Question 1: What is NoSQL database? How is it different from the relational database?

- **What is a NoSQL database?**
    - NoSQL (Not only SQL) databases are a class of non-relational databases that provide a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.
    - They are designed to handle large volumes of rapidly changing data, often unstructured or semi-structured, that relational databases struggle with.
    - NoSQL databases prioritize flexibility, scalability, and performance over strict data consistency and ACID (Atomicity, Consistency, Isolation, Durability) properties, which are hallmarks of traditional relational databases.
- **How is it different from the relational database?**
    - **Data Model:**
        - **NoSQL:** Uses various data models such as key-value, document, column-family, or graph. This allows for flexible schemas and handling diverse data types.
        - **Relational Database:** Uses a tabular data model with rows and columns, where data is organized into predefined schemas with strict relationships between tables.

- **Schema:**
  - **NoSQL:** Typically schema-less or has a flexible schema. This means you can add new fields or change data structures without affecting existing data or requiring downtime.
  - **Relational Database:** Requires a rigid, predefined schema. Any changes to the schema can be complex and require altering table structures.
- **Scalability:**
  - **NoSQL:** Designed for horizontal scalability, meaning it can handle increasing data loads by adding more servers (nodes) to a distributed cluster. This makes it suitable for big data applications.
  - **Relational Database:** Primarily scales vertically (adding more resources like CPU, RAM to a single server). Horizontal scaling is more complex and often limited.
- **ACID Properties vs. BASE:**
  - **NoSQL:** Often follows the BASE (Basically Available, Soft state, Eventually consistent) model. This prioritizes availability and partition tolerance over immediate strong consistency, making it suitable for distributed systems where some data inconsistency might be acceptable for better performance.

        - **Relational Database:** Adheres strictly to ACID properties, ensuring strong consistency and reliability of transactions.
    - **Query Language:**
        - **NoSQL:** Uses various query languages, often specific to the database type (e.g., MongoDB's query language for documents, Cassandra Query Language for column-family). Some may have SQL-like interfaces, but they are not standard SQL.
        - **Relational Database:** Uses SQL (Structured Query Language) as its standard query language for defining, manipulating, and retrieving data.
    - **Examples:**
        - **NoSQL:** MongoDB (document), Cassandra (column-family), Redis (key-value), Neo4j (graph).
        - **Relational Database:** MySQL, PostgreSQL, Oracle, SQL Server.

OR

Question 1: Differentiate between structured, semi-structured, and unstructured data. Give an example of each.

- **Structured Data:**
    - **Definition:** Structured data is highly organized and follows a predefined model. It resides in fixed fields within a record or file,

making it easy to store, access, and process. It typically conforms to a tabular format with rows and columns, with well-defined relationships between different data points.

- **Characteristics:**
  - Strict schema.
  - Easy to search and analyze using traditional query languages like SQL.
  - Data types are predefined.
  - Often stored in relational databases.
- **Example:** A customer relationship management (CRM) system's data, where each customer record has fields like "Customer ID," "First Name," "Last Name," "Email Address," "Phone Number," and "Order History," all neatly organized in tables with specific data types.

- **Semi-structured Data:**
  - **Definition:** Semi-structured data does not conform to a rigid, fixed schema like structured data, but it does contain tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. It has some organizational properties but still offers flexibility.
  - **Characteristics:**
    - Does not obey the tabular structure of relational databases.

        - Contains tags or markers to indicate elements.
        - Flexible schema, allowing for variations in data structure.
        - Often uses self-describing formats.
    - **Example:** JSON (JavaScript Object Notation) or XML (Extensible Markup Language) files. For instance, a JSON file containing product information where some products might have fields like "color" and "size," while others might only have "weight" and "dimensions," but all products consistently have "ProductID" and "ProductName."

- **Unstructured Data:**
    - **Definition:** Unstructured data is data that does not have a predefined format or organization. It does not fit into a traditional row-column database. It typically accounts for a vast majority of the data generated today.
    - **Characteristics:**
        - No identifiable structure.
        - Cannot be stored in a traditional relational database without significant preprocessing.
        - Requires advanced analytics tools (like natural language processing or image recognition) to extract insights.
        - Volume can be massive.

- **Example:** A collection of social media posts (e.g., tweets, Facebook updates) that include text, images, and videos without a consistent format; or an archive of email messages where the content of the email body, attachments, and subject lines vary wildly.

Question 2 (Section A, Q1):

- (a) Fill in the blanks:
  - **MapReduce** can best be described as a programming model used to develop Hadoop-based applications that can process massive amounts of data.
  - MapReduce is implemented **using various** programming language.
  - **TaskTracker** node serves as the slave and is responsible for carrying out the tasks that have been assigned to it by the JobTracker.
  - During start-up, the **NameNode** loads the file system state from the fsimage and the EditLogs file.
  - **Mapper** maps input key pairs to a set of intermediate value pairs.
  - **MySQL** is an example of relational database.
  - **NameNode** acts as a master node in a Hadoop cluster.
  - **Broadcast** join is usually used when one data set is large and the other data set is small.

- (b) State whether the following statements are true or false:
    - "Big data is often difficult to collect, but easy to analyze." **False** (Big data is often difficult to collect AND difficult to analyze due to its volume, velocity, and variety).
    - "Raw data should be processed only one time." **False** (Raw data often needs to be processed multiple times for different analytical purposes, transformations, and refinements).

Question 3 (Section B, Q2): How NameNode failure is handled in Hadoop?

- **How NameNode failure is handled in Hadoop:**
    - In earlier versions of Hadoop (before Hadoop 2.x), NameNode was a single point of failure (SPOF). If the NameNode went down, the entire Hadoop Distributed File System (HDFS) became unavailable, meaning no data could be accessed or written.
    - To address this SPOF issue, Hadoop 2.x and later versions introduced the **High Availability (HA) architecture for NameNode**.
    - **Key components of NameNode HA:**
        - **Active NameNode:** This NameNode is responsible for handling all client operations, managing the filesystem namespace, and regulating access to files.

- **Standby NameNode:** This NameNode is a hot standby of the Active NameNode. It maintains an up-to-date copy of the filesystem namespace.
- **Shared Storage (JournalNode/Quorum Journal Manager - QJM):** Both Active and Standby NameNodes communicate with a set of lightweight processes called JournalNodes. When the Active NameNode performs any namespace modification (e.g., file creation, deletion), it logs a record of this modification to a majority of the JournalNodes. The Standby NameNode continuously reads these logs from the JournalNodes and applies them to its own namespace, ensuring it is always synchronized with the Active NameNode's state.
- **Failover Controller (ZooKeeper and ZKFailoverController - ZKFC):** Each NameNode runs a ZKFC process. These ZKFCs are responsible for monitoring the health of their respective NameNodes and for performing failover if the Active NameNode fails. They use Apache ZooKeeper for coordination and to determine which NameNode is currently active.

- **Failure Handling Process:**
  - If the **Active NameNode fails or becomes unresponsive**, its ZKFC detects the failure.
  - The ZKFC of the **Standby NameNode then initiates a failover process**.

- First, it performs a **fencing process** to ensure that the old Active NameNode is completely isolated and cannot accidentally write to the JournalNodes or cause data corruption (e.g., by killing its processes or revoking its network access).
- Once fenced, the **Standby NameNode takes over the role of the Active NameNode**. It ensures it has processed all transactions from the JournalNodes and then begins accepting client requests.
- This automated failover mechanism ensures that the HDFS remains available even if the primary NameNode fails, providing high availability and fault tolerance.

Question 4 (Section B, Q3): What are the two main phases of a MapReduce operation? Explain with the help of an example.

- **The Two Main Phases of a MapReduce Operation:**
  - A MapReduce operation fundamentally consists of two main phases: the **Map phase** and the **Reduce phase**.
- **Map Phase:**
  - **Purpose:** The Map phase's primary role is to process raw input data and transform it into a set of intermediate key-value pairs. It takes a piece of input data, performs some computation on it, and emits zero, one, or multiple key-value pairs.
  - **Process:**

        - **Input Split:** The input data is first divided into smaller, manageable chunks called input splits. Each split is processed independently by a single Mapper task.
        - **Record Reader:** Within each Mapper, a RecordReader converts the input split into records (key-value pairs) suitable for the Mapper function.
        - **Mapper Function:** The user-defined Mapper function takes each input key-value pair, performs the core logic (e.g., filtering, parsing, transforming), and emits intermediate key-value pairs.
    - **Characteristics:** Mappers run in parallel and independently on different data chunks. They do not communicate with each other.

- **Reduce Phase:**
    - **Purpose:** The Reduce phase's primary role is to aggregate, summarize, or compute a final result from the intermediate key-value pairs generated by the Map phase. It processes all values associated with the same key.
    - **Process:**
        - **Shuffle and Sort:** After the Map phase, the intermediate key-value pairs are shuffled and sorted. All values associated with the same key are grouped together and sent to the same Reducer task.

        - **Reducer Function:** The user-defined Reducer function receives a single key and an iterator of all values associated with that key. It then performs aggregation (e.g., sum, count, average) and emits the final output key-value pairs.
    - **Characteristics:** Reducers also run in parallel, but they operate on grouped data. The number of Reducers is typically configurable.

- **Example: Word Count**

Let's use the classic Word Count example to illustrate the MapReduce phases.

Input Data:

"The quick brown fox jumps over the lazy dog."

"The lazy dog barks loudly."

    - **1. Map Phase:**
        - **Input to Mappers:** The input data is split. Let's say:
            - Mapper 1 receives: "The quick brown fox jumps over the lazy dog."
            - Mapper 2 receives: "The lazy dog barks loudly."
        - **Mapper Function Logic:** Each Mapper will read the text, tokenize it into words, convert words to lowercase, and for

each word, emit a key-value pair where the key is the word and the value is 1 (representing one occurrence).

- Intermediate Output from Mapper 1:

(The, 1), (quick, 1), (brown, 1), (fox, 1), (jumps, 1), (over, 1), (the, 1), (lazy, 1), (dog, 1)

- Intermediate Output from Mapper 2:

(The, 1), (lazy, 1), (dog, 1), (barks, 1), (loudly, 1)

- **2. Shuffle and Sort (between Map and Reduce):**
  - The system collects all intermediate key-value pairs from all Mappers.
  - It then groups all values for the same key.
  - Input to Reducer(s) after Shuffle and Sort:

(barks, [1])

(brown, [1])

(dog, [1, 1])

(fox, [1])

(jumps, [1])

(lazy, [1, 1])

(loudly, [1])

(over, [1])

(quick, [1])

(The, [1, 1])

- **3. Reduce Phase:**
  - **Reducer Function Logic:** Each Reducer receives a key and a list of values. It then iterates through the list of values (which are all '1's in this case) and sums them up to get the total count for that word.
  - Final Output from Reducer(s):

(barks, 1)

(brown, 1)

(dog, 2)

(fox, 1)

(jumps, 1)

(lazy, 2)

(loudly, 1)

(over, 1)

(quick, 1)

(The, 2)

This example clearly shows how the Map phase distributes the work and generates intermediate data, and the Reduce phase aggregates that data to produce the final desired result.

Question 5 (Section B, Q4): How do you copy data from the local file system to HDFS and vice-versa?

- **Copying Data from Local File System to HDFS:**
    - To copy a file or directory from your local machine's file system into HDFS, you use the hadoop fs -put command (or its alias hadoop fs -copyFromLocal).
    - **Command Syntax:**

Bash

hadoop fs -put <local_source_path> <hdfs_destination_path>

or

Bash

hadoop fs -copyFromLocal <local_source_path> <hdfs_destination_path>

- **Explanation:**
    - <local_source_path>: This is the absolute or relative path to the file or directory on your local machine that you want to copy.
    - <hdfs_destination_path>: This is the target path within HDFS where you want to store the file or directory. If the destination directory does not exist, HDFS will create it (if

hdfs_destination_path is a directory path). If it's a file path, the file will be created.

- **Example:**
  - To copy a file named mylocalfile.txt from your current local directory to the /user/hadoop/input directory in HDFS:

Bash

hadoop fs -put mylocalfile.txt /user/hadoop/input/

  - To copy a local directory named mydata (and all its contents) to the root of HDFS:

Bash

hadoop fs -put mydata /

- **Copying Data from HDFS to Local File System:**
  - To copy a file or directory from HDFS to your local machine's file system, you use the hadoop fs -get command (or its alias hadoop fs -copyToLocal).
  - **Command Syntax:**

Bash

hadoop fs -get <hdfs_source_path> <local_destination_path>

or

Bash

hadoop fs -copyToLocal <hdfs_source_path> <local_destination_path>

- **Explanation:**
  - <hdfs_source_path>: This is the path to the file or directory within HDFS that you want to retrieve.
  - <local_destination_path>: This is the target path on your local machine where you want the file or directory to be copied. If local_destination_path is a directory, the file/directory from HDFS will be copied inside it. If it's a file path, the HDFS file will be copied as that new file.
- **Example:**
  - To copy a file named output.txt from the /user/hadoop/output directory in HDFS to your current local directory:

Bash

hadoop fs -get /user/hadoop/output/output.txt .

(The . represents the current local directory)

- To copy an HDFS directory named processed_data (and its contents) to a local directory /home/user/reports:

Bash

hadoop fs -get /processed_data /home/user/reports