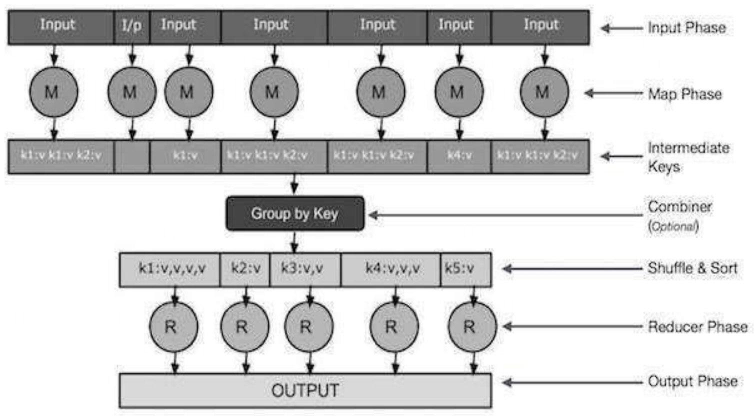
Sub	Big Data Analytics – IAT 3- Solution							Code	22MCA412
Date	30.08.2025	Duration	90 mins	Max Marks	50	Sem:	IV	Branch	MCA

1 **What is MapReduce? Illustrate the logical data flow of MapReduce with a neat diagram**.

Hadoop to move the MapReduce computation to each machine hosting a part of the data. MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function. The MapReduce algorithm contains two important tasks, namely Map and Reduce.



The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).

The Reduce task takes the output from the Map as an input and combines those data tuples (key- value pairs) into a smaller set of tuples. The reduce task is always performed after the map job.

Input Phase − Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.

Map − Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.
Intermediate Keys − They key-value pairs generated by the mapper are known as intermediate keys.

Combiner − A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper. It is not a part of the main MapReduce algorithm; it is optional.

Shuffle and Sort − The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.

Reducer − The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.

Output Phase − In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

2 **Explain the working of the MapReduce model with a single reducer task. Support your answer with a neat diagram.**

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker. Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user- defined map function for each record in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the pro-cessing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.
Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization since it doesn’t use valuable cluster bandwidth. Sometimes, however, all three nodes hosting the HDFS block replicas for a map task’s input split are running other map tasks so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer.

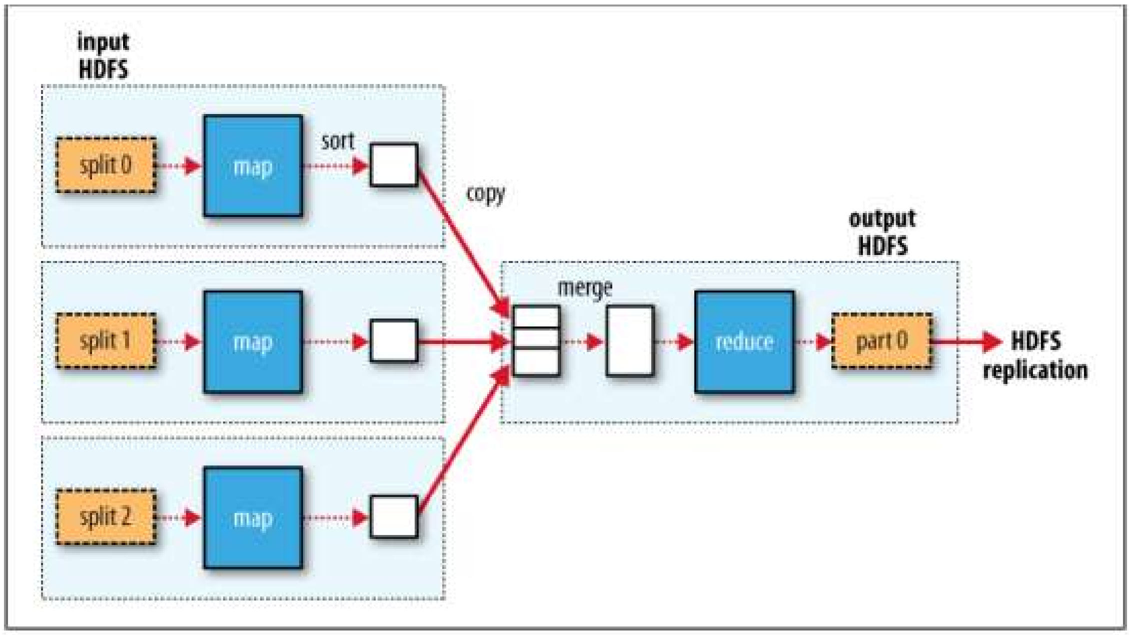


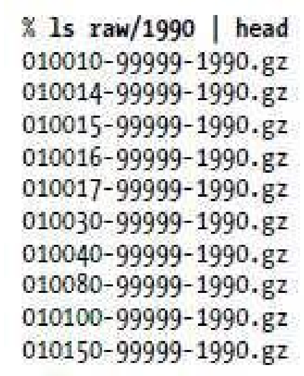
Figure 2-3. MapReduce data flow with a single reduce task

3 **Describe the data format of the Weather Dataset and write a Unix command to retrieve the maximum temperature.**

The data we will use is from the National Climatic Data Center (NCDC, http://www.ncdc.noaa.gov/). The data is stored using a line-oriented ASCII format, in which each line is a record.

The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we shall focus on the basic elements, such as temperature, which are always present and are of fixed width.

Dataset shows a sample line with some of the salient fields highlighted. The line has been split into multiple lines to show each field: in the real file, fields are packed into one line with no delimiters.



```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

Data files are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file. What's the highest recorded global temperature for each year in the dataset? We will answer this first without using Hadoop, as this information will provide a performance baseline, as well as a useful means to check our results. The classic tool for processing line-oriented data is awk. Example 5-2 is a small script to calculate the maximum temperature for each year.
A program for finding the maximum recorded temperature by year from NCDC weather records

```
#!/usr/bin/env bash for year in all/* do
echo -ne `basename $year .gz`"\t" gunzip -c $year | \
awk '{ temp = substr($0, 88, 5) + 0;
q = substr($0, 93, 1);
if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp } END { print max  }'
```

4 **Explain how the MapReduce model functions with multiple reducer tasks. Illustrate with a neat diagram.**

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker. Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user- defined map function for each record in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the pro-cessing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.
Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization since it doesn't use valuable cluster bandwidth. Sometimes, however, all three nodes hosting the HDFS block replicas for a map task's input split are running other map tasks so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer.

The number of reduce tasks is not governed by the size of the input, but is specified independently. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in below Figure. This diagram makes it clear why the data flow between map and reduce tasks is collo-quially known as "the shuffle," as each reduce task is fed by many map tasks.

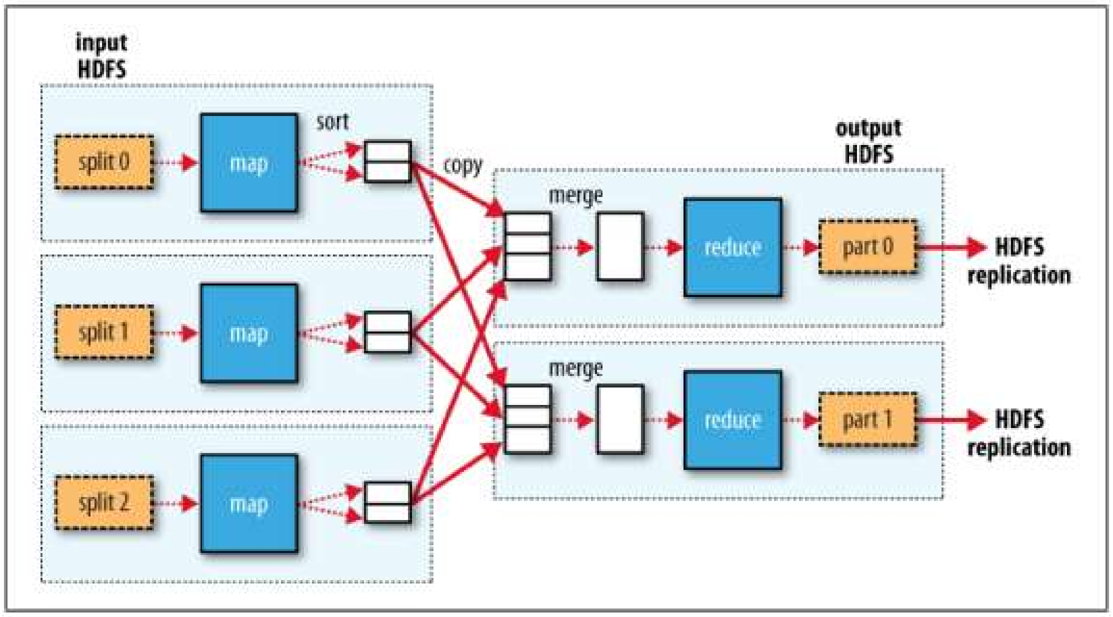
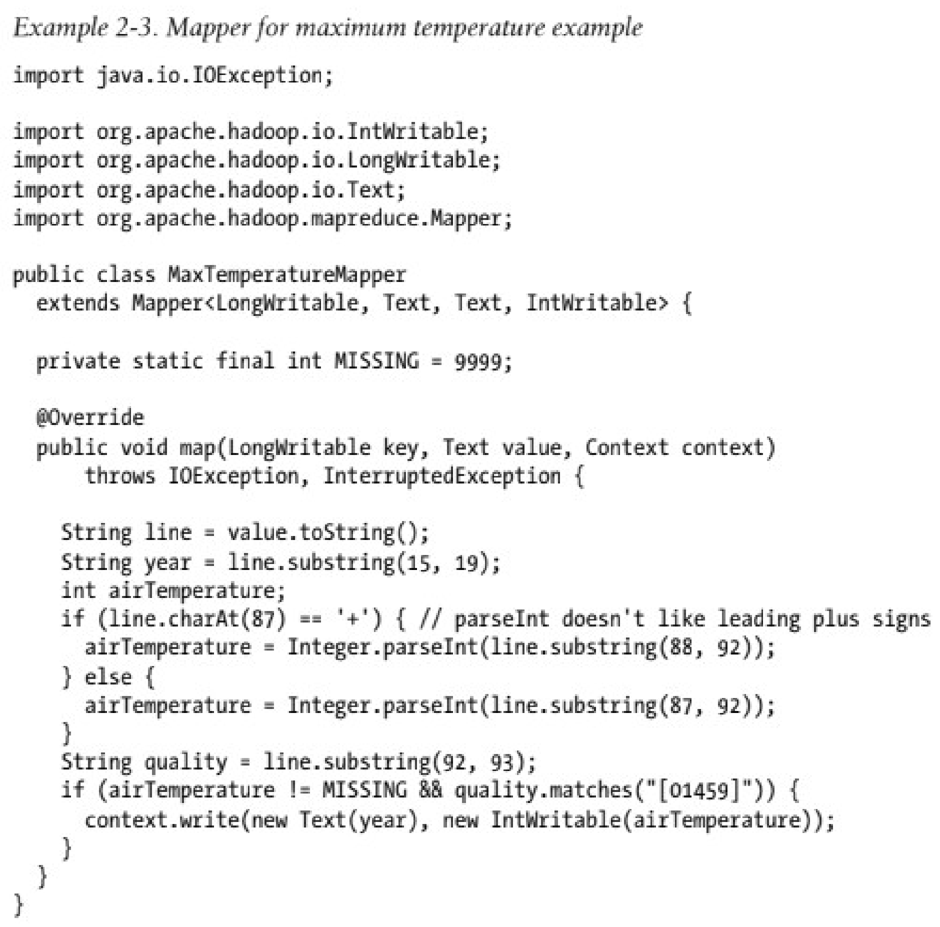


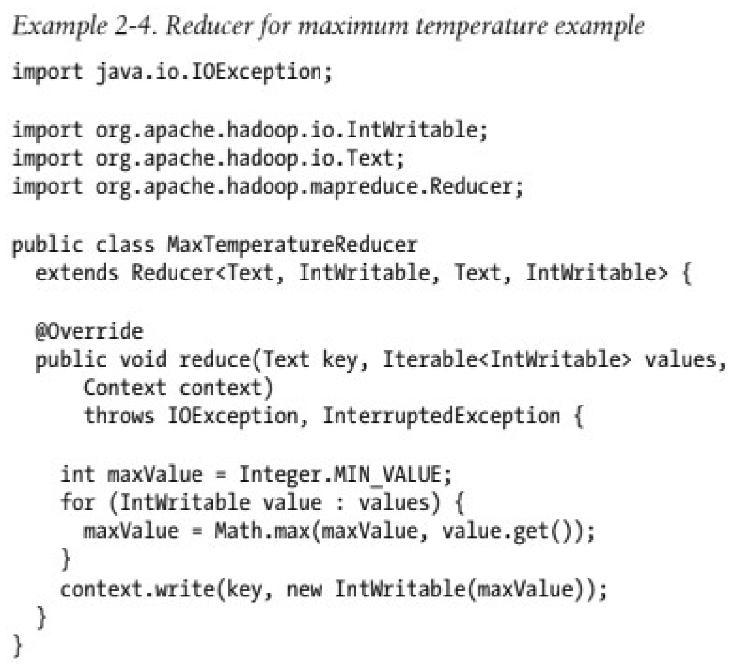
Figure 2-4. MapReduce data flow with multiple reduce tasks

5. **Write a Java MapReduce program to determine the maximum temperature from the Weather Dataset.**



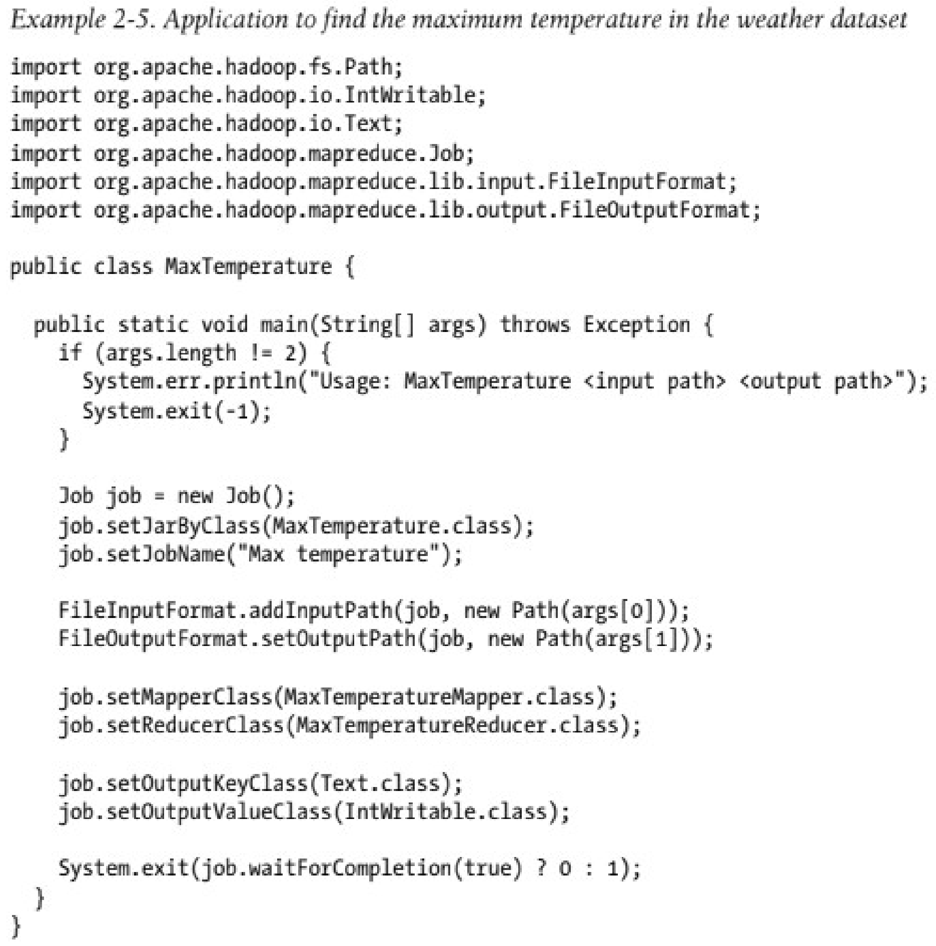
Example 2-3. Mapper for maximum temperature example

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```



Example 2-4. Reducer for maximum temperature example

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```



Example 2-5. Application to find the maximum temperature in the weather dataset

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

6. **What is Hadoop Streaming? Demonstrate how to find the maximum temperature from the NCDC dataset using Python and Ruby code.**

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. Hadoop Streaming uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program. Streaming is naturally suited for text processing (although, as of version 0.21.0, it can handle binary streams, too), and when used in text mode, it has a line-oriented view of data. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output. Let's illustrate this by rewriting our MapReduce program for finding maximum temperatures by year in Streaming.

```
#!/usr/bin/env ruby
STDIN.each_line do |line|
val = line
year, temp, q = val[15,4], val[87,5], val[92,1]
puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

```
#!/usr/bin/env ruby
last_key, max_val = nil, 0
STDIN.each_line do |line|
key, val = line.split("\t")
```

```
if last_key && last_key != key
puts "#{last_key}\t#{max_val}"
last_key, max_val = key, val.to_i
else
last_key, max_val = key, [max_val, val.to_i].max
end
end
puts "#{last_key}\t#{max_val}" if last_key
```

```
#!/usr/bin/env python
import re
import sys
for line in sys.stdin:
val = line.strip()
(year, temp, q) = (val[15:19], val[87:92], val[92:93])
if (temp != "+9999" and re.match("[01459]", q)):
print "%s\t%s" % (year, temp)
```

```
#!/usr/bin/env python
import sys
(last_key, max_val) = (None, 0)
for line in sys.stdin:
(key, val) = line.strip().split("\t")
if last_key and last_key != key:
print "%s\t%s" % (last_key, max_val)
(last_key, max_val) = (key, int(val))
else:
(last_key, max_val) = (key, max(max_val, int(val)))
if last_key:
print "%s\t%s" % (last_key, max_val)
```

7. **Explain in detail the steps involved in executing a MapReduce program on a Hadoop cluster.**

Running on a Cluster

Now that we are happy with the program running on a small test dataset, we are ready to try it on the full dataset on a Hadoop cluster. Chapter 9 covers how to set up a fully distributed cluster, although you can also work through this section on a pseudo-distributed cluster.

Packaging

We don't need to make any modifications to the program to run on a cluster rather than on a single machine, but we do need to package the program as a JAR file to send to the cluster. This is conveniently achieved using Ant, using a task such as this (you can find the complete build file in the example code):

```
<jar destfile="hadoop-examples.jar" basedir="${classes.dir}"/>
```

If you have a single job per JAR, then you can specify the main class to run in the JAR file's manifest. If the main class is not in the manifest, then it must be specified on the command line (as you will see shortly). Also, any dependent JAR files should be pack-aged in a lib subdirectory in the JAR file. (This is analogous to a Java Web application archive, or WAR file, except in that case the JAR files go in a WEB-INF/lib subdirectory in the WAR file.)

Launching a Job To launch the job, we need to run the driver, specifying the cluster that we want to run the job on with the -conf option (we could equally have used the -fs and -jt options):

```
% hadoop jar hadoop-examples.jar v3.MaxTemperatureDriver -conf conf/hadoop-cluster.xml \

input/ncdc/all max-temp
```

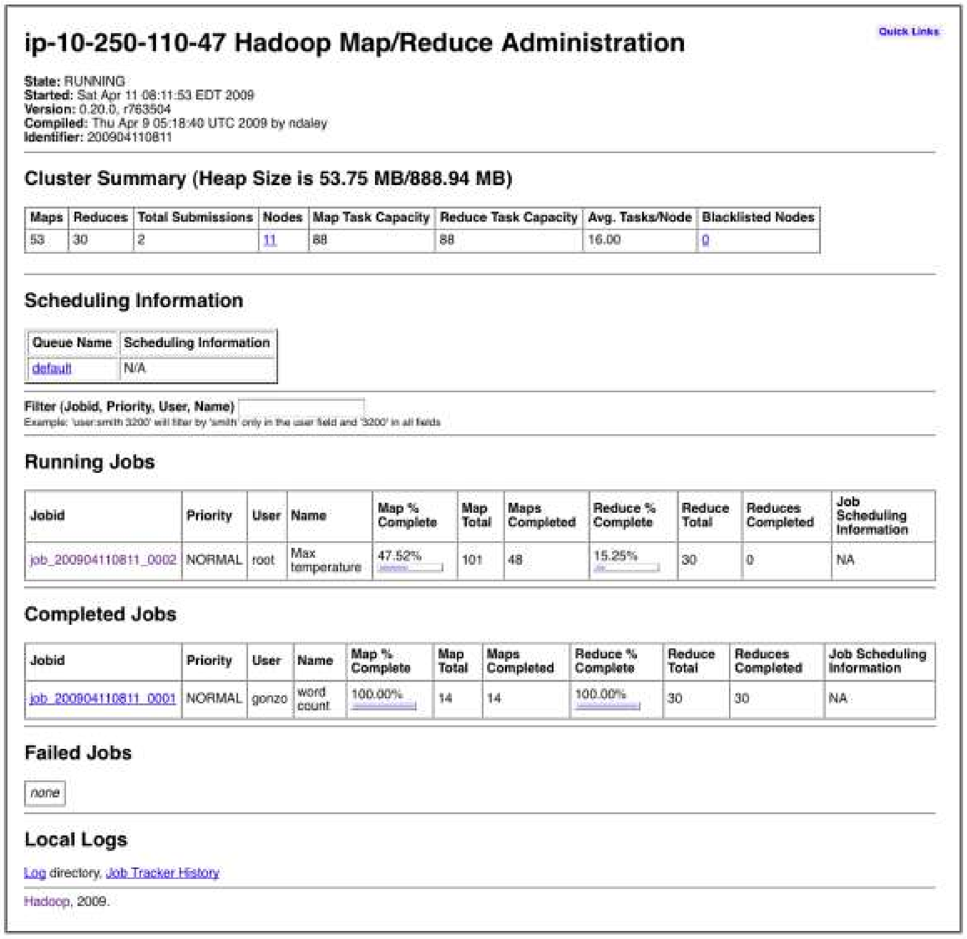
The MapReduce Web UI

Hadoop comes with a web UI for viewing information about your jobs. It is useful for following a job's progress while it is running, as well as finding job statistics and logs after the job has completed. You can find the UI at http://jobtracker-host:50030/.

The jobtracker page A screenshot of the home page is shown in Figure below. The first section of the page gives details of the Hadoop installation, such as the version number and when it was compiled, and the current state of the jobtracker (in this case, running), and when it was started.

Next is a summary of the cluster, which has measures of cluster capacity and utilization. This shows the number of maps and reduces currently running on the cluster, the total number of job submissions, the number of tasktracker nodes currently available, and the cluster's capacity: in terms of the number of map and reduce slots available across the cluster ("Map Task Capacity" and "Reduce Task Capacity"), and the number of available slots per node, on average. The number of tasktrackers that have been black listed by the jobtracker is listed as well.

Below the summary, there is a section about the job scheduler that is running (here the default). You can click through to see job queues. Further down, we see sections for running, (successfully) completed, and failed jobs. Each of these sections has a table of jobs, with a row per job that shows the job's ID,owner, name (as set in the Job constructor or setJobName() method, both of which internally set the mapred.job.name property) and progress information. Finally, at the foot of the page, there are links to the jobtracker's logs, and the jobtracker's history: information on all the jobs that the jobtracker has run. The main view displays only 100 jobs (configurable via the mapred.jobtracker.completeuserjobs.maximum property), before consigning them to the history page. Note also that the job history is persistent, so you can find jobs here from previous runs of the jobtracker.



ip-10-250-110-47 Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2009.

Figure 5-1. Screenshot of the jobtracker page

The job page

Clicking on a job ID brings you to a page for the job, illustrated in Figure 5-2. At the top of the page is a summary of the job, with basic information such as job owner and name, and how long the job has been running for. The job file is the consolidated configuration file for the job, containing all the properties and their values that were in effect during the job run. If you are unsure of what a particular property was set to, you can click through to inspect the file.

While the job is running, you can monitor its progress on this page, which periodically updates itself. Below the summary is a table that shows the map progress and the reduce progress. "Num Tasks" shows the total number of map and reduce tasks for this job (a row for each). The other columns then show the state of these tasks: "Pending" (waiting to run), "Running," "Complete" (successfully run), "Killed" (tasks that have failed—this column would be more accurately labeled "Failed"). The final column shows the total number of failed and killed task attempts for all the map or reduce tasks for the job (task attempts may be marked as killed if they are a speculative execution duplicate, if the tasktracker they are running on dies or if they are killed by a user). See "Task Failure" on page 200 for background on task failure.

In the middle of the page is a table of job counters. These are dynamically updated during the job run, and provide another useful window into the job's progress and general health.

Retrieving the Results

Once the job is finished, there are various ways to retrieve the results. Each reducer produces one output file, so there are 30 part files named part-r-00000 to part-r-00029 in the max-temp directory. This job produces a very small amount of output, so it is convenient to copy it from HDFS to our files in the directory specified in the source pattern and merges them into a single file on the local filesystem:

```
% hadoop fs -getmerge max-temp max-temp-local

    % sort max-temp-local | tail
    1991 607
    1992 605
    1993 567
    1994 568
    1995 567
    1996 561
    1997 565
    1998 568
    1999 568
    2000 558
```

8. **Explain the Hadoop configuration files, their APIs, and how properties can be accessed.**

Components in Hadoop are configured using Hadoop's own configuration API. An instance of the Configuration class (found in the org.apache.hadoop.conf package) represents a collection of configuration properties and their values. Each property is named by a String, and the type of a value may be one of several types, including Java primitives such as boolean, int, long, float, and other useful types such as String, Class, java.io.File, and collections of Strings.

Configurations read their properties from resources—XML files with a simple structure for defining name-value pairs.

```
<?xml version="1.0"?>

<configuration>

<property>

<name>color</name>

<value>yellow</value>

<description>Color</description>

</property>

<property>

<name>size</name>

<value>10</value>

<description>Size</description>

</property>

<property>
```

```
<name>weight</name>

<value>heavy</value>

<final>true</final>

<description>Weight</description>

</property>

<property>

<name>size-weight</name>

<value>${size},${weight}</value>

<description>Size and weight</description>

</property>

</configuration>
```

Assuming this configuration file is in a file called configuration-1.xml, we can access its properties using a piece of code like this:

```
Configuration conf = new Configuration();

conf.addResource("configuration-1.xml");

assertThat(conf.get("color"), is("yellow"));

assertThat(conf.getInt("size", 0), is(10));

assertThat(conf.get("breadth", "wide"), is("wide"));
```

There are a couple of things to note: type information is not stored in the XML file; instead, properties can be interpreted as a given type when they are read. Also, the get() methods allow you to specify a default value, which is used if the property is not defined in the XML file, as in the case of breadth here.

9. **Write short notes on the following:**

Remote Debugging

When a task fails and there is not enough information logged to diagnose the error, you may want to resort to running a debugger for that task. This is hard to arrange when running the job on a cluster, as you don't know which node is going to process which part of the input, so you can't set up your debugger ahead of the failure. However, there are a few other options available:

Reproduce the failure locally: Often the failing task fails consistently on a particular input. You can try to repoduce the problem locally by downloading the file that the task is failing on and running the job locally, possibly using a debugger such as Java's VisualVM.

Use JVM debugging options A common cause of failure is a Java out of memory error in the task

JVM.

You can set mapred.child.java.opts to include -XX:-HeapDumpOnOutOfMemoryError -XX:Heap DumpPath=/path/to/dumps to produce a heap dump which can be examined after- wards with tools like jhat or the Eclipse Memory Analyzer. Note that the JVM options should be added to the existing memory settings specified by mapred.child.java.opts. Use task profiling Java profilers give a lot of insight into the JVM, and Hadoop provides a mechanism to profile a subset of the tasks in a job.

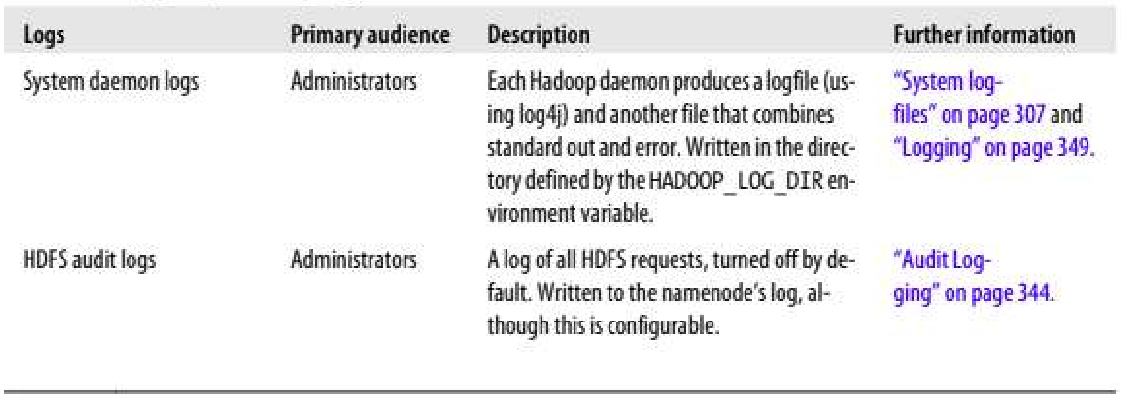
Use IsolationRunner: Older versions of Hadoop provided a special task runner called IsolationRunner that could rerun failed tasks in situ on the cluster. Unfortunately, it is no longer available in recent versions, but you can track its replacement at https://issues.apache.org/jira/browse/MAPREDUCE-2637.

In some cases it's useful to keep the intermediate files for a failed task attempt for later inspection, particularly if supplementary dump or profile files are created in the task's working directory. You can set keep.failed.task.files to true to keep a failed task's files. You can keep the intermediate files for successful tasks, too, which may be handy if you want to examine a task that isn't failing. In this case, set the property keep.task.files.pattern to a regular expression that matches the IDs of the tasks you want to keep.

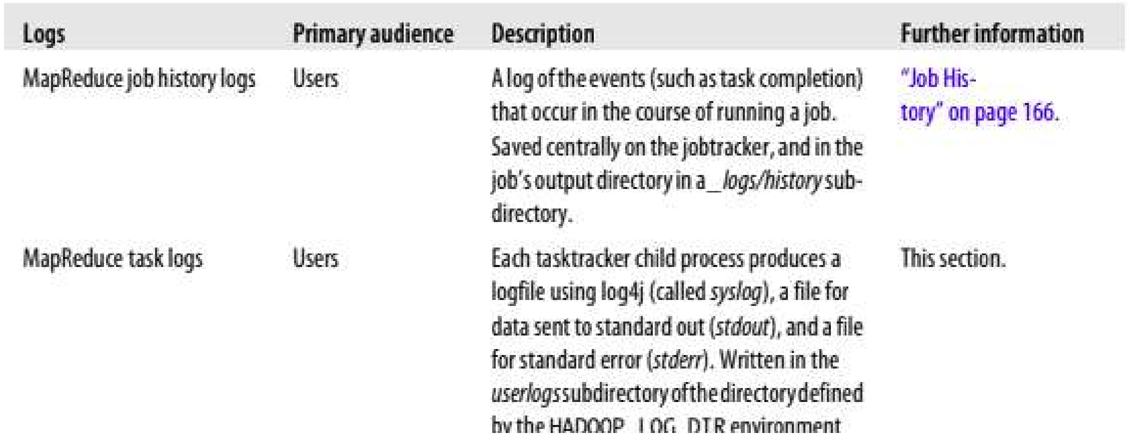
To examine the intermediate files, log into the node that the task failed on and look for the directory for that task attempt. It will be under one of the local MapReduce directories, as set by the mapred.local.dir property. If this property is a comma-separated list of directories (to spread load across the physical disks on a machine), then you may need to look in all of the directories before you find the directory for that particular task attempt. The task attempt directory is in the following location: mapred.local.dir/taskTracker/jobcache/job-ID/task-attempt-ID

Hadoop Logs: Hadoop produces logs in various places, for various audiences. These are summarized in Table below.

Table 5-2. Types of Hadoop logs



Logs	Primary audience	Description	Further information
System daemon logs	Administrators	Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the HADOOP_LOG_DIR environment variable.	"System logfiles" on page 307 and "Logging" on page 349.
HDFS audit logs	Administrators	A log of all HDFS requests, turned off by default. Written to the namenode's log, although this is configurable.	"Audit Logging" on page 344.

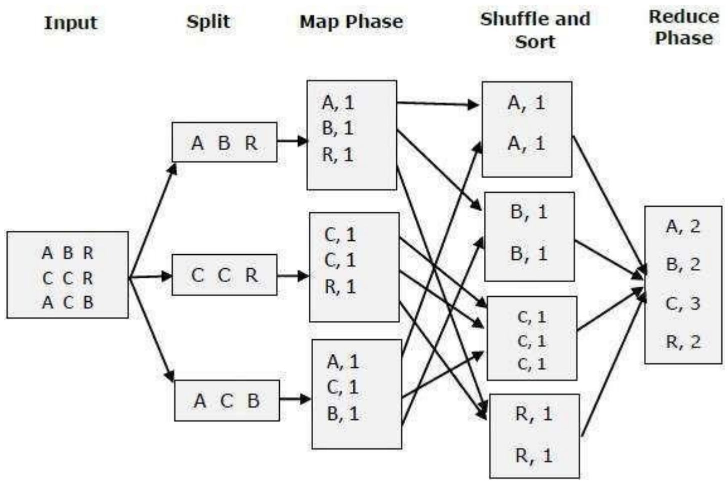


Logs	Primary audience	Description	Further information
MapReduce job history logs	Users	A log of the events (such as task completion) that occur in the course of running a job. Saved centrally on the jobtracker, and in the job's output directory in a _logs/history subdirectory.	"Job History" on page 166.
MapReduce task logs	Users	Each tasktracker child process produces a logfile using log4j (called *syslog*), a file for data sent to standard out (*stdout*), and a file for standard error (*stderr*). Written in the *userlogs* subdirectory of the directory defined by the HADOOP_LOG_DIR environment	This section.

10 **Explain the concept of MapReduce with the help of a Word Count problem.**

MapReduce is a powerful programming model introduced by Google to handle large-scale data processing in a distributed and parallel manner. The idea is to break a huge task into smaller independent subtasks that can run across multiple machines. It operates mainly in two phases: the **Map phase** and the **Reduce phase**. In the Map phase, input data is split into chunks, and each chunk is processed independently to generate key-value pairs. The framework then performs a **shuffle and sort** operation, where values belonging to the same key are grouped together. Finally, in the Reduce phase, the grouped values are aggregated or combined to generate the final output. This makes MapReduce highly scalable and fault-tolerant, suitable for processing terabytes or petabytes of data.

The **Word Count problem** is a classic example to illustrate how MapReduce works. Suppose the input is a set of text documents. In the Map phase, each line of text is read, split into words, and each word is mapped to the key-value pair (word, 1). Next, during the shuffle and sort phase, all the occurrences of the same word are grouped together. For instance, multiple (Hello, 1) pairs will be combined under the key Hello. In the Reduce phase, these values are summed up to get the final frequency of each word. Thus, for an input like *"Hello World Hello Hadoop"*, the output will be *Hello → 2, World → 1, Hadoop → 1*. This simple yet effective example demonstrates how MapReduce simplifies big data analysis by parallelizing work and automating complex tasks like data distribution and fault recovery.


Input
Split
Map Phase
Shuffle and Sort
Reduce Phase
A B R
C C R
A C B
A B R
C C R
A C B
A, 1
B, 1
R, 1
C, 1
C, 1
R, 1
A, 1
C, 1
B, 1
A, 1
A, 1
B, 1
B, 1
C, 1
C, 1
C, 1
R, 1
R, 1
A, 2
B, 2
C, 3
R, 2