#UpSkillWithKalpesh

Day 02

Data Science Unlocked

From Zero to Data Hero

SQL & Database for Data Science

Kalpesh Pathade
@DataSimplified

SQL Mastery Notes

I. 🗂 Comprehensive Guide to Databases

1.1 What is a Database?

- A **database** is a structured collection of data stored electronically, designed to efficiently manage and retrieve data.
- Used in almost every domain: **e-commerce, banking, healthcare, education, social media**, etc.

1.2 Why Do We Need Databases?

1. **Efficient Data Storage**: Handle large datasets in an organized manner.
2. **Data Integrity**: Prevent duplication and maintain accuracy.
3. **Scalability**: Accommodate growing data needs.
4. **Concurrency**: Allow multiple users to access data simultaneously.
5. **Data Security**: Protect sensitive information from unauthorized access.

1.3 Types of Databases

1. **Relational Databases (RDBMS)**:
    - Data is stored in **tables** (rows and columns).
    - Relational model: Uses keys to establish relationships.
    - Example: MySQL, PostgreSQL, OracleDB.
2. **NoSQL Databases**:
    - Designed for **unstructured or semi-structured** data.

   - Types: Document-based, key-value, column-family, graph databases.
   - Example: MongoDB, Cassandra.

3. **Hierarchical Databases**:
   - Data is stored in a **tree-like structure** (parent-child relationships).
   - Example: IBM IMS.
4. **Network Databases**:
   - Data is stored as a graph with multiple relationships.
5. **Object-Oriented Databases**:
   - Stores data as objects with attributes and methods.

1.4 Key Components of a Database

1. **Tables**: Organized into rows (records) and columns (fields).
2. **Schema**: Blueprint that defines the structure of tables, columns, relationships, and constraints.
3. **Keys**:
   - **Primary Key**: Uniquely identifies each record.
   - **Foreign Key**: Links two tables.
   - **Composite Key**: Combination of two or more columns to form a unique identifier.
4. **Indexes**: Speed up data retrieval.
5. **Queries**: SQL commands for data manipulation and retrieval.
6. **Views**: Virtual tables created from queries.
7. **Constraints**: Rules to ensure data integrity (e.g., UNIQUE, NOT NULL).

1.5 Database Design Principles

Designing a database requires careful planning to ensure efficiency, scalability, and data integrity.

1.5.1 Normalization

Normalization is the process of organizing data to reduce redundancy and improve data integrity. It involves breaking a database into smaller, related tables.

Normal Forms:

1. **First Normal Form (1NF)**:
   - Ensure that all columns contain **atomic (indivisible)** values.
   - No repeating groups or arrays.
   - Example:
     - Bad: `{Name: John, Phones: [123, 456]}`
     - Good: `{Name: John, Phone: 123}, {Name: John, Phone: 456}`
2. **Second Normal Form (2NF)**:
   - Achieve **1NF** and remove partial dependencies.
   - Partial Dependency: When a non-key attribute depends on part of a composite key.
   - Example:
     - Bad: `{OrderID, ProductID, ProductName}` (ProductName depends only on ProductID, not OrderID).
     - Good: Split into two tables: `Orders` and `Products`.
3. **Third Normal Form (3NF)**:
   - Achieve **2NF** and remove transitive dependencies.
   - Transitive Dependency: When a non-key attribute depends on another non-key attribute.
   - Example:
     - Bad: `{StudentID, DeptID, DeptName}` (DeptName depends on DeptID, not StudentID).
     - Good: Split into `Students` and `Departments`.
4. **Boyce-Codd Normal Form (BCNF)**:

- A stricter version of 3NF, ensuring every determinant is a candidate key.

5. **Fourth Normal Form (4NF)**:
   - Achieve **BCNF** and remove multivalued dependencies.
   - Example:
     - A student can have multiple hobbies and multiple subjects, which should be stored separately.
6. **Fifth Normal Form (5NF)**:
   - Break tables further to eliminate redundancy caused by **join dependencies**.
7. **Denormalization**:
   - Sometimes, databases are denormalized (combine tables) for better performance, especially in read-heavy systems.

1.5.2 Database Relationships

1. **One-to-One (1:1)**:
   - Example: One person has one passport.
2. **One-to-Many (1:N)**:
   - Example: One customer places multiple orders.
3. **Many-to-Many (M:N)**:
   - Example: Students enroll in multiple courses, and courses have multiple students. Requires a **junction table**.

1.5.3 Entity-Relationship (ER) Model

1. **Entity**: Object with data (e.g., `Student`, `Course`).
2. **Attributes**: Properties of an entity (e.g., `Name`, `Age`).
3. **Relationships**: Links between entities (e.g., `Enrolls` relationship between `Students` and `Courses`).
4. **ER Diagram**: A graphical representation of the database structure.

1.5.4 Data Integrity

1. **Entity Integrity**:
   - Each table must have a unique **Primary Key**.
2. **Referential Integrity**:
   - **Foreign Keys** must reference valid data in another table.
3. **Domain Integrity**:
   - Columns must contain valid data types and constraints.

1.5.5 ACID Properties

1. **Atomicity**: Transactions are all-or-nothing.
2. **Consistency**: Data remains consistent before and after a transaction.
3. **Isolation**: Transactions do not interfere with each other.
4. **Durability**: Data is permanently saved after a transaction.

1.6 Popular Database Management Systems

1. **Relational DBMS**:
   - Examples: MySQL, PostgreSQL, OracleDB, SQL Server.
2. **NoSQL DBMS**:
   - Examples: MongoDB (Document-based), Redis (Key-Value), Cassandra (Column-Family).
3. **Cloud Databases**:
   - Examples: Amazon RDS, Google Firestore, Azure SQL Database.

II. SQL Basics

2.1 What is SQL?

- **SQL (Structured Query Language)**: A standardized programming language used to manage and manipulate relational databases.
- Key Features:
  - Query data efficiently.
  - Insert, update, delete, and retrieve data.
  - Create and manage database schemas.
- Pronunciation: "S-Q-L" or "Sequel."

2.2 SQL Syntax Basics

1. **Case-insensitive**: `SELECT`, `select`, and `SeLeCt` are the same.
2. **Statements End with a Semicolon** (`;`): This signals the end of a command.

2.3 SQL Commands Classification (CRUD Operations)

1. **Data Definition Language (DDL)**:
   - Used to define or modify the database structure.
   - Commands:
     - `CREATE`: Create a database or table.
     - `ALTER`: Modify a table structure.
     - `DROP`: Delete a table or database.
     - `TRUNCATE`: Delete all rows in a table without logging individual row deletions.
2. **Data Manipulation Language (DML)**:
   - Used to manipulate data in the database.
   - Commands:
     - `INSERT`: Add new records to a table.
     - `UPDATE`: Modify existing records.

    - `DELETE`: Remove records.
    - `SELECT`: Retrieve data from tables.

3. **Data Control Language (DCL)**:
    - Controls access to data.
    - Commands:
        - `GRANT`: Give user permissions.
        - `REVOKE`: Remove user permissions.
4. **Transaction Control Language (TCL)**:
    - Manages transactions in a database.
    - Commands:
        - `COMMIT`: Save changes permanently.
        - `ROLLBACK`: Undo changes made by a transaction.
        - `SAVEPOINT`: Set a point in a transaction to roll back to.

---

24 Basic SQL Commands

1. **CREATE DATABASE**:

```
CREATE DATABASE my_database;
```

2. **USE DATABASE**:

```
USE my_database;
```

3. **CREATE TABLE**:

```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

4. **INSERT INTO TABLE**:

```
INSERT INTO users (name, email)
VALUES ('John Doe', 'john@example.com');
```

5. **SELECT FROM TABLE**:

```
SELECT * FROM users;
```

6. **UPDATE TABLE**:

```
UPDATE users
SET email = 'john.doe@example.com'
WHERE id = 1;
```

7. **DELETE FROM TABLE**:

```
DELETE FROM users
WHERE id = 1;
```

8. **DROP TABLE**:

```
DROP TABLE users;
```

2.5 SQL Constraints

Constraints enforce rules on the data in a table.

1. **NOT NULL**: Ensures a column cannot have a `NULL` value.

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

3. **PRIMARY KEY**:

   - Combines `NOT NULL` and `UNIQUE`.
   - Example:

     ```
     id INT PRIMARY KEY;
     ```

4. **FOREIGN KEY**: Links two tables by referencing a column in another table.
5. **CHECK**: Ensures a condition is met for all values in a column.
6. **DEFAULT**: Sets a default value for a column.
   - Example:

     ```
     status VARCHAR(20) DEFAULT 'active';
     ```

2.6 Filtering Data with `WHERE`

- Use the `WHERE` clause to filter rows based on conditions.
- Example:

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

2.7 Operators in SQL

1. **Comparison Operators**:
   - `=`: Equal to.
   - `!=`: Not equal to.
   - `<, >, <=, >=`: Comparison operators.
2. **Logical Operators**:
   - `AND`: Combine multiple conditions.
   - `OR`: Satisfy at least one condition.
   - `NOT`: Negate a condition.

3. **LIKE** (Pattern Matching):
    - `%`: Matches zero or more characters.
    - `_`: Matches a single character.
    - Example:

    ```
    SELECT * FROM users WHERE name LIKE 'J%';
    ```

4. **IN**:
    - Match a value in a list.
    - Example:

    ```
    SELECT * FROM users WHERE id IN (1, 2, 3);
    ```

5. **BETWEEN**:
    - Select a range of values.
    - Example:

    ```
    SELECT * FROM users WHERE id BETWEEN 1 AND 10;
    ```

---

28 Sorting Data with `ORDER BY`

- Sort data in ascending (`ASC`) or descending (`DESC`) order.
- Example:

```
SELECT * FROM users
ORDER BY name ASC;
```

---

29 Limiting Data with `LIMIT`

- Retrieve a specific number of rows.
- Example:

```
SELECT * FROM users LIMIT 5;
```

2.10 Aggregation Functions

1. `COUNT`: Count rows.

   ```
   SELECT COUNT(*) FROM users;
   ```

2. `SUM`: Sum numeric values.

   ```
   SELECT SUM(salary) FROM employees;
   ```

3. `AVG`: Calculate average.

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

4. `MAX` / `MIN`: Find maximum or minimum value.

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

2.11 Grouping Data with `GROUP BY`

- Group rows based on column values.
- Example:

  ```
  SELECT department, COUNT(*)
  FROM employees
  GROUP BY department;
  ```

2.12 Joining Tables

Combine data from multiple tables using **joins**:

1. **INNER JOIN**:

  - Returns rows with matching values in both tables.

  ```
  SELECT users.name, orders.order_id
  FROM users
  INNER JOIN orders ON users.id = orders.user_id;
  ```

2. **LEFT JOIN**:

  - Returns all rows from the left table and matching rows from the right table.

  ```
  SELECT users.name, orders.order_id
  FROM users
  LEFT JOIN orders ON users.id = orders.user_id;
  ```

3. **RIGHT JOIN**:

  - Opposite of `LEFT JOIN`.

4. **FULL OUTER JOIN**:

  - Combines `LEFT` and `RIGHT JOIN`.

2.13 Aliases in SQL

- Use aliases to rename tables or columns.
- Example:

```
SELECT u.name AS username, o.order_id AS orderID
FROM users u INNER JOIN orders o ON u.id = o.user_id;
```

2.14 Subqueries

- A query inside another query.
- Example:

```
SELECT name
FROM users
```

```
WHERE id = (SELECT MAX(id) FROM users);
```

2.15 Views

- Virtual tables created using queries.
- Example:

  ```
  CREATE VIEW active_users AS
  SELECT * FROM users WHERE status = 'active';
  ```

III. SQL Advanced Topics

3.1 Advanced Joins

1. **SELF JOIN**:
   - A table is joined with itself.
   - Example:

     ```
     SELECT e1.name AS Employee, e2.name AS Manager
     FROM employees e1
     INNER JOIN employees e2
     ON e1.manager_id = e2.id;
     ```

2. **CROSS JOIN**:
   - Returns the Cartesian product of two tables.
   - Example:

     ```
     SELECT * FROM products CROSS JOIN categories;
     ```

3. **NATURAL JOIN**:
   - Automatically matches columns with the same name and compatible data types.
   - Example:

```
SELECT * FROM orders NATURAL JOIN customers;
```

3.2 Window Functions

- Perform calculations across rows that are related to the current row.

1. **ROW_NUMBER()**:
   - Assigns a unique number to each row.
   - Example:

```
SELECT ROW_NUMBER() OVER (PARTITION BY department ORDER
BY salary DESC) AS rank, name, salary
FROM employees;
```

2. **RANK()**:
   - Assigns a rank to each row, with gaps if there are ties.
   - Example:

```
SELECT RANK() OVER (ORDER BY salary DESC) AS rank, nam
e, salary
FROM employees;
```

3. **DENSE_RANK()**:
   - Similar to `RANK()`, but without gaps.
   - Example:

```
SELECT DENSE_RANK() OVER (ORDER BY salary DESC) AS ran
k, name, salary
FROM employees;
```

4. **NTILE()**:
    - Divides rows into a specified number of groups.
    - Example:

    ```
    SELECT NTILE(4) OVER (ORDER BY salary DESC) AS quartil
    e, name, salary
    FROM employees;
    ```

5. **LAG() and LEAD()**:
    - Access data from previous or next rows.
    - Example:

    ```
    SELECT name, salary, LAG(salary) OVER (ORDER BY salary)
    AS previous_salary
    FROM employees;
    ```

3.3 Common Table Expressions (CTEs)

- Temporary result sets that simplify complex queries.

1. **Simple CTE**:

```
WITH EmployeeCTE AS (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
)
SELECT * FROM EmployeeCTE;
```

2. **Recursive CTE**:

   - Used for hierarchical data (e.g., org charts).

```
WITH RECURSIVE OrgChart AS (
    SELECT id, name, manager_id
    FROM employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.id, e.name, e.manager_id
    FROM employees e
    INNER JOIN OrgChart o ON e.manager_id = o.id
)
SELECT * FROM OrgChart;
```

3.4 Advanced Subqueries

1. **Correlated Subquery**:

   - Subquery depends on the outer query.
   - Example:

```
SELECT name, salary
FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WH
ERE e1.department = e2.department);
```

2. **EXISTS**:

   - Checks for the existence of rows in a subquery.
   - Example:

```
SELECT name FROM customers
WHERE EXISTS (SELECT 1 FROM orders WHERE customers.id =
orders.customer_id);
```

3.5 Indexing

1. **What is an Index?**
   - Speeds up data retrieval by creating pointers to data in a table.
   - Types:
     - **Single-column Index**: Based on one column.
     - **Composite Index**: Based on multiple columns.
2. **Creating Indexes**:

```sql
CREATE INDEX idx_name ON employees(name);
```

3. **Removing Indexes**:

```sql
DROP INDEX idx_name;
```

3.6 Transactions

- A transaction is a sequence of SQL operations performed as a single unit.

1. **Properties of Transactions (ACID)**:
   - **Atomicity**: All or nothing.
   - **Consistency**: Maintains database integrity.
   - **Isolation**: Transactions do not interfere.
   - **Durability**: Changes are permanent.
2. **Commands**:
   - `START TRANSACTION`: Begin a transaction.
   - `COMMIT`: Save changes permanently.
   - `ROLLBACK`: Undo changes.
   - Example:

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

3.7 Stored Procedures

1. **What are Stored Procedures?**
   - Reusable SQL code stored in the database.
   - Benefits: Faster execution, reduced network traffic, better security.
2. **Creating a Stored Procedure**:

```
DELIMITER //
CREATE PROCEDURE GetEmployeesByDept(dept_id INT)
BEGIN
    SELECT * FROM employees WHERE department_id = dept_id;
END //
DELIMITER ;
```

3. **Executing a Stored Procedure**:

```
CALL GetEmployeesByDept(101);
```

3.8 Triggers

1. **What are Triggers?**
   - Automatically executed actions in response to certain database events (INSERT, UPDATE, DELETE).
2. **Creating a Trigger**:

```
CREATE TRIGGER BeforeInsertEmployee
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    SET NEW.created_at = NOW();
END;
```

3.9 Partitioning

1. **What is Partitioning?**
   - Splitting large tables into smaller pieces for improved performance.
2. **Types of Partitioning**:
   - **Range Partitioning**: Based on ranges of values.
   - **Hash Partitioning**: Based on a hash function.
   - **List Partitioning**: Based on a predefined list of values.
3. **Example**:

```
CREATE TABLE sales (
    id INT,
    amount DECIMAL,
    sale_date DATE
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p1 VALUES LESS THAN (2020),
    PARTITION p2 VALUES LESS THAN (2025)
);
```

3.10 Advanced Query Optimization

1. **EXPLAIN**: Analyze query performance.

```
EXPLAIN SELECT * FROM employees WHERE department_id = 101;
```

2. **Query Optimization Tips**:
   - Use indexes on frequently queried columns.
   - Avoid using `SELECT *`; specify the columns.
   - Use joins instead of subqueries when possible. • Limit the number of rows returned.