Q.1. Apply the performance of a fully distributed Hadoop cluster versus a pseudo-distributed mode

Ans: Fully Distributed Hadoop Cluster:

1. **Definition:** In a fully distributed cluster, Hadoop runs on multiple machines, with dedicated nodes for specific roles (NameNode, DataNodes, ResourceManager, etc.).
2. **Performance:**
    - Optimized for scalability and handling large datasets.
    - True parallelism, as tasks are distributed across multiple nodes.
    - Suitable for real-world big data workloads and high throughput.
3. **Use Case:** Ideal for production environments requiring high fault tolerance and resource utilization.

Pseudo-Distributed Mode:

1. **Definition:** In this mode, Hadoop runs on a single machine, simulating a multi-node cluster by running all services (NameNode, DataNode, etc.) on one machine.
2. **Performance:**
    - Limited by the single machine's hardware resources.
    - Lacks true parallelism; tasks are sequentially managed.
    - Suitable for development, testing, and learning purposes.
3. **Use Case:** Useful for debugging or developing applications on small datasets.

Q. 2. Using the weather dataset, how would you write a MapReduce program to calculate the average temperature per year?

Ans: To calculate the average temperature per year using MapReduce, we divide the task into two phases: **Mapper** and **Reducer**.

---

1. Mapper Phase:

The mapper processes each line of the weather dataset, extracts the year and temperature, and emits key-value pairs.

- **Input:**
  Each line contains weather data, including a timestamp, location, and temperature (e.g., 2024-12-08, New York, 15.0).
- **Operation:**

    - Extract the year from the date field (e.g., 2024).
    - Extract the temperature as a numeric value.
    - Emit the year as the key and the temperature as the value.
- **Output Example:**
    - Input: 2024-12-08, New York, 15.0
    - Mapper Output: (2024, 15.0)

---

2. Shuffle and Sort Phase:

After the mapper phase, Hadoop automatically groups all values (temperatures) by their keys (years).

- **Grouped Output Example:**
    - (2024, [15.0, 18.2, 20.5])
    - (2023, [10.0, 12.3, 11.7])

---

3. Reducer Phase:

The reducer calculates the average temperature for each year.

- **Input:**
  Key-value pairs where the key is the year and the value is a list of temperatures.
- **Operation:**
    - Sum the temperatures for the year.
    - Count the number of temperature entries.
    - Compute the average: Average=Sum of TemperaturesCount\text{Average} = \frac{\text{Sum of Temperatures}}{\text{Count}}Average=CountSum of Temperatures.
    - Emit the year and the average temperature.
- **Output Example:**
    - Input: (2024, [15.0, 18.2, 20.5])
    - Reducer Output: (2024, 17.9)

Q. 3. Illustrate the pros and cons of using HDFS over traditional file systems when working with Big Data.

Ans: **Pros of HDFS:**

1. **Scalability:**

- HDFS is designed to scale horizontally by adding more nodes to the cluster, allowing storage and processing of petabytes of data.
- Traditional file systems struggle with scaling efficiently for large datasets.

2. **Fault Tolerance:**
   - HDFS replicates data across multiple nodes, ensuring data availability even if a node fails.
   - Traditional file systems rely on manual backups and are vulnerable to single-point failures.
3. **Distributed Storage:**
   - HDFS splits large files into blocks and distributes them across multiple nodes for parallel processing.
   - Traditional systems are limited to local or network-attached storage, making large-scale data processing slower.

Cons of HDFS:

1. **Latency for Small Files:**
   - HDFS is inefficient for storing and processing a large number of small files due to high metadata overhead.
   - Traditional file systems handle small files better and with lower latency.
2. **Complex Setup and Maintenance:**
   - Setting up and maintaining an HDFS cluster requires specialized knowledge and resources.
   - Traditional file systems are simpler to set up and manage.
3. **Resource Requirements:**
   - HDFS requires a dedicated cluster with sufficient computational and storage resources.
   - Traditional file systems can operate on standalone machines or simpler setups.

Q.4. Analyze the key differences between the old Hadoop API and the new Hadoop API for writing MapReduce programs.

Ans. **1. Package Structure:**

- **Old API:**
  Located in the org.apache.hadoop.mapred package.
- **New API:**
  Located in the org.apache.hadoop.mapreduce package, emphasizing better organization and a modern structure.

---

2. Interfaces:

- **Old API:**
    - Used Mapper and Reducer as classes, requiring overriding of specific methods (map, reduce).
    - Relied on implementing job configurations via JobConf directly.
- **New API:**
    - Uses Mapper and Reducer as interfaces, promoting flexibility.
    - Introduced the Job class for cleaner job configuration and submission.

---

3. Data Types:

- **Old API:**
  Relied on raw data types like Writable and WritableComparable for keys and values, often requiring verbose code.
- **New API:**
  Retains Writable types but allows the use of more expressive, type-safe generic templates (e.g., <KEYIN, VALUEIN, KEYOUT, VALUEOUT>), improving readability and reducing type mismatches.

---

4. Configuration:

- **Old API:**
    - Configurations were managed through JobConf, which could become cluttered in complex applications.
- **New API:**
    - Introduced the Job class to encapsulate configuration details, leading to more modular and readable code.

---

5. Output Committers:

- **Old API:**
  Required custom logic to handle output committing processes explicitly.
- **New API:**
  Provides built-in support for managing output committers, simplifying the handling of partially completed jobs or failures.

Q.5. Predict the effectiveness of the Job Tracker and Task Tracker in ensuring the completion of large-scale data processing jobs.

Ans. **Effectiveness of job tracker:**

- **Job Scheduling and Monitoring:**
  The Job Tracker efficiently schedules jobs by splitting them into smaller tasks and assigning them to available Task Trackers. It monitors the progress of each task and ensures retries for failed tasks.
- **Fault Tolerance:**
  When a Task Tracker fails, the Job Tracker reschedules the task on another node, ensuring fault tolerance and job completion.
- **Resource Management:**
  It allocates tasks based on the availability of data and computational resources, optimizing job performance.

Effectiveness of task tracker:

- **Task Execution:**
  Task Trackers execute map and reduce tasks and report progress back to the Job Tracker.
- **Localized Processing:**
  Tasks are executed closer to the data, reducing data transfer overhead and improving performance.
- **Failure Handling:**
  If a task fails, the Task Tracker alerts the Job Tracker, which then reschedules the task on another node.

Q.6. Demonstrate the importance of the Google File System in enabling Hadoop's scalability and fault tolerance.

Ans. 1. Scalability: GFS divides files into fixed-size blocks (typically 64MB or larger) and distributes them across multiple machines. This approach is mirrored in HDFS, allowing the system to store petabytes of data efficiently. By enabling storage expansion through the addition of inexpensive commodity hardware, GFS (and thus HDFS) ensures that the system can grow with increasing data demands.

2. Fault Tolerance: GFS stores multiple replicas (typically 3) of each data block on different machines. Similarly, HDFS replicates blocks across nodes to ensure data redundancy. If one replica fails, another can be used without disrupting processing. GFS uses regular heartbeats from chunk servers (storage nodes) to the master to monitor system health. HDFS employs a similar mechanism to ensure that the NameNode (HDFS equivalent of the GFS master) is aware of DataNode statuses.

Q.7. Examine the bottlenecks in a MapReduce job when processing extremely large data and suggest ways to optimize the job.

Ans. **Bottlenecks in MapReduce Jobs**

1. **I/O Overhead:**
    - **Cause:** Frequent disk reads/writes due to intermediate data being stored on disk during the shuffle and sort phase.
    - **Impact:** Slows down the job significantly, as disk operations are much slower than in-memory processing.
2. **Skewed Data Distribution:**
    - **Cause:** Uneven distribution of input data among mappers and reducers.
    - **Impact:** Some nodes end up processing significantly more data, causing delays in task completion.
3. **Network Congestion:**
    - **Cause:** Large volumes of data being shuffled between the map and reduce phases.
    - **Impact:** Network bandwidth becomes a bottleneck, increasing latency.

Optimization Strategies

1. **Reduce I/O Overhead:**
    - Use **in-memory storage** for intermediate data where possible (e.g., using frameworks like Apache Spark).
    - Increase the **block size** in HDFS to reduce the number of splits and disk operations.
2. **Balance Data Distribution:**
    - Use a **combiner function** to aggregate intermediate data locally on the mapper side, reducing the load on reducers.
    - Employ **custom partitioners** to distribute data evenly across reducers.
3. **Minimize Network Traffic:**
    - Compress intermediate data using codecs like Snappy or Gzip to reduce the volume of data transferred during the shuffle phase.
    - Optimize the **shuffle/sort phase** by tuning buffer sizes and spill thresholds.

Q.8. Given a large dataset, how would you analyze the role of the Partitioner in controlling how data is distributed across Reducers in Hadoop?

Ans. **Key Responsibilities of the Partitioner**

1. **Key-to-Reducer Assignment:**
    - The Partitioner decides which Reducer a particular key-value pair should be sent to.
    - By default, Hadoop uses the **HashPartitioner**, which assigns keys to Reducers based on the hash value of the key modulo the number of Reducers.
2. **Load Balancing:**
    - Proper partitioning ensures an even distribution of data across Reducers, preventing some Reducers from being overburdened while others remain underutilized.
3. **Data Grouping:**
    - The Partitioner ensures that all key-value pairs with the same key are sent to the same Reducer, which is essential for operations like counting, summing, or sorting.

Example: Default HashPartitioner

If the dataset contains user transactions and Reducers are tasked with summarizing purchases by user, the Partitioner will calculate:
Reducer Index = (Key.hashCode() & Integer.MAX_VALUE) % Number of Reducers

This ensures a deterministic mapping of keys to Reducers.