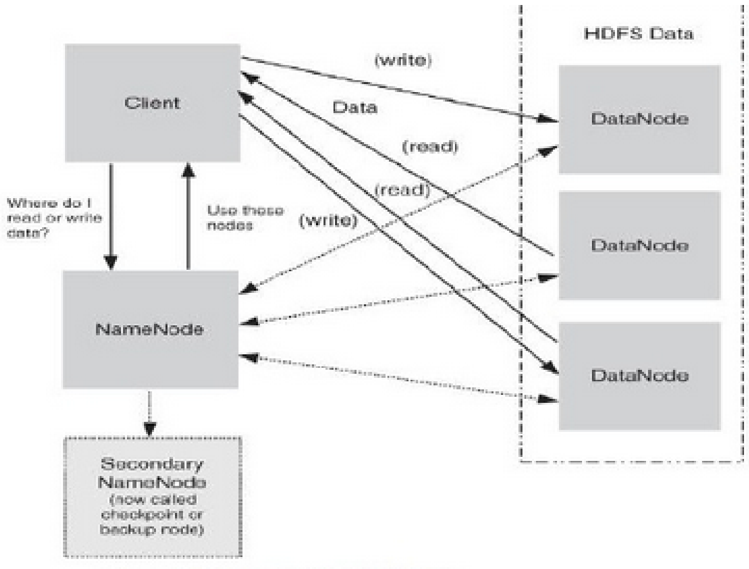
1.	Explain HDFS File system	**4**
	The Hadoop Distributed File System (HDFS) was designed for Big Data processing. Although capable of supporting many users simultaneously, HDFS is not designed as a true parallel file system. Rather, the design assumes a large file write-once/read-many model that enables other optimizations and relaxes many of the concurrency and coherence overhead requirements of a true parallel file system. For instance, HDFS rigorously restricts data writing to one user at a time. All additional writes are "append-only," and there is no random writing to HDFS files. Bytes are always appended to the end of a stream, and byte streams are guaranteed to be stored in the order written. HDFS is designed for data streaming where large amounts of data are read from disk in bulk. The HDFS block size is typically 64MB or 128MB. Thus, this approach is entirely unsuitable for standard POSIX file system use. In addition, due to the sequential nature of the data, there is no local caching mechanism. The large block and file sizes make it more efficient to reread data from HDFS than to try to cache the data.	
	Perhaps the most interesting aspect of HDFS—and the one that separates it from other file systems—is its data locality. A principal design aspect of Hadoop MapReduce is the emphasis on moving the computation to the data rather than moving the data to the computation. This distinction is reflected in how Hadoop clusters are implemented. In other high-performance systems, a parallel file system will exist on hardware separate from the compute hardware. Data is then moved to and from the computer components via high-speed interfaces to the parallel file system array. HDFS, in contrast, is designed to work on the same hardware as the compute portion of the cluster. That is, a single server node in the cluster is often both a computation engine and a storage engine for the application.	

	Finally, Hadoop clusters assume node (and even rack) failure will occur at some point. To deal with this situation, HDFS has a redundant design that can tolerate system failure and still provide the data needed by the compute part of the program.	
2.	Explain the important aspects of HDFS	**8**
	• The write-once/read-many design is intended to facilitate streaming reads. • Files may be appended, but random seeks are not permitted. There is no caching of data. • Converged data storage and processing happen on the same server nodes. • "Moving computation is cheaper than moving data." • A reliable file system maintains multiple copies of data across the cluster. Consequently, failure of a single node (or even a rack in a large cluster) will not bring down the file system. • A specialized file system is used, which is not designed for general use.	

2.	Show the client/NameNode/DataNode interaction in HDFS	**8**
	The client/NameNode/DataNode interaction is provided in Figure 3.1. When a client writes data, it first communicates with the NameNode and requests to create a file. The NameNode determines how many blocks are needed and provides the client with the DataNodes that will store the data. As part of the storage process, the data blocks are replicated after they are written to the assigned node. Depending on how many nodes are in the cluster, the NameNode will attempt to write replicas of the data blocks on nodes that are in other separate racks (if possible). If there is only one rack, then the replicated blocks are written to other servers in the same rack. After the DataNode acknowledges that the file block replication is complete, the client closes the file and informs the NameNode that the operation is complete. Note that the NameNode does not write any data directly to the DataNodes. It does, however, give the client a limited amount of time to complete the operation. If it does not complete in the time period, the operation is canceled.	

	 HDFS Data Client (write) Data (read) (read) (write) Where do I read or write data? Use these nodes DataNode DataNode DataNode NameNode Secondary NameNode (now called checkpoint or backup node) Figure 3.1 Various system roles in an HDFS deployment	
5	**3.** Explain the various roles in HDFS.	**10**
	The various roles in HDFS can be summarized as follows: • HDFS uses a master/slave model designed for large file reading/streaming. • The NameNode is a metadata server or "data traffic cop." • HDFS provides a single namespace that is managed by the NameNode. • Data is redundantly stored on DataNodes; there is no data on the NameNode. • The SecondaryNameNode performs checkpoints of NameNode file system's state but is not a failover node.	

7	*Explain* **HDFS Block Replication process**	**6**
	When HDFS writes a file, it is replicated across the cluster. The amount of replication is based on the value of dfs.replication in the hdfs-site.xml file. This default value can be overruled with the hdfsdfs-setrep command. For Hadoop clusters containing more than eight DataNodes, the replication value is usually set to 3. In a Hadoop cluster of eight or fewer DataNodes but more than one DataNode, a replication factor of 2 is adequate. If several machines must be involved in the serving of a file, then a file could	

be rendered unavailable by the loss of any one of those machines. HDFS combats this problem by replicating each block across a number of machines (three is the default).

In addition, the HDFS default block size is often 64MB. In a typical operating system, the block size is 4KB or 8KB. The HDFS default block size is not the minimum block size, however. If a 20KB file is written to HDFS, it will create a block that is approximately 20KB in size. (The underlying file system may have a minimal block size that increases the actual file size.) If a file of size 80MB is written to HDFS, a 64MB block and a 16MB block will be created.

HDFS are not exactly the same as the data splits used by the MapReduce process. The HDFS blocks are based on size, while the splits are based on a logical partitioning of the data. For instance, if a file contains discrete records, the logical split ensures that a record is not split physically across two separate servers during processing. Each HDFS block may consist of one or more splits.

Figure 3.2 provides an example of how a file is broken into blocks and replicated across the cluster. In this case, a replication factor of 3 ensures that any one DataNode can fail and the replicated blocks will be available on other nodes—and then sı

nodes—and then sı

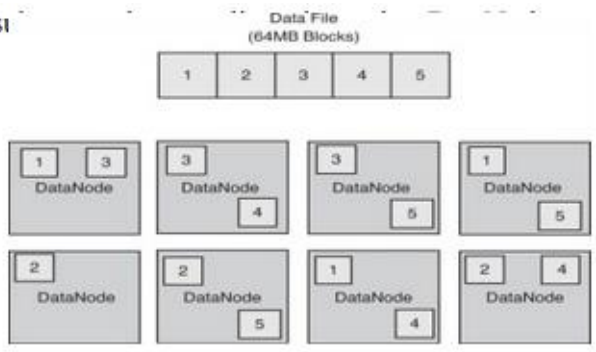
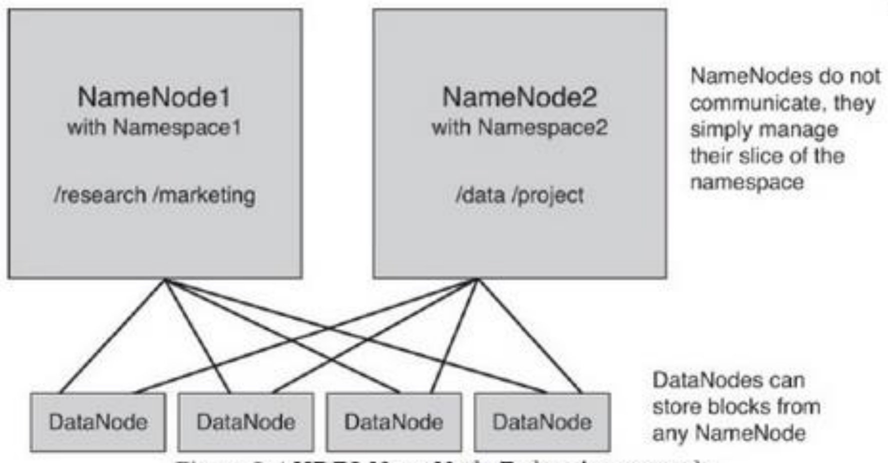


Figure 3.2

8	**Explain** how HDFS NameNode Federation is accomplished	**6**

	HDFS NameNode Federation Another important feature of HDFS is NameNode Federation. Older versions ofHDFS provided a single namespace for the entire cluster managed by a singleNameNode. Thus, the resources of a single NameNode determined the size ofthe namespace. Federation addresses this limitation by adding support formultiple NameNodes/namespaces to the HDFS file system. The key benefits areas follows: *Namespace scalability.* HDFS cluster storage scales horizontally without placing a burden on the NameNode. *Better performance.* Adding more NameNodes to the cluster scales the file system read/write operations throughput by separating the total namespace. *System isolation.* Multiple NameNodes enable different categories of applications to be distinguished, and users can be isolated to different name spaces. Figure 3.4 illustrates how HDFS NameNode Federation is accomplished.NameNode1 manages the /research and /marketing namespaces, andNameNode2 manages the /data and /project namespaces. The NameNodes do not communicate with each other and the DataNodes "just store data block" as directed by either NameNode.  NameNode1 with Namespace1 /research /marketing NameNode2 with Namespace2 /data /project NameNodes do not communicate, they simply manage their slice of the namespace DataNode DataNode DataNode DataNode DataNodes can store blocks from any NameNode Figure 3.4 HDFS NameNode Federation example	
9	**Explain HDFS Checkpoints and Backups**	**6**
	NameNode stores the metadata of the HDFS files ystem in a file called fs_image. File systems modifications are written to anedits log file, and at	

	startup the NameNode merges the edits into a new fs_image. The Secondary NameNode or CheckpointNode periodically fetchesedits from the NameNode, merges them, and returns an updated fs_image to theNameNode. An HDFS BackupNode is similar, but also maintains an up-to-date copy of the file system namespace both in memory and on disk. Unlike a CheckpointNode,the BackupNode does not need to download the fsi_mage and edits files from the active NameNode because it already has an up-to-date namespace state in memory. A NameNode supports one BackupNode at a time. NoCheckpointNodes may be registered if a Backup node is in use.	
10.	**Illustrate HDFS Snapshots**	**6**
	HDFS snapshots are similar to backups, but are created by administrators using the hdfs dfs snapshot command. HDFS snapshots are read-only point-in time copies of the file system. They offer the following features: • Snapshots can be taken of a sub-tree of the file system or the entire files system. • Snapshots can be used for data backup, protection against user errors, and disaster recovery. • Snapshot creation is instantaneous. • Blocks on the DataNodes are not copied, because the snapshot files record the block list and the file size. There is no data copying, although	

to/from their local file system.

- Users can stream data directly to HDFS through the mount point. Appending to a file is supported, but random write capability is not supported.

10	**List General HDFS Commands**	**8**

List Files in HDFS

To list the files in the root HDFS directory, enter the following command:

- $ hdfsdfs -ls /

To list files in your home directory, enter the following command:

- $ hdfsdfs –ls

Make a Directory in HDFS

To make a directory in HDFS, use the following command. As with the –ls command, when no path is supplied, the user's home directory is used (e.g.,/users/hdfs).

- $ hdfsdfs -mkdir stuff

Copy Files to HDFS

To copy a file from your current local directory into HDFS, use the following command. If a full path is not supplied, your home directory is assumed. In this case, the file test is placed in the directory stuff that was created previously.

- $ hdfsdfs -put test stuff

The file transfer can be confirmed by using the -ls command:

- $ hdfsdfs -ls stuff

```
Found 1 items
-rw-r--r-- 2 hdfshdfs 12857 2015-05-29 13:12 stuff/test
```

Copy Files from HDFS

Files can be copied back to your local file system using the following command.

In this case, the file we copied into HDFS, test, will be copied back to the current local directory with the name test-local.

- $ hdfsdfs -get stuff/test test-local

Copy Files within HDFS

The following command will copy a file in HDFS:

- $ hdfsdfs -cp stuff/test test.hdfs

Delete a File within HDFS

The following command will delete the HDFS file test.dhfsthat was created previously:

- $ hdfsdfs -rmtest.hdfs

Moved: 'hdfs://limulus:8020/user/hdfs/stuff/test' to trash at:hdfs://limulus:8020/user/hdfs/.Trash/Current

Note that when the fs.trash.interval option is set to a non-zero value in core-site.xml, all deleted files are moved to the user's .Trash directory. This can be avoided by including the -skipTrashoption.

- $ hdfsdfs -rm –skip Trash stuff/test

Deleted stuff/test

Delete a Directory in HDFS

The following command will delete the HDFS directory stuff and all its contents:

- $ hdfsdfs -rm -r -skipTrash stuff

Deleted stuff

Get an HDFS Status Report

Regular users can get an abbreviated HDFS status report using the following command. Those with HDFS administrator privileges will get a full (and potentially long) report. Also, this command uses dfs admin instead of dfs to invoke administrative commands. The status report is similar to the data

	presented in the HDFS web GUI . • $ hdfsdfsadmin –report	
11	**Explain the compilation steps for a java file on LINUX system.**	**5**
	Linux systems using the following steps. First, create a directory tohold the classes: • $ mkdir HDFSClient-classes Next, compile the program using 'hadoopclasspath' to ensure all theclass paths are available: • $ javac -cp 'hadoopclasspath' -d HDFSClient-classes HDFSClient.java Finally, create a Java archive file: $ jar -cvfe HDFSClient.jar org/myorg.HDFSClient -C HDFSClientclasses/ . The program can be run to check for available options as follows: • $ hadoop jar ./HDFSClient.jar Usage: hdfsclient add/read/delete/mkdir [<local_path><hdfs_path>] A simple file copy from the local system to HDFS can be accomplished using the following command: • $ hadoop jar ./HDFSClient.jar add ./NOTES.txt /user/hdfs The file can be seen in HDFS by using the hdfsdfs -ls command: • $ hdfsdfs -ls NOTES.txt -rw-r--r-- 2 hdfshdfs 502 2015-06-03 15:43 NOTES.txt	
12	**Explain The MapReduce Model**	**4**
	The MapReduce computation model provides a very powerful tool for many applications and is more common than most users realize. Its underlying idea is very simple. There are two stages: a mapping stage and a reducing stage. In the mapping stage, a *mapping procedure* is applied to input data. The map is usually some kind of filter or sorting process. For instance, assume you need to count how many times the name “Kutuzov” appears in the novel *War and Peace*. One solution is to gather 20 friends and	

	give them each a section of the book to search. This step is the map stage. The reduce phase happens when everyone is done counting	
13	Write simple Mapper and Reducer Scripts and write a brief description on it.	**8**

	#!/bin/bash
	while read line ; do
	for token in $line; do
	if ["$token" = "Kutuzov"] ; then
	echo "Kutuzov,1"
	elif ["$token" = "Petersburg"] ; then
	echo "Petersburg,1"
	fi
	done
	done
	Listing 5.2 **Simple Reducer Script**
	
	#!/bin/bash
	kcount=0
	pcount=0
	while read line ; do
	if ["$line" = "Kutuzov,1"] ; then
	let kcount=kcount+1
	elif ["$line" = "Petersburg,1"] ; then	

	let kcount=kcount+1 elif ["$line" = "Petersburg,1"] ; then let pcount=pcount+1 fi done echo "Kutuzov,$kcount" echo "Petersburg,$pcount" Formally, the MapReduce process can be described as follows. The mapper and reducer functions are both defined with respect to data structured in (key,value) pairs. The mapper takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain: Map(key1,value1) → list(key2,value2) The reducer function is then applied to each key–value pair, which in turn produces a collection of values in the same domain:	

	Reduce(key2, list (value2)) → list(value3) Each reducer call typically produces either one value (value3) or an empty response. Thus, the MapReduce framework transforms a list of (key, value) pairs into a list of values. The MapReduce model is inspired by the map and reduce functions commonly used in many functional programming languages. The functional nature of MapReduce has some important properties: Data flow is in one direction (map to reduce). It is possible to use the output of a reduce step as the input to another MapReduce process. As with functional programing, the input data are not changed. By applying the mapping and reduction functions to the input data, new data are produced. Because there is no dependency on how the mapping and reducing functions are applied to the data, the mapper and reducer data flow can be implemented in any number of ways to provide better performance. Distributed (parallel) implementations of MapReduce enable large amounts of data to be analyzed quickly. In general, the mapper process is fully scalable and can be applied to any subset of the input data. Results from multiple parallel mapping functions are then combined in the reducer phase.	

14	**What are** The important properties of functional nature of MapReduce?	**8**
	The functional nature of MapReduce has some important properties: • Data flow is in one direction (map to reduce). It is possible to use the output of a reduce step as the input to another MapReduce process. • As with functional programing, the input data are not changed. By applying the mapping and reduction functions to the input data, new data are produced. In effect, the original state of the Hadoop data lake is always preserved. • Because there is no dependency on how the mapping and reducing functions are applied to the data, the mapper and reducer data flow can be implemented in any number of ways to provide better performance. Distributed (parallel) implementations of MapReduce enable large amounts of data to be analyzed quickly. In general, the mapper process is fully scalable and can be applied to any subset of the input data. Results from multiple parallel mapping functions are then combined in the reducer phase.	

15	**Explain MapReduce Parallel Data Flow process.**	4
	From a programmer's perspective, the MapReduce algorithm is fairly simple. The programmer must provide a mapping function and a reducing function. Operationally, however, the Apache Hadoop parallel MapReduce data flow can be quite complex. Parallel execution of MapReduce requires other steps in addition to the mapper and reducer processes. The basic steps are as follows: **1. Input Splits.** As mentioned, HDFS distributes and replicates data over multiple servers. The default data chunk or block size is 64MB. Thus, a500MB file would be broken into 8 blocks and written to different machines in the cluster. The data are also replicated on multiple machines (typically three machines). These data slices are physical boundaries determined by HDFS and have nothing to do with the data in the file. Also, while not considered part of the MapReduce process, the time required to load and distribute data throughout HDFS servers can be considered part of the total processing time. The input splits used by MapReduce are logical boundaries based on the input data. For example, the split size can be based on the number of records in a file (if the data exist as records) or an actual size in bytes. Splits are almost always smaller than the HDFS block size. The number of splits corresponds to the number of mapping processes used in the map stage.	

2. Map Step. The mapping process is where the parallel nature of Hadoop comes into play. For large amounts of data, many mappers can be operating at the same time. The user provides the specific mapping process. MapReduce will try to execute the mapper on the machines where the block resides. Because the file is replicated in HDFS, the least busy node with the data will be chosen. If all nodes holding the data are too busy, MapReduce will try to pick a node that is closest to the node that hosts the data block (a characteristic called rack-awareness). The last choice is any node in the cluster that has access to HDFS.

3. Combiner Step. It is possible to provide an optimization or pre-reduction as part of the map stage where key–value pairs are combined prior to the next stage. The combiner stage is optional.

4. Shuffle Step. Before the parallel reduction stage can complete, all similar keys must be combined and counted by the same reducer process. Therefore, results of the map stage must be collected by key–value pairs and shuffled to the same reducer process. If only a single reducer process is used, the shuffle stage is not needed.

5. Reduce Step. The final step is the actual reduction. In this stage, the data reduction is performed as per the programmer's design. The reduce step is also optional. The results are written to HDFS. Each reducer will write an output file. For example, a MapReduce job running four reducers will create files called part-0000, part-0001, part-0002, and part-0003.

Figure 5.1 is an example of a simple Hadoop MapReduce data flow for a word count program. The map process counts the words in the split, and the reduce process calculates the total for each word. As mentioned earlier, the actual computation of the map and reduce stages are up to the programmer. The MapReduce data flow shown in Figure 5.1 is the same regardless of the specific map and reduce tasks.

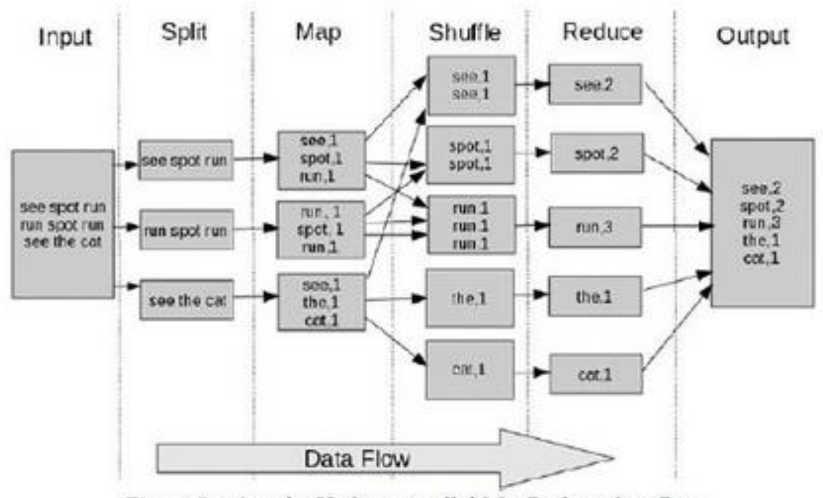


Figure 5.1 Apache Hadoop parallel MapReduce data flow

The input to the MapReduce application is the following file in HDFS with three lines of text. The goal is to count the number of times each word is used.

see spot run

run spot run

see the cat

The first thing MapReduce will do is create the data splits. For simplicity, each line will be one split. Since each split will require a map task, there are three mapper processes that count the number of words in the split. On a cluster, the results of each map task are written to local disk and not to HDFS. Next, similar keys need to be collected and sent to a reducer process. The shuffle step requires data movement and can be expensive in terms of processing time.

Depending on the nature of the application, the amount of data that must be shuffled throughout the cluster can vary from small to large.

Once the data have been collected and sorted by key, the reduction step can begin (even if only partial results are available). It is not necessary—and not normally recommended—to have a reducer for each key–value pair as shown in Figure 5.1.

In some cases, a single reducer will provide adequate performance;in other cases, multiple reducers may be required to speed up the reduce phase.The number of reducers is a tunable option for many applications. The final step is to write the output to HDFS.

As mentioned, a combiner step enables some pre-reduction of the map output data. For instance, in the previous example, one map produced the following.

counts:

(run,1)

(spot,1)

(run,1)

As shown in Figure 5.2, the count for run can be combined into (run,2)before the shuffle. This optimization can help minimize the amount of data transfer needed for the shuffle phase.

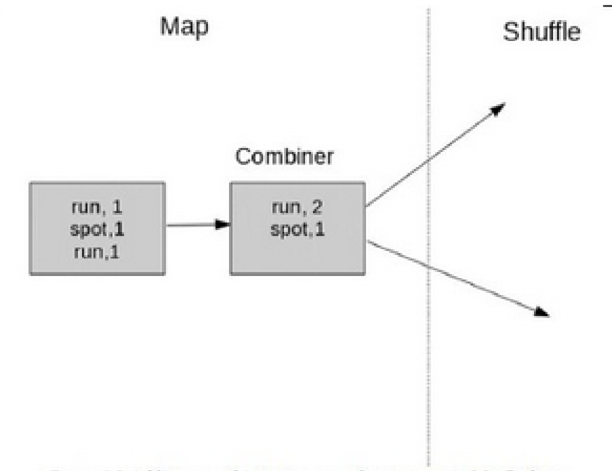
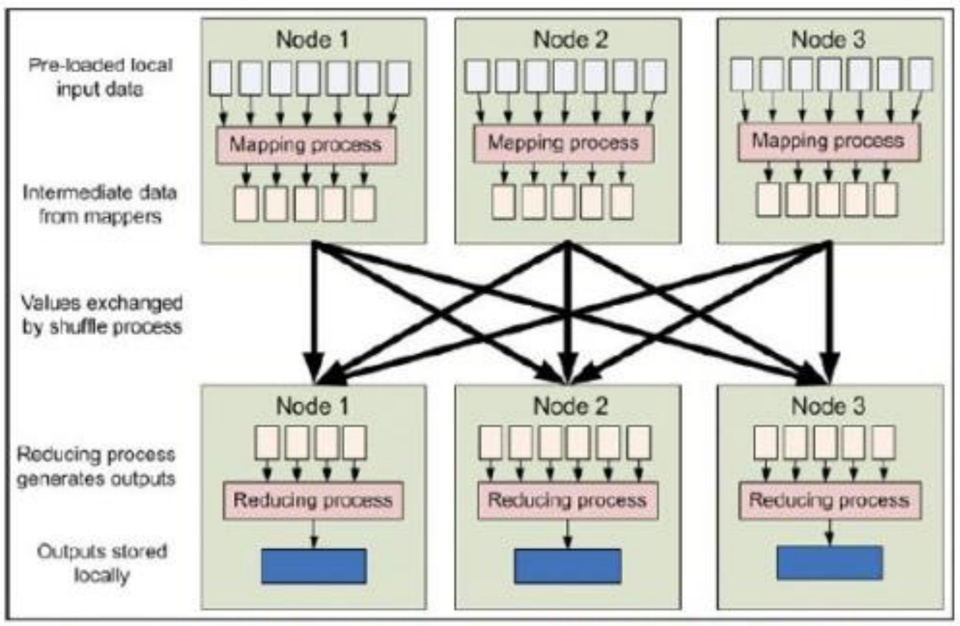


Figure 5.2 Adding a combiner process to the map step in MapReduce

16	**Explain** three-node MapReduce process.	**6**

Figure 5.3 shows a simple three-node MapReduce process. Once the mapping is complete, the same nodes begin the reduce process. The shuffle stage makes sure the necessary data are sent to each mapper. Also note that there is no requirement that all the mappers complete at the same time or that the mapper on a specific node be complete before a reducer is started. Reducers can be set to start shuffling based on a threshold of percentage of mappers that have finished.


Node 1
Node 2
Node 3
Pre-loaded local input data
Mapping process
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Reducing process
Outputs stored locally
Figure 5.3 Process placement during MapReduce (Adapted from Yahoo Hadoop Documentation)

	Finally, although the examples are simple in nature, the parallel MapReduce algorithm can be scaled up to extremely large data sizes. For instance, the Hadoop word count sample application can be run on the three lines given earlier or on a 3TB file. The application requires no changes to account for the scale of the problem—a feature that is one of the remarkable advantages of MapReduce processing.	
17	**Write a note on Hadoop MapReduce Hardware.**	**8**
	Hadoop MapReduce Hardware The capability of Hadoop MapReduce and HDFS to tolerate server—or even whole rack—failures can influence hardware designs. The use of commodity(typically x86_64) servers for Hadoop clusters has made low-cost, high availability implementations of Hadoop possible for many data centres. Indeed, the Apache Hadoop philosophy seems to assume servers will always fail and takes steps to keep failure from stopping application progress on a cluster. The use of server nodes for both storage (HDFS) and processing (mappers, reducers) is somewhat different from the traditional separation of these two tasks in the data centre. It is possible to build Hadoop systems and separate the roles(discrete storage and processing nodes). However, a majority of Hadoop systems use the general approach where servers enact both roles. Another interesting feature of dynamic MapReduce execution is the capability to tolerate dissimilar servers. That is, old and new hardware can be used together. Of course, large disparities in performance will limit the faster systems, but the dynamic nature of MapReduce execution will still work effectively on such systems.	
18	**Explain briefly the steps for compiling and running the program from the command line.**	**6**
	To compile and run the program from the command line, perform the following steps:	

	1. Make a local wordcount_classesdirectory. • $ mkdirwordcount_classes **2.** Compile the WordCount.java program using the 'hadoopclasspath' command to include all the available Hadoop class paths. • $ javac -cp \`hadoopclasspath\` -d wordcount_classes WordCount.java **3.** The jar file can be created using the following command: • $ jar -cvf wordcount.jar -C wordcount_classes/ **4.** To run the example, create an input directory in HDFS and place a text filein the new directory. • $ hdfsdfs -mkdir war-and-peace-input • $ hdfsdfs -put war-and-peace.txt war-and-peace-input **5.** Run the WordCount application using the following command: • $ hadoop jar wordcount.jar WordCount war-and-peace-input war-and peace-Output	
19	List and Explain the HDFS Components	**6**
	The design of HDFS is based on two types of nodes: a **NameNode and multiple DataNodes**. In a basic design, a single NameNode manages all the **metadata** needed to store and retrieve the actual data from the DataNodes. No data is actually storedon the NameNode. However. For a minimal Hadoop installation, there needs to be a single NameNode daemon and a single DataNode daemon running on at least one machine.	
20	**Explain how the pipe interfaces are used for mapper and reducer for an application**	
	Pipes is a library that allows C++ source code to be used for mapper and reducer code. Applications that require high performance when crunching numbers may achieve better throughput if written in C++ and used through the Pipes interface. Both key and value inputs to pipes programs are provided as STL strings(std::string). As shown in Listing 6.4, the program must define an instance of a mapper and an instance of a reducer. A program to use with Pipes is defined by writing classes extending Mapper and Reducer. Hadoop must then be informed as to which classes to use to run the job. The Pipes framework on each machine assigned to your job will start an instance of C++ program. Therefore, the executable must be placed inHDFS prior to use.	