COMPLETE GUIDE

SQL

Complete **Beginner** to **Advanced** Guide

🔧 Fundamentals

RDBMS, Schema Design, Keys & Relationships

📊 Queries

SELECT, WHERE, JOIN, Subqueries

⚡ Advanced

Functions, Aggregations, Optimization

💼 Practice

Real Examples & Exercises

BEGINNER INTERMEDIATE ADVANCED

COMPREHENSIVE TUTORIAL

SQL Tutorial: Complete Beginner to Advanced Guide

Table of Contents

Key Database Concepts

1. What is RDBMS and Schema?

What is RDBMS (Relational Database Management System)?

An **RDBMS** stands for Relational Database Management System. It is a type of database management system that stores and manages data in the form of **tables**, which are related to each other through **relationships**.

RDBMS allows data to be stored in a relational format, meaning that data can be linked across multiple tables using relationships. These relationships are established using keys, such as **Primary Keys (PK)** and **Foreign Keys (FK)**.

RDBMS supports **SQL (Structured Query Language)**, which is used to interact with the database for tasks such as:

- Inserting new records into tables
- Updating existing records
- Deleting unnecessary or incorrect data

- Retrieving specific information based on queries

Key Features of RDBMS

1. **Data Integrity** - Ensures that the data is accurate and consistent by enforcing rules such as uniqueness constraints and foreign key dependencies
2. **Normalization** - Helps reduce data redundancy and improves efficiency by dividing a database into smaller, well-structured tables
3. **Scalability** - RDBMS systems can handle large amounts of data efficiently, making them suitable for enterprise-level applications
4. **ACID Compliance** - Ensures Atomicity, Consistency, Isolation, and Durability for transactions

Examples of RDBMS:

- MySQL
- PostgreSQL
- Oracle Database
- Microsoft SQL Server
- SQLite
- IBM Db2

What is a Database Schema?

A **schema** defines the logical structure of a database, specifying how data is organized and how relationships are established between tables.

A database schema includes:

- **Tables** - Define the storage structure
- **Columns & Data Types** - Specify attributes and their format
- **Relationships between Tables** - Established using keys (Primary and Foreign)
- **Constraints** - Define rules for data integrity

---

2. Primary Key & Foreign Key

What is a Primary Key?

A **Primary Key (PK)** is a unique identifier for each record in a table. It ensures that no two records within a table have the same value for the primary key column.

Key Characteristics:

- **Uniqueness** - No two records can have the same primary key value
- **Not Null** - The primary key field cannot contain NULL values
- **Single Key per Table** - Every table should have only one primary key

What is a Foreign Key?

A **Foreign Key (FK)** is a column in a table that creates a relationship with another table by referring to its Primary Key. Foreign keys help maintain referential integrity.

Key Characteristics:

- Links two tables together by referencing the Primary Key in another table
- Ensures data consistency by preventing actions that would break relationships
- Can have duplicate values because multiple records in one table may refer to a single record in another table

Example: In an Orders table, customer_id acts as a Foreign Key referencing customer_id in the Customers table.

---

3. Database Design Challenge: E-commerce Schema

Design a basic database schema for an online store with five tables:

1. Customers Table

sql

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   last_login TIMESTAMP
);
```

2. Categories Table

sql

```
CREATE TABLE Categories (
   category_id INT PRIMARY KEY AUTO_INCREMENT,
   category_name VARCHAR(100) UNIQUE NOT NULL,
   description TEXT,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

3. Products Table

sql

```
CREATE TABLE Products (
   product_id INT PRIMARY KEY AUTO_INCREMENT,
   product_name VARCHAR(200) UNIQUE NOT NULL,
   price DECIMAL(10,2) NOT NULL,
   stock_quantity INT DEFAULT 0,
   category_id INT,
   FOREIGN KEY (category_id) REFERENCES Categories(category_id)
);
```

4. Orders Table

sql

```
CREATE TABLE Orders (
   order_id INT PRIMARY KEY AUTO_INCREMENT,
   customer_id INT NOT NULL,
   order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   total_amount DECIMAL(10,2) NOT NULL,
   order_status VARCHAR(50) DEFAULT 'Pending',
   FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

5. Order_Items Table

sql

```
CREATE TABLE Order_Items (
    order_item_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    price_at_purchase DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

SQL Fundamentals

SQL Command Categories

1. Data Definition Language (DDL)

Purpose: Define and modify database structure

Common Commands:

- CREATE - Create new database objects
- ALTER - Modify existing database objects
- DROP - Delete database objects

- TRUNCATE - Remove all data from a table

2. Data Manipulation Language (DML)

Purpose: Manipulate data within tables

Common Commands:

- INSERT - Add new records
- UPDATE - Modify existing records
- DELETE - Remove records

3. Data Query Language (DQL)

Purpose: Retrieve data from database

Common Commands:

- SELECT - Query data from tables

4. Data Control Language (DCL)

Purpose: Control access rights and permissions

Common Commands:

- GRANT - Give user permissions
- REVOKE - Remove user permissions

5. Transaction Control Language (TCL)

Purpose: Manage database transactions

Common Commands:

- COMMIT - Save changes permanently
- ROLLBACK - Undo changes
- SAVEPOINT - Create transaction checkpoint

Basic Queries

1. SELECT and FROM Statements

Basic Syntax

sql

```
SELECT column1, column2, ...

FROM database.schema.table;
```

Examples

sql

```
-- Select all columns

SELECT * FROM products;


-- Select specific columns

SELECT product_name, price FROM products;


-- Select with calculation

SELECT product_name, price, (price * 0.9) AS discounted_price

FROM products;
```

2. WHERE Clause

Basic Syntax

sql

```
SELECT columns

FROM table

WHERE condition;
```

Examples

sql

```
-- Text filtering

SELECT * FROM products WHERE category = 'Electronics';


-- Numeric filtering
```

```
SELECT * FROM products WHERE price > 100;

-- Combined conditions
SELECT * FROM products
WHERE category = 'Electronics' AND price > 100;
```

3. ORDER BY Clause

Basic Syntax

sql

```
SELECT columns
FROM table
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

Examples

sql

```
-- Single column sort (ascending by default)
SELECT * FROM products ORDER BY price;

-- Descending order
SELECT * FROM products ORDER BY price DESC;

-- Multiple column sort
SELECT * FROM products
ORDER BY category ASC, price DESC;
```

4. LIMIT Clause

Basic Syntax

sql

```
SELECT columns
```

```
FROM table

LIMIT number;
```

Example

sql

```
-- Get top 10 most expensive products

SELECT * FROM products

ORDER BY price DESC

LIMIT 10;
```

Advanced Filtering

1. NULL Values

Checking for NULL

sql

```
-- Find products without category

SELECT * FROM products WHERE category_id IS NULL;


-- Find products with category

SELECT * FROM products WHERE category_id IS NOT NULL;
```

2. Logical Operators

AND Operator

sql

```
SELECT * FROM products

WHERE price > 100 AND category = 'Electronics';
```

OR Operator

sql

```
SELECT * FROM products
```

```
WHERE category = 'Electronics' OR category = 'Books';
```

NOT Operator

sql

```
SELECT * FROM products

WHERE NOT category = 'Electronics';
```

3. Range and List Operators

BETWEEN Operator

sql

```
SELECT * FROM products

WHERE price BETWEEN 50 AND 200;
```

IN Operator

sql

```
SELECT * FROM products

WHERE category IN ('Electronics', 'Books', 'Clothing');
```

LIKE Operator

sql

```
-- Products starting with 'iPhone'

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


-- Products containing 'phone'

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


-- Products ending with 'Pro'

SELECT * FROM products WHERE product_name LIKE '%Pro';
```

Functions and Aggregations

1. Single Row Functions

String Functions

sql

```
-- Convert to uppercase

SELECT UPPER(product_name) FROM products;


-- Get substring

SELECT SUBSTRING(product_name, 1, 10) FROM products;


-- Get string length

SELECT LENGTH(product_name) FROM products;
```

Numeric Functions

sql

```
-- Round to 2 decimal places

SELECT ROUND(price, 2) FROM products;


-- Absolute value

SELECT ABS(price) FROM products;


-- Ceiling and floor

SELECT CEIL(price), FLOOR(price) FROM products;
```

Date Functions

sql

```
-- Current date and time

SELECT NOW();
```

```
-- Extract year from date

SELECT YEAR(order_date) FROM orders;


-- Format date

SELECT DATE_FORMAT(order_date, '%Y-%m-%d') FROM orders;
```

2. Aggregate Functions

Basic Aggregations

sql

```
-- Count total products

SELECT COUNT(*) FROM products;


-- Sum of all prices

SELECT SUM(price) FROM products;


-- Average price

SELECT AVG(price) FROM products;


-- Minimum and maximum price

SELECT MIN(price), MAX(price) FROM products;
```

3. GROUP BY and HAVING

GROUP BY Syntax

sql

```
SELECT column1, aggregate_function(column2)

FROM table

GROUP BY column1;
```

Examples

sql

```
-- Count products by category
SELECT category, COUNT(*) as product_count
FROM products
GROUP BY category;


-- Average price by category
SELECT category, AVG(price) as avg_price
FROM products
GROUP BY category;
```

HAVING Clause

sql

```
-- Categories with more than 5 products
SELECT category, COUNT(*) as product_count
FROM products
GROUP BY category
HAVING COUNT(*) > 5;
```

Joins

1. Inner Join

Returns only matching records from both tables.

sql

```
SELECT customers.first_name, orders.order_date, orders.total_amount
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id;
```

2. Left Join (Left Outer Join)

Returns all records from the left table and matching records from the right table.

sql

```
SELECT customers.first_name, orders.order_date
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

3. Right Join (Right Outer Join)

Returns all records from the right table and matching records from the left table.

sql

```
SELECT customers.first_name, orders.order_date
FROM customers
RIGHT JOIN orders ON customers.customer_id = orders.customer_id;
```

4. Full Outer Join

Returns all records when there is a match in either table.

sql

```
SELECT customers.first_name, orders.order_date
FROM customers
FULL OUTER JOIN orders ON customers.customer_id = orders.customer_id;
```

5. Cross Join

Returns the Cartesian product of both tables.

sql

```
SELECT products.product_name, categories.category_name
FROM products
CROSS JOIN categories;
```

6. Self Join

Joins a table to itself.

sql

```
SELECT e1.employee_name as Employee, e2.employee_name as Manager
FROM employees e1
INNER JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

Set Operations

1. UNION

Combines results from two queries, removing duplicates.

sql

```
SELECT customer_id FROM customers WHERE city = 'New York'
UNION
SELECT customer_id FROM customers WHERE city = 'Los Angeles';
```

2. UNION ALL

Combines results from two queries, including duplicates.

sql

```
SELECT customer_id FROM customers WHERE city = 'New York'
UNION ALL
SELECT customer_id FROM customers WHERE city = 'Los Angeles';
```

3. INTERSECT

Returns only rows that appear in both queries.

sql

```
SELECT customer_id FROM customers WHERE city = 'New York'
INTERSECT
SELECT customer_id FROM orders WHERE order_date > '2024-01-01';
```

4. EXCEPT (MINUS)

Returns rows from the first query that are not in the second query.

sql

```
SELECT customer_id FROM customers

EXCEPT

SELECT customer_id FROM orders WHERE order_date > '2024-01-01';
```

Subqueries

1. Subquery in WHERE Clause

sql

```
-- Find products with above-average price

SELECT * FROM products

WHERE price > (SELECT AVG(price) FROM products);
```

2. Subquery in FROM Clause

sql

```
-- Use subquery as a table

SELECT category, avg_price

FROM (

  SELECT category, AVG(price) as avg_price

  FROM products

  GROUP BY category

) AS category_averages

WHERE avg_price > 100;
```

3. Correlated Subquery

sql

```
-- Find customers with above-average order total

SELECT * FROM customers c

WHERE (

  SELECT AVG(total_amount)
```

```
    FROM orders o
    WHERE o.customer_id = c.customer_id
) > 100;
```

Practice Exercises

Beginner Level

1. **Basic Selection**

sql

```
-- Select all products with price greater than $50
SELECT * FROM products WHERE price > 50;
```

2. **Sorting and Limiting**

sql

```
-- Get top 5 most expensive products
SELECT * FROM products ORDER BY price DESC LIMIT 5;
```

3. **Text Filtering**

sql

```
-- Find products containing 'phone' in the name
SELECT * FROM products WHERE product_name LIKE '%phone%';
```

Intermediate Level

4. **Aggregation with Grouping**

sql

```
-- Count orders by customer
SELECT customer_id, COUNT(*) as order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 2;
```

5. **Multiple Conditions**

sql

```
-- Products in Electronics or Books category with price between $20-$100
SELECT * FROM products
WHERE category IN ('Electronics', 'Books')
AND price BETWEEN 20 AND 100;
```

Advanced Level

6. **Complex Join**

sql

```
-- Customer details with their order information
SELECT c.first_name, c.email, o.order_date, o.total_amount
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
ORDER BY c.first_name;
```

7. **Subquery with Aggregation**

sql

```
-- Find customers who spent more than the average order amount
SELECT * FROM customers
WHERE customer_id IN (
   SELECT customer_id
   FROM orders
   WHERE total_amount > (SELECT AVG(total_amount) FROM orders)
);
```

Best Practices

1. Query Optimization

- Use appropriate indexes on frequently queried columns
- Avoid SELECT * in production queries
- Use LIMIT to prevent large result sets
- Use appropriate data types

2. Code Style

- Use consistent naming conventions
- Format SQL queries for readability
- Add comments for complex queries
- Use aliases for table names in joins

3. Security

- Use parameterized queries to prevent SQL injection
- Grant minimum necessary permissions
- Regularly update database software
- Monitor database access logs