| **Date: 28.04.2025** | **Test - 1** | **Max. Marks: 50+10** |
| --- | --- | --- |
| **Semester: VI** | **UG** | **Duration: 1 Hrs + ½ Hr** |
| **Course Title: Big Data Technologies** | | **Course Code:AI362IA** |

**Department of Artificial Intelligence and Machine Leaning**

**SCHEME & SOLUTIONS**

| **S No** | **Questions** | **M** | **BT** | **CO** |
| --- | --- | --- | --- | --- |
|  | What is the role of Name node in HDFS  Manage the metadata of the file system | 2 | 2 | 1 |
|  | How does Hadoop handle read consistency in a distributed environment?  By reading data from the most recently updated replica | 2 | 2 | 1 |
|  | What is the role of the Namenode in a read operation in Hadoop?  To manage file system metadata and locate data blocks | 2 | 2 | 1 |
|  | Define fencing in High Availabilty?  The concept which prevents the old Active Namenode from making changes after a failover | 2 | 2 | 1 |
|  | Define data locality Optimization  Hadoop run the map task on a node where the input data resides in HDFS | 2 | 2 | 1 |

| **PART- B** | | | | |
| --- | --- | --- | --- | --- |
| 1(a) | Summarize the different characteristics of Big Data 1. Volume Volume refers to the amount of data that you have. We measure the volume of our data in Gigabytes, Zettabytes (ZB), and Yottabytes (YB). According to the industry trends, the volume of data will rise substantially in the coming years. 2. Velocity Velocity refers to the speed of data processing. High velocity is crucial for the performance of any big data process. It consists of the rate of change, activity bursts, and the linking of incoming data sets. 3. Value Value refers to the benefits that your organization derives from the data. Does it match your organization’s goals? Does it help your organization enhance itself? It’s among the most important big data core characteristics. 4. Variety Variety refers to the different types of big data. It is among the biggest issues the big data industry faces as it affects performance. It’s vital to properly manage the variety of your data by organizing it. Variety is the various types of data you gather from various sources. 5. Veracity Veracity refers to the accuracy of your data. It is among the most important Big Data characteristics as low veracity can greatly damage the accuracy of your results. 6. Validity How valid and relevant is the data to be used for the intended purpose. 7. Volatility Big data is constantly changing. The data you gathered from a source a day ago might be different from what you found today. This is called variability of data, and it affects your data homogenization. 8. Visualization Visualization refers to showing your big data-generated insights through visual representations such as charts and graphs. It has become prevalent recently as big data professionals regularly share their insights with non-technical audiences. | 5 | 2 | 1 |

| 1(b) | Why are distributed systems essential for modern data storage and analysis?   * **Access speeds** — The rate at which data can be read from drives * Access have not kept up. One typical drive from 1990 could store 1,370 MB of data and had a transfer speed of 4.4 MB/s, so you could read all the data from a full drive in around five minutes. * Over 20 years later, 1-terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk. * Writing to the disk is even more slower compare to the reading * The obvious way to reduce the time is to read from multiple disks at once. * Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in under two minutes * The first problem to solve is hardware failure   + As soon as start using many pieces of hardware, the chance that one will fail is fairly high.   + A common way of avoiding data loss is through replication   + Replication - Redundant copies of the data are kept by the system, so that in the event of failure, there is another copy available.   + This is how RAID works, for instance, although Hadoop’s filesystem, the Hadoop Distributed Filesystem (HDFS), takes a slightly different approaches   + The second problem is that most analysis tasks need to be able to combine the data in some way, and data read from one disk may need to be combined with data from any of the other 99 disks.   + Various distributed systems allow data to be combined from multiple sources, but doing this correctly is notoriously challenging.   + MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values. * The approach taken by MapReduce may seem like a brute-force approach * MapReduce is a batch query processor, and the ability to run an adhoc query against your whole dataset and get the results in a reasonable time is transformative * It changes the way you think about data and unlocks data that was previously archived on tape or disk. | 5 | 2 | 1 |
| --- | --- | --- | --- | --- |
| 2(a) | Define HDFS? Examine the applications for which HDFS does not work?  HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware  ***Very large files***  “Very large” in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data  ***Streaming Data Access***   * + HDFS is built around the idea that the most efficient data processing pattern is a   write-once, read-many-times pattern   * + A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time   + Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record   ***Commodity Hardware***   * + Hadoop doesn’t require expensive, highly reliable hardware   + It’s designed to run on clusters of commodity hardware (commonly available hardware that can be obtained from multiple vendors)for which the chance of node failure across the cluster is high, at least for large clusters.   + HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure   ***Low-Latency Data Access***   * + Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low-latency access   ***Lots of small files***   * + Because the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode   + As a rule of thumb, each file, directory, and block takes about 150 bytes.   + So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory | 2+4 | 2 | 1 |
| 2(b) | Explain the roles of the two types of nodes functioning in the master–worker architecture of HDFS.  The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts. A client accesses the filesystem on behalf of the user by communicating with the name‐node and datanodes. The client presents a filesystem interface similar to a Portable Operating System Interface (POSIX), so the user code does not need to know about the namenode and datanodes to function.  Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing. Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this. The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount. It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged name‐ space image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of  the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode’s metadata files that are on NFS to the secondary and run it as the new primary. (Note that it is possible to run a hot standby namenode instead of a secondary. | 4 | 2 | 2 |
| 3 | HDFS is built write-once, read-many-times pattern. Justify this statement through write operation?  Diagram – 02 Marks  Explanation – 08 Marks    step 1   * The client creates the file by calling create() on DistributedFileSystem   step 2   * DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem’s namespace, with no blocks associated with it * The namenode performs various checks to make sure the file doesn’t already exist and that the client has the right permissions to create the file. * If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. * The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to.   step 3   * The client writes data, the DFSOutputStream splits it into packets, which it writes to an internal queue called the *data queue*. * The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. * The list of datanodes forms a pipeline   step 4   * The DataStreamer streams the packets to the first datanode in the pipeline, * which stores each packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last)d atanode in the pipeline   step 5   * The DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. * A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline * If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.   + First , the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets.   + The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.   + The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes.   + The remainder of the block’s data is written to the good datanodes in the pipeline.   + The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node   + Subsequent blocks are then treated as normal.   step 6   * When the client has finished writing data, it calls close() on the stream   Step 7   * This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete * The namenode already knows which blocks the file is made up of (because Data Streamer asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully | 10 | 2 | 2 |
| 4(a) | How does the Combiner function contribute to optimization in data processing?  Many Map Reduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner function’s output forms the input to the reduce function. Because the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer. The combiner function doesn’t replace the reduce function. But it can help cut down the amount of data shuffled between the mappers and the reducers, and for this reason alone it is always worth considering whether you can use a combiner function in your Map Reduce job. | 5 | 2 | 2 |
| 4(b) | Why is it necessary to have multiple partitions during the reducer phase in data processing  Parallel Processing  Efficient Resource Utilization  Scalability  Fault Tolerance  Better Load Balancing | 5 | 2 | 2 |
| 5 | Write a Map reduce program using Java to find word count for a given text file?  // Importing libraries  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapred.MapReduceBase;  import org.apache.hadoop.mapred.Mapper;  import org.apache.hadoop.mapred.OutputCollector;  import org.apache.hadoop.mapred.Reporter;  public class WCMapper extends MapReduceBase implements Mapper<LongWritable,Text, Text, IntWritable> {  // Map function  public void map(LongWritable key, Text value, OutputCollector<Text,  IntWritable> output, Reporter rep) throws IOException  {  String line = value.toString();  // Splitting the line on spaces  for (String word : line.split(" "))  {  if (word.length() > 0)  {  output.collect(new Text(word), new IntWritable(1));  }  }  }  }  **Reduce function**  // Importing libraries  import java.io.IOException;  import java.util.Iterator;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapred.MapReduceBase;  import org.apache.hadoop.mapred.OutputCollector;  import org.apache.hadoop.mapred.Reducer;  import org.apache.hadoop.mapred.Reporter;  public class WCReducer extends MapReduceBase implements Reducer<Text,  IntWritable, Text, IntWritable> {  // Reduce function  public void reduce(Text key, Iterator<IntWritable> value,  OutputCollector<Text, IntWritable> output, Reporter rep) throws IOException  {  int count = 0;  // Counting the frequency of each words  while (value.hasNext())  {  IntWritable i = value.next();  count += i.get();  }  output.collect(key, new IntWritable(count));  }  }  **Driver Code**  // Importing libraries  import java.io.IOException;  import org.apache.hadoop.conf.Configured;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapred.FileInputFormat;  import org.apache.hadoop.mapred.FileOutputFormat;  import org.apache.hadoop.mapred.JobClient;  import org.apache.hadoop.mapred.JobConf;  import org.apache.hadoop.util.Tool;  import org.apache.hadoop.util.ToolRunner;  public class WCDriver extends Configured implements Tool {  public int run(String args[]) throws IOException  {  if (args.length < 2)  {  System.out.println("Please give valid inputs");  return -1;  }  JobConf conf = new JobConf(WCDriver.class);  FileInputFormat.setInputPaths(conf, new Path(args[0]));  FileOutputFormat.setOutputPath(conf, new Path(args[1]));  conf.setMapperClass(WCMapper.class);  conf.setReducerClass(WCReducer.class);  conf.setMapOutputKeyClass(Text.class);  conf.setMapOutputValueClass(IntWritable.class);  conf.setOutputKeyClass(Text.class);  conf.setOutputValueClass(IntWritable.class);  JobClient.runJob(conf);  return 0;  }  // Main Method  public static void main(String args[]) throws Exception  {  int exitCode = ToolRunner.run(new WCDriver(), args);  System.out.println(exitCode);  }  } | 10 | 3 | 2 |