## PART – A (Questions)

1. What is MapReduce?

MapReduce is a programming model used for processing and generating large data sets with a parallel, distributed algorithm on a cluster. It is a two-phase process where the first phase is the map function, which takes a set of data and converts it into another set of data, where individual elements are broken down into key-value pairs. The second phase is the reduce function, which takes the output of the map function and combines the data to form a smaller set of tuples.

2. What are some applications of MapReduce?

MapReduce is commonly used in big data processing applications such as data mining, log processing, web indexing, and machine learning. It is used to process and analyze large data sets and to perform complex calculations on distributed systems.

3. How does MapReduce work?

MapReduce works by breaking down a large dataset into smaller chunks, distributing those chunks across multiple machines in a cluster, and processing them in parallel. Each machine runs the map function on its chunk of data, producing intermediate key-value pairs. These intermediate results are then grouped together and sent to the reduce function, which aggregates and summarizes the data to produce the final output.

4. What are some popular MapReduce frameworks?

Apache Hadoop is the most popular open-source MapReduce framework, while other popular MapReduce frameworks include Apache Spark, Apache Flink, and Apache Storm. These frameworks provide developers with tools and libraries to build scalable and fault-tolerant MapReduce applications.

5. What is the typical workflow for a MapReduce job?

The typical workflow for a MapReduce job includes several steps, including input data preparation, map task execution, shuffle and sort of intermediate data, reduce task execution, and final output generation.

6. What is the role of the map task in the MapReduce workflow?

The map task is responsible for processing input data and producing a set of intermediate key-value pairs. Each map task processes a portion of the input data in parallel.

7. What is the role of the reduce task in the MapReduce workflow?

The reduce task is responsible for processing the intermediate data generated by the map tasks and producing the final output. Each reduce task processes a subset of the intermediate data in parallel.

8. What is the shuffle and sort phase in the MapReduce workflow?

The shuffle and sort phase is responsible for transferring the intermediate data generated by the map tasks to the reduce tasks and sorting the data by key. This phase ensures that all the intermediate data with the same key is sent to the same reduce task for processing.

9. What are the main components of MapReduce Job.

Mapping phase: Filters and prepares the input for the next phase that may be Combining or Reducing.
Reduction phase: Takes care of the aggregation and compilation of the final result.

10. How is the final output generated in the MapReduce workflow?

The final output is generated by aggregating the results from the reduce tasks. Each reduce task produces a subset of the final output, which is combined to produce the overall output of the MapReduce job.

11. What is MRUnit?

MRUnit is an open-source testing framework designed to help developers test MapReduce jobs written in Java. It provides a set of utilities and classes for writing unit tests that run in a local mode, simulating a Hadoop cluster.

12. What is the anatomy of a MapReduce job run?

A MapReduce job run typically consists of several phases, including job submission, job initialization, map task execution, shuffle and sort, reduce task execution, and job completion.

13. What happens during the reduce task execution phase of a MapReduce job?

During the reduce task execution phase, each reduce task processes the intermediate data received from the map tasks and produces the final output. The reduce tasks may also perform additional operations, such as filtering or sorting the output.

14. What happens during the job completion phase of a MapReduce job?

During the job completion phase, the final output is written to the specified output path, and the job status is reported to the user. The Hadoop cluster releases any allocated resources, and the job is marked as complete.

15. What is YARN in Hadoop?

YARN (Yet Another Resource Negotiator) is the resource management layer in Hadoop that provides a central platform to manage resources and schedule applications across a Hadoop cluster.

16. What are the components of YARN?

YARN consists of several components, including the Resource Manager, Node Manager, Application Master, and Containers.

17. What is a container in YARN?

A container in YARN is a lightweight, isolated environment that runs a specific task or process within an application. Containers are used to manage resource allocation and scheduling across the Hadoop cluster.

18. How does YARN improve resource utilization in Hadoop?

YARN improves resource utilization in Hadoop by providing a central platform to manage resources and applications across the cluster. It allows multiple applications to run on the same cluster, dynamically allocates resources based on application requirements, and provides fine-grained control over resource utilization.

19. Difference between MapReduce and Spark.

| Key Features | Apache Spark | Hadoop MapReduce |
|---|---|---|
| Speed | 10–100 times faster than MapReduce | Slower |
| Analytics | Supports streaming, Machine Learning, complex analytics, etc. | Comprises simple Map and Reduce tasks |
| Suitable for | Real-time streaming | Batch processing |
| Coding | Lesser lines of code | More lines of code |
| Processing Location | In-memory | Local disk |

20. How does shuffling work in the MapReduce framework?

In the MapReduce framework, shuffling refers to the process of redistributing the intermediate key-value pairs produced by the map tasks to the appropriate reduce task. The intermediate data is sorted by the key, and then partitioned into multiple buckets based on the reduce task that will process the data.

21. Why is shuffling an important step in MapReduce?

Shuffling is an important step in MapReduce because it enables parallelism and load balancing. By redistributing the intermediate data to the appropriate reduce tasks, MapReduce can process the data in parallel on multiple machines, which can significantly speed up processing time. Shuffling also helps to balance the workload across the reduce tasks, ensuring that no single task is overloaded with data.

22. What is the purpose of sorting in the MapReduce framework?
The purpose of sorting in the MapReduce framework is to group together all key-value pairs with the same key, so that they can be processed together by the same reduce task. Sorting also helps to optimize performance by reducing the amount of data that needs to be processed by each reduce task. By grouping together all key-value pairs with the same key, MapReduce can reduce the number of times that each key needs to be processed, which can significantly improve processing time.

23. What are the two main types of functions in MapReduce?
The two main types of functions in MapReduce are the map function and the reduce function.

24. What is the purpose of the map function in MapReduce?
The map function in MapReduce is used to process input data and produce a set of intermediate key-value pairs. The input data is typically split into multiple chunks, and each chunk is processed independently by a map task running on a separate machine in a distributed environment.

25. Define Text Input Format.
TextInputFormat is one of the file formats of Hadoop. It is a default type format of hadoop MapReduce that is if we do not specify any file formats then RecordReader will consider the input file format as textinputformat.

26. What is the purpose of the reduce function in MapReduce?
The reduce function in MapReduce is used to aggregate and process the intermediate key-value pairs produced by the map tasks. The reduce tasks receive a subset of the intermediate data that has been grouped by key, and then they process the data to produce the final output.

27. What are the different input formats that can be used in MapReduce?
There are several input formats that can be used in MapReduce, including text input format, sequence file input format, and Hadoop archive input format. Each input format is designed to handle a specific type of data, and provides a different set of features and functionality.

28. What is the purpose of the input format in MapReduce?
The input format in MapReduce is used to specify how the input data should be read and processed by the map tasks. The input format defines the format of the input data, the location of the input data, and how the input data should be split into chunks for processing by the map tasks.

29. What are the different output formats that can be used in MapReduce?
There are several output formats that can be used in MapReduce, including text output format, sequence file output format, and Hadoop archive output format. Each output format is designed to handle a specific type of data, and provides a different set of features and functionality.

30. What is the purpose of the output format in MapReduce?
The output format in MapReduce is used to specify how the output data should be written and processed by the reduce tasks. The output format defines the format of the output data, the location where the output data should be stored, and how the output data should be partitioned and sorted before it is written to disk.

**PART – B (Questions)**

1. Demonstrate the algorithms using Map Reduce.
2. Analyze the steps of Map reduce Algorithms.
3. Elaborate the map reduce algorithm with an example.
4. What is the role of combiner and partitioner in a map reduce application?
5. List the details of reducer size and replication rate.
6. Explain in detail about HADOOP YARN.
7. Consider a collection of literature survey made by a researcher in the form of a text document with respect to cloud and big data analytics. Analyze Using Hadoop and Map Reduce, write a program to count the occurrence of pre dominant key words.
8. Evaluate a procedure to find the number of occurrence of a word in a document.
9. Illustrate the challenges of using MapReduce for processing big data.
10. Explain the different types of failures in Hadoop and how they are handled automatically.

**Additional Questions:**

1. **A Tweet in a twitter can have hashTags (#helloTwitter) and a certain hashTag used most number of times in tweets globally is said to have highest trend.**

We thought of applying MapReduce algorithm to find the trends in Twitter.

A Tweet in a twitter can have hashTags (#helloTwitter) and a certain hashTag used most number of times in tweets globally is said to have highest trend.

This data is huge and also keeps on increasing, so processing it in traditional manner would not be possible.

Hence we would require hadoop to help us solve this problem.
Twiiter uses Cassandra to store the data in key-value format. Lets assume for simplicity that the key value pair for tweet data looks something like this < twitterHandle,Tweet >.

So, in order to find the top n trends in a given snapshot, we would need to:
1. Process all Tweets and parse out tokens with HashTags.
2. Count all the hashTags.
3. Find out top n hashtags by sorting them.

So, the input data for our Mapper could be a flat file generated out of Values of the Key-Value of <twitterHandle,Tweet>.
It would look something like this :

```
1    I love #agileNCR.
2    Attenindg sesssion on #hadoop.
3    .....
4    ....
```

click here to see the sample input file.
**Assumption:** As one tweet can be of maximum 140 characters we can store the data in such a format that each line is new tweet.
**Step 1: Mapper**
Mapper while tokenising would collect only the tokens which start by #.

```
@Override


public void map(LongWritable key, Text value, Context context) throws
IOException,
```

```
   InterruptedException {

        String line = value.toString();

        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {

         String token = tokenizer.nextToken();

       if(token.startsWith("#")){

          word.set(token.toLowerCase());
          // Context here is like a multi set which allocates value "one" for key
"word".
          context.write(word, one);
          }
    }
```

**Step 2: Combiner**

Combiner would combine all the same hashtags together.

**Step3: Reducer**

```
@Override

    protected void reduce(Text key, Iterable<IntWritable> values, Context
context)

    throws IOException, InterruptedException {

        int sum = 0;

        for (IntWritable val : values) {

            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
```

Reducer will generate the output something like this :

```
1    #agileNCR 21
2    #hello 4
3    #xomato 88
4    #zo 36
```

The problem with this output is that it is sorted by the key values, as in the mapping phase the shuffle and sort step sorts them alphabeticaly on the basis of keys.

To get the desired out of sorting it on the basis of number of occurances of each hashTag, we would need them to be sorted on the basis of values.
So we decided to pass this output to second Map-Reduce job which will swap the key and the value and then perform sort.

Hence :

**Step4: Mapper 2**

Tokenise the input and put 2nd token(the number) as key and 1st token (hashtag) as value.
While mapping it will shuffle and sort on the basis of key.
However, the sorting of the keys by default is in ascending order and we would not get out desired list.

So, we would need to use a Comparator.
We would need to use LongWritable.ReverseComparator.

```
 @Override

public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        String line = value.toString(); // agilencr 4

        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {

                String token = tokenizer.nextToken();

               // Context here is like a multi set which allocates value "one"
for key "word".

                context.write(new
LongWritable(Long.parseLong(tokenizer.nextToken().toString())), new Text(token));


        }
    }
```

**Step5: Reducer 2**
In reducer we will swap back the result again.

```
@Override

       public void map(LongWritable key, Text value, Context context)

        throws IOException, InterruptedException {

        String line = value.toString(); // agilencr 4

        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {


              String token = tokenizer.nextToken();

        // Context here is like a multi set which allocates value "one" for key
"word".

              context.write(new
LongWritable(Long.parseLong(tokenizer.nextToken().toString())),
                new Text(token));

        }
    }
```

So that we get the desired output like this:

```
1    #xomato 88
2    #zo 36
3    #agileNCR 21
4    #hello 4
```

## Explain the different types of failures in Hadoop and how they are handled automatically.

One of the major advantage of using Hadoop is its ability to handle failures and allow jobs to complete successfully. In this article we are going to discuss about the different types of Failures that can occur in Hadoop and how they are handled.

Let us begin with the HDFS failure and then discuss about the YARN failures. In HDFS there are two main daemons, Namenode and Datanode.

**Namenode Failure:**

Namenode is the master node which stores metadata like filename, number of blocks, number of replicas, location of blocks and block IDs.

In Hadoop 1x, Namenode is the single point of failure. Even if the metadata is persisted in the disk, the time taken to recover it is very high (i.e. 30 minutes approximately ). The addition of High Availability in Hadoop 2 solves this problem.

In Hadoop 2 there will be two namenodes with active and passive configuration. At a given point of time only one namenode will be active and if the active namenode goes down then the passive namenode will take the responsibility of serving clients without any interruption.

The active and passive node should always be in sync with each other and must have same metadata(i.e. fsimage and editlogs). The namenodes must use highly available shared storage to share the edit log. When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.

Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk. The passive node takes periodic checkpoints of the active namenode's namespace.

One more important point to know here is the Fencing. Fencing is a process to ensure that only one namenode is active at a time. Namenode Failover fencing can be done using two ways, One way is by using Quorum Journal Manager (QJM) for storing the editlogs, where only the active namenode can write the editlogs to the Journal Node and passive or standby node can only read the editlogs. Other way is using the shared storage where the active node applies the edit logs information and passive node will constantly look for the changes. When a namenode fails it is killed from accessing this shared storage device. This way we ensure that only one namenode is active.

**DataNode Failure:**

In HDFS, to prevent the failure or loss of data we use replication, where a single data block is stored in multiple locations. By default the replication factor for HDFS is 3, which means that along with the original block there will be two replica's.

Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first (off-rack), chosen at random. The third replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes in the cluster, although the system tries to avoid placing too many replicas on the same rack.

Datanode will continuously sends heartbeat signals to namenode for every 3 seconds by default. If Namenode does not receive a heartbeat from datanode for 10 minutes (by default), the Namenode will consider the datanode to be dead. Namenode will now check the data available in that dead datanode and initiates the data replication. So that the replication strategy discussed above will be satisfied. Even if the one datanode goes down, the namenode will provide the address for its replica, so that there wont be any interruption.

Now let us see the failure in YARN, Hadoop handles the failures of Application Master, Resource Manager, Node Manager and Tasks.

**Application Master Failure:**

If an Application Master fails in Hadoop, then all the information related to that particular job execution will be lost. By default the maximum number of attempts to run an application master is 2, so if an application master fails twice it will not be tried again and the job will fail. This can be controlled using the property yarn.resourcemanager.am.max-attempts.

An application master sends periodic heartbeats to the RM, and in the event of application master failure, the RM will detect the failure and starts a new instance of the master running in a new container, it will use the job history to recover the state of the tasks that were already run by the (failed) application so they don't have to be rerun. Recovery is enabled by default, but can be disabled by setting yarn.app.mapreduce.am.job.recovery.enable to false.

The MapReduce client polls the application master for progress reports, but if its application master fails, the client needs to locate the new instance. During job initialization, the client asks the resource manager for the application master's address, and then caches it so it doesn't overload the resource manager with a request every time it needs to poll the application master. If the application master fails, however, the client will experience a timeout when it issues a status update, at which point the client will go back to the resource manager to ask for the new application master's address.

**Node Manager Failure:**

Nodemanager will send a heartbeat signal for every 3 seconds to the resource manager. If the node manager fails due to crash or running very slowly the RM will wait for the heartbeat for 10 minutes. If it is not received, the RM will remove the node from its pool to schedule the containers.

If an AM is running in the failed node manager, then the AM will be launched in another node and this will not be considered as an attempt because it was not the mistake of the AM (it will not be counted in am max attempt counter). The completed tasks of the dead node manager are to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed node manager's local filesystem may not be accessible to the reduce task.

Node managers may be blacklisted if the number of failures for the application is high, even if the node manager itself has not failed. Blacklisting is done by the application master if more than three tasks fail on a node manager.

**Resource Manager:**

Resource Manager is the single point of failure in YARN. To achieve high availability (HA), it is necessary to run a pair of resource managers in an active-standby configuration. If the active resource manager fails, then the standby can take over without a significant interruption to the client.

Information about all the running applications is stored in a highly available state store (backed by ZooKeeper or HDFS), so that the standby can recover the core state of the failed active resource manager. Node manager information is not stored in the state store since it can be reconstructed relatively quickly by the new resource manager as the node managers send their first heartbeats.

The transition of a resource manager from standby to active is handled by a failover controller. The default failover controller is an automatic one, which uses ZooKeeper leader election to ensure that there is only a single active resource manager at one time.

**Task Failure:**

Task failure generally occurs due to run time exceptions or due to sudden exit of task JVM. The task will send a heartbeat signal to the AM for every 3 seconds and if the AM doesn't receive any update for 10 minutes, it will consider the task as failed and will rerun the task attempt.

When the application master is notified of a task attempt that has failed, it will reschedule execution of the task. The application master will try to avoid rescheduling the task on a node manager where it has previously failed. Furthermore, if a task fails four times, it will not be retried again. The job will return the failed status.

Some task attempt may also be killed, which is different from it failing. A task attempt may be killed because it is a speculative duplicate or because the node manager it was running on failed and the application master marked all the task

attempts running on it as killed. Killed task attempts do not count against the number of attempts to run the task, because it wasn’t the task’s fault that an attempt was killed.