CMR INSTITUTE OF TECHNOLOGY

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

Internal Assessment Test 2 – February 2023

Sub:	BIG DATA ANALYTICS							Sub Code:	20MCA352
Date:	08/02/2023	Duration:	90 min's	Max Marks:	50	Sem:	III	Branch:	MCA

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

		MARKS	OBE	
			CO	RBT
	PART I			
1	Write briefly about the following : a) History of Hadoop. b) Hadoop Releases.	[10]	CO3	L2
	OR			
2	What are Grid Computing and Volunteer Computing? List their differences from MapReduce.	[10]	CO3	L3
	PART II			
3	Briefly explain what is HDFS? Discuss about its strengths and weaknesses.	[10]	CO4	L2
	OR			
4	Explain the following HDFS concepts. a) Blocks b) Data Node and Name Node.	[10]	CO4	L2

CMR INSTITUTE OF TECHNOLOGY

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

Internal Assessment Test 2 – February 2023

Sub:	BIG DATA ANALYTICS							Sub Code:	20MCA352
Date:	08/02/2023	Duration:	90 min's	Max Marks:	50	Sem:	III	Branch:	MCA

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

		MARKS	OBE	
			CO	RBT
1	**PART I** Write briefly about the following : c) History of Hadoop d) Hadoop Releases.	[10]	CO3	L2
	OR			
2	What are Grid Computing and Volunteer Computing? List their differences from Map Reduce.	[10]	CO3	L3
	PART II			
3	Briefly explain what is HDFS? Discuss about its strengths and weaknesses.	[10]	CO4	L2
	OR			
4	Explain the following HDFS concepts. a) Blocks b) Data Node and Name Node.	[10]	CO4	L2

PART III

5	a) On what regards or rather, how is MapReduce different from RDBMS? b) Explain HDFS Federation and HDFS High Availability.	[4+6]	CO3 & CO4	L2
	OR			
6	List and very briefly explain the various components of Hadoop Ecosystem.	[10]	CO3	L2
	PART IV			
7	How will you read data using the File System API?	[10]	CO4	L4
	OR			
8	Write the code for How will you read data from a Hadoop URL.	[10]	CO4	L4
	PART V			
9	Write a code to display files from a Hadoop file system on standard output?	[10]	CO4	L4
	OR			
10	Write a code to copy a local file to a Hadoop filesystem?	[10]	CO4	L4

PART III

5	a) On what regards or rather, how is MapReduce different from RDBMS? b) Explain HDFS Federation and HDFS High Availability.	[4+6]	CO3 & CO4	L2
	OR			
6	List and very briefly explain the various components of Hadoop Ecosystem.	[10]	CO3	L2
	PART IV			
7	How will you read data using the File System API?	[10]	CO4	L4
	OR			
8	Write the code for How will you read data from a Hadoop URL.	[10]	CO4	L4
	PART V			
9	Write a code to display files from a Hadoop file system on standard output?	[10]	CO4	L4
	OR			
10	Write a code to copy a local file to a Hadoop filesystem?	[10]	CO4	L4

IAT 2- Solution – Big Data Analytics – 20MCA352

1.**History of Hadoop**

- In 2002, Nutch was started
- A working crawler and search system emerged
- Its architecture wouldn't scale to the billions of pages on the Web
- In 2003, Google published a paper describing the architecture of Google's distributed filesystem, GFS
- In 2004, Nutch project implemented the GFS idea into the Nutch Distributed Filesystem, NDFS
- In 2004, Google published the paper introducing MapReduce
- In 2005, Nutch had a working MapReduce implementation in Nutch
- By the middle of that year, all the major Nutch algorithms had been ported to run using MapReduce and NDFS
- In Feb. 2006, Doug Cutting started an independent subproject of Lucene, called Hadoop
- In Jan. 2006, Doug Cutting joined Yahoo!
- Yahoo! Provided a dedicated team and the resources to turn Hadoop into a system at web scale
- In Feb. 2008, Yahoo! announced its search index was being generated by a 10,000 core Hadoop cluster
- In Apr. 2008, Hadoop broke a world record to sort a terabytes of data
- In Nov. 2008, Google reported that its MapReduce implementation sorted one terabytes in 68 seconds.
- In May 2009, Yahoo! used Hadoop to sort one terabytes in 62 seconds

b. **Hadoop Releases**

There are a few active release series. The 1.x release series is a continuation of the 0.20 release series, and contains the most stable versions of Hadoop currently available. This series includes secure Kerberos authentication, which prevents unauthorized access to Hadoop data. Almost all production clusters use these releases, or derived versions (such as commercial distributions).The 0.22 and 0.23 release series are currently marked as alpha releases (as of early 2012), but this is likely to change by the time you read this as they get more real-world testing and become more stable (consult the Apache Hadoop releases page for the latest status). 0.23 includes several major new features:

A new MapReduce runtime, called MapReduce 2, implemented on a new system called YARN (Yet Another Resource Negotiator), which is a general resource management system for running distributed applications. MapReduce 2 replaces the classic runtime in previous releases. It is described in more depth in "YARN".HDFS federation, which partitions the HDFS namespace across multiple namenodes to support clusters with very large numbers of files.HDFS high-availability, which removes the namenode as a single point of failure by supporting standby namenodes for failover.

The following figure shows the configuration of Hadoop 1.0 and Hadoop 2.0.

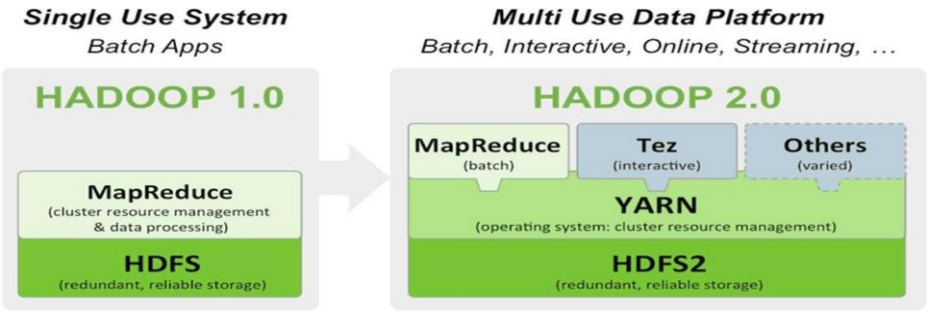

Table 1 covers features in HDFS and MapReduce. Other projects in the Hadoop ecosystem are continually evolving too, and picking a combination of components that work well together can be a challenge.

Table 1. Features Supported by Hadoop Release Series

Feature	1.x	0.22	0.23
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old MapReduce API	Yes	Deprecated	Deprecated
New MapReduce API	Partial	Yes	Yes
MapReduce 1 runtime (Classic)	Yes	Yes	No
MapReduce 2 runtime (YARN)	No	No	Yes
HDFS federation	No	No	Yes
HDFS high-availability	No	No	Planned

2. **Grid Computing:** can be defined as a network of computers working together to perform a task that would rather be difficult for a single machine. All machines on that network work under the same protocol to act as a virtual supercomputer. The task that they work on may include analyzing huge datasets or simulating situations that require high computing power. Computers on the network contribute resources like processing power and storage capacity to the network.

   Both Grid Computing and MapReduce are efficient and work well with the predominant computer intensive, but it comes a problem when nodes need to access large data volumes (hundreds of gigabytes), since network bandwidth is the bottleneck problem and compute becomes idle. When comes to the MapReduce it works very well even there is a need of accessing large data volumes even in hundreds of gigabytes.

   Volunteer Computing: Volunteer processing ventures work by breaking the issues they are attempting to settle into pieces called work units, which are sent to PCs around the globe to be dissected. For instance, a **SETI@home** work unit is about 0.35 MB of radio telescope information and takes hours or days to examine on a commonplace home PC. At the point when the investigation is finished, the results are sent back to the server, and the customer gets another work unit.

   MapReduce also works in the similar way of breaking a problem into independent pieces that work in parallel. Volunteer computing problem is very CPU intensive,which makes it suitable for running on hundreds of thousands of computers across the world because the time to transfer the work unit is dwarfed by time to run the computation time. Volunteers are donating CPU cycle not the bandwidth.

3. **Hadoop Distributed File System(HDFS)** 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).

Strengths of HDFS

1. Very large files: There are Hadoop clusters running today that store petabytes of data.
2. Streaming data access HDFS cares about the time to read the whole dataset than the latency of reading the first record.
3. Commodity hardware Hadoop is designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors) for which the chance of node failure across the cluster is high. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

Weaknesses of HDFS

1. **Low-latency data access:** Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS.
2. **Lots of small files**: Since 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, while storing millions of files is feasible, billions is beyond the capability of current hardware.
3. **Multiple writers, arbitrary file modifications:** Files in HDFS may be written to by a single writer. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

4a. **Blocks:** A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes.

HDFS too has the concept of a block, but it is a much larger unit of 64 MB or 128MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full blocks worth of underlying storage. There are tools to perform filesystem maintenance, such as *df* and *fsck*, that operate on the filesystem block level.

Benefits of using block abstraction

1. *A file can be larger than any single disk in the network*: There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.
2. *Making the unit of abstraction a block rather than a file simplifies the storage subsystem:* Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied.

4b. A HDFS cluster has two types of node operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers).

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.

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.

1. 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.

2. 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, since 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.

5a. Differences between RDBMS and MapReduce are:

#	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: spread sheets, 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

5b. **HDFS Federation**: The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. *For example, one namenode might manage all the files rooted under /user, say, and a second namenode might handle files under /share.*

Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

HDFS High-Availability : 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. 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, 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.

The architectural changes are needed to replace active name node with standby name node:

i. 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.

ii. Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

iii. If the active namenode fails, then the standby can take over very quickly (in a few tens of seconds) since it has the latest state available in memory: both the latest edit log entries, and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), since the system needs to be conservative in deciding that the active namenode has failed.

iv. In the unlikely event of the standby being down when the active fails, the administrator can still start the standby from cold. This is no worse than the non- HA case, and from an operational point of view it's an improvement, since the process is a standard operational procedure built into Hadoop.

6. The components of Hadoop 1.0 and Hadoop 2.0 are listed as follows:

- Common – a set of components and interfaces for filesystems and I/O.
- Avro – a serialization system for RPC and persistent data storage.
- MapReduce – a distributed data processing model. MapReduce is a software framework which helps in writing applications that processes large data sets using distributed and parallel algorithms inside Hadoop environment.
- YARN - YARN is responsible for allocating system resources to the various applications running in a Hadoop cluster and scheduling tasks to be executed on different cluster nodes.
- HDFS – a distributed filesystem running on large clusters of machines.
- Pig – a data flow language and execution environment for large datasets. It gives you a platform for building data flow for ETL (Extract, Transform and Load), processing and analyzing huge data sets.
- Hive – a distributed data warehouse providing SQL-like query language.
- 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.
- Mahout - Mahout provides an environment for creating machine learning applications which are scalable.
- Apache Drill -SQL on Hadoop
- ZooKeeper – a distributed, highly available coordination service.
- Sqoop – Sqoop can import as well as export structured data from RDBMS or Enterprise data warehouses to HDFS or vice versa..
- Oozie - For Apache jobs, Oozie has been just like a scheduler. It schedules Hadoop jobs and binds them together as one logical work.
- Flume- The Flume is a service which helps in ingesting unstructured and semi-structured data into HDFS.
- Solr & Lucene - Searching & Indexing
- Ambari - Provision, Monitor and Maintain cluster

7.

```
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);
}
  }
}
```

8.

```
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);
      }
  }
}
```

9.

```
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);
    }
  }
}
```

10.

```
public class FileCopyWithProgress {
  public static void main(String[] args) throws Exception {
    String localSrc = args[0];
    String dst = args[1];
    InputStream in = new BufferedInputStream(new FileInputStream(localSrc));
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(dst), conf);
    OutputStream out = fs.create(new Path(dst), new Progressable() {
      public void progress() {
        System.out.print(".");
      } });
    IOUtils.copyBytes(in, out, 4096, true);
  }
}
```