Database Design and Development

What is a Database?

Definition of a Database

- **Definition**: A database is an organized collection of structured data, typically stored and accessed electronically. Databases allow data to be efficiently stored, retrieved, and managed.
- **Purpose**: Databases are used to store large amounts of information systematically so users can easily access, update, and manage data.

Purpose of Databases

Main Purpose: Efficient storage and retrieval of data

Additional Purposes:

- Data sharing across users or systems
- Maintaining data consistency and integrity
- Supporting complex queries and data analysis

Difference between Data, Information, and Knowledge

Data: Raw facts and figures (e.g., "50", "John Doe")

Information: Processed data that is meaningful (e.g., "John Doe is 50 years old")

Knowledge: The understanding derived from information (e.g., "Middle-aged individuals are more likely to...")
Example:

- Data: "50", "John Doe", "New York"
- Information: "John Doe is 50 years old and lives in New York"
- Knowledge: "People in New York aged 50 are more likely to shop online during the weekend"

Examples of Real-World Databases

Banking: Storing customer accounts, transactions, loans, and credit card information.

- Example: A bank's database holds millions of customer records with account balances.

E-Commerce: Storing product information, customer details, and transaction history.

- Example: Amazon uses a database to store customer purchase history, which helps in recommendations.

Healthcare: Managing patient records, treatment plans, and medical history.

- Example: A hospital database tracks patient visits, prescriptions, and diagnoses.

Types of Databases

Relational Databases

- **Relational Database (RDBMS)**: A database that organizes data into tables (rows and columns) with relationships between them.
- **Key Concepts**:
  - Tables represent entities (e.g., Customers, Products)
  - Relationships are maintained using keys (Primary and Foreign Keys)

Example:

- A **Customer** table linked to an **Orders** table. Each customer has one or more orders.

Non-Relational Databases (NoSQL)

- **Non-Relational Databases (NoSQL)**: Databases that do not use the traditional table structure. They can store unstructured or semi-structured data.
- **Key Types**:
  - **Document-based** (e.g., MongoDB)
  - **Key-Value** (e.g., Redis)
  - **Column-Family** (e.g., Cassandra)
  - **Graph-based** (e.g., Neo4j)

Example:

- In **MongoDB**, data is stored as **JSON-like documents** instead of rows in tables. This is useful for applications like storing user profiles in a flexible format.

Relational vs. Non-Relational Databases

Relational Databases:

- Structured data with fixed schema
- Strong consistency and ACID properties
- Good for transactional applications (e.g., banking, ERP systems)

Non-Relational Databases:

- Flexible schema, can store semi-structured data
- BASE properties (Basically Available, Soft state, Eventual consistency)
- Good for large-scale, unstructured data (e.g., social networks, big data)

Use Cases for SQL vs. NoSQL

SQL Use Cases:

- Applications that require complex querying and transactions
- Examples: Banking systems, Enterprise Resource Planning (ERP)

NoSQL Use Cases:

- Applications with massive amounts of unstructured data
- Examples: Social media platforms, Real-time analytics systems, IoT applications

Installing and Running Azure MSSQL on Mac Using Docker

Install Docker on Mac

1. **Download Docker**:
    - Visit the official Docker website: https://www.docker.com/products/docker-desktop/.
    - Download and install **Docker Desktop** for Mac by following the on-screen instructions.

Pull the SQL Server Docker Image

Open your **Terminal** on Mac.

Run the following command to pull the official Microsoft SQL Server Docker image:

```
docker pull mcr.microsoft.com/mssql/server:2022-latest
```

This will download the latest version of Microsoft SQL Server for Linux, which can be run using Docker.

Run SQL Server Container

Once the image is pulled, run the following command to start the SQL Server container:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=YourStrong!Passw0rd' \

-p 1433:1433 --name sqlserver -d mcr.microsoft.com/mssql/server:2022-latest
```

Replace `YourStrong!Passw0rd` with a secure password of your choice. This command runs SQL Server on port 1433 (default SQL Server port). Verify the container is running with the following command:

```
docker ps
```

Connecting to SQL Server

Install **Azure Data Studio** on your Mac:

- Download it from the official website: https://aka.ms/azuredatastudio.
- Follow the instructions to install it.

Launch **Azure Data Studio** and connect to the SQL Server:

- Server: `localhost`
- Authentication type: `SQL Login`
- Username: `sa`
- Password: The password you provided in the Docker run command.

Create a Database in SQL Server

Once connected, open a new query window.

Create a database by running the following SQL command:

```
CREATE DATABASE SchoolDB;

GO

USE SchoolDB;

CREATE TABLE Students (

    StudentID INT PRIMARY KEY,

    FirstName NVARCHAR(50),

    LastName NVARCHAR(50),

    EnrollmentDate DATE

);

GO
```

This creates a **SchoolDB** database and a **Students** table inside it.To insert data into the `Students` table you just created, you can use the following `INSERT INTO` SQL statements,

SQL Code to Insert Data into the `Students` Table:

```
USE SchoolDB;



-- Inserting sample student records

INSERT INTO Students (StudentID, FirstName, LastName, EnrollmentDate)

VALUES

        (1, 'John', 'Doe', '2023-09-01'),

        (2, 'Jane', 'Smith', '2023-09-02'),

        (3, 'Michael', 'Johnson', '2023-09-03'),

        (4, 'Emily', 'Davis', '2023-09-04'),

        (5, 'David', 'Brown', '2023-09-05');
```

INSERT INTO Students: This command specifies that we're inserting data into the `Students` table.

StudentID: Integer values representing unique student IDs.

FirstName, LastName: These are `NVARCHAR(50)` fields for storing first and last names, respectively.

EnrollmentDate: The date the student enrolled.

Update (Edit) a Record in the Table

To update a student's record, use the `UPDATE` statement. For example, to change the last name of the student with `StudentID = 1` from "Doe" to "Miller"

```
USE SchoolDB;



-- Update LastName for the student with StudentID = 1

UPDATE Students

SET LastName = 'Miller'

WHERE StudentID = 1;
```

Delete a Record from the Table

To delete a student's record, use the `DELETE` statement. For example, to delete the student with `StudentID = 3` from the table:

```
USE SchoolDB;



-- Delete the student with StudentID = 3

DELETE FROM Students

WHERE StudentID = 3;
```

Blind SQL Injection

What is Blind SQL Injection?

- Blind SQL Injection occurs when the database does not display error messages, making it more difficult to exploit, but attackers can still infer data based on the application's response (true/false).

Performing a Blind SQL Injection

In a login form, an attacker may use conditional statements to retrieve data bit by bit.

1. In the username field, you can inject:

```
' OR 1=1 AND (SELECT SUBSTRING(database(), 1, 1)) = 'S' --
```

This checks if the first character of the database name is "S".

If the application behaves normally, it implies "S" is the first character.

Repeat this by incrementing the substring index until you get the full name of the database:

```
' OR 1=1 AND (SELECT SUBSTRING(database(), 2, 1)) = 'c' --
```

Securing Against Blind SQL Injection

- **Prepared Statements**: Use parameterized queries to prevent SQL injection, even when error messages are hidden.
- **Error Handling**: Ensure that applications handle errors gracefully and never expose sensitive information.

Union-Based SQL Injection

What is Union-Based SQL Injection?

- Union-based SQL Injection allows an attacker to retrieve additional data by appending the results of one query to another using the `UNION` keyword.

Performing a Union-Based SQL Injection

- Suppose the application has a URL parameter like this:

http://example.com/products.php?id=1

An attacker can modify this to:

http://example.com/products.php?id=1 UNION SELECT null, username, password FROM users --

Securing Against Union-Based SQL Injection

- **Input Validation**: Validate and sanitize all user inputs, especially in URL parameters.
- **Web Application Firewalls**: Use a WAF to detect and block SQL injection attempts.

Error-Based SQL Injection

What is Error-Based SQL Injection?

- Error-based SQL Injection takes advantage of detailed error messages returned by the database to extract information.

Step 1: Performing an Error-Based SQL Injection

- Injecting incorrect SQL into a field may cause the database to return error messages that reveal the structure of the database.
- For example, injecting:

```
' OR 1=1 –
```

might return an error like:

```
Error: Unknown column 'username' in 'field list'
```

Attackers can use this information to build more sophisticated SQL injection attacks.

Securing Against Error-Based SQL Injection

- **Disable Error Reporting in Production**: Ensure that detailed error messages are not exposed to users in production environments.
- **Use Error Handlers**: Implement error handlers that catch exceptions and return user-friendly error messages without revealing sensitive information.

Boolean-Based SQL Injection

What is Boolean-Based SQL Injection?

- Boolean-based SQL Injection relies on evaluating true/false statements by manipulating SQL queries. The application behaves differently depending on whether the condition is true or false.

Step 1: Performing a Boolean-Based SQL Injection

- An attacker may inject a SQL statement that tests a condition, like:

' OR 1=1 –

- If the condition is true, the page may load normally.
- If the condition is false, the page may behave differently (showing an error, blank page, etc.).

Attackers can repeat this process with more complex conditions to infer information.

Brute Force Attack on the Database Login.

Exploiting Misconfigured Docker Settings (exposed ports or volumes).

Privilege Escalation via Docker.

Man-in-the-Middle (MITM) Attack on Unencrypted SQL Server Connections.