CMR
INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test II – June 2024

Sub:	Big Data Analytics							Sub Code:	22MCA412
Date:	28/06/2024	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

	PART I	MARKS	OBE	
			CO	RBT
1	Write a note on the following: a) Name node and data node b) HDFS high availability **OR**	5+5	CO2	L2
2	Write a note on the following: a) Failover b)fencing	5+5	CO2	L2
	PART II			
3	Write a note on hadoop eco system with a neat diagram **OR**	10	CO3	L3
4	Write a note on the following: a) Network topology b) Replica replacement	5+5	CO3	L3
	PART III			
5	Explain the anatomy of file write in hadoop **OR**	10	CO3	L3
6	Explain the anatomy of file read in hadoop	10	CO3	L3
	PART IV			
7	Differentiate between map reduce and RDBMS **OR**	10	CO2	L2
8	Differentiate between map reduce and grid computing	10	CO2	L2
	PART V			
9	Write a java program for the following: a) Displaying files from a Hadoop filesystem on standard output using a URLStreamHandler b) Displaying files from a Hadoop filesystem on standard output by using the FileSystem directly **OR**	10	CO3	L4
10	Write a java program for the following: a) Showing the file statuses for a collection of paths in a Hadoop filesystem b)Displaying files from a Hadoop filesystem on standard output twice, by using seek	10	CO3	L4

1. A) Name node and data node:

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 (fsImage)

-
-

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, since this information is reconstructed from datanodes when the system starts. A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes.

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.

B) HDFS high avialability

The combination of replicating namenode metadata on multiple filesystems, and using the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. The namenode is still a *single point of failure* (SPOF), since 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.
iii.

replayed its edit log, and

iv.
v.

received enough block reports from the datanodes to leave safe mode.

vi.

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, since unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.
The 0.23 release series of Hadoop remedies this situation by adding support for HDFS high-availability (HA). In this implementation there is a pair of- namenodes in an activestand by 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.

2. A) failover:

The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.

Failover may also be initiated manually by an adminstrator, in the case of routine maintenance, for example. This is known as a *graceful failover*, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger

a failover transition, even though the previously active namenode is still running, and thinks it is still the active namenode.

B)Fencing

The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing. The system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory (typically by using a vendor-specific NFS command), and disabling its network port via a remote management command. As a last resort, *the previously active namenode can be fenced with a technique rather graphically known as STONITH, or "shoot the other node in the head", which uses a specialized power distribution unit to forcibly power down the host machine.*

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

3. Hadoop eco system

The description of each component is as follows:

HDFS

-

Hadoop Distributed File System is the core component or the backbone of Hadoop Ecosystem.

-
-

HDFS is the one, which makes it possible to store different types of large data sets (i.e. structured, unstructured and semi structured data).

HDFS creates a level of abstraction over the resources, from where we can see the whole HDFS as a single unit.

It helps us in storing our data across various nodes and maintaining the log file about the stored data (metadata).

HDFS has two core components, i.e. **NameNode and DataNode**.

1.

The **NameNode** is the main node and it doesn't store the actual data. It contains metadata, just like a log file or you can say as a table of content. Therefore, it requires less storage and high computational resources.

2.
3.

On the other hand, all your data is stored on the **DataNodes** and hence it requires more storage resources. These DataNodes are commodity hardware (like your laptops and desktops) in the distributed environment. That's the reason, why Hadoop solutions are very cost effective.

4.
5.

You always communicate to the NameNode while writing the data. Then, it internally sends a request to the client to store and replicate data on various DataNodes.

6.

MapReduce

It is the core component of processing in a Hadoop Ecosystem as it provides the logic of processing. In other words, MapReduce is a software framework which helps in writing applications that processes large data sets using distributed and parallel algorithms inside Hadoop environment.

In a MapReduce program, **Map() and Reduce()** are two functions.

1.

The **Map function** performs actions like filtering, grouping and sorting.

2.
3.

While **Reduce function** aggregates and summarizes the result produced by map function.

4.
5.

The result generated by the Map function is a key value pair (K, V) which acts as the input for Reduce function.

6.

Apache PIG

-

PIG has two parts: **Pig Latin**, the language and **the pig runtime,** for the execution environment. You can better understand it as Java and JVM.

-
-

It supports *pig latin* language, which has SQL like command structure.

-

-

The compiler internally converts pig latin to MapReduce. It produces a sequential set of MapReduce jobs, and that's an abstraction (which works like black box).

-
-

PIG was initially developed by Yahoo.

-

-

It gives you a platform for building data flow for ETL (Extract, Transform and Load), processing and analys ing huge data sets.

-

How Pig works?

In PIG, first the load command, loads the data. Then we perform various functions on it like grouping, filtering, joining, sorting, etc. At last, either you can dump the data on the screen or you can store the result back in HDFS.

Apache Hive

-

Facebook created HIVE for people who are fluent with SQL. Thus, HIVE makes them feel at home

-

-

Basically, HIVE is a data warehousing component which performs reading, writing and managing large data sets in a distributed environment using SQL-like interface.

-

HIVE + SQL = HQL

-

The query language of Hive is called Hive Query Language(HQL), which is very similar like SQL.

-
-

It has 2 basic components: **Hive Command Line and JDBC/ODBC driver**.

-
-

The **Hive Command line** interface is used to execute HQL commands.

-
-

While, Java Database Connectivity (**JDBC**) and Object Database Connectivity (**ODBC**) is used to establish connection from data storage.

-
-

Secondly, Hive is highly scalable. As, it can serve both the purposes, i.e. large data set processing (i.e. Batch query processing) and real time processing (i.e. Interactive query processing).

-
-

It supports all primitive data types of SQL.

-
-

You can use predefined functions, or write tailored user defined functions (UDF) also to accomplish your specific needs.

-

Apache Mahout

Now, let us talk about Mahout which is renowned for machine learning. Mahout provides an environment for creating machine learning applications which are scalable.

Apache Spark

-

Apache Spark is a framework for real time data analytics in a distributed computing environment.

-
-

The Spark is written in Scala and was originally developed at the University of California, Berkeley.

-
-

It executes in-memory computations to increase speed of data processing over Map-Reduce.

-
-

It is 100x faster than Hadoop for large scale data processing by exploiting in-memory computations and other optimizations. Therefore, it requires high processing power than Map-Reduce.

-

As you can see, Spark comes packed with high-level libraries, including support for R, SQL, Python, Scala, Java etc. These standard libraries increase the seamless integrations in complex workflow. Over this, it also allows various sets of services to integrate with it like MLlib, GraphX, SQL + Data Frames, Streaming services etc. to increase its capabilities.

Apache HBase

-

HBase is an open source, non-relational distributed database. In other words, it is a NoSQL database.

-
-

It supports all types of data and that is why, it's capable of handling anything and everything inside a Hadoop ecosystem.

-
-

It is modelled after Google's BigTable, which is a distributed storage system designed to cope up with large data sets.

-
-

The HBase was designed to run on top of HDFS and provides BigTable like capabilities.

-

-

It gives us a fault tolerant way of storing sparse data, which is common in most Big Data use cases.

-
-

The HBase is written in Java, whereas HBase applications can be written in REST, Avro and Thrift APIs.

For better understanding, let us take an example. You have billions of customer emails and you need to find out the number of customers who has used the word complaint in their emails. The request needs to be processed quickly (i.e. at real time). So, here we are handling a large data set while retrieving a small amount of data. For solving these kind of problems, HBase was designed.

Apache Zookeeper

Apache Zookeeper is the coordinator of any Hadoop job which includes a combination of various services in a Hadoop Ecosystem.

Apache Zookeeper coordinates with various services in a distributed environment.

Before Zookeeper, it was very difficult and time consuming to coordinate between different services in Hadoop Ecosystem. The services earlier had many problems with interactions like common configuration while synchronizing data. Even if the services are configured, changes in the configurations of the services make it complex and difficult to handle. The grouping and naming was also a time-consuming factor.
Due to the above problems, Zookeeper was introduced. It saves a lot of time by performing **synchronization, configuration maintenance, grouping and naming.**
Although it's a simple service, it can be used to build powerful solutions.

Big names like Rackspace, Yahoo, eBay use this service in many of their use cases and therefore, you can have an idea about the importance of Zookeeper.

Apache Oozie

Consider Apache Oozie as a clock and alarm service inside Hadoop Ecosystem. For Apache jobs, Oozie has been just like a scheduler. It schedules Hadoop jobs and binds them together as one logical work.
There are two kinds of Oozie jobs:

1. **Oozie workflow**: These are sequential set of actions to be executed**.** You can assume it as a relay race. Where each athlete waits for the last one to complete his part.
2.
3. **Oozie Coordinator**: These are the Oozie jobs which are triggered when the data is made available to it. Think of this as the response-stimuli system in our body. In the same manner as we respond to an external stimulus, an Oozie coordinator responds to the availability of data and it rests otherwise.
4.

Apache Flume

Ingesting data is an important part of our Hadoop Ecosystem.

The Flume is a service which helps in ingesting unstructured and semi-structured data into HDFS.

-
-

It gives us a solution which is reliable and distributed and helps us in **collecting, aggregating** and **moving large amount of data sets**.

-
-

It helps us to ingest online streaming data from various sources like network traffic, social media, email messages, log files etc. in HDFS.

-

Apache Sqoop

Now, let us talk about another data ingesting service i.e. Sqoop. The major difference between Flume and Sqoop is that:

-

Flume only ingests unstructured data or semi-structured data into HDFS.

-
-

While Sqoop can import as well as export structured data from RDBMS or Enterprise data warehouses to HDFS or vice versa.

-

When we submit Sqoop command, our main task gets divided into sub tasks which is handled by individual Map Task internally. Map Task is the sub task, which imports part of data to the Hadoop Ecosystem. Collectively, all Map tasks imports the whole data.

Apache Ambari

Ambari is an Apache Software Foundation Project which aims at making Hadoop ecosystem more manageable.
It includes software for **provisioning, managing and monitoring**Apache Hadoop clusters. The Ambari provides:

1.

Hadoop cluster provisioning:

2.
    -

    It gives us step by step process for installing Hadoop services across a number of hosts.

    -
    -

    It also handles configuration of Hadoop services over a cluster.

    -

3.

Hadoop cluster management:

4.

-

It provides a central management service for starting, stopping and re-configuring Hadoop services across the cluster.

-

5.

Hadoop cluster monitoring:

6.

-

For monitoring health and status, Ambari provides us a dashboard.

-
-

The **Amber Alert framework** is an alerting service which notifies the user, whenever the attention is needed. For example, if a node goes down or low disk space on a node, etc.

-

4. a)Network topology
What does it mean for two nodes in a local network to be “close” to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity. The idea is to use the bandwidth between two nodes as a measure of distance.

Rather than measuring bandwidth between nodes, which can be difficult to do in practice (it requires a quiet cluster, and the number of pairs of nodes in a cluster grows as

the square of the number of nodes), Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on. The idea is that the bandwidth available for each of the following scenarios becomes progressively less:
• Processes on the same node
• Different nodes on the same rack
• Nodes on different racks in the same data center
• Nodes in different data centers7
For example, imagine a node n1 on rack r1 in data center d1. This can be represented as /d1/r1/n1. Using this notation, here are the distances for the four scenarios:
• distance(/d1/r1/n1, /d1/r1/n1) = 0 (processes on the same node)
• distance(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes on the same rack)
• distance(/d1/r1/n1, /d1/r2/n3) = 4 (nodes on different racks in the same data center)
• distance(/d1/r1/n1, /d2/r3/n4) = 6 (nodes in different data centers)

B)Replica replacement
How does the namenode choose which datanodes to store replicas on? There's a trade-off between reliability and write bandwidth and read bandwidth here. For example,

placing all replicas on a single node incurs the lowest write bandwidth penalty since the replication pipeline runs on a single node, but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of placement strategies. Indeed, Hadoop changed its placement strategy in release 0.17.0 to one that helps keep a fairly even distribution of blocks across the cluster. (See "balancer" on page 348 for details on keeping a cluster balanced.) And from 0.21.0, block placement policies are pluggable.
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 on the cluster, although the system tries to avoid placing too many replicas on the same rack.

Once the replica locations have been chosen, a pipeline is built, taking network topology into account. For a replication factor of 3, the pipeline might look like Figure 3-5.

Overall, this strategy gives a good balance among reliability (blocks are stored on two racks), write bandwidth (writes only have to traverse a single network switch), read performance (there's a choice of two racks to read from), and block distribution across the cluster (clients only write a single block on the local rack).

5.Anatomy of file write

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 (step 5). If a 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 the remainder of the block's data is written to the two 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.
It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication.min replicas (default one) are written, the write will succeed,

and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to three).

When the client has finished writing data, it calls close() on the stream (step 6). 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 (step

7). The namenode already knows which blocks the file is made up of (via Data Streamer asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

6.Anatomy of file read

The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1 in Figure 3-2). DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for 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 (according to the topology of the cluster's network; see "Network Topology and Hadoop" on page 71). If

the client is itself a datanode (in the case of a MapReduce task, for instance), then it will read from the local datanode, if it hosts a copy of the block (see also Figure 2-2).

The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps

a DFSInputStream, which manages the datanode and namenode I/O.

The 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 datanodes 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).

During reading, if the DFSInputStream encounters an error while communicating with a datanode, then 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 DFSInputStream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the DFSInput Stream attempts to read a replica of the block from another datanode.

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, since the data traffic is spread across all the datanodes in the cluster. The namenode meanwhile 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.

7.Map reduce VS RDBMS

Some more differences between Mapreduce and RDBMS are listed out in the table 3.3 Table 3.3:
Difference between MapReduce and RDBMS

#	MapReduce	RDBMS
1	Good fit for problems that analyze the whole data set in a batch fashion	RDBMS is good for point queries or updates, where the data set has been ordered to deliver low latency retrieval
2	It suits well for applications where the data is written once and read many times	Relational database is good for data that are continuously updated
3	Works on semi-structured or unstructured data	Operates on structured data
4	Ex: spreadsheets, images, text etc..	Ex: database tables, XML docs etc..
5	It is designed to interpret the data at processing time (Schema on Read)	It is designed to interpret the data run time (Schema on Write)
6	Normalization creates a problem in Hadoop because, it reading a record is non-local operation, instead Hadoop makes it possible to perform streaming reads and writes	RDBMS data is often normalized to avoid redundancy and to retain integrity.
7	MapReduce can process the data in parallel	Parallel processing is not true for SQL RDBMS queries

8.Map reauce VS Grid Computing

Sl. No	Grid Computing MPI	MapReduce
1.	Works well for predominantly compute intensive jobs, but it becomes a problem where nodes access larger data volumes since the network bandwidth computing node becomes idle	Hadoop tries to collocate with the data and compute nodes so that the data access is fast because of data locality.
2.	MPI programs have to explicitly manage their own checkpoint and recovery which gives more control to the programmer but makes them more difficult	In MapReduce, since the implementation detects the failed task and re-schedules replacement on the machine.
3.	MPI is a shared architecture	MapReduce is a share nothing architecture
4.	Gives great control to the programmer, it also requires that they explicitly handle the mechanics of data flow	Processing in Hadoop happens only at higher level the program thinks in terms of data model since the data flow is implicit

9. a)

```
public class URLCat {
static {
URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
}
public static void main(String[] args) throws Exception {
InputStream in = null;
try {
in = new URL(args[0]).openStream();
IOUtils.copyBytes(in, System.out, 4096, false);
```

```
} finally {
IOUtils.closeStream(in);
}
}
}
```

Output:

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

B)

```
public class FileSystemCat {
public static void main(String[] args) throws Exception {
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
InputStream in = null;
try {
in = fs.open(new Path(uri));
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
}
}
}
```

The program runs as follows:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

10.A)

```
public class ListStatus {
public static void main(String[] args) throws Exception {
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
Path[] paths = new Path[args.length];
for (int i = 0; i < paths.length; i++) {
paths[i] = new Path(args[i]);
}
FileStatus[] status = fs.listStatus(paths);
Path[] listedPaths = FileUtil.stat2Paths(status);
for (Path p : listedPaths) {
System.out.println(p);
}
}
}
```

Output:

```
% hadoop ListStatus hdfs://localhost/ hdfs://localhost/user/tom
hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

B)

```
public class FileSystemDoubleCat {
public static void main(String[] args) throws Exception {
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
FSDataInputStream in = null;
try {
in = fs.open(new Path(uri));
IOUtils.copyBytes(in, System.out, 4096, false);
in.seek(0); // go back to the start of the file
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
}
}
}
```

Here's the result of running it on a small file:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```