MODULE 1
HDFS BASICS, RUNNING EXAMPLE PROGRAMS AND BENCHMARKS, HADOOP MAPREDUCE FRAMEWORK, MAPREDUCE PROGRAMMING

Hadoop Distributed File System Basics

- HADOOP DISTRIBUTED FILE SYSTEM DESIGN FEATURES
- HDFS COMPONENTS:
  - ✓ HDFS Block Replication
  - ✓ HDFS Safe Mode
  - ✓ Rack Awareness
  - ✓ NameNode High Availability
  - ✓ HDFS NameNode Federation
  - ✓ HDFS Checkpoints and Backups
  - ✓ HDFS Snapshots
  - ✓ HDFS NFS Gateway
- HDFS USER COMMANDS

- Google File System (GFS)
- Hadoop Distributed File System (HDFS)
- HDFS block size is typically 64MB or 128MB

Important aspects of HDFS:

- ✓ 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
- ✓ NameNode manages all the metadata needed to store and retrieve the actual data from the DataNodes.
- ✓ No data is actually stored on the NameNode.

- ✓ 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 NameNode manages block creation, deletion, and replication
- ✓ The slaves (DataNodes) are responsible for serving read and write
- ✓ requests from the file system to the clients

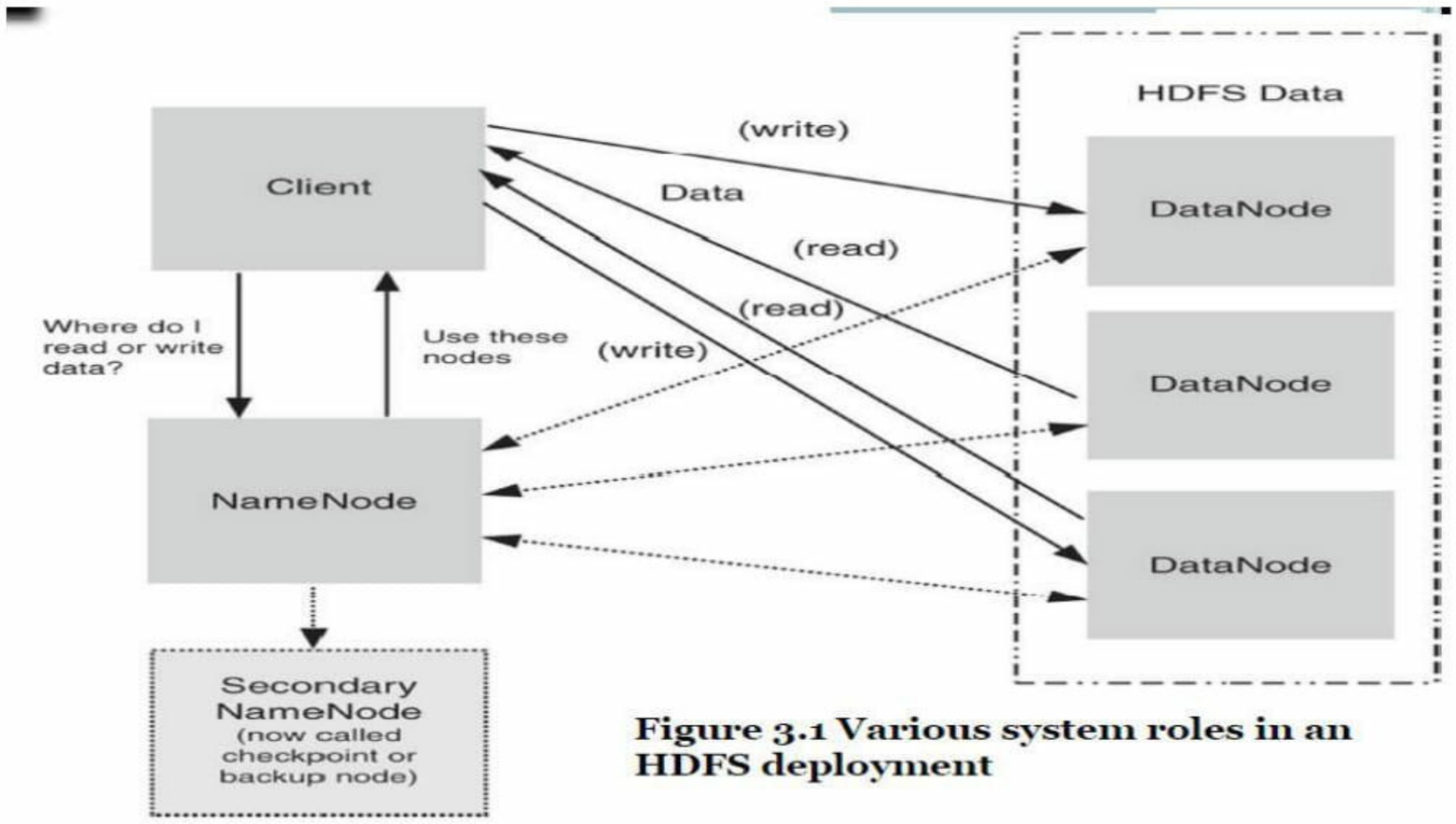


Figure 3.1 Various system roles in an HDFS deployment

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:

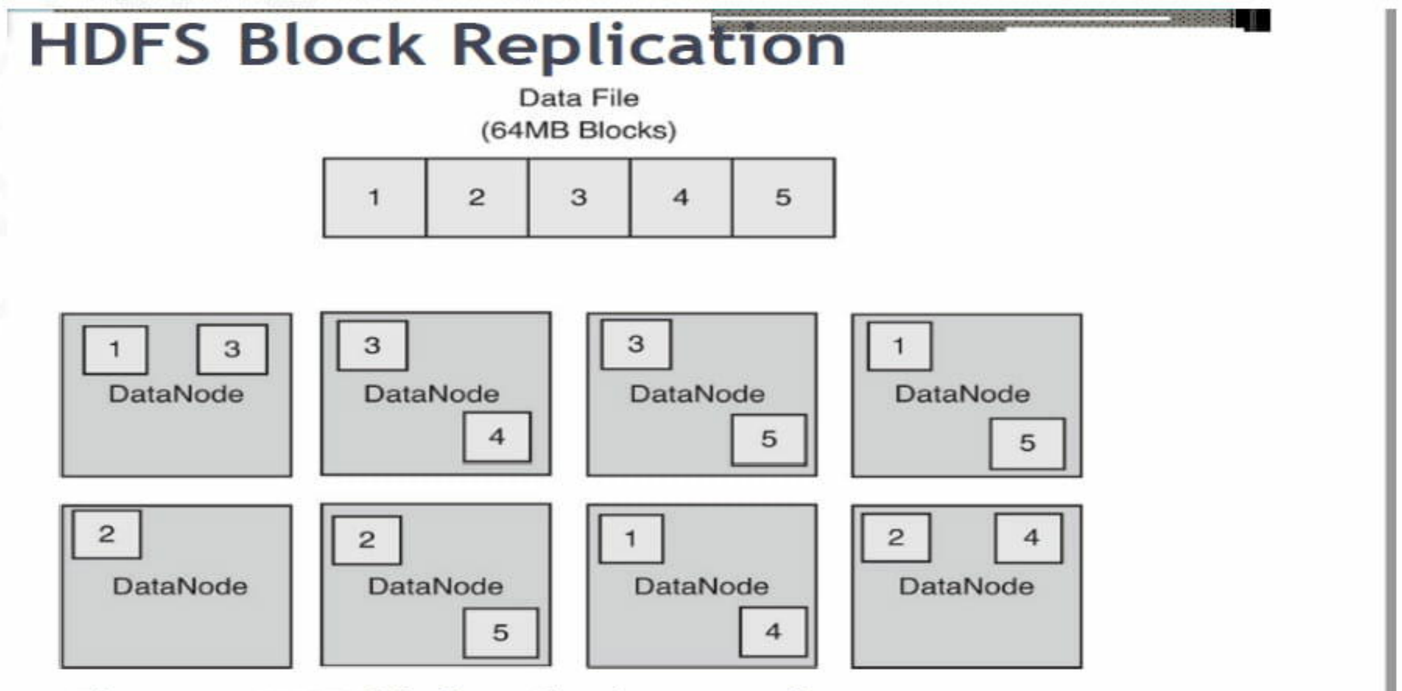


Figure 3.2 HDFS block replication example

1. The previous file system state is reconstructed by loading the fsimage file into memory and replaying the edit log.
2. 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.

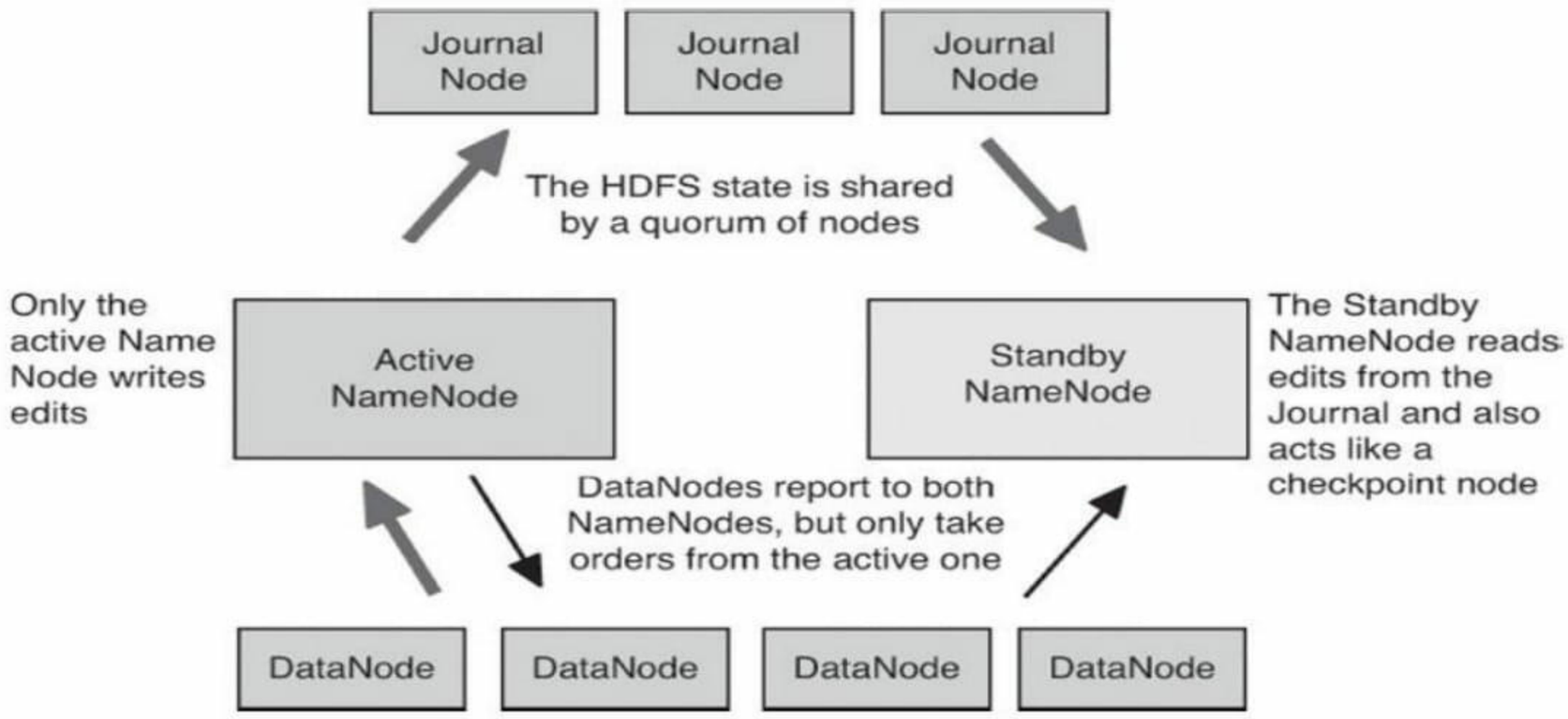
Rack Awareness

- Rack awareness deals with data locality.
- 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:
  1. Data resides on the local machine (best).
  2. Data resides in the same rack (better).
  3. Data resides in a different rack (good).
- 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

- The NameNode was a single point of 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) separatNameNode machines.
- Each machine is configured with exactly the same software.
- 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).

Figure 3.3 HDFS High Availability design



HDFS Checkpoints and Backups

- *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 Secondary NameNode or CheckpointNode periodically 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:
  1. Snapshots can be taken of a sub-tree of the file system or the entire file system.
  2. Snapshots can be used for data backup, protection against user errors, and disaster recovery.
  3. Snapshot creation is instantaneous.
  4. 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.
  5. Snapshots do not adversely affect regular HDFS operations.

HDFS NFS Gateway

- The HDFS NFS Gateway supports NFSv3 and enables HDFS to be mounted as part of the client's local file system.
- Users can browse the HDFS file system through their local file systems that provide an NFSv3 client compatible operating system. This feature offers users the following capabilities:
    - Users can easily download/upload files from/to the HDFS file system 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.

HDFS commands

- Syntax
- **hdfs [--config confdir] COMMAND**
- where COMMAND is one of:
    1. **dfs** :run a file system command on the file systems supported in Hadoop.
    2. **namenode –format:** format the DFS file system
    3. **secondarynamenode** : run the DFS secondary namenode
    4. **namenode:** run the DFS namenode
    5. **journalnode:** run the DFS journalnode
    6. **zkfc**: run the ZK Failover Controller daemon
    7. **datanode** : run a DFS datanode
    8. **dfsadmin:** run a DFS admin client
    9. **haadmin** : run a DFS HA admin client
    10. **fsck:** run a DFS file system checking utility
    11. **balancer** : run a cluster balancing utility
    12. **jmxget :** get JMX exported values from NameNode or DataNode.
    13. **mover:** run a utility to move block replicas across storage types
    14. **oiv:** apply the *offline fsimage viewer* to an fsimage
    15. **oiv_legacy :** apply the *offline fsimage viewer* to an legacy fsimage
    16. **oev** : apply the *offline edits viewer* to an edits file
    17. **fetchdt:** fetch a delegation token from the NameNode
    18. **getconf** : get config values from configuration
    19. **groups** : get the groups which users belong to
    20. **snapshotDiff** : diff two snapshots of a directory or diff the current directory contents with a snapshot
    21. **lsSnapshottableDir** : list all snapshottable dirs owned by the current user Use -help to see options
    22. **portmap :** run a portmap service
    23. **nfs3 :** run an NFS version 3 gateway
    24. **cacheadmin** : configure the HDFS cache
    25. **crypto** : configure HDFS encryption zones

26. **storagepolicies :** get all the existing block storage policies
27. **version :** print the version

General HDFS Commands

• **hdfs version**

• **hdfs dfs**

Generic options supported are

- -conf <configuration file> specify an application configuration file
- -D <property=value> use value for given property
- -fs <local|namenode:port> specify a namenode
- -jt <local|resourcemanager:port> specify a ResourceManager
- -files <comma separated list of files> specify comma separated
- files to be copied to the map reduce cluster
- -libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
- -archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute machines.

List Files in HDFS

To list the files in the root HDFS directory

$ hdfs dfs -ls /

To list files in your home directory

$ hdfs dfs -ls

The same result can be obtained by issuing the following command

$ hdfs dfs -ls /user/hdfs

Make a Directory in HDFS

To make a directory in HDFS

$ hdfs dfs -mkdir stuff

Copy Files to HDFS

To copy a file from your current local directory into HDFS

$ hdfs dfs -put test stuff

Copy Files from HDFS

Files can be copied back to your local file system

$ hdfs dfs -get stuff/test test-local

Copy Files within HDFS

The following command will copy a file in HDFS

$ hdfs dfs -cp stuff/test test.hdfs

Delete a File within HDFS

To delete the HDFS file test.dhfs that was created previously

$ hdfs dfs -rm test.hdfs

Moved:'hdfs://limulus:8020/user/hdfs/stuff/test'totrashat: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 - skipTrash option.

$ hdfs dfs -rm -skipTrash stuff/test

Deleted stuff/test

Delete a Directory in HDFS

To delete the HDFS directory stuff and all its contents

$ hdfs dfs -rm -r -skipTrash stuff

Deleted stuff

Get an HDFS Status Report

users can get an abbreviated HDFS status report using the following command

$ hdfs dfsadmin -report

Hadoop MapReduce Framework

THE MAPREDUCE MODEL

- The MapReduce computation model provides a very powerful tool for many applications and is more common than most users realize.
- There are two stages: a mapping stage and a reducing stage.
- 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:
  1. 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.
  2. 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.
- Distributed (parallel) implementations of MapReduce enable large amounts of data to be analyzed quickly.
- 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.
- Hadoop accomplishes parallelism by using a distributed file system (HDFS) to slice and spread data over multiple servers.
- MapReduce will try to move the mapping tasks to the server that contains the data slice. Results from each data slice are then combined in the reducer step.

MAPREDUCE PARALLEL DATA FLOW

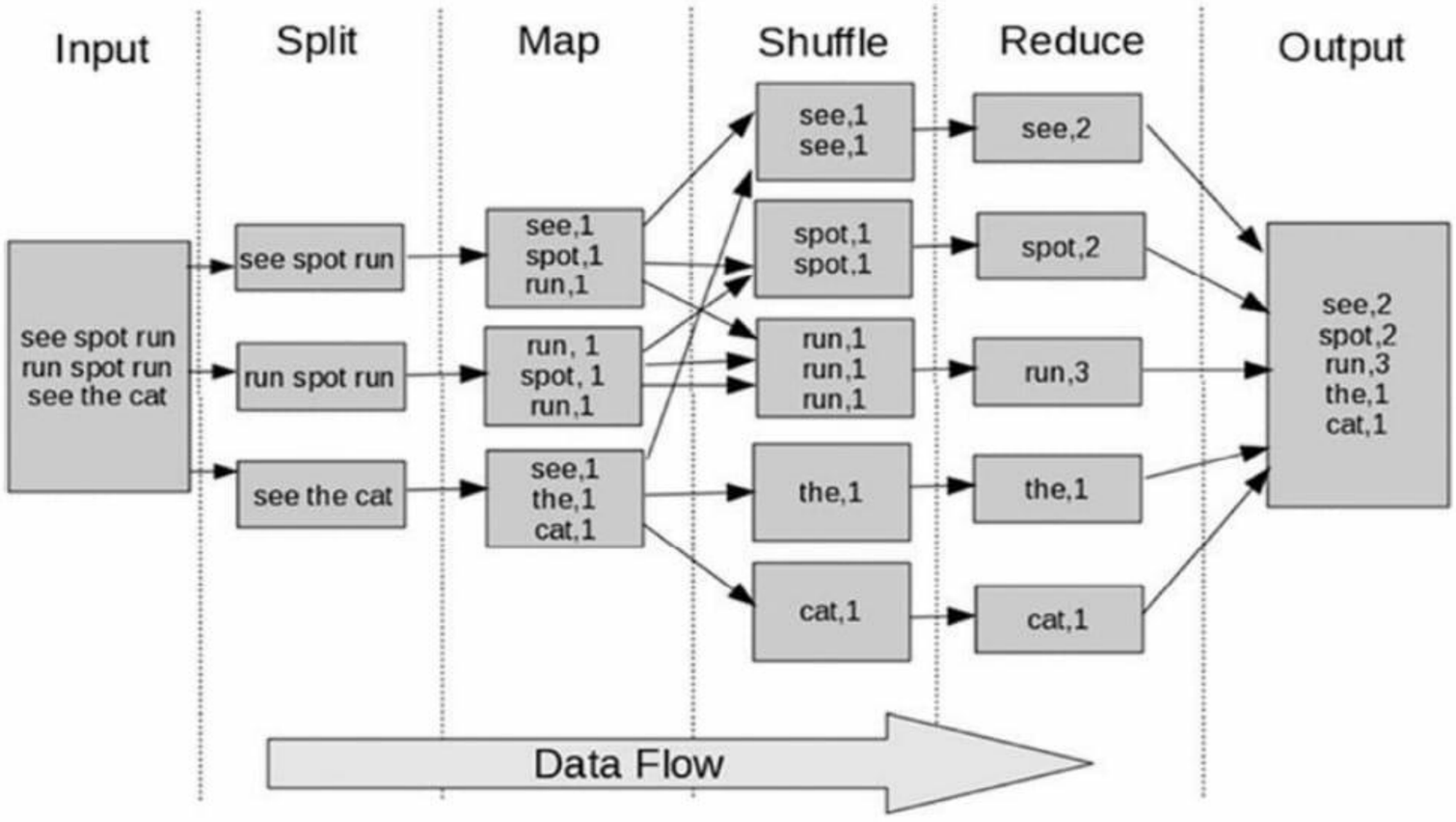
Parallel execution of MapReduce requires other steps in addition to the mapper and reducer processes. The basic steps are as follows:

1. **Input Splits.**
   - 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).
   - The input splits used by MapReduce are logical boundaries based on the input data.
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 prereduction 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

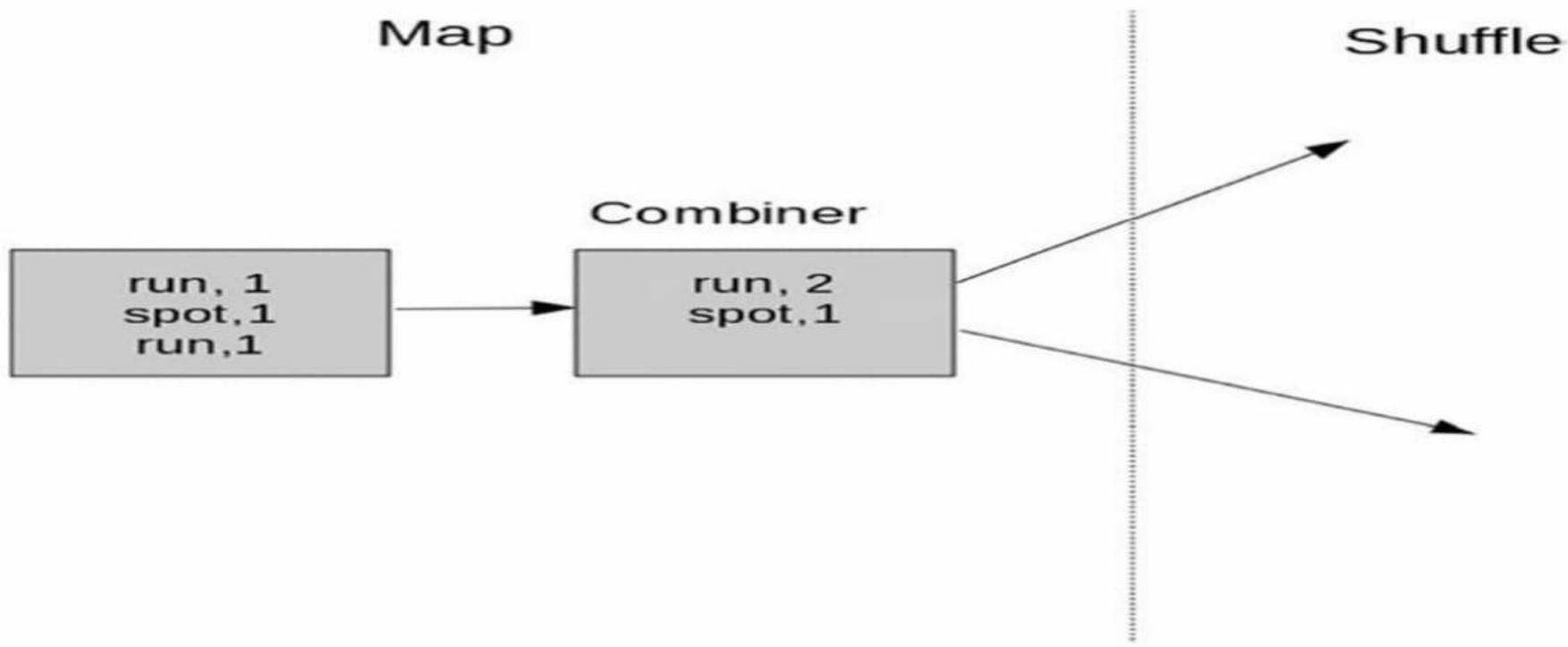
- The **first** thing MapReduce will do is create the data splits.
- For simplicity, each line will be one split.
- Since each split will require a map task, there are three ***mapper processes*** that count the number of words in the split. On a cluster, the results of each map task are written to local 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***.
- 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.
- 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

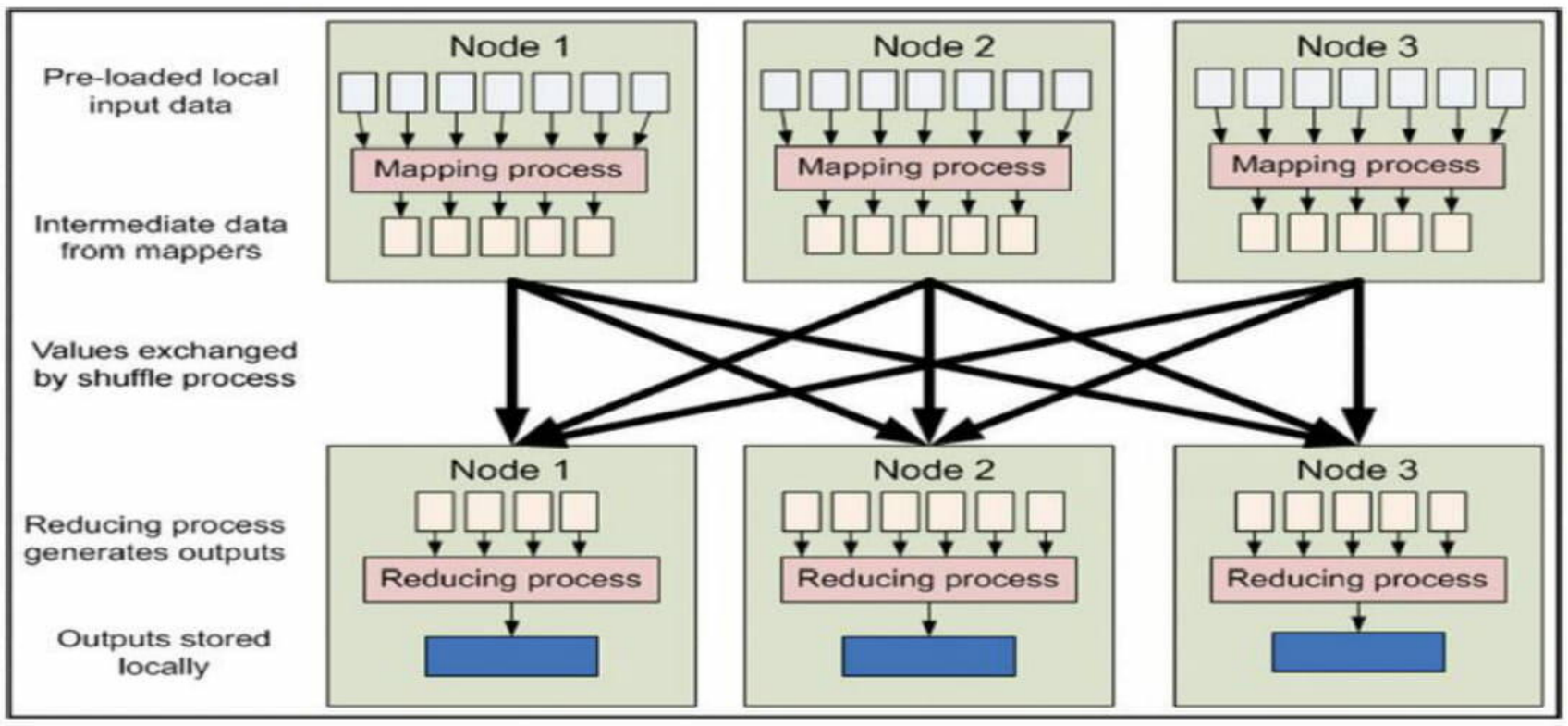


- The Hadoop YARN resource manager and the MapReduce framework determine the actual placement of mappers and reducers.
- The MapReduce framework will try to place the map task as close to the data as possible.
- It will request the placement from the YARN scheduler but may not get the best placement due to the load on the cluster.
- In general, nodes can run both mapper and reducer tasks.
- Indeed, the dynamic nature of YARN enables the work containers used by completed map tasks to be returned to the pool of available resources.

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.

Figure 5.3 Process placement during MapReduce (Adapted from Yahoo Hadoop Documentation)



FAULT TOLERANCE AND SPECULATIVE EXECUTION

- One of the most interesting aspects of parallel MapReduce operation is the strict control of data flow throughout the execution of the program.
  For example, mapper processes do not exchange data with other mapper processes, and data can only go from mappers to reducers—not the other direction.
- The confined data flow enables MapReduce to operate in a fault-tolerant fashion.
- The design of MapReduce makes it possible to easily recover from the failure of one or many map processes.
  For example, should a server fail, the map tasks that were running on that machine could easily be restarted on another working server because there is no dependence on any other map task.
- In a similar fashion, failed reducers can be restarted.
- If reduce tasks remain to be completed on a down node, the MapReduce ApplicationMaster will need to restart the reducer tasks.
- If the mapper output is not available for the newly restarted reducer, then these map tasks will need to be restarted.
- This process is totally transparent to the user and provides a fault-tolerant system to run applications.

Speculative Execution

- One of the challenges with many large clusters is the inability to predict or manage unexpected system bottlenecks or failures. this problem represents a difficult challenge for large systems.
- Thus, it is possible that a congested network, slow disk controller, failing disk, high processor load, or some other similar problem might lead to slow performance without anyone noticing.

- When one part of a MapReduce process runs slowly, it ultimately slows down everything else because the application cannot complete until all processes are finished.
- The nature of the parallel MapReduce model provides an interesting solution to this problem. By starting a copy of a running map process without disturbing any other running mapper processes.

For example, suppose that as most of the map tasks are coming to a close, the ApplicationMaster notices that some are still running and schedules redundant copies of the remaining jobs on less busy or free servers. *Should the secondary processes finish first, the other first processes are then terminated (or vice versa). This process is known as* ***speculative execution.***

The same approach can be applied to reducer processes that seem to be taking a long time. Speculative execution can reduce cluster efficiency because redundant resources are assigned to applications that seem to have a slow spot.

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 centers.
- 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 center.
- 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.

Running MapReduce Examples

All Hadoop releases come with MapReduce example applications. Running the existing MapReduce examples is a simple process once the example files are located, that is.

For example, if you installed Hadoop version 2.6.0 from the Apache sources under /opt, the examples will be in the following directory:

```
/opt/hadoop-2.6.0/share/hadoop/mapreduce/
```

In other versions, the examples may be in

```
/usr/lib/hadoopmapreduce/
```

or some other location. The exact location of the example jar file can be found using the find command:

```
$ find / -name "hadoop-mapreduce-examples*.jar" -print
```

The environment variable called HADOOP_EXAMPLES can be defined as follows:

```
$ export HADOOP_EXAMPLES=/usr/hdp/2.2.4.2-2/hadoop-mapreduce
```

Listing Available Examples

A list of the available examples can be found by running the following command. In some cases, the version number may be part of the jar file.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar
```

Running the Pi Example

The pi example calculates the digits of π using a quasi-Monte Carlo method. If you have not added users to HDFS, run these tests as user hdfs. To run the pi example with 16 maps and 1,000,000 samples per map, enter the following command:

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar pi 16
1000000
```

Running Basic Hadoop Benchmarks

Many Hadoop benchmarks can provide insight into cluster performance. The best benchmarks are always those that reflect real application performance.

Running the Terasort Test

The terasort benchmark sorts a specified amount of randomly generated data. This benchmark provides combined testing of the HDFS and MapReduce layers of a Hadoop cluster. A full terasort benchmark run consists of the following three steps:

1. Generating the input data via teragen program.
2. Running the actual terasort benchmark on the input data.
3. Validating the sorted output data via the teravalidate program.

1. Run teragen to generate rows of random data to sort.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar teragen 500000000
/user/hdfs/TeraGen-50GB
```

2. Run terasort to sort the database.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar terasort
/user/hdfs/TeraGen-50GB /user/hdfs/TeraSort-50GB
```

3. Run teravalidate to validate the sort.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar teravalidate
/user/hdfs/TeraSort-50GB /user/hdfs/TeraValid-50GB
```

For example, the following command will instruct terasort to use four reducer tasks:

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-examples.jar terasort -
Dmapred.reduce.tasks=4 /user/hdfs/TeraGen-50GB /user/hdfs/TeraSort-50GB
```

Also, do not forget to clean up the terasort data between runs (and after testing is finished). The following command will perform the cleanup for the previous example:

```
$ hdfs dfs -rm -r -skipTrash Tera*
```

Running the TestDFSIO Benchmark

The steps to run TestDFSIO are as follows:

1. Run TestDFSIO in write mode and create data.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-client-jobclienttests.
jar TestDFSIO -write -nrFiles 16 -fileSize 1000
```

Example results are as follows (date and time prefix removed).

```
fs.TestDFSIO: ----- TestDFSIO ----- : write
fs.TestDFSIO: Date & time: Thu May 14 10:39:33 EDT 2015
fs.TestDFSIO: Number of files: 16
fs.TestDFSIO: Total MBytes processed: 16000.0
fs.TestDFSIO: Throughput mb/sec: 14.890106361891005
fs.TestDFSIO: Average IO rate mb/sec: 15.690713882446289
fs.TestDFSIO: IO rate std deviation: 4.0227035201665595
fs.TestDFSIO: Test exec time sec: 105.631
```

2. Run TestDFSIO in read mode.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-client-jobclienttests.
jar TestDFSIO -read -nrFiles 16 -fileSize 1000
```

Example results are as follows (date and time prefix removed). The large standard deviation is due to the placement of tasks in the cluster on a small four-node cluster.

```
fs.TestDFSIO: ----- TestDFSIO ----- : read
fs.TestDFSIO: Date & time: Thu May 14 10:44:09 EDT 2015
fs.TestDFSIO: Number of files: 16
fs.TestDFSIO: Total MBytes processed: 16000.0
fs.TestDFSIO: Throughput mb/sec: 32.38643494172466
fs.TestDFSIO: Average IO rate mb/sec: 58.72880554199219
fs.TestDFSIO: IO rate std deviation: 64.60017624360337
```

```
fs.TestDFSIO: Test exec time sec: 62.798
```

3. Clean up the TestDFSIO data.

```
$ yarn jar $HADOOP_EXAMPLES/hadoop-mapreduce-client-jobclienttests.
jar TestDFSIO -clean
```

Managing Hadoop MapReduce Jobs

Hadoop MapReduce jobs can be managed using the mapred job command. The most important options for this command in terms of the examples and benchmarks are -list, -kill, and -status. In particular, if you need to kill one of the examples or benchmarks, you can use the mapred job -list command to find the job-id and then use mapred job -kill <jobid> to kill the job across the cluster. MapReduce jobs can also be controlled at the application level with the yarn application command. The possible options for mapred job are as follows:

```
$ mapred job
Usage: CLI <command> <args>
[-submit <job-file>]
[-status <job-id>]
[-counter <job-id> <group-name> <counter-name>]
[-kill <job-id>]
[-set-priority <job-id> <priority>]. Valid values for
priorities
are: VERY_HIGH HIGH NORMAL LOW VERY_LOW
[-events <job-id> <from-event-#> <#-of-events>]
[-history <jobHistoryFile>]
[-list [all]]
[-list-active-trackers]
[-list-blacklisted-trackers]
[-list-attempt-ids <job-id> <task-type> <task-state>]. Valid
values
for <task-type> are REDUCE MAP. Valid values for <task-state>
are
running, completed
[-kill-task <task-attempt-id>]
[-fail-task <task-attempt-id>]
[-logs <job-id> <task-attempt-id>]
Generic options supported are
-conf <configuration file> specify an application configuration
file
-D <property=value> use value for given property
```

```
-fs <local|namenode:port> specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated
files to
be copied to the map reduce cluster
-libjars <comma separated list of jars> specify comma separated
jar
files to include in the classpath.
-archives <comma separated list of archives> specify comma
separated
archives to be unarchived on the compute machines.
The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]
```

MapReduce Programming

The classic Java WordCount program for Hadoop is compiled and run.

Compiling and Running the Hadoop WordCount Example

The Apache Hadoop WordCount.java program for Hadoop version 2, is the equivalent of the C programing language helloworld.c example. It should be noted that two versions of this program can be found on the Internet. The Hadoop version1 example uses the older org.apache.hadoop.mapred API, while the Hadoop version2 example, shown here in Listing 6.1, uses the newer org.apache.hadoop.mapreduce API. If you experience errors compiling WordCount.java, double-check the source code and Hadoop versions.

WordCount.java

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class WordCount {
      public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();
```

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        }
}
//REDUCER
public static class IntSumReducer
extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values,Context
        context) throws IOException, InterruptedException {
                int sum = 0;
                for (IntWritable val : values) {
                        sum += val.get();
                }
                result.set(sum);
                context.write(key, result);
        }
}
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

WordCount is a simple application that counts the number of occurrences of each word in a given input set. The MapReduce job proceeds as follows:

(input) <k1, v1> -> **map** -> <k2, v2> -> **combine** -> <k2, v2> -> **reduce** -> <k3, v3>

(output)

The mapper implementation, via the map method, processes one line at a time as provided by the specified TextInputFormat class. It then splits the line into tokens separated by whitespaces using the StringTokenizer and emits a key–value pair of <word, 1>. The relevant code section is as follows:

```
public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        }
}
```

Given two input files with contents Hello World Bye World and Hello Hadoop Goodbye Hadoop, the WordCount mapper will produce two maps:

< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>

WordCount sets a mapper

```
job.setMapperClass(TokenizerMapper.class);
```

A combiner

```
job.setCombinerClass(IntSumReducer.class);
```

A reducer

```
job.setReducerClass(IntSumReducer.class);
```

Hence, the output of each map is passed through the local combiner (which sums the values in the same way as the reducer) for local aggregation and then sends the data on to the final reducer. Thus, each map above the combiner performs the following pre-reductions:

< Bye, 1>
< Hello, 1>

```
< World, 2>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
```

The reducer implementation, via the reduce method, simply sums the values, which are the occurrence counts for each key. The relevant code section is as follows:

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
                sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
}
```

The final output of the reducer is the following:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>`
```

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

1. Make a local wordcount_classes directory.

```
$ mkdir wordcount_classes
```

2. Compile the WordCount.java program using the 'hadoop classpath' command to include all the available Hadoop class paths.

```
$ javac -cp `hadoop 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-andpeacetxt:

```
$ hdfs dfs -mkdir war-and-peace-input
$ hdfs dfs -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-andpeace-
output
```

References:

Douglas Eadline,**"Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem",** 1stEdition, Pearson Education, 2016. ISBN-13: 978-9332570351