



Department of Artificial Intelligence and Machine Learning

Course Code:21AI62

Sem: VI

Date:

Duration: 90 Minutes

CIE-I

Big Data Technologies

Answer all Questions

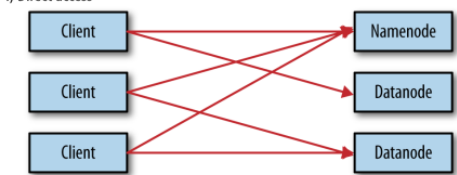
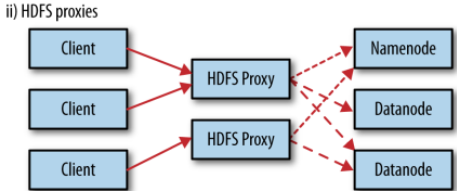
SL. No	Questions	Marks	BT	CO
1	<p>a) How is HDFS structured, and in what situations might HDFS be less suitable?</p> <p>HDFS is a files system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.¹ Let's examine this statement in more detail:</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>Although this may change in the future, these are areas where HDFS is not a good fit today:</p> <p>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.</p> <p>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.</p>	5	1	1
	<p>b) There two types of nodes operating in a master-worker pattern in HDFS. Justify the statement?</p> <p>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.</p> <p>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</p>	5	3	1



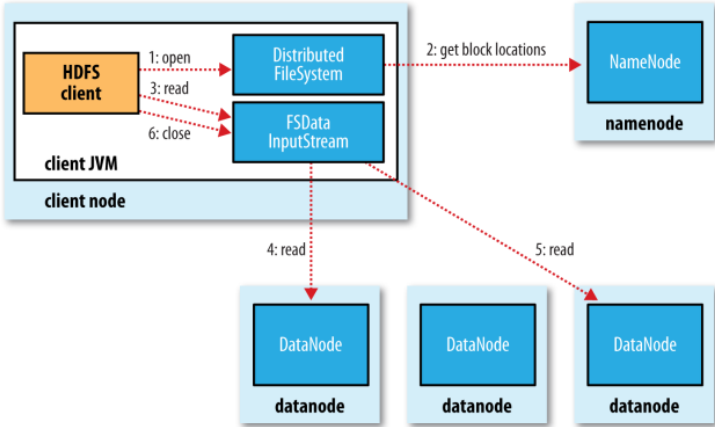
Department of Artificial Intelligence and Machine Learning

		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 namespace 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.			
2	a)	<p>Explore the reasons for stating name node single point of failure (SPOF) and how it has overcome in high availability</p> <p>The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high availability of the filesystem. The namenode is still a single point of failure (SPOF). If it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event, the whole Hadoop system would effectively be out of service until a new namenode could be brought online. To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has (i) loaded its namespace image into memory, (ii) replayed its edit log, and (iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more. The long recovery time is a problem for routine maintenance, too. In fact, because unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.</p> <p>Hadoop 2 remedied this situation by adding support for HDFS high availability (HA). In this implementation, there are a pair of namenodes in an active-standby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption. A few architectural changes are needed to allow this to happen:</p> <ul style="list-style-type: none"> • The namenodes must use highly available shared storage to share the edit log. <p>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.</p> <ul style="list-style-type: none"> • Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk. • Clients must be configured to handle namenode failover, using a mechanism that is transparent to users. • The secondary namenode's role is subsumed by the standby, which takes periodic 	6	2	2

Department of Artificial Intelligence and Machine Learning

	checkpoints of the active namenode's namespace.			
b)	<p>What are the two methods for accessing HDFS over HTTP, and how do they work?</p> <p>There are two ways of accessing HDFS over HTTP: directly, where the HDFS daemons serve HTTP requests to clients; and via a proxy (or proxies), which accesses HDFS on the client's behalf using the usual Distributed File System API. The two ways are illustrated in Figure .Both use the Web HDFS protocol.</p> <p>i) Direct access</p>  <p>ii) HDFS proxies</p>  <p>→ HTTP request - - - - -> RPC request - . - . -> block request</p> <p>In the first case, the embedded web servers in the namenode and datanodes act as WebHDFS endpoints. (WebHDFS is enabled by default, since <code>fs.webhdfs.enabled</code> is set to true.) File metadata operations are handled by the namenode, while file read (and write) operations are sent first to the namenode, which sends an HTTP redirect to the client indicating the datanode to stream file data from (or to). The second way of accessing HDFS over HTTP relies on one or more standalone proxy servers. (The proxies are stateless, so they can run behind a standard load balancer.) All traffic to the cluster passes through the proxy, so the client never accesses the namenode or datanode directly. This allows for stricter firewall and bandwidth-limiting policies to be put in place. It's common to use a proxy for transfers between Hadoop clusters located in different data centers, or when accessing a Hadoop cluster running in the cloud from an external network. The HttpFS proxy exposes the same HTTP (and HTTPS) interface as WebHDFS, so clients can access both using <code>webhdfs</code> (or <code>swebhdfs</code>) URIs. The HttpFS proxy is started independently of the namenode and datanode daemons, using the <code>httpfs.sh</code> script, and by default listens on a different port number (14000).</p>	4	2	2
3	<p>HDFS is built write-once, read-many-times pattern. Justify this statement through read operation?</p> <p>Diagram : 02 Marks</p> <p>Explanation : 08 Marks</p> <p>The client opens the file it wishes to read by calling <code>open()</code> on the <code>FileSystem</code> object, which for HDFS is an instance of <code>DistributedFileSystem</code> (step 1 in Figure). <code>DistributedFileSystem</code> calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client . If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block . The <code>DistributedFileSystem</code> returns an <code>FSDaataInputStream</code> (an input stream that supports file seeks) to the client for it to read data from. <code>FSDaataInputStream</code> in turn wraps a <code>DFSInputStream</code>, which manages the datanode and namenode I/O. The</p>	10	2	1

Department of Artificial Intelligence and Machine Learning

	<p>client then calls read() on the stream (step 3). DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4). When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream. Blocks are read in order, with the DFSInputStream opening new connections to data nodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls close() on the FSDataInputStream (step 6).</p> <p>During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInput Stream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, the DFSInputStream attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode. One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.</p>  <pre> graph LR subgraph ClientNode [client node] subgraph ClientJVM [client JVM] HDFSClient[HDFS client] end DFSInputStream[DFSInputStream] FSDataInputStream[FSDataInputStream] end NameNode[NameNode] subgraph DataNodes [datanode] DN1[DataNode] DN2[DataNode] DN3[DataNode] end HDFSClient -- "1: open" --> DFSInputStream DFSInputStream -- "3: read" --> FSDataInputStream FSDataInputStream -- "6: close" --> DFSInputStream DFSInputStream -- "2: get block locations" --> NameNode NameNode -- "4: read" --> DN1 NameNode -- "5: read" --> DN3 </pre>			
4	<p>Write a Mapreduce program using Java to find the largest integer taking large files of integers?</p> <p>Mapper Code</p> <pre> import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.LongWritable; import org.apache.hadoop.io.Text; import org.apache.hadoop.mapreduce.Mapper; import java.io.IOException; public class MaxIntegerMapper extends Mapper<LongWritable, Text, IntWritable, IntWritable> { private final static IntWritable one = new IntWritable(1); </pre>	10	4	2



Department of Artificial Intelligence and Machine Learning

```
private IntWritable number = new IntWritable();

@Override
protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
    String line = value.toString();
    try {
        int num = Integer.parseInt(line);
        number.set(num);
        context.write(number, one);
    } catch (NumberFormatException e) {
        // Ignore invalid integers
    }
}

Reducer Code
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class MaxIntegerReducer extends Reducer<IntWritable, IntWritable,
IntWritable, IntWritable> {

    private IntWritable result = new IntWritable();

    @Override
    protected void reduce(IntWritable key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
        int max = Integer.MIN_VALUE;

        for (IntWritable val : values) {
            max = Math.max(max, key.get());
        }

        result.set(max);
        context.write(result, null);
    }
}

Main Program
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxIntegerDriver {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
```



Department of Artificial Intelligence and Machine Learning

		<pre> System.err.println("Usage: MaxIntegerDriver <input path> <output path>"); System.exit(-1); } Configuration conf = new Configuration(); Job job = Job.getInstance(conf, "Max Integer Finder"); job.setJarByClass(MaxIntegerDriver.class); job.setMapperClass(MaxIntegerMapper.class); job.setReducerClass(MaxIntegerReducer.class); job.setOutputKeyClass(IntWritable.class); job.setOutputValueClass(IntWritable.class); FileInputFormat.addInputPath(job, new Path(args[0])); FileOutputFormat.setOutputPath(job, new Path(args[1])); System.exit(job.waitForCompletion(true) ? 0 : 1); } </pre>			
5	a)	<p>Combiner function is an optimization. Justify the statement?</p> <p>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. (How could it? The reduce function is still needed to process records with the same key from different maps.) 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.</p>	5	4	1
	b)	<p>Why Map tasks write their output to the local disk, not to HDFS?</p> <p>Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output. Reduce tasks don't have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for</p>	5	4	1



Department of Artificial Intelligence and Machine Learning

		reliability. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.			
--	--	---	--	--	--

Course Outcome	
CO1	Understand and apply the different building blocks of Big Data Technologies to a given problem
CO2	Articulate the programming aspect of Big Data Technologies to obtain solution to the problem through lifelong learning
CO3	Exhibit effective communication to represent the analytical aspects of Big Data Technologies for obtaining solution to the problems
CO4	Demonstrate solutions for societal and environmental concern problems using modern engineering tools through writing effective reports
CO5	Appraise the knowledge of Big Data Technologies as an Individual /as a team member

M-Marks, BT-Blooms Taxonomy Levels, CO-Course Outcomes

Marks Distribution	Particulars	CO1	CO2	CO3	CO4	CO5	L1	L2	L3	L4	L5	L6
	Max Marks	20	30	--	--	--	4	26	14	6	--	--