**Deadlock in JS:**

* **Definition (JS Context)**: A deadlock is an issue in JavaScript that occurs when two asynchronous operations wait for each other, causing the entire operation to halt.
  + Happens with **shared data/resources** in **async/await** programs.
* **General Definition**: A deadlock in programming occurs when two or more processes are unable to proceed because they are waiting for each other to release a shared resource.
* **Steps to Avoid Deadlock**:
  + Avoid **circular dependencies**.
  + Set **timeouts** for each operation to ensure they execute independently.

**Git Operations:**

* **Merge**: Used to combine code from one branch into another with a **new commit**.
* **Rebase (Reapply)**: Similar to merge but **no commit history** is preserved as it treats the changes as if they started from the latest commit.
* **Cherry-pick**: Applies a **specific commit** from one branch to another without transferring the entire history of commits.

**Database Keys:**

* **PK (Primary Key)**:
  + Unique identifier for each row in a table.
  + **Only one** PK per table.
  + Cannot be **null**.
* **UK (Unique Key)**:
  + Ensures that a column's value is **unique** across rows in a table.
  + Can have **null** values.
  + A table can have **multiple unique keys**.

**Promises in JavaScript:**

* **Definition**: A **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation.
* **Capabilities**:
  + **Resolve**: Marks the promise as completed successfully.
  + **Reject**: Marks the promise as failed.
* **Properties**:
  + promiseState: Indicates whether the promise is **pending**, **fulfilled**, or **rejected**.
  + promiseResult: Stores the value or error from the promise.
  + promiseFulfillReactions: Handlers for when the promise is **fulfilled**.
  + promiseRejectReactions: Handlers for when the promise is **rejected**.
  + promiseIsHandled: Indicates if the promise is **handled** or not.
* **Handlers**:
  + **Resolve**: The handler for successful completion.
  + **Reject**: The handler for errors or failure.
* **Methods**:
  + .then(): Used to handle **fulfilled** promises.
  + .catch(): Used to handle **rejected** promises.

**Database vs File System:**

|  |  |  |
| --- | --- | --- |
| \*\*Aspect\*\* | \*\*File System (FS)\*\* | \*\*Database (DB)\*\* |
|  |  |  |
| \*\*Definition\*\* | Stores data in files and folders on disk. | Stores data in a structured format like tables. |
|  |  |  |
| \*\*Data Structure\*\* | Unstructured or simple file format (e.g., text, images). | Structured data with rows and columns (e.g., tables). |
|  |  |  |
| \*\*Data Access\*\* | Accessed through file operations (open, read, write). | Accessed through queries (e.g., SQL queries). |
|  |  |  |
| \*\*Concurrency\*\* | Limited concurrency, requires file locks for simultaneous access. | Supports multiple users simultaneously through transactions. |
|  |  |  |
| \*\*Data Integrity\*\* | No built-in data consistency mechanisms. | ACID properties ensure data integrity. |
|  |  |  |
| \*\*Example\*\* | Storing customer data as `customer1.txt` and `invoice1.jpg` in a folder. | Storing customer data in a `customers` table with structured columns. |
|  |  |  |

* **DB (Database)**:
  + **Definition**: An organized collection of data stored in a **tabular structure** with rows and columns.
  + **Managed by**: **DBMS** (Database Management System).
  + **Querying**: Data can be queried using **SQL** or other query languages.
  + **Concurrency**: Supports **multiple clients** accessing and modifying data simultaneously.
* **FS (File System)**:
  + **Definition**: A storage system that organizes and manages **files** on a storage device.
  + **Data**: Can store both **structured** and **unstructured** data in files.
  + **Querying**: Limited querying (mainly through file paths and names).
  + **Concurrency**: Requires **locking systems** for handling simultaneous operations on the same file.
  + **Presence**: Integrated within the **OS** (Operating System).

**Connection Pool in DB:**

* **Definition**: A **connection pool** is a collection of reusable database connections managed by a pool manager. It reduces the overhead of frequently opening and closing database connections, improving performance and resource management.

**How Connection Pool Works:**

1. **Initialization**: The pool starts with a predefined number of connections.
2. **Connection Request**: The application requests a connection from the pool.
3. **Use of Connection**: The application uses the connection to perform database operations (queries, updates).
4. **Release Connection**: Once the operation is complete, the connection is returned to the pool for reuse.

**Benefits:**

* **Reduced Overhead**: Avoids time-consuming connection setup and teardown.
* **Improved Performance**: Faster response time due to reuse of existing connections.
* **Concurrency**: Efficiently manages multiple users or processes concurrently.

**Connection Pool Parameters:**

* **Max Connections** (max): The maximum number of connections allowed in the pool at any given time.
* **Min Connections** (min): The minimum number of idle connections the pool should maintain.
* **Timeout** (timeout): The maximum time a request can wait to obtain a connection from the pool.
* **Idle Timeout** (idleTimeout): The time a connection can stay idle before it is closed or recycled.

**Comparison of Connection Pool Parameters across DBMS:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DBMS | Max Connections | Min Connections | Timeout | Idle Timeout |  |
|  |  |  |  |  |  |
| SQL Server | Default: 100 | Min: 0-1 | 15-30 seconds | Idle timeout: 10 minutes |  |
| MySQL | Default: 10, Prod: 100+ | Min: 0 | Connect timeout: 10 seconds | Idle timeout: 10 minutes |  |
| PostgreSQL | Default: 100 | Min: 1 | Connection timeout: 30 seconds | Idle timeout: 5 minutes |  |
| MongoDB | Default: 100 | Min: 0 | Connection timeout: 30 seconds | Idle timeout: 10 minutes |  |

**Mongo vs sql**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **MongoDB (NoSQL)** | **MySQL (SQL)** |
| **Database Type** | NoSQL (Document-Oriented) | Relational (RDBMS) |
| **Data Model** | Document-based (BSON format) | Table-based (rows and columns) |
| **Schema** | Schema-less (Flexible structure) | Fixed schema (defined by columns and data types) |
| **Query Language** | MongoDB Query Language (similar to JavaScript) | SQL (Structured Query Language) |
| **Transactions** | Supports ACID for single documents, multi-doc transactions (since v4.0) | Fully supports ACID transactions |
| **Scalability** | Horizontal scaling (Sharding) | Vertical scaling (increasing server resources) |
| **Indexing** | Supports various types of indexes (e.g., single-field, compound) | Supports various types of indexes (e.g., B-tree) |
| **Use Cases** | Big data, real-time applications, unstructured data | Transactional applications, structured data |
| **Data Integrity** | Supports replication for high availability | Ensures data integrity with relational constraints |
| **Joins** | Does not natively support joins, but supports aggregation and embedding documents | Supports JOINs (inner, outer, etc.) for relationships between tables |
| **Data Storage** | Stores data in documents (JSON-like format) | Stores data in tables (rows and columns) |
| **Scaling Method** | Horizontal scaling via Sharding | Horizontal scaling with replication, vertical scaling with server upgrades |
| **Example DBMS** | MongoDB | MySQL |

**Indexes:**

An **index** is a database object that **improves the speed of data retrieval** operations on a database table at the cost of additional space and slower writes (inserts, updates, deletes). Indexes are used to optimize query performance by allowing the database engine to find rows more quickly.

**Types of Indexes**:

* **Primary Index**: The default index created on the **primary key** of a table.
* **Unique Index**: Ensures that no two rows have the same value for a column.
* **Composite Index**: An index on multiple columns.
* **Full-text Index**: Optimized for text searching, used in full-text search queries.

**Triggers:**

A **trigger** is a set of instructions that are **automatically executed** (or "triggered") in response to a specific event on a particular table or view in a database. Triggers can be used to enforce business rules, audit changes, or automatically update related tables.

**Types of Triggers**:

* **Before Triggers**: Fired before the operation (INSERT, UPDATE, DELETE) is applied to the table.
* **After Triggers**: Fired after the operation (INSERT, UPDATE, DELETE) is completed on the table.
* **Instead of Triggers**: Used to replace the operation with custom logic, like overriding an update operation.

**Comparison of Indexes and Triggers in SQL and NoSQL:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Feature** | **SQL Databases (e.g., MySQL, PostgreSQL)** | **NoSQL Databases (e.g., MongoDB, Cassandra)** | | --- | --- | --- | | **Indexes** | Fully supported with various types (B-tree, hash, etc.) | Supported (MongoDB uses B-tree, hash, and text indexes) | | **Triggers** | Supported natively for automatic actions on INSERT, UPDATE, DELETE | Not natively supported (MongoDB uses Change Streams as an alternative) | | **Use Cases for Indexes** | Speed up SELECT queries, joins, and filtering | Speed up queries, especially in document-based or key-value stores | | **Use Cases for Triggers** | Automate actions such as logging, cascading updates, enforcing rules | MongoDB: Use Change Streams to monitor changes; NoSQL often uses application logic for triggers | |

**What is a Tree in Computer Science?**

In computer science, a **tree** is a hierarchical data structure consisting of nodes connected by edges. It is a non-linear data structure, where each node has a value and can have a set of child nodes, which are the nodes that stem from the current node.

* **Root Node**: The topmost node in the tree from which all nodes descend.
* **Parent Node**: A node that has child nodes.
* **Leaf Nodes**: Nodes that have no children.
* **Child Node**: A node that has a parent node.
* **Subtree**: Any node and its descendants.
* **Height**: The length of the longest path from the node to a leaf node.
* **Depth**: The length of the path from the root node to the node.

**Types of Trees in Computer Science:**

1. **Binary Tree**:
   * Each node has at most **two children** (left and right).
   * Used for efficient searching and sorting (e.g., binary search tree).
2. **B-tree**:
   * A balanced **multi-way search tree** that maintains sorted data and allows efficient searching, insertion, and deletion in logarithmic time.
   * **Use**: Used in databases and file systems for indexing and storing large amounts of data efficiently.
3. **B+ Tree**:
   * A **variant of B-tree** that stores actual data only in the **leaf nodes** and maintains a **linked list** of leaf nodes to speed up range queries.
   * **Use**: Database indexing, especially for range queries.

### **Merkle Tree: Basic Information**

A **Merkle Tree** (also known as a **Hash Tree**) is a **binary tree** in which every leaf node contains a **hash of data** and each non-leaf node is a **hash of its children**. The primary purpose of a Merkle tree is to ensure data integrity and efficient verification in distributed systems, such as blockchain, version control systems (like Git), and file systems.

A **Merkle Tree** is a cryptographic tree structure that helps ensure data integrity, efficient verification, and secure management of data in various systems. It's widely used in **blockchains**, **distributed systems**, and **file storage systems**. By leveraging hashes, Merkle trees allow fast, reliable verification of data with minimal overhead.