 3: Customer acquisition by month
customer_acquisition AS (
```

```sql
    SELECT
        DATE_FORMAT(created_at, '%Y-%m') AS month,
        COUNT(*) AS new_customers
    FROM customers
    WHERE created_at >= DATE_SUB(CURDATE(), INTERVAL 12 MONTH)
    GROUP BY DATE_FORMAT(created_at, '%Y-%m')
)
-- Main query combining all CTEs
SELECT
    ms.month,
    ms.monthly_revenue,
    ms.monthly_orders,
    ca.new_customers,
    ROUND(ms.monthly_revenue / ms.monthly_orders, 2) AS avg_order_value,
    ROUND(ms.monthly_revenue / ca.new_customers, 2) AS revenue_per_new_customer
FROM monthly_sales ms
LEFT JOIN customer_acquisition ca ON ms.month = ca.month
ORDER BY ms.month;

-- CTE for window function calculations
WITH sales_with_running_totals AS (
    SELECT
        order_date,
        total_amount,
        SUM(total_amount) OVER (
            ORDER BY order_date
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS running_total,
        AVG(total_amount) OVER (
            ORDER BY order_date
            ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
        ) AS moving_avg_7_days,
        LAG(total_amount, 1) OVER (ORDER BY order_date) AS prev_day_sales,
        LEAD(total_amount, 1) OVER (ORDER BY order_date) AS next_day_sales
    FROM (
        SELECT
            DATE(order_date) AS order_date,
            SUM(total_amount) AS total_amount
        FROM orders
        WHERE status = 'completed'
        GROUP BY DATE(order_date)
    ) daily_sales
)
SELECT
    order_date,
    total_amount,
    running_total,
    ROUND(moving_avg_7_days, 2) AS moving_avg_7_days,
    ROUND(((total_amount - prev_day_sales) / prev_day_sales) * 100, 2) AS
day_over_day_growth
FROM sales_with_running_totals
WHERE order_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)
ORDER BY order_date;
```

Recursive CTEs

Organizational Hierarchy:

```sql
-- Create employee hierarchy table
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    title VARCHAR(100),
    manager_id INT,
    salary DECIMAL(10,2),
    hire_date DATE,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);

-- Sample data
INSERT INTO employees VALUES
(1, 'John', 'CEO', 'Chief Executive Officer', NULL, 200000, '2020-01-01'),
(2, 'Jane', 'Smith', 'VP Sales', 1, 150000, '2020-02-01'),
(3, 'Bob', 'Johnson', 'VP Engineering', 1, 160000, '2020-02-15'),
(4, 'Alice', 'Brown', 'Sales Manager', 2, 80000, '2020-03-01'),
(5, 'Charlie', 'Davis', 'Senior Developer', 3, 90000, '2020-03-15'),
(6, 'Diana', 'Wilson', 'Sales Rep', 4, 50000, '2020-04-01'),
(7, 'Eve', 'Miller', 'Junior Developer', 5, 60000, '2020-05-01');

-- Recursive CTE to get full hierarchy
WITH RECURSIVE employee_hierarchy AS (
    -- Base case: top-level employees (CEOs)
    SELECT
        employee_id,
        first_name,
        last_name,
        title,
        manager_id,
        salary,
        0 AS level,
        CAST(first_name || ' ' || last_name AS VARCHAR(1000)) AS hierarchy_path,
        CAST(employee_id AS VARCHAR(1000)) AS id_path
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive case: employees with managers
    SELECT
        e.employee_id,
        e.first_name,
        e.last_name,
        e.title,
        e.manager_id,
        e.salary,
```

```
        eh.level + 1,
        eh.hierarchy_path || ' -> ' || e.first_name || ' ' || e.last_name,
        eh.id_path || ',' || e.employee_id
    FROM employees e
    JOIN employee_hierarchy eh ON e.manager_id = eh.employee_id
)
SELECT
    REPEAT('  ', level) || first_name || ' ' || last_name AS indented_name,
    title,
    salary,
    level,
    hierarchy_path
FROM employee_hierarchy
ORDER BY id_path;

-- Get all subordinates of a specific manager
WITH RECURSIVE subordinates AS (
    -- Start with the manager
    SELECT
        employee_id,
        first_name,
        last_name,
        title,
        manager_id,
        salary,
        0 AS level
    FROM employees
    WHERE employee_id = 2  -- VP Sales

    UNION ALL

    -- Get all direct and indirect reports
    SELECT
        e.employee_id,
        e.first_name,
        e.last_name,
        e.title,
        e.manager_id,
        e.salary,
        s.level + 1
    FROM employees e
    JOIN subordinates s ON e.manager_id = s.employee_id
)
SELECT
    first_name || ' ' || last_name AS name,
    title,
    salary,
    level,
    CASE
        WHEN level = 0 THEN 'Manager'
        WHEN level = 1 THEN 'Direct Report'
        ELSE 'Indirect Report (Level ' || level || ')'
    END AS relationship
FROM subordinates
```

```
ORDER BY level, last_name;

-- Calculate total team salary for each manager
WITH RECURSIVE team_hierarchy AS (
    SELECT
        employee_id,
        first_name,
        last_name,
        title,
        manager_id,
        salary,
        employee_id AS root_manager
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    SELECT
        e.employee_id,
        e.first_name,
        e.last_name,
        e.title,
        e.manager_id,
        e.salary,
        th.root_manager
    FROM employees e
    JOIN team_hierarchy th ON e.manager_id = th.employee_id
)
SELECT
    m.first_name || ' ' || m.last_name AS manager_name,
    m.title AS manager_title,
    COUNT(th.employee_id) - 1 AS team_size,  -- Subtract 1 to exclude manager
    SUM(th.salary) AS total_team_salary,
    AVG(th.salary) AS avg_team_salary,
    m.salary AS manager_salary
FROM team_hierarchy th
JOIN employees m ON th.root_manager = m.employee_id
GROUP BY m.employee_id, m.first_name, m.last_name, m.title, m.salary
ORDER BY total_team_salary DESC;
```

Category Tree Navigation:

```
-- Create category hierarchy table
CREATE TABLE category_tree (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(100),
    parent_category_id INT,
    sort_order INT DEFAULT 0,
    FOREIGN KEY (parent_category_id) REFERENCES category_tree(category_id)
);

-- Sample hierarchical categories
```

```sql
INSERT INTO category_tree VALUES
(1, 'Electronics', NULL, 1),
(2, 'Clothing', NULL, 2),
(3, 'Home & Garden', NULL, 3),
(4, 'Computers', 1, 1),
(5, 'Mobile Phones', 1, 2),
(6, 'Audio', 1, 3),
(7, 'Laptops', 4, 1),
(8, 'Desktops', 4, 2),
(9, 'Gaming Laptops', 7, 1),
(10, 'Business Laptops', 7, 2),
(11, 'Men\'s Clothing', 2, 1),
(12, 'Women\'s Clothing', 2, 2),
(13, 'Shirts', 11, 1),
(14, 'Pants', 11, 2);

-- Get full category tree with breadcrumbs
WITH RECURSIVE category_breadcrumbs AS (
    -- Root categories
    SELECT
        category_id,
        category_name,
        parent_category_id,
        sort_order,
        0 AS level,
        category_name AS breadcrumb,
        LPAD('', 0, '  ') || category_name AS indented_name,
        CAST(sort_order AS VARCHAR(1000)) AS sort_path
    FROM category_tree
    WHERE parent_category_id IS NULL

    UNION ALL

    -- Child categories
    SELECT
        ct.category_id,
        ct.category_name,
        ct.parent_category_id,
        ct.sort_order,
        cb.level + 1,
        cb.breadcrumb || ' > ' || ct.category_name,
        LPAD('', (cb.level + 1) * 2, '  ') || ct.category_name,
        cb.sort_path || '.' || LPAD(ct.sort_order::TEXT, 3, '0')
    FROM category_tree ct
    JOIN category_breadcrumbs cb ON ct.parent_category_id = cb.category_id
)
SELECT
    category_id,
    indented_name AS category_hierarchy,
    breadcrumb,
    level
FROM category_breadcrumbs
ORDER BY sort_path;
```

```sql
-- Find all products in a category and its subcategories
WITH RECURSIVE category_descendants AS (
    -- Start with specific category
    SELECT category_id
    FROM category_tree
    WHERE category_id = 1  -- Electronics

    UNION ALL

    -- Get all descendant categories
    SELECT ct.category_id
    FROM category_tree ct
    JOIN category_descendants cd ON ct.parent_category_id = cd.category_id
)
SELECT
    p.product_id,
    p.product_name,
    p.price,
    ct.category_name,
    cb.breadcrumb AS category_path
FROM products p
JOIN category_descendants cd ON p.category_id = cd.category_id
JOIN category_tree ct ON p.category_id = ct.category_id
JOIN (
    WITH RECURSIVE category_breadcrumbs AS (
        SELECT
            category_id,
            category_name,
            parent_category_id,
            category_name AS breadcrumb
        FROM category_tree
        WHERE parent_category_id IS NULL

        UNION ALL

        SELECT
            ct.category_id,
            ct.category_name,
            ct.parent_category_id,
            cb.breadcrumb || ' > ' || ct.category_name
        FROM category_tree ct
        JOIN category_breadcrumbs cb ON ct.parent_category_id = cb.category_id
    )
    SELECT category_id, breadcrumb FROM category_breadcrumbs
) cb ON p.category_id = cb.category_id
ORDER BY cb.breadcrumb, p.product_name;
```

🏗️ Materialized Views (PostgreSQL)

```sql
-- Create materialized view for expensive analytics
CREATE MATERIALIZED VIEW customer_analytics_materialized AS
SELECT
    c.customer_id,
    c.first_name || ' ' || c.last_name AS full_name,
    c.email,
    c.created_at AS registration_date,
    COUNT(o.order_id) AS total_orders,
    COALESCE(SUM(o.total_amount), 0) AS lifetime_value,
    COALESCE(AVG(o.total_amount), 0) AS avg_order_value,
    MAX(o.order_date) AS last_order_date,
    MIN(o.order_date) AS first_order_date,
    EXTRACT(DAYS FROM (MAX(o.order_date) - MIN(o.order_date))) AS
customer_lifespan_days,
    COUNT(DISTINCT DATE_TRUNC('month', o.order_date)) AS active_months,
    CASE
        WHEN COUNT(o.order_id) = 0 THEN 'Never Purchased'
        WHEN MAX(o.order_date) < CURRENT_DATE - INTERVAL '6 months' THEN
'Inactive'
        WHEN COUNT(o.order_id) = 1 THEN 'One-time Buyer'
        WHEN COUNT(o.order_id) BETWEEN 2 AND 5 THEN 'Occasional Buyer'
        WHEN COUNT(o.order_id) BETWEEN 6 AND 15 THEN 'Regular Customer'
        ELSE 'VIP Customer'
    END AS customer_segment,
    -- RFM Analysis components
    EXTRACT(DAYS FROM (CURRENT_DATE - MAX(o.order_date))) AS recency_days,
    COUNT(o.order_id) AS frequency,
    COALESCE(SUM(o.total_amount), 0) AS monetary
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id AND o.status = 'completed'
GROUP BY c.customer_id, c.first_name, c.last_name, c.email, c.created_at;

-- Create index on materialized view
CREATE INDEX idx_customer_analytics_segment ON
customer_analytics_materialized(customer_segment);
CREATE INDEX idx_customer_analytics_ltv ON
customer_analytics_materialized(lifetime_value);
CREATE INDEX idx_customer_analytics_recency ON
customer_analytics_materialized(recency_days);

-- Refresh materialized view
REFRESH MATERIALIZED VIEW customer_analytics_materialized;

-- Concurrent refresh (non-blocking)
REFRESH MATERIALIZED VIEW CONCURRENTLY customer_analytics_materialized;

-- Create materialized view for product recommendations
CREATE MATERIALIZED VIEW product_recommendations AS
WITH product_pairs AS (
    SELECT
        oi1.product_id AS product_a,
        oi2.product_id AS product_b,
        COUNT(*) AS co_occurrence_count
```

```
    FROM order_items oi1
    JOIN order_items oi2 ON oi1.order_id = oi2.order_id
    WHERE oi1.product_id < oi2.product_id  -- Avoid duplicates
    GROUP BY oi1.product_id, oi2.product_id
    HAVING COUNT(*) >= 3  -- Minimum co-occurrence threshold
),
product_stats AS (
    SELECT
        product_id,
        COUNT(DISTINCT order_id) AS total_orders
    FROM order_items
    GROUP BY product_id
)
SELECT
    pp.product_a,
    pa.product_name AS product_a_name,
    pp.product_b,
    pb.product_name AS product_b_name,
    pp.co_occurrence_count,
    psa.total_orders AS product_a_orders,
    psb.total_orders AS product_b_orders,
    ROUND(
        pp.co_occurrence_count::DECIMAL / LEAST(psa.total_orders,
psb.total_orders),
        3
    ) AS confidence_score,
    ROUND(
        pp.co_occurrence_count::DECIMAL / (psa.total_orders + psb.total_orders -
pp.co_occurrence_count),
        3
    ) AS jaccard_similarity
FROM product_pairs pp
JOIN products pa ON pp.product_a = pa.product_id
JOIN products pb ON pp.product_b = pb.product_id
JOIN product_stats psa ON pp.product_a = psa.product_id
JOIN product_stats psb ON pp.product_b = psb.product_id
ORDER BY confidence_score DESC, co_occurrence_count DESC;

-- Schedule automatic refresh using pg_cron
SELECT cron.schedule('refresh-analytics', '0 2 * * *',
    'REFRESH MATERIALIZED VIEW CONCURRENTLY customer_analytics_materialized;');

SELECT cron.schedule('refresh-recommendations', '0 3 * * 0',
    'REFRESH MATERIALIZED VIEW CONCURRENTLY product_recommendations;');
```

Security Views

```
-- PostgreSQL Row Level Security with Views

-- Create user roles
```

```sql
CREATE ROLE sales_manager;
CREATE ROLE sales_rep;
CREATE ROLE customer_service;

-- Create secure customer view for customer service
CREATE VIEW customer_service_view AS
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    phone,
    status,
    created_at,
    -- Mask sensitive information
    CASE
        WHEN LENGTH(phone) >= 10 THEN
            SUBSTRING(phone, 1, 3) || '-XXX-' || SUBSTRING(phone, -4)
        ELSE 'XXX-XXXX'
    END AS masked_phone
FROM customers
WHERE status IN ('active', 'inactive');  -- Exclude deleted customers

-- Grant appropriate permissions
GRANT SELECT ON customer_service_view TO customer_service;

-- Create sales view with territory restrictions
CREATE VIEW sales_territory_view AS
SELECT
    o.order_id,
    o.order_date,
    o.status,
    o.total_amount,
    c.customer_id,
    c.first_name || ' ' || c.last_name AS customer_name,
    c.email,
    c.territory,
    u.username AS sales_rep
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN users u ON o.sales_rep_id = u.user_id
WHERE
    -- Only show orders for current user's territory
    c.territory = COALESCE(
        (SELECT territory FROM users WHERE username = current_user),
        c.territory  -- Fallback for managers
    )
    OR
    -- Or if user is a manager
    EXISTS (
        SELECT 1 FROM users
        WHERE username = current_user
          AND role = 'sales_manager'
    );
```

```sql
GRANT SELECT ON sales_territory_view TO sales_rep, sales_manager;

-- Create financial summary view for managers only
CREATE VIEW financial_summary_view AS
SELECT
    DATE_TRUNC('month', order_date) AS month,
    COUNT(*) AS total_orders,
    SUM(total_amount) AS total_revenue,
    AVG(total_amount) AS avg_order_value,
    SUM(CASE WHEN status = 'completed' THEN total_amount ELSE 0 END) AS
completed_revenue,
    SUM(CASE WHEN status = 'cancelled' THEN total_amount ELSE 0 END) AS
cancelled_revenue,
    ROUND(
        SUM(CASE WHEN status = 'completed' THEN total_amount ELSE 0 END) /
        NULLIF(SUM(total_amount), 0) * 100,
        2
    ) AS completion_rate
FROM orders
WHERE order_date >= DATE_TRUNC('year', CURRENT_DATE)
GROUP BY DATE_TRUNC('month', order_date)
ORDER BY month;

GRANT SELECT ON financial_summary_view TO sales_manager;

-- MySQL equivalent using views for security
-- (MySQL doesn't have RLS, so we use application-level security)

CREATE VIEW mysql_customer_service_view AS
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    CASE
        WHEN CHAR_LENGTH(phone) >= 10 THEN
            CONCAT(LEFT(phone, 3), '-XXX-', RIGHT(phone, 4))
        ELSE 'XXX-XXXX'
    END AS masked_phone,
    status,
    created_at
FROM customers
WHERE status IN ('active', 'inactive');

-- Create application user context table
CREATE TABLE user_context (
    session_id VARCHAR(255) PRIMARY KEY,
    user_id INT,
    username VARCHAR(100),
    role VARCHAR(50),
    territory VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    expires_at TIMESTAMP
```

```
);

-- Territory-based sales view (requires application to set context)
CREATE VIEW mysql_sales_territory_view AS
SELECT
    o.order_id,
    o.order_date,
    o.status,
    o.total_amount,
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    c.email,
    c.territory
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN user_context uc ON uc.expires_at > NOW()
WHERE
    (c.territory = uc.territory OR uc.role = 'sales_manager')
    AND uc.session_id = @session_id;  -- Application sets this variable
```

Real-World Business Examples

Example 1: Executive Dashboard Views

```
-- PostgreSQL: Comprehensive executive dashboard
CREATE VIEW executive_dashboard AS
WITH
-- Current month metrics
current_month AS (
    SELECT
        COUNT(DISTINCT o.order_id) AS orders_this_month,
        COUNT(DISTINCT o.customer_id) AS customers_this_month,
        SUM(o.total_amount) AS revenue_this_month,
        AVG(o.total_amount) AS avg_order_this_month
    FROM orders o
    WHERE DATE_TRUNC('month', o.order_date) = DATE_TRUNC('month', CURRENT_DATE)
      AND o.status = 'completed'
),
-- Previous month metrics
previous_month AS (
    SELECT
        COUNT(DISTINCT o.order_id) AS orders_last_month,
        COUNT(DISTINCT o.customer_id) AS customers_last_month,
        SUM(o.total_amount) AS revenue_last_month,
        AVG(o.total_amount) AS avg_order_last_month
    FROM orders o
    WHERE DATE_TRUNC('month', o.order_date) = DATE_TRUNC('month', CURRENT_DATE) -
INTERVAL '1 month'
      AND o.status = 'completed'
),
```

```sql
-- Year-to-date metrics
ytd_metrics AS (
    SELECT
        COUNT(DISTINCT o.order_id) AS orders_ytd,
        COUNT(DISTINCT o.customer_id) AS customers_ytd,
        SUM(o.total_amount) AS revenue_ytd,
        AVG(o.total_amount) AS avg_order_ytd
    FROM orders o
    WHERE DATE_TRUNC('year', o.order_date) = DATE_TRUNC('year', CURRENT_DATE)
      AND o.status = 'completed'
),
-- Customer metrics
customer_metrics AS (
    SELECT
        COUNT(*) AS total_customers,
        COUNT(CASE WHEN created_at >= DATE_TRUNC('month', CURRENT_DATE) THEN 1
END) AS new_customers_this_month,
        COUNT(CASE WHEN status = 'active' THEN 1 END) AS active_customers
    FROM customers
),
-- Product metrics
product_metrics AS (
    SELECT
        COUNT(*) AS total_products,
        COUNT(CASE WHEN stock_quantity = 0 THEN 1 END) AS out_of_stock_products,
        COUNT(CASE WHEN stock_quantity < 10 THEN 1 END) AS low_stock_products
    FROM products
    WHERE status = 'active'
)
SELECT
    -- Current month
    cm.orders_this_month,
    cm.customers_this_month,
    cm.revenue_this_month,
    ROUND(cm.avg_order_this_month, 2) AS avg_order_this_month,

    -- Month-over-month growth
    ROUND(
        ((cm.orders_this_month - pm.orders_last_month)::DECIMAL /
         NULLIF(pm.orders_last_month, 0)) * 100, 2
    ) AS orders_mom_growth,
    ROUND(
        ((cm.revenue_this_month - pm.revenue_last_month)::DECIMAL /
         NULLIF(pm.revenue_last_month, 0)) * 100, 2
    ) AS revenue_mom_growth,

    -- Year-to-date
    ytd.orders_ytd,
    ytd.revenue_ytd,
    ROUND(ytd.avg_order_ytd, 2) AS avg_order_ytd,

    -- Customer metrics
    cust.total_customers,
    cust.new_customers_this_month,
```

```
    cust.active_customers,
    ROUND(
        (cust.active_customers::DECIMAL / cust.total_customers) * 100, 2
    ) AS customer_retention_rate,

    -- Product metrics
    prod.total_products,
    prod.out_of_stock_products,
    prod.low_stock_products,
    ROUND(
        ((prod.total_products - prod.out_of_stock_products)::DECIMAL /
         prod.total_products) * 100, 2
    ) AS product_availability_rate

FROM current_month cm
CROSS JOIN previous_month pm
CROSS JOIN ytd_metrics ytd
CROSS JOIN customer_metrics cust
CROSS JOIN product_metrics prod;

-- Create view for top performing products
CREATE VIEW top_products_dashboard AS
WITH product_performance AS (
    SELECT
        p.product_id,
        p.product_name,
        p.sku,
        c.category_name,
        p.price,
        p.stock_quantity,
        COUNT(oi.order_item_id) AS times_ordered,
        SUM(oi.quantity) AS total_quantity_sold,
        SUM(oi.quantity * oi.unit_price) AS total_revenue,
        AVG(oi.unit_price) AS avg_selling_price,
        RANK() OVER (ORDER BY SUM(oi.quantity * oi.unit_price) DESC) AS
revenue_rank,
        RANK() OVER (ORDER BY SUM(oi.quantity) DESC) AS quantity_rank
    FROM products p
    JOIN categories c ON p.category_id = c.category_id
    LEFT JOIN order_items oi ON p.product_id = oi.product_id
    LEFT JOIN orders o ON oi.order_id = o.order_id
        AND o.status = 'completed'
        AND o.order_date >= DATE_TRUNC('month', CURRENT_DATE) - INTERVAL '11
months'
    WHERE p.status = 'active'
    GROUP BY p.product_id, p.product_name, p.sku, c.category_name, p.price,
p.stock_quantity
)
SELECT
    product_id,
    product_name,
    sku,
    category_name,
    price,
```

```
    stock_quantity,
    times_ordered,
    total_quantity_sold,
    ROUND(total_revenue, 2) AS total_revenue,
    ROUND(avg_selling_price, 2) AS avg_selling_price,
    revenue_rank,
    quantity_rank,
    CASE
        WHEN stock_quantity = 0 THEN 'Out of Stock'
        WHEN stock_quantity < 10 THEN 'Low Stock'
        WHEN stock_quantity < 50 THEN 'Medium Stock'
        ELSE 'Well Stocked'
    END AS stock_status
FROM product_performance
WHERE revenue_rank <= 50 OR quantity_rank <= 50
ORDER BY total_revenue DESC;
```

Example 2: Customer 360 View

```
-- Comprehensive customer 360 view
CREATE VIEW customer_360_view AS
WITH
-- Customer order history
customer_orders AS (
    SELECT
        customer_id,
        COUNT(*) AS total_orders,
        SUM(total_amount) AS lifetime_value,
        AVG(total_amount) AS avg_order_value,
        MIN(order_date) AS first_order_date,
        MAX(order_date) AS last_order_date,
        COUNT(CASE WHEN status = 'completed' THEN 1 END) AS completed_orders,
        COUNT(CASE WHEN status = 'cancelled' THEN 1 END) AS cancelled_orders,
        SUM(CASE WHEN order_date >= CURRENT_DATE - INTERVAL '12 months' THEN
total_amount ELSE 0 END) AS value_12m
    FROM orders
    GROUP BY customer_id
),
-- Customer product preferences
customer_categories AS (
    SELECT
        o.customer_id,
        c.category_name,
        COUNT(*) AS category_orders,
        SUM(oi.quantity * oi.unit_price) AS category_spend,
        ROW_NUMBER() OVER (PARTITION BY o.customer_id ORDER BY COUNT(*) DESC) AS
preference_rank
    FROM orders o
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN products p ON oi.product_id = p.product_id
    JOIN categories c ON p.category_id = c.category_id
```

```sql
    WHERE o.status = 'completed'
    GROUP BY o.customer_id, c.category_id, c.category_name
),
-- Customer communication preferences
customer_communications AS (
    SELECT
        customer_id,
        COUNT(*) AS total_communications,
        COUNT(CASE WHEN communication_type = 'email' THEN 1 END) AS email_count,
        COUNT(CASE WHEN communication_type = 'phone' THEN 1 END) AS phone_count,
        COUNT(CASE WHEN communication_type = 'chat' THEN 1 END) AS chat_count,
        MAX(communication_date) AS last_communication_date
    FROM customer_communications
    WHERE communication_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY customer_id
),
-- Customer support tickets
customer_support AS (
    SELECT
        customer_id,
        COUNT(*) AS total_tickets,
        COUNT(CASE WHEN status = 'resolved' THEN 1 END) AS resolved_tickets,
        COUNT(CASE WHEN priority = 'high' THEN 1 END) AS high_priority_tickets,
        AVG(EXTRACT(DAYS FROM (resolved_date - created_date))) AS
avg_resolution_days
    FROM support_tickets
    WHERE created_date >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY customer_id
)
SELECT
    c.customer_id,
    c.first_name || ' ' || c.last_name AS full_name,
    c.email,
    c.phone,
    c.status,
    c.created_at AS registration_date,
    EXTRACT(DAYS FROM (CURRENT_DATE - c.created_at)) AS days_since_registration,

    -- Order metrics
    COALESCE(co.total_orders, 0) AS total_orders,
    COALESCE(co.lifetime_value, 0) AS lifetime_value,
    COALESCE(co.avg_order_value, 0) AS avg_order_value,
    co.first_order_date,
    co.last_order_date,
    EXTRACT(DAYS FROM (CURRENT_DATE - co.last_order_date)) AS
days_since_last_order,
    COALESCE(co.value_12m, 0) AS value_12m,

    -- Customer segmentation
    CASE
        WHEN co.total_orders IS NULL THEN 'Never Purchased'
        WHEN co.last_order_date < CURRENT_DATE - INTERVAL '12 months' THEN
'Inactive'
        WHEN co.total_orders = 1 THEN 'One-time Buyer'
```

```
        WHEN co.lifetime_value >= 1000 THEN 'VIP'
        WHEN co.lifetime_value >= 500 THEN 'High Value'
        WHEN co.total_orders >= 5 THEN 'Loyal'
        ELSE 'Regular'
    END AS customer_segment,

    -- Preferences
    cc.category_name AS preferred_category,
    cc.category_spend AS preferred_category_spend,

    -- Communication metrics
    COALESCE(comm.total_communications, 0) AS total_communications,
    COALESCE(comm.email_count, 0) AS email_communications,
    COALESCE(comm.phone_count, 0) AS phone_communications,
    COALESCE(comm.chat_count, 0) AS chat_communications,
    comm.last_communication_date,

    -- Support metrics
    COALESCE(cs.total_tickets, 0) AS total_support_tickets,
    COALESCE(cs.resolved_tickets, 0) AS resolved_tickets,
    COALESCE(cs.high_priority_tickets, 0) AS high_priority_tickets,
    ROUND(COALESCE(cs.avg_resolution_days, 0), 1) AS avg_resolution_days,

    -- Risk indicators
    CASE
        WHEN co.cancelled_orders > co.completed_orders THEN 'High Risk'
        WHEN cs.high_priority_tickets > 2 THEN 'Support Risk'
        WHEN co.last_order_date < CURRENT_DATE - INTERVAL '6 months' THEN 'Churn
Risk'
        ELSE 'Low Risk'
    END AS risk_level

FROM customers c
LEFT JOIN customer_orders co ON c.customer_id = co.customer_id
LEFT JOIN customer_categories cc ON c.customer_id = cc.customer_id AND
cc.preference_rank = 1
LEFT JOIN customer_communications comm ON c.customer_id = comm.customer_id
LEFT JOIN customer_support cs ON c.customer_id = cs.customer_id;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"What's the difference between views and CTEs?"**
    - Views: Persistent, reusable, can be indexed (materialized)
    - CTEs: Query-scoped, temporary, better for complex query organization
    - Use views for consistent data access, CTEs for query readability
2. **"When would you use a materialized view?"**

- Expensive aggregations that don't need real-time data
- Complex joins across large tables
- Reporting and analytics workloads
- When query performance is more important than data freshness

3. **"How do you handle view dependencies?"**
   - Document dependencies clearly
   - Use dependency tracking tools
   - Consider impact when modifying base tables
   - Plan migration strategies for schema changes

4. **"What are the limitations of updatable views?"**
   - Must be based on single table (usually)
   - Cannot contain aggregations, DISTINCT, GROUP BY
   - Cannot have calculated columns in UPDATE
   - Complex views may require INSTEAD OF triggers

Best Practices:

1. **Use descriptive names for views and CTEs**
2. **Document complex view logic thoroughly**
3. **Consider performance implications of nested views**
4. **Use materialized views for expensive operations**
5. **Implement proper security through views**
6. **Test view updates and dependencies**

⚠ Things to Watch Out For

1. **View Performance Issues**

```
-- Problem: Nested views can cause performance issues
CREATE VIEW slow_nested_view AS
SELECT * FROM expensive_view
WHERE some_condition = 'value';

-- Solution: Flatten the query or use materialized views
CREATE MATERIALIZED VIEW optimized_view AS
SELECT
    -- Direct query instead of nesting views
    column1, column2, calculated_field
FROM base_table
WHERE conditions
  AND some_condition = 'value';
```

2. **CTE Recursion Limits**

```sql
-- Problem: Infinite recursion
WITH RECURSIVE infinite_loop AS (
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM infinite_loop  -- No termination condition!
)
SELECT * FROM infinite_loop;

-- Solution: Add proper termination conditions
WITH RECURSIVE safe_recursion AS (
    SELECT 1 AS n, 0 AS level
    UNION ALL
    SELECT n + 1, level + 1
    FROM safe_recursion
    WHERE level < 100  -- Termination condition
)
SELECT * FROM safe_recursion;
```

3. **Materialized View Staleness**

```sql
-- Problem: Forgetting to refresh materialized views
-- Data becomes stale and reports are inaccurate

-- Solution: Set up automatic refresh schedules
SELECT cron.schedule('refresh-daily-analytics', '0 2 * * *',
    'REFRESH MATERIALIZED VIEW CONCURRENTLY daily_analytics;');

-- Or use triggers for critical data
CREATE OR REPLACE FUNCTION refresh_customer_stats()
RETURNS TRIGGER AS $$
BEGIN
    REFRESH MATERIALIZED VIEW CONCURRENTLY customer_stats_mv;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER refresh_stats_on_order
    AFTER INSERT OR UPDATE OR DELETE ON orders
    FOR EACH STATEMENT
    EXECUTE FUNCTION refresh_customer_stats();
```

4. **Security Through Obscurity**

```sql
-- Problem: Relying only on views for security
CREATE VIEW public_customers AS
SELECT customer_id, first_name, last_name
FROM customers;
-- Users can still access the base table!
```

```sql
-- Solution: Combine with proper permissions
REVOKE ALL ON customers FROM public;
GRANT SELECT ON public_customers TO application_role;

-- Or use Row Level Security (PostgreSQL)
ALTER TABLE customers ENABLE ROW LEVEL SECURITY;
CREATE POLICY customer_access_policy ON customers
    FOR ALL TO application_role
    USING (customer_id = current_setting('app.current_customer_id')::INT);
```

🚀 Next Steps

In the next chapter, we'll explore **Transactions and Concurrency Control** - essential concepts for maintaining data integrity in multi-user environments. You'll learn about ACID properties, isolation levels, locking mechanisms, and how to handle concurrent access to your database.

📝 Quick Practice

Try these view and CTE exercises:

1. **Business Intelligence Views**: Create a set of views for executive reporting
2. **Security Views**: Implement role-based data access through views
3. **Recursive CTEs**: Build hierarchical data navigation (org charts, categories)
4. **Materialized Views**: Create and maintain analytics tables
5. **Complex CTEs**: Break down complex analytical queries into readable parts

Consider:

- When to use views vs CTEs vs materialized views
- Performance implications of your design choices
- Security and access control requirements
- Maintenance and refresh strategies for materialized views
- Documentation and dependency management

Chapter 15: Transactions and Concurrency Control

🗄 What You'll Learn

Transactions are fundamental to database integrity, ensuring that operations either complete successfully or fail completely. Concurrency control manages how multiple users access the same data simultaneously. This chapter covers ACID properties, isolation levels, locking mechanisms, and strategies for handling concurrent database access.

🎯 Learning Objectives

By the end of this chapter, you will:

- Understand ACID properties and their importance
- Master transaction control statements (BEGIN, COMMIT, ROLLBACK)
- Implement proper error handling in transactions
- Configure and use different isolation levels
- Handle deadlocks and concurrency issues
- Optimize transaction performance
- Implement distributed transactions

🔍 Concept Explanation

What are Transactions?

A transaction is a sequence of database operations that are treated as a single unit of work. Transactions ensure data integrity by following the ACID properties:

ACID Properties:

- **Atomicity**: All operations succeed or all fail (all-or-nothing)
- **Consistency**: Database remains in a valid state before and after
- **Isolation**: Concurrent transactions don't interfere with each other
- **Durability**: Committed changes persist even after system failure

Concurrency Control

Concurrency control manages simultaneous access to data by multiple users/processes:

Common Issues:

- **Dirty Reads**: Reading uncommitted changes
- **Non-repeatable Reads**: Same query returns different results
- **Phantom Reads**: New rows appear between reads
- **Lost Updates**: Concurrent updates overwrite each other

Solutions:

- **Locking**: Prevent concurrent access to data
- **Isolation Levels**: Control what changes are visible
- **Optimistic Concurrency**: Detect conflicts at commit time
- **Pessimistic Concurrency**: Prevent conflicts with locks

🛠️ Syntax Comparison

MySQL vs PostgreSQL Transactions

Feature	MySQL	PostgreSQL
Auto-commit	ON by default	OFF by default

Feature	MySQL	PostgreSQL
Transaction Start	START TRANSACTION, BEGIN	BEGIN, START TRANSACTION
Savepoints	✅	✅
Nested Transactions	❌	✅ (via savepoints)
Isolation Levels	4 standard levels	4 standard + custom
Lock Types	Table, row, metadata	Table, row, advisory
Deadlock Detection	✅ Automatic	✅ Automatic
MVCC	✅ (InnoDB)	✅ Native
Read Committed	Default	Default
Serializable	✅	✅

🔄 Basic Transaction Control

MySQL Transactions

```sql
-- Basic transaction structure
START TRANSACTION;

-- Your SQL operations here
INSERT INTO customers (first_name, last_name, email)
VALUES ('John', 'Doe', 'john.doe@email.com');

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

-- Commit if everything is successful
COMMIT;

-- Example with rollback on error
START TRANSACTION;

DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
    RESIGNAL;
END;

-- Transfer money between accounts
UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;

COMMIT;

-- Using savepoints
```

```sql
START TRANSACTION;

INSERT INTO orders (customer_id, total_amount) VALUES (1, 100.00);
SAVEPOINT order_created;

INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (LAST_INSERT_ID(), 1, 2, 25.00);
SAVEPOINT items_added;

UPDATE products SET stock_quantity = stock_quantity - 2 WHERE product_id = 1;

-- If stock update fails, rollback to savepoint
IF ROW_COUNT() = 0 THEN
    ROLLBACK TO SAVEPOINT items_added;
    -- Handle insufficient stock
END IF;

COMMIT;

-- Auto-commit control
SET autocommit = 0;  -- Disable auto-commit
SELECT @@autocommit; -- Check current setting

-- Manual transaction without START TRANSACTION
INSERT INTO customers (first_name, last_name, email)
VALUES ('Jane', 'Smith', 'jane.smith@email.com');
COMMIT;

SET autocommit = 1;  -- Re-enable auto-commit
```

PostgreSQL Transactions

```sql
-- Basic transaction structure
BEGIN;

-- Your SQL operations here
INSERT INTO customers (first_name, last_name, email)
VALUES ('John', 'Doe', 'john.doe@email.com');

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

-- Commit if everything is successful
COMMIT;

-- Example with exception handling
DO $$
BEGIN
    BEGIN
        -- Start transaction block
        INSERT INTO orders (customer_id, total_amount) VALUES (1, 100.00);
```

```
        -- This might fail
        UPDATE products SET stock_quantity = stock_quantity - 10
        WHERE product_id = 1 AND stock_quantity >= 10;

        IF NOT FOUND THEN
            RAISE EXCEPTION 'Insufficient stock for product %', 1;
        END IF;

        -- If we get here, commit
        COMMIT;

    EXCEPTION
        WHEN OTHERS THEN
            ROLLBACK;
            RAISE NOTICE 'Transaction failed: %', SQLERRM;
    END;
END $$;

-- Using savepoints (nested transactions)
BEGIN;

INSERT INTO orders (customer_id, total_amount) VALUES (1, 100.00);
SAVEPOINT order_created;

INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (currval('orders_order_id_seq'), 1, 2, 25.00);
SAVEPOINT items_added;

-- Try to update stock
UPDATE products SET stock_quantity = stock_quantity - 2
WHERE product_id = 1 AND stock_quantity >= 2;

IF NOT FOUND THEN
    -- Rollback to savepoint and handle error
    ROLLBACK TO SAVEPOINT items_added;
    INSERT INTO order_items (order_id, product_id, quantity, unit_price, status)
    VALUES (currval('orders_order_id_seq'), 1, 2, 25.00, 'backordered');
END IF;

COMMIT;

-- Transaction isolation levels
BEGIN ISOLATION LEVEL READ COMMITTED;
-- or
BEGIN ISOLATION LEVEL REPEATABLE READ;
-- or
BEGIN ISOLATION LEVEL SERIALIZABLE;

-- Set isolation level for session
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Check current isolation level
SHOW TRANSACTION ISOLATION LEVEL;
```

Isolation Levels

Understanding Isolation Levels

```
-- READ UNCOMMITTED (lowest isolation)
-- Can see uncommitted changes from other transactions
-- Allows: Dirty reads, non-repeatable reads, phantom reads

-- Session 1
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;
SELECT balance FROM accounts WHERE account_id = 1; -- Shows 1000

-- Session 2 (concurrent)
BEGIN;
UPDATE accounts SET balance = 500 WHERE account_id = 1;
-- Don't commit yet

-- Back to Session 1
SELECT balance FROM accounts WHERE account_id = 1; -- Shows 500 (dirty read!)
COMMIT;

-- READ COMMITTED (default in most databases)
-- Only sees committed changes
-- Prevents: Dirty reads
-- Allows: Non-repeatable reads, phantom reads

-- Session 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
SELECT balance FROM accounts WHERE account_id = 1; -- Shows 1000

-- Session 2
BEGIN;
UPDATE accounts SET balance = 500 WHERE account_id = 1;
COMMIT; -- Now committed

-- Back to Session 1
SELECT balance FROM accounts WHERE account_id = 1; -- Shows 500 (non-repeatable
read)
COMMIT;

-- REPEATABLE READ
-- Same reads return same results within transaction
-- Prevents: Dirty reads, non-repeatable reads
-- Allows: Phantom reads

-- Session 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN;
SELECT COUNT(*) FROM customers WHERE city = 'New York'; -- Shows 10

-- Session 2
BEGIN;
INSERT INTO customers (first_name, last_name, city)
VALUES ('New', 'Customer', 'New York');
COMMIT;

-- Back to Session 1
SELECT COUNT(*) FROM customers WHERE city = 'New York'; -- Still shows 10
SELECT * FROM customers WHERE city = 'New York'; -- Might show 11 rows (phantom
read)
COMMIT;

-- SERIALIZABLE (highest isolation)
-- Transactions appear to run sequentially
-- Prevents: All concurrency issues
-- May cause: More deadlocks and performance issues

-- Session 1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
SELECT SUM(balance) FROM accounts; -- Shows total

-- Session 2
BEGIN ISOLATION LEVEL SERIALIZABLE;
INSERT INTO accounts (customer_id, balance) VALUES (999, 1000);
-- This might block or cause serialization failure
COMMIT;

-- Back to Session 1
SELECT SUM(balance) FROM accounts; -- Same result as before
COMMIT;
```

Practical Isolation Level Examples

E-commerce Order Processing:

```
-- PostgreSQL: Handling concurrent order processing
CREATE OR REPLACE FUNCTION process_order(
    p_customer_id INT,
    p_product_id INT,
    p_quantity INT
) RETURNS TABLE(
    order_id INT,
    status TEXT,
    message TEXT
) AS $$
DECLARE
    v_order_id INT;
```

```
    v_available_stock INT;
    v_unit_price DECIMAL(10,2);
BEGIN
    -- Use REPEATABLE READ to ensure consistent stock checks
    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

    BEGIN
        -- Check product availability and get price
        SELECT stock_quantity, price INTO v_available_stock, v_unit_price
        FROM products
        WHERE product_id = p_product_id AND status = 'active'
        FOR UPDATE; -- Lock the row

        IF NOT FOUND THEN
            RETURN QUERY SELECT NULL::INT, 'ERROR'::TEXT, 'Product not
found'::TEXT;
            RETURN;
        END IF;

        IF v_available_stock < p_quantity THEN
            RETURN QUERY SELECT NULL::INT, 'ERROR'::TEXT,
                format('Insufficient stock. Available: %s, Requested: %s',
                       v_available_stock, p_quantity)::TEXT;
            RETURN;
        END IF;

        -- Create order
        INSERT INTO orders (customer_id, total_amount, status, order_date)
        VALUES (p_customer_id, v_unit_price * p_quantity, 'pending',
CURRENT_TIMESTAMP)
        RETURNING orders.order_id INTO v_order_id;

        -- Add order item
        INSERT INTO order_items (order_id, product_id, quantity, unit_price)
        VALUES (v_order_id, p_product_id, p_quantity, v_unit_price);

        -- Update stock
        UPDATE products
        SET stock_quantity = stock_quantity - p_quantity
        WHERE product_id = p_product_id;

        -- Update order status
        UPDATE orders SET status = 'confirmed' WHERE orders.order_id = v_order_id;

        RETURN QUERY SELECT v_order_id, 'SUCCESS'::TEXT, 'Order processed
successfully'::TEXT;

    EXCEPTION
        WHEN serialization_failure THEN
            RETURN QUERY SELECT NULL::INT, 'RETRY'::TEXT, 'Serialization conflict,
please retry'::TEXT;
        WHEN OTHERS THEN
            RETURN QUERY SELECT NULL::INT, 'ERROR'::TEXT, SQLERRM::TEXT;
    END;
```

```
END;
$$ LANGUAGE plpgsql;

-- Usage with retry logic
DO $$
DECLARE
    result RECORD;
    retry_count INT := 0;
    max_retries INT := 3;
BEGIN
    LOOP
        SELECT * INTO result FROM process_order(1, 101, 2);

        IF result.status = 'SUCCESS' THEN
            RAISE NOTICE 'Order processed: %', result.message;
            EXIT;
        ELSIF result.status = 'RETRY' AND retry_count < max_retries THEN
            retry_count := retry_count + 1;
            RAISE NOTICE 'Retrying... Attempt %', retry_count;
            PERFORM pg_sleep(0.1); -- Brief delay
        ELSE
            RAISE NOTICE 'Order failed: %', result.message;
            EXIT;
        END IF;
    END LOOP;
END $$;
```

Banking Transfer with Deadlock Prevention:

```
-- MySQL: Safe money transfer with deadlock prevention
DELIMITER //
CREATE PROCEDURE transfer_money(
    IN from_account INT,
    IN to_account INT,
    IN amount DECIMAL(10,2),
    OUT result_status VARCHAR(50),
    OUT result_message TEXT
)
BEGIN
    DECLARE from_balance DECIMAL(10,2);
    DECLARE account1 INT;
    DECLARE account2 INT;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        GET DIAGNOSTICS CONDITION 1
            result_message = MESSAGE_TEXT;
        SET result_status = 'ERROR';
    END;

    -- Prevent deadlocks by always locking accounts in same order
    IF from_account < to_account THEN
```

```
        SET account1 = from_account;
        SET account2 = to_account;
    ELSE
        SET account1 = to_account;
        SET account2 = from_account;
    END IF;

    START TRANSACTION;

    -- Lock accounts in consistent order
    SELECT balance INTO from_balance
    FROM accounts
    WHERE account_id = account1
    FOR UPDATE;

    SELECT balance INTO @temp
    FROM accounts
    WHERE account_id = account2
    FOR UPDATE;

    -- Check sufficient funds
    SELECT balance INTO from_balance
    FROM accounts
    WHERE account_id = from_account;

    IF from_balance < amount THEN
        SET result_status = 'INSUFFICIENT_FUNDS';
        SET result_message = CONCAT('Insufficient funds. Available: ',
from_balance, ', Requested: ', amount);
        ROLLBACK;
    ELSE
        -- Perform transfer
        UPDATE accounts SET balance = balance - amount WHERE account_id =
from_account;
        UPDATE accounts SET balance = balance + amount WHERE account_id =
to_account;

        -- Log transaction
        INSERT INTO transaction_log (from_account, to_account, amount,
transaction_date)
        VALUES (from_account, to_account, amount, NOW());

        SET result_status = 'SUCCESS';
        SET result_message = CONCAT('Transfer completed: $', amount, ' from
account ', from_account, ' to account ', to_account);

        COMMIT;
    END IF;
END //
DELIMITER ;

-- Usage
CALL transfer_money(1, 2, 500.00, @status, @message);
SELECT @status, @message;
```

Locking Mechanisms

Explicit Locking

MySQL Locking:

```sql
-- Table-level locking
LOCK TABLES customers READ, orders WRITE;
-- Perform operations
SELECT * FROM customers WHERE customer_id = 1;
INSERT INTO orders (customer_id, total_amount) VALUES (1, 100.00);
UNLOCK TABLES;

-- Row-level locking with SELECT FOR UPDATE
START TRANSACTION;

-- Lock specific rows for update
SELECT * FROM products
WHERE category_id = 1 AND stock_quantity > 0
FOR UPDATE;

-- Update the locked rows
UPDATE products
SET stock_quantity = stock_quantity - 1
WHERE category_id = 1 AND stock_quantity > 0;

COMMIT;

-- Shared locks (SELECT FOR SHARE)
START TRANSACTION;

-- Multiple transactions can hold shared locks
SELECT * FROM customers WHERE customer_id = 1 FOR SHARE;

-- This allows other SELECT FOR SHARE but blocks SELECT FOR UPDATE
COMMIT;

-- Lock with timeout (MySQL 8.0+)
SELECT * FROM products WHERE product_id = 1
FOR UPDATE NOWAIT; -- Fail immediately if can't lock

SELECT * FROM products WHERE product_id = 1
FOR UPDATE SKIP LOCKED; -- Skip locked rows
```

PostgreSQL Locking:

```
-- Table-level locking
BEGIN;
LOCK TABLE customers IN ACCESS SHARE MODE; -- Least restrictive
LOCK TABLE orders IN ROW EXCLUSIVE MODE;   -- For INSERT/UPDATE/DELETE
LOCK TABLE products IN EXCLUSIVE MODE;     -- Most restrictive
COMMIT;

-- Row-level locking
BEGIN;

-- FOR UPDATE (exclusive row lock)
SELECT * FROM accounts WHERE account_id = 1 FOR UPDATE;

-- FOR NO KEY UPDATE (allows foreign key references)
SELECT * FROM customers WHERE customer_id = 1 FOR NO KEY UPDATE;

-- FOR SHARE (shared row lock)
SELECT * FROM products WHERE product_id = 1 FOR SHARE;

-- FOR KEY SHARE (weakest row lock)
SELECT * FROM categories WHERE category_id = 1 FOR KEY SHARE;

COMMIT;

-- Advisory locks (application-level)
SELECT pg_advisory_lock(12345); -- Block until lock acquired
SELECT pg_try_advisory_lock(12345); -- Return immediately (true/false)

-- Perform application logic
-- ...

SELECT pg_advisory_unlock(12345);

-- Session-level advisory locks
SELECT pg_advisory_lock_shared(12345); -- Shared advisory lock
SELECT pg_advisory_unlock_shared(12345);

-- Lock with timeout
BEGIN;
SET lock_timeout = '5s';
SELECT * FROM products WHERE product_id = 1 FOR UPDATE;
COMMIT;
```

Deadlock Handling

Deadlock Detection and Resolution:

```
-- PostgreSQL: Deadlock detection example
CREATE OR REPLACE FUNCTION safe_transfer(
    from_acc INT,
    to_acc INT,
```

```
    amount DECIMAL
) RETURNS TEXT AS $$
DECLARE
    retry_count INT := 0;
    max_retries INT := 3;
BEGIN
    LOOP
        BEGIN
            -- Always lock accounts in same order to prevent deadlocks
            IF from_acc < to_acc THEN
                PERFORM * FROM accounts WHERE account_id = from_acc FOR UPDATE;
                PERFORM * FROM accounts WHERE account_id = to_acc FOR UPDATE;
            ELSE
                PERFORM * FROM accounts WHERE account_id = to_acc FOR UPDATE;
                PERFORM * FROM accounts WHERE account_id = from_acc FOR UPDATE;
            END IF;

            -- Perform transfer
            UPDATE accounts SET balance = balance - amount WHERE account_id =
from_acc;
            UPDATE accounts SET balance = balance + amount WHERE account_id =
to_acc;

            RETURN 'Transfer completed successfully';

        EXCEPTION
            WHEN deadlock_detected THEN
                retry_count := retry_count + 1;
                IF retry_count >= max_retries THEN
                    RAISE EXCEPTION 'Transfer failed after % retries due to
deadlocks', max_retries;
                END IF;

                -- Random delay to reduce deadlock probability
                PERFORM pg_sleep(random() * 0.1);

        END;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

-- MySQL: Deadlock handling
DELIMITER //
CREATE PROCEDURE safe_update_inventory(
    IN product_id_1 INT,
    IN quantity_1 INT,
    IN product_id_2 INT,
    IN quantity_2 INT
)
BEGIN
    DECLARE deadlock_count INT DEFAULT 0;
    DECLARE max_retries INT DEFAULT 3;
    DECLARE done INT DEFAULT FALSE;
```

```
    retry_loop: LOOP
        BEGIN
            DECLARE EXIT HANDLER FOR 1213 -- Deadlock error code
            BEGIN
                SET deadlock_count = deadlock_count + 1;
                IF deadlock_count >= max_retries THEN
                    RESIGNAL;
                END IF;
                ROLLBACK;
                -- Random delay
                DO SLEEP(RAND() * 0.1);
            END;

            START TRANSACTION;

            -- Lock products in consistent order
            IF product_id_1 < product_id_2 THEN
                SELECT stock_quantity FROM products WHERE product_id =
product_id_1 FOR UPDATE;
                SELECT stock_quantity FROM products WHERE product_id =
product_id_2 FOR UPDATE;
            ELSE
                SELECT stock_quantity FROM products WHERE product_id =
product_id_2 FOR UPDATE;
                SELECT stock_quantity FROM products WHERE product_id =
product_id_1 FOR UPDATE;
            END IF;

            -- Update inventory
            UPDATE products SET stock_quantity = stock_quantity - quantity_1 WHERE
product_id = product_id_1;
            UPDATE products SET stock_quantity = stock_quantity - quantity_2 WHERE
product_id = product_id_2;

            COMMIT;
            SET done = TRUE;

        END;

        IF done THEN
            LEAVE retry_loop;
        END IF;
    END LOOP;
END //
DELIMITER ;
```

⚡ Performance Optimization

Transaction Best Practices

```sql
-- Keep transactions short
-- BAD: Long-running transaction
BEGIN;
SELECT * FROM large_table; -- Long-running query
UPDATE products SET price = price * 1.1; -- Locks many rows
-- User input or external API call
COMMIT;

-- GOOD: Short, focused transaction
BEGIN;
UPDATE products SET price = price * 1.1 WHERE category_id = 1;
COMMIT;

-- Batch processing for large operations
-- PostgreSQL: Process large updates in batches
DO $$
DECLARE
    batch_size INT := 1000;
    processed INT := 0;
    total_rows INT;
BEGIN
    SELECT COUNT(*) INTO total_rows FROM old_orders WHERE status = 'pending';

    WHILE processed < total_rows LOOP
        BEGIN
            UPDATE old_orders
            SET status = 'expired'
            WHERE order_id IN (
                SELECT order_id
                FROM old_orders
                WHERE status = 'pending'
                  AND order_date < CURRENT_DATE - INTERVAL '30 days'
                LIMIT batch_size
            );

            processed := processed + batch_size;

            -- Commit each batch
            COMMIT;

            -- Brief pause to allow other transactions
            PERFORM pg_sleep(0.01);

        EXCEPTION
            WHEN OTHERS THEN
                ROLLBACK;
                RAISE NOTICE 'Batch failed at row %: %', processed, SQLERRM;
                EXIT;
        END;
    END LOOP;

    RAISE NOTICE 'Processed % rows in batches of %', processed, batch_size;
END $$;
```

```
-- MySQL: Batch processing with cursor
DELIMITER //
CREATE PROCEDURE batch_update_prices()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE batch_count INT DEFAULT 0;
    DECLARE product_cursor CURSOR FOR
        SELECT product_id FROM products WHERE last_updated < DATE_SUB(NOW(),
INTERVAL 1 YEAR);
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN product_cursor;

    read_loop: LOOP
        START TRANSACTION;

        SET batch_count = 0;

        batch_loop: LOOP
            FETCH product_cursor INTO @product_id;

            IF done OR batch_count >= 100 THEN
                LEAVE batch_loop;
            END IF;

            UPDATE products
            SET price = price * 1.05, last_updated = NOW()
            WHERE product_id = @product_id;

            SET batch_count = batch_count + 1;
        END LOOP;

        COMMIT;

        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Brief pause
        DO SLEEP(0.01);
    END LOOP;

    CLOSE product_cursor;
END //
DELIMITER ;
```

Connection Pooling and Transaction Management

```
-- PostgreSQL: Connection pooling considerations
-- Set appropriate timeout values
```

```sql
SET statement_timeout = '30s';
SET lock_timeout = '10s';
SET idle_in_transaction_session_timeout = '60s';

-- Monitor long-running transactions
SELECT
    pid,
    now() - pg_stat_activity.query_start AS duration,
    query,
    state,
    wait_event_type,
    wait_event
FROM pg_stat_activity
WHERE state IN ('active', 'idle in transaction')
  AND now() - pg_stat_activity.query_start > interval '5 minutes'
ORDER BY duration DESC;

-- Kill long-running transactions if needed
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE state = 'idle in transaction'
  AND now() - query_start > interval '1 hour';

-- MySQL: Transaction monitoring
-- Check for long-running transactions
SELECT
    trx_id,
    trx_state,
    trx_started,
    TIMESTAMPDIFF(SECOND, trx_started, NOW()) AS duration_seconds,
    trx_mysql_thread_id,
    trx_query
FROM information_schema.INNODB_TRX
WHERE TIMESTAMPDIFF(SECOND, trx_started, NOW()) > 300
ORDER BY duration_seconds DESC;

-- Check for locks
SELECT
    r.trx_id AS waiting_trx_id,
    r.trx_mysql_thread_id AS waiting_thread,
    r.trx_query AS waiting_query,
    b.trx_id AS blocking_trx_id,
    b.trx_mysql_thread_id AS blocking_thread,
    b.trx_query AS blocking_query
FROM information_schema.INNODB_LOCK_WAITS w
INNER JOIN information_schema.INNODB_TRX b ON b.trx_id = w.blocking_trx_id
INNER JOIN information_schema.INNODB_TRX r ON r.trx_id = w.requesting_trx_id;

-- Kill blocking transaction if needed
KILL 12345; -- Replace with actual thread ID
```

🌐 Distributed Transactions

Two-Phase Commit (2PC)

```
-- PostgreSQL: Prepared transactions (2PC)
-- Phase 1: Prepare
BEGIN;
INSERT INTO orders (customer_id, total_amount) VALUES (1, 100.00);
UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1;
PREPARE TRANSACTION 'order_txn_12345';

-- Phase 2: Commit or Rollback
-- On success:
COMMIT PREPARED 'order_txn_12345';

-- On failure:
ROLLBACK PREPARED 'order_txn_12345';

-- Check prepared transactions
SELECT * FROM pg_prepared_xacts;

-- Cleanup orphaned prepared transactions
SELECT
    gid,
    prepared,
    owner,
    database
FROM pg_prepared_xacts
WHERE prepared < NOW() - INTERVAL '1 hour';

-- Example distributed transaction coordinator
CREATE OR REPLACE FUNCTION distributed_order_process(
    p_customer_id INT,
    p_product_id INT,
    p_quantity INT
) RETURNS BOOLEAN AS $$
DECLARE
    txn_id TEXT;
    inventory_success BOOLEAN := FALSE;
    payment_success BOOLEAN := FALSE;
BEGIN
    txn_id := 'order_' || p_customer_id || '_' || extract(epoch from now());

    -- Phase 1: Prepare all participants
    BEGIN
        -- Prepare inventory transaction
        BEGIN;
        UPDATE inventory SET quantity = quantity - p_quantity
        WHERE product_id = p_product_id AND quantity >= p_quantity;

        IF NOT FOUND THEN
            ROLLBACK;
```

```
            RETURN FALSE;
        END IF;

        PREPARE TRANSACTION txn_id || '_inventory';
        inventory_success := TRUE;

        -- Prepare payment transaction
        BEGIN;
        INSERT INTO orders (customer_id, product_id, quantity, status)
        VALUES (p_customer_id, p_product_id, p_quantity, 'pending');

        -- Simulate payment processing
        IF random() > 0.1 THEN -- 90% success rate
            PREPARE TRANSACTION txn_id || '_payment';
            payment_success := TRUE;
        ELSE
            ROLLBACK;
            payment_success := FALSE;
        END IF;

    EXCEPTION
        WHEN OTHERS THEN
            -- Cleanup on error
            IF inventory_success THEN
                ROLLBACK PREPARED txn_id || '_inventory';
            END IF;
            RETURN FALSE;
    END;

    -- Phase 2: Commit or rollback all
    IF inventory_success AND payment_success THEN
        COMMIT PREPARED txn_id || '_inventory';
        COMMIT PREPARED txn_id || '_payment';
        RETURN TRUE;
    ELSE
        IF inventory_success THEN
            ROLLBACK PREPARED txn_id || '_inventory';
        END IF;
        IF payment_success THEN
            ROLLBACK PREPARED txn_id || '_payment';
        END IF;
        RETURN FALSE;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Saga Pattern (Alternative to 2PC)

```
-- PostgreSQL: Saga pattern implementation
CREATE TABLE saga_transactions (
    saga_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
    saga_type VARCHAR(50) NOT NULL,
    status VARCHAR(20) DEFAULT 'started',
    current_step INT DEFAULT 1,
    total_steps INT NOT NULL,
    data JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE saga_steps (
    step_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    saga_id UUID REFERENCES saga_transactions(saga_id),
    step_number INT NOT NULL,
    step_name VARCHAR(100) NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    forward_action TEXT,
    compensate_action TEXT,
    result JSONB,
    executed_at TIMESTAMP,
    compensated_at TIMESTAMP
);

-- Order processing saga
CREATE OR REPLACE FUNCTION start_order_saga(
    p_customer_id INT,
    p_product_id INT,
    p_quantity INT,
    p_amount DECIMAL
) RETURNS UUID AS $$
DECLARE
    saga_id UUID;
BEGIN
    -- Create saga transaction
    INSERT INTO saga_transactions (saga_type, total_steps, data)
    VALUES (
        'order_processing',
        4,
        jsonb_build_object(
            'customer_id', p_customer_id,
            'product_id', p_product_id,
            'quantity', p_quantity,
            'amount', p_amount
        )
    ) RETURNING saga_transactions.saga_id INTO saga_id;

    -- Define saga steps
    INSERT INTO saga_steps (saga_id, step_number, step_name, forward_action,
compensate_action)
    VALUES
    (saga_id, 1, 'reserve_inventory', 'reserve_product_inventory',
'release_product_inventory'),
    (saga_id, 2, 'process_payment', 'charge_customer_payment',
'refund_customer_payment'),
    (saga_id, 3, 'create_order', 'create_customer_order',
```

```
'cancel_customer_order'),
    (saga_id, 4, 'send_confirmation', 'send_order_confirmation',
'send_cancellation_notice');

    -- Start processing
    PERFORM process_saga_step(saga_id, 1);

    RETURN saga_id;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION process_saga_step(
    p_saga_id UUID,
    p_step_number INT
) RETURNS BOOLEAN AS $$
DECLARE
    saga_data JSONB;
    step_record RECORD;
    success BOOLEAN := FALSE;
BEGIN
    -- Get saga data
    SELECT data INTO saga_data FROM saga_transactions WHERE saga_id = p_saga_id;

    -- Get step details
    SELECT * INTO step_record FROM saga_steps
    WHERE saga_id = p_saga_id AND step_number = p_step_number;

    -- Execute step based on step name
    CASE step_record.step_name
        WHEN 'reserve_inventory' THEN
            success := execute_inventory_reservation(saga_data);
        WHEN 'process_payment' THEN
            success := execute_payment_processing(saga_data);
        WHEN 'create_order' THEN
            success := execute_order_creation(saga_data);
        WHEN 'send_confirmation' THEN
            success := execute_confirmation_sending(saga_data);
    END CASE;

    -- Update step status
    UPDATE saga_steps
    SET status = CASE WHEN success THEN 'completed' ELSE 'failed' END,
        executed_at = CURRENT_TIMESTAMP
    WHERE saga_id = p_saga_id AND step_number = p_step_number;

    IF success THEN
        -- Move to next step or complete saga
        IF p_step_number < (SELECT total_steps FROM saga_transactions WHERE
saga_id = p_saga_id) THEN
            UPDATE saga_transactions
            SET current_step = p_step_number + 1
            WHERE saga_id = p_saga_id;

            PERFORM process_saga_step(p_saga_id, p_step_number + 1);
```

```
        ELSE
            UPDATE saga_transactions
            SET status = 'completed'
            WHERE saga_id = p_saga_id;
        END IF;
    ELSE
        -- Start compensation
        UPDATE saga_transactions
        SET status = 'compensating'
        WHERE saga_id = p_saga_id;

        PERFORM compensate_saga(p_saga_id, p_step_number - 1);
    END IF;

    RETURN success;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION compensate_saga(
    p_saga_id UUID,
    p_from_step INT
) RETURNS VOID AS $$
DECLARE
    step_record RECORD;
    saga_data JSONB;
BEGIN
    SELECT data INTO saga_data FROM saga_transactions WHERE saga_id = p_saga_id;

    -- Compensate steps in reverse order
    FOR step_record IN
        SELECT * FROM saga_steps
        WHERE saga_id = p_saga_id
          AND step_number <= p_from_step
          AND status = 'completed'
        ORDER BY step_number DESC
    LOOP
        -- Execute compensation action
        CASE step_record.step_name
            WHEN 'reserve_inventory' THEN
                PERFORM compensate_inventory_reservation(saga_data);
            WHEN 'process_payment' THEN
                PERFORM compensate_payment_processing(saga_data);
            WHEN 'create_order' THEN
                PERFORM compensate_order_creation(saga_data);
        END CASE;

        UPDATE saga_steps
        SET status = 'compensated',
            compensated_at = CURRENT_TIMESTAMP
        WHERE step_id = step_record.step_id;
    END LOOP;

    UPDATE saga_transactions
    SET status = 'compensated'
```

```
    WHERE saga_id = p_saga_id;
END;
$$ LANGUAGE plpgsql;
```

Real-World Business Examples

Example 1: E-commerce Checkout Process

```
-- Complete checkout transaction with multiple validations
CREATE OR REPLACE FUNCTION process_checkout(
    p_customer_id INT,
    p_cart_items JSONB,
    p_payment_method VARCHAR(50),
    p_shipping_address JSONB
) RETURNS TABLE(
    success BOOLEAN,
    order_id INT,
    message TEXT,
    total_amount DECIMAL(10,2)
) AS $$
DECLARE
    v_order_id INT;
    v_total_amount DECIMAL(10,2) := 0;
    v_item JSONB;
    v_product RECORD;
    v_customer RECORD;
BEGIN
    -- Start transaction with appropriate isolation level
    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

    BEGIN
        -- Validate customer
        SELECT * INTO v_customer FROM customers
        WHERE customer_id = p_customer_id AND status = 'active';

        IF NOT FOUND THEN
            RETURN QUERY SELECT FALSE, NULL::INT, 'Invalid customer'::TEXT,
0::DECIMAL(10,2);
            RETURN;
        END IF;

        -- Create order
        INSERT INTO orders (customer_id, status, order_date, shipping_address)
        VALUES (p_customer_id, 'processing', CURRENT_TIMESTAMP,
p_shipping_address)
        RETURNING orders.order_id INTO v_order_id;

        -- Process each cart item
        FOR v_item IN SELECT * FROM jsonb_array_elements(p_cart_items)
        LOOP
```

```
            -- Lock and validate product
            SELECT * INTO v_product FROM products
            WHERE product_id = (v_item->>'product_id')::INT
              AND status = 'active'
            FOR UPDATE;

            IF NOT FOUND THEN
                RAISE EXCEPTION 'Product % not found', v_item->>'product_id';
            END IF;

            -- Check stock
            IF v_product.stock_quantity < (v_item->>'quantity')::INT THEN
                RAISE EXCEPTION 'Insufficient stock for product %. Available: %,
Requested: %',
                    v_product.product_name, v_product.stock_quantity, v_item-
>>'quantity';
            END IF;

            -- Add order item
            INSERT INTO order_items (
                order_id, product_id, quantity, unit_price, total_price
            ) VALUES (
                v_order_id,
                v_product.product_id,
                (v_item->>'quantity')::INT,
                v_product.price,
                v_product.price * (v_item->>'quantity')::INT
            );

            -- Update stock
            UPDATE products
            SET stock_quantity = stock_quantity - (v_item->>'quantity')::INT
            WHERE product_id = v_product.product_id;

            -- Add to total
            v_total_amount := v_total_amount + (v_product.price * (v_item-
>>'quantity')::INT);
        END LOOP;

        -- Update order total
        UPDATE orders SET total_amount = v_total_amount WHERE order_id =
v_order_id;

        -- Process payment (simplified)
        INSERT INTO payments (
            order_id, customer_id, amount, payment_method, status, processed_at
        ) VALUES (
            v_order_id, p_customer_id, v_total_amount, p_payment_method,
'completed', CURRENT_TIMESTAMP
        );

        -- Update order status
        UPDATE orders SET status = 'confirmed' WHERE order_id = v_order_id;
```

```
        -- Update customer stats
        INSERT INTO customer_stats (customer_id, total_orders, total_spent,
last_order_date)
        VALUES (p_customer_id, 1, v_total_amount, CURRENT_DATE)
        ON CONFLICT (customer_id) DO UPDATE SET
            total_orders = customer_stats.total_orders + 1,
            total_spent = customer_stats.total_spent + v_total_amount,
            last_order_date = CURRENT_DATE;

        RETURN QUERY SELECT TRUE, v_order_id, 'Order processed
successfully'::TEXT, v_total_amount;

    EXCEPTION
        WHEN OTHERS THEN
            RETURN QUERY SELECT FALSE, NULL::INT, SQLERRM::TEXT, 0::DECIMAL(10,2);
    END;
END;
$$ LANGUAGE plpgsql;

-- Usage with retry logic for serialization failures
DO $$
DECLARE
    result RECORD;
    retry_count INT := 0;
    max_retries INT := 3;
    cart_data JSONB := '[
        {"product_id": 1, "quantity": 2},
        {"product_id": 3, "quantity": 1}
    ]';
    shipping_addr JSONB := '{
        "street": "123 Main St",
        "city": "Anytown",
        "state": "CA",
        "zip": "12345"
    }';
BEGIN
    LOOP
        BEGIN
            SELECT * INTO result FROM process_checkout(
                1, -- customer_id
                cart_data,
                'credit_card',
                shipping_addr
            );

            IF result.success THEN
                RAISE NOTICE 'Checkout successful! Order ID: %, Total: $%',
                    result.order_id, result.total_amount;
                EXIT;
            ELSE
                RAISE NOTICE 'Checkout failed: %', result.message;
                EXIT;
            END IF;
```

```
        EXCEPTION
            WHEN serialization_failure THEN
                retry_count := retry_count + 1;
                IF retry_count >= max_retries THEN
                    RAISE NOTICE 'Checkout failed after % retries due to
serialization conflicts', max_retries;
                    EXIT;
                END IF;

                RAISE NOTICE 'Serialization conflict, retrying... (attempt %)',
retry_count;
                PERFORM pg_sleep(random() * 0.1);
        END;
    END LOOP;
END $$;
```

Example 2: Banking System with Audit Trail

```
-- Comprehensive banking transaction system
CREATE TABLE accounts (
    account_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    account_number VARCHAR(20) UNIQUE NOT NULL,
    account_type VARCHAR(20) NOT NULL,
    balance DECIMAL(15,2) NOT NULL DEFAULT 0,
    status VARCHAR(20) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE transactions (
    transaction_id SERIAL PRIMARY KEY,
    from_account_id INT,
    to_account_id INT,
    transaction_type VARCHAR(20) NOT NULL,
    amount DECIMAL(15,2) NOT NULL,
    description TEXT,
    reference_number VARCHAR(50) UNIQUE,
    status VARCHAR(20) DEFAULT 'pending',
    processed_at TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (from_account_id) REFERENCES accounts(account_id),
    FOREIGN KEY (to_account_id) REFERENCES accounts(account_id)
);

CREATE TABLE transaction_audit (
    audit_id SERIAL PRIMARY KEY,
    transaction_id INT,
    action VARCHAR(20),
    old_values JSONB,
    new_values JSONB,
```

```
    user_id INT,
    ip_address INET,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Secure money transfer function
CREATE OR REPLACE FUNCTION transfer_funds(
    p_from_account VARCHAR(20),
    p_to_account VARCHAR(20),
    p_amount DECIMAL(15,2),
    p_description TEXT DEFAULT NULL,
    p_user_id INT DEFAULT NULL
) RETURNS TABLE(
    success BOOLEAN,
    transaction_id INT,
    reference_number VARCHAR(50),
    message TEXT
) AS $$
DECLARE
    v_from_account_id INT;
    v_to_account_id INT;
    v_from_balance DECIMAL(15,2);
    v_to_balance DECIMAL(15,2);
    v_transaction_id INT;
    v_reference_number VARCHAR(50);
    v_from_account_rec RECORD;
    v_to_account_rec RECORD;
BEGIN
    -- Generate unique reference number
    v_reference_number := 'TXN' || to_char(CURRENT_TIMESTAMP, 'YYYYMMDDHH24MISS')
||
                          lpad(floor(random() * 1000)::text, 3, '0');

    -- Use serializable isolation for consistency
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

    BEGIN
        -- Lock and validate accounts in consistent order to prevent deadlocks
        IF p_from_account < p_to_account THEN
            SELECT * INTO v_from_account_rec FROM accounts
            WHERE account_number = p_from_account AND status = 'active' FOR
UPDATE;

            SELECT * INTO v_to_account_rec FROM accounts
            WHERE account_number = p_to_account AND status = 'active' FOR UPDATE;
        ELSE
            SELECT * INTO v_to_account_rec FROM accounts
            WHERE account_number = p_to_account AND status = 'active' FOR UPDATE;

            SELECT * INTO v_from_account_rec FROM accounts
            WHERE account_number = p_from_account AND status = 'active' FOR
UPDATE;
        END IF;
```

```
        -- Validate accounts exist
        IF v_from_account_rec.account_id IS NULL THEN
            RETURN QUERY SELECT FALSE, NULL::INT, NULL::VARCHAR(50),
                'Source account not found or inactive'::TEXT;
            RETURN;
        END IF;

        IF v_to_account_rec.account_id IS NULL THEN
            RETURN QUERY SELECT FALSE, NULL::INT, NULL::VARCHAR(50),
                'Destination account not found or inactive'::TEXT;
            RETURN;
        END IF;

        -- Validate amount
        IF p_amount <= 0 THEN
            RETURN QUERY SELECT FALSE, NULL::INT, NULL::VARCHAR(50),
                'Transfer amount must be positive'::TEXT;
            RETURN;
        END IF;

        -- Check sufficient funds
        IF v_from_account_rec.balance < p_amount THEN
            RETURN QUERY SELECT FALSE, NULL::INT, NULL::VARCHAR(50),
                format('Insufficient funds. Available: $%.2f, Requested: $%.2f',
                       v_from_account_rec.balance, p_amount)::TEXT;
            RETURN;
        END IF;

        -- Create transaction record
        INSERT INTO transactions (
            from_account_id, to_account_id, transaction_type, amount,
            description, reference_number, status
        ) VALUES (
            v_from_account_rec.account_id, v_to_account_rec.account_id,
'transfer',
            p_amount, p_description, v_reference_number, 'processing'
        ) RETURNING transactions.transaction_id INTO v_transaction_id;

        -- Audit: Transaction created
        INSERT INTO transaction_audit (
            transaction_id, action, new_values, user_id, ip_address
        ) VALUES (
            v_transaction_id, 'created',
            jsonb_build_object(
                'from_account', p_from_account,
                'to_account', p_to_account,
                'amount', p_amount,
                'reference', v_reference_number
            ),
            p_user_id, inet_client_addr()
        );

        -- Update account balances
        UPDATE accounts
```

```
        SET balance = balance - p_amount, updated_at = CURRENT_TIMESTAMP
        WHERE account_id = v_from_account_rec.account_id;

        UPDATE accounts
        SET balance = balance + p_amount, updated_at = CURRENT_TIMESTAMP
        WHERE account_id = v_to_account_rec.account_id;

        -- Update transaction status
        UPDATE transactions
        SET status = 'completed', processed_at = CURRENT_TIMESTAMP
        WHERE transaction_id = v_transaction_id;

        -- Audit: Transaction completed
        INSERT INTO transaction_audit (
            transaction_id, action, new_values, user_id, ip_address
        ) VALUES (
            v_transaction_id, 'completed',
            jsonb_build_object(
                'from_balance_after', v_from_account_rec.balance - p_amount,
                'to_balance_after', v_to_account_rec.balance + p_amount,
                'processed_at', CURRENT_TIMESTAMP
            ),
            p_user_id, inet_client_addr()
        );

        RETURN QUERY SELECT TRUE, v_transaction_id, v_reference_number,
            format('Transfer completed successfully. Reference: %s',
v_reference_number)::TEXT;

    EXCEPTION
        WHEN serialization_failure THEN
            -- Update transaction status to failed
            IF v_transaction_id IS NOT NULL THEN
                UPDATE transactions
                SET status = 'failed'
                WHERE transaction_id = v_transaction_id;

                INSERT INTO transaction_audit (
                    transaction_id, action, new_values, user_id, ip_address
                ) VALUES (
                    v_transaction_id, 'failed',
                    jsonb_build_object('error', 'serialization_failure'),
                    p_user_id, inet_client_addr()
                );
            END IF;

            RETURN QUERY SELECT FALSE, v_transaction_id, v_reference_number,
                'Transfer failed due to concurrent access. Please retry.'::TEXT;

        WHEN OTHERS THEN
            -- Update transaction status to failed
            IF v_transaction_id IS NOT NULL THEN
                UPDATE transactions
                SET status = 'failed'
```

```
                WHERE transaction_id = v_transaction_id;

                INSERT INTO transaction_audit (
                    transaction_id, action, new_values, user_id, ip_address
                ) VALUES (
                    v_transaction_id, 'failed',
                    jsonb_build_object('error', SQLERRM),
                    p_user_id, inet_client_addr()
                );
            END IF;

            RETURN QUERY SELECT FALSE, v_transaction_id, v_reference_number,
                format('Transfer failed: %s', SQLERRM)::TEXT;
    END;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

-- Usage example with retry logic
CREATE OR REPLACE FUNCTION safe_transfer_with_retry(
    p_from_account VARCHAR(20),
    p_to_account VARCHAR(20),
    p_amount DECIMAL(15,2),
    p_description TEXT DEFAULT NULL,
    p_user_id INT DEFAULT NULL
) RETURNS TABLE(
    success BOOLEAN,
    transaction_id INT,
    reference_number VARCHAR(50),
    message TEXT,
    attempts INT
) AS $$
DECLARE
    result RECORD;
    retry_count INT := 0;
    max_retries INT := 3;
BEGIN
    LOOP
        SELECT * INTO result FROM transfer_funds(
            p_from_account, p_to_account, p_amount, p_description, p_user_id
        );

        IF result.success OR retry_count >= max_retries THEN
            RETURN QUERY SELECT result.success, result.transaction_id,
                result.reference_number, result.message, retry_count + 1;
            RETURN;
        END IF;

        -- Only retry on serialization failures
        IF result.message LIKE '%concurrent access%' THEN
            retry_count := retry_count + 1;
            PERFORM pg_sleep(random() * 0.1); -- Random delay
        ELSE
            RETURN QUERY SELECT result.success, result.transaction_id,
                result.reference_number, result.message, retry_count + 1;
```

```
            RETURN;
        END IF;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"Explain the ACID properties with examples."**
    - **Atomicity**: Bank transfer - both debit and credit must succeed
    - **Consistency**: Account balances must follow business rules
    - **Isolation**: Concurrent transfers don't interfere
    - **Durability**: Committed transactions survive system crashes
2. **"What's the difference between optimistic and pessimistic locking?"**
    - **Pessimistic**: Lock data when reading (SELECT FOR UPDATE)
    - **Optimistic**: Check for conflicts at commit time (version numbers)
    - **Use pessimistic** for high contention, critical data
    - **Use optimistic** for low contention, better performance
3. **"How do you handle deadlocks?"**
    - **Prevention**: Lock resources in consistent order
    - **Detection**: Database automatically detects and resolves
    - **Recovery**: Retry failed transactions with exponential backoff
    - **Monitoring**: Track deadlock frequency and patterns
4. **"When would you use different isolation levels?"**
    - **READ UNCOMMITTED**: Rarely, only for approximate analytics
    - **READ COMMITTED**: Default for most applications
    - **REPEATABLE READ**: Financial calculations, reporting
    - **SERIALIZABLE**: Critical business logic, audit requirements

Best Practices:

1. **Keep transactions short and focused**
2. **Use appropriate isolation levels**
3. **Handle deadlocks gracefully with retries**
4. **Implement proper error handling and logging**
5. **Monitor transaction performance and blocking**
6. **Use connection pooling effectively**

⚠️ Things to Watch Out For

1. **Long-Running Transactions**

```
-- Problem: Transaction holds locks too long
BEGIN;
SELECT * FROM products FOR UPDATE; -- Locks all products
-- Long processing time or user input
UPDATE products SET price = price * 1.1;
COMMIT;

-- Solution: Minimize lock time
BEGIN;
SELECT product_id, price FROM products WHERE category_id = 1;
COMMIT;

-- Process data in application

BEGIN;
UPDATE products SET price = new_price WHERE product_id = specific_id;
COMMIT;
```

2. **Implicit Transactions**

```
-- Problem: Forgetting about auto-commit behavior
-- MySQL (auto-commit ON by default)
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
-- This commits immediately!
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
-- If this fails, first update is already committed

-- Solution: Explicit transaction control
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

3. **Nested Transaction Confusion**

```
-- Problem: Misunderstanding nested transactions
-- PostgreSQL doesn't support true nested transactions
BEGIN;
  INSERT INTO orders VALUES (...);
  BEGIN; -- This is ignored!
    INSERT INTO order_items VALUES (...);
  ROLLBACK; -- This rolls back the entire transaction!
COMMIT; -- This will fail because transaction was rolled back

-- Solution: Use savepoints
BEGIN;
```

```
  INSERT INTO orders VALUES (...);
  SAVEPOINT order_created;
    INSERT INTO order_items VALUES (...);
  ROLLBACK TO SAVEPOINT order_created; -- Only rolls back to savepoint
  -- Order insert is still valid
COMMIT;
```

4. **Deadlock-Prone Code Patterns**

```
-- Problem: Inconsistent lock ordering
-- Session 1:
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;

-- Session 2 (concurrent):
BEGIN;
UPDATE accounts SET balance = balance - 50 WHERE account_id = 2;
UPDATE accounts SET balance = balance + 50 WHERE account_id = 1;
COMMIT;
-- Deadlock!

-- Solution: Consistent lock ordering
-- Both sessions:
BEGIN;
-- Always lock lower account_id first
IF from_account < to_account THEN
  SELECT * FROM accounts WHERE account_id = from_account FOR UPDATE;
  SELECT * FROM accounts WHERE account_id = to_account FOR UPDATE;
ELSE
  SELECT * FROM accounts WHERE account_id = to_account FOR UPDATE;
  SELECT * FROM accounts WHERE account_id = from_account FOR UPDATE;
END IF;
-- Perform updates
COMMIT;
```

5. **Isolation Level Misunderstanding**

```
-- Problem: Using wrong isolation level
-- Using SERIALIZABLE for everything (performance impact)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT COUNT(*) FROM page_views; -- Simple analytics query

-- Solution: Use appropriate isolation level
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT COUNT(*) FROM page_views; -- Sufficient for analytics

-- Use SERIALIZABLE only when necessary
```

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- Critical financial calculations
SELECT SUM(balance) FROM accounts WHERE customer_id = 123;
```

6. **Connection Pool Exhaustion**

```
-- Problem: Holding connections too long
-- Application code (pseudo-code):
connection = get_connection()
connection.begin()
result = connection.execute("SELECT * FROM large_table")
-- Long processing time
process_large_result_set(result)
connection.commit()
connection.close()

-- Solution: Minimize connection hold time
connection = get_connection()
result = connection.execute("SELECT * FROM large_table")
connection.close()

-- Process data without holding connection
process_large_result_set(result)

-- New connection for updates
connection = get_connection()
connection.begin()
connection.execute("UPDATE table SET ...")
connection.commit()
connection.close()
```

7. **Phantom Read Confusion**

```
-- Problem: Not understanding phantom reads
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
SELECT COUNT(*) FROM orders WHERE status = 'pending'; -- Returns 10

-- Another session inserts new pending order

SELECT * FROM orders WHERE status = 'pending'; -- Might show 11 rows!
COMMIT;

-- Solution: Use SERIALIZABLE if phantom reads are problematic
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
SELECT COUNT(*) FROM orders WHERE status = 'pending';
-- New inserts will be blocked or cause serialization failure
```

```
SELECT * FROM orders WHERE status = 'pending';
COMMIT;
```

Summary

Transactions and concurrency control are fundamental to maintaining data integrity in multi-user database systems. Key takeaways:

- **ACID properties** ensure reliable transaction processing
- **Isolation levels** control what changes are visible between transactions
- **Locking mechanisms** prevent data corruption from concurrent access
- **Deadlock prevention** requires consistent resource ordering
- **Performance optimization** involves keeping transactions short and using appropriate isolation levels
- **Error handling** must account for serialization failures and deadlocks
- **Monitoring** helps identify performance bottlenecks and blocking issues

Master these concepts to build robust, scalable database applications that handle concurrent users safely and efficiently.

Next Steps

In the next chapter, we'll explore **Database Security and User Management**, covering authentication, authorization, encryption, and security best practices for protecting sensitive data.

Remember: Transactions are not just about data consistency—they're about building trust in your application's reliability.

Chapter 16: Database Security and User Management

What You'll Learn

Database security is critical for protecting sensitive data and ensuring compliance with regulations. This chapter covers authentication, authorization, encryption, auditing, and security best practices for MySQL and PostgreSQL.

Learning Objectives

By the end of this chapter, you will:

- Understand database security fundamentals
- Create and manage database users and roles
- Implement proper access control and permissions
- Configure SSL/TLS encryption
- Set up database auditing and monitoring

- Apply security best practices
- Handle common security vulnerabilities
- Implement data masking and anonymization

🔍 Concept Explanation

Database Security Fundamentals

The CIA Triad:

- **Confidentiality**: Data is accessible only to authorized users
- **Integrity**: Data remains accurate and unmodified
- **Availability**: Data is accessible when needed

Security Layers:

1. **Network Security**: Firewalls, VPNs, network segmentation
2. **Authentication**: Verifying user identity
3. **Authorization**: Controlling what users can do
4. **Encryption**: Protecting data in transit and at rest
5. **Auditing**: Tracking database activities
6. **Application Security**: Preventing SQL injection, etc.

Authentication vs Authorization

Authentication ("Who are you?"):

- Password-based authentication
- Certificate-based authentication
- Multi-factor authentication (MFA)
- LDAP/Active Directory integration
- OAuth/SAML integration

Authorization ("What can you do?"):

- Role-based access control (RBAC)
- Attribute-based access control (ABAC)
- Principle of least privilege
- Granular permissions

🛠️ User and Role Management

MySQL User Management

```
-- Create users
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'SecurePassword123!';
CREATE USER 'readonly_user'@'%' IDENTIFIED BY 'ReadOnlyPass456!';
CREATE USER 'admin_user'@'10.0.0.%' IDENTIFIED BY 'AdminPass789!';
```

```sql
-- Create user with SSL requirement
CREATE USER 'secure_user'@'%'
IDENTIFIED BY 'SecurePass123!'
REQUIRE SSL;

-- Create user with certificate authentication
CREATE USER 'cert_user'@'%'
IDENTIFIED BY 'CertPass123!'
REQUIRE X509;

-- View users
SELECT User, Host, ssl_type, ssl_cipher, x509_issuer, x509_subject
FROM mysql.user;

-- Modify user password
ALTER USER 'app_user'@'localhost' IDENTIFIED BY 'NewSecurePassword123!';

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

-- Account locking
ALTER USER 'suspicious_user'@'%' ACCOUNT LOCK;
ALTER USER 'suspicious_user'@'%' ACCOUNT UNLOCK;

-- Drop user
DROP USER 'old_user'@'localhost';

-- Create roles (MySQL 8.0+)
CREATE ROLE 'app_read', 'app_write', 'app_admin';

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

-- Grant roles to users
GRANT 'app_read' TO 'readonly_user'@'%';
GRANT 'app_write' TO 'app_user'@'localhost';
GRANT 'app_admin' TO 'admin_user'@'10.0.0.%';

-- Set default roles
SET DEFAULT ROLE 'app_read' TO 'readonly_user'@'%';
SET DEFAULT ROLE 'app_write' TO 'app_user'@'localhost';

-- Activate roles in session
SET ROLE 'app_admin';
SET ROLE ALL;
SET ROLE NONE;

-- View current roles
SELECT CURRENT_ROLE();
SHOW GRANTS FOR CURRENT_USER();
```

PostgreSQL User Management

```
-- Create users (roles)
CREATE USER app_user WITH PASSWORD 'SecurePassword123!';
CREATE USER readonly_user WITH PASSWORD 'ReadOnlyPass456!';
CREATE ROLE admin_user WITH LOGIN PASSWORD 'AdminPass789!';

-- Create user with connection limits
CREATE USER limited_user WITH
  PASSWORD 'LimitedPass123!'
  CONNECTION LIMIT 5;

-- Create user with expiration
CREATE USER temp_user WITH
  PASSWORD 'TempPass123!'
  VALID UNTIL '2024-12-31';

-- Create role without login (for grouping permissions)
CREATE ROLE app_readers;
CREATE ROLE app_writers;
CREATE ROLE app_admins;

-- Grant role membership
GRANT app_readers TO readonly_user;
GRANT app_writers TO app_user;
GRANT app_admins TO admin_user;

-- Create role with inheritance
CREATE ROLE manager WITH
  LOGIN
  PASSWORD 'ManagerPass123!'
  INHERIT; -- Can use privileges of granted roles automatically

-- Create role without inheritance
CREATE ROLE auditor WITH
  LOGIN
  PASSWORD 'AuditorPass123!'
  NOINHERIT; -- Must explicitly SET ROLE to use granted privileges

-- Modify user attributes
ALTER USER app_user WITH PASSWORD 'NewSecurePassword123!';
ALTER USER app_user VALID UNTIL '2025-12-31';
ALTER USER app_user CONNECTION LIMIT 10;

-- Disable/enable user
ALTER USER suspicious_user WITH NOLOGIN;
ALTER USER suspicious_user WITH LOGIN;

-- View users and roles
\du
```

```
-- or
SELECT rolname, rolsuper, rolinherit, rolcreaterole, rolcreatedb,
       rolcanlogin, rolconnlimit, rolvaliduntil
FROM pg_roles;

-- Drop user/role
DROP USER old_user;
DROP ROLE old_role;

-- Set role in session
SET ROLE app_admins;
SET ROLE NONE;
RESET ROLE;

-- View current user and roles
SELECT current_user, session_user;
SELECT * FROM pg_roles WHERE pg_has_role(current_user, oid, 'member');
```

Permissions and Privileges

MySQL Privilege System

```
-- Database-level privileges
GRANT ALL PRIVILEGES ON ecommerce.* TO 'admin_user'@'10.0.0.%';
GRANT SELECT, INSERT, UPDATE, DELETE ON ecommerce.* TO 'app_user'@'localhost';
GRANT SELECT ON ecommerce.* TO 'readonly_user'@'%';

-- Table-level privileges
GRANT SELECT, INSERT, UPDATE ON ecommerce.customers TO 'customer_service'@'%';
GRANT SELECT ON ecommerce.orders TO 'reporting_user'@'%';

-- Column-level privileges
GRANT SELECT (customer_id, first_name, last_name, email)
ON ecommerce.customers TO 'limited_user'@'%';

GRANT UPDATE (status, updated_at)
ON ecommerce.orders TO 'order_processor'@'%';

-- Stored procedure privileges
GRANT EXECUTE ON PROCEDURE ecommerce.process_order TO 'app_user'@'localhost';
GRANT EXECUTE ON FUNCTION ecommerce.calculate_tax TO 'app_user'@'localhost';

-- Administrative privileges
GRANT RELOAD, PROCESS, SHOW DATABASES ON *.* TO 'monitor_user'@'localhost';
GRANT REPLICATION SLAVE ON *.* TO 'replica_user'@'replica.example.com';

-- Grant with grant option (allows user to grant privileges to others)
GRANT SELECT ON ecommerce.products TO 'team_lead'@'%' WITH GRANT OPTION;

-- Revoke privileges
```

```
REVOKE INSERT, UPDATE ON ecommerce.customers FROM 'customer_service'@'%';
REVOKE ALL PRIVILEGES ON ecommerce.* FROM 'old_user'@'%';

-- View privileges
SHOW GRANTS FOR 'app_user'@'localhost';
SHOW GRANTS FOR CURRENT_USER();

-- View all user privileges
SELECT
    GRANTEE,
    TABLE_SCHEMA,
    TABLE_NAME,
    PRIVILEGE_TYPE,
    IS_GRANTABLE
FROM information_schema.TABLE_PRIVILEGES
WHERE GRANTEE = "'app_user'@'localhost'";

-- Complex privilege example: E-commerce application
-- Create specialized roles
CREATE ROLE 'ecommerce_customer_service';
CREATE ROLE 'ecommerce_inventory_manager';
CREATE ROLE 'ecommerce_financial_analyst';

-- Customer service role
GRANT SELECT, UPDATE ON ecommerce.customers TO 'ecommerce_customer_service';
GRANT SELECT ON ecommerce.orders TO 'ecommerce_customer_service';
GRANT SELECT ON ecommerce.order_items TO 'ecommerce_customer_service';
GRANT INSERT ON ecommerce.customer_support_tickets TO
'ecommerce_customer_service';

-- Inventory manager role
GRANT SELECT, INSERT, UPDATE ON ecommerce.products TO
'ecommerce_inventory_manager';
GRANT SELECT, INSERT, UPDATE ON ecommerce.inventory TO
'ecommerce_inventory_manager';
GRANT SELECT ON ecommerce.orders TO 'ecommerce_inventory_manager';
GRANT SELECT ON ecommerce.order_items TO 'ecommerce_inventory_manager';

-- Financial analyst role (read-only with specific tables)
GRANT SELECT ON ecommerce.orders TO 'ecommerce_financial_analyst';
GRANT SELECT ON ecommerce.payments TO 'ecommerce_financial_analyst';
GRANT SELECT ON ecommerce.refunds TO 'ecommerce_financial_analyst';
GRANT SELECT (customer_id, total_spent, registration_date)
ON ecommerce.customers TO 'ecommerce_financial_analyst';

-- Assign roles to users
GRANT 'ecommerce_customer_service' TO 'cs_agent1'@'%', 'cs_agent2'@'%';
GRANT 'ecommerce_inventory_manager' TO 'inventory_mgr'@'%';
GRANT 'ecommerce_financial_analyst' TO 'finance_team'@'%';
```

PostgreSQL Privilege System

```sql
-- Database-level privileges
GRANT ALL PRIVILEGES ON DATABASE ecommerce TO admin_user;
GRANT CONNECT ON DATABASE ecommerce TO app_user;
GRANT CONNECT ON DATABASE ecommerce TO readonly_user;

-- Schema-level privileges
GRANT ALL ON SCHEMA public TO admin_user;
GRANT USAGE ON SCHEMA public TO app_user;
GRANT USAGE ON SCHEMA public TO readonly_user;

-- Table-level privileges
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin_user;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO app_user;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly_user;

-- Grant on future tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_user;

ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT ON TABLES TO readonly_user;

-- Sequence privileges (for auto-increment)
GRANT USAGE ON ALL SEQUENCES IN SCHEMA public TO app_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT USAGE ON SEQUENCES TO app_user;

-- Function/procedure privileges
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public TO app_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT EXECUTE ON FUNCTIONS TO app_user;

-- Column-level privileges
GRANT SELECT (customer_id, first_name, last_name, email)
ON customers TO limited_user;

GRANT UPDATE (status, updated_at)
ON orders TO order_processor;

-- Row-level security (RLS)
ALTER TABLE customers ENABLE ROW LEVEL SECURITY;

-- Create policies
CREATE POLICY customer_isolation ON customers
    FOR ALL TO app_user
    USING (customer_id = current_setting('app.current_customer_id')::int);

CREATE POLICY admin_all_access ON customers
    FOR ALL TO admin_user
    USING (true);

-- Enable RLS for specific users
ALTER TABLE customers FORCE ROW LEVEL SECURITY;
```

```sql
-- Revoke privileges
REVOKE INSERT, UPDATE ON customers FROM customer_service;
REVOKE ALL PRIVILEGES ON DATABASE ecommerce FROM old_user;

-- View privileges
\dp customers
-- or
SELECT
    grantee,
    table_schema,
    table_name,
    privilege_type,
    is_grantable
FROM information_schema.table_privileges
WHERE grantee = 'app_user';

-- View row-level security policies
\d+ customers
-- or
SELECT schemaname, tablename, policyname, permissive, roles, cmd, qual
FROM pg_policies
WHERE tablename = 'customers';

-- Complex example: Multi-tenant application
-- Create tenant-specific roles
CREATE ROLE tenant_admin;
CREATE ROLE tenant_user;
CREATE ROLE tenant_readonly;

-- Set up row-level security for multi-tenancy
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;
ALTER TABLE customers ENABLE ROW LEVEL SECURITY;
ALTER TABLE products ENABLE ROW LEVEL SECURITY;

-- Tenant isolation policies
CREATE POLICY tenant_orders ON orders
    FOR ALL TO tenant_user, tenant_readonly
    USING (tenant_id = current_setting('app.current_tenant_id')::int);

CREATE POLICY tenant_customers ON customers
    FOR ALL TO tenant_user, tenant_readonly
    USING (tenant_id = current_setting('app.current_tenant_id')::int);

-- Admin can see all
CREATE POLICY admin_all_orders ON orders
    FOR ALL TO tenant_admin
    USING (true);

CREATE POLICY admin_all_customers ON customers
    FOR ALL TO tenant_admin
    USING (true);

-- Set tenant context in application
```

```
-- This would be done by the application before each request
SET app.current_tenant_id = '123';

-- Now queries will only return data for tenant 123
SELECT * FROM orders; -- Only shows orders for tenant 123
```

SSL/TLS Encryption

MySQL SSL Configuration

```
-- Check SSL status
SHOW VARIABLES LIKE '%ssl%';
STATUS; -- Shows SSL status in client

-- Check if SSL is enabled
SELECT @@have_ssl;

-- View SSL certificates
SHOW STATUS LIKE 'Ssl%';

-- Create user requiring SSL
CREATE USER 'ssl_user'@'%'
IDENTIFIED BY 'SecurePass123!'
REQUIRE SSL;

-- Create user requiring specific SSL certificate
CREATE USER 'cert_user'@'%'
IDENTIFIED BY 'CertPass123!'
REQUIRE X509;

-- Create user requiring specific certificate attributes
CREATE USER 'specific_cert_user'@'%'
IDENTIFIED BY 'SpecificCertPass123!'
REQUIRE ISSUER '/C=US/ST=CA/L=San Francisco/O=MyCompany/CN=MyCA'
AND SUBJECT '/C=US/ST=CA/L=San Francisco/O=MyCompany/CN=client';

-- Modify existing user to require SSL
ALTER USER 'existing_user'@'%' REQUIRE SSL;
ALTER USER 'existing_user'@'%' REQUIRE NONE; -- Remove SSL requirement

-- Check user SSL requirements
SELECT User, Host, ssl_type, ssl_cipher, x509_issuer, x509_subject
FROM mysql.user
WHERE User = 'ssl_user';

-- Connect with SSL (command line)
-- mysql --ssl-mode=REQUIRED -u ssl_user -p
-- mysql --ssl-ca=ca.pem --ssl-cert=client-cert.pem --ssl-key=client-key.pem -u
cert_user -p
```

PostgreSQL SSL Configuration

```
-- Check SSL status
SHOW ssl;
SELECT * FROM pg_stat_ssl WHERE pid = pg_backend_pid();

-- View SSL connection info
SELECT
    pid,
    usename,
    application_name,
    client_addr,
    ssl,
    ssl_version,
    ssl_cipher
FROM pg_stat_ssl
JOIN pg_stat_activity USING (pid)
WHERE ssl = true;

-- Configure SSL in postgresql.conf
-- ssl = on
-- ssl_cert_file = 'server.crt'
-- ssl_key_file = 'server.key'
-- ssl_ca_file = 'ca.crt'
-- ssl_crl_file = 'root.crl'

-- Configure client authentication in pg_hba.conf
-- hostssl all all 0.0.0.0/0 md5
-- hostssl all ssl_users 0.0.0.0/0 cert

-- Create user requiring SSL
CREATE USER ssl_user WITH PASSWORD 'SecurePass123!';
-- Then configure in pg_hba.conf:
-- hostssl ecommerce ssl_user 0.0.0.0/0 md5

-- Create user requiring client certificate
CREATE USER cert_user;
-- Configure in pg_hba.conf:
-- hostssl ecommerce cert_user 0.0.0.0/0 cert

-- Connect with SSL (command line)
-- psql "sslmode=require host=localhost dbname=ecommerce user=ssl_user"
-- psql "sslmode=require sslcert=client.crt sslkey=client.key sslrootcert=ca.crt
host=localhost dbname=ecommerce user=cert_user"
```

📊 Database Auditing

MySQL Audit Logging

```sql
-- Enable general query log
SET GLOBAL general_log = 'ON';
SET GLOBAL general_log_file = '/var/log/mysql/general.log';
SHOW VARIABLES LIKE 'general_log%';

-- Enable slow query log
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL slow_query_log_file = '/var/log/mysql/slow.log';
SET GLOBAL long_query_time = 2; -- Log queries taking more than 2 seconds
SHOW VARIABLES LIKE 'slow_query_log%';

-- Enable binary log (for replication and point-in-time recovery)
SHOW VARIABLES LIKE 'log_bin%';
SHOW BINARY LOGS;
SHOW BINLOG EVENTS IN 'mysql-bin.000001';

-- MySQL Enterprise Audit (commercial feature)
-- Install audit plugin
-- INSTALL PLUGIN audit_log SONAME 'audit_log.so';

-- Configure audit logging
-- SET GLOBAL audit_log_policy = 'ALL';
-- SET GLOBAL audit_log_format = 'JSON';
-- SET GLOBAL audit_log_file = '/var/log/mysql/audit.log';

-- View audit log status
-- SHOW STATUS LIKE 'audit_log%';

-- Custom audit table approach
CREATE TABLE audit_log (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    user_name VARCHAR(100),
    host_name VARCHAR(100),
    command_type VARCHAR(50),
    table_name VARCHAR(100),
    query_text TEXT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    affected_rows INT,
    INDEX idx_user_time (user_name, timestamp),
    INDEX idx_table_time (table_name, timestamp)
);

-- Create audit trigger
DELIMITER //
CREATE TRIGGER customers_audit_insert
AFTER INSERT ON customers
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (user_name, host_name, command_type, table_name,
query_text, affected_rows)
    VALUES (USER(), CONNECTION_ID(), 'INSERT', 'customers',
            CONCAT('INSERT INTO customers VALUES (', NEW.customer_id, ', "',
NEW.first_name, '", ...)'), 1);
```

```
END //

CREATE TRIGGER customers_audit_update
AFTER UPDATE ON customers
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (user_name, host_name, command_type, table_name,
query_text, affected_rows)
    VALUES (USER(), CONNECTION_ID(), 'UPDATE', 'customers',
            CONCAT('UPDATE customers SET ... WHERE customer_id = ',
NEW.customer_id), 1);
END //

CREATE TRIGGER customers_audit_delete
AFTER DELETE ON customers
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (user_name, host_name, command_type, table_name,
query_text, affected_rows)
    VALUES (USER(), CONNECTION_ID(), 'DELETE', 'customers',
            CONCAT('DELETE FROM customers WHERE customer_id = ', OLD.customer_id),
1);
END //
DELIMITER ;

-- Query audit log
SELECT
    user_name,
    command_type,
    table_name,
    COUNT(*) as operation_count,
    MIN(timestamp) as first_operation,
    MAX(timestamp) as last_operation
FROM audit_log
WHERE timestamp >= DATE_SUB(NOW(), INTERVAL 24 HOUR)
GROUP BY user_name, command_type, table_name
ORDER BY operation_count DESC;
```

PostgreSQL Audit Logging

```
-- Enable logging in postgresql.conf
-- log_statement = 'all'  # Log all statements
-- log_min_duration_statement = 1000  # Log slow queries (1 second)
-- log_connections = on
-- log_disconnections = on
-- log_duration = on
-- log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '

-- Check current logging settings
SHOW log_statement;
SHOW log_min_duration_statement;
```

```
SHOW log_connections;

-- pgAudit extension (third-party)
-- CREATE EXTENSION pgaudit;

-- Configure pgAudit
-- SET pgaudit.log = 'all';
-- SET pgaudit.log_catalog = off;
-- SET pgaudit.log_parameter = on;
-- SET pgaudit.log_relation = on;
-- SET pgaudit.log_statement_once = on;

-- Custom audit table approach
CREATE TABLE audit_log (
    id BIGSERIAL PRIMARY KEY,
    schema_name TEXT,
    table_name TEXT,
    user_name TEXT,
    action TEXT,
    original_data JSONB,
    new_data JSONB,
    query TEXT,
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_audit_log_table_time ON audit_log (table_name, timestamp);
CREATE INDEX idx_audit_log_user_time ON audit_log (user_name, timestamp);

-- Create audit function
CREATE OR REPLACE FUNCTION audit_trigger_function()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO audit_log (schema_name, table_name, user_name, action,
new_data, query)
        VALUES (TG_TABLE_SCHEMA, TG_TABLE_NAME, current_user, TG_OP,
row_to_json(NEW), current_query());
        RETURN NEW;
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO audit_log (schema_name, table_name, user_name, action,
original_data, new_data, query)
        VALUES (TG_TABLE_SCHEMA, TG_TABLE_NAME, current_user, TG_OP,
row_to_json(OLD), row_to_json(NEW), current_query());
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO audit_log (schema_name, table_name, user_name, action,
original_data, query)
        VALUES (TG_TABLE_SCHEMA, TG_TABLE_NAME, current_user, TG_OP,
row_to_json(OLD), current_query());
        RETURN OLD;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

```sql
-- Apply audit trigger to tables
CREATE TRIGGER customers_audit_trigger
    AFTER INSERT OR UPDATE OR DELETE ON customers
    FOR EACH ROW EXECUTE FUNCTION audit_trigger_function();

CREATE TRIGGER orders_audit_trigger
    AFTER INSERT OR UPDATE OR DELETE ON orders
    FOR EACH ROW EXECUTE FUNCTION audit_trigger_function();

-- Query audit log
SELECT
    user_name,
    table_name,
    action,
    COUNT(*) as operation_count,
    MIN(timestamp) as first_operation,
    MAX(timestamp) as last_operation
FROM audit_log
WHERE timestamp >= CURRENT_TIMESTAMP - INTERVAL '24 hours'
GROUP BY user_name, table_name, action
ORDER BY operation_count DESC;

-- View specific changes
SELECT
    timestamp,
    user_name,
    action,
    original_data,
    new_data
FROM audit_log
WHERE table_name = 'customers'
  AND timestamp >= CURRENT_TIMESTAMP - INTERVAL '1 hour'
ORDER BY timestamp DESC;
```

Security Best Practices

1. **Password Security**

```sql
-- MySQL: Password validation
-- Install password validation plugin
INSTALL PLUGIN validate_password SONAME 'validate_password.so';

-- Configure password policy
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;
```

```sql
-- Check password strength
SELECT VALIDATE_PASSWORD_STRENGTH('WeakPass');
SELECT VALIDATE_PASSWORD_STRENGTH('StrongP@ssw0rd123!');

-- PostgreSQL: Password encryption
-- Set password encryption method
SET password_encryption = 'scram-sha-256';

-- Create user with strong password
CREATE USER secure_user WITH PASSWORD 'StrongP@ssw0rd123!';

-- Check password encryption method
SELECT rolname, rolpassword FROM pg_authid WHERE rolname = 'secure_user';
```

2. **Network Security**

```sql
-- MySQL: Bind to specific interfaces
-- In my.cnf:
-- bind-address = 127.0.0.1  # Local only
-- bind-address = 10.0.0.5   # Specific IP

-- Skip name resolution for better security
-- skip-name-resolve = 1

-- PostgreSQL: Configure allowed connections
-- In postgresql.conf:
-- listen_addresses = 'localhost'  # Local only
-- listen_addresses = '10.0.0.5'   # Specific IP
-- port = 5432

-- In pg_hba.conf:
-- # TYPE  DATABASE        USER            ADDRESS                 METHOD
-- local   all             all                                     peer
-- host    all             all             127.0.0.1/32            md5
-- host    all             all             10.0.0.0/24             md5
-- hostssl all             all             0.0.0.0/0               md5

-- Disable remote root/superuser access
-- host    all             postgres        0.0.0.0/0               reject
```

3. **Principle of Least Privilege**

```sql
-- Create application-specific database
CREATE DATABASE ecommerce_app;

-- Create dedicated user for application
CREATE USER 'ecommerce_app'@'app-server.example.com'
IDENTIFIED BY 'SecureAppPassword123!';
```

```
-- Grant only necessary privileges
GRANT SELECT, INSERT, UPDATE, DELETE ON ecommerce_app.* TO 'ecommerce_app'@'app-
server.example.com';

-- Don't grant:
-- - CREATE, DROP (structure changes)
-- - GRANT OPTION (privilege escalation)
-- - SUPER, RELOAD (administrative functions)
-- - FILE (file system access)

-- Create read-only user for reporting
CREATE USER 'reporting_user'@'report-server.example.com'
IDENTIFIED BY 'ReportingPassword123!';

GRANT SELECT ON ecommerce_app.orders TO 'reporting_user'@'report-
server.example.com';
GRANT SELECT ON ecommerce_app.customers TO 'reporting_user'@'report-
server.example.com';
GRANT SELECT ON ecommerce_app.products TO 'reporting_user'@'report-
server.example.com';

-- Create backup user with minimal privileges
CREATE USER 'backup_user'@'backup-server.example.com'
IDENTIFIED BY 'BackupPassword123!';

GRANT SELECT, LOCK TABLES ON ecommerce_app.* TO 'backup_user'@'backup-
server.example.com';
GRANT RELOAD ON *.* TO 'backup_user'@'backup-server.example.com';
```

4. **SQL Injection Prevention**

```
-- Use parameterized queries (application level)
-- BAD (vulnerable to SQL injection):
-- query = "SELECT * FROM users WHERE username = '" + username + "' AND password =
'" + password + "'"

-- GOOD (parameterized query):
-- query = "SELECT * FROM users WHERE username = ? AND password = ?"
-- execute(query, [username, password])

-- Database-level protections
-- Disable dangerous functions if not needed
-- SET GLOBAL local_infile = 0;  -- MySQL

-- Use stored procedures for complex operations
DELIMITER //
CREATE PROCEDURE authenticate_user(
    IN p_username VARCHAR(100),
    IN p_password VARCHAR(255),
    OUT p_user_id INT,
    OUT p_is_valid BOOLEAN
```

```
)
BEGIN
    DECLARE v_stored_password VARCHAR(255);
    DECLARE v_user_id INT DEFAULT NULL;

    -- Get stored password hash
    SELECT user_id, password_hash INTO v_user_id, v_stored_password
    FROM users
    WHERE username = p_username AND status = 'active';

    -- Verify password (use proper hashing function)
    IF v_user_id IS NOT NULL AND v_stored_password = SHA2(CONCAT(p_password,
'salt'), 256) THEN
        SET p_user_id = v_user_id;
        SET p_is_valid = TRUE;

        -- Log successful login
        INSERT INTO login_log (user_id, login_time, ip_address, status)
        VALUES (v_user_id, NOW(), CONNECTION_ID(), 'success');
    ELSE
        SET p_user_id = NULL;
        SET p_is_valid = FALSE;

        -- Log failed login attempt
        INSERT INTO login_log (username, login_time, ip_address, status)
        VALUES (p_username, NOW(), CONNECTION_ID(), 'failed');
    END IF;
END //
DELIMITER ;

-- Grant execute permission only
GRANT EXECUTE ON PROCEDURE authenticate_user TO 'app_user'@'%';
```

5. **Data Encryption at Rest**

```
-- MySQL: Transparent Data Encryption (TDE)
-- Enable encryption for new tables
CREATE TABLE sensitive_data (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ssn VARCHAR(11),
    credit_card VARCHAR(16),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENCRYPTION='Y';

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

-- Check encryption status
SELECT
    TABLE_SCHEMA,
    TABLE_NAME,
```

```
    CREATE_OPTIONS
FROM information_schema.TABLES
WHERE CREATE_OPTIONS LIKE '%ENCRYPTION%';

-- PostgreSQL: Column-level encryption
-- Install pgcrypto extension
CREATE EXTENSION IF NOT EXISTS pgcrypto;

-- Encrypt sensitive data
CREATE TABLE sensitive_data (
    id SERIAL PRIMARY KEY,
    ssn_encrypted BYTEA,
    credit_card_encrypted BYTEA,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert encrypted data
INSERT INTO sensitive_data (ssn_encrypted, credit_card_encrypted)
VALUES (
    pgp_sym_encrypt('123-45-6789', 'encryption_key'),
    pgp_sym_encrypt('1234-5678-9012-3456', 'encryption_key')
);

-- Decrypt data
SELECT
    id,
    pgp_sym_decrypt(ssn_encrypted, 'encryption_key') AS ssn,
    pgp_sym_decrypt(credit_card_encrypted, 'encryption_key') AS credit_card
FROM sensitive_data;

-- Create function for secure access
CREATE OR REPLACE FUNCTION get_customer_sensitive_data(
    p_customer_id INT,
    p_encryption_key TEXT
) RETURNS TABLE(
    customer_id INT,
    ssn TEXT,
    credit_card TEXT
) AS $$
BEGIN
    -- Check if user has permission
    IF NOT pg_has_role(current_user, 'sensitive_data_access', 'member') THEN
        RAISE EXCEPTION 'Access denied: insufficient privileges';
    END IF;

    RETURN QUERY
    SELECT
        sd.id,
        pgp_sym_decrypt(sd.ssn_encrypted, p_encryption_key),
        pgp_sym_decrypt(sd.credit_card_encrypted, p_encryption_key)
    FROM sensitive_data sd
    WHERE sd.id = p_customer_id;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

🎭 Data Masking and Anonymization

Dynamic Data Masking

```
-- MySQL: Create masked views for sensitive data
CREATE VIEW customers_masked AS
SELECT
    customer_id,
    first_name,
    last_name,
    CASE
        WHEN CURRENT_USER() LIKE '%admin%' THEN email
        ELSE CONCAT(LEFT(email, 2), '***@', SUBSTRING_INDEX(email, '@', -1))
    END AS email,
    CASE
        WHEN CURRENT_USER() LIKE '%admin%' THEN phone
        ELSE CONCAT('***-***-', RIGHT(phone, 4))
    END AS phone,
    CASE
        WHEN CURRENT_USER() LIKE '%admin%' THEN address
        ELSE 'Address Hidden'
    END AS address,
    registration_date
FROM customers;

-- Grant access to masked view instead of table
GRANT SELECT ON customers_masked TO 'customer_service'@'%';
REVOKE SELECT ON customers FROM 'customer_service'@'%';

-- PostgreSQL: Row-level security with masking
CREATE OR REPLACE FUNCTION mask_sensitive_data()
RETURNS TRIGGER AS $$
BEGIN
    -- Check if user should see masked data
    IF NOT pg_has_role(current_user, 'full_data_access', 'member') THEN
        NEW.email := regexp_replace(NEW.email, '^(.{2}).*(@.*)$', '\1***\2');
        NEW.phone := regexp_replace(NEW.phone, '^(.{3}).*(.{4})$', '\1-***-\2');
        NEW.ssn := '***-**-' || right(NEW.ssn, 4);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Apply masking trigger
CREATE TRIGGER mask_customer_data
    BEFORE SELECT ON customers
    FOR EACH ROW
```

```
    WHEN (NOT pg_has_role(current_user, 'full_data_access', 'member'))
    EXECUTE FUNCTION mask_sensitive_data();
```

Data Anonymization for Testing

```
-- Create anonymized copy of production data
CREATE TABLE customers_test AS
SELECT
    customer_id,
    CONCAT('TestUser', customer_id) AS first_name,
    CONCAT('TestLast', customer_id) AS last_name,
    CONCAT('test', customer_id, '@example.com') AS email,
    CONCAT('555-', LPAD(FLOOR(RAND() * 10000), 4, '0'), '-', LPAD(FLOOR(RAND() *
10000), 4, '0')) AS phone,
    CONCAT(FLOOR(RAND() * 9999) + 1, ' Test St, Test City, TS ', LPAD(FLOOR(RAND()
* 100000), 5, '0')) AS address,
    registration_date,
    status
FROM customers;

-- PostgreSQL: More sophisticated anonymization
CREATE TABLE customers_anonymized AS
SELECT
    customer_id,
    'Anonymous' || customer_id AS first_name,
    'User' || customer_id AS last_name,
    'user' || customer_id || '@testdomain.com' AS email,
    '555-' || lpad((random() * 10000)::int::text, 4, '0') || '-' || lpad((random()
* 10000)::int::text, 4, '0') AS phone,
    (random() * 9999 + 1)::int || ' Anonymous St, Test City, TS ' ||
lpad((random() * 100000)::int::text, 5, '0') AS address,
    registration_date + (random() * interval '365 days') AS registration_date, --
Shift dates
    status
FROM customers;

-- Anonymize order amounts while preserving patterns
CREATE TABLE orders_anonymized AS
SELECT
    order_id,
    customer_id,
    -- Multiply by random factor to preserve relative amounts
    (total_amount * (0.5 + random()))::decimal(10,2) AS total_amount,
    order_date + (random() * interval '365 days') AS order_date,
    status
FROM orders;
```

Real-World Security Examples

Example 1: Healthcare Database Security

```
-- HIPAA-compliant patient data system
CREATE TABLE patients (
    patient_id SERIAL PRIMARY KEY,
    medical_record_number VARCHAR(20) UNIQUE NOT NULL,
    first_name_encrypted BYTEA,
    last_name_encrypted BYTEA,
    ssn_encrypted BYTEA,
    date_of_birth_encrypted BYTEA,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    last_accessed TIMESTAMP,
    last_accessed_by VARCHAR(100)
);

-- Enable row-level security
ALTER TABLE patients ENABLE ROW LEVEL SECURITY;

-- Create roles
CREATE ROLE healthcare_provider;
CREATE ROLE nurse;
CREATE ROLE admin_staff;
CREATE ROLE billing_staff;

-- Doctors can see all patient data
CREATE POLICY doctor_full_access ON patients
    FOR ALL TO healthcare_provider
    USING (true);

-- Nurses can only see patients they're assigned to
CREATE POLICY nurse_assigned_patients ON patients
    FOR SELECT TO nurse
    USING (
        patient_id IN (
            SELECT patient_id FROM patient_assignments
            WHERE nurse_id = current_setting('app.current_user_id')::int
        )
    );

-- Billing staff can only see billing-relevant data
CREATE POLICY billing_limited_access ON patients
    FOR SELECT TO billing_staff
    USING (true); -- But they'll use a view with limited columns

-- Create secure view for billing
CREATE VIEW patients_billing AS
SELECT
    patient_id,
    medical_record_number,
    'PROTECTED' AS first_name,
    'PROTECTED' AS last_name,
```

```
    created_at
FROM patients;

GRANT SELECT ON patients_billing TO billing_staff;

-- Audit access to patient data
CREATE OR REPLACE FUNCTION audit_patient_access()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO patient_access_log (
        patient_id,
        accessed_by,
        access_type,
        access_time,
        ip_address
    ) VALUES (
        COALESCE(NEW.patient_id, OLD.patient_id),
        current_user,
        TG_OP,
        CURRENT_TIMESTAMP,
        inet_client_addr()
    );

    -- Update last accessed info
    IF TG_OP = 'SELECT' OR TG_OP = 'UPDATE' THEN
        UPDATE patients
        SET last_accessed = CURRENT_TIMESTAMP,
            last_accessed_by = current_user
        WHERE patient_id = NEW.patient_id;
    END IF;

    RETURN COALESCE(NEW, OLD);
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER patient_access_audit
    AFTER SELECT OR INSERT OR UPDATE OR DELETE ON patients
    FOR EACH ROW EXECUTE FUNCTION audit_patient_access();
```

Example 2: Financial Services Security

```
-- Bank account system with strict security
CREATE TABLE accounts (
    account_id SERIAL PRIMARY KEY,
    account_number VARCHAR(20) UNIQUE NOT NULL,
    customer_id INT NOT NULL,
    account_type VARCHAR(20) NOT NULL,
    balance_encrypted BYTEA NOT NULL,
    status VARCHAR(20) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_transaction_at TIMESTAMP
```

```
);

CREATE TABLE transactions (
    transaction_id SERIAL PRIMARY KEY,
    from_account_id INT,
    to_account_id INT,
    amount_encrypted BYTEA NOT NULL,
    transaction_type VARCHAR(50) NOT NULL,
    description_encrypted BYTEA,
    reference_number VARCHAR(50) UNIQUE NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    processed_at TIMESTAMP,
    processed_by VARCHAR(100)
);

-- Create security roles
CREATE ROLE bank_teller;
CREATE ROLE account_manager;
CREATE ROLE compliance_officer;
CREATE ROLE system_admin;

-- Tellers can only see basic account info
CREATE POLICY teller_account_access ON accounts
    FOR SELECT TO bank_teller
    USING (
        customer_id IN (
            SELECT customer_id FROM teller_assignments
            WHERE teller_id = current_setting('app.current_user_id')::int
        )
    );

-- Account managers can see their assigned customers
CREATE POLICY manager_account_access ON accounts
    FOR ALL TO account_manager
    USING (
        customer_id IN (
            SELECT customer_id FROM customer_assignments
            WHERE manager_id = current_setting('app.current_user_id')::int
        )
    );

-- Compliance officers can see all data but read-only
CREATE POLICY compliance_read_access ON accounts
    FOR SELECT TO compliance_officer
    USING (true);

CREATE POLICY compliance_read_transactions ON transactions
    FOR SELECT TO compliance_officer
    USING (true);

-- Secure transaction function
CREATE OR REPLACE FUNCTION secure_transfer(
    p_from_account VARCHAR(20),
```

```
    p_to_account VARCHAR(20),
    p_amount DECIMAL(15,2),
    p_description TEXT,
    p_encryption_key TEXT
) RETURNS TABLE(
    success BOOLEAN,
    transaction_id INT,
    message TEXT
) SECURITY DEFINER AS $$
DECLARE
    v_from_balance DECIMAL(15,2);
    v_transaction_id INT;
BEGIN
    -- Verify user has transfer privileges
    IF NOT pg_has_role(current_user, 'transfer_authorized', 'member') THEN
        RETURN QUERY SELECT FALSE, NULL::INT, 'Unauthorized: Transfer privilege
required';
        RETURN;
    END IF;

    -- Additional security checks
    IF p_amount > 10000 AND NOT pg_has_role(current_user, 'high_value_transfer',
'member') THEN
        RETURN QUERY SELECT FALSE, NULL::INT, 'Unauthorized: High value transfer
requires special authorization';
        RETURN;
    END IF;

    -- Proceed with secure transfer logic
    -- (Implementation similar to previous examples but with encryption)

    RETURN QUERY SELECT TRUE, v_transaction_id, 'Transfer completed successfully';
END;
$$ LANGUAGE plpgsql;

-- Grant execute permission only to authorized roles
GRANT EXECUTE ON FUNCTION secure_transfer TO bank_teller, account_manager;
```

🎯 Use Cases & Interview Tips

Common Interview Questions:

1. **"How do you secure a database?"**
    - **Network**: Firewalls, VPNs, SSL/TLS
    - **Authentication**: Strong passwords, MFA, certificates
    - **Authorization**: RBAC, principle of least privilege
    - **Encryption**: Data at rest and in transit
    - **Auditing**: Log all access and changes
    - **Monitoring**: Detect suspicious activities

2. **"What's the difference between authentication and authorization?"**
    - **Authentication**: Verifying identity ("Who are you?")
    - **Authorization**: Controlling access ("What can you do?")
    - Example: Login (authentication) → Check permissions (authorization)
3. **"How do you prevent SQL injection?"**
    - **Parameterized queries**: Use placeholders for user input
    - **Input validation**: Sanitize and validate all inputs
    - **Stored procedures**: Encapsulate database logic
    - **Least privilege**: Limit database user permissions
    - **WAF**: Web Application Firewall for additional protection
4. **"Explain row-level security."**
    - **Concept**: Filter rows based on user context
    - **Implementation**: Policies that define access rules
    - **Use cases**: Multi-tenant applications, data privacy
    - **Performance**: Consider impact on query performance

Security Best Practices:

1. **Defense in depth**: Multiple security layers
2. **Regular updates**: Keep database software current
3. **Strong authentication**: Complex passwords, MFA
4. **Principle of least privilege**: Minimal necessary access
5. **Encryption everywhere**: Data at rest and in transit
6. **Comprehensive auditing**: Log all database activities
7. **Regular security assessments**: Penetration testing, vulnerability scans
8. **Incident response plan**: Prepare for security breaches

⚠ Things to Watch Out For

1. **Overprivileged Users**

```
-- Problem: Giving too many privileges
GRANT ALL PRIVILEGES ON *.* TO 'app_user'@'%'; -- Too broad!

-- Solution: Grant specific privileges
GRANT SELECT, INSERT, UPDATE, DELETE ON ecommerce.* TO 'app_user'@'%';
```

2. **Weak Password Policies**

```
-- Problem: Weak passwords
CREATE USER 'admin'@'%' IDENTIFIED BY '123456';
```

```
-- Solution: Strong password requirements
CREATE USER 'admin'@'%' IDENTIFIED BY 'StrongP@ssw0rd123!';
```

3. **Unencrypted Connections**

```
-- Problem: Allowing unencrypted connections
-- host all all 0.0.0.0/0 md5

-- Solution: Require SSL
-- hostssl all all 0.0.0.0/0 md5
```

4. **Missing Audit Trails**

```
-- Problem: No logging of sensitive operations
-- Solution: Comprehensive audit logging
CREATE TRIGGER audit_sensitive_table
    AFTER INSERT OR UPDATE OR DELETE ON sensitive_table
    FOR EACH ROW EXECUTE FUNCTION audit_function();
```

5. **Storing Sensitive Data in Plain Text**

```
-- Problem: Plain text sensitive data
CREATE TABLE users (
    id INT,
    username VARCHAR(50),
    password VARCHAR(50), -- Plain text!
    ssn VARCHAR(11)       -- Plain text!
);

-- Solution: Encrypt sensitive data
CREATE TABLE users (
    id INT,
    username VARCHAR(50),
    password_hash VARCHAR(255), -- Hashed
    ssn_encrypted BYTEA         -- Encrypted
);
```

Summary

Database security is a multi-layered approach that requires:

- **Strong authentication** and authorization mechanisms
- **Proper user and role management** with least privilege
- **Encryption** for data protection at rest and in transit

- **Comprehensive auditing** for compliance and monitoring
- **Regular security assessments** and updates
- **Defense in depth** strategy with multiple security layers

Implementing these security measures protects sensitive data, ensures compliance with regulations, and maintains user trust in your applications.

🔗 Next Steps

In the next chapter, we'll explore **Backup and Recovery**, covering backup strategies, point-in-time recovery, disaster recovery planning, and high availability configurations.

Remember: Security is not a one-time setup—it's an ongoing process that requires constant vigilance and updates.

Chapter 17: Backup and Recovery

🗄️ What You'll Learn

Database backup and recovery are critical for protecting data against hardware failures, human errors, corruption, and disasters. This chapter covers comprehensive backup strategies, recovery procedures, and high availability configurations for MySQL and PostgreSQL.

🎯 Learning Objectives

By the end of this chapter, you will:

- Understand different types of database backups
- Implement automated backup strategies
- Perform point-in-time recovery
- Configure high availability and replication
- Plan disaster recovery procedures
- Monitor backup integrity and performance
- Handle various failure scenarios

🔍 Concept Explanation

Why Backup and Recovery Matter

Data Loss Scenarios:

- **Hardware Failures**: Disk crashes, server failures
- **Human Errors**: Accidental deletions, wrong updates
- **Software Bugs**: Application errors, corruption
- **Security Breaches**: Malicious attacks, ransomware
- **Natural Disasters**: Fire, flood, earthquakes

Business Impact:

- **Downtime Costs**: Lost revenue, productivity
- **Data Loss**: Irreplaceable business information
- **Compliance**: Legal and regulatory requirements
- **Reputation**: Customer trust and brand damage

Key Concepts

Recovery Point Objective (RPO): Maximum acceptable data loss **Recovery Time Objective (RTO)**: Maximum acceptable downtime **Backup Types**: Full, incremental, differential **Recovery Types**: Complete, point-in-time, partial

---

🛠️ Backup Strategies

MySQL Backup Methods

1. Logical Backups with mysqldump

```
# Full database backup
mysqldump -u root -p --all-databases > full_backup.sql

# Single database backup
mysqldump -u root -p ecommerce_db > ecommerce_backup.sql

# Backup with additional options
mysqldump -u root -p \
  --single-transaction \
  --routines \
  --triggers \
  --events \
  --all-databases > complete_backup.sql

# Compressed backup
mysqldump -u root -p ecommerce_db | gzip > ecommerce_backup.sql.gz

# Backup specific tables
mysqldump -u root -p ecommerce_db customers orders > tables_backup.sql
```

2. Physical Backups

```
# MySQL Enterprise Backup (Commercial)
mysqlbackup --user=root --password \
  --backup-dir=/backup/mysql \
  backup-and-apply-log

# Percona XtraBackup (Open Source)
xtrabackup --user=root --password=password \
```

```
  --backup --target-dir=/backup/mysql/

# Hot backup with XtraBackup
xtrabackup --user=root --password=password \
  --backup \
  --target-dir=/backup/mysql/$(date +%Y%m%d_%H%M%S)
```

3. Binary Log Backups

```
-- Enable binary logging in my.cnf
[mysqld]
log-bin=mysql-bin
server-id=1
binlog-format=ROW
expire_logs_days=7

-- Flush and backup binary logs
FLUSH LOGS;

-- Copy binary log files
-- mysql-bin.000001, mysql-bin.000002, etc.
```

PostgreSQL Backup Methods

1. Logical Backups with pg_dump

```
# Full database backup
pg_dump -U postgres -h localhost ecommerce_db > ecommerce_backup.sql

# Compressed backup
pg_dump -U postgres -h localhost -Fc ecommerce_db > ecommerce_backup.dump

# All databases backup
pg_dumpall -U postgres -h localhost > all_databases.sql

# Backup with specific options
pg_dump -U postgres -h localhost \
  --verbose \
  --clean \
  --if-exists \
  --format=custom \
  ecommerce_db > ecommerce_backup.dump

# Parallel backup (faster for large databases)
pg_dump -U postgres -h localhost \
  --format=directory \
  --jobs=4 \
```

```
  --file=ecommerce_backup_dir \
  ecommerce_db
```

2. Physical Backups with pg_basebackup

```
# Base backup
pg_basebackup -U postgres -h localhost \
  -D /backup/postgresql/base \
  -Ft -z -P

# Streaming backup
pg_basebackup -U postgres -h localhost \
  -D /backup/postgresql/$(date +%Y%m%d_%H%M%S) \
  -X stream -P

# WAL archiving setup in postgresql.conf
wal_level = replica
archive_mode = on
archive_command = 'cp %p /backup/postgresql/wal/%f'
```

🔄 Automated Backup Scripts

MySQL Automated Backup Script

```
#!/bin/bash
# mysql_backup.sh

# Configuration
DB_USER="backup_user"
DB_PASS="secure_password"
BACKUP_DIR="/backup/mysql"
DATE=$(date +%Y%m%d_%H%M%S)
RETENTION_DAYS=30

# Create backup directory
mkdir -p $BACKUP_DIR

# Function to backup database
backup_database() {
    local db_name=$1
    local backup_file="$BACKUP_DIR/${db_name}_${DATE}.sql.gz"

    echo "Backing up database: $db_name"
    mysqldump -u $DB_USER -p$DB_PASS \
        --single-transaction \
        --routines \
        --triggers \
        --events \
```

```
        $db_name | gzip > $backup_file

    if [ $? -eq 0 ]; then
        echo "Backup successful: $backup_file"
    else
        echo "Backup failed for: $db_name"
        exit 1
    fi
}

# Backup all databases
DATABASES=$(mysql -u $DB_USER -p$DB_PASS -e "SHOW DATABASES;" | grep -Ev
"^(Database|information_schema|performance_schema|mysql|sys)$")

for db in $DATABASES; do
    backup_database $db
done

# Clean old backups
find $BACKUP_DIR -name "*.sql.gz" -mtime +$RETENTION_DAYS -delete

echo "Backup process completed at $(date)"
```

PostgreSQL Automated Backup Script

```
#!/bin/bash
# postgresql_backup.sh

# Configuration
export PGUSER="postgres"
export PGPASSWORD="secure_password"
BACKUP_DIR="/backup/postgresql"
DATE=$(date +%Y%m%d_%H%M%S)
RETENTION_DAYS=30

# Create backup directory
mkdir -p $BACKUP_DIR

# Function to backup database
backup_database() {
    local db_name=$1
    local backup_file="$BACKUP_DIR/${db_name}_${DATE}.dump"

    echo "Backing up database: $db_name"
    pg_dump -h localhost \
        --format=custom \
        --compress=9 \
        --verbose \
        --file=$backup_file \
        $db_name
```

```
    if [ $? -eq 0 ]; then
        echo "Backup successful: $backup_file"
    else
        echo "Backup failed for: $db_name"
        exit 1
    fi
}

# Get list of databases
DATABASES=$(psql -h localhost -t -c "SELECT datname FROM pg_database WHERE NOT
datistemplate AND datname != 'postgres';")

for db in $DATABASES; do
    # Remove whitespace
    db=$(echo $db | xargs)
    backup_database $db
done

# Clean old backups
find $BACKUP_DIR -name "*.dump" -mtime +$RETENTION_DAYS -delete

echo "Backup process completed at $(date)"
```

🔧 Recovery Procedures

MySQL Recovery

1. Full Database Restore

```
# Restore from mysqldump
mysql -u root -p < full_backup.sql

# Restore specific database
mysql -u root -p ecommerce_db < ecommerce_backup.sql

# Restore compressed backup
gunzip < ecommerce_backup.sql.gz | mysql -u root -p ecommerce_db
```

2. Point-in-Time Recovery

```
# Step 1: Restore from full backup
mysql -u root -p < full_backup_20241201_020000.sql

# Step 2: Apply binary logs up to specific time
mysqlbinlog --stop-datetime="2024-12-01 14:30:00" \
    mysql-bin.000001 mysql-bin.000002 | mysql -u root -p
```

```
# Alternative: Stop at specific position
mysqlbinlog --stop-position=12345 \
    mysql-bin.000001 | mysql -u root -p
```

3. XtraBackup Recovery

```
# Prepare the backup
xtrabackup --prepare --target-dir=/backup/mysql/20241201_020000

# Stop MySQL service
sudo systemctl stop mysql

# Remove old data directory
sudo rm -rf /var/lib/mysql/*

# Copy backup to data directory
xtrabackup --copy-back --target-dir=/backup/mysql/20241201_020000

# Fix permissions
sudo chown -R mysql:mysql /var/lib/mysql

# Start MySQL service
sudo systemctl start mysql
```

PostgreSQL Recovery

1. Database Restore

```
# Restore from pg_dump (SQL format)
psql -U postgres -h localhost -d ecommerce_db < ecommerce_backup.sql

# Restore from custom format
pg_restore -U postgres -h localhost \
    --dbname=ecommerce_db \
    --verbose \
    ecommerce_backup.dump

# Restore with parallel jobs
pg_restore -U postgres -h localhost \
    --dbname=ecommerce_db \
    --jobs=4 \
    --verbose \
    ecommerce_backup_dir

# Create database and restore
createdb -U postgres ecommerce_db_restored
pg_restore -U postgres -h localhost \
```

```
    --dbname=ecommerce_db_restored \
    ecommerce_backup.dump
```

2. Point-in-Time Recovery (PITR)

```
# Step 1: Stop PostgreSQL
sudo systemctl stop postgresql

# Step 2: Restore base backup
rm -rf /var/lib/postgresql/13/main/*
tar -xzf base_backup.tar.gz -C /var/lib/postgresql/13/main/

# Step 3: Create recovery.conf (PostgreSQL < 12) or recovery.signal (PostgreSQL >=
12)
echo "restore_command = 'cp /backup/postgresql/wal/%f %p'" >
/var/lib/postgresql/13/main/recovery.conf
echo "recovery_target_time = '2024-12-01 14:30:00'" >>
/var/lib/postgresql/13/main/recovery.conf

# For PostgreSQL >= 12
touch /var/lib/postgresql/13/main/recovery.signal
echo "restore_command = 'cp /backup/postgresql/wal/%f %p'" >>
/var/lib/postgresql/13/main/postgresql.conf
echo "recovery_target_time = '2024-12-01 14:30:00'" >>
/var/lib/postgresql/13/main/postgresql.conf

# Step 4: Start PostgreSQL
sudo systemctl start postgresql
```

High Availability and Replication

MySQL Replication

Master-Slave Replication Setup

```
-- Master configuration (my.cnf)
[mysqld]
server-id=1
log-bin=mysql-bin
binlog-format=ROW
binlog-do-db=ecommerce_db

-- Create replication user on master
CREATE USER 'replication'@'%' IDENTIFIED BY 'secure_password';
GRANT REPLICATION SLAVE ON *.* TO 'replication'@'%';
FLUSH PRIVILEGES;
```

```
-- Get master status
SHOW MASTER STATUS;
-- Note: File and Position values

-- Slave configuration (my.cnf)
[mysqld]
server-id=2
relay-log=mysql-relay-bin
read-only=1

-- Configure slave
CHANGE MASTER TO
    MASTER_HOST='master_ip',
    MASTER_USER='replication',
    MASTER_PASSWORD='secure_password',
    MASTER_LOG_FILE='mysql-bin.000001',
    MASTER_LOG_POS=12345;

-- Start replication
START SLAVE;

-- Check slave status
SHOW SLAVE STATUS\G
```

MySQL Group Replication

```
-- Install Group Replication plugin
INSTALL PLUGIN group_replication SONAME 'group_replication.so';

-- Configure Group Replication
SET GLOBAL group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa";
SET GLOBAL group_replication_start_on_boot=off;
SET GLOBAL group_replication_local_address="192.168.1.10:33061";
SET GLOBAL
group_replication_group_seeds="192.168.1.10:33061,192.168.1.11:33061,192.168.1.12:
33061";
SET GLOBAL group_replication_bootstrap_group=on;

-- Start Group Replication
START GROUP_REPLICATION;

-- Check group members
SELECT * FROM performance_schema.replication_group_members;
```

PostgreSQL Replication

Streaming Replication Setup

```
# Primary server configuration (postgresql.conf)
wal_level = replica
max_wal_senders = 3
wal_keep_segments = 64
archive_mode = on
archive_command = 'cp %p /backup/postgresql/wal/%f'

# Create replication user
psql -U postgres -c "CREATE USER replication REPLICATION LOGIN PASSWORD
'secure_password';"

# Configure pg_hba.conf on primary
echo "host replication replication 192.168.1.0/24 md5" >>
/etc/postgresql/13/main/pg_hba.conf

# Setup standby server
pg_basebackup -h primary_ip -D /var/lib/postgresql/13/main -U replication -P -W

# Configure standby (postgresql.conf)
hot_standby = on

# Create standby.signal file
touch /var/lib/postgresql/13/main/standby.signal

# Configure primary connection info
echo "primary_conninfo = 'host=primary_ip port=5432 user=replication
password=secure_password'" >> /var/lib/postgresql/13/main/postgresql.conf
```

PostgreSQL Logical Replication

```
-- Publisher setup
ALTER SYSTEM SET wal_level = logical;
SELECT pg_reload_conf();

-- Create publication
CREATE PUBLICATION ecommerce_pub FOR TABLE customers, orders, products;

-- Subscriber setup
CREATE SUBSCRIPTION ecommerce_sub
CONNECTION 'host=publisher_ip dbname=ecommerce_db user=replication
password=secure_password'
PUBLICATION ecommerce_pub;

-- Monitor replication
SELECT * FROM pg_stat_replication;
SELECT * FROM pg_stat_subscription;
```

📊 Monitoring and Maintenance

Backup Monitoring

```
-- MySQL: Check binary log status
SHOW BINARY LOGS;
SHOW MASTER STATUS;

-- Check slave lag
SHOW SLAVE STATUS\G

-- PostgreSQL: Check WAL status
SELECT pg_current_wal_lsn();
SELECT * FROM pg_stat_archiver;

-- Check replication lag
SELECT
    client_addr,
    state,
    pg_wal_lsn_diff(pg_current_wal_lsn(), sent_lsn) as send_lag,
    pg_wal_lsn_diff(sent_lsn, flush_lsn) as receive_lag
FROM pg_stat_replication;
```

Backup Verification Script

```
#!/bin/bash
# verify_backup.sh

BACKUP_FILE=$1
TEST_DB="backup_test_$(date +%s)"

if [ -z "$BACKUP_FILE" ]; then
    echo "Usage: $0 <backup_file>"
    exit 1
fi

echo "Verifying backup: $BACKUP_FILE"

# Create test database
mysql -u root -p -e "CREATE DATABASE $TEST_DB;"

# Restore backup to test database
if [[ $BACKUP_FILE == *.gz ]]; then
    gunzip < $BACKUP_FILE | mysql -u root -p $TEST_DB
else
    mysql -u root -p $TEST_DB < $BACKUP_FILE
fi

if [ $? -eq 0 ]; then
    echo "Backup verification successful"
    # Run basic integrity checks
    mysql -u root -p $TEST_DB -e "CHECK TABLE customers, orders, products;"
```

```
else
    echo "Backup verification failed"
fi

# Cleanup
mysql -u root -p -e "DROP DATABASE $TEST_DB;"
```

Disaster Recovery Planning

Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO)

Scenario	RTO Target	RPO Target	Strategy
Hardware Failure	< 1 hour	< 15 minutes	Hot standby with replication
Data Corruption	< 4 hours	< 1 hour	Point-in-time recovery
Site Disaster	< 24 hours	< 1 hour	Geographic replication
Human Error	< 2 hours	< 30 minutes	Transaction log replay

Disaster Recovery Checklist

```
## Pre-Disaster Preparation

- [ ] Document all backup procedures
- [ ] Test recovery procedures monthly
- [ ] Maintain off-site backup copies
- [ ] Configure monitoring and alerting
- [ ] Train staff on recovery procedures
- [ ] Maintain emergency contact list

## During Disaster

- [ ] Assess the scope of the problem
- [ ] Activate disaster recovery team
- [ ] Communicate with stakeholders
- [ ] Execute recovery procedures
- [ ] Document all actions taken
- [ ] Monitor recovery progress

## Post-Disaster

- [ ] Verify data integrity
- [ ] Test application functionality
- [ ] Update documentation
- [ ] Conduct post-mortem analysis
- [ ] Improve procedures based on lessons learned
```

Cloud Backup Strategies

```
# AWS S3 backup
aws s3 cp backup.sql.gz s3://company-db-backups/mysql/$(date +%Y/%m/%d)/

# Google Cloud Storage
gsutil cp backup.sql.gz gs://company-db-backups/mysql/$(date +%Y/%m/%d)/

# Azure Blob Storage
az storage blob upload \
    --account-name companystorage \
    --container-name db-backups \
    --name mysql/$(date +%Y/%m/%d)/backup.sql.gz \
    --file backup.sql.gz
```

Real-World Examples

Example 1: E-commerce Platform Recovery

Scenario: Online store database corruption during peak shopping season

```
# 1. Immediate response
# Switch to read-only mode
mysql -u root -p -e "SET GLOBAL read_only = ON;"

# 2. Assess damage
mysql -u root -p ecommerce_db -e "CHECK TABLE orders, customers, products;"

# 3. Point-in-time recovery to just before corruption
# Restore from last good backup (2 hours ago)
mysql -u root -p < ecommerce_backup_20241201_140000.sql

# Apply binary logs up to corruption point
mysqlbinlog --stop-datetime="2024-12-01 16:25:00" \
    mysql-bin.000015 | mysql -u root -p ecommerce_db

# 4. Verify data integrity
mysql -u root -p ecommerce_db -e "
    SELECT COUNT(*) FROM orders WHERE order_date = CURDATE();
    SELECT COUNT(*) FROM customers WHERE created_at = CURDATE();
"

# 5. Resume normal operations
mysql -u root -p -e "SET GLOBAL read_only = OFF;"
```

Example 2: PostgreSQL Failover

Scenario: Primary database server hardware failure

```
# 1. Promote standby to primary
psql -U postgres -c "SELECT pg_promote();"

# 2. Update application connection strings
# Point applications to new primary server

# 3. Verify replication status
psql -U postgres -c "SELECT pg_is_in_recovery();"

# 4. Setup new standby server
pg_basebackup -h new_primary_ip -D /var/lib/postgresql/13/main -U replication -P -
W

# 5. Configure new standby
echo "primary_conninfo = 'host=new_primary_ip port=5432 user=replication'" >>
postgresql.conf
touch standby.signal
```

🎯 Use Cases & Interview Tips

Common Interview Questions

1. **"How would you design a backup strategy for a 24/7 e-commerce application?"**
    - Continuous replication for high availability
    - Automated daily full backups
    - Hourly incremental backups
    - Off-site backup storage
    - Regular recovery testing
2. **"What's the difference between hot, warm, and cold backups?"**
    - **Hot**: Database remains online and accessible
    - **Warm**: Database is online but in read-only mode
    - **Cold**: Database is shut down during backup
3. **"How do you ensure backup integrity?"**
    - Checksum verification
    - Regular restore testing
    - Backup validation scripts
    - Monitoring backup completion
    - Multiple backup copies
4. **"Explain point-in-time recovery."**
    - Restore from full backup
    - Apply transaction logs up to specific time
    - Useful for recovering from human errors

- Requires continuous log archiving

Best Practices

1. **3-2-1 Rule**: 3 copies of data, 2 different media types, 1 off-site
2. **Test Regularly**: Recovery procedures should be tested monthly
3. **Automate Everything**: Reduce human error with automation
4. **Monitor Continuously**: Alert on backup failures immediately
5. **Document Thoroughly**: Clear procedures for emergency situations

---

⚠ Things to Watch Out For

Common Backup Mistakes

1. **Not Testing Restores**

```
# Bad: Only creating backups
mysqldump -u root -p database > backup.sql

# Good: Testing restore process
mysql -u root -p test_db < backup.sql
```

2. **Insufficient Retention**

```
# Bad: Only keeping recent backups
find /backup -name "*.sql" -mtime +7 -delete

# Good: Graduated retention policy
# Keep daily for 30 days, weekly for 12 weeks, monthly for 12 months
```

3. **No Off-site Storage**

```
# Bad: All backups on same server
cp backup.sql /local/backup/

# Good: Remote storage
rsync backup.sql remote-server:/backup/
aws s3 cp backup.sql s3://backup-bucket/
```

4. **Ignoring Binary Logs**

```
-- Bad: Only full backups
mysqldump --all-databases > backup.sql
```

```
-- Good: Enable binary logging for point-in-time recovery
SET GLOBAL log_bin = ON;
```

Security Considerations

1. **Encrypt Backup Files**

```
# Encrypt during backup
mysqldump -u root -p database | gpg --encrypt -r backup@company.com >
backup.sql.gpg

# Decrypt during restore
gpg --decrypt backup.sql.gpg | mysql -u root -p database
```

2. **Secure Backup Storage**

```
# Set proper permissions
chmod 600 backup.sql
chown backup:backup backup.sql

# Use secure transfer protocols
scp backup.sql backup-server:/secure/location/
```

3. **Backup User Privileges**

```
-- Create dedicated backup user with minimal privileges
CREATE USER 'backup_user'@'localhost' IDENTIFIED BY 'secure_password';
GRANT SELECT, LOCK TABLES, SHOW VIEW, EVENT, TRIGGER ON *.* TO
'backup_user'@'localhost';
```

🔗 Next Steps

🎉 **Congratulations!** You've now completed the comprehensive SQL learning path covering all essential database topics!

🚀 You've Mastered:

- **Database Fundamentals** (RDBMS, ACID properties)
- **Core SQL Operations** (CRUD, joins, subqueries)
- **Advanced Analytics** (Window functions, CTEs)
- **Performance Optimization** (Indexes, query tuning)
- **Database Programming** (Stored procedures, triggers)
- **Enterprise Features** (Transactions, security, user management)
- **Backup and Recovery** (Disaster recovery, high availability)

🎯 What's Next?

- **Practice Projects**: Build real-world applications using your SQL skills
- **Specialization**: Choose MySQL or PostgreSQL for deeper expertise
- **Integration**: Learn how SQL works with programming languages (Python, Node.js, etc.)
- **Advanced Topics**: Explore database administration, replication, and cloud databases
- **Certification**: Consider MySQL or PostgreSQL certification exams

📚 Continue Learning:

- Check out other branches in this repository (JavaScript, React, etc.)
- Build full-stack applications combining SQL with web technologies
- Explore NoSQL databases for comparison and broader database knowledge
- Learn about database DevOps and infrastructure as code

You're now ready to work as a confident database professional! 🚀

This completes the comprehensive SQL learning journey. You now have the knowledge and skills to design, implement, optimize, and maintain production database systems.

Chapter 18: Database Replication & High Availability

📚 What You'll Learn

Database replication and high availability are critical for production systems that require minimal downtime and data protection. This chapter covers master-slave replication, clustering, failover strategies, and load balancing for both MySQL and PostgreSQL.

🎯 Learning Objectives

- **Master-Slave Replication**: Understand primary-replica architecture
- **Master-Master Replication**: Bidirectional data synchronization
- **Read Replicas**: Scale read operations across multiple servers
- **Failover Strategies**: Automatic and manual failover procedures
- **Load Balancing**: Distribute database load effectively
- **Monitoring**: Track replication lag and health
- **Conflict Resolution**: Handle replication conflicts

📖 Concept Explanation

What is Database Replication?

Database replication is the process of copying and maintaining database objects in multiple database instances. It ensures data availability, improves performance, and provides disaster recovery capabilities.

Master-Slave vs Master-Master

Master-Slave (Primary-Replica):

- One primary server handles writes
- Multiple replica servers handle reads
- Unidirectional data flow
- Simpler to manage, less conflict potential

Master-Master (Multi-Master):

- Multiple servers can handle writes
- Bidirectional data synchronization
- More complex, requires conflict resolution
- Higher availability but increased complexity

🔧 MySQL Replication

Setting Up Master-Slave Replication

1. Configure Master Server

```
-- Edit /etc/mysql/mysql.conf.d/mysqld.cnf
[mysqld]
server-id = 1
log-bin = mysql-bin
binlog-format = ROW
binlog-do-db = production_db

-- Restart MySQL service
-- sudo systemctl restart mysql
```

2. Create Replication User

```
-- On Master server
CREATE USER 'replication_user'@'%'
IDENTIFIED BY 'strong_password';

GRANT REPLICATION SLAVE ON *.*
TO 'replication_user'@'%';

FLUSH PRIVILEGES;

-- Get master status
SHOW MASTER STATUS;
-- Note: File and Position values
```

3. Configure Slave Server

```
-- Edit /etc/mysql/mysql.conf.d/mysqld.cnf
[mysqld]
server-id = 2
relay-log = relay-bin
log-slave-updates = 1
read-only = 1

-- Restart MySQL service
```

4. Start Replication

```
-- On Slave server
CHANGE MASTER TO
  MASTER_HOST = '192.168.1.100',
  MASTER_USER = 'replication_user',
  MASTER_PASSWORD = 'strong_password',
  MASTER_LOG_FILE = 'mysql-bin.000001',
  MASTER_LOG_POS = 154;

START SLAVE;

-- Check replication status
SHOW SLAVE STATUS\G
```

MySQL Group Replication (Master-Master)

```
-- Install Group Replication plugin
INSTALL PLUGIN group_replication SONAME 'group_replication.so';

-- Configure group replication
SET GLOBAL group_replication_group_name = 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa';
SET GLOBAL group_replication_start_on_boot = OFF;
SET GLOBAL group_replication_local_address = '192.168.1.100:33061';
SET GLOBAL group_replication_group_seeds =
'192.168.1.100:33061,192.168.1.101:33061';

-- Start group replication
START GROUP_REPLICATION;
```

PostgreSQL Replication

Setting Up Streaming Replication

1. Configure Primary Server

```
-- Edit postgresql.conf
wal_level = replica
max_wal_senders = 3
max_replication_slots = 3
archive_mode = on
archive_command = 'cp %p /var/lib/postgresql/archive/%f'

-- Edit pg_hba.conf
host replication replication_user 192.168.1.0/24 md5
```

2. Create Replication User

```
-- On Primary server
CREATE USER replication_user WITH REPLICATION PASSWORD 'strong_password';

-- Create replication slot
SELECT pg_create_physical_replication_slot('replica_slot');
```

3. Set Up Standby Server

```
# Stop PostgreSQL on standby
sudo systemctl stop postgresql

# Remove existing data directory
sudo rm -rf /var/lib/postgresql/13/main/*

# Create base backup
sudo -u postgres pg_basebackup -h 192.168.1.100 -D /var/lib/postgresql/13/main -U
replication_user -P -W -R

# Start PostgreSQL
sudo systemctl start postgresql
```

4. Configure Standby

```
-- standby.signal file is created automatically by pg_basebackup -R
-- Edit postgresql.conf on standby
primary_conninfo = 'host=192.168.1.100 port=5432 user=replication_user
password=strong_password'
primary_slot_name = 'replica_slot'
hot_standby = on
```

PostgreSQL Logical Replication

```
-- On Publisher (Primary)
CREATE PUBLICATION my_publication FOR ALL TABLES;

-- On Subscriber (Replica)
CREATE SUBSCRIPTION my_subscription
CONNECTION 'host=192.168.1.100 dbname=mydb user=replication_user
password=strong_password'
PUBLICATION my_publication;

-- Monitor replication
SELECT * FROM pg_stat_replication;
SELECT * FROM pg_stat_subscription;
```

Failover Strategies

Automatic Failover with MySQL

```
# Using MySQL Router for automatic failover
mysqlrouter --bootstrap root@mysql-cluster --user=mysqlrouter

# Start MySQL Router
mysqlrouter &

# Application connects to router port (6446 for read-write, 6447 for read-only)
```

PostgreSQL Automatic Failover

```
# Using Patroni for automatic failover
# Install Patroni
pip install patroni[etcd]

# Configure patroni.yml
scope: postgres-cluster
namespace: /db/
name: node1

restapi:
  listen: 0.0.0.0:8008
  connect_address: 192.168.1.100:8008

etcd:
  hosts: 192.168.1.100:2379,192.168.1.101:2379

bootstrap:
  dcs:
    ttl: 30
    loop_wait: 10
```

```
    retry_timeout: 30
    maximum_lag_on_failover: 1048576

postgresql:
  listen: 0.0.0.0:5432
  connect_address: 192.168.1.100:5432
  data_dir: /var/lib/postgresql/13/main
  bin_dir: /usr/lib/postgresql/13/bin
```

⚖️ Load Balancing

HAProxy Configuration

```
# /etc/haproxy/haproxy.cfg
global
    daemon
    maxconn 4096

defaults
    mode tcp
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

# MySQL Load Balancing
listen mysql-cluster
    bind *:3306
    mode tcp
    balance roundrobin
    option mysql-check user haproxy_check
    server mysql1 192.168.1.100:3306 check
    server mysql2 192.168.1.101:3306 check backup

# PostgreSQL Load Balancing
listen postgres-cluster
    bind *:5432
    mode tcp
    balance roundrobin
    option pgsql-check user haproxy_check
    server postgres1 192.168.1.100:5432 check
    server postgres2 192.168.1.101:5432 check backup
```

Application-Level Load Balancing

```
# Python example with read/write splitting
import random
from sqlalchemy import create_engine
```

```python
class DatabaseRouter:
    def __init__(self):
        self.write_engine = create_engine('mysql://user:pass@master:3306/db')
        self.read_engines = [
            create_engine('mysql://user:pass@slave1:3306/db'),
            create_engine('mysql://user:pass@slave2:3306/db')
        ]

    def get_write_connection(self):
        return self.write_engine.connect()

    def get_read_connection(self):
        return random.choice(self.read_engines).connect()

# Usage
db_router = DatabaseRouter()

# For writes
with db_router.get_write_connection() as conn:
    conn.execute("INSERT INTO users (name) VALUES ('John')")

# For reads
with db_router.get_read_connection() as conn:
    result = conn.execute("SELECT * FROM users")
```

📊 Monitoring Replication

MySQL Monitoring

```sql
-- Check master status
SHOW MASTER STATUS;

-- Check slave status
SHOW SLAVE STATUS\G

-- Monitor replication lag
SELECT
    CASE
        WHEN Slave_lag_seconds IS NULL THEN 'No Delay'
        WHEN Slave_lag_seconds = 0 THEN 'No Delay'
        ELSE CONCAT(Slave_lag_seconds, ' seconds')
    END AS replication_delay
FROM (
    SELECT
        CASE
            WHEN Seconds_Behind_Master IS NULL THEN NULL
            ELSE Seconds_Behind_Master
        END AS Slave_lag_seconds
    FROM INFORMATION_SCHEMA.REPLICA_HOST_STATUS
) AS lag_info;
```

```
-- Check binary log status
SHOW BINARY LOGS;
```

PostgreSQL Monitoring

```
-- Check replication status on primary
SELECT
    client_addr,
    state,
    sent_lsn,
    write_lsn,
    flush_lsn,
    replay_lsn,
    write_lag,
    flush_lag,
    replay_lag
FROM pg_stat_replication;

-- Check replication status on standby
SELECT
    pg_is_in_recovery() AS is_standby,
    pg_last_wal_receive_lsn() AS last_received,
    pg_last_wal_replay_lsn() AS last_replayed,
    pg_last_xact_replay_timestamp() AS last_replay_time;

-- Calculate replication lag
SELECT
    EXTRACT(EPOCH FROM (now() - pg_last_xact_replay_timestamp())) AS lag_seconds;
```

Conflict Resolution

MySQL Conflict Resolution

```
-- Skip replication errors (use carefully)
SET GLOBAL sql_slave_skip_counter = 1;
START SLAVE;

-- Or set slave to ignore specific errors
SET GLOBAL slave_skip_errors = '1062,1053';

-- Check for duplicate key errors
SHOW SLAVE STATUS\G
-- Look for Last_SQL_Error field
```

PostgreSQL Conflict Resolution

```sql
-- Monitor conflicts
SELECT
    datname,
    confl_tablespace,
    confl_lock,
    confl_snapshot,
    confl_bufferpin,
    confl_deadlock
FROM pg_stat_database_conflicts;

-- Handle logical replication conflicts
SELECT
    subname,
    received_lsn,
    latest_end_lsn,
    latest_end_time
FROM pg_stat_subscription;

-- Resolve conflicts by dropping and recreating subscription
DROP SUBSCRIPTION my_subscription;
CREATE SUBSCRIPTION my_subscription
CONNECTION 'host=primary dbname=mydb user=replication_user'
PUBLICATION my_publication;
```

🏗️ Real-World Example: E-commerce High Availability

Architecture Setup

```sql
-- Master database (writes)
CREATE DATABASE ecommerce_master;
USE ecommerce_master;

CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT NOT NULL,
    total_amount DECIMAL(10,2),
    status ENUM('pending', 'processing', 'shipped', 'delivered'),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE order_items (
    id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    product_id INT,
    quantity INT,
    price DECIMAL(10,2),
    FOREIGN KEY (order_id) REFERENCES orders(id)
);
```

```
-- Read replica optimization
CREATE INDEX idx_orders_user_created ON orders(user_id, created_at);
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_order_items_order ON order_items(order_id);
```

Application Implementation

```
class EcommerceDatabase:
    def __init__(self):
        self.master = create_engine('mysql://user:pass@master:3306/ecommerce')
        self.slaves = [
            create_engine('mysql://user:pass@slave1:3306/ecommerce'),
            create_engine('mysql://user:pass@slave2:3306/ecommerce')
        ]

    def create_order(self, user_id, items):
        """Write operation - use master"""
        with self.master.connect() as conn:
            with conn.begin():
                # Insert order
                result = conn.execute(
                    "INSERT INTO orders (user_id, total_amount, status) VALUES
(%s, %s, 'pending')",
                    (user_id, sum(item['price'] * item['quantity'] for item in
items))
                )
                order_id = result.lastrowid

                # Insert order items
                for item in items:
                    conn.execute(
                        "INSERT INTO order_items (order_id, product_id, quantity,
price) VALUES (%s, %s, %s, %s)",
                        (order_id, item['product_id'], item['quantity'],
item['price'])
                    )

                return order_id

    def get_user_orders(self, user_id):
        """Read operation - use slave"""
        slave = random.choice(self.slaves)
        with slave.connect() as conn:
            return conn.execute(
                "SELECT * FROM orders WHERE user_id = %s ORDER BY created_at
DESC",
                (user_id,)
            ).fetchall()

    def get_order_analytics(self):
```

```
        """Heavy read operation - use dedicated analytics slave"""
        with self.slaves[1].connect() as conn:  # Dedicated analytics slave
            return conn.execute("""
                SELECT
                    DATE(created_at) as order_date,
                    COUNT(*) as total_orders,
                    SUM(total_amount) as total_revenue,
                    AVG(total_amount) as avg_order_value
                FROM orders
                WHERE created_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
                GROUP BY DATE(created_at)
                ORDER BY order_date DESC
            """).fetchall()
```

🎯 Interview Questions

Basic Questions

1. **What is the difference between master-slave and master-master replication?**
2. **How do you handle replication lag?**
3. **What are the benefits of read replicas?**

Intermediate Questions

1. **How would you implement automatic failover?**
2. **What strategies would you use for conflict resolution in multi-master setup?**
3. **How do you monitor replication health?**

Advanced Questions

1. **Design a high-availability architecture for a global e-commerce platform**
2. **How would you handle split-brain scenarios in database clustering?**
3. **What are the trade-offs between synchronous and asynchronous replication?**

⚠️ Common Pitfalls

1. **Replication Lag Issues**

```
-- Problem: Reading immediately after write from replica
-- Solution: Use master for critical reads or implement read-after-write
consistency

-- Check lag before critical reads
SELECT Seconds_Behind_Master FROM SHOW SLAVE STATUS;
```

2. **Split-Brain Scenarios**

```
# Problem: Multiple masters in failover
# Solution: Use proper quorum and fencing

# Implement proper fencing in cluster configuration
STONITH_enabled = true
no_quorum_policy = stop
```

3. **Inadequate Monitoring**

```
-- Problem: Not monitoring replication health
-- Solution: Implement comprehensive monitoring

-- Create monitoring queries
CREATE VIEW replication_health AS
SELECT
    'MySQL' as db_type,
    Slave_IO_Running,
    Slave_SQL_Running,
    Seconds_Behind_Master,
    Last_Error
FROM SHOW SLAVE STATUS;
```

🎯 Next Steps

Congratulations! You've mastered database replication and high availability. You now understand:

- ✅ **Master-slave and master-master replication**
- ✅ **Failover strategies and automation**
- ✅ **Load balancing and read scaling**
- ✅ **Monitoring and conflict resolution**
- ✅ **Production-ready high availability architectures**

Continue to → Chapter 19: Database Proxies & Connection Pooling

---

You're now equipped to design and manage highly available database systems that can handle enterprise-scale workloads! 🚀

Chapter 19: Database Proxies & Connection Pooling

🎯 What You'll Learn

Database proxies and connection pooling are essential for scalable applications. This chapter covers proxy databases, connection management, load balancing, query routing, and performance optimization for both MySQL and PostgreSQL.

🎯 Learning Objectives

- **Database Proxies**: Understand proxy architecture and benefits
- **Connection Pooling**: Manage database connections efficiently
- **Query Routing**: Route queries based on type and load
- **Load Balancing**: Distribute database load across multiple servers
- **Caching**: Implement query result caching
- **Monitoring**: Track proxy performance and health
- **Security**: Secure proxy configurations and access control

---

📖 Concept Explanation

What is a Database Proxy?

Database proxy is a middleware layer that sits between applications and database servers. It acts as an intermediary, providing:

- **Connection pooling**: Reuse database connections
- **Load balancing**: Distribute queries across multiple databases
- **Query routing**: Direct queries to appropriate servers
- **Caching**: Store frequently accessed data
- **Security**: Centralized access control and monitoring
- **Failover**: Automatic switching to healthy servers

Proxy vs Direct Connection

Direct Connection:

```
Application → Database Server
```

Proxy Architecture:

```
Application → Database Proxy → Database Server(s)
```

🔧 MySQL Proxy Solutions

MySQL Router

Installation and Setup

```
# Install MySQL Router
sudo apt-get install mysql-router

# Bootstrap router for InnoDB Cluster
```

```
mysqlrouter --bootstrap root@mysql-cluster --user=mysqlrouter

# Start MySQL Router
sudo systemctl start mysqlrouter
sudo systemctl enable mysqlrouter
```

Configuration

```
# /etc/mysqlrouter/mysqlrouter.conf
[DEFAULT]
logging_folder = /var/log/mysqlrouter
runtime_folder = /var/run/mysqlrouter
config_folder = /etc/mysqlrouter

[logger]
level = INFO

# Read-Write connections
[routing:primary]
bind_address = 0.0.0.0
bind_port = 6446
destinations = 192.168.1.100:3306,192.168.1.101:3306
routing_strategy = first-available
mode = read-write

# Read-Only connections
[routing:secondary]
bind_address = 0.0.0.0
bind_port = 6447
destinations = 192.168.1.102:3306,192.168.1.103:3306
routing_strategy = round-robin
mode = read-only

# Connection pooling
[connection_pool]
max_connections = 1000
max_idle_connections = 100
max_idle_time = 3600
```

ProxySQL

Installation

```
# Install ProxySQL
wget https://github.com/sysown/proxysql/releases/download/v2.4.4/proxysql_2.4.4-
ubuntu20_amd64.deb
sudo dpkg -i proxysql_2.4.4-ubuntu20_amd64.deb
```

```
# Start ProxySQL
sudo systemctl start proxysql
sudo systemctl enable proxysql
```

Configuration

```
-- Connect to ProxySQL admin interface
mysql -u admin -padmin -h 127.0.0.1 -P6032

-- Add MySQL servers
INSERT INTO mysql_servers(hostgroup_id, hostname, port, weight) VALUES
(0, '192.168.1.100', 3306, 1000),  -- Master
(1, '192.168.1.101', 3306, 900),   -- Slave 1
(1, '192.168.1.102', 3306, 900);   -- Slave 2

-- Load servers to runtime
LOAD MYSQL SERVERS TO RUNTIME;
SAVE MYSQL SERVERS TO DISK;

-- Add users
INSERT INTO mysql_users(username, password, default_hostgroup) VALUES
('app_user', 'password123', 0);

LOAD MYSQL USERS TO RUNTIME;
SAVE MYSQL USERS TO DISK;

-- Configure query routing rules
INSERT INTO mysql_query_rules(rule_id, active, match_pattern,
destination_hostgroup, apply) VALUES
(1, 1, '^SELECT.*', 1, 1),  -- Route SELECT to read replicas
(2, 1, '^INSERT.*', 0, 1),  -- Route INSERT to master
(3, 1, '^UPDATE.*', 0, 1),  -- Route UPDATE to master
(4, 1, '^DELETE.*', 0, 1);  -- Route DELETE to master

LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL QUERY RULES TO DISK;

-- Configure connection pooling
SET mysql-max_connections=2000;
SET mysql-default_query_timeout=3600000;
SET mysql-have_compress=true;
SET mysql-poll_timeout=2000;

LOAD MYSQL VARIABLES TO RUNTIME;
SAVE MYSQL VARIABLES TO DISK;
```

PostgreSQL Proxy Solutions

PgBouncer

Installation

```
# Install PgBouncer
sudo apt-get install pgbouncer

# Create configuration directory
sudo mkdir -p /etc/pgbouncer
```

Configuration

```
# /etc/pgbouncer/pgbouncer.ini
[databases]
mydb = host=192.168.1.100 port=5432 dbname=production_db
mydb_ro = host=192.168.1.101 port=5432 dbname=production_db

[pgbouncer]
listen_port = 6432
listen_addr = *
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid

# Connection pooling settings
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 25
max_db_connections = 100
max_user_connections = 100

# Performance settings
server_reset_query = DISCARD ALL
server_check_query = SELECT 1
server_check_delay = 30
server_lifetime = 3600
server_idle_timeout = 600

# Security
ignore_startup_parameters = extra_float_digits
```

```
# /etc/pgbouncer/userlist.txt
"app_user" "md5d6a35858d61d85e4a82ab1fb044aba9d"
"read_user" "md5b6d767d2f8ed5d21a44b0e5886680cb9"
```

Start PgBouncer

```
# Start PgBouncer
sudo systemctl start pgbouncer
sudo systemctl enable pgbouncer

# Connect through PgBouncer
psql -h localhost -p 6432 -U app_user mydb
```

PgPool-II

Installation

```
# Install PgPool-II
sudo apt-get install pgpool2
```

Configuration

```
# /etc/pgpool2/pgpool.conf

# Connection settings
listen_addresses = '*'
port = 9999
pcp_listen_addresses = '*'
pcp_port = 9898

# Backend settings
backend_hostname0 = '192.168.1.100'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/var/lib/postgresql/13/main'
backend_flag0 = 'ALLOW_TO_FAILOVER'

backend_hostname1 = '192.168.1.101'
backend_port1 = 5432
backend_weight1 = 1
backend_data_directory1 = '/var/lib/postgresql/13/main'
backend_flag1 = 'ALLOW_TO_FAILOVER'

# Load balancing
load_balance_mode = on
ignore_leading_white_space = on
white_function_list = ''
black_function_list = 'currval,lastval,nextval,setval'

# Connection pooling
connection_cache = on
max_pool = 4
child_life_time = 300
child_max_connections = 0
```

```
connection_life_time = 0
client_idle_limit = 0

# Failover
failover_command = '/etc/pgpool2/failover.sh %d %h %p %D %m %H %M %P %r %R'
failback_command = '/etc/pgpool2/failback.sh %d %h %p %D %m %H %M %P %r %R'
fail_over_on_backend_error = on
search_primary_node_timeout = 10
```

🔄 Connection Pooling Strategies

Pool Modes

1. Session Pooling

```
-- Connection assigned for entire session
-- Best for: Applications with long-running transactions
-- Pros: Simple, maintains session state
-- Cons: Higher memory usage

-- PgBouncer configuration
pool_mode = session
```

2. Transaction Pooling

```
-- Connection returned after each transaction
-- Best for: Web applications with short transactions
-- Pros: Better connection reuse
-- Cons: Cannot use session-level features

-- PgBouncer configuration
pool_mode = transaction
```

3. Statement Pooling

```
-- Connection returned after each statement
-- Best for: Simple query applications
-- Pros: Maximum connection reuse
-- Cons: Very limited functionality

-- PgBouncer configuration
pool_mode = statement
```

Application-Level Connection Pooling

Python with SQLAlchemy

```python
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

# Configure connection pool
engine = create_engine(
    'postgresql://user:password@proxy:6432/mydb',
    poolclass=QueuePool,
    pool_size=20,          # Number of connections to maintain
    max_overflow=30,       # Additional connections when needed
    pool_pre_ping=True,    # Validate connections before use
    pool_recycle=3600,     # Recycle connections after 1 hour
    echo=False
)

class DatabaseManager:
    def __init__(self):
        self.engine = engine

    def execute_query(self, query, params=None):
        with self.engine.connect() as conn:
            return conn.execute(query, params or {})

    def get_pool_status(self):
        pool = self.engine.pool
        return {
            'size': pool.size(),
            'checked_in': pool.checkedin(),
            'checked_out': pool.checkedout(),
            'overflow': pool.overflow(),
            'invalid': pool.invalid()
        }

# Usage
db = DatabaseManager()
result = db.execute_query("SELECT * FROM users WHERE id = :id", {'id': 123})
print(f"Pool status: {db.get_pool_status()}")
```

Node.js with pg-pool

```javascript
const { Pool } = require("pg");

// Configure connection pool
const pool = new Pool({
  host: "proxy-server",
  port: 6432,
```

```
  database: "mydb",
  user: "app_user",
  password: "password",
  max: 20, // Maximum connections
  idleTimeoutMillis: 30000, // Close idle connections after 30s
  connectionTimeoutMillis: 2000, // Timeout when connecting
  maxUses: 7500, // Close connection after 7500 uses
});

class DatabaseManager {
  async executeQuery(query, params = []) {
    const client = await pool.connect();
    try {
      const result = await client.query(query, params);
      return result.rows;
    } finally {
      client.release();
    }
  }

  getPoolStatus() {
    return {
      totalCount: pool.totalCount,
      idleCount: pool.idleCount,
      waitingCount: pool.waitingCount,
    };
  }
}

// Usage
const db = new DatabaseManager();

async function getUser(id) {
  const users = await db.executeQuery("SELECT * FROM users WHERE id = $1", [
    id,
  ]);
  console.log("Pool status:", db.getPoolStatus());
  return users[0];
}
```

🎯 Query Routing and Load Balancing

Read/Write Splitting

ProxySQL Query Routing

```
-- Route based on query type
INSERT INTO mysql_query_rules(rule_id, active, match_pattern,
destination_hostgroup, apply) VALUES
(10, 1, '^SELECT.*FOR UPDATE', 0, 1),     -- SELECT FOR UPDATE to master
```

```
(20, 1, '^SELECT.*', 1, 1),              -- Regular SELECT to slaves
(30, 1, '^INSERT.*', 0, 1),              -- INSERT to master
(40, 1, '^UPDATE.*', 0, 1),              -- UPDATE to master
(50, 1, '^DELETE.*', 0, 1),              -- DELETE to master
(60, 1, '^REPLACE.*', 0, 1),             -- REPLACE to master
(70, 1, '^CALL.*', 0, 1);                -- Stored procedures to master

-- Route based on user
INSERT INTO mysql_query_rules(rule_id, active, username, destination_hostgroup,
apply) VALUES
(100, 1, 'analytics_user', 2, 1);        -- Analytics user to dedicated server

-- Route based on schema
INSERT INTO mysql_query_rules(rule_id, active, schemaname, destination_hostgroup,
apply) VALUES
(200, 1, 'reporting_db', 2, 1);          -- Reporting queries to analytics server

LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL QUERY RULES TO DISK;
```

Application-Level Routing

```
class SmartDatabaseRouter:
    def __init__(self):
        self.write_pool = create_engine('mysql://user:pass@master-proxy:6446/db')
        self.read_pool = create_engine('mysql://user:pass@slave-proxy:6447/db')
        self.analytics_pool = create_engine('mysql://user:pass@analytics-
proxy:6448/db')

    def route_query(self, query, query_type='auto'):
        if query_type == 'auto':
            query_type = self._detect_query_type(query)

        if query_type == 'write':
            return self.write_pool
        elif query_type == 'analytics':
            return self.analytics_pool
        else:
            return self.read_pool

    def _detect_query_type(self, query):
        query_upper = query.upper().strip()

        if any(query_upper.startswith(cmd) for cmd in ['INSERT', 'UPDATE',
'DELETE', 'REPLACE']):
            return 'write'
        elif 'FOR UPDATE' in query_upper or 'LOCK IN SHARE MODE' in query_upper:
            return 'write'
        elif any(func in query_upper for func in ['COUNT(*)', 'SUM(', 'AVG(',
'GROUP BY']):
            return 'analytics'
```

```
        else:
            return 'read'

    def execute(self, query, params=None):
        engine = self.route_query(query)
        with engine.connect() as conn:
            return conn.execute(query, params or {})

# Usage
router = SmartDatabaseRouter()

# Automatically routed to read replica
users = router.execute("SELECT * FROM users WHERE status = 'active'")

# Automatically routed to master
router.execute("INSERT INTO users (name, email) VALUES ('John',
'john@example.com')")

# Automatically routed to analytics server
stats = router.execute("SELECT COUNT(*), AVG(order_total) FROM orders GROUP BY
DATE(created_at)")
```

🚀 Caching Strategies

Query Result Caching

Redis Integration with Proxy

```
import redis
import json
import hashlib
from datetime import timedelta

class CachingDatabaseProxy:
    def __init__(self):
        self.db = create_engine('postgresql://user:pass@pgbouncer:6432/db')
        self.cache = redis.Redis(host='redis-server', port=6379, db=0)
        self.default_ttl = 300  # 5 minutes

    def _generate_cache_key(self, query, params):
        """Generate unique cache key for query and parameters"""
        query_hash = hashlib.md5(f"{query}{params}".encode()).hexdigest()
        return f"query_cache:{query_hash}"

    def _is_cacheable_query(self, query):
        """Determine if query results should be cached"""
        query_upper = query.upper().strip()

        # Only cache SELECT queries
        if not query_upper.startswith('SELECT'):
```

```
            return False

        # Don't cache queries with functions that return current time/data
        non_cacheable = ['NOW()', 'CURRENT_TIMESTAMP', 'RANDOM()', 'UUID()']
        return not any(func in query_upper for func in non_cacheable)

    def execute_with_cache(self, query, params=None, ttl=None):
        params = params or {}
        ttl = ttl or self.default_ttl

        if not self._is_cacheable_query(query):
            return self._execute_direct(query, params)

        cache_key = self._generate_cache_key(query, params)

        # Try to get from cache
        cached_result = self.cache.get(cache_key)
        if cached_result:
            return json.loads(cached_result)

        # Execute query and cache result
        result = self._execute_direct(query, params)

        # Cache the result
        self.cache.setex(
            cache_key,
            ttl,
            json.dumps(result, default=str)
        )

        return result

    def _execute_direct(self, query, params):
        with self.db.connect() as conn:
            result = conn.execute(query, params)
            return [dict(row) for row in result]

    def invalidate_cache_pattern(self, pattern):
        """Invalidate cache entries matching pattern"""
        keys = self.cache.keys(f"query_cache:*{pattern}*")
        if keys:
            self.cache.delete(*keys)

    def get_cache_stats(self):
        """Get cache statistics"""
        info = self.cache.info()
        return {
            'used_memory': info['used_memory_human'],
            'keyspace_hits': info['keyspace_hits'],
            'keyspace_misses': info['keyspace_misses'],
            'hit_rate': info['keyspace_hits'] / (info['keyspace_hits'] +
info['keyspace_misses']) * 100
        }
```

```
# Usage
caching_proxy = CachingDatabaseProxy()

# This will be cached
users = caching_proxy.execute_with_cache(
    "SELECT * FROM users WHERE department = :dept",
    {'dept': 'engineering'},
    ttl=600  # Cache for 10 minutes
)

# Invalidate cache when data changes
caching_proxy.invalidate_cache_pattern('users')

print(f"Cache stats: {caching_proxy.get_cache_stats()}")
```

ProxySQL Query Caching

```
-- Enable query caching in ProxySQL
SET mysql-query_cache_size_MB=256;
SET mysql-query_cache_stores_empty_result=true;

-- Configure caching rules
INSERT INTO mysql_query_rules(
    rule_id, active, match_pattern, cache_ttl, apply
) VALUES
(1000, 1, '^SELECT.*FROM products.*', 300000, 1),    -- Cache product queries for
5 minutes
(1001, 1, '^SELECT.*FROM categories.*', 600000, 1),   -- Cache category queries
for 10 minutes
(1002, 1, '^SELECT COUNT\(\*\).*', 60000, 1);         -- Cache count queries for 1
minute

LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL QUERY RULES TO DISK;

-- Monitor cache performance
SELECT * FROM stats_mysql_query_cache;
```

📊 Monitoring and Performance

ProxySQL Monitoring

```
-- Connection statistics
SELECT * FROM stats_mysql_connection_pool;

-- Query statistics
SELECT
    hostgroup,
```

```
    schemaname,
    username,
    digest_text,
    count_star,
    sum_time,
    avg_time
FROM stats_mysql_query_digest
ORDER BY sum_time DESC
LIMIT 10;

-- Backend server health
SELECT
    hostname,
    port,
    status,
    ConnUsed,
    ConnFree,
    ConnOK,
    ConnERR,
    Queries,
    Bytes_data_sent,
    Bytes_data_recv
FROM stats_mysql_servers;

-- Query rules statistics
SELECT
    rule_id,
    hits,
    match_pattern,
    destination_hostgroup
FROM stats_mysql_query_rules
ORDER BY hits DESC;
```

PgBouncer Monitoring

```
-- Connect to PgBouncer admin
psql -h localhost -p 6432 -U pgbouncer pgbouncer

-- Show pool statistics
SHOW POOLS;

-- Show client connections
SHOW CLIENTS;

-- Show server connections
SHOW SERVERS;

-- Show configuration
SHOW CONFIG;
```

```
-- Show statistics
SHOW STATS;
```

Custom Monitoring Script

```
import psutil
import time
from datetime import datetime

class ProxyMonitor:
    def __init__(self, proxy_type='pgbouncer'):
        self.proxy_type = proxy_type
        self.metrics = []

    def collect_system_metrics(self):
        """Collect system-level metrics"""
        return {
            'timestamp': datetime.now(),
            'cpu_percent': psutil.cpu_percent(),
            'memory_percent': psutil.virtual_memory().percent,
            'disk_io': psutil.disk_io_counters()._asdict(),
            'network_io': psutil.net_io_counters()._asdict()
        }

    def collect_pgbouncer_metrics(self):
        """Collect PgBouncer-specific metrics"""
        import psycopg2

        conn = psycopg2.connect(
            host='localhost',
            port=6432,
            user='pgbouncer',
            database='pgbouncer'
        )

        with conn.cursor() as cur:
            # Pool statistics
            cur.execute("SHOW POOLS")
            pools = cur.fetchall()

            # Statistics
            cur.execute("SHOW STATS")
            stats = cur.fetchall()

        conn.close()

        return {
            'pools': pools,
            'stats': stats
        }
```

```
    def collect_proxysql_metrics(self):
        """Collect ProxySQL-specific metrics"""
        import mysql.connector

        conn = mysql.connector.connect(
            host='127.0.0.1',
            port=6032,
            user='admin',
            password='admin'
        )

        with conn.cursor() as cur:
            # Connection pool stats
            cur.execute("SELECT * FROM stats_mysql_connection_pool")
            pool_stats = cur.fetchall()

            # Query digest stats
            cur.execute("""
                SELECT hostgroup, count_star, sum_time, avg_time
                FROM stats_mysql_query_digest
                ORDER BY sum_time DESC LIMIT 10
            """)
            query_stats = cur.fetchall()

        conn.close()

        return {
            'pool_stats': pool_stats,
            'query_stats': query_stats
        }

    def monitor_continuously(self, interval=60):
        """Continuously monitor proxy performance"""
        while True:
            try:
                metrics = {
                    'system': self.collect_system_metrics()
                }

                if self.proxy_type == 'pgbouncer':
                    metrics['proxy'] = self.collect_pgbouncer_metrics()
                elif self.proxy_type == 'proxysql':
                    metrics['proxy'] = self.collect_proxysql_metrics()

                self.metrics.append(metrics)

                # Keep only last 1000 metrics
                if len(self.metrics) > 1000:
                    self.metrics = self.metrics[-1000:]

                print(f"Collected metrics at {metrics['system']['timestamp']}")

            except Exception as e:
                print(f"Error collecting metrics: {e}")
```

```
            time.sleep(interval)

# Usage
monitor = ProxyMonitor('pgbouncer')
monitor.monitor_continuously(30)  # Monitor every 30 seconds
```

Security Considerations

Secure Proxy Configuration

SSL/TLS Configuration

```
# PgBouncer SSL configuration
[pgbouncer]
server_tls_sslmode = require
server_tls_ca_file = /etc/ssl/certs/ca-certificates.crt
server_tls_key_file = /etc/pgbouncer/server.key
server_tls_cert_file = /etc/pgbouncer/server.crt

client_tls_sslmode = require
client_tls_ca_file = /etc/ssl/certs/ca-certificates.crt
client_tls_key_file = /etc/pgbouncer/client.key
client_tls_cert_file = /etc/pgbouncer/client.crt
```

```
-- ProxySQL SSL configuration
UPDATE global_variables SET variable_value='true' WHERE variable_name='mysql-
have_ssl';
UPDATE global_variables SET variable_value='/etc/proxysql/ssl/server-cert.pem'
WHERE variable_name='mysql-ssl_cert';
UPDATE global_variables SET variable_value='/etc/proxysql/ssl/server-key.pem'
WHERE variable_name='mysql-ssl_key';
UPDATE global_variables SET variable_value='/etc/proxysql/ssl/ca-cert.pem' WHERE
variable_name='mysql-ssl_ca';

LOAD MYSQL VARIABLES TO RUNTIME;
SAVE MYSQL VARIABLES TO DISK;
```

Access Control

```
-- ProxySQL user access control
INSERT INTO mysql_users(username, password, default_hostgroup, max_connections)
VALUES
('app_user', 'hashed_password', 0, 100),
('read_only_user', 'hashed_password', 1, 50),
```

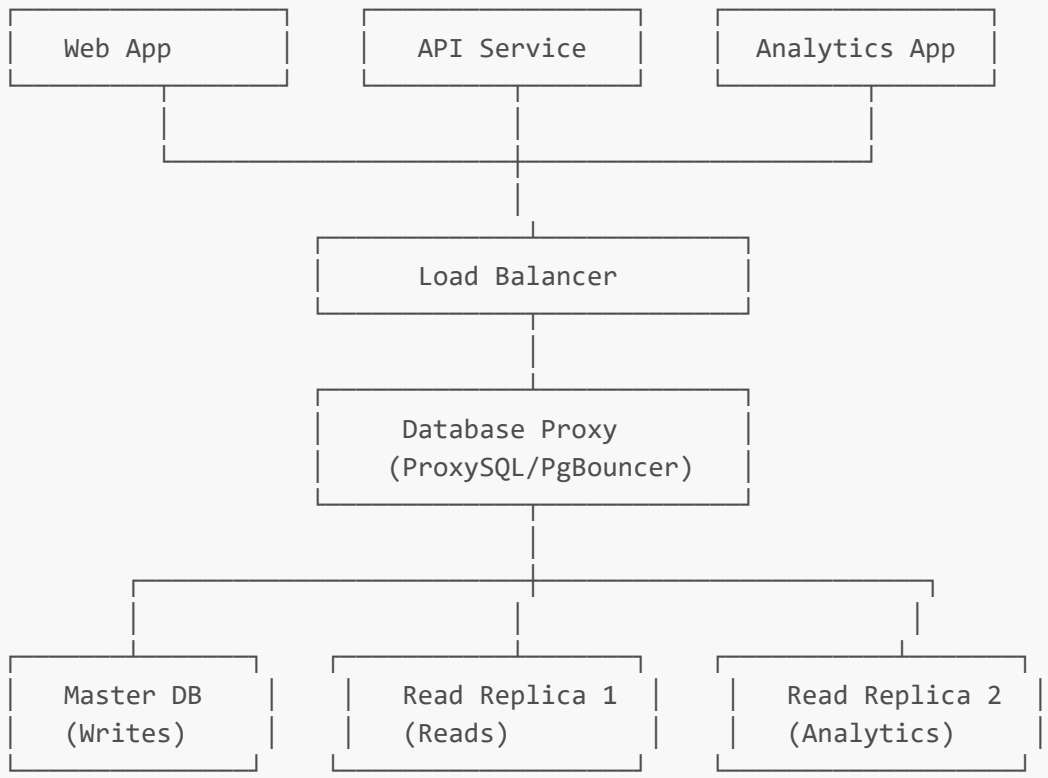
```
('analytics_user', 'hashed_password', 2, 10);

-- Restrict access by source IP
INSERT INTO mysql_query_rules(rule_id, active, client_addr, apply) VALUES
(5000, 1, '192.168.1.0/24', 1);  -- Only allow from internal network

LOAD MYSQL USERS TO RUNTIME;
LOAD MYSQL QUERY RULES TO RUNTIME;
```

🏗️ Real-World Example: E-commerce Proxy Architecture

Architecture Design



Implementation

```
class EcommerceProxyManager:
    def __init__(self):
        # Different connection pools for different workloads
        self.write_pool = self._create_pool('master-proxy:6446', pool_size=50)
        self.read_pool = self._create_pool('read-proxy:6447', pool_size=100)
        self.analytics_pool = self._create_pool('analytics-proxy:6448',
pool_size=20)

        # Cache for frequently accessed data
        self.cache = redis.Redis(host='redis-cluster', port=6379)
```

```
    def _create_pool(self, host, pool_size):
        return create_engine(
            f'mysql://app_user:password@{host}/ecommerce',
            pool_size=pool_size,
            max_overflow=pool_size // 2,
            pool_pre_ping=True,
            pool_recycle=3600
        )

    def create_order(self, user_id, items):
        """Write operation - use master"""
        with self.write_pool.connect() as conn:
            with conn.begin():
                # Insert order
                result = conn.execute(
                    "INSERT INTO orders (user_id, total_amount, status) VALUES
(%s, %s, 'pending')",
                    (user_id, sum(item['price'] * item['quantity'] for item in
items))
                )
                order_id = result.lastrowid

                # Insert order items
                for item in items:
                    conn.execute(
                        "INSERT INTO order_items (order_id, product_id, quantity,
price) VALUES (%s, %s, %s, %s)",
                        (order_id, item['product_id'], item['quantity'],
item['price'])
                    )

                # Invalidate relevant caches
                self.cache.delete(f"user_orders:{user_id}")
                self.cache.delete("order_stats")

                return order_id

    def get_user_orders(self, user_id):
        """Read operation with caching"""
        cache_key = f"user_orders:{user_id}"

        # Try cache first
        cached_orders = self.cache.get(cache_key)
        if cached_orders:
            return json.loads(cached_orders)

        # Query from read replica
        with self.read_pool.connect() as conn:
            orders = conn.execute(
                "SELECT * FROM orders WHERE user_id = %s ORDER BY created_at DESC
LIMIT 50",
                (user_id,)
            ).fetchall()
```

```
            orders_list = [dict(order) for order in orders]

            # Cache for 5 minutes
            self.cache.setex(cache_key, 300, json.dumps(orders_list, default=str))

            return orders_list

    def get_sales_analytics(self, start_date, end_date):
        """Analytics operation - use dedicated analytics server"""
        with self.analytics_pool.connect() as conn:
            return conn.execute("""
                SELECT
                    DATE(created_at) as order_date,
                    COUNT(*) as total_orders,
                    SUM(total_amount) as total_revenue,
                    AVG(total_amount) as avg_order_value,
                    COUNT(DISTINCT user_id) as unique_customers
                FROM orders
                WHERE created_at BETWEEN %s AND %s
                GROUP BY DATE(created_at)
                ORDER BY order_date DESC
            """, (start_date, end_date)).fetchall()

    def get_connection_stats(self):
        """Monitor connection pool health"""
        return {
            'write_pool': {
                'size': self.write_pool.pool.size(),
                'checked_out': self.write_pool.pool.checkedout(),
                'overflow': self.write_pool.pool.overflow()
            },
            'read_pool': {
                'size': self.read_pool.pool.size(),
                'checked_out': self.read_pool.pool.checkedout(),
                'overflow': self.read_pool.pool.overflow()
            },
            'analytics_pool': {
                'size': self.analytics_pool.pool.size(),
                'checked_out': self.analytics_pool.pool.checkedout(),
                'overflow': self.analytics_pool.pool.overflow()
            }
        }

# Usage
ecommerce_proxy = EcommerceProxyManager()

# Create order (routed to master)
order_id = ecommerce_proxy.create_order(123, [
    {'product_id': 1, 'quantity': 2, 'price': 29.99},
    {'product_id': 2, 'quantity': 1, 'price': 49.99}
])

# Get user orders (routed to read replica, cached)
orders = ecommerce_proxy.get_user_orders(123)
```

```
# Get analytics (routed to analytics server)
stats = ecommerce_proxy.get_sales_analytics('2024-01-01', '2024-01-31')

# Monitor connection health
print(f"Connection stats: {ecommerce_proxy.get_connection_stats()}")
```

Interview Questions

Basic Questions

1. **What is a database proxy and why would you use one?**
2. **Explain the difference between session, transaction, and statement pooling.**
3. **How does connection pooling improve application performance?**

Intermediate Questions

1. **How would you implement read/write splitting in a database proxy?**
2. **What are the trade-offs between application-level and proxy-level connection pooling?**
3. **How do you handle failover in a proxy configuration?**

Advanced Questions

1. **Design a database proxy architecture for a global e-commerce platform**
2. **How would you implement query result caching in a database proxy?**
3. **What strategies would you use to monitor and optimize proxy performance?**

⚠ Common Pitfalls

1. **Inadequate Pool Sizing**

```
# Problem: Pool too small causes connection queuing
pool_size = 5  # Too small for high-traffic app

# Solution: Size based on concurrent users and query duration
pool_size = min(100, max(10, concurrent_users * avg_query_time_seconds))
```

2. **Ignoring Connection Leaks**

```
# Problem: Not properly releasing connections
def bad_query():
    conn = pool.get_connection()
    result = conn.execute("SELECT * FROM users")
    # Connection never released!
    return result
```

```
# Solution: Always use context managers
def good_query():
    with pool.get_connection() as conn:
        return conn.execute("SELECT * FROM users")
```

3. **Improper Query Routing**

```
-- Problem: Routing read queries to master unnecessarily
SELECT * FROM users WHERE id = 123;  -- Could go to replica

-- Solution: Implement proper routing rules
-- Route only writes and critical reads to master
```

🎯 Next Steps

Congratulations! You've mastered database proxies and connection pooling. You now understand:

- ✅ **Database proxy architecture and benefits**
- ✅ **Connection pooling strategies and optimization**
- ✅ **Query routing and load balancing**
- ✅ **Caching and performance optimization**
- ✅ **Monitoring and security best practices**

Continue to → Chapter 20: Partitioning & Sharding

You're now equipped to design and implement scalable database proxy architectures that can handle enterprise-scale workloads! 🚀

Chapter 20: Partitioning & Sharding

🗃️ What You'll Learn

Partitioning and sharding are essential techniques for scaling databases beyond single-server limitations. This chapter covers horizontal and vertical partitioning, sharding strategies, data distribution, and managing distributed databases for both MySQL and PostgreSQL.

🎯 Learning Objectives

- **Database Partitioning**: Understand table partitioning strategies
- **Horizontal Sharding**: Distribute data across multiple servers
- **Vertical Sharding**: Split tables by columns or functionality
- **Shard Key Selection**: Choose optimal data distribution keys
- **Cross-Shard Queries**: Handle queries spanning multiple shards
- **Rebalancing**: Redistribute data as requirements change

- **Consistency**: Maintain data integrity across shards

📖 Concept Explanation

What is Partitioning?

Partitioning divides a large table into smaller, more manageable pieces called partitions, while keeping them within the same database instance.

What is Sharding?

Sharding distributes data across multiple database servers (shards), where each shard contains a subset of the total data.

Partitioning vs Sharding

Aspect	Partitioning	Sharding
Scope	Single database instance	Multiple database servers
Complexity	Lower	Higher
Scalability	Limited by single server	Unlimited horizontal scaling
Queries	Transparent to application	May require application changes
Consistency	ACID guaranteed	Eventual consistency challenges

🔧 MySQL Partitioning

Range Partitioning

```
-- Partition orders by date range
CREATE TABLE orders (
    id INT AUTO_INCREMENT,
    user_id INT NOT NULL,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10,2),
    status ENUM('pending', 'processing', 'shipped', 'delivered'),
    PRIMARY KEY (id, order_date)
) PARTITION BY RANGE (YEAR(order_date)) (
    PARTITION p2020 VALUES LESS THAN (2021),
    PARTITION p2021 VALUES LESS THAN (2022),
    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
);

-- Add new partition for 2025
ALTER TABLE orders ADD PARTITION (
```

```
    PARTITION p2025 VALUES LESS THAN (2026)
);

-- Drop old partition
ALTER TABLE orders DROP PARTITION p2020;
```

Hash Partitioning

```
-- Partition users by hash of user_id
CREATE TABLE users (
    id INT AUTO_INCREMENT,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id)
) PARTITION BY HASH(id) PARTITIONS 8;

-- Partition by custom hash function
CREATE TABLE user_sessions (
    session_id VARCHAR(64),
    user_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    expires_at TIMESTAMP,
    PRIMARY KEY (session_id, user_id)
) PARTITION BY HASH(CRC32(session_id)) PARTITIONS 16;
```

List Partitioning

```
-- Partition by specific values (regions)
CREATE TABLE sales (
    id INT AUTO_INCREMENT,
    region VARCHAR(20) NOT NULL,
    product_id INT,
    amount DECIMAL(10,2),
    sale_date DATE,
    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', 'central_east'),
    PARTITION p_west VALUES IN ('west', 'central_west'),
    PARTITION p_international VALUES IN ('europe', 'asia', 'africa')
);
```

Subpartitioning

```sql
-- Combine range and hash partitioning
CREATE TABLE order_details (
    id INT AUTO_INCREMENT,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT,
    price DECIMAL(10,2),
    order_date DATE NOT NULL,
    PRIMARY KEY (id, order_date, order_id)
) PARTITION BY RANGE (YEAR(order_date))
SUBPARTITION BY HASH(order_id)
SUBPARTITIONS 4 (
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025),
    PARTITION p_future VALUES LESS THAN MAXVALUE
);
```

PostgreSQL Partitioning

Declarative Partitioning (PostgreSQL 10+)

Range Partitioning

```sql
-- Create parent table
CREATE TABLE orders (
    id SERIAL,
    user_id INTEGER NOT NULL,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10,2),
    status VARCHAR(20)
) PARTITION BY RANGE (order_date);

-- Create partitions
CREATE TABLE orders_2023 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE orders_2024 PARTITION OF orders
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE orders_default PARTITION OF orders DEFAULT;

-- Create indexes on partitions
CREATE INDEX idx_orders_2023_user_date ON orders_2023 (user_id, order_date);
CREATE INDEX idx_orders_2024_user_date ON orders_2024 (user_id, order_date);
```

Hash Partitioning

```sql
-- Create hash-partitioned table
CREATE TABLE user_activities (
    id SERIAL,
    user_id INTEGER NOT NULL,
    activity_type VARCHAR(50),
    activity_data JSONB,
    created_at TIMESTAMP DEFAULT NOW()
) PARTITION BY HASH (user_id);

-- Create hash partitions
CREATE TABLE user_activities_0 PARTITION OF user_activities
    FOR VALUES WITH (modulus 4, remainder 0);

CREATE TABLE user_activities_1 PARTITION OF user_activities
    FOR VALUES WITH (modulus 4, remainder 1);

CREATE TABLE user_activities_2 PARTITION OF user_activities
    FOR VALUES WITH (modulus 4, remainder 2);

CREATE TABLE user_activities_3 PARTITION OF user_activities
    FOR VALUES WITH (modulus 4, remainder 3);
```

List Partitioning

```sql
-- Create list-partitioned table
CREATE TABLE products (
    id SERIAL,
    name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2),
    created_at TIMESTAMP DEFAULT NOW()
) PARTITION BY LIST (category);

-- Create list partitions
CREATE TABLE products_electronics PARTITION OF products
    FOR VALUES IN ('electronics', 'computers', 'phones');

CREATE TABLE products_clothing PARTITION OF products
    FOR VALUES IN ('clothing', 'shoes', 'accessories');

CREATE TABLE products_books PARTITION OF products
    FOR VALUES IN ('books', 'ebooks', 'audiobooks');

CREATE TABLE products_other PARTITION OF products DEFAULT;
```

Partition Pruning and Constraint Exclusion

```sql
-- Enable constraint exclusion
SET constraint_exclusion = partition;

-- Query that benefits from partition pruning
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM orders
WHERE order_date BETWEEN '2024-01-01' AND '2024-03-31';

-- Check partition pruning
SELECT
    schemaname,
    tablename,
    attname,
    n_distinct,
    correlation
FROM pg_stats
WHERE tablename LIKE 'orders_%';
```

🌐 Horizontal Sharding Strategies

Shard Key Selection

1. User ID Sharding

```python
class UserShardRouter:
    def __init__(self, shard_count=4):
        self.shard_count = shard_count
        self.shards = {
            0: create_engine('mysql://user:pass@shard0:3306/app'),
            1: create_engine('mysql://user:pass@shard1:3306/app'),
            2: create_engine('mysql://user:pass@shard2:3306/app'),
            3: create_engine('mysql://user:pass@shard3:3306/app')
        }

    def get_shard(self, user_id):
        """Route based on user_id hash"""
        shard_id = hash(str(user_id)) % self.shard_count
        return self.shards[shard_id]

    def create_user(self, user_data):
        """Create user in appropriate shard"""
        user_id = user_data['id']
        shard = self.get_shard(user_id)

        with shard.connect() as conn:
            conn.execute("""
                INSERT INTO users (id, username, email, created_at)
                VALUES (%(id)s, %(username)s, %(email)s, %(created_at)s)
            """, user_data)
```

```
    def get_user(self, user_id):
        """Get user from appropriate shard"""
        shard = self.get_shard(user_id)

        with shard.connect() as conn:
            result = conn.execute(
                "SELECT * FROM users WHERE id = %s", (user_id,)
            )
            return result.fetchone()

    def get_user_orders(self, user_id):
        """Get orders for user from same shard"""
        shard = self.get_shard(user_id)

        with shard.connect() as conn:
            result = conn.execute("""
                SELECT o.*, oi.product_id, oi.quantity, oi.price
                FROM orders o
                JOIN order_items oi ON o.id = oi.order_id
                WHERE o.user_id = %s
                ORDER BY o.created_at DESC
            """, (user_id,))
            return result.fetchall()

# Usage
router = UserShardRouter()

# Create user (automatically routed to correct shard)
router.create_user({
    'id': 12345,
    'username': 'john_doe',
    'email': 'john@example.com',
    'created_at': datetime.now()
})

# Get user (automatically routed to correct shard)
user = router.get_user(12345)
orders = router.get_user_orders(12345)
```

2. Geographic Sharding

```
class GeographicShardRouter:
    def __init__(self):
        self.region_shards = {
            'us_east': create_engine('mysql://user:pass@us-east-shard:3306/app'),
            'us_west': create_engine('mysql://user:pass@us-west-shard:3306/app'),
            'europe': create_engine('mysql://user:pass@eu-shard:3306/app'),
            'asia': create_engine('mysql://user:pass@asia-shard:3306/app')
        }
```

```
        self.country_to_region = {
            'US': 'us_east', 'CA': 'us_east',
            'GB': 'europe', 'DE': 'europe', 'FR': 'europe',
            'JP': 'asia', 'CN': 'asia', 'IN': 'asia'
        }

    def get_shard_by_country(self, country_code):
        """Route based on country"""
        region = self.country_to_region.get(country_code, 'us_east')
        return self.region_shards[region]

    def create_order(self, order_data):
        """Create order in geographically appropriate shard"""
        shard = self.get_shard_by_country(order_data['country'])

        with shard.connect() as conn:
            with conn.begin():
                # Insert order
                result = conn.execute("""
                    INSERT INTO orders (user_id, country, total_amount,
created_at)
                    VALUES (%(user_id)s, %(country)s, %(total_amount)s, %
(created_at)s)
                """, order_data)

                order_id = result.lastrowid

                # Insert order items
                for item in order_data['items']:
                    item['order_id'] = order_id
                    conn.execute("""
                        INSERT INTO order_items (order_id, product_id, quantity,
price)
                        VALUES (%(order_id)s, %(product_id)s, %(quantity)s, %
(price)s)
                    """, item)

                return order_id

    def get_regional_analytics(self, region, start_date, end_date):
        """Get analytics for specific region"""
        shard = self.region_shards[region]

        with shard.connect() as conn:
            return conn.execute("""
                SELECT
                    DATE(created_at) as order_date,
                    COUNT(*) as total_orders,
                    SUM(total_amount) as total_revenue,
                    AVG(total_amount) as avg_order_value
                FROM orders
                WHERE created_at BETWEEN %s AND %s
                GROUP BY DATE(created_at)
                ORDER BY order_date
```

```
            """, (start_date, end_date)).fetchall()

# Usage
geo_router = GeographicShardRouter()

# Create order (routed to appropriate geographic shard)
order_id = geo_router.create_order({
    'user_id': 12345,
    'country': 'US',
    'total_amount': 99.99,
    'created_at': datetime.now(),
    'items': [
        {'product_id': 1, 'quantity': 2, 'price': 29.99},
        {'product_id': 2, 'quantity': 1, 'price': 39.99}
    ]
})

# Get regional analytics
us_stats = geo_router.get_regional_analytics('us_east', '2024-01-01', '2024-01-
31')
```

3. Time-Based Sharding

```
class TimeBasedShardRouter:
    def __init__(self):
        self.time_shards = {
            '2023': create_engine('mysql://user:pass@shard-2023:3306/app'),
            '2024': create_engine('mysql://user:pass@shard-2024:3306/app'),
            '2025': create_engine('mysql://user:pass@shard-2025:3306/app')
        }

    def get_shard_by_date(self, date_obj):
        """Route based on year"""
        year = str(date_obj.year)
        return self.time_shards.get(year, self.time_shards['2024'])  # Default to
current

    def create_log_entry(self, log_data):
        """Create log entry in time-appropriate shard"""
        shard = self.get_shard_by_date(log_data['timestamp'])

        with shard.connect() as conn:
            conn.execute("""
                INSERT INTO activity_logs (user_id, action, details, timestamp)
                VALUES (%(user_id)s, %(action)s, %(details)s, %(timestamp)s)
            """, log_data)

    def get_logs_for_period(self, start_date, end_date):
        """Get logs across multiple time shards"""
        all_logs = []
```

```
        # Determine which shards to query
        years = set()
        current_date = start_date
        while current_date <= end_date:
            years.add(str(current_date.year))
            current_date = current_date.replace(year=current_date.year + 1,
month=1, day=1)

        # Query each relevant shard
        for year in years:
            if year in self.time_shards:
                shard = self.time_shards[year]
                with shard.connect() as conn:
                    logs = conn.execute("""
                        SELECT * FROM activity_logs
                        WHERE timestamp BETWEEN %s AND %s
                        ORDER BY timestamp
                    """, (start_date, end_date)).fetchall()
                    all_logs.extend(logs)

        # Sort combined results
        return sorted(all_logs, key=lambda x: x['timestamp'])

# Usage
time_router = TimeBasedShardRouter()

# Create log entry (routed to appropriate time shard)
time_router.create_log_entry({
    'user_id': 12345,
    'action': 'login',
    'details': {'ip': '192.168.1.100', 'user_agent': 'Chrome/91.0'},
    'timestamp': datetime.now()
})

# Get logs across time period (may query multiple shards)
logs = time_router.get_logs_for_period(
    datetime(2023, 12, 1),
    datetime(2024, 2, 1)
)
```

🔄 Cross-Shard Operations

Distributed Queries

```
class DistributedQueryManager:
    def __init__(self, shard_router):
        self.router = shard_router

    def aggregate_across_shards(self, query, params=None):
        """Execute query across all shards and aggregate results"""
```

```
        all_results = []

        for shard_id, shard in self.router.shards.items():
            try:
                with shard.connect() as conn:
                    result = conn.execute(query, params or {})
                    shard_results = result.fetchall()

                    # Add shard_id to each result
                    for row in shard_results:
                        row_dict = dict(row)
                        row_dict['_shard_id'] = shard_id
                        all_results.append(row_dict)

            except Exception as e:
                print(f"Error querying shard {shard_id}: {e}")
                continue

        return all_results

    def get_global_user_count(self):
        """Get total user count across all shards"""
        results = self.aggregate_across_shards("SELECT COUNT(*) as user_count FROM
users")
        return sum(row['user_count'] for row in results)

    def get_top_users_by_orders(self, limit=10):
        """Get top users by order count across all shards"""
        query = """
            SELECT
                user_id,
                COUNT(*) as order_count,
                SUM(total_amount) as total_spent
            FROM orders
            GROUP BY user_id
            ORDER BY order_count DESC
            LIMIT %s
        """

        all_results = self.aggregate_across_shards(query, (limit * 2,))  # Get
more from each shard

        # Aggregate results across shards
        user_aggregates = {}
        for row in all_results:
            user_id = row['user_id']
            if user_id not in user_aggregates:
                user_aggregates[user_id] = {
                    'user_id': user_id,
                    'order_count': 0,
                    'total_spent': 0
                }

            user_aggregates[user_id]['order_count'] += row['order_count']
```

```
            user_aggregates[user_id]['total_spent'] += row['total_spent']

        # Sort and return top users
        sorted_users = sorted(
            user_aggregates.values(),
            key=lambda x: x['order_count'],
            reverse=True
        )

        return sorted_users[:limit]

    def search_users_by_email(self, email_pattern):
        """Search for users by email pattern across all shards"""
        query = "SELECT * FROM users WHERE email LIKE %s"
        return self.aggregate_across_shards(query, (f"%{email_pattern}%",))

# Usage
dist_query = DistributedQueryManager(router)

# Get global statistics
total_users = dist_query.get_global_user_count()
top_users = dist_query.get_top_users_by_orders(10)

# Search across shards
users = dist_query.search_users_by_email("gmail.com")
```

Distributed Transactions

```
from contextlib import contextmanager

class DistributedTransactionManager:
    def __init__(self, shard_router):
        self.router = shard_router

    @contextmanager
    def distributed_transaction(self, shard_ids):
        """Two-phase commit across multiple shards"""
        connections = {}
        transactions = {}

        try:
            # Phase 1: Prepare all transactions
            for shard_id in shard_ids:
                shard = self.router.shards[shard_id]
                conn = shard.connect()
                trans = conn.begin()

                connections[shard_id] = conn
                transactions[shard_id] = trans

            yield connections
```

```
            # Phase 2: Commit all transactions
            for shard_id in shard_ids:
                transactions[shard_id].commit()

        except Exception as e:
            # Rollback all transactions on error
            for shard_id in shard_ids:
                if shard_id in transactions:
                    try:
                        transactions[shard_id].rollback()
                    except:
                        pass
            raise e

        finally:
            # Close all connections
            for conn in connections.values():
                conn.close()

    def transfer_user_data(self, from_user_id, to_user_id, amount):
        """Transfer data between users potentially on different shards"""
        from_shard_id = hash(str(from_user_id)) % len(self.router.shards)
        to_shard_id = hash(str(to_user_id)) % len(self.router.shards)

        shard_ids = list(set([from_shard_id, to_shard_id]))

        with self.distributed_transaction(shard_ids) as connections:
            # Deduct from source user
            from_conn = connections[from_shard_id]
            from_conn.execute("""
                UPDATE user_balances
                SET balance = balance - %s
                WHERE user_id = %s AND balance >= %s
            """, (amount, from_user_id, amount))

            # Add to destination user
            to_conn = connections[to_shard_id]
            to_conn.execute("""
                UPDATE user_balances
                SET balance = balance + %s
                WHERE user_id = %s
            """, (amount, to_user_id))

            # Log transaction in both shards if different
            if from_shard_id != to_shard_id:
                from_conn.execute("""
                    INSERT INTO transfer_logs (from_user_id, to_user_id, amount,
type)
                    VALUES (%s, %s, %s, 'outgoing')
                """, (from_user_id, to_user_id, amount))

                to_conn.execute("""
                    INSERT INTO transfer_logs (from_user_id, to_user_id, amount,
```

```
type)
                    VALUES (%s, %s, %s, 'incoming')
                """, (from_user_id, to_user_id, amount))
            else:
                from_conn.execute("""
                    INSERT INTO transfer_logs (from_user_id, to_user_id, amount,
type)
                    VALUES (%s, %s, %s, 'internal')
                """, (from_user_id, to_user_id, amount))

# Usage
dist_tx = DistributedTransactionManager(router)

# Transfer between users (potentially on different shards)
try:
    dist_tx.transfer_user_data(12345, 67890, 100.00)
    print("Transfer completed successfully")
except Exception as e:
    print(f"Transfer failed: {e}")
```

⚖️ Shard Rebalancing

Consistent Hashing

```
import hashlib
import bisect

class ConsistentHashRouter:
    def __init__(self, shards, virtual_nodes=150):
        self.shards = shards
        self.virtual_nodes = virtual_nodes
        self.ring = {}
        self.sorted_keys = []

        self._build_ring()

    def _hash(self, key):
        """Hash function for consistent hashing"""
        return int(hashlib.md5(str(key).encode()).hexdigest(), 16)

    def _build_ring(self):
        """Build the consistent hash ring"""
        for shard_id in self.shards:
            for i in range(self.virtual_nodes):
                virtual_key = self._hash(f"{shard_id}:{i}")
                self.ring[virtual_key] = shard_id

        self.sorted_keys = sorted(self.ring.keys())

    def get_shard(self, key):
```

```
        """Get shard for given key"""
        if not self.ring:
            return None

        hash_key = self._hash(key)

        # Find the first shard clockwise from the hash
        idx = bisect.bisect_right(self.sorted_keys, hash_key)
        if idx == len(self.sorted_keys):
            idx = 0

        return self.ring[self.sorted_keys[idx]]

    def add_shard(self, shard_id):
        """Add new shard to the ring"""
        self.shards.append(shard_id)

        for i in range(self.virtual_nodes):
            virtual_key = self._hash(f"{shard_id}:{i}")
            self.ring[virtual_key] = shard_id

        self.sorted_keys = sorted(self.ring.keys())

    def remove_shard(self, shard_id):
        """Remove shard from the ring"""
        self.shards.remove(shard_id)

        # Remove virtual nodes for this shard
        keys_to_remove = []
        for key, shard in self.ring.items():
            if shard == shard_id:
                keys_to_remove.append(key)

        for key in keys_to_remove:
            del self.ring[key]

        self.sorted_keys = sorted(self.ring.keys())

    def get_affected_keys(self, old_shard, new_shard, key_range):
        """Get keys that need to be moved when rebalancing"""
        affected_keys = []

        for key in key_range:
            current_shard = self.get_shard(key)
            if current_shard != old_shard:
                affected_keys.append(key)

        return affected_keys

# Usage
shards = ['shard0', 'shard1', 'shard2', 'shard3']
consistent_router = ConsistentHashRouter(shards)

# Test distribution
```

```
user_ids = range(1, 10001)
shard_distribution = {}

for user_id in user_ids:
    shard = consistent_router.get_shard(user_id)
    shard_distribution[shard] = shard_distribution.get(shard, 0) + 1

print("Distribution before adding shard:")
for shard, count in shard_distribution.items():
    print(f"{shard}: {count} users ({count/len(user_ids)*100:.1f}%)")

# Add new shard
consistent_router.add_shard('shard4')

# Check new distribution
new_distribution = {}
for user_id in user_ids:
    shard = consistent_router.get_shard(user_id)
    new_distribution[shard] = new_distribution.get(shard, 0) + 1

print("\nDistribution after adding shard4:")
for shard, count in new_distribution.items():
    print(f"{shard}: {count} users ({count/len(user_ids)*100:.1f}%)")
```

Data Migration

```
class ShardMigrationManager:
    def __init__(self, source_router, target_router):
        self.source_router = source_router
        self.target_router = target_router

    def migrate_user_data(self, user_id):
        """Migrate single user's data between shards"""
        source_shard = self.source_router.get_shard(user_id)
        target_shard = self.target_router.get_shard(user_id)

        if source_shard == target_shard:
            return  # No migration needed

        source_conn = self.source_router.shards[source_shard].connect()
        target_conn = self.target_router.shards[target_shard].connect()

        try:
            with source_conn.begin() as source_tx:
                with target_conn.begin() as target_tx:
                    # Copy user data
                    user_data = source_conn.execute(
                        "SELECT * FROM users WHERE id = %s", (user_id,)
                    ).fetchone()

                    if user_data:
```

```
                        target_conn.execute("""
                            INSERT INTO users (id, username, email, created_at)
                            VALUES (%(id)s, %(username)s, %(email)s, %
(created_at)s)
                            ON DUPLICATE KEY UPDATE
                            username = VALUES(username),
                            email = VALUES(email)
                        """, dict(user_data))

                    # Copy user orders
                    orders = source_conn.execute(
                        "SELECT * FROM orders WHERE user_id = %s", (user_id,)
                    ).fetchall()

                    for order in orders:
                        target_conn.execute("""
                            INSERT INTO orders (id, user_id, total_amount, status,
created_at)
                            VALUES (%(id)s, %(user_id)s, %(total_amount)s, %
(status)s, %(created_at)s)
                            ON DUPLICATE KEY UPDATE
                            total_amount = VALUES(total_amount),
                            status = VALUES(status)
                        """, dict(order))

                        # Copy order items
                        order_items = source_conn.execute(
                            "SELECT * FROM order_items WHERE order_id = %s",
(order['id'],)
                        ).fetchall()

                        for item in order_items:
                            target_conn.execute("""
                                INSERT INTO order_items (id, order_id, product_id,
quantity, price)
                                VALUES (%(id)s, %(order_id)s, %(product_id)s, %
(quantity)s, %(price)s)
                                ON DUPLICATE KEY UPDATE
                                quantity = VALUES(quantity),
                                price = VALUES(price)
                            """, dict(item))

                    # Verify migration
                    source_count = source_conn.execute(
                        "SELECT COUNT(*) FROM orders WHERE user_id = %s",
(user_id,)
                    ).scalar()

                    target_count = target_conn.execute(
                        "SELECT COUNT(*) FROM orders WHERE user_id = %s",
(user_id,)
                    ).scalar()

                    if source_count != target_count:
```

```
                        raise Exception(f"Migration verification failed:
{source_count} != {target_count}")

                    # Delete from source after successful migration
                    source_conn.execute("DELETE FROM order_items WHERE order_id IN
(SELECT id FROM orders WHERE user_id = %s)", (user_id,))
                    source_conn.execute("DELETE FROM orders WHERE user_id = %s",
(user_id,))
                    source_conn.execute("DELETE FROM users WHERE id = %s",
(user_id,))

        finally:
            source_conn.close()
            target_conn.close()

    def migrate_batch(self, user_ids, batch_size=100):
        """Migrate users in batches"""
        total_users = len(user_ids)
        migrated = 0

        for i in range(0, total_users, batch_size):
            batch = user_ids[i:i + batch_size]

            for user_id in batch:
                try:
                    self.migrate_user_data(user_id)
                    migrated += 1

                    if migrated % 10 == 0:
                        print(f"Migrated {migrated}/{total_users} users
({migrated/total_users*100:.1f}%)")

                except Exception as e:
                    print(f"Failed to migrate user {user_id}: {e}")

            # Small delay between batches
            time.sleep(0.1)

        print(f"Migration completed: {migrated}/{total_users} users migrated")

# Usage
old_router = UserShardRouter(shard_count=4)
new_router = UserShardRouter(shard_count=5)  # Added one more shard

migration_manager = ShardMigrationManager(old_router, new_router)

# Migrate specific users
users_to_migrate = [12345, 67890, 11111]
migration_manager.migrate_batch(users_to_migrate)
```

🏗️ Real-World Example: E-commerce Sharding Architecture

Multi-Dimensional Sharding

```
class EcommerceShardingStrategy:
    def __init__(self):
        # User shards (by user_id hash)
        self.user_shards = {
            0: create_engine('mysql://user:pass@user-shard-0:3306/ecommerce'),
            1: create_engine('mysql://user:pass@user-shard-1:3306/ecommerce'),
            2: create_engine('mysql://user:pass@user-shard-2:3306/ecommerce'),
            3: create_engine('mysql://user:pass@user-shard-3:3306/ecommerce')
        }

        # Product shards (by category)
        self.product_shards = {
            'electronics': create_engine('mysql://user:pass@product-
electronics:3306/ecommerce'),
            'clothing': create_engine('mysql://user:pass@product-
clothing:3306/ecommerce'),
            'books': create_engine('mysql://user:pass@product-
books:3306/ecommerce'),
            'home': create_engine('mysql://user:pass@product-home:3306/ecommerce')
        }

        # Order shards (by time + user_id)
        self.order_shards = {
            ('2024', 0): create_engine('mysql://user:pass@orders-2024-
0:3306/ecommerce'),
            ('2024', 1): create_engine('mysql://user:pass@orders-2024-
1:3306/ecommerce'),
            ('2024', 2): create_engine('mysql://user:pass@orders-2024-
2:3306/ecommerce'),
            ('2024', 3): create_engine('mysql://user:pass@orders-2024-
3:3306/ecommerce')
        }

    def get_user_shard(self, user_id):
        """Get shard for user data"""
        shard_id = hash(str(user_id)) % len(self.user_shards)
        return self.user_shards[shard_id]

    def get_product_shard(self, category):
        """Get shard for product data"""
        return self.product_shards.get(category,
self.product_shards['electronics'])

    def get_order_shard(self, user_id, order_date):
        """Get shard for order data"""
        year = str(order_date.year)
        user_shard_id = hash(str(user_id)) % 4
        return self.order_shards.get((year, user_shard_id))

    def create_user(self, user_data):
```

```
        """Create user in appropriate shard"""
        shard = self.get_user_shard(user_data['id'])

        with shard.connect() as conn:
            conn.execute("""
                INSERT INTO users (id, username, email, country, created_at)
                VALUES (%(id)s, %(username)s, %(email)s, %(country)s, %
(created_at)s)
            """, user_data)

    def create_product(self, product_data):
        """Create product in category-specific shard"""
        shard = self.get_product_shard(product_data['category'])

        with shard.connect() as conn:
            conn.execute("""
                INSERT INTO products (id, name, category, price, description,
created_at)
                VALUES (%(id)s, %(name)s, %(category)s, %(price)s, %
(description)s, %(created_at)s)
            """, product_data)

    def create_order(self, order_data):
        """Create order with items from multiple product shards"""
        order_shard = self.get_order_shard(order_data['user_id'],
order_data['created_at'])

        with order_shard.connect() as conn:
            with conn.begin():
                # Insert order
                result = conn.execute("""
                    INSERT INTO orders (user_id, total_amount, status, created_at)
                    VALUES (%(user_id)s, %(total_amount)s, %(status)s, %
(created_at)s)
                """, order_data)

                order_id = result.lastrowid

                # Insert order items
                for item in order_data['items']:
                    item['order_id'] = order_id
                    conn.execute("""
                        INSERT INTO order_items (order_id, product_id, quantity,
price)
                        VALUES (%(order_id)s, %(product_id)s, %(quantity)s, %
(price)s)
                    """, item)

                return order_id

    def get_user_orders(self, user_id, start_date=None, end_date=None):
        """Get user orders, potentially from multiple time shards"""
        all_orders = []
```

```
        # Determine which order shards to query
        if start_date and end_date:
            years = set()
            current_year = start_date.year
            while current_year <= end_date.year:
                years.add(str(current_year))
                current_year += 1
        else:
            years = ['2024']  # Default to current year

        user_shard_id = hash(str(user_id)) % 4

        for year in years:
            shard_key = (year, user_shard_id)
            if shard_key in self.order_shards:
                shard = self.order_shards[shard_key]

                with shard.connect() as conn:
                    query = "SELECT * FROM orders WHERE user_id = %s"
                    params = [user_id]

                    if start_date and end_date:
                        query += " AND created_at BETWEEN %s AND %s"
                        params.extend([start_date, end_date])

                    query += " ORDER BY created_at DESC"

                    orders = conn.execute(query, params).fetchall()
                    all_orders.extend([dict(order) for order in orders])

        # Sort combined results
        return sorted(all_orders, key=lambda x: x['created_at'], reverse=True)

    def get_product_analytics(self, category, start_date, end_date):
        """Get product analytics for specific category"""
        product_shard = self.get_product_shard(category)

        # Need to query across all order shards for this category
        all_sales = []

        for shard_key, order_shard in self.order_shards.items():
            with order_shard.connect() as conn:
                sales = conn.execute("""
                    SELECT
                        oi.product_id,
                        SUM(oi.quantity) as total_quantity,
                        SUM(oi.quantity * oi.price) as total_revenue
                    FROM order_items oi
                    JOIN orders o ON oi.order_id = o.id
                    WHERE o.created_at BETWEEN %s AND %s
                    GROUP BY oi.product_id
                """, (start_date, end_date)).fetchall()

                all_sales.extend([dict(sale) for sale in sales])
```

```
        # Aggregate sales across shards
        product_totals = {}
        for sale in all_sales:
            product_id = sale['product_id']
            if product_id not in product_totals:
                product_totals[product_id] = {
                    'product_id': product_id,
                    'total_quantity': 0,
                    'total_revenue': 0
                }

            product_totals[product_id]['total_quantity'] += sale['total_quantity']
            product_totals[product_id]['total_revenue'] += sale['total_revenue']

        # Get product details from product shard
        with product_shard.connect() as conn:
            for product_total in product_totals.values():
                product = conn.execute(
                    "SELECT name, category FROM products WHERE id = %s",
                    (product_total['product_id'],)
                ).fetchone()

                if product:
                    product_total.update(dict(product))

        return list(product_totals.values())

# Usage
ecommerce_sharding = EcommerceShardingStrategy()

# Create user (routed to user shard)
ecommerce_sharding.create_user({
    'id': 12345,
    'username': 'john_doe',
    'email': 'john@example.com',
    'country': 'US',
    'created_at': datetime.now()
})

# Create products (routed to category shards)
ecommerce_sharding.create_product({
    'id': 1001,
    'name': 'iPhone 15',
    'category': 'electronics',
    'price': 999.99,
    'description': 'Latest iPhone model',
    'created_at': datetime.now()
})

# Create order (routed to time+user shard)
order_id = ecommerce_sharding.create_order({
    'user_id': 12345,
    'total_amount': 1049.98,
```

```
    'status': 'pending',
    'created_at': datetime.now(),
    'items': [
        {'product_id': 1001, 'quantity': 1, 'price': 999.99},
        {'product_id': 1002, 'quantity': 1, 'price': 49.99}
    ]
})

# Get user orders (may query multiple time shards)
orders = ecommerce_sharding.get_user_orders(12345)

# Get analytics (aggregates across multiple shards)
electronics_analytics = ecommerce_sharding.get_product_analytics(
    'electronics',
    datetime(2024, 1, 1),
    datetime(2024, 1, 31)
)
```

🎯 Interview Questions

Basic Questions

1. **What's the difference between partitioning and sharding?**
2. **What are the main types of partitioning strategies?**
3. **How do you choose a good shard key?**

Intermediate Questions

1. **How would you handle cross-shard queries efficiently?**
2. **What are the challenges of distributed transactions across shards?**
3. **How do you rebalance data when adding new shards?**

Advanced Questions

1. **Design a sharding strategy for a global social media platform**
2. **How would you implement consistent hashing for dynamic shard management?**
3. **What are the trade-offs between different sharding strategies?**

⚠️ Common Pitfalls

1. **Poor Shard Key Selection**

```
# Problem: Hotspot sharding
def bad_shard_key(timestamp):
    return timestamp.hour % 4  # All traffic goes to one shard during peak hours

# Solution: Combine multiple factors
```

```
def good_shard_key(user_id, timestamp):
    return hash(f"{user_id}:{timestamp.date()}") % 4
```

2. **Cross-Shard Join Dependencies**

```
-- Problem: Joins across shards are expensive
SELECT u.username, o.total_amount
FROM users u  -- On user shard
JOIN orders o ON u.id = o.user_id  -- On order shard
WHERE o.created_at > '2024-01-01';

-- Solution: Denormalize or use application-level joins
-- Store user info in order records
ALTER TABLE orders ADD COLUMN username VARCHAR(50);
```

3. **Inadequate Monitoring**

```
# Problem: Not monitoring shard distribution
# Solution: Regular distribution analysis
def analyze_shard_distribution(router, sample_keys):
    distribution = {}
    for key in sample_keys:
        shard = router.get_shard(key)
        distribution[shard] = distribution.get(shard, 0) + 1

    total = len(sample_keys)
    for shard, count in distribution.items():
        percentage = (count / total) * 100
        print(f"Shard {shard}: {count} keys ({percentage:.1f}%)")

        if percentage > 30:  # Alert if any shard has >30% of data
            print(f"WARNING: Shard {shard} is overloaded!")
```

Next Steps

Congratulations! You've mastered database partitioning and sharding. You now understand:

- ✅ **Table partitioning strategies and implementation**
- ✅ **Horizontal and vertical sharding techniques**
- ✅ **Shard key selection and distribution strategies**
- ✅ **Cross-shard operations and distributed transactions**
- ✅ **Rebalancing and migration strategies**
- ✅ **Real-world sharding architectures**

Continue to → Chapter 21: ETL & Data Warehousing

You're now equipped to design and implement scalable database architectures that can handle massive datasets across multiple servers! 🚀

Chapter 21: ETL & Data Warehousing

🗄️ What You'll Learn

ETL (Extract, Transform, Load) and data warehousing are fundamental for business intelligence and analytics. This chapter covers data extraction from multiple sources, transformation processes, loading strategies, dimensional modeling, and building robust data pipelines for both MySQL and PostgreSQL environments.

🎯 Learning Objectives

- **ETL Fundamentals**: Understand Extract, Transform, Load processes
- **Data Extraction**: Pull data from various sources (databases, APIs, files)
- **Data Transformation**: Clean, validate, and transform data
- **Data Loading**: Efficiently load data into warehouses
- **Dimensional Modeling**: Design star and snowflake schemas
- **Data Pipelines**: Build automated, scalable data workflows
- **Performance Optimization**: Optimize ETL processes for large datasets

---

📖 Concept Explanation

What is ETL?

ETL is a data integration process that:

- **Extract**: Retrieves data from source systems
- **Transform**: Cleans, validates, and converts data
- **Load**: Inserts processed data into target systems

What is Data Warehousing?

Data Warehousing involves:

- Centralized repository for integrated data
- Optimized for analytical queries (OLAP)
- Historical data storage
- Support for business intelligence tools

ETL vs ELT

Aspect	ETL	ELT
Processing	Transform before loading	Transform after loading
Performance	Slower for large datasets	Faster initial loading

Aspect	ETL	ELT
Flexibility	Less flexible	More flexible
Storage	Requires staging area	Uses target system storage
Best For	Traditional warehouses	Cloud/modern platforms

🔧 Data Extraction

Database Extraction

MySQL Source Extraction

```
import mysql.connector
import pandas as pd
from datetime import datetime, timedelta
import logging

class MySQLExtractor:
    def __init__(self, config):
        self.config = config
        self.connection = None

    def connect(self):
        """Establish database connection"""
        try:
            self.connection = mysql.connector.connect(**self.config)
            logging.info("Connected to MySQL source")
        except Exception as e:
            logging.error(f"Failed to connect to MySQL: {e}")
            raise

    def extract_incremental(self, table, timestamp_column, last_extracted):
        """Extract data incrementally based on timestamp"""
        query = f"""
            SELECT * FROM {table}
            WHERE {timestamp_column} > %s
            ORDER BY {timestamp_column}
        """

        try:
            df = pd.read_sql(query, self.connection, params=[last_extracted])
            logging.info(f"Extracted {len(df)} rows from {table}")
            return df
        except Exception as e:
            logging.error(f"Failed to extract from {table}: {e}")
            raise

    def extract_full(self, table, batch_size=10000):
        """Extract full table in batches"""
        offset = 0
```

```
        all_data = []

        while True:
            query = f"""
                SELECT * FROM {table}
                LIMIT {batch_size} OFFSET {offset}
            """

            batch_df = pd.read_sql(query, self.connection)

            if batch_df.empty:
                break

            all_data.append(batch_df)
            offset += batch_size
            logging.info(f"Extracted batch {offset//batch_size}, {len(batch_df)}
rows")

        if all_data:
            return pd.concat(all_data, ignore_index=True)
        return pd.DataFrame()

    def extract_with_joins(self, query):
        """Extract data using complex queries with joins"""
        try:
            df = pd.read_sql(query, self.connection)
            logging.info(f"Extracted {len(df)} rows from custom query")
            return df
        except Exception as e:
            logging.error(f"Failed to execute custom query: {e}")
            raise

    def close(self):
        """Close database connection"""
        if self.connection:
            self.connection.close()
            logging.info("MySQL connection closed")

# Usage
mysql_config = {
    'host': 'localhost',
    'user': 'etl_user',
    'password': 'password',
    'database': 'ecommerce'
}

extractor = MySQLExtractor(mysql_config)
extractor.connect()

# Extract incremental data
last_run = datetime.now() - timedelta(days=1)
orders_df = extractor.extract_incremental('orders', 'created_at', last_run)

# Extract with complex query
```

```
customer_analytics_query = """
    SELECT
        u.id as user_id,
        u.username,
        u.email,
        COUNT(o.id) as total_orders,
        SUM(o.total_amount) as total_spent,
        AVG(o.total_amount) as avg_order_value,
        MAX(o.created_at) as last_order_date
    FROM users u
    LEFT JOIN orders o ON u.id = o.user_id
    WHERE u.created_at >= %s
    GROUP BY u.id, u.username, u.email
"""

customer_df = extractor.extract_with_joins(customer_analytics_query)
extractor.close()
```

PostgreSQL Source Extraction

```
import psycopg2
import pandas as pd
from sqlalchemy import create_engine

class PostgreSQLExtractor:
    def __init__(self, config):
        self.config = config
        self.engine = None

    def connect(self):
        """Create SQLAlchemy engine"""
        connection_string = f"postgresql://{self.config['user']}:
{self.config['password']}@{self.config['host']}:
{self.config['port']}/{self.config['database']}"
        self.engine = create_engine(connection_string)
        logging.info("Connected to PostgreSQL source")

    def extract_with_cursor(self, query, chunk_size=10000):
        """Extract large datasets using server-side cursor"""
        connection = self.engine.raw_connection()
        cursor = connection.cursor('server_side_cursor')

        try:
            cursor.execute(query)

            while True:
                rows = cursor.fetchmany(chunk_size)
                if not rows:
                    break

                # Convert to DataFrame
```

```
                columns = [desc[0] for desc in cursor.description]
                chunk_df = pd.DataFrame(rows, columns=columns)

                yield chunk_df

        finally:
            cursor.close()
            connection.close()

    def extract_partitioned_table(self, table_name, partition_column, start_date,
end_date):
        """Extract from partitioned table with date range"""
        query = f"""
            SELECT * FROM {table_name}
            WHERE {partition_column} BETWEEN %s AND %s
            ORDER BY {partition_column}
        """

        return pd.read_sql(query, self.engine, params=[start_date, end_date])

    def extract_json_data(self, table, json_column, json_path):
        """Extract and flatten JSON data"""
        query = f"""
            SELECT
                id,
                {json_column}->>'$.{json_path}' as extracted_value,
                {json_column}
            FROM {table}
            WHERE {json_column} IS NOT NULL
        """

        return pd.read_sql(query, self.engine)

# Usage
postgres_config = {
    'host': 'localhost',
    'port': 5432,
    'user': 'etl_user',
    'password': 'password',
    'database': 'analytics'
}

pg_extractor = PostgreSQLExtractor(postgres_config)
pg_extractor.connect()

# Extract large dataset in chunks
for chunk in pg_extractor.extract_with_cursor("SELECT * FROM large_table"):
    # Process each chunk
    print(f"Processing chunk with {len(chunk)} rows")
```

API Data Extraction

```
import requests
import json
import time
from typing import List, Dict

class APIExtractor:
    def __init__(self, base_url, api_key=None, rate_limit=100):
        self.base_url = base_url
        self.api_key = api_key
        self.rate_limit = rate_limit
        self.session = requests.Session()

        if api_key:
            self.session.headers.update({'Authorization': f'Bearer {api_key}'})

    def extract_paginated(self, endpoint, params=None, page_size=100):
        """Extract data from paginated API"""
        all_data = []
        page = 1

        while True:
            request_params = params.copy() if params else {}
            request_params.update({
                'page': page,
                'per_page': page_size
            })

            response = self._make_request(endpoint, request_params)

            if not response or 'data' not in response:
                break

            data = response['data']
            if not data:
                break

            all_data.extend(data)
            page += 1

            # Rate limiting
            time.sleep(1 / self.rate_limit)

            logging.info(f"Extracted page {page-1}, total records:
{len(all_data)}")

        return all_data

    def extract_with_cursor(self, endpoint, params=None):
        """Extract data using cursor-based pagination"""
        all_data = []
        cursor = None

        while True:
```

```
            request_params = params.copy() if params else {}
            if cursor:
                request_params['cursor'] = cursor

            response = self._make_request(endpoint, request_params)

            if not response or 'data' not in response:
                break

            data = response['data']
            if not data:
                break

            all_data.extend(data)
            cursor = response.get('next_cursor')

            if not cursor:
                break

            time.sleep(1 / self.rate_limit)

        return all_data

    def _make_request(self, endpoint, params=None):
        """Make HTTP request with error handling"""
        url = f"{self.base_url}/{endpoint}"

        try:
            response = self.session.get(url, params=params, timeout=30)
            response.raise_for_status()
            return response.json()

        except requests.exceptions.RequestException as e:
            logging.error(f"API request failed: {e}")
            return None

# Usage
api_extractor = APIExtractor('https://api.example.com/v1', api_key='your_api_key')

# Extract customer data
customers = api_extractor.extract_paginated('customers', {'status': 'active'})
customer_df = pd.DataFrame(customers)

# Extract orders with cursor pagination
orders = api_extractor.extract_with_cursor('orders', {'date_from': '2024-01-01'})
orders_df = pd.DataFrame(orders)
```

File Data Extraction

```
import pandas as pd
import json
```

```
import xml.etree.ElementTree as ET
from pathlib import Path

class FileExtractor:
    def __init__(self, base_path):
        self.base_path = Path(base_path)

    def extract_csv(self, filename, **kwargs):
        """Extract data from CSV files"""
        file_path = self.base_path / filename

        try:
            df = pd.read_csv(file_path, **kwargs)
            logging.info(f"Extracted {len(df)} rows from {filename}")
            return df
        except Exception as e:
            logging.error(f"Failed to read CSV {filename}: {e}")
            raise

    def extract_excel(self, filename, sheet_name=None):
        """Extract data from Excel files"""
        file_path = self.base_path / filename

        try:
            if sheet_name:
                df = pd.read_excel(file_path, sheet_name=sheet_name)
            else:
                # Read all sheets
                excel_file = pd.ExcelFile(file_path)
                dfs = {}
                for sheet in excel_file.sheet_names:
                    dfs[sheet] = pd.read_excel(file_path, sheet_name=sheet)
                return dfs

            logging.info(f"Extracted {len(df)} rows from {filename}")
            return df
        except Exception as e:
            logging.error(f"Failed to read Excel {filename}: {e}")
            raise

    def extract_json(self, filename):
        """Extract data from JSON files"""
        file_path = self.base_path / filename

        try:
            with open(file_path, 'r') as f:
                data = json.load(f)

            # Convert to DataFrame if it's a list of objects
            if isinstance(data, list):
                df = pd.DataFrame(data)
                logging.info(f"Extracted {len(df)} rows from {filename}")
                return df
            else:
```

```
                return data
        except Exception as e:
            logging.error(f"Failed to read JSON {filename}: {e}")
            raise

    def extract_xml(self, filename, record_path):
        """Extract data from XML files"""
        file_path = self.base_path / filename

        try:
            tree = ET.parse(file_path)
            root = tree.getroot()

            records = []
            for record in root.findall(record_path):
                record_data = {}
                for child in record:
                    record_data[child.tag] = child.text
                records.append(record_data)

            df = pd.DataFrame(records)
            logging.info(f"Extracted {len(df)} rows from {filename}")
            return df
        except Exception as e:
            logging.error(f"Failed to read XML {filename}: {e}")
            raise

    def extract_multiple_files(self, pattern, extract_func):
        """Extract data from multiple files matching pattern"""
        files = list(self.base_path.glob(pattern))
        all_data = []

        for file_path in files:
            try:
                df = extract_func(file_path.name)
                df['source_file'] = file_path.name
                all_data.append(df)
            except Exception as e:
                logging.error(f"Failed to process {file_path}: {e}")
                continue

        if all_data:
            combined_df = pd.concat(all_data, ignore_index=True)
            logging.info(f"Combined {len(files)} files, total {len(combined_df)}
rows")
            return combined_df

        return pd.DataFrame()

# Usage
file_extractor = FileExtractor('/data/sources')

# Extract from single files
sales_df = file_extractor.extract_csv('sales_2024.csv')
```

```
products_df = file_extractor.extract_excel('products.xlsx', sheet_name='Products')
config_data = file_extractor.extract_json('config.json')

# Extract from multiple CSV files
all_sales = file_extractor.extract_multiple_files('sales_*.csv',
file_extractor.extract_csv)
```

Data Transformation

Data Cleaning and Validation

```
import pandas as pd
import numpy as np
from datetime import datetime
import re

class DataTransformer:
    def __init__(self):
        self.transformation_log = []

    def clean_text_data(self, df, text_columns):
        """Clean and standardize text data"""
        df_clean = df.copy()

        for col in text_columns:
            if col in df_clean.columns:
                # Remove extra whitespace
                df_clean[col] = df_clean[col].astype(str).str.strip()

                # Standardize case
                df_clean[col] = df_clean[col].str.title()

                # Remove special characters (keep alphanumeric and spaces)
                df_clean[col] = df_clean[col].str.replace(r'[^\w\s]', '',
regex=True)

                # Replace multiple spaces with single space
                df_clean[col] = df_clean[col].str.replace(r'\s+', ' ', regex=True)

        self._log_transformation(f"Cleaned text columns: {text_columns}")
        return df_clean

    def validate_email(self, df, email_column):
        """Validate and clean email addresses"""
        df_clean = df.copy()

        email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

        # Mark invalid emails
        df_clean['email_valid'] = df_clean[email_column].str.match(email_pattern,
```

```
na=False)

        # Clean valid emails
        df_clean[email_column] = df_clean[email_column].str.lower().str.strip()

        invalid_count = (~df_clean['email_valid']).sum()
        self._log_transformation(f"Email validation: {invalid_count} invalid
emails found")

        return df_clean

    def standardize_phone_numbers(self, df, phone_column):
        """Standardize phone number format"""
        df_clean = df.copy()

        def clean_phone(phone):
            if pd.isna(phone):
                return None

            # Remove all non-digit characters
            digits = re.sub(r'\D', '', str(phone))

            # Handle different formats
            if len(digits) == 10:
                return f"({digits[:3]}) {digits[3:6]}-{digits[6:]}"
            elif len(digits) == 11 and digits[0] == '1':
                return f"({digits[1:4]}) {digits[4:7]}-{digits[7:]}"
            else:
                return digits  # Return as-is if format is unclear

        df_clean[f"{phone_column}_standardized"] =
df_clean[phone_column].apply(clean_phone)

        self._log_transformation(f"Standardized phone numbers in {phone_column}")
        return df_clean

    def handle_missing_values(self, df, strategies):
        """Handle missing values with different strategies"""
        df_clean = df.copy()

        for column, strategy in strategies.items():
            if column not in df_clean.columns:
                continue

            missing_count = df_clean[column].isna().sum()

            if strategy == 'drop':
                df_clean = df_clean.dropna(subset=[column])
            elif strategy == 'mean':
                df_clean[column].fillna(df_clean[column].mean(), inplace=True)
            elif strategy == 'median':
                df_clean[column].fillna(df_clean[column].median(), inplace=True)
            elif strategy == 'mode':
                mode_value = df_clean[column].mode().iloc[0] if not
```

```
df_clean[column].mode().empty else 'Unknown'
                df_clean[column].fillna(mode_value, inplace=True)
            elif isinstance(strategy, (str, int, float)):
                df_clean[column].fillna(strategy, inplace=True)

            self._log_transformation(f"Handled {missing_count} missing values in
{column} using {strategy}")

        return df_clean

    def detect_outliers(self, df, numeric_columns, method='iqr'):
        """Detect outliers in numeric data"""
        outliers_info = {}

        for col in numeric_columns:
            if col not in df.columns or not
pd.api.types.is_numeric_dtype(df[col]):
                continue

            if method == 'iqr':
                Q1 = df[col].quantile(0.25)
                Q3 = df[col].quantile(0.75)
                IQR = Q3 - Q1
                lower_bound = Q1 - 1.5 * IQR
                upper_bound = Q3 + 1.5 * IQR

                outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]

            elif method == 'zscore':
                z_scores = np.abs((df[col] - df[col].mean()) / df[col].std())
                outliers = df[z_scores > 3]

            outliers_info[col] = {
                'count': len(outliers),
                'percentage': len(outliers) / len(df) * 100,
                'outlier_indices': outliers.index.tolist()
            }

        return outliers_info

    def create_derived_columns(self, df, derivations):
        """Create derived columns based on existing data"""
        df_enhanced = df.copy()

        for new_col, formula in derivations.items():
            try:
                if callable(formula):
                    df_enhanced[new_col] = formula(df_enhanced)
                else:
                    df_enhanced[new_col] = eval(formula, {'df': df_enhanced, 'pd':
pd, 'np': np})

                self._log_transformation(f"Created derived column: {new_col}")
            except Exception as e:
```

```
                logging.error(f"Failed to create derived column {new_col}: {e}")

        return df_enhanced

    def _log_transformation(self, message):
        """Log transformation steps"""
        timestamp = datetime.now().isoformat()
        self.transformation_log.append(f"{timestamp}: {message}")
        logging.info(message)

# Usage
transformer = DataTransformer()

# Clean customer data
customers_clean = transformer.clean_text_data(
    customers_df,
    ['first_name', 'last_name', 'company']
)

# Validate emails
customers_clean = transformer.validate_email(customers_clean, 'email')

# Standardize phone numbers
customers_clean = transformer.standardize_phone_numbers(customers_clean, 'phone')

# Handle missing values
missing_strategies = {
    'age': 'median',
    'income': 'mean',
    'city': 'Unknown',
    'country': 'US'
}
customers_clean = transformer.handle_missing_values(customers_clean,
missing_strategies)

# Create derived columns
derivations = {
    'full_name': lambda df: df['first_name'] + ' ' + df['last_name'],
    'age_group': lambda df: pd.cut(df['age'], bins=[0, 25, 35, 50, 65, 100],
labels=['18-25', '26-35', '36-50', '51-65', '65+']),
    'customer_lifetime_days': lambda df: (pd.Timestamp.now() -
pd.to_datetime(df['created_at'])).dt.days
}
customers_enhanced = transformer.create_derived_columns(customers_clean,
derivations)

# Check transformation log
for log_entry in transformer.transformation_log:
    print(log_entry)
```

Data Type Conversions and Formatting

```
class DataTypeConverter:
    def __init__(self):
        pass

    def convert_data_types(self, df, type_mapping):
        """Convert data types based on mapping"""
        df_converted = df.copy()

        for column, target_type in type_mapping.items():
            if column not in df_converted.columns:
                continue

            try:
                if target_type == 'datetime':
                    df_converted[column] = pd.to_datetime(df_converted[column])
                elif target_type == 'category':
                    df_converted[column] = df_converted[column].astype('category')
                elif target_type == 'numeric':
                    df_converted[column] = pd.to_numeric(df_converted[column],
errors='coerce')
                else:
                    df_converted[column] =
df_converted[column].astype(target_type)

                logging.info(f"Converted {column} to {target_type}")
            except Exception as e:
                logging.error(f"Failed to convert {column} to {target_type}: {e}")

        return df_converted

    def normalize_currency(self, df, currency_columns, target_currency='USD'):
        """Normalize currency values"""
        # Exchange rates (in real implementation, fetch from API)
        exchange_rates = {
            'EUR': 1.1,
            'GBP': 1.25,
            'JPY': 0.0067,
            'CAD': 0.74
        }

        df_normalized = df.copy()

        for col in currency_columns:
            if col not in df_normalized.columns:
                continue

            # Assume currency is in a separate column or can be detected
            currency_col = f"{col}_currency"

            if currency_col in df_normalized.columns:
                for currency, rate in exchange_rates.items():
                    mask = df_normalized[currency_col] == currency
                    df_normalized.loc[mask, col] = df_normalized.loc[mask, col] *
```

```
rate

                df_normalized[currency_col] = target_currency

            logging.info(f"Normalized currency for {col} to {target_currency}")

        return df_normalized

    def standardize_dates(self, df, date_columns, target_format='%Y-%m-%d'):
        """Standardize date formats"""
        df_standardized = df.copy()

        for col in date_columns:
            if col not in df_standardized.columns:
                continue

            # Convert to datetime first
            df_standardized[col] = pd.to_datetime(df_standardized[col],
errors='coerce')

            # Format as string if needed
            if target_format:
                df_standardized[f"{col}_formatted"] =
df_standardized[col].dt.strftime(target_format)

            logging.info(f"Standardized date format for {col}")

        return df_standardized

# Usage
converter = DataTypeConverter()

# Convert data types
type_mapping = {
    'user_id': 'int64',
    'created_at': 'datetime',
    'status': 'category',
    'amount': 'numeric'
}
orders_converted = converter.convert_data_types(orders_df, type_mapping)

# Normalize currencies
orders_normalized = converter.normalize_currency(orders_converted, ['amount',
'tax'])

# Standardize dates
orders_final = converter.standardize_dates(orders_normalized, ['created_at',
'updated_at'])
```

📥 Data Loading

Bulk Loading Strategies

MySQL Bulk Loading

```
import mysql.connector
from mysql.connector import Error
import pandas as pd
import csv
import tempfile

class MySQLLoader:
    def __init__(self, config):
        self.config = config
        self.connection = None

    def connect(self):
        """Establish database connection"""
        try:
            self.connection = mysql.connector.connect(**self.config)
            self.connection.autocommit = False
            logging.info("Connected to MySQL target")
        except Error as e:
            logging.error(f"Failed to connect to MySQL: {e}")
            raise

    def bulk_insert_csv(self, df, table_name, batch_size=10000):
        """Bulk insert using LOAD DATA INFILE"""
        cursor = self.connection.cursor()

        try:
            # Create temporary CSV file
            with tempfile.NamedTemporaryFile(mode='w', delete=False,
suffix='.csv') as temp_file:
                df.to_csv(temp_file.name, index=False, header=False)
                temp_file_path = temp_file.name

            # Use LOAD DATA INFILE for fast bulk insert
            load_query = f"""
                LOAD DATA INFILE '{temp_file_path}'
                INTO TABLE {table_name}
                FIELDS TERMINATED BY ','
                LINES TERMINATED BY '\n'
                IGNORE 1 LINES
            """

            cursor.execute(load_query)
            self.connection.commit()

            logging.info(f"Bulk inserted {len(df)} rows into {table_name}")

        except Error as e:
            self.connection.rollback()
```

```
            logging.error(f"Bulk insert failed: {e}")
            raise
        finally:
            cursor.close()
            # Clean up temp file
            import os
            if 'temp_file_path' in locals():
                os.unlink(temp_file_path)

    def batch_insert(self, df, table_name, batch_size=1000,
on_duplicate='IGNORE'):
        """Insert data in batches with duplicate handling"""
        cursor = self.connection.cursor()

        try:
            columns = ', '.join(df.columns)
            placeholders = ', '.join(['%s'] * len(df.columns))

            if on_duplicate == 'UPDATE':
                update_clause = ', '.join([f"{col} = VALUES({col})" for col in
df.columns])
                query = f"""
                    INSERT INTO {table_name} ({columns})
                    VALUES ({placeholders})
                    ON DUPLICATE KEY UPDATE {update_clause}
                """
            else:
                query = f"""
                    INSERT {on_duplicate} INTO {table_name} ({columns})
                    VALUES ({placeholders})
                """

            total_rows = len(df)
            inserted_rows = 0

            for start_idx in range(0, total_rows, batch_size):
                end_idx = min(start_idx + batch_size, total_rows)
                batch_data = df.iloc[start_idx:end_idx].values.tolist()

                cursor.executemany(query, batch_data)
                inserted_rows += len(batch_data)

                logging.info(f"Inserted batch {start_idx//batch_size + 1},
{inserted_rows}/{total_rows} rows")

            self.connection.commit()
            logging.info(f"Successfully inserted {inserted_rows} rows into
{table_name}")

        except Error as e:
            self.connection.rollback()
            logging.error(f"Batch insert failed: {e}")
            raise
        finally:
```

```
            cursor.close()

    def upsert_data(self, df, table_name, key_columns):
        """Upsert data (insert or update based on key columns)"""
        cursor = self.connection.cursor()

        try:
            columns = ', '.join(df.columns)
            placeholders = ', '.join(['%s'] * len(df.columns))

            # Create update clause for non-key columns
            non_key_columns = [col for col in df.columns if col not in
key_columns]
            update_clause = ', '.join([f"{col} = VALUES({col})" for col in
non_key_columns])

            query = f"""
                INSERT INTO {table_name} ({columns})
                VALUES ({placeholders})
                ON DUPLICATE KEY UPDATE {update_clause}
            """

            data = df.values.tolist()
            cursor.executemany(query, data)

            self.connection.commit()
            logging.info(f"Upserted {len(df)} rows into {table_name}")

        except Error as e:
            self.connection.rollback()
            logging.error(f"Upsert failed: {e}")
            raise
        finally:
            cursor.close()

    def create_staging_table(self, table_name, df):
        """Create staging table with same structure as DataFrame"""
        cursor = self.connection.cursor()

        try:
            # Drop staging table if exists
            cursor.execute(f"DROP TABLE IF EXISTS {table_name}_staging")

            # Create staging table structure
            column_definitions = []
            for col in df.columns:
                dtype = df[col].dtype

                if pd.api.types.is_integer_dtype(dtype):
                    sql_type = "BIGINT"
                elif pd.api.types.is_float_dtype(dtype):
                    sql_type = "DECIMAL(15,2)"
                elif pd.api.types.is_datetime64_any_dtype(dtype):
                    sql_type = "DATETIME"
```

```
                else:
                    sql_type = "TEXT"

                column_definitions.append(f"`{col}` {sql_type}")

            create_query = f"""
                CREATE TABLE {table_name}_staging (
                    {', '.join(column_definitions)}
                )
            """

            cursor.execute(create_query)
            self.connection.commit()

            logging.info(f"Created staging table {table_name}_staging")

        except Error as e:
            self.connection.rollback()
            logging.error(f"Failed to create staging table: {e}")
            raise
        finally:
            cursor.close()

    def close(self):
        """Close database connection"""
        if self.connection:
            self.connection.close()
            logging.info("MySQL connection closed")

# Usage
mysql_loader = MySQLLoader(mysql_config)
mysql_loader.connect()

# Create staging table
mysql_loader.create_staging_table('customers', customers_enhanced)

# Bulk insert into staging
mysql_loader.bulk_insert_csv(customers_enhanced, 'customers_staging')

# Upsert from staging to main table
mysql_loader.upsert_data(customers_enhanced, 'customers', ['id'])

mysql_loader.close()
```

PostgreSQL Bulk Loading

```
import psycopg2
from psycopg2.extras import execute_values
import pandas as pd
from io import StringIO
```

```python
class PostgreSQLLoader:
    def __init__(self, config):
        self.config = config
        self.connection = None

    def connect(self):
        """Establish database connection"""
        try:
            self.connection = psycopg2.connect(**self.config)
            self.connection.autocommit = False
            logging.info("Connected to PostgreSQL target")
        except Exception as e:
            logging.error(f"Failed to connect to PostgreSQL: {e}")
            raise

    def bulk_copy(self, df, table_name):
        """Bulk insert using COPY command"""
        cursor = self.connection.cursor()

        try:
            # Create CSV string buffer
            output = StringIO()
            df.to_csv(output, sep='\t', header=False, index=False, na_rep='\\N')
            output.seek(0)

            # Use COPY command for fast bulk insert
            cursor.copy_from(
                output,
                table_name,
                columns=df.columns.tolist(),
                sep='\t',
                null='\\N'
            )

            self.connection.commit()
            logging.info(f"Bulk copied {len(df)} rows into {table_name}")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"Bulk copy failed: {e}")
            raise
        finally:
            cursor.close()

    def batch_upsert(self, df, table_name, key_columns, batch_size=1000):
        """Batch upsert using ON CONFLICT"""
        cursor = self.connection.cursor()

        try:
            columns = df.columns.tolist()
            placeholders = ', '.join(['%s'] * len(columns))

            # Create conflict resolution clause
            non_key_columns = [col for col in columns if col not in key_columns]
```

```
            update_clause = ', '.join([f"{col} = EXCLUDED.{col}" for col in
non_key_columns])
            conflict_columns = ', '.join(key_columns)

            query = f"""
                INSERT INTO {table_name} ({', '.join(columns)})
                VALUES %s
                ON CONFLICT ({conflict_columns})
                DO UPDATE SET {update_clause}
            """

            total_rows = len(df)
            processed_rows = 0

            for start_idx in range(0, total_rows, batch_size):
                end_idx = min(start_idx + batch_size, total_rows)
                batch_data = df.iloc[start_idx:end_idx].values.tolist()

                execute_values(
                    cursor,
                    query,
                    batch_data,
                    template=None,
                    page_size=batch_size
                )

                processed_rows += len(batch_data)
                logging.info(f"Upserted batch {start_idx//batch_size + 1},
{processed_rows}/{total_rows} rows")

            self.connection.commit()
            logging.info(f"Successfully upserted {processed_rows} rows into
{table_name}")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"Batch upsert failed: {e}")
            raise
        finally:
            cursor.close()

    def create_temp_table(self, table_name, df):
        """Create temporary table for staging"""
        cursor = self.connection.cursor()

        try:
            # Drop temp table if exists
            cursor.execute(f"DROP TABLE IF EXISTS temp_{table_name}")

            # Get column definitions from main table
            cursor.execute(f"""
                SELECT column_name, data_type, character_maximum_length
                FROM information_schema.columns
                WHERE table_name = '{table_name}'
```

```
                ORDER BY ordinal_position
            """)

            columns_info = cursor.fetchall()

            if columns_info:
                # Use existing table structure
                cursor.execute(f"CREATE TEMP TABLE temp_{table_name} (LIKE
{table_name})")
            else:
                # Create based on DataFrame structure
                column_definitions = []
                for col in df.columns:
                    dtype = df[col].dtype

                    if pd.api.types.is_integer_dtype(dtype):
                        sql_type = "BIGINT"
                    elif pd.api.types.is_float_dtype(dtype):
                        sql_type = "DECIMAL"
                    elif pd.api.types.is_datetime64_any_dtype(dtype):
                        sql_type = "TIMESTAMP"
                    else:
                        sql_type = "TEXT"

                    column_definitions.append(f"{col} {sql_type}")

                create_query = f"""
                    CREATE TEMP TABLE temp_{table_name} (
                        {', '.join(column_definitions)}
                    )
                """

                cursor.execute(create_query)

            self.connection.commit()
            logging.info(f"Created temporary table temp_{table_name}")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"Failed to create temp table: {e}")
            raise
        finally:
            cursor.close()

    def merge_from_temp(self, table_name, key_columns):
        """Merge data from temp table to main table"""
        cursor = self.connection.cursor()

        try:
            # Get all columns from temp table
            cursor.execute(f"""
                SELECT column_name
                FROM information_schema.columns
                WHERE table_name = 'temp_{table_name}'
```

```
                ORDER BY ordinal_position
            """)

            columns = [row[0] for row in cursor.fetchall()]
            non_key_columns = [col for col in columns if col not in key_columns]

            # Create merge query
            merge_query = f"""
                INSERT INTO {table_name} ({', '.join(columns)})
                SELECT {', '.join(columns)}
                FROM temp_{table_name}
                ON CONFLICT ({', '.join(key_columns)})
                DO UPDATE SET
                {', '.join([f"{col} = EXCLUDED.{col}" for col in
non_key_columns])}
            """

            cursor.execute(merge_query)
            rows_affected = cursor.rowcount

            self.connection.commit()
            logging.info(f"Merged {rows_affected} rows from temp table to
{table_name}")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"Merge from temp table failed: {e}")
            raise
        finally:
            cursor.close()

    def close(self):
        """Close database connection"""
        if self.connection:
            self.connection.close()
            logging.info("PostgreSQL connection closed")

# Usage
postgres_loader = PostgreSQLLoader(postgres_config)
postgres_loader.connect()

# Create temp table and bulk load
postgres_loader.create_temp_table('orders', orders_final)
postgres_loader.bulk_copy(orders_final, 'temp_orders')
postgres_loader.merge_from_temp('orders', ['id'])

postgres_loader.close()
```

🏗 Dimensional Modeling

Star Schema Design

```sql
-- Fact Table: Sales
CREATE TABLE fact_sales (
    sale_id BIGINT PRIMARY KEY,
    date_key INT NOT NULL,
    customer_key INT NOT NULL,
    product_key INT NOT NULL,
    store_key INT NOT NULL,
    quantity INT NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,
    total_amount DECIMAL(12,2) NOT NULL,
    discount_amount DECIMAL(10,2) DEFAULT 0,
    tax_amount DECIMAL(10,2) DEFAULT 0,
    profit_amount DECIMAL(10,2),

    -- Foreign keys to dimension tables
    FOREIGN KEY (date_key) REFERENCES dim_date(date_key),
    FOREIGN KEY (customer_key) REFERENCES dim_customer(customer_key),
    FOREIGN KEY (product_key) REFERENCES dim_product(product_key),
    FOREIGN KEY (store_key) REFERENCES dim_store(store_key)
);

-- Dimension Table: Date
CREATE TABLE dim_date (
    date_key INT PRIMARY KEY,
    full_date DATE NOT NULL,
    day_of_week INT NOT NULL,
    day_name VARCHAR(10) NOT NULL,
    day_of_month INT NOT NULL,
    day_of_year INT NOT NULL,
    week_of_year INT NOT NULL,
    month_number INT NOT NULL,
    month_name VARCHAR(10) NOT NULL,
    quarter INT NOT NULL,
    year INT NOT NULL,
    is_weekend BOOLEAN NOT NULL,
    is_holiday BOOLEAN DEFAULT FALSE,
    fiscal_year INT,
    fiscal_quarter INT
);

-- Dimension Table: Customer
CREATE TABLE dim_customer (
    customer_key INT PRIMARY KEY,
    customer_id VARCHAR(50) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    full_name VARCHAR(200),
    email VARCHAR(255),
    phone VARCHAR(20),
    birth_date DATE,
    age_group VARCHAR(20),
    gender VARCHAR(10),
```

```
    address_line1 VARCHAR(255),
    address_line2 VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(50),
    postal_code VARCHAR(20),
    country VARCHAR(50),
    customer_segment VARCHAR(50),
    registration_date DATE,
    is_active BOOLEAN DEFAULT TRUE,

    -- SCD Type 2 fields
    effective_date DATE NOT NULL,
    expiry_date DATE,
    is_current BOOLEAN DEFAULT TRUE,

    UNIQUE(customer_id, effective_date)
);

-- Dimension Table: Product
CREATE TABLE dim_product (
    product_key INT PRIMARY KEY,
    product_id VARCHAR(50) NOT NULL,
    product_name VARCHAR(255) NOT NULL,
    product_description TEXT,
    sku VARCHAR(100),
    brand VARCHAR(100),
    category VARCHAR(100),
    subcategory VARCHAR(100),
    product_line VARCHAR(100),
    size VARCHAR(50),
    color VARCHAR(50),
    weight DECIMAL(8,2),
    unit_cost DECIMAL(10,2),
    list_price DECIMAL(10,2),
    supplier_name VARCHAR(255),
    is_active BOOLEAN DEFAULT TRUE,

    -- SCD Type 2 fields
    effective_date DATE NOT NULL,
    expiry_date DATE,
    is_current BOOLEAN DEFAULT TRUE,

    UNIQUE(product_id, effective_date)
);

-- Dimension Table: Store
CREATE TABLE dim_store (
    store_key INT PRIMARY KEY,
    store_id VARCHAR(50) NOT NULL,
    store_name VARCHAR(255) NOT NULL,
    store_type VARCHAR(50),
    address_line1 VARCHAR(255),
    address_line2 VARCHAR(255),
    city VARCHAR(100),
```

```
    state VARCHAR(50),
    postal_code VARCHAR(20),
    country VARCHAR(50),
    region VARCHAR(100),
    district VARCHAR(100),
    manager_name VARCHAR(255),
    phone VARCHAR(20),
    email VARCHAR(255),
    opening_date DATE,
    store_size_sqft INT,
    is_active BOOLEAN DEFAULT TRUE,

    -- SCD Type 2 fields
    effective_date DATE NOT NULL,
    expiry_date DATE,
    is_current BOOLEAN DEFAULT TRUE,

    UNIQUE(store_id, effective_date)
);

-- Create indexes for performance
CREATE INDEX idx_fact_sales_date ON fact_sales(date_key);
CREATE INDEX idx_fact_sales_customer ON fact_sales(customer_key);
CREATE INDEX idx_fact_sales_product ON fact_sales(product_key);
CREATE INDEX idx_fact_sales_store ON fact_sales(store_key);
CREATE INDEX idx_fact_sales_amount ON fact_sales(total_amount);

CREATE INDEX idx_dim_date_full_date ON dim_date(full_date);
CREATE INDEX idx_dim_date_year_month ON dim_date(year, month_number);

CREATE INDEX idx_dim_customer_id ON dim_customer(customer_id);
CREATE INDEX idx_dim_customer_current ON dim_customer(is_current);
CREATE INDEX idx_dim_customer_segment ON dim_customer(customer_segment);

CREATE INDEX idx_dim_product_id ON dim_product(product_id);
CREATE INDEX idx_dim_product_current ON dim_product(is_current);
CREATE INDEX idx_dim_product_category ON dim_product(category, subcategory);

CREATE INDEX idx_dim_store_id ON dim_store(store_id);
CREATE INDEX idx_dim_store_current ON dim_store(is_current);
CREATE INDEX idx_dim_store_region ON dim_store(region, district);
```

Slowly Changing Dimensions (SCD)

```
class SCDManager:
    def __init__(self, connection):
        self.connection = connection

    def handle_scd_type1(self, table_name, new_data, key_column):
        """Handle SCD Type 1 - Overwrite existing data"""
        cursor = self.connection.cursor()
```

```
        try:
            for _, row in new_data.iterrows():
                # Get non-key columns for update
                columns = [col for col in new_data.columns if col != key_column]
                set_clause = ', '.join([f"{col} = %s" for col in columns])

                update_query = f"""
                    UPDATE {table_name}
                    SET {set_clause}
                    WHERE {key_column} = %s
                """

                values = [row[col] for col in columns] + [row[key_column]]
                cursor.execute(update_query, values)

            self.connection.commit()
            logging.info(f"Updated {len(new_data)} records in {table_name} (SCD
Type 1)")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"SCD Type 1 update failed: {e}")
            raise
        finally:
            cursor.close()

    def handle_scd_type2(self, table_name, new_data, key_column, effective_date):
        """Handle SCD Type 2 - Keep history with versioning"""
        cursor = self.connection.cursor()

        try:
            for _, row in new_data.iterrows():
                # Check if record exists and is current
                cursor.execute(f"""
                    SELECT * FROM {table_name}
                    WHERE {key_column} = %s AND is_current = TRUE
                """, (row[key_column],))

                existing_record = cursor.fetchone()

                if existing_record:
                    # Expire the current record
                    cursor.execute(f"""
                        UPDATE {table_name}
                        SET expiry_date = %s, is_current = FALSE
                        WHERE {key_column} = %s AND is_current = TRUE
                    """, (effective_date, row[key_column]))

                # Insert new version
                columns = list(new_data.columns) + ['effective_date',
'expiry_date', 'is_current']
                placeholders = ', '.join(['%s'] * len(columns))
```

```
                insert_query = f"""
                    INSERT INTO {table_name} ({', '.join(columns)})
                    VALUES ({placeholders})
                """

                values = list(row.values) + [effective_date, None, True]
                cursor.execute(insert_query, values)

            self.connection.commit()
            logging.info(f"Processed {len(new_data)} records in {table_name} (SCD
Type 2)")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"SCD Type 2 processing failed: {e}")
            raise
        finally:
            cursor.close()

    def handle_scd_type3(self, table_name, new_data, key_column, tracked_columns):
        """Handle SCD Type 3 - Keep limited history in same record"""
        cursor = self.connection.cursor()

        try:
            for _, row in new_data.iterrows():
                # For each tracked column, move current to previous and update
current
                set_clauses = []
                values = []

                for col in tracked_columns:
                    set_clauses.append(f"previous_{col} = {col}")
                    set_clauses.append(f"{col} = %s")
                    values.append(row[col])

                # Add other non-tracked columns
                other_columns = [col for col in new_data.columns
                                 if col not in tracked_columns and col !=
key_column]

                for col in other_columns:
                    set_clauses.append(f"{col} = %s")
                    values.append(row[col])

                values.append(row[key_column])

                update_query = f"""
                    UPDATE {table_name}
                    SET {', '.join(set_clauses)}
                    WHERE {key_column} = %s
                """

                cursor.execute(update_query, values)
```

```
            self.connection.commit()
            logging.info(f"Updated {len(new_data)} records in {table_name} (SCD
Type 3)")

        except Exception as e:
            self.connection.rollback()
            logging.error(f"SCD Type 3 update failed: {e}")
            raise
        finally:
            cursor.close()

# Usage
scd_manager = SCDManager(postgres_loader.connection)

# Handle customer dimension changes
customer_changes = pd.DataFrame([
    {'customer_id': 'CUST001', 'email': 'newemail@example.com', 'city': 'New
York'},
    {'customer_id': 'CUST002', 'phone': '+1-555-0199', 'address_line1': '456 Oak
St'}
])

# Type 2 SCD for customer changes (keep history)
scd_manager.handle_scd_type2(
    'dim_customer',
    customer_changes,
    'customer_id',
    datetime.now().date()
)
```

🔄 ETL Pipeline Orchestration

Complete ETL Pipeline

```
import schedule
import time
from datetime import datetime, timedelta
import logging
from dataclasses import dataclass
from typing import List, Dict, Any

@dataclass
class ETLConfig:
    source_configs: Dict[str, Any]
    target_config: Dict[str, Any]
    transformation_rules: Dict[str, Any]
    schedule_config: Dict[str, Any]
    notification_config: Dict[str, Any]

class ETLPipeline:
```

```
    def __init__(self, config: ETLConfig):
        self.config = config
        self.extractors = {}
        self.transformers = {}
        self.loaders = {}
        self.last_run_times = {}

        self._initialize_components()

    def _initialize_components(self):
        """Initialize ETL components"""
        # Initialize extractors
        for source_name, source_config in self.config.source_configs.items():
            if source_config['type'] == 'mysql':
                self.extractors[source_name] =
MySQLExtractor(source_config['connection'])
            elif source_config['type'] == 'postgresql':
                self.extractors[source_name] =
PostgreSQLExtractor(source_config['connection'])
            elif source_config['type'] == 'api':
                self.extractors[source_name] = APIExtractor(
                    source_config['base_url'],
                    source_config.get('api_key')
                )
            elif source_config['type'] == 'file':
                self.extractors[source_name] =
FileExtractor(source_config['base_path'])

        # Initialize transformer
        self.transformer = DataTransformer()

        # Initialize loader
        if self.config.target_config['type'] == 'mysql':
            self.loader = MySQLLoader(self.config.target_config['connection'])
        elif self.config.target_config['type'] == 'postgresql':
            self.loader =
PostgreSQLLoader(self.config.target_config['connection'])

    def extract_data(self, source_name: str, extraction_config: Dict) ->
pd.DataFrame:
        """Extract data from specified source"""
        extractor = self.extractors[source_name]

        try:
            if extraction_config['method'] == 'incremental':
                last_run = self.last_run_times.get(source_name, datetime.now() -
timedelta(days=1))
                data = extractor.extract_incremental(
                    extraction_config['table'],
                    extraction_config['timestamp_column'],
                    last_run
                )
            elif extraction_config['method'] == 'full':
                data = extractor.extract_full(extraction_config['table'])
```

```
            elif extraction_config['method'] == 'custom':
                data = extractor.extract_with_joins(extraction_config['query'])

            logging.info(f"Extracted {len(data)} rows from {source_name}")
            return data

        except Exception as e:
            logging.error(f"Extraction failed for {source_name}: {e}")
            raise

    def transform_data(self, data: pd.DataFrame, transformation_rules: Dict) ->
pd.DataFrame:
        """Apply transformations to data"""
        try:
            transformed_data = data.copy()

            # Apply data cleaning
            if 'clean_text' in transformation_rules:
                transformed_data = self.transformer.clean_text_data(
                    transformed_data,
                    transformation_rules['clean_text']
                )

            # Handle missing values
            if 'missing_values' in transformation_rules:
                transformed_data = self.transformer.handle_missing_values(
                    transformed_data,
                    transformation_rules['missing_values']
                )

            # Create derived columns
            if 'derived_columns' in transformation_rules:
                transformed_data = self.transformer.create_derived_columns(
                    transformed_data,
                    transformation_rules['derived_columns']
                )

            # Data type conversions
            if 'data_types' in transformation_rules:
                converter = DataTypeConverter()
                transformed_data = converter.convert_data_types(
                    transformed_data,
                    transformation_rules['data_types']
                )

            logging.info(f"Transformed data: {len(transformed_data)} rows")
            return transformed_data

        except Exception as e:
            logging.error(f"Transformation failed: {e}")
            raise

    def load_data(self, data: pd.DataFrame, load_config: Dict):
        """Load data to target system"""
```

```
        try:
            self.loader.connect()

            if load_config['method'] == 'bulk_insert':
                self.loader.bulk_insert_csv(data, load_config['table'])
            elif load_config['method'] == 'batch_insert':
                self.loader.batch_insert(
                    data,
                    load_config['table'],
                    batch_size=load_config.get('batch_size', 1000)
                )
            elif load_config['method'] == 'upsert':
                self.loader.upsert_data(
                    data,
                    load_config['table'],
                    load_config['key_columns']
                )

            logging.info(f"Loaded {len(data)} rows to {load_config['table']}")

        except Exception as e:
            logging.error(f"Loading failed: {e}")
            raise
        finally:
            self.loader.close()

    def run_pipeline(self, pipeline_name: str):
        """Run complete ETL pipeline"""
        start_time = datetime.now()
        logging.info(f"Starting ETL pipeline: {pipeline_name}")

        try:
            pipeline_config = self.config.transformation_rules[pipeline_name]

            # Extract phase
            all_data = {}
            for source_name, extraction_config in
pipeline_config['sources'].items():
                data = self.extract_data(source_name, extraction_config)
                all_data[source_name] = data

            # Transform phase
            if 'joins' in pipeline_config:
                # Handle data joins
                transformed_data = self._join_data(all_data,
pipeline_config['joins'])
            else:
                # Single source transformation
                source_name = list(all_data.keys())[0]
                transformed_data = all_data[source_name]

            # Apply transformations
            if 'transformations' in pipeline_config:
                transformed_data = self.transform_data(
```

```
                    transformed_data,
                    pipeline_config['transformations']
                )

            # Load phase
            self.load_data(transformed_data, pipeline_config['target'])

            # Update last run time
            self.last_run_times[pipeline_name] = start_time

            duration = datetime.now() - start_time
            logging.info(f"ETL pipeline {pipeline_name} completed in {duration}")

            # Send success notification
            self._send_notification(
                f"ETL Success: {pipeline_name}",
                f"Pipeline completed successfully. Processed
{len(transformed_data)} rows in {duration}"
            )

        except Exception as e:
            duration = datetime.now() - start_time
            logging.error(f"ETL pipeline {pipeline_name} failed after {duration}:
{e}")

            # Send failure notification
            self._send_notification(
                f"ETL Failure: {pipeline_name}",
                f"Pipeline failed after {duration}. Error: {str(e)}"
            )
            raise

    def _join_data(self, data_sources: Dict[str, pd.DataFrame], join_config:
List[Dict]) -> pd.DataFrame:
        """Join multiple data sources"""
        result_data = None

        for join in join_config:
            left_source = join['left']
            right_source = join['right']

            if result_data is None:
                left_df = data_sources[left_source]
            else:
                left_df = result_data

            right_df = data_sources[right_source]

            result_data = left_df.merge(
                right_df,
                left_on=join['left_on'],
                right_on=join['right_on'],
                how=join.get('how', 'inner'),
                suffixes=join.get('suffixes', ('', '_right'))
```

```
            )

        return result_data

    def _send_notification(self, subject: str, message: str):
        """Send notification about pipeline status"""
        # Implementation depends on notification method (email, Slack, etc.)
        logging.info(f"Notification: {subject} - {message}")

    def schedule_pipelines(self):
        """Schedule ETL pipelines"""
        for pipeline_name, schedule_config in self.config.schedule_config.items():
            if schedule_config['type'] == 'daily':
                schedule.every().day.at(schedule_config['time']).do(
                    self.run_pipeline, pipeline_name
                )
            elif schedule_config['type'] == 'hourly':
                schedule.every().hour.do(self.run_pipeline, pipeline_name)
            elif schedule_config['type'] == 'weekly':
                getattr(schedule.every(), schedule_config['day']).at(
                    schedule_config['time']
                ).do(self.run_pipeline, pipeline_name)

        logging.info("ETL pipelines scheduled")

        # Keep the scheduler running
        while True:
            schedule.run_pending()
            time.sleep(60)  # Check every minute

# ETL Configuration Example
etl_config = ETLConfig(
    source_configs={
        'ecommerce_mysql': {
            'type': 'mysql',
            'connection': {
                'host': 'localhost',
                'user': 'etl_user',
                'password': 'password',
                'database': 'ecommerce'
            }
        },
        'analytics_api': {
            'type': 'api',
            'base_url': 'https://api.analytics.com/v1',
            'api_key': 'your_api_key'
        }
    },
    target_config={
        'type': 'postgresql',
        'connection': {
            'host': 'warehouse.company.com',
            'port': 5432,
            'user': 'warehouse_user',
```

```
            'password': 'warehouse_password',
            'database': 'data_warehouse'
        }
    },
    transformation_rules={
        'daily_sales_etl': {
            'sources': {
                'ecommerce_mysql': {
                    'method': 'incremental',
                    'table': 'orders',
                    'timestamp_column': 'created_at'
                }
            },
            'transformations': {
                'clean_text': ['customer_name', 'product_name'],
                'missing_values': {
                    'discount_amount': 0,
                    'tax_amount': 'mean'
                },
                'derived_columns': {
                    'profit_margin': lambda df: (df['total_amount'] -
df['cost_amount']) / df['total_amount'] * 100,
                    'order_month': lambda df:
pd.to_datetime(df['created_at']).dt.to_period('M')
                },
                'data_types': {
                    'created_at': 'datetime',
                    'total_amount': 'numeric'
                }
            },
            'target': {
                'method': 'upsert',
                'table': 'fact_sales',
                'key_columns': ['order_id']
            }
        }
    },
    schedule_config={
        'daily_sales_etl': {
            'type': 'daily',
            'time': '02:00'
        }
    },
    notification_config={
        'email': {
            'smtp_server': 'smtp.company.com',
            'recipients': ['data-team@company.com']
        }
    }
)

# Usage
etl_pipeline = ETLPipeline(etl_config)
```

```
# Run single pipeline
etl_pipeline.run_pipeline('daily_sales_etl')

# Schedule all pipelines
# etl_pipeline.schedule_pipelines()
```

📊 Data Quality and Monitoring

Data Quality Checks

```
class DataQualityChecker:
    def __init__(self):
        self.quality_rules = {}
        self.quality_results = []

    def add_rule(self, rule_name: str, rule_function, severity='ERROR'):
        """Add data quality rule"""
        self.quality_rules[rule_name] = {
            'function': rule_function,
            'severity': severity
        }

    def check_completeness(self, df: pd.DataFrame, required_columns: List[str]) ->
Dict:
        """Check data completeness"""
        results = {}

        for column in required_columns:
            if column in df.columns:
                null_count = df[column].isnull().sum()
                null_percentage = (null_count / len(df)) * 100

                results[f"{column}_completeness"] = {
                    'passed': null_count == 0,
                    'null_count': null_count,
                    'null_percentage': null_percentage,
                    'message': f"{column} has {null_count} null values
({null_percentage:.2f}%)"
                }

        return results

    def check_uniqueness(self, df: pd.DataFrame, unique_columns: List[str]) ->
Dict:
        """Check data uniqueness"""
        results = {}

        for column in unique_columns:
            if column in df.columns:
                total_count = len(df)
```

```
                unique_count = df[column].nunique()
                duplicate_count = total_count - unique_count

                results[f"{column}_uniqueness"] = {
                    'passed': duplicate_count == 0,
                    'duplicate_count': duplicate_count,
                    'uniqueness_percentage': (unique_count / total_count) * 100,
                    'message': f"{column} has {duplicate_count} duplicate values"
                }

        return results

    def check_data_types(self, df: pd.DataFrame, expected_types: Dict[str, str]) -
> Dict:
        """Check data types"""
        results = {}

        for column, expected_type in expected_types.items():
            if column in df.columns:
                actual_type = str(df[column].dtype)
                type_match = self._types_match(actual_type, expected_type)

                results[f"{column}_data_type"] = {
                    'passed': type_match,
                    'expected_type': expected_type,
                    'actual_type': actual_type,
                    'message': f"{column} type mismatch: expected {expected_type},
got {actual_type}"
                }

        return results

    def check_value_ranges(self, df: pd.DataFrame, range_rules: Dict[str, Dict]) -
> Dict:
        """Check value ranges"""
        results = {}

        for column, rule in range_rules.items():
            if column in df.columns:
                if 'min' in rule:
                    below_min = (df[column] < rule['min']).sum()
                    results[f"{column}_min_range"] = {
                        'passed': below_min == 0,
                        'violations': below_min,
                        'message': f"{column} has {below_min} values below minimum
{rule['min']}"
                    }

                if 'max' in rule:
                    above_max = (df[column] > rule['max']).sum()
                    results[f"{column}_max_range"] = {
                        'passed': above_max == 0,
                        'violations': above_max,
                        'message': f"{column} has {above_max} values above maximum
```

```
{rule['max']}"
                    }

        return results

    def check_referential_integrity(self, df: pd.DataFrame, reference_data:
Dict[str, pd.DataFrame]) -> Dict:
        """Check referential integrity"""
        results = {}

        for column, ref_df in reference_data.items():
            if column in df.columns:
                ref_column = ref_df.columns[0]  # Assume first column is the key
                valid_values = set(ref_df[ref_column].values)
                df_values = set(df[column].dropna().values)

                invalid_values = df_values - valid_values
                invalid_count = len(invalid_values)

                results[f"{column}_referential_integrity"] = {
                    'passed': invalid_count == 0,
                    'invalid_count': invalid_count,
                    'invalid_values': list(invalid_values)[:10],  # Show first 10
                    'message': f"{column} has {invalid_count} invalid reference
values"
                }

        return results

    def run_quality_checks(self, df: pd.DataFrame, rules_config: Dict) -> Dict:
        """Run all configured quality checks"""
        all_results = {}

        if 'completeness' in rules_config:
            completeness_results = self.check_completeness(df,
rules_config['completeness'])
            all_results.update(completeness_results)

        if 'uniqueness' in rules_config:
            uniqueness_results = self.check_uniqueness(df,
rules_config['uniqueness'])
            all_results.update(uniqueness_results)

        if 'data_types' in rules_config:
            type_results = self.check_data_types(df, rules_config['data_types'])
            all_results.update(type_results)

        if 'value_ranges' in rules_config:
            range_results = self.check_value_ranges(df,
rules_config['value_ranges'])
            all_results.update(range_results)

        if 'referential_integrity' in rules_config:
            ref_results = self.check_referential_integrity(df,
```

```
rules_config['referential_integrity'])
            all_results.update(ref_results)

        # Run custom rules
        for rule_name, rule_config in self.quality_rules.items():
            try:
                custom_result = rule_config['function'](df)
                all_results[rule_name] = {
                    'passed': custom_result,
                    'severity': rule_config['severity'],
                    'message': f"Custom rule {rule_name}: {'PASSED' if
custom_result else 'FAILED'}"
                }
            except Exception as e:
                all_results[rule_name] = {
                    'passed': False,
                    'severity': 'ERROR',
                    'message': f"Custom rule {rule_name} failed with error:
{str(e)}"
                }

        return all_results

    def _types_match(self, actual_type: str, expected_type: str) -> bool:
        """Check if data types match"""
        type_mappings = {
            'int': ['int64', 'int32', 'int16', 'int8'],
            'float': ['float64', 'float32'],
            'string': ['object', 'string'],
            'datetime': ['datetime64[ns]', 'datetime64'],
            'bool': ['bool']
        }

        if expected_type in type_mappings:
            return actual_type in type_mappings[expected_type]

        return actual_type == expected_type

    def generate_quality_report(self, results: Dict) -> str:
        """Generate data quality report"""
        report = ["\n=== DATA QUALITY REPORT ==="]

        passed_count = sum(1 for result in results.values() if
result.get('passed', False))
        total_count = len(results)

        report.append(f"\nOverall: {passed_count}/{total_count} checks passed
({passed_count/total_count*100:.1f}%)")

        # Group by severity
        errors = []
        warnings = []
        passed = []
```

```
        for check_name, result in results.items():
            if result.get('passed', False):
                passed.append((check_name, result))
            elif result.get('severity', 'ERROR') == 'ERROR':
                errors.append((check_name, result))
            else:
                warnings.append((check_name, result))

        if errors:
            report.append("\n🟢 ERRORS:")
            for check_name, result in errors:
                report.append(f"  - {check_name}: {result['message']}")

        if warnings:
            report.append("\n🟡 WARNINGS:")
            for check_name, result in warnings:
                report.append(f"  - {check_name}: {result['message']}")

        if passed:
            report.append(f"\n🟢 PASSED: {len(passed)} checks")

        return "\n".join(report)

# Usage
quality_checker = DataQualityChecker()

# Add custom rule
quality_checker.add_rule(
    'positive_amounts',
    lambda df: (df['total_amount'] > 0).all(),
    severity='ERROR'
)

# Define quality rules
quality_rules = {
    'completeness': ['customer_id', 'order_date', 'total_amount'],
    'uniqueness': ['order_id'],
    'data_types': {
        'customer_id': 'int',
        'order_date': 'datetime',
        'total_amount': 'float'
    },
    'value_ranges': {
        'total_amount': {'min': 0, 'max': 10000},
        'quantity': {'min': 1, 'max': 100}
    }
}

# Run quality checks
quality_results = quality_checker.run_quality_checks(orders_df, quality_rules)

# Generate report
report = quality_checker.generate_quality_report(quality_results)
print(report)
```

Data Warehousing Concepts

Star Schema Design

```sql
-- Fact Table: Sales
CREATE TABLE fact_sales (
    sale_id BIGINT PRIMARY KEY,
    date_key INT NOT NULL,
    customer_key INT NOT NULL,
    product_key INT NOT NULL,
    store_key INT NOT NULL,
    quantity INT NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,
    total_amount DECIMAL(12,2) NOT NULL,
    discount_amount DECIMAL(10,2) DEFAULT 0,
    tax_amount DECIMAL(10,2) NOT NULL,
    profit_amount DECIMAL(10,2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Foreign Keys
    FOREIGN KEY (date_key) REFERENCES dim_date(date_key),
    FOREIGN KEY (customer_key) REFERENCES dim_customer(customer_key),
    FOREIGN KEY (product_key) REFERENCES dim_product(product_key),
    FOREIGN KEY (store_key) REFERENCES dim_store(store_key)
);

-- Dimension Table: Date
CREATE TABLE dim_date (
    date_key INT PRIMARY KEY,
    full_date DATE NOT NULL,
    day_of_week VARCHAR(10),
    day_of_month INT,
    day_of_year INT,
    week_of_year INT,
    month_name VARCHAR(10),
    month_number INT,
    quarter INT,
    year INT,
    is_weekend BOOLEAN,
    is_holiday BOOLEAN,
    fiscal_year INT,
    fiscal_quarter INT
);

-- Dimension Table: Customer
CREATE TABLE dim_customer (
    customer_key INT PRIMARY KEY,
    customer_id VARCHAR(50) NOT NULL,
    first_name VARCHAR(100),
```

```
    last_name VARCHAR(100),
    email VARCHAR(255),
    phone VARCHAR(20),
    birth_date DATE,
    gender VARCHAR(10),
    customer_segment VARCHAR(50),
    registration_date DATE,
    city VARCHAR(100),
    state VARCHAR(100),
    country VARCHAR(100),
    postal_code VARCHAR(20),
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Dimension Table: Product
CREATE TABLE dim_product (
    product_key INT PRIMARY KEY,
    product_id VARCHAR(50) NOT NULL,
    product_name VARCHAR(255) NOT NULL,
    brand VARCHAR(100),
    category VARCHAR(100),
    subcategory VARCHAR(100),
    product_line VARCHAR(100),
    unit_cost DECIMAL(10,2),
    unit_price DECIMAL(10,2),
    product_size VARCHAR(50),
    product_color VARCHAR(50),
    product_weight DECIMAL(8,2),
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Dimension Table: Store
CREATE TABLE dim_store (
    store_key INT PRIMARY KEY,
    store_id VARCHAR(50) NOT NULL,
    store_name VARCHAR(255) NOT NULL,
    store_type VARCHAR(50),
    manager_name VARCHAR(100),
    address VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(100),
    country VARCHAR(100),
    postal_code VARCHAR(20),
    phone VARCHAR(20),
    opening_date DATE,
    store_size_sqft INT,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Snowflake Schema Example

```sql
-- Normalized Product Dimension
CREATE TABLE dim_product_category (
    category_key INT PRIMARY KEY,
    category_name VARCHAR(100) NOT NULL,
    category_description TEXT,
    department VARCHAR(100)
);

CREATE TABLE dim_product_brand (
    brand_key INT PRIMARY KEY,
    brand_name VARCHAR(100) NOT NULL,
    brand_description TEXT,
    country_of_origin VARCHAR(100)
);

CREATE TABLE dim_product_normalized (
    product_key INT PRIMARY KEY,
    product_id VARCHAR(50) NOT NULL,
    product_name VARCHAR(255) NOT NULL,
    category_key INT,
    brand_key INT,
    unit_cost DECIMAL(10,2),
    unit_price DECIMAL(10,2),
    is_active BOOLEAN DEFAULT TRUE,

    FOREIGN KEY (category_key) REFERENCES dim_product_category(category_key),
    FOREIGN KEY (brand_key) REFERENCES dim_product_brand(brand_key)
);
```

Slowly Changing Dimensions (SCD)

```sql
-- SCD Type 1: Overwrite
UPDATE dim_customer
SET
    email = 'new_email@example.com',
    updated_at = CURRENT_TIMESTAMP
WHERE customer_key = 12345;

-- SCD Type 2: Add New Record
CREATE TABLE dim_customer_scd2 (
    customer_key INT PRIMARY KEY,
    customer_id VARCHAR(50) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    email VARCHAR(255),
    customer_segment VARCHAR(50),
```

```
    effective_date DATE NOT NULL,
    expiration_date DATE,
    is_current BOOLEAN DEFAULT TRUE,
    version_number INT DEFAULT 1
);

-- Insert new version for SCD Type 2
INSERT INTO dim_customer_scd2 (
    customer_key, customer_id, first_name, last_name,
    email, customer_segment, effective_date, is_current, version_number
)
SELECT
    (SELECT MAX(customer_key) + 1 FROM dim_customer_scd2),
    customer_id,
    first_name,
    last_name,
    'updated_email@example.com',
    'Premium',
    CURRENT_DATE,
    TRUE,
    (SELECT MAX(version_number) + 1 FROM dim_customer_scd2 WHERE customer_id =
'CUST001')
FROM dim_customer_scd2
WHERE customer_id = 'CUST001' AND is_current = TRUE;

-- Update previous version
UPDATE dim_customer_scd2
SET
    expiration_date = CURRENT_DATE - INTERVAL '1 day',
    is_current = FALSE
WHERE customer_id = 'CUST001'
  AND is_current = TRUE
  AND customer_key != (SELECT MAX(customer_key) FROM dim_customer_scd2 WHERE
customer_id = 'CUST001');

-- SCD Type 3: Add New Column
ALTER TABLE dim_customer
ADD COLUMN previous_segment VARCHAR(50),
ADD COLUMN segment_change_date DATE;

UPDATE dim_customer
SET
    previous_segment = customer_segment,
    customer_segment = 'Premium',
    segment_change_date = CURRENT_DATE
WHERE customer_key = 12345;
```

Data Mart Creation

```
-- Sales Data Mart
CREATE VIEW sales_data_mart AS
```

```sql
SELECT
    fs.sale_id,
    dd.full_date,
    dd.year,
    dd.quarter,
    dd.month_name,
    dc.customer_segment,
    dc.city as customer_city,
    dc.country as customer_country,
    dp.product_name,
    dp.category,
    dp.brand,
    ds.store_name,
    ds.store_type,
    ds.city as store_city,
    fs.quantity,
    fs.unit_price,
    fs.total_amount,
    fs.discount_amount,
    fs.profit_amount,

    -- Calculated measures
    fs.total_amount - fs.discount_amount as net_sales,
    fs.profit_amount / NULLIF(fs.total_amount, 0) * 100 as profit_margin_pct,

    -- Running totals
    SUM(fs.total_amount) OVER (
        PARTITION BY dd.year, dd.month_number
        ORDER BY dd.full_date
        ROWS UNBOUNDED PRECEDING
    ) as running_monthly_sales

FROM fact_sales fs
JOIN dim_date dd ON fs.date_key = dd.date_key
JOIN dim_customer dc ON fs.customer_key = dc.customer_key
JOIN dim_product dp ON fs.product_key = dp.product_key
JOIN dim_store ds ON fs.store_key = ds.store_key
WHERE dd.year >= EXTRACT(YEAR FROM CURRENT_DATE) - 2;

-- Customer Analytics Data Mart
CREATE VIEW customer_analytics_mart AS
SELECT
    dc.customer_key,
    dc.customer_id,
    dc.customer_segment,
    dc.city,
    dc.country,

    -- Customer metrics
    COUNT(DISTINCT fs.sale_id) as total_orders,
    SUM(fs.total_amount) as lifetime_value,
    AVG(fs.total_amount) as avg_order_value,
    MAX(dd.full_date) as last_purchase_date,
    MIN(dd.full_date) as first_purchase_date,
```

```
    -- Customer behavior
    COUNT(DISTINCT dd.year || '-' || dd.month_number) as active_months,
    COUNT(DISTINCT dp.category) as categories_purchased,

    -- Recency, Frequency, Monetary (RFM)
    CURRENT_DATE - MAX(dd.full_date) as recency_days,
    COUNT(DISTINCT fs.sale_id) as frequency,
    SUM(fs.total_amount) as monetary_value,

    -- Customer classification
    CASE
        WHEN SUM(fs.total_amount) > 10000 THEN 'High Value'
        WHEN SUM(fs.total_amount) > 5000 THEN 'Medium Value'
        ELSE 'Low Value'
    END as value_segment

FROM dim_customer dc
LEFT JOIN fact_sales fs ON dc.customer_key = fs.customer_key
LEFT JOIN dim_date dd ON fs.date_key = dd.date_key
WHERE dc.is_active = TRUE
GROUP BY
    dc.customer_key, dc.customer_id, dc.customer_segment,
    dc.city, dc.country;
```

🎯 Real-World ETL Project: E-commerce Analytics

Project Overview

Build a complete ETL pipeline for an e-commerce company that:

- Extracts data from multiple sources (MySQL, APIs, CSV files)
- Transforms data for analytics
- Loads into a PostgreSQL data warehouse
- Implements data quality checks
- Provides automated reporting

Implementation

```
# Complete E-commerce ETL Project
class EcommerceETLProject:
    def __init__(self):
        self.config = self._load_config()
        self.logger = self._setup_logging()
        self.quality_checker = DataQualityChecker()
        self.metrics = ETLMetrics()

    def _load_config(self):
        return {
            'sources': {
```

```
                'orders_db': {
                    'type': 'mysql',
                    'host': 'prod-mysql.company.com',
                    'database': 'ecommerce',
                    'tables': ['orders', 'order_items', 'customers', 'products']
                },
                'web_analytics': {
                    'type': 'api',
                    'base_url': 'https://api.analytics.com/v1',
                    'endpoints': ['sessions', 'page_views', 'conversions']
                },
                'inventory_files': {
                    'type': 'file',
                    'path': '/data/inventory/',
                    'format': 'csv'
                }
            },
            'target': {
                'type': 'postgresql',
                'host': 'warehouse.company.com',
                'database': 'analytics_dw'
            },
            'schedule': {
                'daily_sales': '02:00',
                'hourly_inventory': '*/1 * * * *',
                'weekly_customer_analysis': '0 6 * * 1'
            }
        }

    def extract_orders_data(self, start_date: str, end_date: str):
        """Extract orders data from MySQL"""
        query = """
        SELECT
            o.order_id,
            o.customer_id,
            o.order_date,
            o.status,
            o.total_amount,
            o.shipping_cost,
            o.tax_amount,
            oi.product_id,
            oi.quantity,
            oi.unit_price,
            oi.discount_amount,
            c.email,
            c.registration_date,
            c.customer_segment,
            p.product_name,
            p.category,
            p.brand
        FROM orders o
        JOIN order_items oi ON o.order_id = oi.order_id
        JOIN customers c ON o.customer_id = c.customer_id
        JOIN products p ON oi.product_id = p.product_id
```

```
        WHERE o.order_date BETWEEN %s AND %s
        """

        extractor = MySQLExtractor(self.config['sources']['orders_db'])
        return extractor.execute_query(query, (start_date, end_date))

    def extract_web_analytics(self, date: str):
        """Extract web analytics data from API"""
        api_extractor = APIExtractor(self.config['sources']['web_analytics'])

        sessions = api_extractor.get_data(f'sessions?date={date}')
        page_views = api_extractor.get_data(f'page_views?date={date}')
        conversions = api_extractor.get_data(f'conversions?date={date}')

        return {
            'sessions': pd.DataFrame(sessions),
            'page_views': pd.DataFrame(page_views),
            'conversions': pd.DataFrame(conversions)
        }

    def transform_sales_data(self, orders_df: pd.DataFrame):
        """Transform orders data for analytics"""
        transformer = DataTransformer()

        # Data cleaning
        orders_df = transformer.clean_text_data(orders_df, ['email',
'product_name'])
        orders_df = transformer.handle_missing_values(orders_df, {
            'discount_amount': 0,
            'shipping_cost': 'mean'
        })

        # Create derived columns
        orders_df['net_amount'] = orders_df['total_amount'] -
orders_df['discount_amount']
        orders_df['profit_amount'] = orders_df['net_amount'] * 0.3  # Assume 30%
margin
        orders_df['order_year'] = pd.to_datetime(orders_df['order_date']).dt.year
        orders_df['order_month'] =
pd.to_datetime(orders_df['order_date']).dt.month
        orders_df['order_quarter'] =
pd.to_datetime(orders_df['order_date']).dt.quarter

        # Customer segmentation
        orders_df['customer_value_segment'] = orders_df.groupby('customer_id')
['total_amount'].transform('sum').apply(
            lambda x: 'High' if x > 10000 else 'Medium' if x > 5000 else 'Low'
        )

        # Product performance metrics
        orders_df['product_revenue_rank'] = orders_df.groupby(['order_year',
'order_month'])['net_amount'].rank(method='dense', ascending=False)

        return orders_df
```

```
    def create_dimensional_model(self, transformed_data: pd.DataFrame):
        """Create dimensional model from transformed data"""

        # Create dimension tables
        dim_date = self._create_date_dimension()
        dim_customer = self._create_customer_dimension(transformed_data)
        dim_product = self._create_product_dimension(transformed_data)

        # Create fact table
        fact_sales = self._create_sales_fact(transformed_data, dim_date,
dim_customer, dim_product)

        return {
            'dim_date': dim_date,
            'dim_customer': dim_customer,
            'dim_product': dim_product,
            'fact_sales': fact_sales
        }

    def _create_date_dimension(self):
        """Create date dimension"""
        start_date = datetime(2020, 1, 1)
        end_date = datetime(2025, 12, 31)

        dates = pd.date_range(start=start_date, end=end_date, freq='D')

        dim_date = pd.DataFrame({
            'date_key': [int(d.strftime('%Y%m%d')) for d in dates],
            'full_date': dates,
            'day_of_week': dates.day_name(),
            'day_of_month': dates.day,
            'day_of_year': dates.dayofyear,
            'week_of_year': dates.isocalendar().week,
            'month_name': dates.month_name(),
            'month_number': dates.month,
            'quarter': dates.quarter,
            'year': dates.year,
            'is_weekend': dates.weekday >= 5,
            'is_holiday': False  # Would need holiday calendar integration
        })

        return dim_date

    def _create_customer_dimension(self, data: pd.DataFrame):
        """Create customer dimension"""
        customers = data.groupby('customer_id').agg({
            'email': 'first',
            'registration_date': 'first',
            'customer_segment': 'first',
            'customer_value_segment': 'first',
            'total_amount': 'sum'
        }).reset_index()
```

```
        customers['customer_key'] = range(1, len(customers) + 1)

        return customers[[
            'customer_key', 'customer_id', 'email', 'registration_date',
            'customer_segment', 'customer_value_segment', 'total_amount'
        ]]

    def _create_product_dimension(self, data: pd.DataFrame):
        """Create product dimension"""
        products = data.groupby('product_id').agg({
            'product_name': 'first',
            'category': 'first',
            'brand': 'first',
            'unit_price': 'mean'
        }).reset_index()

        products['product_key'] = range(1, len(products) + 1)

        return products[[
            'product_key', 'product_id', 'product_name',
            'category', 'brand', 'unit_price'
        ]]

    def _create_sales_fact(self, data: pd.DataFrame, dim_date: pd.DataFrame,
                           dim_customer: pd.DataFrame, dim_product: pd.DataFrame):
        """Create sales fact table"""

        # Create date keys
        data['date_key'] =
pd.to_datetime(data['order_date']).dt.strftime('%Y%m%d').astype(int)

        # Merge with dimensions to get keys
        fact_sales = data.merge(
            dim_customer[['customer_key', 'customer_id']],
            on='customer_id'
        ).merge(
            dim_product[['product_key', 'product_id']],
            on='product_id'
        )

        return fact_sales[[
            'order_id', 'date_key', 'customer_key', 'product_key',
            'quantity', 'unit_price', 'total_amount', 'discount_amount',
            'net_amount', 'profit_amount'
        ]]

    def run_quality_checks(self, data: Dict[str, pd.DataFrame]):
        """Run comprehensive data quality checks"""
        quality_results = {}

        # Check fact table
        fact_rules = {
            'completeness': ['order_id', 'date_key', 'customer_key',
'product_key'],
```

```
            'uniqueness': ['order_id'],
            'value_ranges': {
                'quantity': {'min': 1, 'max': 1000},
                'unit_price': {'min': 0, 'max': 10000},
                'total_amount': {'min': 0, 'max': 100000}
            }
        }

        quality_results['fact_sales'] = self.quality_checker.run_quality_checks(
            data['fact_sales'], fact_rules
        )

        # Check dimension tables
        dim_customer_rules = {
            'completeness': ['customer_id', 'email'],
            'uniqueness': ['customer_id', 'email']
        }

        quality_results['dim_customer'] = self.quality_checker.run_quality_checks(
            data['dim_customer'], dim_customer_rules
        )

        return quality_results

    def load_to_warehouse(self, dimensional_data: Dict[str, pd.DataFrame]):
        """Load data to PostgreSQL warehouse"""
        loader = PostgreSQLLoader(self.config['target'])
        loader.connect()

        try:
            # Load dimensions first
            for table_name, df in dimensional_data.items():
                if table_name.startswith('dim_'):
                    loader.upsert_data(df, table_name, ['customer_id'] if
'customer' in table_name else ['product_id'])

            # Load fact table
            loader.upsert_data(
                dimensional_data['fact_sales'],
                'fact_sales',
                ['order_id']
            )

            self.logger.info("Data loaded successfully to warehouse")

        except Exception as e:
            self.logger.error(f"Failed to load data: {e}")
            raise
        finally:
            loader.close()

    def generate_business_reports(self):
        """Generate automated business reports"""
        warehouse_conn = PostgreSQLExtractor(self.config['target'])
```

```
        # Daily sales report
        daily_sales_query = """
        SELECT
            dd.full_date,
            SUM(fs.total_amount) as total_sales,
            SUM(fs.profit_amount) as total_profit,
            COUNT(DISTINCT fs.order_id) as total_orders,
            AVG(fs.total_amount) as avg_order_value
        FROM fact_sales fs
        JOIN dim_date dd ON fs.date_key = dd.date_key
        WHERE dd.full_date >= CURRENT_DATE - INTERVAL '30 days'
        GROUP BY dd.full_date
        ORDER BY dd.full_date DESC
        """

        daily_sales = warehouse_conn.execute_query(daily_sales_query)

        # Customer segment analysis
        segment_analysis_query = """
        SELECT
            dc.customer_segment,
            COUNT(DISTINCT dc.customer_id) as customer_count,
            SUM(fs.total_amount) as total_revenue,
            AVG(fs.total_amount) as avg_order_value,
            SUM(fs.profit_amount) as total_profit
        FROM dim_customer dc
        JOIN fact_sales fs ON dc.customer_key = fs.customer_key
        JOIN dim_date dd ON fs.date_key = dd.date_key
        WHERE dd.full_date >= CURRENT_DATE - INTERVAL '90 days'
        GROUP BY dc.customer_segment
        ORDER BY total_revenue DESC
        """

        segment_analysis = warehouse_conn.execute_query(segment_analysis_query)

        return {
            'daily_sales': daily_sales,
            'segment_analysis': segment_analysis
        }

    def run_full_pipeline(self, date: str):
        """Run the complete ETL pipeline"""
        start_time = datetime.now()
        self.logger.info(f"Starting ETL pipeline for {date}")

        try:
            # Extract
            orders_data = self.extract_orders_data(date, date)
            web_data = self.extract_web_analytics(date)

            # Transform
            transformed_orders = self.transform_sales_data(orders_data)
            dimensional_data = self.create_dimensional_model(transformed_orders)
```

```
            # Quality checks
            quality_results = self.run_quality_checks(dimensional_data)

            # Check if quality passed
            failed_checks = sum(1 for table_results in quality_results.values()
                                for result in table_results.values()
                                if not result.get('passed', True))

            if failed_checks > 0:
                self.logger.warning(f"Quality checks failed: {failed_checks}
issues found")
                # Decide whether to continue or halt

            # Load
            self.load_to_warehouse(dimensional_data)

            # Generate reports
            reports = self.generate_business_reports()

            # Record metrics
            duration = datetime.now() - start_time
            self.metrics.record_pipeline_run({
                'pipeline_name': 'ecommerce_etl',
                'date': date,
                'duration': duration,
                'records_processed': len(transformed_orders),
                'quality_score': (len([r for table_results in
quality_results.values()
                                       for r in table_results.values()
                                       if r.get('passed', True)]) /
                                  max(1, sum(len(table_results) for table_results in
quality_results.values()))) * 100
            })

            self.logger.info(f"ETL pipeline completed successfully in {duration}")
            return True

        except Exception as e:
            self.logger.error(f"ETL pipeline failed: {e}")
            return False

# Usage
etl_project = EcommerceETLProject()
etl_project.run_full_pipeline('2024-01-15')
```

📈 Performance Optimization

ETL Performance Best Practices

1. **Parallel Processing**

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import multiprocessing as mp

class ParallelETL:
    def __init__(self, max_workers=None):
        self.max_workers = max_workers or mp.cpu_count()

    def parallel_extract(self, source_configs):
        """Extract from multiple sources in parallel"""
        with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
            futures = {}
            for source_name, config in source_configs.items():
                future = executor.submit(self._extract_single_source, source_name,
config)
                futures[source_name] = future

            results = {}
            for source_name, future in futures.items():
                results[source_name] = future.result()

            return results

    def parallel_transform(self, data_chunks):
        """Transform data chunks in parallel"""
        with ProcessPoolExecutor(max_workers=self.max_workers) as executor:
            futures = [executor.submit(self._transform_chunk, chunk) for chunk in
data_chunks]
            results = [future.result() for future in futures]

            # Combine results
            return pd.concat(results, ignore_index=True)
```

2. **Incremental Loading**

```
class IncrementalETL:
    def __init__(self):
        self.watermark_table = 'etl_watermarks'

    def get_last_watermark(self, table_name: str) -> datetime:
        """Get last processed timestamp"""
        query = f"""
        SELECT last_processed_timestamp
        FROM {self.watermark_table}
        WHERE table_name = %s
        """
        result = self.db.execute_query(query, (table_name,))
        return result[0][0] if result else datetime(1900, 1, 1)

    def update_watermark(self, table_name: str, timestamp: datetime):
        """Update watermark after successful processing"""
```

```
        query = f"""
        INSERT INTO {self.watermark_table} (table_name, last_processed_timestamp)
        VALUES (%s, %s)
        ON DUPLICATE KEY UPDATE last_processed_timestamp = %s
        """
        self.db.execute_query(query, (table_name, timestamp, timestamp))

    def incremental_extract(self, table_name: str, timestamp_column: str):
        """Extract only new/modified records"""
        last_watermark = self.get_last_watermark(table_name)

        query = f"""
        SELECT * FROM {table_name}
        WHERE {timestamp_column} > %s
        ORDER BY {timestamp_column}
        """

        data = self.db.execute_query(query, (last_watermark,))

        if data:
            max_timestamp = max(row[timestamp_column] for row in data)
            self.update_watermark(table_name, max_timestamp)

        return data
```

3. **Batch Processing**

```
class BatchProcessor:
    def __init__(self, batch_size=10000):
        self.batch_size = batch_size

    def process_in_batches(self, data: pd.DataFrame, process_func):
        """Process large datasets in batches"""
        total_rows = len(data)
        processed_rows = 0

        for start_idx in range(0, total_rows, self.batch_size):
            end_idx = min(start_idx + self.batch_size, total_rows)
            batch = data.iloc[start_idx:end_idx]

            try:
                process_func(batch)
                processed_rows += len(batch)

                # Progress logging
                progress = (processed_rows / total_rows) * 100
                logging.info(f"Processed {processed_rows}/{total_rows} rows
({progress:.1f}%)")

            except Exception as e:
                logging.error(f"Failed to process batch {start_idx}-{end_idx}:
```

```
{e}")
                raise
```

🎓 Interview Questions & Answers

Beginner Level

Q1: What is ETL and why is it important?

A: ETL stands for Extract, Transform, Load. It's a data integration process that:

- **Extract**: Retrieves data from various sources (databases, APIs, files)
- **Transform**: Cleans, validates, and converts data into the desired format
- **Load**: Stores the processed data into a target system (data warehouse, database)

ETL is important because it enables organizations to consolidate data from multiple sources, ensure data quality, and make data available for analytics and reporting.

Q2: What's the difference between ETL and ELT?

A:

- **ETL**: Transform data before loading into the target system. Better for complex transformations and when target system has limited processing power.
- **ELT**: Load raw data first, then transform within the target system. Better for cloud data warehouses with powerful processing capabilities like Snowflake, BigQuery.

Intermediate Level

Q3: How do you handle slowly changing dimensions (SCD)?

A: There are three main types:

- **Type 1**: Overwrite old values (no history preserved)
- **Type 2**: Create new records for changes (full history preserved)
- **Type 3**: Add columns for previous values (limited history)

Choice depends on business requirements for historical data tracking.

Q4: What are the key considerations for ETL performance optimization?

A:

- **Parallel processing**: Process multiple data streams simultaneously
- **Incremental loading**: Only process new/changed data
- **Batch processing**: Handle large datasets in manageable chunks
- **Indexing**: Optimize database queries with proper indexes
- **Data partitioning**: Distribute data across multiple storage units
- **Compression**: Reduce data transfer and storage costs

Advanced Level

Q5: How do you design an ETL pipeline for real-time data processing?

A: Real-time ETL requires:

- **Streaming platforms**: Apache Kafka, Amazon Kinesis
- **Stream processing**: Apache Spark Streaming, Apache Flink
- **Micro-batch processing**: Process small batches frequently
- **Change Data Capture (CDC)**: Capture database changes in real-time
- **Event-driven architecture**: React to data changes immediately

Q6: How do you ensure data quality in ETL pipelines?

A:

- **Data profiling**: Understand data characteristics and patterns
- **Validation rules**: Check completeness, uniqueness, referential integrity
- **Data lineage**: Track data flow from source to destination
- **Monitoring and alerting**: Detect anomalies and failures
- **Data quality metrics**: Measure and report quality scores
- **Automated testing**: Unit tests for transformation logic

⚠ Common Pitfalls

1. **Not handling data quality early**
    - Always validate data at extraction point
    - Implement comprehensive quality checks
2. **Ignoring incremental loading**
    - Full loads become unsustainable with large datasets
    - Implement proper watermarking strategies
3. **Poor error handling**
    - ETL pipelines should be resilient to failures
    - Implement retry mechanisms and proper logging
4. **Not considering data lineage**
    - Track data flow for debugging and compliance
    - Document transformation logic
5. **Inadequate monitoring**
    - Monitor pipeline performance and data quality
    - Set up alerts for failures and anomalies

Next Steps

Congratulations! You've completed Chapter 21 on ETL and Data Warehousing. You should now understand:

✅ ETL vs ELT concepts and when to use each ✅ Data extraction from multiple sources (databases, APIs, files) ✅ Data transformation techniques and best practices ✅ Data loading strategies (bulk, batch, incremental) ✅ Data warehousing concepts (star schema, snowflake schema) ✅ Slowly changing dimensions (SCD) implementation ✅ Data quality management and monitoring ✅ Performance optimization techniques ✅ Real-world ETL project implementation

Continue to: Chapter 22: Cloud Databases & Database as a Service (DBaaS)

Practice Projects:

1. Build an ETL pipeline for your own data sources
2. Implement a real-time streaming ETL with Kafka
3. Create a data quality monitoring dashboard
4. Design a multi-source data warehouse

Additional Resources:

- Apache Airflow for workflow orchestration
- dbt (data build tool) for transformation
- Great Expectations for data quality
- Apache Spark for big data processing

Chapter 22: Cloud Databases & Database as a Service (DBaaS)

🗄 Learning Objectives

By the end of this chapter, you will:

- Understand cloud database concepts and benefits
- Learn about major cloud database providers (AWS, Azure, GCP)
- Master database migration strategies to the cloud
- Implement cloud-native database solutions
- Understand serverless database concepts
- Learn about multi-cloud and hybrid database architectures
- Implement database monitoring and cost optimization in the cloud
- Understand compliance and security in cloud databases

🌐 Introduction to Cloud Databases

What are Cloud Databases?

Cloud databases are databases that run on cloud computing platforms, providing:

- **Scalability**: Automatic scaling based on demand
- **Availability**: High availability with built-in redundancy

- **Managed Services**: Reduced operational overhead
- **Global Reach**: Deploy databases closer to users worldwide
- **Cost Efficiency**: Pay-as-you-use pricing models

Types of Cloud Database Services

1. **Infrastructure as a Service (IaaS)**
    - Virtual machines with self-managed databases
    - Full control over database configuration
    - Examples: EC2 with MySQL, Azure VMs with SQL Server
2. **Platform as a Service (PaaS)**
    - Managed database services
    - Automated backups, updates, and scaling
    - Examples: AWS RDS, Azure SQL Database, Google Cloud SQL
3. **Software as a Service (SaaS)**
    - Fully managed database applications
    - No infrastructure management required
    - Examples: Salesforce, Office 365
4. **Database as a Service (DBaaS)**
    - Specialized managed database services
    - Optimized for specific use cases
    - Examples: MongoDB Atlas, Redis Cloud, Snowflake

☁ Major Cloud Database Providers

Amazon Web Services (AWS)

AWS RDS (Relational Database Service)

```
-- Creating an RDS MySQL instance via AWS CLI
aws rds create-db-instance \
    --db-instance-identifier myapp-mysql \
    --db-instance-class db.t3.micro \
    --engine mysql \
    --engine-version 8.0.35 \
    --master-username admin \
    --master-user-password MySecurePassword123! \
    --allocated-storage 20 \
    --storage-type gp2 \
    --vpc-security-group-ids sg-12345678 \
    --db-subnet-group-name myapp-subnet-group \
    --backup-retention-period 7 \
    --multi-az \
```

```
    --storage-encrypted \
    --tags Key=Environment,Value=Production Key=Application,Value=MyApp

-- Connecting to RDS MySQL
mysql -h myapp-mysql.cluster-xyz.us-east-1.rds.amazonaws.com \
      -u admin \
      -p \
      -P 3306 \
      myapp_database
```

AWS Aurora (MySQL/PostgreSQL Compatible)

```
-- Creating Aurora Cluster
aws rds create-db-cluster \
    --db-cluster-identifier myapp-aurora-cluster \
    --engine aurora-mysql \
    --engine-version 8.0.mysql_aurora.3.02.0 \
    --master-username admin \
    --master-user-password MySecurePassword123! \
    --vpc-security-group-ids sg-12345678 \
    --db-subnet-group-name myapp-subnet-group \
    --backup-retention-period 7 \
    --storage-encrypted \
    --enable-cloudwatch-logs-exports error general slowquery

-- Creating Aurora instances
aws rds create-db-instance \
    --db-instance-identifier myapp-aurora-writer \
    --db-instance-class db.r6g.large \
    --engine aurora-mysql \
    --db-cluster-identifier myapp-aurora-cluster

aws rds create-db-instance \
    --db-instance-identifier myapp-aurora-reader \
    --db-instance-class db.r6g.large \
    --engine aurora-mysql \
    --db-cluster-identifier myapp-aurora-cluster
```

AWS DynamoDB (NoSQL)

```
import boto3
from boto3.dynamodb.conditions import Key, Attr

# Initialize DynamoDB client
dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

# Create table
table = dynamodb.create_table(
```

```
    TableName='Users',
    KeySchema=[
        {
            'AttributeName': 'user_id',
            'KeyType': 'HASH'  # Partition key
        },
        {
            'AttributeName': 'created_at',
            'KeyType': 'RANGE'  # Sort key
        }
    ],
    AttributeDefinitions=[
        {
            'AttributeName': 'user_id',
            'AttributeType': 'S'
        },
        {
            'AttributeName': 'created_at',
            'AttributeType': 'S'
        },
        {
            'AttributeName': 'email',
            'AttributeType': 'S'
        }
    ],
    GlobalSecondaryIndexes=[
        {
            'IndexName': 'EmailIndex',
            'KeySchema': [
                {
                    'AttributeName': 'email',
                    'KeyType': 'HASH'
                }
            ],
            'Projection': {
                'ProjectionType': 'ALL'
            },
            'BillingMode': 'PAY_PER_REQUEST'
        }
    ],
    BillingMode='PAY_PER_REQUEST',
    Tags=[
        {
            'Key': 'Environment',
            'Value': 'Production'
        }
    ]
)

# Wait for table to be created
table.wait_until_exists()

# Insert data
table = dynamodb.Table('Users')
```

```
table.put_item(
    Item={
        'user_id': 'user123',
        'created_at': '2024-01-15T10:30:00Z',
        'email': 'user@example.com',
        'name': 'John Doe',
        'age': 30,
        'preferences': {
            'theme': 'dark',
            'notifications': True
        }
    }
)

# Query data
response = table.query(
    KeyConditionExpression=Key('user_id').eq('user123')
)

# Scan with filter
response = table.scan(
    FilterExpression=Attr('age').gt(25) &
Attr('preferences.notifications').eq(True)
)

# Update item
table.update_item(
    Key={
        'user_id': 'user123',
        'created_at': '2024-01-15T10:30:00Z'
    },
    UpdateExpression='SET age = :age, preferences.theme = :theme',
    ExpressionAttributeValues={
        ':age': 31,
        ':theme': 'light'
    }
)
```

Microsoft Azure

Azure SQL Database

```
-- Creating Azure SQL Database via Azure CLI
az sql server create \
    --name myapp-sql-server \
    --resource-group myapp-rg \
    --location eastus \
    --admin-user sqladmin \
    --admin-password MySecurePassword123!

az sql db create \
```

```
    --resource-group myapp-rg \
    --server myapp-sql-server \
    --name myapp-database \
    --service-objective S2 \
    --backup-storage-redundancy Local

-- Connection string
-- Server=myapp-sql-server.database.windows.net;Database=myapp-database;User
ID=sqladmin;Password=MySecurePassword123!;Encrypt=True;TrustServerCertificate=Fals
e;

-- Azure SQL Database specific features
-- Automatic tuning
ALTER DATABASE myapp_database SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON);
ALTER DATABASE myapp_database SET AUTOMATIC_TUNING (CREATE_INDEX = ON);
ALTER DATABASE myapp_database SET AUTOMATIC_TUNING (DROP_INDEX = ON);

-- Query Store (automatically enabled)
SELECT
    qsq.query_id,
    qsq.query_sql_text,
    qrs.avg_duration/1000.0 as avg_duration_ms,
    qrs.avg_cpu_time/1000.0 as avg_cpu_time_ms,
    qrs.execution_count
FROM sys.query_store_query qsq
JOIN sys.query_store_plan qsp ON qsq.query_id = qsp.query_id
JOIN sys.query_store_runtime_stats qrs ON qsp.plan_id = qrs.plan_id
WHERE qrs.last_execution_time > DATEADD(hour, -1, GETUTCDATE())
ORDER BY qrs.avg_duration DESC;

-- Elastic Pool for multiple databases
az sql elastic-pool create \
    --resource-group myapp-rg \
    --server myapp-sql-server \
    --name myapp-elastic-pool \
    --edition Standard \
    --dtu 100 \
    --db-dtu-max 20 \
    --db-dtu-min 5
```

Azure Database for PostgreSQL

```
-- Creating Azure PostgreSQL via Azure CLI
az postgres server create \
    --resource-group myapp-rg \
    --name myapp-postgres-server \
    --location eastus \
    --admin-user postgres \
    --admin-password MySecurePassword123! \
    --sku-name GP_Gen5_2 \
    --version 13
```

```
az postgres db create \
    --resource-group myapp-rg \
    --server-name myapp-postgres-server \
    --name myapp_database

-- Connection via psql
psql "host=myapp-postgres-server.postgres.database.azure.com port=5432
dbname=myapp_database user=postgres@myapp-postgres-server
password=MySecurePassword123! sslmode=require"

-- Azure PostgreSQL specific features
-- Read replicas
az postgres server replica create \
    --name myapp-postgres-replica \
    --resource-group myapp-rg \
    --source-server myapp-postgres-server

-- Point-in-time restore
az postgres server restore \
    --resource-group myapp-rg \
    --name myapp-postgres-restored \
    --restore-point-in-time "2024-01-15T10:00:00Z" \
    --source-server myapp-postgres-server
```

Google Cloud Platform (GCP)

Google Cloud SQL

```
-- Creating Cloud SQL MySQL via gcloud CLI
gcloud sql instances create myapp-mysql-instance \
    --database-version=MYSQL_8_0 \
    --tier=db-n1-standard-2 \
    --region=us-central1 \
    --root-password=MySecurePassword123! \
    --backup-start-time=03:00 \
    --enable-bin-log \
    --maintenance-window-day=SUN \
    --maintenance-window-hour=04 \
    --storage-auto-increase \
    --storage-size=20GB \
    --storage-type=SSD

-- Create database
gcloud sql databases create myapp_database \
    --instance=myapp-mysql-instance

-- Create user
gcloud sql users create appuser \
    --instance=myapp-mysql-instance \
    --password=AppUserPassword123!
```

```
-- Connection via Cloud SQL Proxy
./cloud_sql_proxy -instances=myproject:us-central1:myapp-mysql-instance=tcp:3306 &
mysql -h 127.0.0.1 -u root -p myapp_database
```

Google Cloud Spanner (Globally Distributed)

```
-- Creating Spanner instance via gcloud
gcloud spanner instances create myapp-spanner-instance \
    --config=regional-us-central1 \
    --description="MyApp Spanner Instance" \
    --nodes=1

-- Create database
gcloud spanner databases create myapp_database \
    --instance=myapp-spanner-instance

-- DDL for Spanner
CREATE TABLE Users (
    UserId STRING(36) NOT NULL,
    Email STRING(255) NOT NULL,
    Name STRING(100),
    CreatedAt TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp=true),
    UpdatedAt TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp=true)
) PRIMARY KEY (UserId);

CREATE INDEX UsersByEmail ON Users(Email);

CREATE TABLE Orders (
    OrderId STRING(36) NOT NULL,
    UserId STRING(36) NOT NULL,
    TotalAmount NUMERIC NOT NULL,
    Status STRING(20) NOT NULL,
    CreatedAt TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp=true)
) PRIMARY KEY (UserId, OrderId),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;

-- Spanner-specific queries
-- Strong consistency read
SELECT * FROM Users WHERE UserId = @user_id;

-- Stale read (for better performance)
SELECT * FROM Users WHERE Email = @email
OPTIONS (read_timestamp = CURRENT_TIMESTAMP() - INTERVAL 10 SECOND);

-- Partitioned DML for large updates
PARTITION UPDATE Users SET UpdatedAt = PENDING_COMMIT_TIMESTAMP()
WHERE CreatedAt < TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 30 DAY);
```

🔄 Database Migration to Cloud

Migration Strategies

1. **Lift and Shift**
   - Move existing database to cloud VMs
   - Minimal changes required
   - Quick migration but limited cloud benefits
2. **Re-platform**
   - Move to managed database services
   - Some application changes may be needed
   - Better cloud integration
3. **Re-architect**
   - Redesign for cloud-native services
   - Significant changes but maximum benefits
   - Microservices, serverless, etc.

AWS Database Migration Service (DMS)

```
import boto3
import json

# Initialize DMS client
dms = boto3.client('dms', region_name='us-east-1')

# Create replication subnet group
response = dms.create_replication_subnet_group(
    ReplicationSubnetGroupIdentifier='myapp-dms-subnet-group',
    ReplicationSubnetGroupDescription='DMS subnet group for MyApp migration',
    SubnetIds=[
        'subnet-12345678',
        'subnet-87654321'
    ],
    Tags=[
        {
            'Key': 'Environment',
            'Value': 'Migration'
        }
    ]
)

# Create replication instance
response = dms.create_replication_instance(
    ReplicationInstanceIdentifier='myapp-dms-instance',
    ReplicationInstanceClass='dms.t3.micro',
    AllocatedStorage=20,
    VpcSecurityGroupIds=[
```

```
            'sg-12345678'
        ],
        ReplicationSubnetGroupIdentifier='myapp-dms-subnet-group',
        MultiAZ=False,
        PubliclyAccessible=False,
        Tags=[
            {
                'Key': 'Environment',
                'Value': 'Migration'
            }
        ]
    )

    # Create source endpoint (on-premises MySQL)
    response = dms.create_endpoint(
        EndpointIdentifier='myapp-source-mysql',
        EndpointType='source',
        EngineName='mysql',
        Username='dms_user',
        Password='dms_password',
        ServerName='onprem-mysql.company.com',
        Port=3306,
        DatabaseName='myapp_production',
        ExtraConnectionAttributes='initstmt=SET foreign_key_checks=0',
        Tags=[
            {
                'Key': 'Type',
                'Value': 'Source'
            }
        ]
    )

    # Create target endpoint (AWS RDS)
    response = dms.create_endpoint(
        EndpointIdentifier='myapp-target-rds',
        EndpointType='target',
        EngineName='mysql',
        Username='admin',
        Password='MySecurePassword123!',
        ServerName='myapp-mysql.cluster-xyz.us-east-1.rds.amazonaws.com',
        Port=3306,
        DatabaseName='myapp_production',
        Tags=[
            {
                'Key': 'Type',
                'Value': 'Target'
            }
        ]
    )

    # Create migration task
    table_mappings = {
        "rules": [
            {
```

```
            "rule-type": "selection",
            "rule-id": "1",
            "rule-name": "1",
            "object-locator": {
                "schema-name": "myapp_production",
                "table-name": "%"
            },
            "rule-action": "include"
        },
        {
            "rule-type": "transformation",
            "rule-id": "2",
            "rule-name": "2",
            "rule-target": "table",
            "object-locator": {
                "schema-name": "myapp_production",
                "table-name": "user_sessions"
            },
            "rule-action": "exclude"
        }
    ]
}

response = dms.create_replication_task(
    ReplicationTaskIdentifier='myapp-migration-task',
    SourceEndpointArn='arn:aws:dms:us-east-1:123456789012:endpoint:myapp-source-
mysql',
    TargetEndpointArn='arn:aws:dms:us-east-1:123456789012:endpoint:myapp-target-
rds',
    ReplicationInstanceArn='arn:aws:dms:us-east-1:123456789012:rep:myapp-dms-
instance',
    MigrationType='full-load-and-cdc',
    TableMappings=json.dumps(table_mappings),
    ReplicationTaskSettings=json.dumps({
        "TargetMetadata": {
            "TargetSchema": "",
            "SupportLobs": True,
            "FullLobMode": False,
            "LobChunkSize": 0,
            "LimitedSizeLobMode": True,
            "LobMaxSize": 32,
            "InlineLobMaxSize": 0,
            "LoadMaxFileSize": 0,
            "ParallelLoadThreads": 0,
            "ParallelLoadBufferSize": 0,
            "BatchApplyEnabled": False,
            "TaskRecoveryTableEnabled": False,
            "ParallelApplyThreads": 0,
            "ParallelApplyBufferSize": 0,
            "ParallelApplyQueuesPerThread": 0
        },
        "FullLoadSettings": {
            "TargetTablePrepMode": "DROP_AND_CREATE",
            "CreatePkAfterFullLoad": False,
```

```
            "StopTaskCachedChangesApplied": False,
            "StopTaskCachedChangesNotApplied": False,
            "MaxFullLoadSubTasks": 8,
            "TransactionConsistencyTimeout": 600,
            "CommitRate": 10000
        },
        "Logging": {
            "EnableLogging": True,
            "LogComponents": [
                {
                    "Id": "SOURCE_UNLOAD",
                    "Severity": "LOGGER_SEVERITY_DEFAULT"
                },
                {
                    "Id": "TARGET_LOAD",
                    "Severity": "LOGGER_SEVERITY_DEFAULT"
                }
            ]
        }
    }),
    Tags=[
        {
            'Key': 'Environment',
            'Value': 'Migration'
        }
    ]
)

# Start migration task
response = dms.start_replication_task(
    ReplicationTaskArn='arn:aws:dms:us-east-1:123456789012:task:myapp-migration-
task',
    StartReplicationTaskType='start-replication'
)

# Monitor migration progress
def check_migration_status(task_arn):
    response = dms.describe_replication_tasks(
        Filters=[
            {
                'Name': 'replication-task-arn',
                'Values': [task_arn]
            }
        ]
    )

    task = response['ReplicationTasks'][0]
    status = task['Status']

    if 'ReplicationTaskStats' in task:
        stats = task['ReplicationTaskStats']
        print(f"Status: {status}")
        print(f"Full Load Progress: {stats.get('FullLoadProgressPercent', 0)}%")
        print(f"Tables Loaded: {stats.get('TablesLoaded', 0)}")
```

```
        print(f"Tables Loading: {stats.get('TablesLoading', 0)}")
        print(f"Tables Queued: {stats.get('TablesQueued', 0)}")
        print(f"Tables Errored: {stats.get('TablesErrored', 0)}")

    return status

# Usage
status = check_migration_status('arn:aws:dms:us-east-1:123456789012:task:myapp-
migration-task')
```

Schema Conversion Tool (SCT)

```
# AWS SCT automation script
import subprocess
import json

class AWSSchemaConversionTool:
    def __init__(self, sct_path):
        self.sct_path = sct_path

    def create_project(self, project_name, source_config, target_config):
        """Create SCT project"""
        project_config = {
            "projectName": project_name,
            "sourceDatabase": source_config,
            "targetDatabase": target_config
        }

        with open(f"{project_name}_config.json", 'w') as f:
            json.dump(project_config, f, indent=2)

        cmd = [
            self.sct_path,
            "--create-project",
            f"{project_name}_config.json"
        ]

        result = subprocess.run(cmd, capture_output=True, text=True)
        return result.returncode == 0

    def convert_schema(self, project_file, output_dir):
        """Convert database schema"""
        cmd = [
            self.sct_path,
            "--project", project_file,
            "--convert-schema",
            "--output-dir", output_dir
        ]

        result = subprocess.run(cmd, capture_output=True, text=True)
```

```
        if result.returncode == 0:
            print("Schema conversion completed successfully")
            print(f"Converted files saved to: {output_dir}")
        else:
            print(f"Schema conversion failed: {result.stderr}")

        return result.returncode == 0

    def generate_assessment_report(self, project_file, report_path):
        """Generate migration assessment report"""
        cmd = [
            self.sct_path,
            "--project", project_file,
            "--assessment-report",
            "--output", report_path
        ]

        result = subprocess.run(cmd, capture_output=True, text=True)
        return result.returncode == 0

# Usage example
sct = AWSSchemaConversionTool("/opt/aws-schema-conversion-tool/bin/aws-sct")

# Source database configuration (Oracle)
source_config = {
    "engine": "oracle",
    "host": "oracle.company.com",
    "port": 1521,
    "database": "ORCL",
    "username": "hr",
    "password": "password",
    "schemas": ["HR", "SALES"]
}

# Target database configuration (PostgreSQL)
target_config = {
    "engine": "postgresql",
    "host": "myapp-postgres.cluster-xyz.us-east-1.rds.amazonaws.com",
    "port": 5432,
    "database": "myapp_production",
    "username": "postgres",
    "password": "MySecurePassword123!"
}

# Create project and convert
sct.create_project("oracle_to_postgres_migration", source_config, target_config)
sct.convert_schema("oracle_to_postgres_migration.sct", "./converted_schema")
sct.generate_assessment_report("oracle_to_postgres_migration.sct",
"./assessment_report.html")
```

Serverless Databases

AWS Aurora Serverless

```
import boto3
import json
from botocore.exceptions import ClientError

# Initialize RDS Data API client
rds_data = boto3.client('rds-data', region_name='us-east-1')

class AuroraServerlessManager:
    def __init__(self, cluster_arn, secret_arn, database_name):
        self.cluster_arn = cluster_arn
        self.secret_arn = secret_arn
        self.database_name = database_name
        self.client = boto3.client('rds-data')

    def execute_sql(self, sql, parameters=None):
        """Execute SQL statement using Data API"""
        try:
            kwargs = {
                'resourceArn': self.cluster_arn,
                'secretArn': self.secret_arn,
                'database': self.database_name,
                'sql': sql
            }

            if parameters:
                kwargs['parameters'] = parameters

            response = self.client.execute_statement(**kwargs)
            return response

        except ClientError as e:
            print(f"Error executing SQL: {e}")
            raise

    def begin_transaction(self):
        """Begin a transaction"""
        response = self.client.begin_transaction(
            resourceArn=self.cluster_arn,
            secretArn=self.secret_arn,
            database=self.database_name
        )
        return response['transactionId']

    def commit_transaction(self, transaction_id):
        """Commit a transaction"""
        response = self.client.commit_transaction(
            resourceArn=self.cluster_arn,
            secretArn=self.secret_arn,
            transactionId=transaction_id
        )
```

```
        return response

    def rollback_transaction(self, transaction_id):
        """Rollback a transaction"""
        response = self.client.rollback_transaction(
            resourceArn=self.cluster_arn,
            secretArn=self.secret_arn,
            transactionId=transaction_id
        )
        return response

    def batch_execute_sql(self, sql, parameter_sets):
        """Execute SQL with multiple parameter sets"""
        response = self.client.batch_execute_statement(
            resourceArn=self.cluster_arn,
            secretArn=self.secret_arn,
            database=self.database_name,
            sql=sql,
            parameterSets=parameter_sets
        )
        return response

# Usage example
aurora_manager = AuroraServerlessManager(
    cluster_arn='arn:aws:rds:us-east-1:123456789012:cluster:myapp-aurora-
serverless',
    secret_arn='arn:aws:secretsmanager:us-east-1:123456789012:secret:myapp-aurora-
secret',
    database_name='myapp_production'
)

# Create table
create_table_sql = """
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
"""

aurora_manager.execute_sql(create_table_sql)

# Insert data with parameters
insert_sql = "INSERT INTO users (email, name) VALUES (:email, :name)"
parameters = [
    {'name': 'email', 'value': {'stringValue': 'john@example.com'}},
    {'name': 'name', 'value': {'stringValue': 'John Doe'}}
]

result = aurora_manager.execute_sql(insert_sql, parameters)
print(f"Inserted user with ID: {result['generatedFields'][0]['longValue']}")
```

```
# Batch insert
batch_insert_sql = "INSERT INTO users (email, name) VALUES (:email, :name)"
batch_parameters = [
    [
        {'name': 'email', 'value': {'stringValue': 'alice@example.com'}},
        {'name': 'name', 'value': {'stringValue': 'Alice Smith'}}
    ],
    [
        {'name': 'email', 'value': {'stringValue': 'bob@example.com'}},
        {'name': 'name', 'value': {'stringValue': 'Bob Johnson'}}
    ]
]

aurora_manager.batch_execute_sql(batch_insert_sql, batch_parameters)

# Query data
select_sql = "SELECT id, email, name, created_at FROM users WHERE email LIKE
:email_pattern"
query_parameters = [
    {'name': 'email_pattern', 'value': {'stringValue': '%@example.com'}}
]

result = aurora_manager.execute_sql(select_sql, query_parameters)

for record in result['records']:
    user_id = record[0]['longValue']
    email = record[1]['stringValue']
    name = record[2]['stringValue']
    created_at = record[3]['stringValue']
    print(f"User {user_id}: {name} ({email}) - Created: {created_at}")

# Transaction example
transaction_id = aurora_manager.begin_transaction()

try:
    # Update user
    update_sql = "UPDATE users SET name = :name WHERE email = :email"
    update_params = [
        {'name': 'name', 'value': {'stringValue': 'John Smith'}},
        {'name': 'email', 'value': {'stringValue': 'john@example.com'}}
    ]

    aurora_manager.execute_sql(update_sql, update_params)

    # Insert audit log
    audit_sql = "INSERT INTO audit_log (action, table_name, record_id) VALUES
(:action, :table_name, :record_id)"
    audit_params = [
        {'name': 'action', 'value': {'stringValue': 'UPDATE'}},
        {'name': 'table_name', 'value': {'stringValue': 'users'}},
        {'name': 'record_id', 'value': {'longValue': 1}}
    ]

    aurora_manager.execute_sql(audit_sql, audit_params)
```

```
    # Commit transaction
    aurora_manager.commit_transaction(transaction_id)
    print("Transaction committed successfully")

except Exception as e:
    # Rollback on error
    aurora_manager.rollback_transaction(transaction_id)
    print(f"Transaction rolled back due to error: {e}")
```

Azure SQL Database Serverless

```
-- Creating Azure SQL Database Serverless
az sql db create \
    --resource-group myapp-rg \
    --server myapp-sql-server \
    --name myapp-serverless-db \
    --edition GeneralPurpose \
    --compute-model Serverless \
    --family Gen5 \
    --capacity 1 \
    --auto-pause-delay 60 \
    --min-capacity 0.5

-- Monitor serverless database usage
SELECT
    start_time,
    end_time,
    avg_cpu_percent,
    avg_data_io_percent,
    avg_log_write_percent,
    max_worker_percent,
    max_session_percent
FROM sys.dm_db_resource_stats
WHERE start_time > DATEADD(hour, -24, GETUTCDATE())
ORDER BY start_time DESC;

-- Check auto-pause status
SELECT
    database_id,
    name,
    state_desc,
    create_date
FROM sys.databases
WHERE name = 'myapp-serverless-db';
```

Security and Compliance

Encryption at Rest and in Transit

```
# AWS RDS Encryption
import boto3

rds = boto3.client('rds')

# Create encrypted RDS instance
response = rds.create_db_instance(
    DBInstanceIdentifier='myapp-encrypted-db',
    DBInstanceClass='db.t3.micro',
    Engine='mysql',
    MasterUsername='admin',
    MasterUserPassword='MySecurePassword123!',
    AllocatedStorage=20,
    StorageEncrypted=True,  # Enable encryption at rest
    KmsKeyId='arn:aws:kms:us-east-1:123456789012:key/12345678-1234-1234-1234-
123456789012',
    VpcSecurityGroupIds=['sg-12345678'],
    BackupRetentionPeriod=7,
    MultiAZ=True,
    Tags=[
        {
            'Key': 'Environment',
            'Value': 'Production'
        },
        {
            'Key': 'Encryption',
            'Value': 'Enabled'
        }
    ]
)

# SSL/TLS connection configuration
connection_config = {
    'host': 'myapp-encrypted-db.cluster-xyz.us-east-1.rds.amazonaws.com',
    'user': 'admin',
    'password': 'MySecurePassword123!',
    'database': 'myapp_production',
    'ssl_ca': '/path/to/rds-ca-2019-root.pem',
    'ssl_verify_cert': True,
    'ssl_verify_identity': True
}
```

Identity and Access Management (IAM)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowRDSAccess",
      "Effect": "Allow",
```

```
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/MyAppDatabaseRole"
      },
      "Action": ["rds-db:connect"],
      "Resource": [
        "arn:aws:rds-db:us-east-1:123456789012:dbuser:myapp-mysql-
instance/myapp_user"
      ]
    },
    {
      "Sid": "AllowSecretsManagerAccess",
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret"
      ],
      "Resource": [
        "arn:aws:secretsmanager:us-east-
1:123456789012:secret:myapp/database/credentials-*"
      ]
    }
  ]
}
```

Database Activity Monitoring

```
# AWS RDS Performance Insights
import boto3
from datetime import datetime, timedelta

pi = boto3.client('pi', region_name='us-east-1')

class PerformanceInsightsMonitor:
    def __init__(self, resource_id):
        self.resource_id = resource_id
        self.client = pi

    def get_top_sql_statements(self, hours=1):
        """Get top SQL statements by CPU usage"""
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(hours=hours)

        response = self.client.get_dimension_key_details(
            ServiceType='RDS',
            Identifier=self.resource_id,
            Group='db.SQL_Innodb.Innodb_rows_read.avg',
            RequestedDimensions=['db.sql.Innodb_rows_read.avg']
        )

        return response
```

```
    def get_resource_metrics(self, metric_queries, hours=1):
        """Get resource metrics"""
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(hours=hours)

        response = self.client.get_resource_metrics(
            ServiceType='RDS',
            Identifier=self.resource_id,
            MetricQueries=metric_queries,
            StartTime=start_time,
            EndTime=end_time,
            PeriodInSeconds=300
        )

        return response

    def analyze_performance(self):
        """Analyze database performance"""
        # Define metric queries
        metric_queries = [
            {
                'Metric': 'db.CPU.Innodb_rows_read.avg',
                'GroupBy': {
                    'Group': 'db.sql_tokenized',
                    'Limit': 10
                }
            },
            {
                'Metric': 'db.IO.Innodb_data_read.avg',
                'GroupBy': {
                    'Group': 'db.sql_tokenized',
                    'Limit': 10
                }
            }
        ]

        metrics = self.get_resource_metrics(metric_queries)

        print("Performance Analysis Results:")
        for metric_result in metrics['MetricList']:
            print(f"\nMetric: {metric_result['Key']['Metric']}")
            for data_point in metric_result['DataPoints']:
                timestamp = data_point['Timestamp']
                value = data_point['Value']
                print(f"  {timestamp}: {value}")

# Usage
monitor = PerformanceInsightsMonitor('myapp-mysql-instance')
monitor.analyze_performance()
```

Cost Optimization

Right-sizing Database Instances

```
import boto3
from datetime import datetime, timedelta

class DatabaseCostOptimizer:
    def __init__(self):
        self.rds = boto3.client('rds')
        self.cloudwatch = boto3.client('cloudwatch')
        self.pricing = boto3.client('pricing', region_name='us-east-1')

    def get_db_instances(self):
        """Get all RDS instances"""
        response = self.rds.describe_db_instances()
        return response['DBInstances']

    def get_cpu_utilization(self, instance_id, days=30):
        """Get average CPU utilization"""
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(days=days)

        response = self.cloudwatch.get_metric_statistics(
            Namespace='AWS/RDS',
            MetricName='CPUUtilization',
            Dimensions=[
                {
                    'Name': 'DBInstanceIdentifier',
                    'Value': instance_id
                }
            ],
            StartTime=start_time,
            EndTime=end_time,
            Period=3600,  # 1 hour
            Statistics=['Average']
        )

        if response['Datapoints']:
            avg_cpu = sum(dp['Average'] for dp in response['Datapoints']) /
len(response['Datapoints'])
            return avg_cpu
        return 0

    def get_connection_count(self, instance_id, days=30):
        """Get average connection count"""
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(days=days)

        response = self.cloudwatch.get_metric_statistics(
            Namespace='AWS/RDS',
            MetricName='DatabaseConnections',
            Dimensions=[
                {
```

```
                    'Name': 'DBInstanceIdentifier',
                    'Value': instance_id
                }
            ],
            StartTime=start_time,
            EndTime=end_time,
            Period=3600,
            Statistics=['Average']
        )

        if response['Datapoints']:
            avg_connections = sum(dp['Average'] for dp in response['Datapoints'])
/ len(response['Datapoints'])
            return avg_connections
        return 0

    def recommend_instance_size(self, instance):
        """Recommend optimal instance size"""
        instance_id = instance['DBInstanceIdentifier']
        current_class = instance['DBInstanceClass']

        avg_cpu = self.get_cpu_utilization(instance_id)
        avg_connections = self.get_connection_count(instance_id)

        recommendations = []

        # CPU-based recommendations
        if avg_cpu < 20:
            recommendations.append({
                'type': 'downsize',
                'reason': f'Low CPU utilization ({avg_cpu:.1f}%)',
                'suggested_action': 'Consider smaller instance class'
            })
        elif avg_cpu > 80:
            recommendations.append({
                'type': 'upsize',
                'reason': f'High CPU utilization ({avg_cpu:.1f}%)',
                'suggested_action': 'Consider larger instance class'
            })

        # Connection-based recommendations
        if avg_connections < 10:
            recommendations.append({
                'type': 'optimize',
                'reason': f'Low connection count ({avg_connections:.1f})',
                'suggested_action': 'Consider connection pooling or smaller
instance'
            })

        return {
            'instance_id': instance_id,
            'current_class': current_class,
            'avg_cpu': avg_cpu,
            'avg_connections': avg_connections,
```

```
            'recommendations': recommendations
        }

    def analyze_all_instances(self):
        """Analyze all RDS instances for cost optimization"""
        instances = self.get_db_instances()
        analysis_results = []

        for instance in instances:
            if instance['DBInstanceStatus'] == 'available':
                result = self.recommend_instance_size(instance)
                analysis_results.append(result)

        return analysis_results

    def generate_cost_report(self):
        """Generate cost optimization report"""
        results = self.analyze_all_instances()

        print("=== RDS Cost Optimization Report ===")
        print(f"Analysis Date: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
        print(f"Total Instances Analyzed: {len(results)}")
        print()

        total_recommendations = 0

        for result in results:
            print(f"Instance: {result['instance_id']}")
            print(f"  Current Class: {result['current_class']}")
            print(f"  Avg CPU: {result['avg_cpu']:.1f}%")
            print(f"  Avg Connections: {result['avg_connections']:.1f}")

            if result['recommendations']:
                print("  Recommendations:")
                for rec in result['recommendations']:
                    print(f"    - {rec['type'].upper()}: {rec['reason']}")
                    print(f"      Action: {rec['suggested_action']}")
                total_recommendations += len(result['recommendations'])
            else:
                print("  No recommendations - instance appears optimally sized")
            print()

        print(f"Total Optimization Opportunities: {total_recommendations}")

# Usage
optimizer = DatabaseCostOptimizer()
optimizer.generate_cost_report()
```

Reserved Instances and Savings Plans

```
# AWS RDS Reserved Instance management
import boto3
from datetime import datetime, timedelta

class RDSReservedInstanceManager:
    def __init__(self):
        self.rds = boto3.client('rds')
        self.ce = boto3.client('ce')  # Cost Explorer

    def get_current_usage(self):
        """Get current RDS usage patterns"""
        instances = self.rds.describe_db_instances()

        usage_summary = {}

        for instance in instances['DBInstances']:
            if instance['DBInstanceStatus'] == 'available':
                instance_class = instance['DBInstanceClass']
                engine = instance['Engine']
                multi_az = instance['MultiAZ']

                key = f"{engine}_{instance_class}_{multi_az}"

                if key not in usage_summary:
                    usage_summary[key] = {
                        'engine': engine,
                        'instance_class': instance_class,
                        'multi_az': multi_az,
                        'count': 0,
                        'instances': []
                    }

                usage_summary[key]['count'] += 1
                usage_summary[key]
['instances'].append(instance['DBInstanceIdentifier'])

        return usage_summary

    def get_reserved_instances(self):
        """Get current reserved instances"""
        response = self.rds.describe_reserved_db_instances()
        return response['ReservedDBInstances']

    def get_available_offerings(self, engine, instance_class):
        """Get available reserved instance offerings"""
        response = self.rds.describe_reserved_db_instances_offerings(
            DBInstanceClass=instance_class,
            ProductDescription=engine,
            MultiAZ=True
        )
        return response['ReservedDBInstancesOfferings']

    def calculate_savings(self, usage_summary):
```

```
        """Calculate potential savings with reserved instances"""
        savings_analysis = []

        for usage_key, usage_data in usage_summary.items():
            if usage_data['count'] > 0:
                offerings = self.get_available_offerings(
                    usage_data['engine'],
                    usage_data['instance_class']
                )

                for offering in offerings:
                    if offering['Duration'] == 31536000:  # 1 year
                        upfront_cost = offering.get('FixedPrice', 0)
                        hourly_cost = offering.get('UsagePrice', 0)

                        # Calculate annual costs
                        annual_reserved_cost = upfront_cost + (hourly_cost * 8760)
# 8760 hours/year
                        annual_on_demand_cost = self.get_on_demand_cost(
                            usage_data['engine'],
                            usage_data['instance_class']
                        ) * 8760

                        annual_savings = (annual_on_demand_cost -
annual_reserved_cost) * usage_data['count']
                        savings_percentage = (annual_savings /
(annual_on_demand_cost * usage_data['count'])) * 100

                        savings_analysis.append({
                            'usage_key': usage_key,
                            'engine': usage_data['engine'],
                            'instance_class': usage_data['instance_class'],
                            'instance_count': usage_data['count'],
                            'offering_id':
offering['ReservedDBInstancesOfferingId'],
                            'duration': offering['Duration'],
                            'payment_option': offering['PaymentOption'],
                            'upfront_cost': upfront_cost,
                            'hourly_cost': hourly_cost,
                            'annual_savings': annual_savings,
                            'savings_percentage': savings_percentage
                        })

        return sorted(savings_analysis, key=lambda x: x['annual_savings'],
reverse=True)

    def get_on_demand_cost(self, engine, instance_class):
        """Get on-demand hourly cost (simplified - would need pricing API)"""
        # This is a simplified example - in practice, you'd use the AWS Pricing
API
        pricing_map = {
            'mysql': {
                'db.t3.micro': 0.017,
                'db.t3.small': 0.034,
```

```
                'db.t3.medium': 0.068,
                'db.r5.large': 0.24,
                'db.r5.xlarge': 0.48
            },
            'postgres': {
                'db.t3.micro': 0.018,
                'db.t3.small': 0.036,
                'db.t3.medium': 0.072,
                'db.r5.large': 0.252,
                'db.r5.xlarge': 0.504
            }
        }

        return pricing_map.get(engine, {}).get(instance_class, 0)

    def purchase_reserved_instance(self, offering_id, instance_count):
        """Purchase reserved instance"""
        try:
            response = self.rds.purchase_reserved_db_instances_offering(
                ReservedDBInstancesOfferingId=offering_id,
                DBInstanceCount=instance_count,
                Tags=[
                    {
                        'Key': 'PurchaseDate',
                        'Value': datetime.now().strftime('%Y-%m-%d')
                    },
                    {
                        'Key': 'Environment',
                        'Value': 'Production'
                    }
                ]
            )
            return response
        except Exception as e:
            print(f"Error purchasing reserved instance: {e}")
            return None

    def generate_ri_recommendation_report(self):
        """Generate reserved instance recommendation report"""
        usage_summary = self.get_current_usage()
        current_ris = self.get_reserved_instances()
        savings_analysis = self.calculate_savings(usage_summary)

        print("=== RDS Reserved Instance Recommendation Report ===")
        print(f"Report Date: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
        print()

        print("Current Usage Summary:")
        for usage_key, usage_data in usage_summary.items():
            print(f"  {usage_data['engine']} {usage_data['instance_class']}
(Multi-AZ: {usage_data['multi_az']}): {usage_data['count']} instances")
        print()

        print("Current Reserved Instances:")
```

```
        if current_ris:
            for ri in current_ris:
                print(f"  {ri['ProductDescription']} {ri['DBInstanceClass']}:
{ri['DBInstanceCount']} instances")
                print(f"    State: {ri['State']}, Remaining: {ri['Duration']}
seconds")
        else:
            print("  No reserved instances found")
        print()

        print("Savings Opportunities:")
        total_potential_savings = 0

        for analysis in savings_analysis[:5]:  # Top 5 opportunities
            print(f"  {analysis['engine']} {analysis['instance_class']}:")
            print(f"    Instance Count: {analysis['instance_count']}")
            print(f"    Payment Option: {analysis['payment_option']}")
            print(f"    Annual Savings: ${analysis['annual_savings']:,.2f}
({analysis['savings_percentage']:.1f}%)")
            print(f"    Upfront Cost: ${analysis['upfront_cost']:,.2f}")
            print(f"    Hourly Cost: ${analysis['hourly_cost']:.4f}")
            print()

            total_potential_savings += analysis['annual_savings']

        print(f"Total Potential Annual Savings: ${total_potential_savings:,.2f}")

# Usage
ri_manager = RDSReservedInstanceManager()
ri_manager.generate_ri_recommendation_report()
```

Next Steps

Congratulations! You've completed Chapter 22 on Cloud Databases & Database as a Service (DBaaS). You should now understand:

✅ Cloud database concepts and service models (IaaS, PaaS, SaaS, DBaaS) ✅ Major cloud providers (AWS, Azure, GCP) and their database offerings ✅ Database migration strategies and tools (DMS, SCT) ✅ Serverless database concepts and implementation ✅ Cloud database security and compliance ✅ Cost optimization strategies and reserved instances ✅ Multi-cloud and hybrid database architectures ✅ Monitoring and performance optimization in the cloud

Continue to: Chapter 23: Database DevOps & CI/CD

Practice Projects:

1. Migrate an on-premises database to the cloud
2. Implement a serverless application with Aurora Serverless
3. Set up multi-region database replication
4. Create a cost optimization dashboard for cloud databases

5. Implement database monitoring and alerting

Additional Resources:

- AWS Database Migration Service documentation
- Azure Database Migration Guide
- Google Cloud Database Migration documentation
- Cloud database security best practices
- Serverless database design patterns

Chapter 23: Database DevOps & CI/CD

Learning Objectives

By the end of this chapter, you will:

- Understand Database DevOps principles and practices
- Learn database version control and schema migration strategies
- Implement CI/CD pipelines for database changes
- Master database testing strategies (unit, integration, performance)
- Understand Infrastructure as Code (IaC) for databases
- Learn database monitoring and observability in DevOps
- Implement automated database deployment strategies
- Understand database rollback and disaster recovery in CI/CD

Introduction to Database DevOps

What is Database DevOps?

Database DevOps extends traditional DevOps practices to database development and operations:

- **Version Control**: Track database schema and data changes
- **Automated Testing**: Ensure database changes don't break functionality
- **Continuous Integration**: Automatically validate database changes
- **Continuous Deployment**: Safely deploy database changes to production
- **Monitoring**: Track database performance and health
- **Collaboration**: Bridge the gap between developers and DBAs

Key Principles

1. **Everything as Code**: Database schemas, configurations, and deployments
2. **Automation**: Reduce manual processes and human error
3. **Collaboration**: Shared responsibility between teams
4. **Continuous Feedback**: Monitor and improve continuously
5. **Fail Fast**: Detect issues early in the development cycle

---

Database Version Control

Schema Migration Tools

Flyway (Java-based)

```
-- V1__Create_users_table.sql
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_email ON users(email);
```

```
-- V2__Add_user_preferences.sql
ALTER TABLE users ADD COLUMN preferences JSON;

CREATE TABLE user_sessions (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    session_token VARCHAR(255) UNIQUE NOT NULL,
    expires_at TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

CREATE INDEX idx_user_sessions_token ON user_sessions(session_token);
CREATE INDEX idx_user_sessions_user_id ON user_sessions(user_id);
```

```
# flyway.conf
flyway.url=jdbc:mysql://localhost:3306/myapp_dev
flyway.user=flyway_user
flyway.password=flyway_password
flyway.locations=filesystem:./migrations
flyway.baselineOnMigrate=true
flyway.validateOnMigrate=true
flyway.cleanDisabled=true
```

```
#!/bin/bash
# flyway-migrate.sh

set -e

ENVIRONMENT=${1:-dev}
```

```
CONFIG_FILE="flyway-${ENVIRONMENT}.conf"

echo "Running Flyway migration for environment: $ENVIRONMENT"

# Validate configuration
if [ ! -f "$CONFIG_FILE" ]; then
    echo "Error: Configuration file $CONFIG_FILE not found"
    exit 1
fi

# Run migration
flyway -configFiles="$CONFIG_FILE" info
flyway -configFiles="$CONFIG_FILE" migrate

echo "Migration completed successfully"
```

Liquibase (XML/YAML/JSON/SQL)

```
<!-- db.changelog-master.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
    http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-4.0.xsd">

    <include file="db/changelog/001-create-users-table.xml"/>
    <include file="db/changelog/002-add-user-preferences.xml"/>
    <include file="db/changelog/003-create-orders-table.xml"/>

</databaseChangeLog>
```

```
<!-- db/changelog/001-create-users-table.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
    http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-4.0.xsd">

    <changeSet id="001-create-users-table" author="developer">
        <createTable tableName="users">
            <column name="id" type="BIGINT" autoIncrement="true">
                <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="email" type="VARCHAR(255)">
                <constraints nullable="false" unique="true"/>
            </column>
            <column name="name" type="VARCHAR(100)">
```

```xml
                <constraints nullable="false"/>
            </column>
            <column name="created_at" type="TIMESTAMP"
defaultValueComputed="CURRENT_TIMESTAMP">
                <constraints nullable="false"/>
            </column>
            <column name="updated_at" type="TIMESTAMP"
defaultValueComputed="CURRENT_TIMESTAMP">
                <constraints nullable="false"/>
            </column>
        </createTable>

        <createIndex tableName="users" indexName="idx_users_email">
            <column name="email"/>
        </createIndex>

        <rollback>
            <dropTable tableName="users"/>
        </rollback>
    </changeSet>

</databaseChangeLog>
```

```yaml
# liquibase.yml
databaseChangeLog:
  - changeSet:
      id: 002-add-user-preferences
      author: developer
      changes:
        - addColumn:
            tableName: users
            columns:
              - column:
                  name: preferences
                  type: JSON
        - createTable:
            tableName: user_sessions
            columns:
              - column:
                  name: id
                  type: BIGINT
                  autoIncrement: true
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: user_id
                  type: BIGINT
                  constraints:
                    nullable: false
              - column:
```

```
                name: session_token
                type: VARCHAR(255)
                constraints:
                  nullable: false
                  unique: true
            - column:
                name: expires_at
                type: TIMESTAMP
                constraints:
                  nullable: false
            - column:
                name: created_at
                type: TIMESTAMP
                defaultValueComputed: CURRENT_TIMESTAMP
                constraints:
                  nullable: false
        - addForeignKeyConstraint:
            baseTableName: user_sessions
            baseColumnNames: user_id
            referencedTableName: users
            referencedColumnNames: id
            constraintName: fk_user_sessions_user_id
            onDelete: CASCADE
        - createIndex:
            tableName: user_sessions
            indexName: idx_user_sessions_token
            columns:
              - column:
                  name: session_token
        - createIndex:
            tableName: user_sessions
            indexName: idx_user_sessions_user_id
            columns:
              - column:
                  name: user_id
      rollback:
        - dropTable:
            tableName: user_sessions
        - dropColumn:
            tableName: users
            columnName: preferences
```

Git-based Database Workflows

```
#!/bin/bash
# git-db-workflow.sh

set -e

# Database development workflow
function db_feature_start() {
```

```
    local feature_name=$1

    if [ -z "$feature_name" ]; then
        echo "Usage: db_feature_start <feature_name>"
        exit 1
    fi

    # Create feature branch
    git checkout -b "feature/db-$feature_name"

    # Create migration directory
    mkdir -p "migrations/$(date +%Y%m%d)_$feature_name"

    echo "Database feature branch created: feature/db-$feature_name"
}

function db_feature_test() {
    # Run database tests
    echo "Running database tests..."

    # Create test database
    mysql -u root -p -e "DROP DATABASE IF EXISTS myapp_test; CREATE DATABASE
myapp_test;"

    # Run migrations on test database
    flyway -configFiles=flyway-test.conf migrate

    # Run database unit tests
    npm run test:db

    # Run integration tests
    npm run test:integration

    echo "Database tests completed successfully"
}

function db_feature_finish() {
    local feature_name=$1

    # Run tests
    db_feature_test

    # Merge to develop
    git checkout develop
    git merge "feature/db-$feature_name"

    # Clean up
    git branch -d "feature/db-$feature_name"

    echo "Database feature merged and cleaned up"
}

# Parse command line arguments
case "$1" in
```

```
    "start")
        db_feature_start "$2"
        ;;
    "test")
        db_feature_test
        ;;
    "finish")
        db_feature_finish "$2"
        ;;
    *)
        echo "Usage: $0 {start|test|finish} [feature_name]"
        exit 1
        ;;
esac
```

Database Testing Strategies

Unit Testing for Database Objects

```
-- tests/test_user_functions.sql
-- Unit tests for user-related stored procedures and functions

DELIMITER //

-- Test setup procedure
CREATE PROCEDURE test_setup()
BEGIN
    -- Clean test data
    DELETE FROM user_sessions WHERE user_id IN (SELECT id FROM users WHERE email
LIKE '%test%');
    DELETE FROM users WHERE email LIKE '%test%';

    -- Insert test data
    INSERT INTO users (email, name) VALUES
        ('test1@example.com', 'Test User 1'),
        ('test2@example.com', 'Test User 2');
END//

-- Test teardown procedure
CREATE PROCEDURE test_teardown()
BEGIN
    DELETE FROM user_sessions WHERE user_id IN (SELECT id FROM users WHERE email
LIKE '%test%');
    DELETE FROM users WHERE email LIKE '%test%';
END//

-- Test create_user_session function
CREATE PROCEDURE test_create_user_session()
BEGIN
    DECLARE user_id BIGINT;
```

```
    DECLARE session_token VARCHAR(255);
    DECLARE session_count INT;

    -- Setup
    CALL test_setup();

    -- Get test user ID
    SELECT id INTO user_id FROM users WHERE email = 'test1@example.com';

    -- Test: Create user session
    CALL create_user_session(user_id, session_token);

    -- Verify: Session was created
    SELECT COUNT(*) INTO session_count
    FROM user_sessions
    WHERE user_id = user_id AND session_token = session_token;

    IF session_count != 1 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Test failed: Session not
created';
    END IF;

    -- Verify: Session token is not null
    IF session_token IS NULL OR LENGTH(session_token) = 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Test failed: Invalid session
token';
    END IF;

    -- Cleanup
    CALL test_teardown();

    SELECT 'test_create_user_session PASSED' AS result;
END//

-- Test validate_user_session function
CREATE PROCEDURE test_validate_user_session()
BEGIN
    DECLARE user_id BIGINT;
    DECLARE session_token VARCHAR(255);
    DECLARE is_valid BOOLEAN;

    -- Setup
    CALL test_setup();

    -- Get test user ID
    SELECT id INTO user_id FROM users WHERE email = 'test1@example.com';

    -- Create a session
    CALL create_user_session(user_id, session_token);

    -- Test: Validate valid session
    SELECT validate_user_session(session_token) INTO is_valid;

    IF NOT is_valid THEN
```

```
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Test failed: Valid session not
recognized';
    END IF;

    -- Test: Validate invalid session
    SELECT validate_user_session('invalid_token') INTO is_valid;

    IF is_valid THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Test failed: Invalid session
recognized as valid';
    END IF;

    -- Cleanup
    CALL test_teardown();

    SELECT 'test_validate_user_session PASSED' AS result;
END//

DELIMITER ;

-- Run all tests
CALL test_create_user_session();
CALL test_validate_user_session();
```

Integration Testing with Python

```
# tests/test_database_integration.py
import pytest
import mysql.connector
from mysql.connector import Error
import os
from datetime import datetime, timedelta
import json

class DatabaseIntegrationTest:
    def __init__(self):
        self.connection = None
        self.cursor = None

    def setup_method(self):
        """Setup test database connection"""
        try:
            self.connection = mysql.connector.connect(
                host=os.getenv('DB_TEST_HOST', 'localhost'),
                database=os.getenv('DB_TEST_NAME', 'myapp_test'),
                user=os.getenv('DB_TEST_USER', 'test_user'),
                password=os.getenv('DB_TEST_PASSWORD', 'test_password')
            )
            self.cursor = self.connection.cursor(dictionary=True)

            # Clean test data
```

```
            self.cleanup_test_data()

        except Error as e:
            pytest.fail(f"Failed to connect to test database: {e}")

    def teardown_method(self):
        """Cleanup after test"""
        if self.connection and self.connection.is_connected():
            self.cleanup_test_data()
            self.cursor.close()
            self.connection.close()

    def cleanup_test_data(self):
        """Remove test data"""
        cleanup_queries = [
            "DELETE FROM user_sessions WHERE user_id IN (SELECT id FROM users
WHERE email LIKE '%test%')",
            "DELETE FROM users WHERE email LIKE '%test%'"
        ]

        for query in cleanup_queries:
            self.cursor.execute(query)

        self.connection.commit()

    def test_user_creation_and_retrieval(self):
        """Test user creation and retrieval"""
        # Insert test user
        insert_query = """
        INSERT INTO users (email, name, preferences)
        VALUES (%s, %s, %s)
        """

        test_preferences = json.dumps({
            "theme": "dark",
            "notifications": True,
            "language": "en"
        })

        self.cursor.execute(insert_query, (
            'integration_test@example.com',
            'Integration Test User',
            test_preferences
        ))

        user_id = self.cursor.lastrowid
        self.connection.commit()

        # Retrieve user
        select_query = "SELECT * FROM users WHERE id = %s"
        self.cursor.execute(select_query, (user_id,))
        user = self.cursor.fetchone()

        # Assertions
```

```
        assert user is not None
        assert user['email'] == 'integration_test@example.com'
        assert user['name'] == 'Integration Test User'
        assert json.loads(user['preferences'])['theme'] == 'dark'
        assert user['created_at'] is not None
        assert user['updated_at'] is not None

    def test_user_session_workflow(self):
        """Test complete user session workflow"""
        # Create user
        insert_user_query = """
        INSERT INTO users (email, name)
        VALUES (%s, %s)
        """

        self.cursor.execute(insert_user_query, (
            'session_test@example.com',
            'Session Test User'
        ))

        user_id = self.cursor.lastrowid
        self.connection.commit()

        # Create session using stored procedure
        session_token = None
        create_session_query = "CALL create_user_session(%s, @session_token)"
        self.cursor.execute(create_session_query, (user_id,))

        # Get the session token
        self.cursor.execute("SELECT @session_token AS session_token")
        result = self.cursor.fetchone()
        session_token = result['session_token']

        self.connection.commit()

        # Verify session exists
        verify_query = """
        SELECT * FROM user_sessions
        WHERE user_id = %s AND session_token = %s
        """

        self.cursor.execute(verify_query, (user_id, session_token))
        session = self.cursor.fetchone()

        assert session is not None
        assert session['user_id'] == user_id
        assert session['session_token'] == session_token
        assert session['expires_at'] > datetime.now()

        # Validate session using function
        validate_query = "SELECT validate_user_session(%s) AS is_valid"
        self.cursor.execute(validate_query, (session_token,))
        validation_result = self.cursor.fetchone()
```

```
        assert validation_result['is_valid'] == 1

        # Test invalid session
        self.cursor.execute(validate_query, ('invalid_token',))
        invalid_result = self.cursor.fetchone()

        assert invalid_result['is_valid'] == 0

    def test_database_constraints(self):
        """Test database constraints and error handling"""
        # Test unique constraint on email
        insert_query = """
        INSERT INTO users (email, name)
        VALUES (%s, %s)
        """

        # Insert first user
        self.cursor.execute(insert_query, (
            'constraint_test@example.com',
            'Constraint Test User 1'
        ))
        self.connection.commit()

        # Try to insert duplicate email
        with pytest.raises(mysql.connector.IntegrityError):
            self.cursor.execute(insert_query, (
                'constraint_test@example.com',
                'Constraint Test User 2'
            ))
            self.connection.commit()

    def test_transaction_rollback(self):
        """Test transaction rollback functionality"""
        # Start transaction
        self.connection.start_transaction()

        try:
            # Insert user
            insert_user_query = """
            INSERT INTO users (email, name)
            VALUES (%s, %s)
            """

            self.cursor.execute(insert_user_query, (
                'transaction_test@example.com',
                'Transaction Test User'
            ))

            user_id = self.cursor.lastrowid

            # Insert session
            insert_session_query = """
            INSERT INTO user_sessions (user_id, session_token, expires_at)
            VALUES (%s, %s, %s)
```

```
            """

            self.cursor.execute(insert_session_query, (
                user_id,
                'test_session_token',
                datetime.now() + timedelta(hours=1)
            ))

            # Simulate error condition
            raise Exception("Simulated error")

        except Exception:
            # Rollback transaction
            self.connection.rollback()

        # Verify no data was inserted
        check_user_query = "SELECT COUNT(*) as count FROM users WHERE email = %s"
        self.cursor.execute(check_user_query, ('transaction_test@example.com',))
        user_count = self.cursor.fetchone()['count']

        check_session_query = "SELECT COUNT(*) as count FROM user_sessions WHERE
session_token = %s"
        self.cursor.execute(check_session_query, ('test_session_token',))
        session_count = self.cursor.fetchone()['count']

        assert user_count == 0
        assert session_count == 0

    def test_performance_with_indexes(self):
        """Test query performance with indexes"""
        import time

        # Insert test data
        insert_query = """
        INSERT INTO users (email, name)
        VALUES (%s, %s)
        """

        test_users = [
            (f'perf_test_{i}@example.com', f'Performance Test User {i}')
            for i in range(1000)
        ]

        self.cursor.executemany(insert_query, test_users)
        self.connection.commit()

        # Test query performance with index
        start_time = time.time()

        search_query = "SELECT * FROM users WHERE email = %s"
        self.cursor.execute(search_query, ('perf_test_500@example.com',))
        result = self.cursor.fetchone()

        end_time = time.time()
```

```
        query_time = end_time - start_time

        # Assert query completed quickly (should be fast with index)
        assert query_time < 0.1  # Less than 100ms
        assert result is not None
        assert result['email'] == 'perf_test_500@example.com'

# Run tests
if __name__ == '__main__':
    pytest.main([__file__, '-v'])
```

Performance Testing

```
# tests/test_database_performance.py
import pytest
import mysql.connector
import time
import statistics
from concurrent.futures import ThreadPoolExecutor, as_completed
import random
import string

class DatabasePerformanceTest:
    def __init__(self):
        self.connection_pool = None
        self.setup_connection_pool()

    def setup_connection_pool(self):
        """Setup connection pool for performance testing"""
        config = {
            'host': 'localhost',
            'database': 'myapp_test',
            'user': 'test_user',
            'password': 'test_password',
            'pool_name': 'performance_test_pool',
            'pool_size': 10,
            'pool_reset_session': True
        }

        self.connection_pool =
mysql.connector.pooling.MySQLConnectionPool(**config)

    def get_connection(self):
        """Get connection from pool"""
        return self.connection_pool.get_connection()

    def generate_test_data(self, count=10000):
        """Generate test data for performance testing"""
        connection = self.get_connection()
        cursor = connection.cursor()
```

```
        # Clean existing test data
        cursor.execute("DELETE FROM users WHERE email LIKE 'perf_%'")

        # Generate test users
        test_users = []
        for i in range(count):
            email = f'perf_{i}@example.com'
            name = f'Performance User {i}'
            test_users.append((email, name))

        # Batch insert
        insert_query = "INSERT INTO users (email, name) VALUES (%s, %s)"
        cursor.executemany(insert_query, test_users)

        connection.commit()
        cursor.close()
        connection.close()

        print(f"Generated {count} test users")

    def test_single_query_performance(self):
        """Test single query performance"""
        connection = self.get_connection()
        cursor = connection.cursor()

        # Test different query types
        queries = [
            ("SELECT * FROM users WHERE email = 'perf_5000@example.com'", "Indexed
lookup"),
            ("SELECT COUNT(*) FROM users WHERE name LIKE 'Performance%'", "Pattern
matching"),
            ("SELECT * FROM users ORDER BY created_at DESC LIMIT 100", "Ordered
limit"),
            ("SELECT name, COUNT(*) FROM users GROUP BY name HAVING COUNT(*) > 1",
"Aggregation")
        ]

        results = []

        for query, description in queries:
            times = []

            # Run query multiple times
            for _ in range(10):
                start_time = time.time()
                cursor.execute(query)
                cursor.fetchall()
                end_time = time.time()

                times.append(end_time - start_time)

            avg_time = statistics.mean(times)
            min_time = min(times)
            max_time = max(times)
```

```
            results.append({
                'description': description,
                'avg_time': avg_time,
                'min_time': min_time,
                'max_time': max_time,
                'query': query
            })

            print(f"{description}: Avg {avg_time:.4f}s, Min {min_time:.4f}s, Max
{max_time:.4f}s")

        cursor.close()
        connection.close()

        # Assert performance thresholds
        for result in results:
            assert result['avg_time'] < 1.0, f"Query too slow:
{result['description']}"

    def test_concurrent_query_performance(self):
        """Test concurrent query performance"""
        def execute_query(query_id):
            """Execute a query and return timing"""
            connection = self.get_connection()
            cursor = connection.cursor()

            start_time = time.time()

            # Random user lookup
            user_id = random.randint(1, 1000)
            cursor.execute("SELECT * FROM users WHERE email = %s",
(f'perf_{user_id}@example.com',))
            result = cursor.fetchone()

            end_time = time.time()

            cursor.close()
            connection.close()

            return {
                'query_id': query_id,
                'execution_time': end_time - start_time,
                'success': result is not None
            }

        # Test with different concurrency levels
        concurrency_levels = [1, 5, 10, 20]

        for concurrency in concurrency_levels:
            print(f"\nTesting with {concurrency} concurrent connections...")

            start_time = time.time()
```

```
            with ThreadPoolExecutor(max_workers=concurrency) as executor:
                futures = [executor.submit(execute_query, i) for i in range(100)]
                results = [future.result() for future in as_completed(futures)]

            end_time = time.time()
            total_time = end_time - start_time

            # Calculate statistics
            execution_times = [r['execution_time'] for r in results]
            success_count = sum(1 for r in results if r['success'])

            avg_query_time = statistics.mean(execution_times)
            throughput = len(results) / total_time

            print(f"  Total time: {total_time:.2f}s")
            print(f"  Average query time: {avg_query_time:.4f}s")
            print(f"  Throughput: {throughput:.2f} queries/second")
            print(f"  Success rate: {success_count}/{len(results)}
({success_count/len(results)*100:.1f}%)")

            # Assert performance thresholds
            assert success_count == len(results), "Some queries failed"
            assert avg_query_time < 0.5, f"Average query time too high:
{avg_query_time:.4f}s"

    def test_bulk_operations_performance(self):
        """Test bulk operations performance"""
        connection = self.get_connection()
        cursor = connection.cursor()

        # Test bulk insert
        print("Testing bulk insert performance...")

        bulk_data = []
        for i in range(1000):
            email = f'bulk_{i}@example.com'
            name = f'Bulk User {i}'
            bulk_data.append((email, name))

        start_time = time.time()

        insert_query = "INSERT INTO users (email, name) VALUES (%s, %s)"
        cursor.executemany(insert_query, bulk_data)
        connection.commit()

        end_time = time.time()
        insert_time = end_time - start_time

        print(f"Bulk insert of 1000 records: {insert_time:.2f}s
({1000/insert_time:.0f} records/second)")

        # Test bulk update
        print("Testing bulk update performance...")
```

```
        start_time = time.time()

        update_query = "UPDATE users SET name = CONCAT(name, ' - Updated') WHERE
email LIKE 'bulk_%'"
        cursor.execute(update_query)
        updated_rows = cursor.rowcount
        connection.commit()

        end_time = time.time()
        update_time = end_time - start_time

        print(f"Bulk update of {updated_rows} records: {update_time:.2f}s
({updated_rows/update_time:.0f} records/second)")

        # Test bulk delete
        print("Testing bulk delete performance...")

        start_time = time.time()

        delete_query = "DELETE FROM users WHERE email LIKE 'bulk_%'"
        cursor.execute(delete_query)
        deleted_rows = cursor.rowcount
        connection.commit()

        end_time = time.time()
        delete_time = end_time - start_time

        print(f"Bulk delete of {deleted_rows} records: {delete_time:.2f}s
({deleted_rows/delete_time:.0f} records/second)")

        cursor.close()
        connection.close()

        # Assert performance thresholds
        assert insert_time < 5.0, f"Bulk insert too slow: {insert_time:.2f}s"
        assert update_time < 2.0, f"Bulk update too slow: {update_time:.2f}s"
        assert delete_time < 1.0, f"Bulk delete too slow: {delete_time:.2f}s"

# Run performance tests
if __name__ == '__main__':
    perf_test = DatabasePerformanceTest()

    print("Setting up test data...")
    perf_test.generate_test_data(10000)

    print("\n=== Single Query Performance Tests ===")
    perf_test.test_single_query_performance()

    print("\n=== Concurrent Query Performance Tests ===")
    perf_test.test_concurrent_query_performance()

    print("\n=== Bulk Operations Performance Tests ===")
    perf_test.test_bulk_operations_performance()
```

```
    print("\nPerformance testing completed!")
```

🚀 CI/CD Pipeline Implementation

GitHub Actions Workflow

```yaml
# .github/workflows/database-ci-cd.yml
name: Database CI/CD Pipeline

on:
  push:
    branches: [main, develop]
    paths:
      - "migrations/**"
      - "database/**"
      - ".github/workflows/database-ci-cd.yml"
  pull_request:
    branches: [main]
    paths:
      - "migrations/**"
      - "database/**"

env:
  MYSQL_ROOT_PASSWORD: root_password
  MYSQL_DATABASE: myapp_test
  MYSQL_USER: test_user
  MYSQL_PASSWORD: test_password

jobs:
  database-lint:
    runs-on: ubuntu-latest
    name: Database Linting

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.9"

      - name: Install SQL linting tools
        run: |
          pip install sqlfluff
          pip install sqlparse

      - name: Lint SQL files
        run: |
```

```
          sqlfluff lint migrations/ --dialect mysql
          find migrations/ -name "*.sql" -exec sqlparse --format {} \;

      - name: Check migration naming convention
        run: |
          python scripts/check_migration_naming.py

  database-test:
    runs-on: ubuntu-latest
    name: Database Testing

    services:
      mysql:
        image: mysql:8.0
        env:
          MYSQL_ROOT_PASSWORD: ${{ env.MYSQL_ROOT_PASSWORD }}
          MYSQL_DATABASE: ${{ env.MYSQL_DATABASE }}
          MYSQL_USER: ${{ env.MYSQL_USER }}
          MYSQL_PASSWORD: ${{ env.MYSQL_PASSWORD }}
        ports:
          - 3306:3306
        options: --health-cmd="mysqladmin ping" --health-interval=10s --health-
timeout=5s --health-retries=3

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Java (for Flyway)
        uses: actions/setup-java@v3
        with:
          distribution: "temurin"
          java-version: "11"

      - name: Install Flyway
        run: |
          wget -qO- https://repo1.maven.org/maven2/org/flywaydb/flyway-
commandline/9.8.1/flyway-commandline-9.8.1-linux-x64.tar.gz | tar xvz
          sudo ln -s `pwd`/flyway-9.8.1/flyway /usr/local/bin

      - name: Wait for MySQL
        run: |
          while ! mysqladmin ping -h"127.0.0.1" -P3306 -u${{ env.MYSQL_USER }} -
p${{ env.MYSQL_PASSWORD }} --silent; do
            sleep 1
          done

      - name: Run database migrations
        run: |
          flyway -url=jdbc:mysql://localhost:3306/${{ env.MYSQL_DATABASE }} \
                 -user=${{ env.MYSQL_USER }} \
                 -password=${{ env.MYSQL_PASSWORD }} \
                 -locations=filesystem:migrations \
                 migrate
```

```
      - name: Setup Python for testing
        uses: actions/setup-python@v4
        with:
          python-version: "3.9"

      - name: Install Python dependencies
        run: |
          pip install -r requirements-test.txt

      - name: Run database unit tests
        run: |
          python -m pytest tests/test_database_unit.py -v
        env:
          DB_TEST_HOST: localhost
          DB_TEST_NAME: ${{ env.MYSQL_DATABASE }}
          DB_TEST_USER: ${{ env.MYSQL_USER }}
          DB_TEST_PASSWORD: ${{ env.MYSQL_PASSWORD }}

      - name: Run database integration tests
        run: |
          python -m pytest tests/test_database_integration.py -v
        env:
          DB_TEST_HOST: localhost
          DB_TEST_NAME: ${{ env.MYSQL_DATABASE }}
          DB_TEST_USER: ${{ env.MYSQL_USER }}
          DB_TEST_PASSWORD: ${{ env.MYSQL_PASSWORD }}

      - name: Run database performance tests
        run: |
          python -m pytest tests/test_database_performance.py -v
        env:
          DB_TEST_HOST: localhost
          DB_TEST_NAME: ${{ env.MYSQL_DATABASE }}
          DB_TEST_USER: ${{ env.MYSQL_USER }}
          DB_TEST_PASSWORD: ${{ env.MYSQL_PASSWORD }}

      - name: Generate test coverage report
        run: |
          python scripts/generate_db_coverage_report.py

      - name: Upload test results
        uses: actions/upload-artifact@v3
        if: always()
        with:
          name: test-results
          path: |
            test-results/
            coverage-reports/

  database-security-scan:
    runs-on: ubuntu-latest
    name: Database Security Scan
```

```
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Run SQL injection vulnerability scan
        run: |
          python scripts/scan_sql_injection.py migrations/

      - name: Check for sensitive data in migrations
        run: |
          python scripts/check_sensitive_data.py migrations/

      - name: Validate database permissions
        run: |
          python scripts/validate_db_permissions.py

  deploy-staging:
    runs-on: ubuntu-latest
    name: Deploy to Staging
    needs: [database-lint, database-test, database-security-scan]
    if: github.ref == 'refs/heads/develop'

    environment:
      name: staging
      url: https://staging.myapp.com

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Java (for Flyway)
        uses: actions/setup-java@v3
        with:
          distribution: "temurin"
          java-version: "11"

      - name: Install Flyway
        run: |
          wget -qO- https://repo1.maven.org/maven2/org/flywaydb/flyway-
commandline/9.8.1/flyway-commandline-9.8.1-linux-x64.tar.gz | tar xvz
          sudo ln -s `pwd`/flyway-9.8.1/flyway /usr/local/bin

      - name: Create database backup
        run: |
          python scripts/create_backup.py staging
        env:
          DB_STAGING_HOST: ${{ secrets.DB_STAGING_HOST }}
          DB_STAGING_NAME: ${{ secrets.DB_STAGING_NAME }}
          DB_STAGING_USER: ${{ secrets.DB_STAGING_USER }}
          DB_STAGING_PASSWORD: ${{ secrets.DB_STAGING_PASSWORD }}

      - name: Deploy to staging database
        run: |
          flyway -url=jdbc:mysql://${{ secrets.DB_STAGING_HOST }}:3306/${{
```

```
  secrets.DB_STAGING_NAME }} \
                  -user=${{ secrets.DB_STAGING_USER }} \
                  -password=${{ secrets.DB_STAGING_PASSWORD }} \
                  -locations=filesystem:migrations \
                  -validateOnMigrate=true \
                  migrate

      - name: Run post-deployment tests
        run: |
          python scripts/post_deployment_tests.py staging
        env:
          DB_STAGING_HOST: ${{ secrets.DB_STAGING_HOST }}
          DB_STAGING_NAME: ${{ secrets.DB_STAGING_NAME }}
          DB_STAGING_USER: ${{ secrets.DB_STAGING_USER }}
          DB_STAGING_PASSWORD: ${{ secrets.DB_STAGING_PASSWORD }}

      - name: Notify deployment status
        uses: 8398a7/action-slack@v3
        with:
          status: ${{ job.status }}
          channel: "#deployments"
          webhook_url: ${{ secrets.SLACK_WEBHOOK }}

  deploy-production:
    runs-on: ubuntu-latest
    name: Deploy to Production
    needs: [database-lint, database-test, database-security-scan]
    if: github.ref == 'refs/heads/main'

    environment:
      name: production
      url: https://myapp.com

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Java (for Flyway)
        uses: actions/setup-java@v3
        with:
          distribution: "temurin"
          java-version: "11"

      - name: Install Flyway
        run: |
          wget -qO- https://repo1.maven.org/maven2/org/flywaydb/flyway-
commandline/9.8.1/flyway-commandline-9.8.1-linux-x64.tar.gz | tar xvz
          sudo ln -s `pwd`/flyway-9.8.1/flyway /usr/local/bin

      - name: Create database backup
        run: |
          python scripts/create_backup.py production
        env:
          DB_PROD_HOST: ${{ secrets.DB_PROD_HOST }}
```

```
          DB_PROD_NAME: ${{ secrets.DB_PROD_NAME }}
          DB_PROD_USER: ${{ secrets.DB_PROD_USER }}
          DB_PROD_PASSWORD: ${{ secrets.DB_PROD_PASSWORD }}

      - name: Deploy to production database (with approval)
        run: |
          echo "Deploying to production database..."
          flyway -url=jdbc:mysql://${{ secrets.DB_PROD_HOST }}:3306/${{
secrets.DB_PROD_NAME }} \
                 -user=${{ secrets.DB_PROD_USER }} \
                 -password=${{ secrets.DB_PROD_PASSWORD }} \
                 -locations=filesystem:migrations \
                 -validateOnMigrate=true \
                 -outOfOrder=false \
                 migrate

      - name: Run post-deployment tests
        run: |
          python scripts/post_deployment_tests.py production
        env:
          DB_PROD_HOST: ${{ secrets.DB_PROD_HOST }}
          DB_PROD_NAME: ${{ secrets.DB_PROD_NAME }}
          DB_PROD_USER: ${{ secrets.DB_PROD_USER }}
          DB_PROD_PASSWORD: ${{ secrets.DB_PROD_PASSWORD }}

      - name: Update monitoring dashboards
        run: |
          python scripts/update_monitoring.py production
        env:
          GRAFANA_API_KEY: ${{ secrets.GRAFANA_API_KEY }}
          DATADOG_API_KEY: ${{ secrets.DATADOG_API_KEY }}

      - name: Notify deployment status
        uses: 8398a7/action-slack@v3
        with:
          status: ${{ job.status }}
          channel: "#deployments"
          webhook_url: ${{ secrets.SLACK_WEBHOOK }}
```

Jenkins Pipeline

```
// Jenkinsfile
pipeline {
    agent any

    environment {
        MYSQL_ROOT_PASSWORD = credentials('mysql-root-password')
        DB_TEST_PASSWORD = credentials('db-test-password')
        DB_STAGING_PASSWORD = credentials('db-staging-password')
        DB_PROD_PASSWORD = credentials('db-prod-password')
    }
```

```
    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Database Lint') {
            steps {
                script {
                    sh '''
                        pip install sqlfluff
                        sqlfluff lint migrations/ --dialect mysql
                    '''
                }
            }
        }

        stage('Setup Test Database') {
            steps {
                script {
                    sh '''
                        docker run -d --name mysql-test \
                            -e MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD} \
                            -e MYSQL_DATABASE=myapp_test \
                            -e MYSQL_USER=test_user \
                            -e MYSQL_PASSWORD=${DB_TEST_PASSWORD} \
                            -p 3307:3306 \
                            mysql:8.0

                        # Wait for MySQL to be ready
                        sleep 30

                        # Test connection
                        docker exec mysql-test mysqladmin ping -h localhost -u
test_user -p${DB_TEST_PASSWORD}
                    '''
                }
            }
        }

        stage('Run Migrations') {
            steps {
                script {
                    sh '''
                        flyway -url=jdbc:mysql://localhost:3307/myapp_test \
                               -user=test_user \
                               -password=${DB_TEST_PASSWORD} \
                               -locations=filesystem:migrations \
                               migrate
                    '''
                }
            }
```

```
        }

        stage('Database Tests') {
            parallel {
                stage('Unit Tests') {
                    steps {
                        script {
                            sh '''
                                export DB_TEST_HOST=localhost
                                export DB_TEST_PORT=3307
                                export DB_TEST_NAME=myapp_test
                                export DB_TEST_USER=test_user
                                export DB_TEST_PASSWORD=${DB_TEST_PASSWORD}

                                python -m pytest tests/test_database_unit.py -v --
junitxml=test-results/unit-tests.xml
                            '''
                        }
                    }
                }

                stage('Integration Tests') {
                    steps {
                        script {
                            sh '''
                                export DB_TEST_HOST=localhost
                                export DB_TEST_PORT=3307
                                export DB_TEST_NAME=myapp_test
                                export DB_TEST_USER=test_user
                                export DB_TEST_PASSWORD=${DB_TEST_PASSWORD}

                                python -m pytest
tests/test_database_integration.py -v --junitxml=test-results/integration-
tests.xml
                            '''
                        }
                    }
                }

                stage('Performance Tests') {
                    steps {
                        script {
                            sh '''
                                export DB_TEST_HOST=localhost
                                export DB_TEST_PORT=3307
                                export DB_TEST_NAME=myapp_test
                                export DB_TEST_USER=test_user
                                export DB_TEST_PASSWORD=${DB_TEST_PASSWORD}

                                python -m pytest
tests/test_database_performance.py -v --junitxml=test-results/performance-
tests.xml
                            '''
                        }
```

```
                }
            }
        }
    }

    stage('Security Scan') {
        steps {
            script {
                sh '''
                    python scripts/scan_sql_injection.py migrations/
                    python scripts/check_sensitive_data.py migrations/
                '''
            }
        }
    }

    stage('Deploy to Staging') {
        when {
            branch 'develop'
        }
        steps {
            script {
                sh '''
                    # Create backup
                    python scripts/create_backup.py staging

                    # Deploy migrations
                    flyway -url=jdbc:mysql://staging-db:3306/myapp_staging \
                           -user=staging_user \
                           -password=${DB_STAGING_PASSWORD} \
                           -locations=filesystem:migrations \
                           migrate

                    # Run post-deployment tests
                    python scripts/post_deployment_tests.py staging
                '''
            }
        }
    }

    stage('Deploy to Production') {
        when {
            branch 'main'
        }
        steps {
            script {
                // Require manual approval for production
                input message: 'Deploy to production?', ok: 'Deploy',
                      submitterParameter: 'DEPLOYER'

                sh '''
                    echo "Deploying to production (approved by ${DEPLOYER})"

                    # Create backup
```

```
                        python scripts/create_backup.py production

                        # Deploy migrations
                        flyway -url=jdbc:mysql://prod-db:3306/myapp_production \
                               -user=prod_user \
                               -password=${DB_PROD_PASSWORD} \
                               -locations=filesystem:migrations \
                               -validateOnMigrate=true \
                               migrate

                        # Run post-deployment tests
                        python scripts/post_deployment_tests.py production

                        # Update monitoring
                        python scripts/update_monitoring.py production
                    '''
                }
            }
        }
    }

    post {
        always {
            // Cleanup test database
            script {
                sh 'docker stop mysql-test || true'
                sh 'docker rm mysql-test || true'
            }

            // Publish test results
            junit 'test-results/*.xml'

            // Archive artifacts
            archiveArtifacts artifacts: 'test-results/*, coverage-reports/*',
allowEmptyArchive: true
        }

        success {
            // Notify success
            slackSend channel: '#deployments',
                      color: 'good',
                      message: "Database deployment successful: ${env.JOB_NAME} -
${env.BUILD_NUMBER}"
        }

        failure {
            // Notify failure
            slackSend channel: '#deployments',
                      color: 'danger',
                      message: "Database deployment failed: ${env.JOB_NAME} -
${env.BUILD_NUMBER}"
        }
    }
}
```

☎ Infrastructure as Code (IaC)

Terraform for Database Infrastructure

```
# terraform/database.tf
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

# Variables
variable "aws_region" {
  description = "AWS region"
  type        = string
  default     = "us-east-1"
}

variable "environment" {
  description = "Environment name"
  type        = string
}

variable "project_name" {
  description = "Project name"
  type        = string
  default     = "myapp"
}

variable "db_instance_class" {
  description = "RDS instance class"
  type        = string
  default     = "db.t3.micro"
}

variable "db_allocated_storage" {
  description = "RDS allocated storage"
  type        = number
  default     = 20
}
```

```
variable "db_max_allocated_storage" {
  description = "RDS max allocated storage"
  type        = number
  default     = 100
}

variable "backup_retention_period" {
  description = "Backup retention period"
  type        = number
  default     = 7
}

variable "multi_az" {
  description = "Enable Multi-AZ deployment"
  type        = bool
  default     = false
}

# Data sources
data "aws_availability_zones" "available" {
  state = "available"
}

data "aws_caller_identity" "current" {}

# VPC and Networking
resource "aws_vpc" "main" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name        = "${var.project_name}-${var.environment}-vpc"
    Environment = var.environment
  }
}

resource "aws_subnet" "private" {
  count             = 2
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.${count.index + 1}.0/24"
  availability_zone = data.aws_availability_zones.available.names[count.index]

  tags = {
    Name        = "${var.project_name}-${var.environment}-private-
subnet-${count.index + 1}"
    Environment = var.environment
  }
}

resource "aws_subnet" "public" {
  count                   = 2
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.${count.index + 10}.0/24"
```

```
  availability_zone       =
data.aws_availability_zones.available.names[count.index]
  map_public_ip_on_launch = true

  tags = {
    Name        = "${var.project_name}-${var.environment}-public-
subnet-${count.index + 1}"
    Environment = var.environment
  }
}

resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name        = "${var.project_name}-${var.environment}-igw"
    Environment = var.environment
  }
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }

  tags = {
    Name        = "${var.project_name}-${var.environment}-public-rt"
    Environment = var.environment
  }
}

resource "aws_route_table_association" "public" {
  count          = length(aws_subnet.public)
  subnet_id      = aws_subnet.public[count.index].id
  route_table_id = aws_route_table.public.id
}

# Security Groups
resource "aws_security_group" "rds" {
  name_prefix = "${var.project_name}-${var.environment}-rds-"
  vpc_id      = aws_vpc.main.id

  ingress {
    from_port       = 3306
    to_port         = 3306
    protocol        = "tcp"
    security_groups = [aws_security_group.app.id]
  }

  egress {
    from_port   = 0
```

```
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name        = "${var.project_name}-${var.environment}-rds-sg"
    Environment = var.environment
  }
}

resource "aws_security_group" "app" {
  name_prefix = "${var.project_name}-${var.environment}-app-"
  vpc_id      = aws_vpc.main.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name        = "${var.project_name}-${var.environment}-app-sg"
    Environment = var.environment
  }
}

# RDS Subnet Group
resource "aws_db_subnet_group" "main" {
  name       = "${var.project_name}-${var.environment}-db-subnet-group"
  subnet_ids = aws_subnet.private[*].id

  tags = {
    Name        = "${var.project_name}-${var.environment}-db-subnet-group"
    Environment = var.environment
  }
}

# RDS Parameter Group
```

```
resource "aws_db_parameter_group" "main" {
  family = "mysql8.0"
  name   = "${var.project_name}-${var.environment}-db-params"

  parameter {
    name  = "innodb_buffer_pool_size"
    value = "{DBInstanceClassMemory*3/4}"
  }

  parameter {
    name  = "slow_query_log"
    value = "1"
  }

  parameter {
    name  = "long_query_time"
    value = "2"
  }

  parameter {
    name  = "general_log"
    value = "0"
  }

  tags = {
    Name        = "${var.project_name}-${var.environment}-db-params"
    Environment = var.environment
  }
}

# KMS Key for RDS Encryption
resource "aws_kms_key" "rds" {
  description             = "KMS key for RDS encryption"
  deletion_window_in_days = 7

  tags = {
    Name        = "${var.project_name}-${var.environment}-rds-key"
    Environment = var.environment
  }
}

resource "aws_kms_alias" "rds" {
  name          = "alias/${var.project_name}-${var.environment}-rds"
  target_key_id = aws_kms_key.rds.key_id
}

# Secrets Manager for Database Credentials
resource "aws_secretsmanager_secret" "db_credentials" {
  name                    =
"${var.project_name}/${var.environment}/database/credentials"
  description             = "Database credentials for ${var.project_name}
${var.environment}"
  recovery_window_in_days = 7
```

```
  tags = {
    Name        = "${var.project_name}-${var.environment}-db-credentials"
    Environment = var.environment
  }
}

resource "aws_secretsmanager_secret_version" "db_credentials" {
  secret_id = aws_secretsmanager_secret.db_credentials.id
  secret_string = jsonencode({
    username = "admin"
    password = random_password.db_password.result
  })
}

resource "random_password" "db_password" {
  length  = 32
  special = true
}

# RDS Instance
resource "aws_db_instance" "main" {
  identifier = "${var.project_name}-${var.environment}-db"

  # Engine
  engine         = "mysql"
  engine_version = "8.0.35"
  instance_class = var.db_instance_class

  # Storage
  allocated_storage     = var.db_allocated_storage
  max_allocated_storage = var.db_max_allocated_storage
  storage_type          = "gp2"
  storage_encrypted     = true
  kms_key_id           = aws_kms_key.rds.arn

  # Database
  db_name  = "${var.project_name}_${var.environment}"
  username = "admin"
  password = random_password.db_password.result

  # Network
  db_subnet_group_name   = aws_db_subnet_group.main.name
  vpc_security_group_ids = [aws_security_group.rds.id]
  publicly_accessible    = false

  # Backup
  backup_retention_period = var.backup_retention_period
  backup_window          = "03:00-04:00"
  maintenance_window     = "sun:04:00-sun:05:00"
  delete_automated_backups = false

  # High Availability
  multi_az = var.multi_az
```

```
  # Monitoring
  monitoring_interval = 60
  monitoring_role_arn = aws_iam_role.rds_monitoring.arn
  enabled_cloudwatch_logs_exports = ["error", "general", "slow"]

  # Performance Insights
  performance_insights_enabled = true
  performance_insights_kms_key_id = aws_kms_key.rds.arn
  performance_insights_retention_period = 7

  # Parameter Group
  parameter_group_name = aws_db_parameter_group.main.name

  # Deletion Protection
  deletion_protection = var.environment == "production" ? true : false
  skip_final_snapshot = var.environment == "production" ? false : true
  final_snapshot_identifier = var.environment == "production" ?
"${var.project_name}-${var.environment}-final-snapshot-${formatdate("YYYY-MM-DD-
hhmm", timestamp())}" : null

  tags = {
    Name        = "${var.project_name}-${var.environment}-db"
    Environment = var.environment
  }
}

# IAM Role for RDS Monitoring
resource "aws_iam_role" "rds_monitoring" {
  name = "${var.project_name}-${var.environment}-rds-monitoring-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "monitoring.rds.amazonaws.com"
        }
      }
    ]
  })

  tags = {
    Name        = "${var.project_name}-${var.environment}-rds-monitoring-role"
    Environment = var.environment
  }
}

resource "aws_iam_role_policy_attachment" "rds_monitoring" {
  role       = aws_iam_role.rds_monitoring.name
  policy_arn = "arn:aws:iam::aws:policy/service-
role/AmazonRDSEnhancedMonitoringRole"
}
```

```
# Read Replica (for production)
resource "aws_db_instance" "read_replica" {
  count = var.environment == "production" ? 1 : 0

  identifier = "${var.project_name}-${var.environment}-db-replica"

  replicate_source_db = aws_db_instance.main.identifier
  instance_class      = var.db_instance_class

  publicly_accessible = false
  monitoring_interval = 60
  monitoring_role_arn = aws_iam_role.rds_monitoring.arn

  performance_insights_enabled = true
  performance_insights_kms_key_id = aws_kms_key.rds.arn

  tags = {
    Name        = "${var.project_name}-${var.environment}-db-replica"
    Environment = var.environment
  }
}

# Outputs
output "rds_endpoint" {
  description = "RDS instance endpoint"
  value       = aws_db_instance.main.endpoint
  sensitive   = true
}

output "rds_port" {
  description = "RDS instance port"
  value       = aws_db_instance.main.port
}

output "database_name" {
  description = "Database name"
  value       = aws_db_instance.main.db_name
}

output "secret_arn" {
  description = "Secrets Manager secret ARN"
  value       = aws_secretsmanager_secret.db_credentials.arn
}

output "read_replica_endpoint" {
  description = "Read replica endpoint"
  value       = var.environment == "production" ?
aws_db_instance.read_replica[0].endpoint : null
  sensitive   = true
}
```

Ansible for Database Configuration

```yaml
# ansible/database-setup.yml
---
- name: Database Setup and Configuration
  hosts: database_servers
  become: yes
  vars:
    mysql_root_password: "{{ vault_mysql_root_password }}"
    mysql_databases:
      - name: myapp_production
        encoding: utf8mb4
        collation: utf8mb4_unicode_ci
      - name: myapp_staging
        encoding: utf8mb4
        collation: utf8mb4_unicode_ci
    mysql_users:
      - name: app_user
        password: "{{ vault_app_user_password }}"
        priv: "myapp_production.*:ALL"
        host: "%"
      - name: readonly_user
        password: "{{ vault_readonly_password }}"
        priv: "myapp_production.*:SELECT"
        host: "%"

  tasks:
    - name: Install MySQL server
      package:
        name: mysql-server
        state: present

    - name: Start and enable MySQL service
      service:
        name: mysql
        state: started
        enabled: yes

    - name: Set MySQL root password
      mysql_user:
        name: root
        password: "{{ mysql_root_password }}"
        login_unix_socket: /var/run/mysqld/mysqld.sock

    - name: Create MySQL configuration file
      template:
        src: my.cnf.j2
        dest: /etc/mysql/mysql.conf.d/custom.cnf
        backup: yes
      notify: restart mysql

    - name: Create databases
```

```
      mysql_db:
        name: "{{ item.name }}"
        encoding: "{{ item.encoding }}"
        collation: "{{ item.collation }}"
        state: present
        login_user: root
        login_password: "{{ mysql_root_password }}"
      loop: "{{ mysql_databases }}"

    - name: Create MySQL users
      mysql_user:
        name: "{{ item.name }}"
        password: "{{ item.password }}"
        priv: "{{ item.priv }}"
        host: "{{ item.host }}"
        state: present
        login_user: root
        login_password: "{{ mysql_root_password }}"
      loop: "{{ mysql_users }}"

    - name: Install Flyway
      unarchive:
        src: https://repo1.maven.org/maven2/org/flywaydb/flyway-
commandline/9.8.1/flyway-commandline-9.8.1-linux-x64.tar.gz
        dest: /opt
        remote_src: yes
        creates: /opt/flyway-9.8.1

    - name: Create Flyway symlink
      file:
        src: /opt/flyway-9.8.1/flyway
        dest: /usr/local/bin/flyway
        state: link

    - name: Create database backup script
      template:
        src: backup-database.sh.j2
        dest: /usr/local/bin/backup-database.sh
        mode: "0755"

    - name: Schedule database backups
      cron:
        name: "Database backup"
        minute: "0"
        hour: "2"
        job: "/usr/local/bin/backup-database.sh"

  handlers:
    - name: restart mysql
      service:
        name: mysql
        state: restarted
```

📊 Monitoring and Observability

Database Monitoring with Prometheus and Grafana

```
# docker-compose.monitoring.yml
version: "3.8"

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"
      - "--storage.tsdb.path=/prometheus"
      - "--web.console.libraries=/etc/prometheus/console_libraries"
      - "--web.console.templates=/etc/prometheus/consoles"
      - "--storage.tsdb.retention.time=200h"
      - "--web.enable-lifecycle"

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin123
    volumes:
      - grafana_data:/var/lib/grafana
      - ./grafana/provisioning:/etc/grafana/provisioning
      - ./grafana/dashboards:/var/lib/grafana/dashboards

  mysql-exporter:
    image: prom/mysqld-exporter:latest
    container_name: mysql-exporter
    ports:
      - "9104:9104"
    environment:
      - DATA_SOURCE_NAME=exporter:exporter_password@(mysql:3306)/
    depends_on:
      - mysql

  mysql:
    image: mysql:8.0
    container_name: mysql
    environment:
      - MYSQL_ROOT_PASSWORD=root_password
```

```
      - MYSQL_DATABASE=myapp_dev
      - MYSQL_USER=app_user
      - MYSQL_PASSWORD=app_password
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
      - ./mysql/init:/docker-entrypoint-initdb.d

volumes:
  prometheus_data:
  grafana_data:
  mysql_data:
```

```
# prometheus/prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - "database_rules.yml"

scrape_configs:
  - job_name: "mysql"
    static_configs:
      - targets: ["mysql-exporter:9104"]
    scrape_interval: 5s
    metrics_path: /metrics

  - job_name: "application"
    static_configs:
      - targets: ["app:8080"]
    scrape_interval: 15s

alerting:
  alertmanagers:
    - static_configs:
        - targets:
            - alertmanager:9093
```

```
# prometheus/database_rules.yml
groups:
  - name: database.rules
    rules:
      - alert: MySQLDown
        expr: mysql_up == 0
        for: 0m
        labels:
          severity: critical
```

```
        annotations:
          summary: MySQL instance is down
          description: "MySQL database is down on {{ $labels.instance }}"

      - alert: MySQLHighConnections
        expr: mysql_global_status_threads_connected /
mysql_global_variables_max_connections * 100 > 80
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: MySQL high connections
          description: "MySQL connections are above 80% ({{ $value }}%)"

      - alert: MySQLSlowQueries
        expr: increase(mysql_global_status_slow_queries[5m]) > 10
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: MySQL slow queries detected
          description: "{{ $value }} slow queries detected in the last 5 minutes"

      - alert: MySQLHighQPS
        expr: rate(mysql_global_status_queries[5m]) > 1000
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: MySQL high QPS
          description: "MySQL QPS is {{ $value }} queries per second"

      - alert: MySQLReplicationLag
        expr: mysql_slave_lag_seconds > 30
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: MySQL replication lag
          description: "MySQL replication lag is {{ $value }} seconds"

      - alert: MySQLInnoDBLogWaits
        expr: rate(mysql_global_status_innodb_log_waits[5m]) > 10
        for: 0m
        labels:
          severity: warning
        annotations:
          summary: MySQL InnoDB log waits
          description: "MySQL InnoDB log waits: {{ $value }} per second"
```

Application Performance Monitoring

```python
# monitoring/database_metrics.py
import time
import mysql.connector
from prometheus_client import Counter, Histogram, Gauge, start_http_server
import threading
from contextlib import contextmanager

class DatabaseMetrics:
    def __init__(self, db_config):
        self.db_config = db_config

        # Prometheus metrics
        self.query_counter = Counter(
            'database_queries_total',
            'Total number of database queries',
            ['query_type', 'status']
        )

        self.query_duration = Histogram(
            'database_query_duration_seconds',
            'Database query duration in seconds',
            ['query_type']
        )

        self.connection_pool_size = Gauge(
            'database_connection_pool_size',
            'Current database connection pool size'
        )

        self.active_connections = Gauge(
            'database_active_connections',
            'Number of active database connections'
        )

        self.slow_queries = Counter(
            'database_slow_queries_total',
            'Total number of slow queries'
        )

        # Start metrics collection
        self.start_metrics_collection()

    @contextmanager
    def measure_query(self, query_type):
        """Context manager to measure query performance"""
        start_time = time.time()
        status = 'success'

        try:
            yield
        except Exception as e:
            status = 'error'
            raise
```

```
        finally:
            duration = time.time() - start_time

            # Record metrics
            self.query_counter.labels(query_type=query_type, status=status).inc()
            self.query_duration.labels(query_type=query_type).observe(duration)

            # Track slow queries (> 1 second)
            if duration > 1.0:
                self.slow_queries.inc()

    def get_database_connection(self):
        """Get database connection with metrics"""
        try:
            connection = mysql.connector.connect(**self.db_config)
            self.active_connections.inc()
            return connection
        except Exception as e:
            self.query_counter.labels(query_type='connection',
status='error').inc()
            raise

    def close_connection(self, connection):
        """Close database connection with metrics"""
        if connection and connection.is_connected():
            connection.close()
            self.active_connections.dec()

    def collect_database_metrics(self):
        """Collect database-specific metrics"""
        try:
            connection = mysql.connector.connect(**self.db_config)
            cursor = connection.cursor(dictionary=True)

            # Get connection count
            cursor.execute("SHOW STATUS LIKE 'Threads_connected'")
            result = cursor.fetchone()
            if result:
                self.active_connections.set(int(result['Value']))

            # Get slow query count
            cursor.execute("SHOW STATUS LIKE 'Slow_queries'")
            result = cursor.fetchone()
            if result:
                # This is cumulative, so we track the rate
                pass

            cursor.close()
            connection.close()

        except Exception as e:
            print(f"Error collecting database metrics: {e}")

    def start_metrics_collection(self):
```

```
        """Start background metrics collection"""
        def collect_metrics():
            while True:
                self.collect_database_metrics()
                time.sleep(30)  # Collect every 30 seconds

        thread = threading.Thread(target=collect_metrics, daemon=True)
        thread.start()

# Usage example
if __name__ == '__main__':
    db_config = {
        'host': 'localhost',
        'database': 'myapp_production',
        'user': 'app_user',
        'password': 'app_password'
    }

    # Initialize metrics
    metrics = DatabaseMetrics(db_config)

    # Start Prometheus metrics server
    start_http_server(8000)

    # Example usage in application
    connection = metrics.get_database_connection()
    cursor = connection.cursor()

    with metrics.measure_query('select'):
        cursor.execute("SELECT * FROM users WHERE id = %s", (1,))
        result = cursor.fetchone()

    cursor.close()
    metrics.close_connection(connection)

    print("Metrics server started on port 8000")

    # Keep the script running
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("Shutting down metrics server")
```

Next Steps

Congratulations! You've completed Chapter 23 on Database DevOps & CI/CD. You should now understand:

✅ Database DevOps principles and practices ✅ Database version control with Flyway and Liquibase ✅ Git-based database workflows ✅ Database testing strategies (unit, integration, performance) ✅ CI/CD pipeline implementation for databases ✅ Infrastructure as Code (IaC) with Terraform and Ansible ✅ Database

monitoring and observability ✅ Automated deployment and rollback strategies ✅ Security scanning and compliance in database pipelines

🎉 **Congratulations! You have completed the comprehensive SQL learning system!**

You have now mastered:

- **Basic SQL**: Data types, CRUD operations, constraints, queries
- **Intermediate SQL**: Joins, functions, subqueries, window functions
- **Advanced SQL**: Performance optimization, stored procedures, triggers, views
- **Enterprise SQL**: Transactions, security, backup/recovery, replication
- **Modern SQL**: Cloud databases, ETL, partitioning, sharding
- **DevOps SQL**: CI/CD, monitoring, infrastructure as code

Practice Projects:

1. Set up a complete CI/CD pipeline for a database project
2. Implement database monitoring with Prometheus and Grafana
3. Create Infrastructure as Code for a multi-environment database setup
4. Build automated testing suite for database changes
5. Implement database security scanning and compliance checks

Additional Resources:

- Database DevOps best practices
- CI/CD tools comparison (Jenkins, GitHub Actions, GitLab CI)
- Infrastructure as Code tools (Terraform, Ansible, CloudFormation)
- Database monitoring and observability platforms
- Database security and compliance frameworks

Career Paths:

- Database Administrator (DBA)
- DevOps Engineer
- Site Reliability Engineer (SRE)
- Database Developer
- Data Engineer
- Cloud Database Architect

Next Learning Recommendations:

- Kubernetes for database orchestration
- Advanced cloud database services
- Data engineering and big data technologies
- Machine learning with databases
- Microservices database patterns