BIG DATA ANALYTICS
MODULE 1

Introduction

- The Hadoop Distributed File System is the backbone of Hadoop MapReduce processing. New users and administrators often find HDFS different than most other UNIX/Linux file systems.

Hadoop Distributed File System Design Features

- 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.

Hadoop Distributed File System
Design Features

- The design of HDFS is based on the design of the Google File System (GFS).
- 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.

HDFS Components

- 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 stored on 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

HDFS Components

- The design is a master/slave architecture in which the master (NameNode) manages the file system namespace and regulates access to files by clients. File system namespace operations such as opening, closing, and
  renaming files and directories are all managed by the NameNode. The NameNode also determines the mapping of blocks to DataNodes and handles DataNode failures.
- The slaves (DataNodes) are responsible for serving read and write requests from the file system to the clients. The NameNode manages block creation, deletion, and replication.
- An example of the client/NameNode/DataNode interaction is provided in
- 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.

HDFS Components

- 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, how_ ever, 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.
- Reading data happens in a similar fashion. The client requests a file from the NameNode, which returns the best DataNodes from which to read the data. The client then accesses the data directly from the DataNodes.
- Thus, once the metadata has been delivered to the client, the NameNode steps back and lets the conversation between the client and the DataNodes proceed. While data transfer is progressing, the NameNode also monitors the DataNodes by listening for heartbeats sent from DataNodes. The lack of a heartbeat signal indicates a potential node failure. In such a case, the NameNode will route around the failed DataNode and begin re-replicating the now-missing blocks.
- Because the file system is redundant, DataNodes can be taken offline (decommissioned) for maintenance by informing the NameNode of the DataNodes to excluded from the HDFS pool.

HDFS Components

- The mappings between data blocks and the physical DataNodes are not kept in persistent storage on the NameNode. For performance reasons, the NameNode stores all metadata in memory. Upon startup, each DataNode provides a block report (which it keeps in persistent storage) to the NameNode. The block reports are sent every 10 heartbeats. (The interval between reports is a configurable property.) The reports enable the NameNode to keep an up-to-date account of all data blocks in the cluster.
- In almost all Hadoop deployments, there is a SecondaryNameNode. While not explicitly required by a NameNode, it is highly recommended. The term "SecondaryNameNode" (now called CheckPointNode) is somewhat misleading. It is not an active failover node and cannot replace the primary NameNode in case of its failure.

HDFS Components

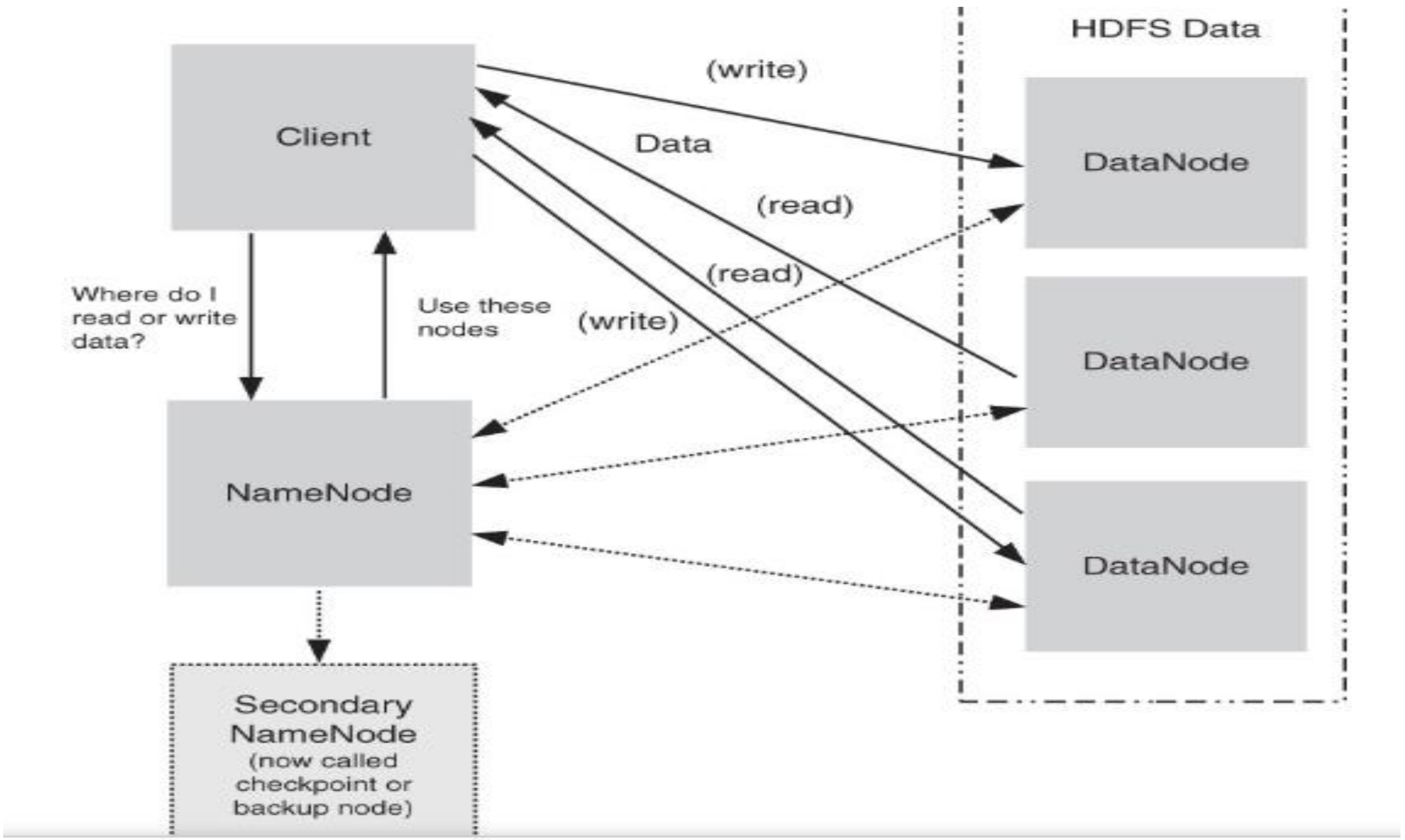
- The purpose of the SecondaryNameNode is to perform periodic checkpoints that evaluate the status of the NameNode. Recall that the NameNode keeps all system metadata memory for fast access. It also has two disk files that track changes to the metadata:
  - – An image of the file system state when the NameNode was started. This file begins with fsimage_* and is used only at startup by the NameNode.
  - – A series of modifications done to the file system after starting the NameNOde• These files begin with edit_* and reflect the changes made after the file was read.
- The location of these files is set by the dfs . namenode . name . dir property in the hdfs-site.xrnl file.

HDFS Components

- The SecondaryNameNode periodically downloads fgimage and edits files, joins them into a new fsimage, and uploads the new fgimage file to the NameNode. Thus, when the NatneNode restarts, the fsimage file is reasonably up-to-date and requires only the edit logs to be applied since

  the last checkpoint. If the SecondaryNameNode were not running, a restart of the NatneNode could take a prohibitively long time due to the nutnber of changes to the file system.
- Thus, the various roles in HDFS can be sutnmarized as follows:
  - HDFS uses a tnaster/slave model designed for large file reading/streaming.
  - The NatneNode 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.



HDFS Components

HDFS Block Replication

- As mentioned, 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 hdfs dfs-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 4K B or 8KB. The HDFS default block size is not the minimum block size, however. If a 20K B file is written to HDFS, it will create a block that is approximately 20KB in size. (The underlying file systetn 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 Block Replication

- HDFS blocks 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. lil 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 subsequently re-replicated on other DataNodes.

HDFS Safe Mode

- When the NameNode starts, it enters a read-only safe mode where blocks cannot be replicated or deleted. Safe Mode enables the NameNode to perform two important processes:
  - The previous file system state is reconstructed by loading the fsimage file into memory and replaying the edit log.
  - The mapping between blocks and data nodes is created by waiting for enough Of the DataNodes to register so that at least one copy of the data is available. Not all DataNodes are required to register before HDFS exits from Safe Mode. The registration process may continue for some time.
- HDFS may also enter Safe Mode for maintenance using the hdfs dfsadmin-safemode command or when there is a file system issue that must be addressed by the administrator.

Rack Awareness

- Rack awareness deals with data locality. Recall that one of the main design goals of Hadoop MapReduce is to move the computation to the data. Assuming that most data center networks do not offer full bisection bandwidth, a typical Hadoop cluster will exhibit three levels of data locality:
  - Data resides on the local machine (best).
  - Data resides in the same rack (better).
  - Data resides in a different rack (good).

Rack Awareness

- When the YARN scheduler is assigning MapReduce containers to work as mappers, it will try to place the container first on the local machine, then on the same rack, and finally on another rack.
- In addition, the NameNode tries to place replicated data blocks on multiple racks for improved fault tolerance. In such a case, an entire rack failure will not cause data loss or stop HDFS from working. Performance may be degraded, however.
- HDFS can be made rack-aware by using a user-derived script that enables the master node to map the network topology of the cluster. A default Hadoop installation assumes all the nodes belong to the same (large) rack. In that case, there is no option 3

NameNode High Availability

- With early Hadoop installations, the NameNode was a single pointof failure that could bring down the entire Hadoop cluster. NameNode hardware often employed redundant power supplies and storage to guard against such problems, but it was still susceptible to other failures. The solution was to implement NameNode High
- Availability (HA) as a means to provide true failover service.
- As shown in Figure 3.3, an HA Hadoop cluster has two (or more) separate
- NameNode machines. Each machine is configured with exactly the same software.

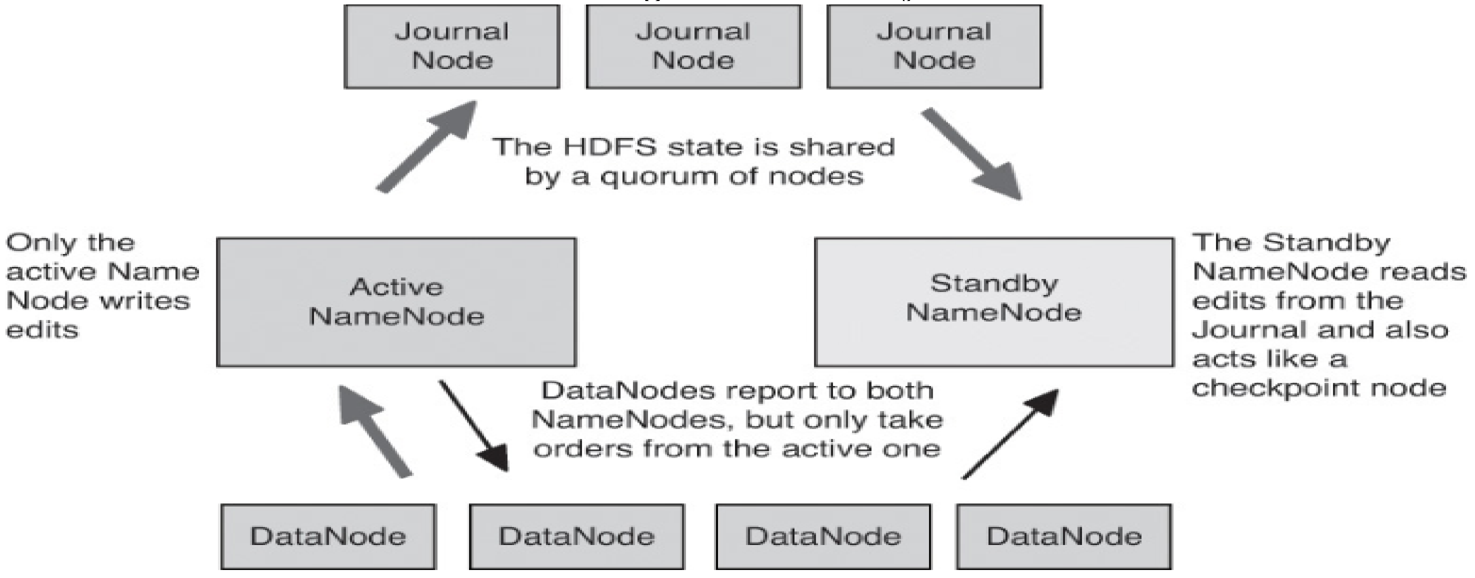

Journal Node
Journal Node
Journal Node
The HDFS state is shared by a quorum of nodes
Only the active Name Node writes edits
Active NameNode
Standby NameNode
The Standby NameNode reads edits from the Journal and also acts like a checkpoint node
DataNodes report to both NameNodes, but only take orders from the active one
DataNode
DataNode
DataNode
DataNode

Figure 3.3 HDFS High Availability design

NameNode High Availability

- One of the NameNode machines is in the Active state, and the other is in the Standby state. Like a single NameNode cluster, the Active NameNode is responsible for all client HDFS operations in the cluster. The Standby NameNode maintains enough state to provide a fast failover (if required).
- To guarantee the file system state is preserved, both the Active and Standby NameNodes receive block reports from the DataNodes. The Active node also sends all file system edits to a quorum ofJournal nodes. At least three physically separate JournalNode daemons are required, because edit log modifications must be written to a majority of the JournalNodes. This design will enable the system to tolerate the failure of a single JournalNode machine. The Standby node continuously reads the edits from the JournalNodes to ensure its namespace is synchronized with that of the Active node. In the event of an Active NameNode failure, the Standby node reads all remaining edits from the JournalNodes before promoting itself to the Active state.

NameNode High Availability

- To prevent confusion between NameNodes, the JournalNodes allow only one NameNode to be a writer at a time. During failover, the NameNode that is chosen to become active takes over the role of writing to the JournalNodes. A SecondaryNameNode is not required in the HA configuration because the Standby node also performs the tasks of the Secondary NameNode.
- Apache Zookeeper is used to monitor the NameNode health. Zookeeper is a highly available service for maintaining small amounts of coordination data, notifying clients of changes in that data, and monitoring clients for failures. HDFS failover relies on ZooKeeper for failure detection and for Standby to Active NameNode election. The Zookeeper components are not depicted in Figure 3.3.

HDFS NameNode Federation

- Another important feature of HDFS is NameNode Federation. Older versions of HDFS provided a single namespace for the entire cluster managed by a single NameNode. Thus, the resources of a single NameNode determined the size of the namespace. Federation addresses this limitation by adding support for multiple
- NameNodes/namespaces to the HDFS file system. The key benefits are as 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 namespaces.

HDFS NameNode Federation

- Figure 3.4 illustrates how HDFS NameNode Federation is accomplished. NameNode1 manages the /research and /marketing namespaces, and NameNode2 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.

HDFS NameNode Federation

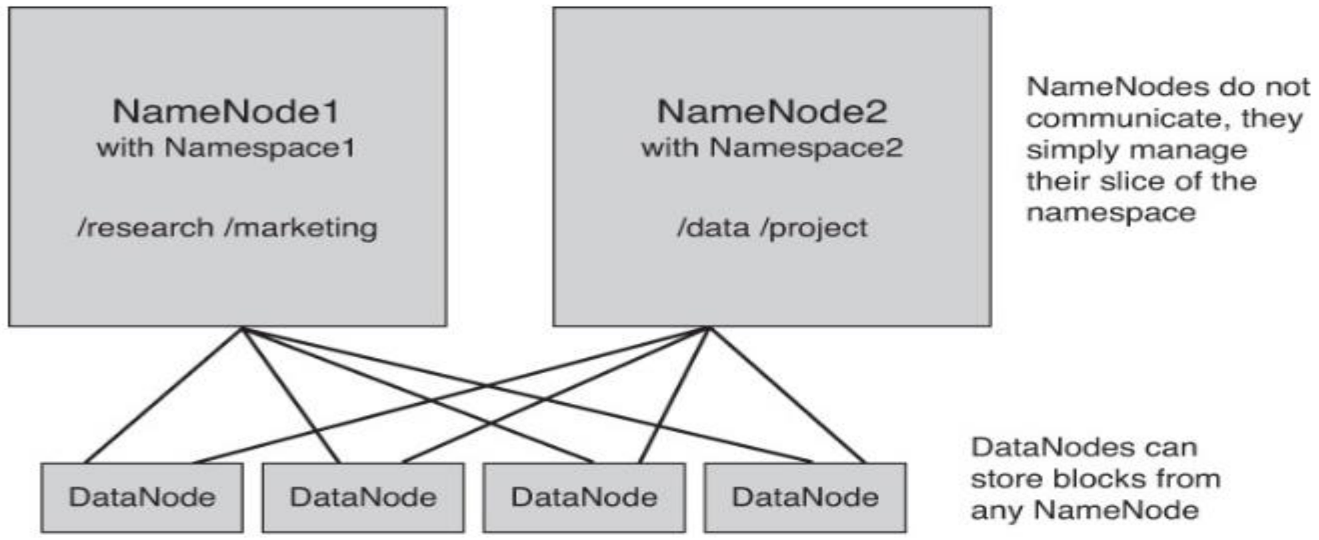

NameNode1
with Namespace1
/research /marketing
NameNode2
with Namespace2
/data /project
NameNodes do not communicate, they simply manage their slice of the namespace
DataNode
DataNode
DataNode
DataNode
DataNodes can store blocks from any NameNode

Figure 3.4 HDFS NameNode Federation example

HDFS Checkpoints and Backups

- As mentioned earlier, the NameNode stores the metadata of the HDFS file system in a file called fsimage. File systems modifications are written to an edits log file, and at startup the NameNode merges the edits into a new fsimage. The SecondaryNameNode or CheckpointNode periodioally fetches edits from the NameNode, merges them, and returns an updated fsimage to the NameNode.

- 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 fsimage 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. No CheckpointNodes may be registered if a Backup node is in use.

HDFS Snapshots

- 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 file 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 it appears to the user that there are duplicate files.

  – Snapshots do not adversely affect regular HDFS operations.

HDFS User Commands

- hdfs [--config confdir--] COMMAND
- Hdfs version
  - Hadoop 2.6.0.2.2.4.3-2
- hdfs dfs –ls /
  - Lists files in the root HDFS directory
- hdfs dfs –ls OR hdfs dfs –ls /user/hdfs
  - Lists the files in user home directory
- Hdfs dfs –mkdir stuff
  - Create directory
- hdfs dfs –put test stuff
  - Copy files to HDFS
- hdfs dfs –get stuff/test test-local
  - Copy files from HDFS
- hdfs dfs –cp stuff/test test.hdfs
  - Copy files within HDFS
- hdfs dfs –rm test.hdfs
  - Delete files within HDFS
- hdfs dfs –rm –r –skipTrash stuff
  - Delete directory in HDFS

MapReduce Model

- The MapReduce computation model provides a very powerful tool for many applications and i.s more common than most users realize. Its underlving 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 “Ram" 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 and you sum the total as your friends tell you their counts.

MapReduce Parallel Data Flow

1. Input Splits. As mentioned, HDFS distributes and replicates data over multiple servers. The default data chunk or block size is 64MB. Thus, a 500MB 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).
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

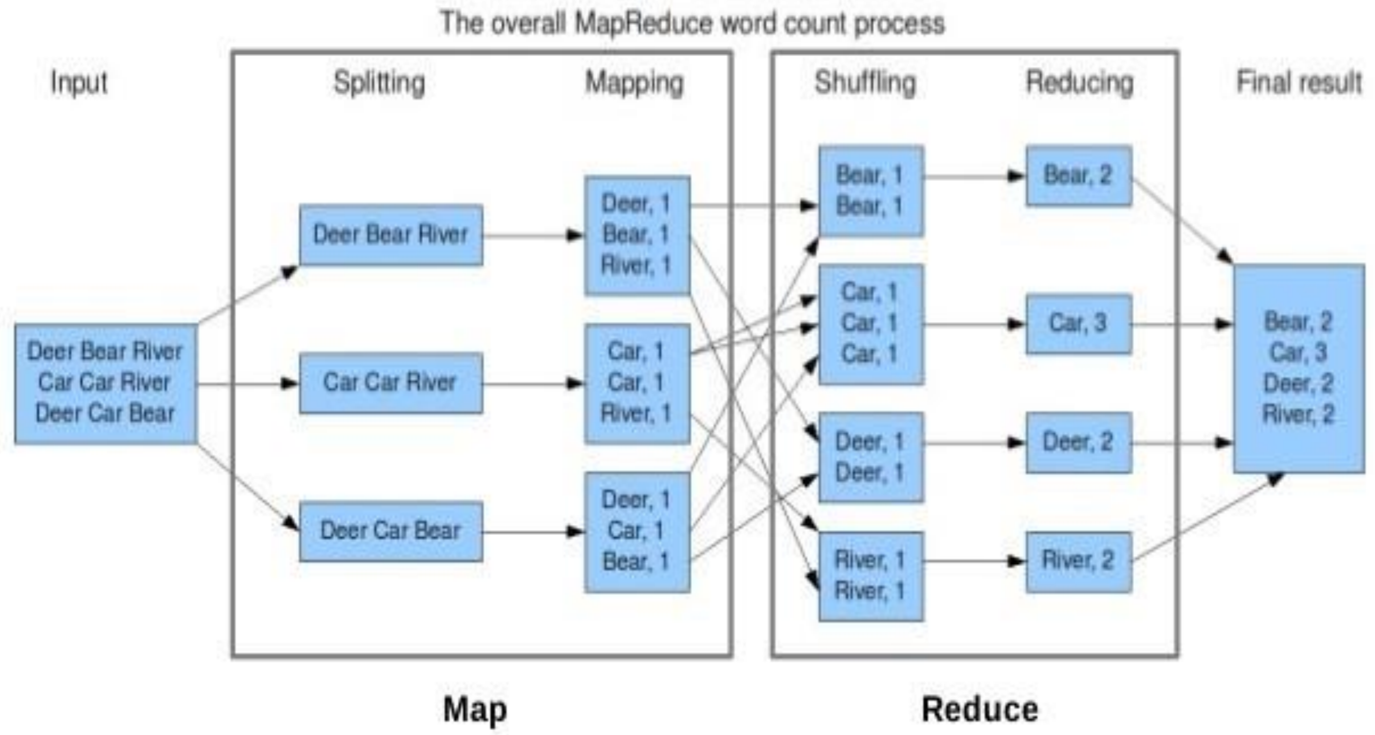
   node that is closest to the node that hosts the data block (a characteristic called rack awareness). The last choice is any node in the cluster that has access to HDFS.

MapReduce Parallel Data Flow

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, aJJ 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.



The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2
Map
Reduce

Mapper Script

```
#!/bin/bash
While read line ;
do
    for token in $line;
    do
         if [“$token” = “Ram”];
         then
             echo “Ram, 1”
         elif [“$token” = “Sita”];
         then
             echo “Sita, 1”
         fi
       done
done
```

Reducer Script

```
#!/bin/bash
Rcount=0
Scount=0
While read line ;
do
    if [ $line =“Ram, 1”];
        then
                Rcount = Rcount+1
        elif [ $line =“Sita, 1”];
        then
               Scount = Scount+1
                fi
           Done
echo “Ram, $Rcount”
echo “Sita, $Scount”
```

To compile and run the program from the comtnand line, perform the following steps:

1. Make a local wordcount_classes directory. $ mkdir
   wordcount—classes
2. Compile the WordCount program using the 'hadoop classpath’ command to include all the available Hadoop class paths.
   $ javac -cp ʾhadoop classpath' -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 file in the new directory. For this example, we will use the war-and-peace. txt file (available from the book download page; see Appendix A):
   $ hdfs dfs -mkdir /Demo
   $ hdfs dfs -put input. txt /Demo
5. Run the WordCount application using the following command:
   $ hadoop jar wordcount.jar WordCount /Demo/input /output

Debugging MapReduce

- The best advice for debugging parallel MapReduce applications is this: Don't.
- The key word here is parallel. Debugging on a distributed system is hard and should be avoided.
- The best approach is to make sure applications run on a simpler system (i.e., the pseudo-distributed single-machine install) with smaller data sets.
- When investigating program behavior at scale, the best approach is to use the application logs to inspect the actual MapReduce progress - The time-tested debug print statements are also visible in the logs.

Listing, Killing, and Job Status

- The jobs can be managed using

  the mapred job command. The most import options are —list, -kill, and -status.

In addition, the yarn application command can be used to control all applications running on the cluster

Hadoop Log Management

- The MapReduce logs provide a comprehensive listing of both mappers and reducers.
- The actual log output consists of three files—stdout, stderr, and syslog (Hadoop system messages)—for the application.
- There are two modes for log storage. The first (and best) method is to use log aggregation.
- In this mode, logs are aggregated in HDFS and can be displayed in the YARN ResourceManager user interface or examined with the yarn logs command.
- If log aggregation is not enabled, the logs will be placed locally on the cluster nodes on which the mapper or reducer ran. The location of the unaggregated local logs is given by the **yarn. nodemanager. log-dirs property in the yarn-site.xml** file

Enabling YARN Log Aggregation

- To manually enable log aggregation, follows these steps:
- As the HDFS superuser administrator (usually user hdfs), create the following directory in HDFS:

```
$ hdfs dfs -mkdir -p /yarn/logs

$ hdfs dfs -chown -R yarn:hadoop /yarn/logs $ hdfs dfs -
chmod -R g+rw /yarn/logs
```

- Add the following properties in the yarn-site.xml and restart all YARN services on all nodes.