ABHIRAMI KANNAN

Date :18/1/25

# DATABASE ASSIGNMENT

## RDBMS:

RDBMS stands for **Relational Database Management System**. It is a type of database management system (DBMS) that stores data in a structured format using tables (also known as relations). In an RDBMS, data is organized into rows and columns, with each table representing a different type of entity (such as customers, orders, products, etc.). Relationships between the tables are established using keys, such as primary keys and foreign keys.

Key features of an RDBMS include:

1. **Tables**: Data is stored in tables, which are organized into rows and columns.
2. **SQL (Structured Query Language)**: RDBMS uses SQL to query, update, and manage the data.
3. **Data Integrity**: It enforces rules to ensure the accuracy and consistency of data, such as primary keys and foreign keys.
4. **ACID Properties**: RDBMS ensures that database transactions are processed reliably and follow the ACID properties (Atomicity, Consistency, Isolation, Durability).
5. **Relationships**: Tables can be related to each other via keys (primary and foreign) to model real-world associations between data entities.

Examples of RDBMS include:

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

These systems are commonly used for applications where structured data and complex queries are essential.

## PHANTOM READS

A **phantom read** happens when a transaction retrieves a set of rows that meet certain criteria, but during the transaction, another transaction adds or deletes rows. As a result, when the first transaction re-runs its query, the set of rows it sees has changed.

For example:

- Transaction A queries all customers with a balance over $1000.
- Transaction B inserts a new customer with a balance over $1000.
- Transaction A then re-runs the same query and sees this new customer, even though it wasn't there when the query ran the first time.

Phantom reads are usually prevented by using higher isolation levels like **Serializable**.

## HOW DOES NO SQL DATABASE WORK

NoSQL databases are designed to handle large amounts of unstructured or semi-structured data. They are different from traditional relational databases because they don't use tables with rows and columns. Instead, they store data in flexible formats like:

- **Key-Value pairs** (like Redis)
- **Documents** (like MongoDB)
- **Columns** (like Cassandra)
- **Graphs** (like Neo4j)

### Key Features:

1. **Flexible Data Model**: You don't need to define a fixed structure beforehand. Data can be stored in different formats for each entry.
2. **Scalable**: NoSQL databases can grow easily by adding more servers.
3. **Faster for Certain Use Cases**: They handle large volumes of data and rapid requests, especially for apps like social media, IoT, and big data.

### Example:

In MongoDB (a document store), data is stored as JSON-like documents, so each document can have different fields and structures. You can easily add new types of data without changing a central schema.

NoSQL databases are great for applications that need to handle lots of data and need flexibility in how that data is stored.

CAPS THEOREM

The **CAP Theorem** (also known as Brewer's Theorem) is a principle that applies to distributed databases. It states that a distributed database system can achieve at most two of the following three goals simultaneously:

1. **Consistency (C)**: Every read operation will return the most recent write. All nodes in the system have the same data at the same time.
2. **Availability (A)**: Every request (read or write) will receive a response, even if some of the database nodes are down. The system is always available for operations.
3. **Partition Tolerance (P)**: The system can continue to operate correctly even if network partitions occur, meaning some parts of the system can't communicate with others.

## The Trade-Off:

The CAP Theorem says that a distributed system can only guarantee two of these three properties at the same time. You can't have all three simultaneously:

- **CA (Consistency + Availability)**: The system ensures data consistency and availability, but it won't work well if a network partition occurs (e.g., if nodes can't communicate with each other).
- **CP (Consistency + Partition Tolerance)**: The system guarantees consistency and partition tolerance, but it may not always be available if a partition occurs (i.e., it may not respond to some requests).
- **AP (Availability + Partition Tolerance)**: The system ensures availability and partition tolerance, but it may not always guarantee that all nodes have the most recent data (consistency).

## Example:

- **Consistency**: If two users try to read the same data at the same time, both will see the same, up-to-date value.
- **Availability**: Even if some nodes are down, the system will still respond to requests (though possibly with outdated or incomplete data).
- **Partition Tolerance**: If parts of the system can’t communicate with each other, the system can still function properly.

## In summary:

The CAP Theorem helps explain why distributed systems need to make trade-offs between consistency, availability, and partition tolerance. You can only pick two of these three to guarantee at the same time, depending on the needs of your application.

ACID PROPERTIES

The **ACID properties** are a set of principles that ensure reliable processing of database transactions. Here's a simple explanation of each:

1. **Atomicity**: A transaction is all or nothing. If one part of the transaction fails, the entire transaction is canceled, and no changes are made to the database.
2. **Consistency**: A transaction brings the database from one valid state to another. It ensures that the database remains in a valid state before and after the transaction.
3. **Isolation**: Transactions run independently of each other. The operations of one transaction are not visible to others until the transaction is complete.
4. **Durability**: Once a transaction is committed, its changes are permanent, even if the system crashes.

## In short:

- **Atomicity**: Everything happens or nothing happens.
- **Consistency**: The database stays valid.
- **Isolation**: Transactions don't interfere with each other.
- **Durability**: Changes are permanent after a commit.

INDEXING

**Indexing** in SQL (and relational databases in general) is a mechanism to improve the speed of data retrieval operations on a database table. It helps databases find and retrieve data more efficiently, especially in large tables. Think of it like an index in a book: instead of reading the entire book to find a topic, you can look it up in the index to quickly find where that topic appears.

CREATE INDEX idx_employee_id ON employees(employee_id);

## ISOLATION = 3 different isolation levels

**1. Read Uncommitted**

- **Description**: Allows reading **uncommitted data** (dirty reads).
- **Issues**: Dirty reads, non-repeatable reads, phantom reads.
- **When to Use**: When performance is more important than data consistency.

**2. Read Committed**

- **Description**: Ensures only **committed data** is read, but data can change between reads (non-repeatable reads).

- **Issues**: Non-repeatable reads, phantom reads.
- **When to Use**: Default for many systems, good balance between consistency and performance.

**3. Repeatable Read**

- **Description**: Prevents **non-repeatable reads** (data can't change once read), but allows phantom reads.
- **Issues**: Phantom reads.
- **When to Use**: For scenarios requiring high consistency with minimal issues.

---

These isolation levels control how transactions interact with each other, balancing **data consistency** and **performance**.

**Advantages of RDBMS (Relational Database Management System):**

1. **Data Integrity**: RDBMS enforces **ACID** (Atomicity, Consistency, Isolation, Durability) properties to ensure data integrity and consistency.
2. **Structured Query Language (SQL)**: SQL allows for easy querying and management of data.
3. **Data Relationships**: Efficient handling of **relationships** between data using **foreign keys** and **joins**.
4. **Normalization**: Reduces data redundancy and ensures efficient storage.

**Disadvantages of RDBMS:**

1. **Scalability Issues**: RDBMS can struggle with **horizontal scaling** (across multiple servers) as the data grows.
2. **Performance**: Large tables and complex queries may degrade performance.
3. **Rigid Schema**: Schema changes can be difficult and costly in large, complex systems.
4. **High Cost**: Enterprise-level RDBMS systems can be expensive to license and maintain.

**Security in RDBMS:**

1. **Authentication**: Users must authenticate before accessing the database.
2. **Authorization**: Users have specific permissions (read, write, execute).
3. **Encryption**: Sensitive data can be encrypted both at rest (stored) and in transit (while being transferred).
4. **Backup and Recovery**: RDBMS systems often include automated backup and restore mechanisms.
5. **Access Control**: Ensures that only authorized users can perform specific actions on the data.

**Horizontal Scaling in RDBMS:**

- **Horizontal scaling** means distributing the load across multiple servers (nodes) rather than upgrading a single server (vertical scaling).
- RDBMS traditionally struggles with horizontal scaling because it relies heavily on **centralized** storage and consistency.

**Sharding in RDBMS:**

**Sharding** is a method of **horizontal scaling** where data is distributed across multiple servers (called **shards**) to improve performance and manage large datasets.

**Two Types of Sharding:**

1. **Horizontal Sharding (Data Sharding)**:
    - **Description**: Data is split across multiple databases or servers based on some criteria (e.g., range of values, hashing).
    - **Example**: Dividing customer data across different shards based on the customer's ID range.
    - **Pros**: Helps scale databases to handle more traffic and larger datasets.
    - **Cons**: Can lead to complex queries and issues with joins across shards.
2. **Vertical Sharding (Database Sharding)**:
    - **Description**: Different tables or types of data are stored on different servers. Each shard holds a subset of the database schema.
    - **Example**: Storing user data in one shard and order data in another.
    - **Pros**: Improves performance for specific data types, reduces load on specific servers.
    - **Cons**: Increased complexity in maintaining multiple databases, and can cause bottlenecks for queries involving multiple types of data.

**Summary:**

- **Advantages** of RDBMS include strong data integrity, SQL support, and structured relationships.
- **Disadvantages** include challenges with scaling, performance issues, and rigid schemas.
- **Security** includes authentication, authorization, encryption, and backup.
- **Horizontal scaling** involves distributing data across multiple servers to handle large loads, and **sharding** is a key technique for achieving this with either horizontal or vertical approaches.

Sure! Here's a simpler and more concise explanation of **Normalization** and **Denormalization** with examples.

**Normalization**

Normalization organizes data to reduce redundancy and improve integrity.

**1NF (First Normal Form):**

- **Rule**: Each column should have atomic (single) values; no repeating groups.

**Example** (before 1NF):

| Order_ID | Customer | Items |
|---|---|---|
| 1 | John | Apple, Banana |

**After 1NF**:

| Order_ID | Customer | Item |
|---|---|---|
| 1 | John | Apple |
| 1 | John | Banana |

**2NF (Second Normal Form):**

- **Rule**: Be in **1NF** and remove partial dependencies (no column should depend on part of a composite key).

**Example** (before 2NF):

| Order_ID | Customer_ID | Customer_Name | Item |
|---|---|---|---|
| 1 | 101 | John | Apple |

**After 2NF**:

- **Orders Table**:

| Order_ID | Customer_ID | Item |
|---|---|---|
| 1 | 101 | Apple |

- **Customers Table**:

| Customer_ID | Customer_Name |
|---|---|
| 101 | John |

**3NF (Third Normal Form):**

- **Rule**: Be in **2NF** and remove transitive dependencies (columns shouldn't depend on other non-key columns).

**Example** (before 3NF):

| Order_ID | Customer_ID | Customer_Name | Customer_Phone | Item |
|---|---|---|---|---|
| 1 | 101 | John | 123-456-7890 | Apple |

**After 3NF**:

- **Orders Table**:

| Order_ID | Customer_ID | Item |
|---|---|---|
| 1 | 101 | Apple |

- **Customers Table**:

| Customer_ID | Customer_Name | Customer_Phone |
|---|---|---|
| 101 | John | 123-456-7890 |

**Denormalization**

Denormalization is the process of adding redundancy to improve **performance** for read-heavy operations, even if it means storing duplicate data.

**Example:**

- **Before Denormalization**: Separate Orders and Customers tables.
- **After Denormalization**: Combine Orders and Customers into one table to avoid joins.

**After Denormalization**:

| Order_ID | Customer_ID | Customer_Name | Item |
|---|---|---|---|
| 1 | 101 | John | Apple |

- This **improves read performance** but adds redundancy (e.g., customer name repeats for every order).

**Summary:**

- **Normalization** reduces data redundancy and improves data integrity.
- **Denormalization** introduces redundancy for better performance in systems that frequently read data.

**MYSQL PRACTICAL**

The commands for **installing Docker**, interacting with Docker containers, and accessing databases within those containers. Here's a step-by-step breakdown:

**Installing Docker on Ubuntu:**

1. **Update package information:**
2. sudo apt update
3. **Install curl** (if it's not already installed):
4. sudo apt install curl

5. **Install Docker**:
    - Download and run the Docker installation script:
    - curl -fsSL https://get.docker.com -o get-docker.sh
    - sudo sh get-docker.sh
6. **Verify Docker Installation**:
    - Check Docker version:
    - docker -v
    - Or, check full version details:
    - docker version
7. **Run Docker without sudo** (optional but recommended):
    - Add your user to the Docker group:
    - sudo usermod -aG docker $USER
    - Log out and log back in for changes to take effect.

**Working with Docker Containers:**

1. **List all Docker containers** (running and stopped):
2. docker ps -a
3. **Run a new container** (example with MySQL):
4. docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=root -d mysql:latest
5. **Access a running container's bash shell**:
6. docker exec -it mysql-container bash
    - This command opens a bash shell inside the mysql-container.
7. **Enter a MySQL shell inside the container**: After entering the container's bash shell:
8. mysql -u root -p
    - Enter the password you set (in this case, root).
9. **Show MySQL Databases**: Once in the MySQL shell:
10. SHOW DATABASES;

**Additional Useful Docker Commands:**

1. **Start a Docker container**:
2. docker start container_name
3. **Stop a Docker container**:
4. docker stop container_name
5. **Remove a Docker container** (if stopped):
6. docker rm container_name
7. **Remove a Docker image**:
8. docker rmi image_name
9. **List Docker images**:
10. docker images

**Exiting the Docker Container Bash:**

Once you're done working in the container, you can exit the bash shell:

exit

**Summary of Key Commands:**

1. Install Docker:
2. sudo apt install curl
3. curl -fsSL https://get.docker.com -o get-docker.sh
4. sudo sh get-docker.sh
5. docker -v
6. Docker container management:
7. docker ps -a
8. docker exec -it container_name bash
9. Access MySQL:
10. mysql -u root -p
11. SHOW DATABASES;

These are the core commands to install Docker, run containers, and access databases inside them, all through the Ubuntu terminal.

```
CREATE TABLE DEPARTMENT (

   DEPT_ID INT PRIMARY KEY,

   DEPT_NAME VARCHAR(50) NOT NULL

);
```

**-- Create the EMPLOYEES table**

```
CREATE TABLE EMPLOYEES (

   EMP_ID INT PRIMARY KEY,

   EMP_NAME VARCHAR(50) NOT NULL,

   SALARY INT,

   DEPT_ID INT,

   CONSTRAINT FK_DEPT FOREIGN KEY (DEPT_ID) REFERENCES
DEPARTMENT(DEPT_ID)

);
```

```
INSERT INTO DEPARTMENT (DEPT_ID, DEPT_NAME) VALUES

(1, 'Human Resources'),
```

(2, 'Finance'),

(3, 'Engineering'),

(4, 'Marketing');

INSERT INTO EMPLOYEES (EMP_ID, EMP_NAME, SALARY, DEPT_ID) VALUES

(101, 'John Doe', 60000, 1),

(102, 'Jane Smith', 75000, 2),

(103, 'Emily Davis', 85000, 3),

(104, 'Michael Brown', 70000, 4),

(105, 'Jessica Wilson', 95000, 3);

TO VIEW ALL CONTENTS IN A TABLE

select * from employees;

select * from department;

**-- Example: Calculate distinct values and null counts for each column in a table**

```
SELECT
    COLUMN_NAME,
    COUNT(DISTINCT COLUMN_NAME) AS DISTINCT_VALUES,
    COUNT(*) - COUNT(COLUMN_NAME) AS NULL_COUNT
FROM
    YOUR_TABLE
GROUP BY
    COLUMN_NAME;
```

```
SELECT
        DEPT_ID,
        COUNT(DISTINCT DEPT_ID) AS DISTINCT_VALUES,
        COUNT(*) - COUNT(DEPT_ID) AS NULL_COUNT
    FROM
        EMPLOYEES
    GROUP BY
        DEPT_ID;
```

**-- Example: Check for orphaned records in a child table (missing corresponding parent records)**

```
SELECT
  CHILD_TABLE.*
FROM
  CHILD_TABLE
LEFT JOIN
  PARENT_TABLE ON CHILD_TABLE.PARENT_ID = PARENT_TABLE.ID
WHERE
  PARENT_TABLE.ID IS NULL;
```

```
SELECT
  EMPLOYEES.*
FROM
  EMPLOYEES
INNER JOIN
  DEPARTMENT ON EMPLOYEES.DEPT_ID = DEPARTMENT.DEPT_ID;
```

RIGHT JOIN EXAMPLE

INCLUDE DATA WHERE THERE ARE NO EMPLOYEES IN A DEPARTMENT

```
INSERT INTO DEPARTMENT (DEPT_ID, DEPT_NAME) VALUES (5, 'Testing');


SELECT
   EMPLOYEES.emp_id,
      DEPARTMENT.dept_id
FROM
   EMPLOYEES
RIGHT JOIN
   DEPARTMENT ON EMPLOYEES.DEPT_ID = DEPARTMENT.DEPT_ID;
```

**LEFT JOIN EXAMPLE**

include data where employees doesn't below to any department

```
INSERT INTO EMPLOYEES (EMP_ID, EMP_NAME, SALARY, DEPT_ID) VALUES
(106, 'Vilas Varghese', 10000, null);


SELECT
   EMPLOYEES.emp_id,
      DEPARTMENT.dept_id
FROM
```

```
    EMPLOYEES
LEFT JOIN
    DEPARTMENT ON EMPLOYEES.DEPT_ID = DEPARTMENT.DEPT_ID;
```

**Data Cleansing Queries:**

---

**-- Example: Normalize phone number formats by removing non-numeric characters**

```
UPDATE
    CUSTOMERS
SET
    PHONE_NUMBER = REGEXP_REPLACE(PHONE_NUMBER, '[^0-9]', '');

ALTER TABLE EMPLOYEES MODIFY SALARY VARCHAR(10);

INSERT INTO EMPLOYEES (EMP_ID, EMP_NAME, SALARY, DEPT_ID) VALUES (106,
'VILAS', '100USD', 3);

select REGEXP_REPLACE(SALARY, '[^0-9]', '') from EMPLOYEES;
```

**Data Quality Metrics Queries:**

---

**-- Example: Calculate completeness percentage for each column in a table**

```
SELECT

    DEPT_ID,

    100.0 * COUNT(*) / (SELECT COUNT(*) FROM EMPLOYEES) AS
COMPLETENESS_PERCENTAGE

FROM

    EMPLOYEES

GROUP BY

    DEPT_ID;
```