UNIT-II

MapReduce

MapReduce is a programming model for data processing. Hadoop can run MapReduce programs written in various languages like Java, Ruby, Python, and C++.MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal.

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with MapReduce, since it is semistructured and record-oriented.

Data Format

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. For simplicity, we shall focus on the basic elements, such as temperature, which are always present and are of fixed width.

Example 2-1 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.

Example 2-1. Format of a National Climate Data Center record

```
0057
332130 # USAF weather station identifier
99999 # WBAN weather station identifier
19500101 # observation date
0300 # observation time
4
+51317 # latitude (degrees x 1000)
+028783 # longitude (degrees x 1000)
FM-12
+0171 # elevation (meters)
99999
V020
320 # wind direction (degrees)
1 # quality code
N
0072
```

```
1
00450 # sky ceiling height (meters)
1 # quality code
C
N
010000 # visibility distance (meters)
1 # quality code
N
9
-0128 # air temperature (degrees Celsius x 10)
1 # quality code
-0139 # dew point temperature (degrees Celsius x 10)
1 # quality code
10268 # atmospheric pressure (hectopascals x 10)
1 # quality code
```

nData 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:

```
% 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
```

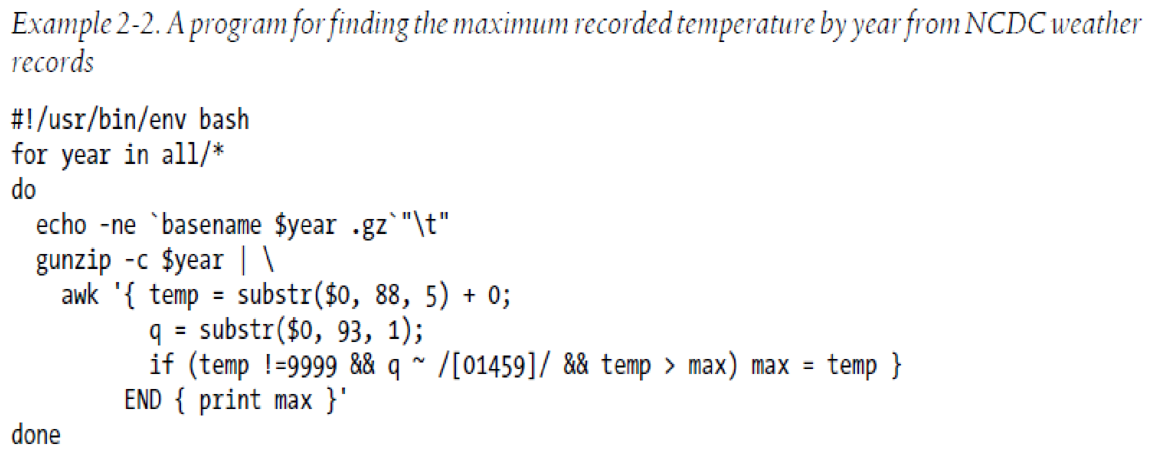
Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file.

Analyzing the Data with Unix Tools:

Find maximum temperature for each year for given weather data set with out using Hadoop**.**The classic tool for processing line-oriented data is *awk*.

Example 2-2 is a small script to calculate the maximum temperature for each year.

Example 2-2. A program for finding the maximum recorded temperature by year from NCDC weather records



Example 2-2. 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 }'
done
```

- The script loops through the compressed year files, then uncompressed files and prints the year first.
- Using awk process each line and extracts two fields from the line.
    - ✓ air temperature
    - ✓ quality code.
- The air temperature value is turned into an integer by adding 0 and then test is applied to see if the temperature is valid and quality code is Ok indicates that the reading is not suspect or erroneous.
- If air temperature is valid and quality code is OK, then value is compared with the maximum value seen so far, which is updated if a new maximum is found.
- The END block is executed after all the lines in the file have been processed, and it prints the maximum value.
- The temperature values in the source file are scaled by a factor of 10, we get maximum temperature of 31.7°C for 1901.

To speed up the processing, we need to run parts of the program in parallel. we could process different years in different processes, using all the available hardware threads on a machine. There are a few problems with this

Problems:

1. Dividing the work into equal-size pieces isn't always easy. If file sizes are different some processes will finish much earlier than others. A better approach is one requires to split the input into fixed-size chunks and assign each chunk to a process.
2. Combining the results from independent processes may need further processing.

3. Single machine has limited processing capacity. Better to use multiple machines but who runs the overall job? How do we deal with failed processes?

It's feasible to parallelize the processing, in practice it's messy. Using a framework like Hadoop to take care of these issues is a great help.

Analyzing the Data with Hadoop

MapReduce works by breaking the processing into two phases

- map phase
- reduce phase

Each phase has key-value pairs as input and output, the types of which
may be chosen by the programmer. The input to our map phase is the raw NCDC data.
The programmer also specifies two functions:

- map function
- reduce function.

We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

The map function extract year and the air temperature, since these are the only fields we are interested in. Here map function is just a data preparation phase, setting up the data and then reducer function finding the maximum temperature for each year.

consider the following sample lines of input data

(some unused columns have been dropped to fit the page, indicated by ellipses):
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
These lines are presented to the map function as the key-value pairs:
(0, 006701199099999**1950**051507004...9999999N9+**0000**1+99999999999...)
(106, 004301199099999**1950**051512004...9999999N9+**0022**1+99999999999...)
(212, 004301199099999**1950**051518004...9999999N9-**0011**1+99999999999...)
(318, 004301265099999**1949**032412004...0500001N9+**0111**1+99999999999...)
(424, 004301265099999**1949**032418004...0500001N9+**0078**1+99999999999...)

map function output:

(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)

The map function output is sorts and groups the key-value pair by key using MapReduce framework before sent to reduce function. Reduce function input key-value pair as follows

(1949, [111, 78])
(1950, [0, 22, −11])

Each year appears with a list of all its air temperature readings. The reduce function iterate through the list and pick up the maximum reading:

(1949, 111)
(1950, 22)

This is the final output: the maximum global temperature recorded in each year.

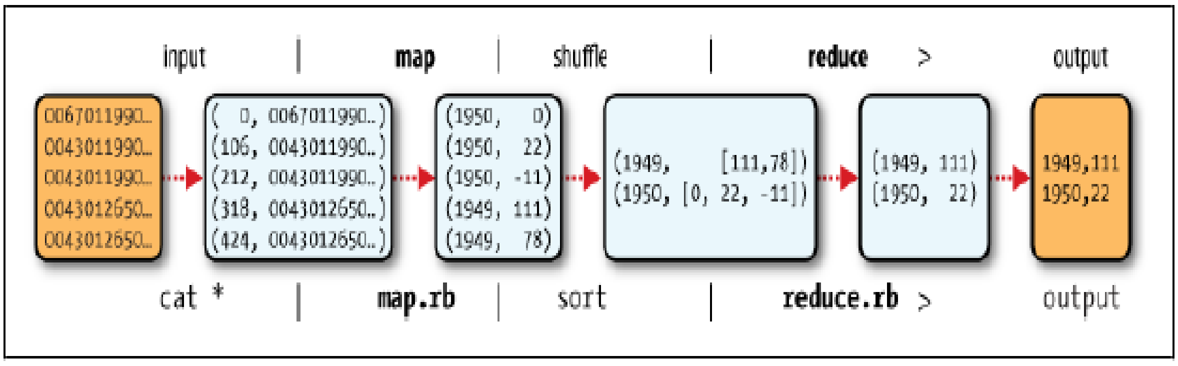


Figure 2-1. MapReduce logical data flow

MapReduce program need three things: a map function, a reduce function, and some code to run the job. The map function is represented by the Mapper class, which declares an abstract map() method. The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function.

Example: input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature.

Hadoop provides its own set of basic types that are found in the org.apache.hadoop.io package. Here we use LongWritable, which corresponds to a Java Long, Text (like Java String), and IntWritable (like Java Integer).

Example 2-3. Mapper for maximum temperature example

```
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));
}
}
}
```

The reduce function is represented by the Reducer class, which declares an abstract reduce() method. Again, four formal type parameters are used to specify the input and output types for the reduce function. The input types of the reduce function must match the output types of the map function.

Example 2-4. Reducer for maximum temperature example

```
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));
}
}
```

The third piece of code runs the MapReduce job (see Example 2-5).

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);
}
}
```

Test run

- Install Hadoop in standalone mode in which Hadoop runs using the local filesystem with a local job runner.
- After writing a MapReduce job, it's normal to **try it out on a small dataset** to flush out any immediate problems with the code.

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
```

Example:
we can follow the number of records that went through the system: five map inputs produced five map outputs, then five reduce inputs in two groups produced two reduce outputs. The output was written to the *output* directory, which contains one output file per reducer.

- The job had a single reducer, so we find a single file, named *part-r-00000*:

```
% cat output/part-r-00000
                1949 111
                1950 22
```

The old and the new Java MapReduce APIs

- The Java MapReduce API used in the previous section was first released in **Hadoop 0.20.0.** This new API, sometimes referas **"Context Objects**," was designed to make the API easier to evolve in the future.
- It is type-incompatible with the old, however, so applications need to be rewritten to take advantage of it.
- The new API is not complete in the 1.x (formerly 0.20) release series, so the old API is recommended for these releases, despite having been marked as deprecated in the early 0.20 releases.
- Previous editions of this book were based on 0.20 releases, and used the old API throughout (although the new API was covered, the code invariably used the old API).

Several notable differences between the two APIs:

Old API	New APT
Favor of interfaces	Favors abstract classes over interfaces.
Found in org.apache.hadoop.mapred package.	Found in org.apache.hadoop.mapreduce package
Use JobConf, the OutputCollector, and the Reporter objects to	use context object to communicate with mapreduce system.

communicate with mapreduce system.	
Only the mapper to control the execution not reducer.	Both mappers and reducers to control the execution flow by overriding the run() method
special JobConf object for job configuration,	Configuration has been unified
Job control is performed through the JobClient	Job control is performed through the Job class
Both map and reduce outputs are named *part-nnnnn*	map outputs are named *partm-nnnnn*, and reduce outputs are named *part-r-nnnnn*
reduce() method passes values as a java.lang.Iterator.	reduce() method passes values as a java.lang.Iterable
Implements the interfaces	User-overridable methods are declared to throw java.lang.InterruptedException.
Mapper and Reducer are interfaces. No default implementation	Mapper and Reducer interfaces in the are abstract classes with default implementation

Example 2-6 shows the MaxTemperature application rewritten to use the old API. The differences are highlighted in bold. *Example 2-6. Application to find the maximum temperature, using the old MapReduce API*

```
public class OldMaxTemperature {
static class OldMaxTemperatureMapper extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
private static final int MISSING = 9999;
public void map(LongWritable key,Text value,
        OutputCollector<Text, IntWritable> output , Reporter reporter) throws
        IOException {
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]")) {
        output.collect(new Text(year), new IntWritable(airTemperature));
}
}
}
static class OldMaxTemperatureReducer extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
int maxValue = Integer.MIN_VALUE;
while (values.hasNext()) {
maxValue = Math.max(maxValue, values.next().get());
}
output.collect(key, new IntWritable(maxValue));
}
}

public static void main(String[] args) throws IOException {
if (args.length != 2) {
System.err.println("Usage: OldMaxTemperature <input path> <output path>");
System.exit(-1);
}
JobConf conf = new JobConf(MaxTemperatureWithCombiner.class);
conf.setJobName("Max temperature");
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
conf.setMapperClass(OldMaxTemperatureMapper.class);
conf.setReducerClass(OldMaxTemperatureReducer.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
JobClient.runJob(conf);
}
}
```

Scaling Out

We have seen how MapReduce works for small inputs; now it’s time to view of the system and look at the data flow for large inputs.

To scale out, we need to store the data in a distributed filesystem(HDFS).This allows Hadoop to move the MapReduce computation to each machine hosting a part of the data

Data Flow:

- 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 into *map tasks* and *reduce tasks*.
- To control the job execution process by 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*.
- Hadoop creates one map task for each split, which runs the user defined map function for each *record* in the split.
- load balancing helps 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.

Optimal split size is the same as **size of an HDFS block, 64 MB by default**. Hadoop 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.

- Data-local: map task and data is on the same node.
- Rack-local: map task and data is on different nodes but both are in the rack.
- Off-rack: map task and data is on different racks.

Job scheduler first look for a free map slot on same node where the data available, if not then look rack-local. Very occasionally look for off-rack, which results in an inter-rack network transfer.The three possibilities are illustrated in Figure 2-2.

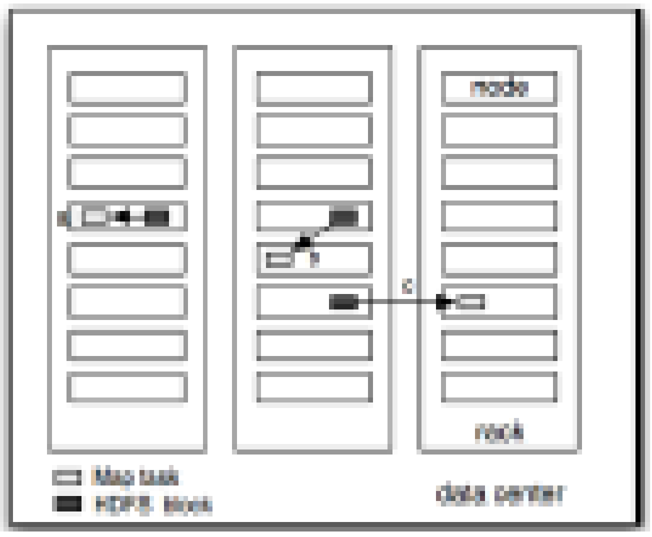


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks.

Map tasks write their output to the local disk, not to HDFS because Map output is intermediate output and it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away.

If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. The sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function.

The output of the reduce is stored in HDFS for reliability. Each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

The whole data flow with a single reduce task is illustrated in Figure 2-3. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes.

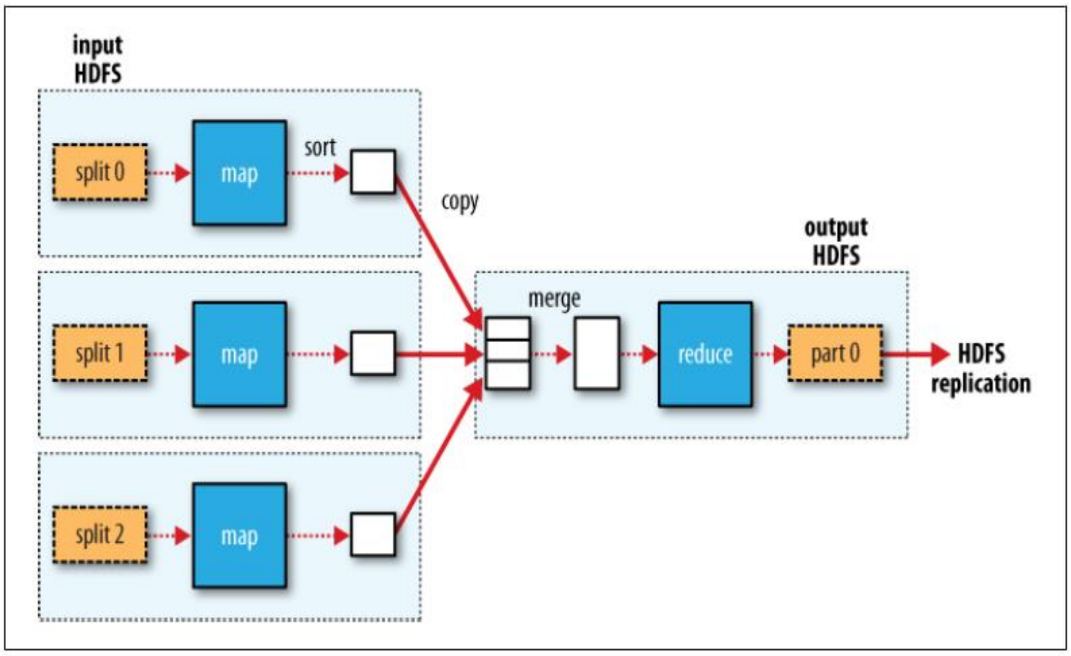


Figure 2-3. MapReduce data flow with a single reduce task

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

- 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.

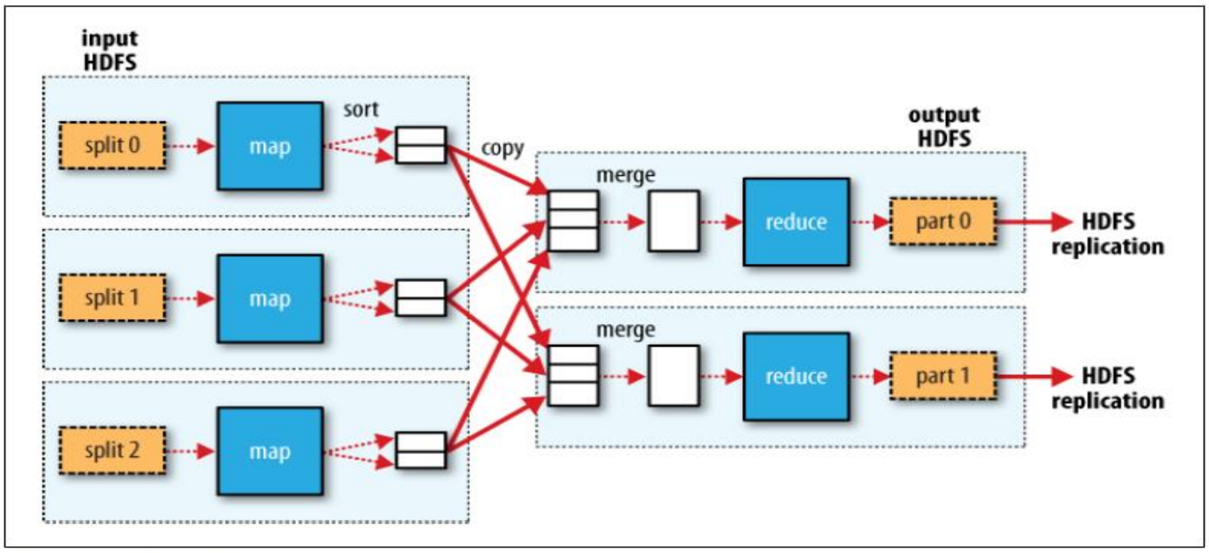


Figure 2-4. MapReduce data flow with multiple reduce tasks

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

- This diagram makes it clear why the data flow between map and reduce tasks is informally known as "the shuffle," as each reduce task is fed by many map tasks.
-
- it's also possible to have zero reduce tasks when we don't need shuffle since the processing can be carried out entirely in parallel.In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).

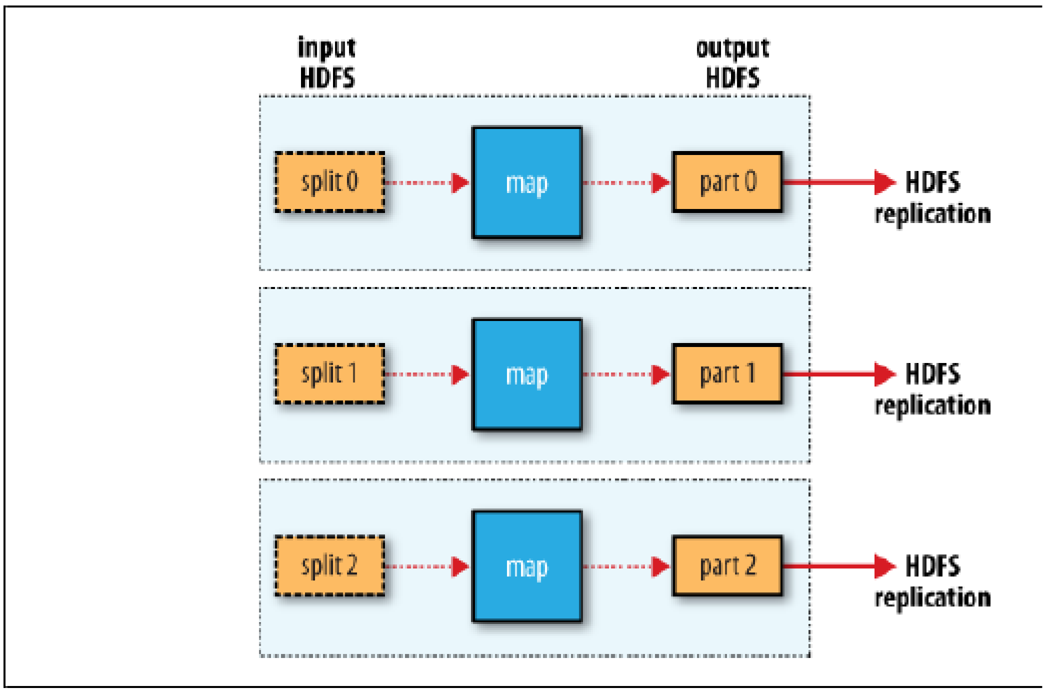


Figure 2-5. MapReduce data flow with no reduce tasks

Combiner Functions:

- Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks.
- Hadoop allows the user to specify a *combiner function* to be run on the map output—the combiner function's output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all.
- Calling the combiner function zero, one, or many times should produce the same output from the reducer.

The contract for the combiner function constrains the type of function that may be used.

Example: Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps.

First map produced the output:

(1950, 0)
(1950, 20)
(1950, 10)

Second map produced the output:

(1950, 25)
(1950, 15)

The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

Output: (1950, 25) since 25 is the maximum value in the list.

The combiner function that finds the maximum temperature for each map output. The reduce would then be called with: **(1950, [20, 25])**

***max*(0, 20, 10, 25, 15) = *max*(*max*(0, 20, 10), *max*(25, 15)) = *max*(20, 25) = 25**

- Not all functions possess this property.
  Example: we calculating mean temperatures but we couldn't use the mean as our combiner function, since:
  ***mean*(0, 20, 10, 25, 15) = 14**

But: ***mean*(*mean*(0, 20, 10), *mean*(25, 15)) = *mean*(10, 20) = 15**

- The combiner function doesn't replace the reduce function because The reduce function is still needed to process records with the same key from different maps.
- But combiner can help cut down the amount of data shuffled between the maps and the reduces, and for this reason alone it is always worth.

Specifying a combiner function

- Java MapReduce program, the combiner function is defined using the Reducer class.
- It is the same implementation as the reducer function in MaxTemperatureReducer but only change is to set the combiner class on the Job.

Example 2-7. Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {
        public static void main(String[] args) throws Exception {
        if (args.length != 2) {
        System.err.println("Usage: MaxTemperatureWithCombiner <input path> "
        +"<output path>");
        System.exit(-1);
        }
        Job job = new Job();
        job.setJarByClass(MaxTemperatureWithCombiner.class);
        job.setJobName("Max temperature");
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(MaxTemperatureMapper.class);
        job.setCombinerClass(MaxTemperatureReducer.class);
        job.setReducerClass(MaxTemperatureReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

Running a Distributed MapReduce Job:

- The same program will run, without alteration, on a full dataset. This is the advantage of MapReduce: it scales to the size of data and the size of hardware.
- On a 10-node EC2 cluster running High-CPU Extra Large Instances, the program took six minutes to run.

Hadoop Streaming

- Hadoop provides an API to MapReduce that allows to write 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 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.

.

Ruby language:

Example 2-8. Map function for maximum temperature in Ruby

-

```
#!/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
```

Faster than the serial run on one machine using *awk* but it doesn't proportionately faster because the input data wasn't evenly partitioned.

The program iterates over lines from standard input by executing a block for each line from STDIN . The block pulls out the relevant fields from each input line, and, if the temperature is valid, writes the year and the temperature separated by a tab character \t to standard output (using puts).

Since the script just operates on standard input and output, it's trivial to test the script without using Hadoop, simply using Unix pipes:

```
%cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb
1950   +0000
1950   +0022
1950    -0011
1949   +0111
1949   +0078
```

Example 2-9. Reduce function for maximum temperature in Ruby

```
#!/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
```

For each line, we pull out the key and value, then if we've just finished a group (last_key && last_key != key), we write the key and the maximum temperature for that group, separated by a tab character, before resetting the maximum temperature for the new key. If we haven't just finished a group, we just update the maximum temperature for the current key.
The last line of the program ensures that a line is written for the last key group in the input.

We can now simulate the whole MapReduce pipeline with a Unix pipeline (which is equivalent to the Unix pipeline shown in Figure 2-1):

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb | \
sort | ch02/src/main/ruby/max_temperature_reduce.rb
1949 111
1950 22
```

The output is the same as the Java program, so the next step is to run it using Hadoop itself.
To specify the Streaming JAR file along with the jar option. Options to the Streaming program specify the input and output paths, and the map and reduce scripts.

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-streaming.jar \
-input input/ncdc/sample.txt \
-output output \
-mapper ch02/src/main/ruby/max_temperature_map.rb \
-reducer ch02/src/main/ruby/max_temperature_reduce.rb
```

When running on a large dataset on a cluster, we should set the combiner, using the combiner option. From release 0.21.0, the combiner can be any Streaming command. For earlier releases,the combiner had to be written in Java.

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-streaming.jar \
-input input/ncdc/all \
-output output \
-mapper "ch02/src/main/ruby/max_temperature_map.rb | sort |
ch02/src/main/ruby/max_temperature_reduce.rb" \
-reducer ch02/src/main/ruby/max_temperature_reduce.rb \
-file ch02/src/main/ruby/max_temperature_map.rb \
-file ch02/src/main/ruby/max_temperature_reduce.rb
```

Note also the use of -file, which we use when running Streaming programs on the cluster to ship the scripts to the cluster.

Python
Streaming supports any programming language that can read from standard input, and write to standard output.
Example 2-10. Map function for maximum temperature in Python

```
#!/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)
```

Example 2-11. Reduce function for maximum temperature in Python

```
#!/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)
```

To run a test:

```
%cat input/ncdc/sample.txt | ch02/src/main/python/max_temperature_map.py
| sort | ch02/src/main/python/max_temperature_reduce.py
        1949 111
        1950 22
```

Hadoop Pipes

Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce. Hadoop Pipes uses standard input and output to communicate with the map and reduce code, Pipes uses sockets as the channel over which the tasktracker communicates with the process running the C++ map or reduce function.

Example 2-12 shows the source code for the map and reduce functions in C++.F
Example 2-12. Maximum temperature in C++

```
#include <algorithm>
#include <limits>
#include <stdint.h>
#include <string>
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"
class MaxTemperatureMapper :  public HadoopPipes::Mapper {
public:
MaxTemperatureMapper(HadoopPipes::TaskContext & context) {
}
void map(HadoopPipes::MapContext& context) {
std::string line = context.getInputValue();
std::string year = line.substr(15, 4);
std::string airTemperature = line.substr(87, 5);
std::string q = line.substr(92, 1);
if (airTemperature != "+9999" &&
(q == "0" || q == "1" || q == "4" || q == "5" || q == "9")) {
context.emit(year, airTemperature);
}
}
};
class MapTemperatureReducer : public HadoopPipes::Reducer {
public:
MapTemperatureReducer(HadoopPipes::TaskContext& context) {
}
void reduce(HadoopPipes::ReduceContext& context) {
int maxValue = INT_MIN;
while (context.nextValue()) {
maxValue = std::max(maxValue, HadoopUtils::toInt(context.getInputValue()));
}
context.emit(context.getInputKey(), HadoopUtils::toString(maxValue));
}
};
int main(int argc, char *argv[]) {
return
HadoopPipes::runTask(HadoopPipes::TemplateFactory<MaxTemperatureMapper,
MapTemperatureReducer>());
}
```

The full name of string is std::string because it resides in namespace std, the namespace in which all of the C++ standard library functions, classes, and objects reside.

Hadoop C++ library, which is a thin wrapper for communicating with the tasktracker child process. The map and reduce functions are defined by extending the Mapper and Reducer classes defined in the HadoopPipes namespace and providing implementations of the map() and reduce() methods in each case.

The map and reduce methods take a context object (of type MapContext or ReduceContext), which provides the reading input and writing output, as well as accessing job configuration information via the JobConf class.

Unlike the Java interface, keys and values in the C++ interface are byte buffers, represented as Standard Template Library (STL) strings.

In MapTemperatureReducer, we have to convert the input value into an integer (using a convenience method in HadoopUtils) and then the maximum value back into a string before it's written out.
In MaxTemperatureMapper where the airTemperature value is never converted to an integer since it is never processed as a number in the map() method.

The main() method is the application entry point. It calls HadoopPipes::runTask, which connects to the Java parent process and marshals data to and from the Mapper or Reducer. The runTask() method is passed a Factory so that it can create instances of the Mapper or Reducer. Which one it creates is controlled by the Java parent over the socket connection.

There are overloaded template factory methods for setting a combiner, partitioner, record reader, or record writer.

Compiling and Running

Now we can compile and link our program using the Makefile in Example 2-13.
Example 2-13. Makefile for C++ MapReduce program

```
CC = g++
CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include
max_temperature: max_temperature.cpp
$(CC) $(CPPFLAGS) $< -Wall -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib –
lhadooppipes \
-lhadooputils -lpthread -g -O2 -o $@
```

The Makefile expects a couple of environment variables to be set. Apart from HADOOP_INSTALL we need to define PLATFORM, which specifies the operating system, architecture, and data model (e.g., 32- or 64-bit). I ran it on a 32-bit Linux. system with the following:

```
% export PLATFORM=Linux-i386-32
% make
```

On successful completion, you'll find the max_temperature executable in the current directory. To run a Pipes job, we need to run Hadoop in *pseudo-distributed* mode.

Pipes doesn't run in standalone (local) mode, since it relies on Hadoop's distributed cache mechanism, which works only when HDFS is running. With the Hadoop daemons now running, the first step is to copy the executable to HDFS so that it can be picked up by tasktrackers when they launch map and reduce tasks:

```
% hadoop fs -put max_temperature bin/max_temperature
```

The sample data also needs to be copied from the local filesystem into HDFS:

```
% hadoop fs -put input/ncdc/sample.txt sample.txt
```

Now we can run the job. For this, we use the Hadoop pipes command, passing the URI
of the executable in HDFS using the -program argument:

```
% hadoop pipes \
-D hadoop.pipes.java.recordreader=true \
-D hadoop.pipes.java.recordwriter=true \
-input sample.txt \
-output output \
-program bin/max_temperature
```

The Hadoop Distributed Filesystem

- Filesystems that manage the storage across a network of machines are called distributed filesystems.

- Distributed filesystems more complex than regular disk filesystems because one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss.

- Hadoop comes with a distributed filesystem called HDFS, which stands for Hadoop Distributed Filesystem. HDFS also called "DFS"—informally or in older documentation or configurations.

The Design of HDFS:

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

Very large files: Files that are hundreds of megabytes, gigabytes,or terabytes in size. There are Hadoop clusters running today that store petabytes of data.2

Streaming data access: HDFS follows write-once, read-many-times pattern. A dataset is generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware: Commodity hardware Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors3)

Low-latency data access: Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS.HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low-latency access.

Lots of small files: Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware.

Multiple writers, arbitrary file modifications : Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

HDFS Concepts

Blocks:

Files in HDFS are broken into block-sized chunks, which are stored as independent units. Each block size 64 MB by default.

several benefits:

- ✓ No limitation on the file size.
- ✓ making the unit of abstraction a block rather than a file simplifies the storage subsystem.
- ✓ Blocks are easy to replicate between DataNodes.

HDFS's fsck command list the blocks that make up each file in the filesys.

```
% hadoop fsck / -files -blocks
```

Namenodes and Datanodes:

An HDFS cluster has two types of node operating in a master-worker pattern:

1. Namenode (the master) : manages the filesystem namespace
2. Datanodes (workers): Datanodes store and retrieve blocks.

Secondary namenode : It merge the namespace image with the edit log to prevent the edit log from becoming too large. It runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

HDFS Federation:

- The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.
- HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace.

  Example: one namenode might manage all the files rooted under /user
  second namenode might handle files under /share.

- Under federation, each namenode manages a namespace volume, which is made up of the metadata for the namespace, and a block pool containing all the blocks for the files in the namespace.
- Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.
- Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.
- To access a federated HDFS cluster, clients use client-side mount tables to map filepaths to namenodes. This is managed in configuration using the ViewFileSystem, and viewfs:// URIs.

HDFS High-Availability:

- The namenode is still a single point of failure (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. The whole Hadoop system would be out of service until a new namenode could be brought online.

- To recover from a failed namenode , an administrator starts a new primary namenode with one of the filesystem metadata replicas, and configures datanodes and clients to use this new namenode.

- The new namenode is not able to serve requests until it has i) loaded its namespace image into memory, ii) replayed its edit log, and iii) received enough block reports from the datanodes to leave safe mode.

- On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

- The 0.23 release series of Hadoop implementation given a pair of namenodes in an active-standby configuration. A few architectural changes are needed to allow this to happen:
    - ✓ namenodes must use highly-available shared storage to share the edit log.
    - ✓ Datanodes must send block reports to both namenodes.
    - ✓ Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.
- If the active namenode fails, then the standby can take over very quickly (in a few tens of seconds) since it has the latest state available in memory.

Failover and fencing:

- The transition from the active namenode to the standby is managed by a new entity in the system called the failover controller. Or The process in which system transfers its control to the standby when it detects a failure is known as **failover**.
- Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.
- Failover may also be initiated manually by an adminstrator, in the case of routine maintenance, for example. This is known as a graceful failover, since the failover controller an orderly transition for both namenodes to switch roles.
- Automatic Failover: It is the process in which system automatically transfers its control to the standby NameNode when the NameNode fails.
- The HA implementation make ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing.

- Fencing Methods: killing the namenode’s process, revoking its access to the shared storage directory and and disabling its network port via a remote management command.

- Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

Basic Filesystem Operations:

filesystem operations such as reading files, creating directories, moving files, deleting data, and listing directories.

Interfaces:

All Hadoop filesystem interactions are mediated through the Java API. Other filesystem interfaces.

- ✓ HTTP : There are two ways of accessing Accessing HDFS over HTTP: directly, where the HDFS daemons serve HTTP requests to clients; and via a proxy (or proxies), which accesses HDFS on the client's behalf using the usual DistributedFileSystem API.

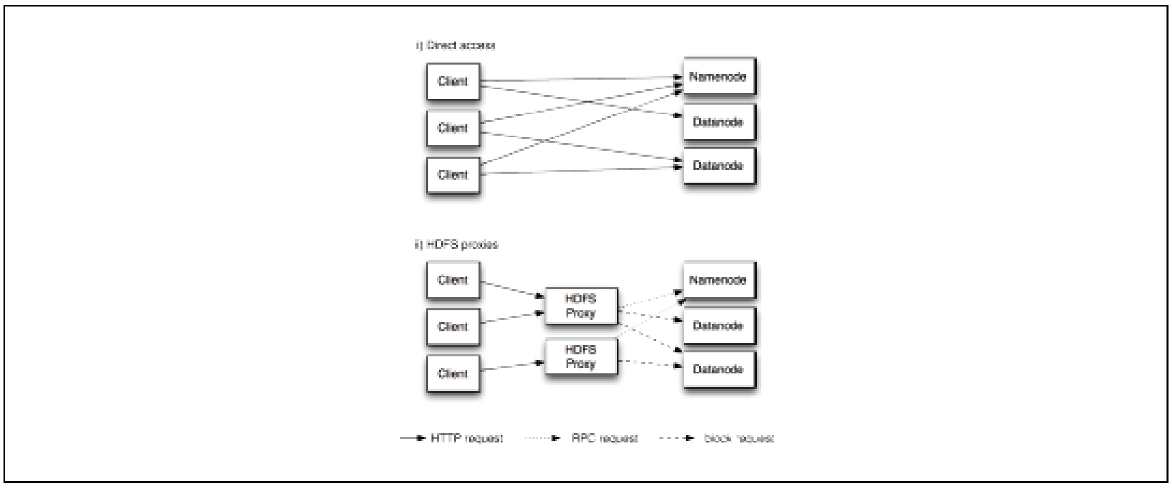


Figure 3-1. Accessing HDFS over HTTP directly, and via a bank of HDFS proxies

- ✓ C: Hadoop provides a C library called libhdfs that mirrors the Java FileSystem interface. It works using the Java Native Interface (JNI) to call a Java filesystem client.

- ✓ Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as a Unix filesystem.

.

Data Flow:

1. Anatomy of a File Read: shows how data flows between the client interacting with HDFS, the namenode and the datanodes.

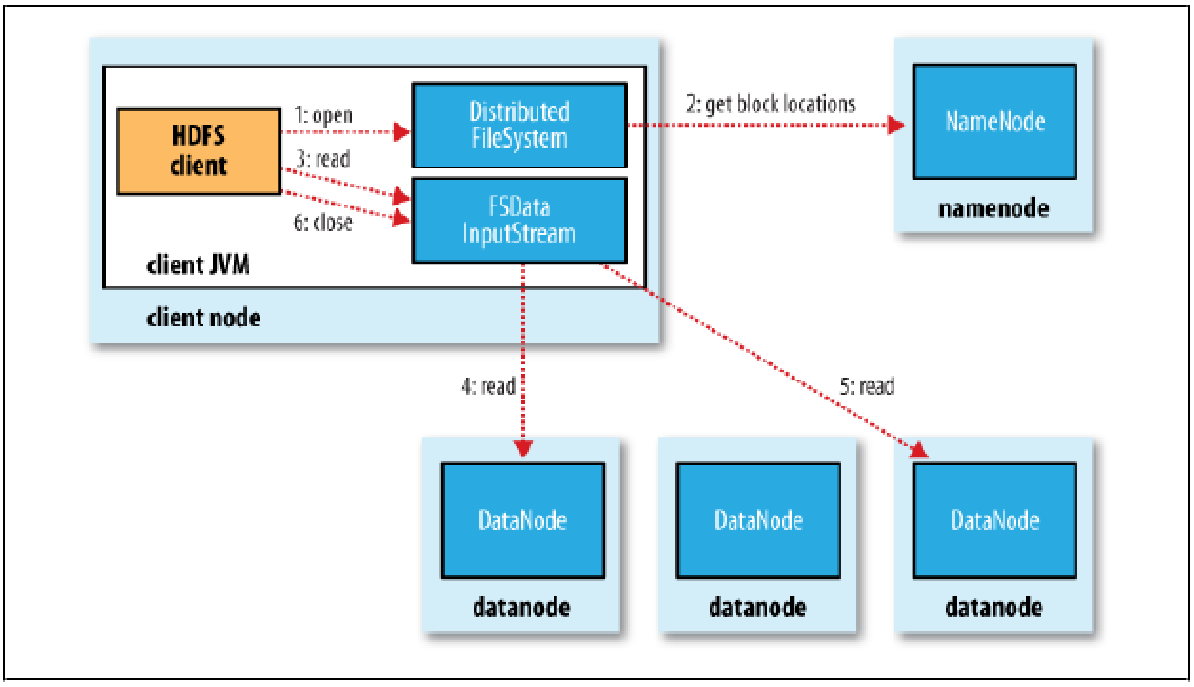


Figure 3-2. A client reading data from HDFS

Step1: client opens the file for read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem.
Step2: DistributedFileSystem calls the namenode, using RPC, to determine the locations ofthe blocks for the first few blocks in the file. For each block, the namenode returns the addresses of the datanodes that have a copy of that block. DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from.

Step3: The client then calls read() on the stream. DFSInputStream, which has stored the info node addresses for the primary few blocks within the file, then connects to the primary (closest) data node for the primary block in the file.

Step 4: Data is streamed from the data node back to the client, which calls read() repeatedly on the stream.

Step 5: When the end of the block is reached, DFSInputStream will close the connection to the data node, then finds the best data node for the next block.

Step 6: When the client has finished reading the file, a function is called, close() on the FSDataInputStream.

2. Anatomy of a File Write: shows how the client files are written to HDFS

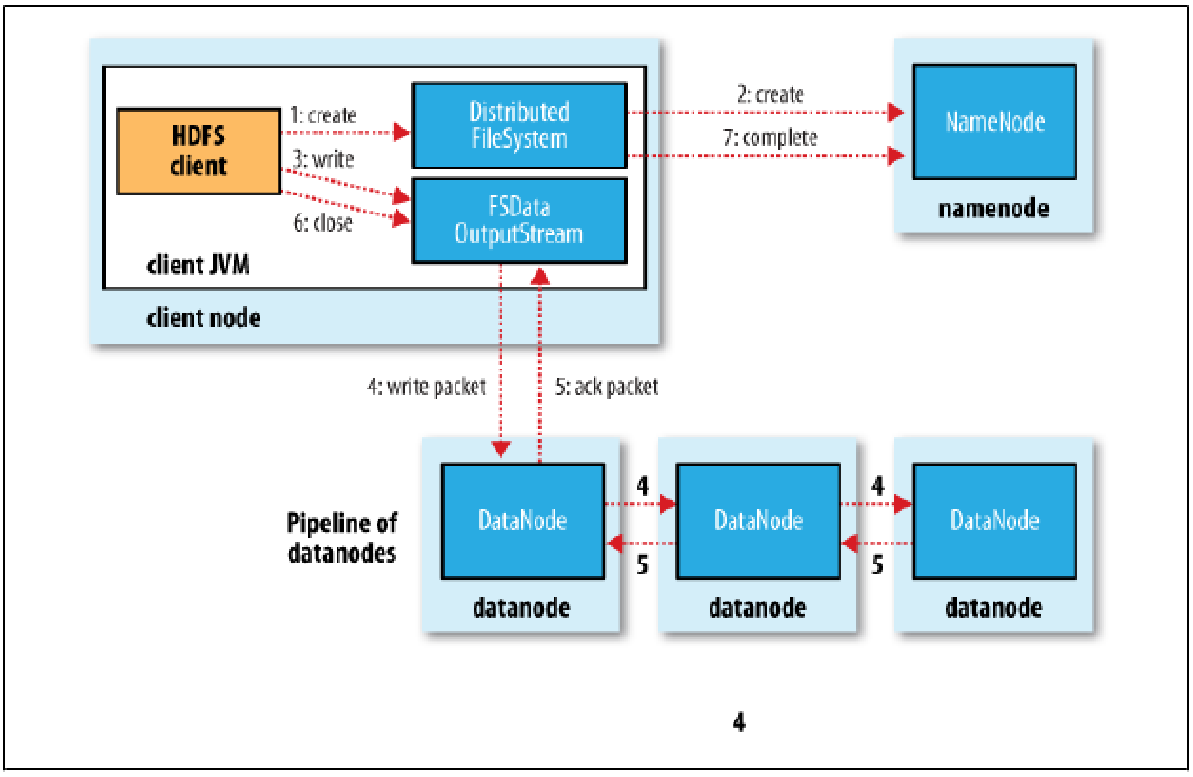


Figure 3-4. A client writing data to HDFS

Step 1: The client creates the file by calling create() on DistributedFileSystem(DFS).

Step 2: DFS makes an RPC call to the name node to create a new file in the file system's namespace, with no blocks associated with it. The name node performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the name node prepares a record of the new file; otherwise, the file can't be created and therefore the client is thrown an error i.e. IOException. The DFS returns an FSDataOutputStream for the client to start out writing data to.

Step 3: Because the client writes data, the DFSOutputStream splits it into packets, which it writes to an internal queue called the data queue. The data queue is consumed by the DataStreamer, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of data nodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first data node within the pipeline, which stores each packet and forwards it to the second data node within the pipeline.

Step 4: Similarly, the second data node stores the packet and forwards it to the third (and last) data node in the pipeline.

Step 5: A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline.

Step 6: This action sends up all the remaining packets to the data node pipeline and waits for acknowledgments before connecting to the name node to signal whether the file is complete or not.

Step-6 When the client has finished writing data, it calls close() on the stream.

If a datanode fails while data is being written to it, then the following actions are taken

- First the pipeline is closed.
- partial block on the failed datanode will be deleted
- failed datanode is removed from the pipeline
- namenode replica to be created on another node because default replication factor is 3.

3. <u>Coherency Model:</u>

A coherency model for a filesystem describes the data visibility of reads and writes for a file.

After creating a file, it is visible in the filesystem namespace. However, any content written to the file is not guaranteed to be visible, even if the stream is flushed. So, the file appears to have a length of zero. Once more than a block's worth of data has been written, the first block will be visible to new readers. This is true of subsequent blocks, too: it is always the current block being
written that is not visible to other readers.

HDFS provides a way to force all buffers to be flushed to the datanodes via the hflush() method on FSDataOutputStream. After a successful return from hflush(), HDFS guarantees that the data written up to that point in the file has reached all the datanodes in the write pipeline and is visible to all new readers.

4. <u>Parallel Copying with distcp:</u>

distcp is for transferring data between two HDFS clusters

```
% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

distcp will skip files that already exist in the destination, but they can be overwritten by supplying the -overwrite option.

```
% hadoop distcp -overwrite hdfs://namenode1/foo hdfs://namenode2/bar
```

update only files that have changed using the -update option

```
% hadoop distcp -update hdfs://namenode1/foo hdfs://namenode2/bar/foo
```

5. <u>Keeping an HDFS Cluster Balanced:</u>

HDFS works best when the file blocks are evenly spread across the cluster, to make ensure that distcp doesn't disrupt this.

it's not always possible to prevent a cluster from becoming unbalanced. We want to limit the number of maps so that some of the nodes can be used by other jobs. In this case, we can use the balancer tool.

6. <u>Using Hadoop Archives:</u>

A Hadoop Archive is created from a collection of files using the archive tool.

```
%hadoop archive -archiveName files.har   /my/files    /my
```

```
% hadoop fs -ls /my

% hadoop fs -ls /my/files.har
Found 3 items
-rw-r--r-- 10 tom supergroup 165 2009-04-09 19:13 /my/files.har/_index
-rw-r--r-- 10 tom supergroup 23 2009-04-09 19:13 /my/files.har/_masterindex
-rw-r--r-- 1 tom supergroup 2 2009-04-09 19:13 /my/files.har/part-0
```

part files contain the contents of a number of the original files concatenated together,
To lists the files in the archive:

```
% hadoop fs -lsr har:///my/files.har
```

To delete a HAR file: %

```
hadoop fs -rmr /my/files.har
```

Limitations:

- ✓ need as much disk space as the files for creating archive.
- ✓ currently no support for archive compression.
- ✓ Archives are immutable once they have been created.

Data Integrity:

- Users of Hadoop expect that no data will be lost or corrupted during storage or processing but every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing.
- when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.
- Detecting corrupted data is by computing a checksum for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data. The data is deemed to be corrupt if the newly generated checksum doesn't exactly match the original. This technique doesn't offer any way to fix the data—merely error detection. (And this is a reason for not using low-end hardware; in particular, be sure to use ECC memory.) Note that it is possible that it's the checksum that is corrupt, not the data, but this is very unlikely,since the checksum is much smaller than the data.
- A commonly used error-detecting code is CRC-32 (cyclic redundancy check), which computes a 32-bit integer checksum for input of any size.

1. <u>Data Integrity in HDFS:</u>

- HDFS transparently checksums all data written to it and by default verifies checksums when reading data. A separate checksum is created for every io.bytes.per.checksumbytes of data. The default is 512 bytes, and since a CRC-32 checksum is 4 bytes long, the storage overhead is less than 1%.
- Datanodes are responsible for verifying the data they receive before storing the data and its checksum.
- This applies to data that they receive from clients and from other datanodes during replication. A client writing data sends it to a pipeline of datanodes and the last datanode in the pipeline verifies the checksum.
- If it detects an error, the client receives a ChecksumException, a subclass of IOException.
- When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanode. Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified.
- When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.

- Each datanode runs a DataBlockScanner in a background thread that periodically verifies all the blocks stored on the datanode.

- HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a ChecksumException. The namenode marks the block replica as corrupt, so it doesn't direct clients to it, or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level. Once this has happened, the corrupt replica is deleted.
- Disable verification of checksums by passing false to the setVerify Checksum() method on FileSystem, before using the open() method to read a file.
- From the shell by using the -ignoreCrc option with the -get or-copyToLocal command.

2. LocalFileSystem:

- The Hadoop LocalFileSystem performs client-side checksumming. When write a file called filename, the filesystem client transparently creates a hidden file, .filename.crc, in the same directory containing the checksums for each chunk of the file.
- Like HDFS, the chunk size is controlled by the io.bytes.per.checksum property, which defaults to 512 bytes. The chunk size is stored as metadata in the .crc file, so the file can be read back correctly even if the setting for the chunk size has changed.
- Checksums are verified when the file is read, and if an error is detected, LocalFileSystem throws a ChecksumException.
- Checksums are fairly cheap to compute (in Java, they are implemented in native code), adding a few percent overhead to the time to read or write a file.
- For most applications, this is an acceptable price to pay for data integrity.
- Disable checksums: use RawLocalFileSystem in place of Local FileSystem. Create a Raw LocalFileSystem instance, to disable checksum verification for only some reads;

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs.initialize(null, conf);
```

3. ChecksumFileSystem

- LocalFileSystem uses ChecksumFileSystem to add checksumming to other (nonchecksummed) filesystems.
- ChecksumFileSystem is just a wrapper around FileSystem.

- General syntax as follows

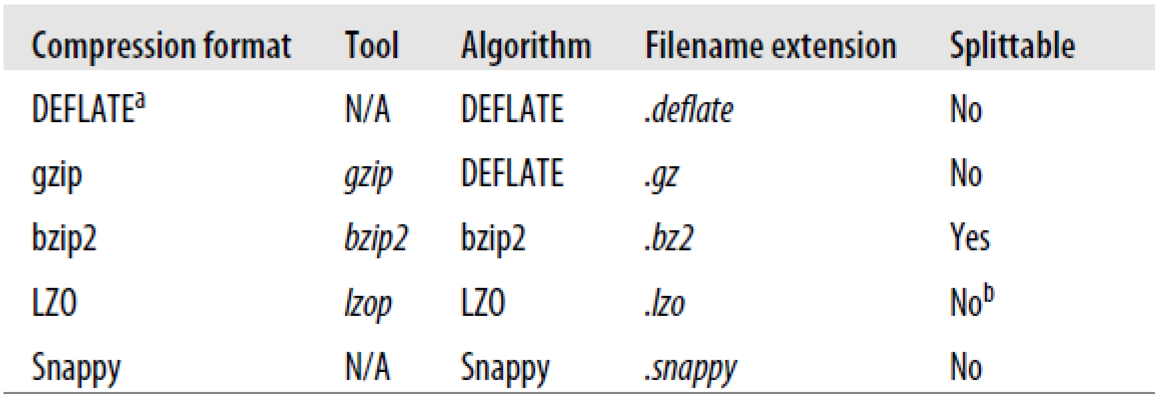
```
FileSystem rawFs = ...
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

- ChecksumFileSystem has few methods.
  getChecksumFile() for getting the path of a checksum file for any file.
  reportChecksumFailure() error is detected and reported error as bad file.
- Administrators should periodically check for these bad files and take action on them.

Compression:

- File compression brings two major benefits:
  it reduces the space needed to store files,
  it speeds up data transfer across the network, or to or from disk.
- Different compression formats, tools and algorithms, each with different characteristics.

Table 4-1. A summary of compression formats



Compression format	Tool	Algorithm	Filename extension	Splittable
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
Snappy	N/A	Snappy	*.snappy*	No

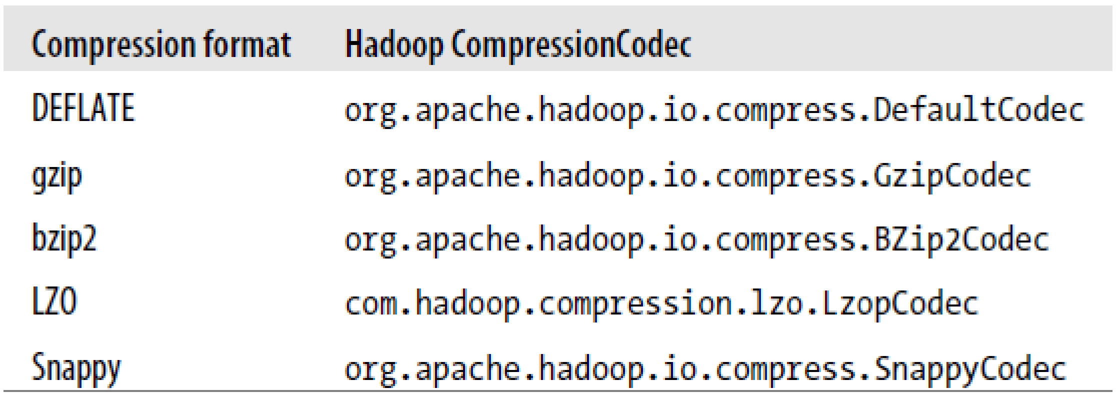
- All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings.
- Compression time by offering nine different options:
  –1 means optimize for speed and -9 means optimize for space.
- Example: gzip -1 file
  It creates a compressed file file.gz using the fastest compression method.
- Different tools have very different compression characteristics
  - ✓ Gzip is a generalpurpose compressor, and sits in the middle of the space/time trade-off.
  - ✓ Bzip2 compresses more effectively than gzip, but is slower. it's decompression speed is faster than its compression speed, but it is still slower than the other formats.

✓ LZO and Snappy both optimize for speed and are around an order of magnitude faster than gzip, but compress less effectively. Snappy is also significantly faster than LZO for decompression.

1. Codecs:

- A codec is the implementation of a compression-decompression algorithm.
- In Hadoop, a codec is represented by an implementation of the CompressionCodec interface.

Table 4-2. Hadoop compression codecs



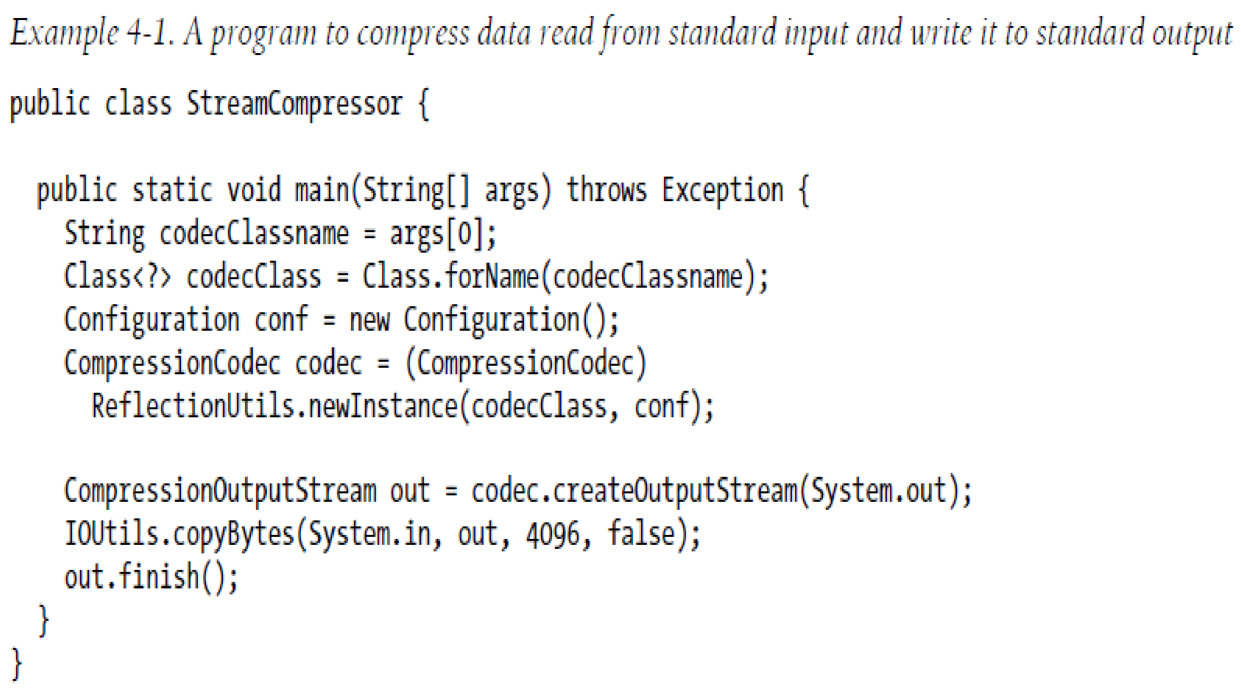
Compression format	Hadoop CompressionCodec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

- Example, GzipCodec encapsulates the compression and decompression algorithm for gzip.

Compressing and decompressing streams with CompressionCodec

CompressionCodec has two methods

1. createOutputStream(OutputStream out) method: create a CompressionOutputStream to write uncompressed data to written in compressed form to the underlying stream.
2. createInputStream(InputStream in) method: obtain a CompressionInputStream, which allows to read uncompressed data from the underlying stream.



Example 4-1. A program to compress data read from standard input and write it to standard output

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

- copyBytes() on IOUtils to copy the input to the output, which is compressed by the CompressionOutputStream.
- finish() on CompressionOutputStream, which tells the compressor to finish writing to the compressed stream, but doesn't close the stream.

```
%echo "Text" | hadoop StreamCompressor
              org.apache.hadoop.io.compress.GzipCodec \
              | gunzip -Text
```

<u>Inferring CompressionCodecs using CompressionCodecFactory:</u>

- If reading a compressed file, we can infer the codec to use by looking at its filename extension. A file ending in .gz can be read with GzipCodec, and so on.
- CompressionCodecFactory mapping a filename extension to a CompressionCodec using its getCodec() method, which takes a Path object for the file in question.

Example 4-2. A program to decompress a compressed file using a codec inferred from the file's extension

```
public class FileDecompressor {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path inputPath = new Path(uri);
    CompressionCodecFactory factory = new CompressionCodecFactory(conf);

    CompressionCodec codec = factory.getCodec(inputPath);
    if (codec == null) {
      System.err.println("No codec found for " + uri);
      System.exit(1);
    }

    String outputUri =
      CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());

    InputStream in = null;
    OutputStream out = null;
    try {
      in = codec.createInputStream(fs.open(inputPath));
      out = fs.create(new Path(outputUri));
      IOUtils.copyBytes(in, out, conf);
    } finally {
      IOUtils.closeStream(in);
      IOUtils.closeStream(out);
    }
  }
}
```

```
% hadoop FileDecompressor file.gz
```

Table 4-3. Compression codec properties

Property name	Type	Default value	Description
io.compression.codecs	comma-separated Class names	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.Bzip2Codec	A list of the CompressionCodec classes for compression/decompression.

Native libraries

- use a native library for compression and decompression because native gzip libraries reduced decompression times by up to 50% and compression times by around 10%.
- By default, Hadoop looks for native libraries for the platform it is running on, and loads them automatically if they are found.

Table 4-4. Compression library implementations

Compression format	Java implementation	Native implementation
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	No
LZO	No	Yes

- Disable use of native libraries by setting the property hadoop.native.lib to false.
- CodecPool: allows to reuse compressors and decompressors, thereby amortizing the cost of creating these objects.

Example 4-3. A program to compress data read from standard input and write it to standard output using a pooled compressor

```
public class PooledStreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);
    Compressor compressor = null;
    try {
      compressor = CodecPool.getCompressor(codec);
      CompressionOutputStream out =
        codec.createOutputStream(System.out, compressor);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    } finally {
      CodecPool.returnCompressor(compressor);
    }
  }
}
```

2. **Compression and Input Splits**:
   - Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 64 MB, the file will be stored as 16 blocks, and a MapReduce job using this file as input will create 16 input splits, each processed independently as input to a separate map task.
   - Gzip file does not support splitting because gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks.
   - Start of each block is not distinguished in any way that would allow a reader positioned at an arbitrary point in the stream to advance to the beginning of the next block, thereby synchronizing itself with the stream.
   - A single map will process the 16 HDFS blocks, most of which will not be local to the map. Also, with fewer maps, the job is less granular, and so may take longer to run.
   - LZO file does not provide a reader to synchronize itself with the stream but possible to preprocess LZO file using an indexer tool that comes with the Hadoop LZO libraries. The tool builds an index of split points, effectively making them splittable when the appropriate MapReduce input format is used.
   - A bzip2 file, does provide a synchronization marker between blocks, so it does support splitting.

3. **Using Compression in MapReduce:**
- If input files are compressed, they will be automatically decompressed as they are read by MapReduce, using the filename extension to determine the codec to use.
- To compress the output of a MapReduce job, in the job configuration, set the mapred.output.compress property to true and the mapred.output.compression.codec property to the classname of the compression codec you want to use.

```
public class MaxTemperatureWithCompression {

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

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);

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

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

    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

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

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

- Run the program over compressed input

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output
```

- Output is compressed

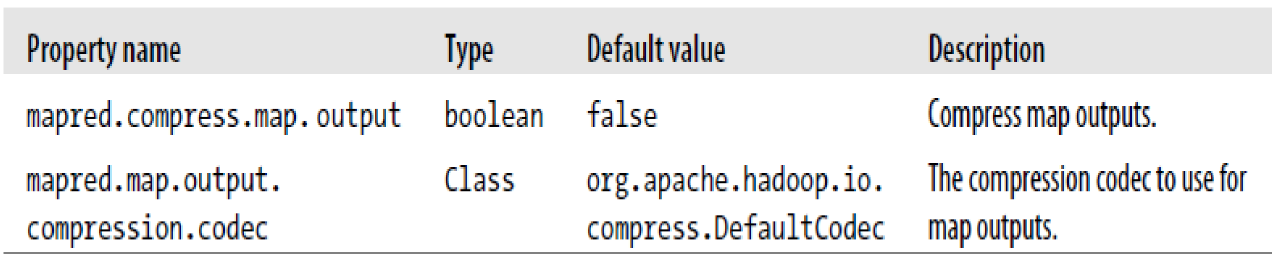
```
% gunzip -c output/part-r-00000.gz
1949 111
1950 22
```

Table 4-5. MapReduce compression properties

Property name	Type	Default value	Description
mapred.output.compress	boolean	false	Compress outputs.
mapred.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	The compression codec to use for outputs.
mapred.output.compression.type	String	RECORD	The type of compression to use for SequenceFile outputs: NONE, RECORD, or BLOCK.

- **Compressing map output**: map output is written to disk and transferred across the network to the reducer nodes, by using a fast compressor such as LZO or Snappy.

Table 4-6. Map output compression properties



Property name	Type	Default value	Description
mapred.compress.map.output	boolean	false	Compress map outputs.
mapred.map.output.compression.codec	Class	org.apache.hadoop.io.compress.DefaultCodec	The compression codec to use for map outputs.

- lines to add to enable gzip map output compression in your job:

```
Configuration conf = new Configuration();
conf.setBoolean("mapred.compress.map.output", true);
conf.setClass("mapred.map.output.compression.codec", GzipCodec.class,
CompressionCodec.class);
Job job = new Job(conf);
```

Serialization

- Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.
- Deserialization is the reverse process of turning a byte stream back into a series of structured objects.
- Serialization appears in two distinct areas of distributed data processing.
  - ✓ For interprocess communication.
  - ✓ For persistent storage.
- In Hadoop, interprocess communication between nodes in the system is implemented using remote procedure calls (RPCs) and RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.

- RPC serialization format is:
  - ✓ **Compact :** it makes the best use of network bandwidth.
  - ✓ **Fast:** IPC forms the backbone for a distributed system but little performance overhead as possible for the serialization and deserialization process.
  - ✓ **Extensible:** Protocols change over time to meet new requirements.it should be possible to add a new argument to a method call, and have the new servers accept messages in the old format (without the new argument) from old clients.
  - ✓ **Interoperable:** it is desirable to be able to support clients that are written in different languages to the server.

- Hadoop uses its own serialization format, Writables, which is certainly compact and fast, but not so easy to extend or use from languages other than Java.

The Writable Interface

The Writable interface defines two methods: one for writing its state to a DataOutput binary stream, and one for reading its state from a DataInput binary stream:

```
package org.apache.hadoop.io;
import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;
public interface Writable {
        void write(DataOutput out) throws IOException;
        void readFields(DataInput in) throws IOException;
}
```

IntWritable Class

This class implements Writable, Comparable, and WritableComparable interfaces. It wraps an integer data type in it. This class provides methods used to serialize and deserialize integer type of data.

Constructors

```
IntWritable()
IntWritable( int value)
```

Methods

- int get(): get the integer value present in the current object.
- void set(int value) : set the value of the current IntWritable object.
- void write(DataOutput out): serialize the data in the current object to the given DataOutput object.
- void readFields(DataInput **in) :** Deserialize the data in the given **DataInput** object.

Example: IntWritable writable = new IntWritable();

writable.set(163)   OR

IntWritable writable = new IntWritable(163);

Serialization in Hadoop:

```
public class Serialization {
        public  byte[] serialize() throws IOException {
                IntWritable writable = new IntWritable(12);
                ByteArrayOutputStream out = new ByteArrayOutputStream();
                DataOutputStream dataOut = new DataOutputStream(out);
                writable.write(dataOut);
                dataOut.close();
                return out.toByteArray();
        }
        public static void main(String args[]) throws IOException{
                Serialization serialization= new Serialization();
                 serialization.serialize();
                 System.out.println();
        }
}
```

Deserializing the Data in Hadoop

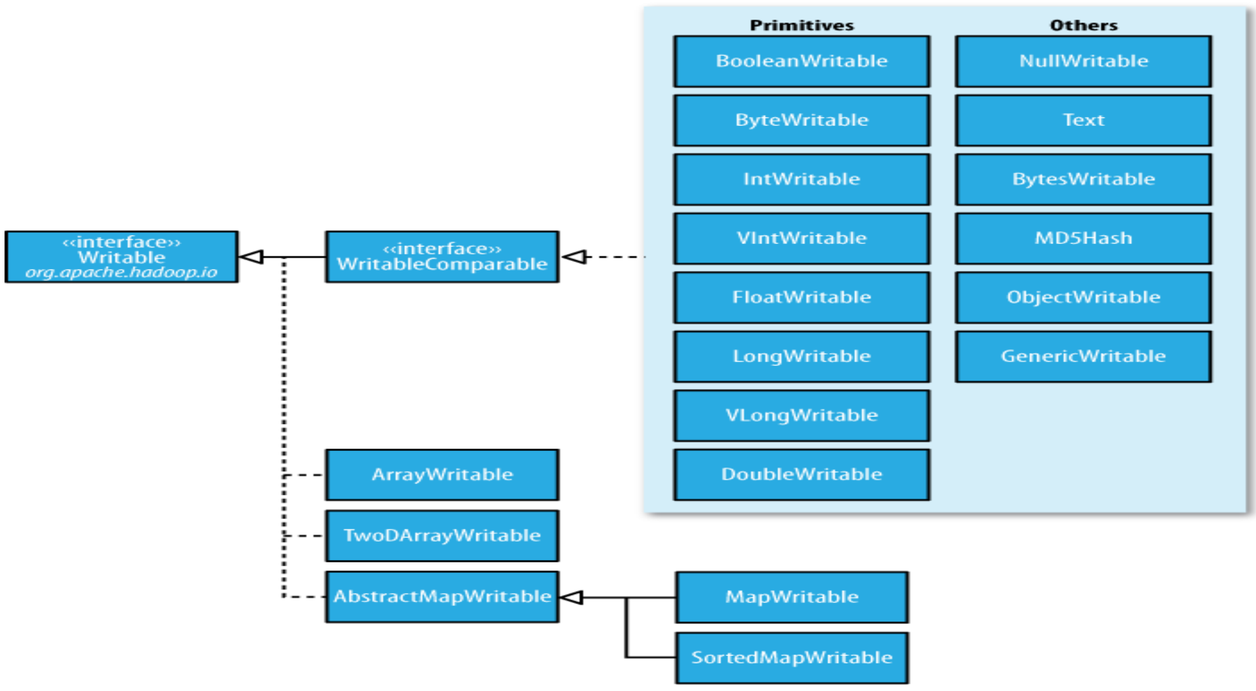
```
public class Deserialization {
        public  byte[] deserialize(byte[] bytes)
        throws IOException {
                IntWritable writable =new IntWritable();
                ByteArrayInputStream in = new ByteArrayInputStream(bytes);
                DataInputStream dataIn = new DataInputStream(in);
                writable.readFields(dataIn);
                dataIn.close();
                return bytes;
        }
        public static void main(String args[]) throws IOException{
            Serialization serialization= new Serialization();
            serialization.serialize();
            System.out.println();
          }

}
```

WritableComparable and comparators:

```
        package org.apache.hadoop.io;
        public interface WritableComparable<T> extends Writable, Comparable<T> {
        }
```

Writable Classes: org.apache.hadoop.io package



Serialization Frameworks:

- Serialization Framework is Implementation of Serialization (in the org.apache.hadoop.io.serializer package).

- WritableSerialization, for example, is the implementation of Serialization for Writable types.

- A Serialization defines a mapping from types to Serializer instances (object into a byte stream) and Deserializer instances (byte stream into an object).

Serialization IDL (an interface description language)

- Other serialization frameworks
  - ✓ Hadoop's own Record I/O generating types that are compatible with MapReduce.
  - ✓ Apache Thrift and Google Protocol Buffers used as a format for persistent binary data. There is limited support for these as MapReduce formats

File-Based Data Structures:

For some applications, we need a specialized data structure to hold your data.

1. SequenceFile: providing a persistent data structure for binary key-value pairs.

- choose a key, such as timestamp represented by a LongWritable, and the value is a Writable that represents the quantity being logged.

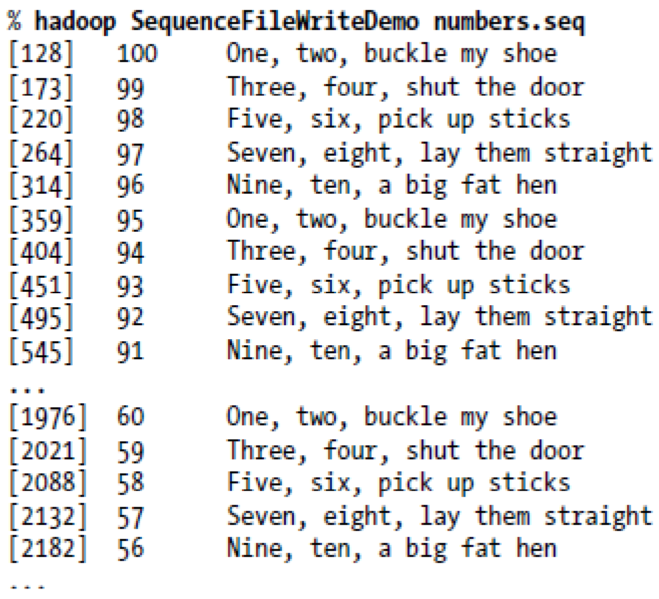
- SequenceFiles also work well as containers for smaller files. HDFS and MapReduce are optimized for large files, so packing files into a SequenceFile makes storing and processing the smaller files more efficient.

Writing a SequenceFile

- createWriter() method which returns a SequenceFile.Writer instance.
- append() method write key-value pairs.
- close() method to close the file.

Example 4-14. Writing a SequenceFile

```
public class SequenceFileWriteDemo {
        private static final String[] DATA = {  "One, two, buckle my shoe",
                                                 "Three, four, shut the door",
                                                 "Five, six, pick up sticks",
                                                 "Seven, eight, lay them straight",
                                                 "Nine, ten, a big fat hen"
                                              };
        public static void main(String[] args) throws IOException {
                String uri = args[0];
                Configuration conf = new Configuration();
                FileSystem fs = FileSystem.get(URI.create(uri), conf);
                Path path = new Path(uri);
                IntWritable key = new IntWritable();
                Text value = new Text();
                SequenceFile.Writer writer = null;
                try {
                writer = SequenceFile.createWriter(fs, conf, path,
                key.getClass(), value.getClass());
                for (int i = 0; i < 100; i++) {
                key.set(100 - i);
                value.set(DATA[i % DATA.length]);
                System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
                writer.append(key, value);
        }
}       finally {
                IOUtils.closeStream(writer);
                }
        }
}
```



```
% hadoop SequenceFileWriteDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
...
[1976]  60      One, two, buckle my shoe
[2021]  59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
```

Reading a SequenceFile

- creating an instance of SequenceFile.Reader
- next() methods: iterating over records

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

Example 4-15. Reading a SequenceFile

```
public class SequenceFileReadDemo {
	public static void main(String[] args) throws IOException {
		String uri = args[0];
		Configuration conf = new Configuration();
		FileSystem fs = FileSystem.get(URI.create(uri), conf);
		Path path = new Path(uri);
		SequenceFile.Reader reader = null;
		try {
			reader = new SequenceFile.Reader(fs, path, conf);
			Writable key = (Writable)
				ReflectionUtils.newInstance(reader.getKeyClass(), conf);
			Writable value = (Writable)
			       ReflectionUtils.newInstance(reader.getValueClass(), conf);
			long position = reader.getPosition();
			while (reader.next(key, value)) {
				String syncSeen = reader.syncSeen() ? "*" : "";
		System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
```

```
                position = reader.getPosition(); // beginning of next record
                }
        } finally {
                        IOUtils.closeStream(reader);
                   }
        }
```

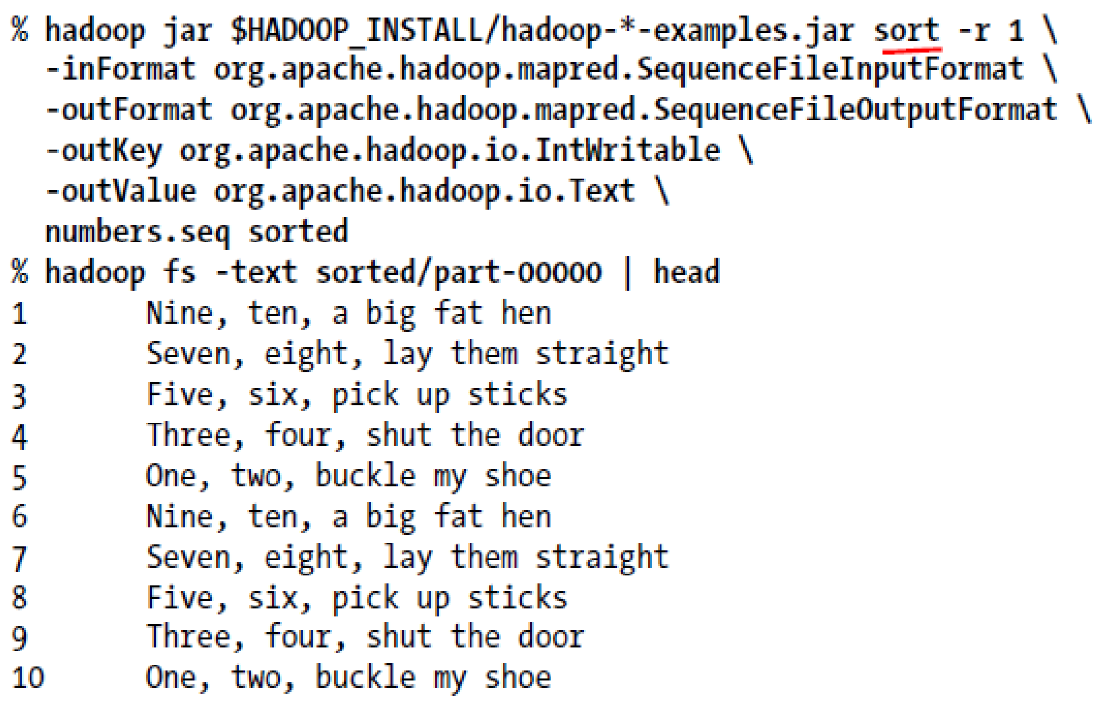
```
% hadoop SequenceFileReadDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
[590]   90      One, two, buckle my shoe
...
[1976]  60      One, two, buckle my shoe
[2021*] 59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
```

Another feature is displays the position of the sync points in the sequence file as asterisks.

Displaying a SequenceFile with the command-line interface.

```
% hadoop fs -text numbers.seq | head
100     One, two, buckle my shoe
99      Three, four, shut the door
98      Five, six, pick up sticks
97      Seven, eight, lay them straight
96      Nine, ten, a big fat hen
95      One, two, buckle my shoe
94      Three, four, shut the door
93      Five, six, pick up sticks
92      Seven, eight, lay them straight
91      Nine, ten, a big fat hen
```

Sorting and merging SequenceFiles: A SequenceFile.Sorter class that has a number of sort() and merge() methods. In general MapReduce is the preferred approach to sort and merge sequence files.



```
% hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort -r 1 \
  -inFormat org.apache.hadoop.mapred.SequenceFileInputFormat \
  -outFormat org.apache.hadoop.mapred.SequenceFileOutputFormat \
  -outKey org.apache.hadoop.io.IntWritable \
  -outValue org.apache.hadoop.io.Text \
  numbers.seq sorted
% hadoop fs -text sorted/part-00000 | head
1       Nine, ten, a big fat hen
2       Seven, eight, lay them straight
3       Five, six, pick up sticks
4       Three, four, shut the door
5       One, two, buckle my shoe
6       Nine, ten, a big fat hen
7       Seven, eight, lay them straight
8       Five, six, pick up sticks
9       Three, four, shut the door
10      One, two, buckle my shoe
```

The SequenceFile format :

- A sequence file consists of a header followed by one or more records.
- The first three bytes of a sequence file are the bytes SEQ, which acts a magic number,followed by a single byte representing the version number.
- The header contains other fields including the names of the key and value classes, compression details, userdefined metadata, and the sync marker.
- Sync marker is used to allow a reader to synchronize to a record boundary from any position in the file. Each file has a randomly generated sync marker, whose value is stored in the header. Sync markers appear between records in the sequence file. They are designed to incur less than a 1% storage overhead.
- Each record is made up of the record length (in bytes), the key length, the key, and then the value.
- Record compression value bytes are compressed using the codec defined in the header and keys are not compressed.

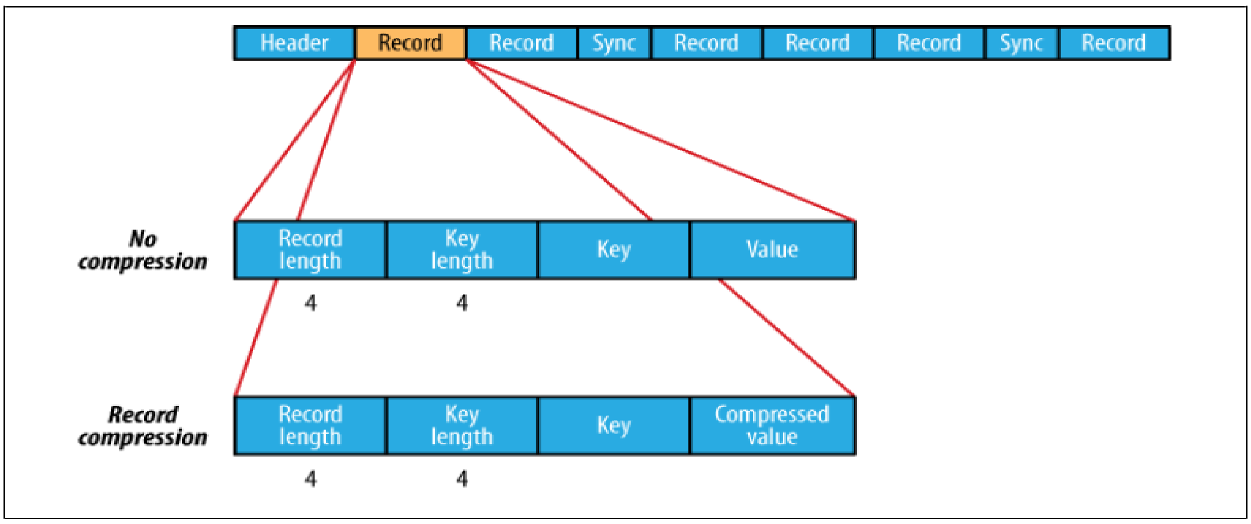


Figure 4-2. The internal structure of a sequence file with no compression and record compression

- Block compression compresses multiple records at once.

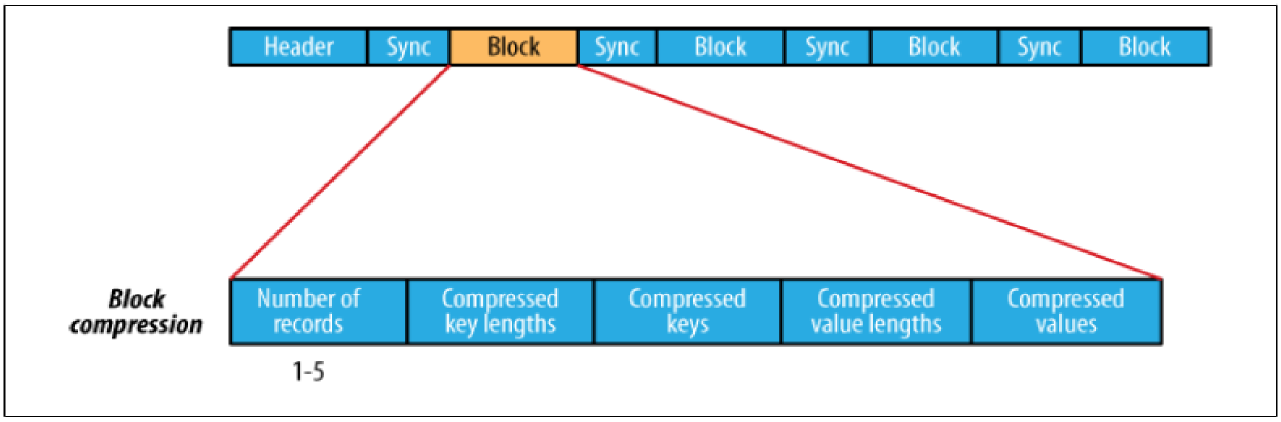


Figure 4-3. The internal structure of a sequence file with block compression

- Block compression is more compact than record compression.
- Records are added to a block until it reaches a minimum size in bytes, defined by the io.seqfile.compress.blocksize property: the default is 1 million bytes.
- A sync marker is written before the start of every block.
- The format of a block is a field indicating the number of records in the block, followed by four compressed fields: the key lengths, the keys, the value lengths, and the values.

MapFile

A MapFile is a sorted SequenceFile with an index to permit lookups by key.

Writing a MapFile

- create an instance of MapFile.Writer
- call the append() method to add entries in order.

- Keys must be instances of WritableComparable, and values must be Writable.

Example 4-16. Writing a MapFile

```
public class MapFileWriteDemo {
        private static final String[] DATA = {
                                                "One, two, buckle my shoe",
                                                "Three, four, shut the door",
                                                "Five, six, pick up sticks",
                                                "Seven, eight, lay them straight",
                                                "Nine, ten, a big fat hen"
                                             };
        public static void main(String[] args) throws IOException {
                String uri = args[0];
                Configuration conf = new Configuration();
                FileSystem fs = FileSystem.get(URI.create(uri), conf);
                IntWritable key = new IntWritable();
                Text value = new Text();
                MapFile.Writer writer = null;
                try {
                        writer = new MapFile.Writer(conf, fs, uri,
                                key.getClass(), value.getClass());
                        for (int i = 0; i < 1024; i++) {
                        key.set(i + 1);
                        value.set(DATA[i % DATA.length]);
                        writer.append(key, value);
                    }
                } finally {
                                IOUtils.closeStream(writer);
                }
        }
}
```

Run:

```
% hadoop MapFileWriteDemo numbers.map
```

MapFile containing two files called data and index:

```
% ls -l numbers.map
total 104
-rw-r--r-- 1 tom tom 47898 Jul 29 22:06 data
-rw-r--r-- 1 tom tom 251 Jul 29 22:06 inde
```

Both files are SequenceFiles. The data file contains all of the entries, in order:

```
% hadoop fs -text numbers.map/data | head
1	One, two, buckle my shoe
2	 Three, four, shut the door
3	Five, six, pick up sticks
4	 Seven, eight, lay them straight
5	 Nine, ten, a big fat hen
6	 One, two, buckle my shoe
7	Three, four, shut the door
8	 Five, six, pick up sticks
9	Seven, eight, lay them straight
10	Nine, ten, a big fat hen
```

The index file contains a fraction of the keys, and contains a mapping from the key to that key's offset in the data file:

```
% hadoop fs -text numbers.map/index
1	128
129	6079
257	12054
385	18030
513	24002
641	29976
769	35947
897	41922
```

Reading a MapFile

- create a MapFile.Reader
- call the next() method until it returns false, that means no entry was read because the end of the file was reached:

```
public boolean next(WritableComparable key, Writable val) throws IOException
```

A random access lookup can be performed by calling the get() method:

```
public Writable get(WritableComparable key, Writable val) throws IOException
```

- Find the key in the index that is less than or equal to the search key, 496. In this example, the index key found is 385, with value 18030, which is the offset in the data file. Next the reader seeks to this offset in the data file and reads entries until the key is greater than or equal to the search key,496. In this case, a match is found and the value is read from the data file.

```
        Text value = new Text();
        reader.get(new IntWritable(496), value);
        assertThat(value.toString(), is("One, two, buckle my shoe"));
```

MapFile variants:

Hadoop comes with a few variants on the general key-value MapFile interface:

- SetFile is a specialization of MapFile for storing a set of Writable keys. The keys must be added in sorted order.
- ArrayFile is a MapFile where the key is an integer representing the index of the element in the array, and the value is a Writable value.
- BloomMapFile is a MapFile which offers a fast version of the get() method, especially for sparsely populated files. The implementation uses a dynamic bloom filter for testing whether a given key is in the map. The test is very fast since it is in-memory, but it has a non-zero probability of false positives, in which case the regular get() method is called.

Converting a SequenceFile to a MapFile:

- sort a SequenceFile
- fix() method on MapFile, which re-creates the index for a MapFile.

Example 4-17. Re-creating the index for a MapFile

```
public class MapFileFixer {
    public static void main(String[] args) throws Exception {
        String mapUri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(mapUri), conf);
        Path map = new Path(mapUri);
        Path mapData = new Path(map, MapFile.DATA_FILE_NAME);
// Get key and value types from data sequence file
    SequenceFile.Reader reader = new SequenceFile.Reader(fs, mapData, conf);
        Class keyClass = reader.getKeyClass();
        Class valueClass = reader.getValueClass();
        reader.close();
        // Create the map file index file
        long entries = MapFile.fix(fs, map, keyClass, valueClass, false, conf);
    System.out.printf("Created MapFile %s with %d entries\n", map, entries);
    }
}
```

Output:

Step-1: Sort the sequence file numbers.seq into a new directory called number.map that will become the MapFile.

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort -r 1 \
-inFormat org.apache.hadoop.mapred.SequenceFileInputFormat \
-outFormat org.apache.hadoop.mapred.SequenceFileOutputFormat \
-outKey org.apache.hadoop.io.IntWritable \
-outValue org.apache.hadoop.io.Text \
numbers.seq numbers.map
```

Step-2: Rename the MapReduce output to be the data file:

```
% hadoop fs -mv numbers.map/part-00000 numbers.map/data
```

Step-3: Create the index file:

```
% hadoop MapFileFixer numbers.map
Created MapFile numbers.map with 100 entries
```

The MapFile numbers.map now exists and can be used.

Differences between MapFile and Sequence File.

Mapreduce jobs depends on the file types(input/intermediate).Depending on the types map reduce jobs can perform faster or slower.Intermediate data transfer may take more/less time depending on the file types,I/O,bandwidth etc may be effected.

HDFS doestn't differentiate between the file types.Hdfs can divide a huge file(map file/sequence file) into blocks in equally in efficient manner.

MapFile:

it is a directory and it has two files (index file(key for position and value for offset) and data file). Both of these files are in sequence file nature. Index file works as a lookup file. Index file is loaded into memory because it is small size then based on key and offset value, the data is retuned.

Sequence File:

Sequence files are containers that contain input data in key and value both are in binary formats.These files are best suited for transferring intermediate data from mapper nodes to other chained nodes(mapper/reducer). It is default file type natively supported by Hadoop.

It has many benefits. Disk space, I/O, bandwidth and splittable at record or block level.