Writing MapReduce Programs: A Weather Dataset, Understanding Hadoop API for MapReduce Framework (Old and New), Basic programs of Hadoop MapReduce: Driver code, Mapper code, Reducer code, RecordReader, Combiner, Partitioner

A Weather Dataset

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 semi structured 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.

Classic MapReduce (MapReduce 1)

A job run in classic MapReduce is illustrated in Figure -1. At the highest level, there are four independent entities:
• The client, which submits the MapReduce job.

• The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.

• The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.

• The distributed filesystem (normally HDFS, covered in Chapter 3), which is used for sharing job files between the other entities.

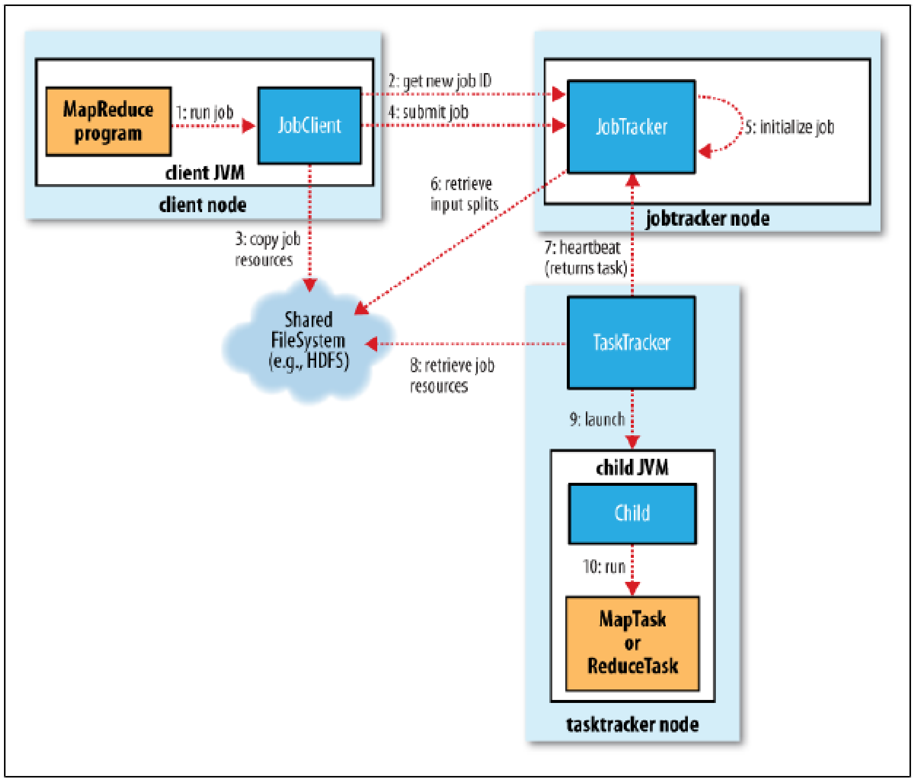

Figure 1. How Hadoop runs a MapReduce job using the classic framework

Job Submission

The submit() method on Job creates an internal JobSummitter instance and calls sub mitJobInternal() on it (step 1 in Figure 6-1). Having submitted the job, waitForCom pletion() polls the job's progress once a second and reports the progress to the console if it has changed since the last report.

The job submission process implemented by JobSummitter does the following:

• Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker) (step 2).

• Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.

• Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.

• Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).

• Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4).

Job Initialization

When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it (step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6). It then creates one map task for each split. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the Job, which is set by the setNumReduceTasks() method, and the scheduler simply creates this number of reduce tasks to be run.

In addition to the map and reduce tasks, two further tasks are created: a job setup task and a job cleanup task. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to cleanup after all the reduce tasks are complete.

Task Assignment

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of memory on the tasktracker.
Task Execution Now that the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk. Then, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Third, it creates an instance of TaskRunner to run the task. TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce functions don't affect the tasktracker.

Progress and Status Updates

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a status, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description.

When a task is running, it keeps track of its progress, that is, the proportion of the task completed. For map tasks, this is the proportion of the input that has been processed. For reduce tasks, it's a little more complex, but the system can still estimate the proportion of the reduce input processed.

Job Completion

When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to "successful." Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method. Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

For very large clusters in the region of 4000 nodes and higher, the MapReduce system described in the previous section begins to hit scalability bottlenecks, so in 2010 a group at Yahoo! began to design the next generation of MapReduce. The result was YARN, short for Yet Another Resource Negotiator.

YARN (MapReduce 2)

YARN meets the scalability shortcomings of "classic" MapReduce by splitting the responsibilities of the jobtracker into separate entities. The jobtracker takes care of both job scheduling (matching tasks with tasktrackers) and task progress monitoring (keeping track of tasks and restarting failed or slow tasks, and doing task bookkeeping such as maintaining counter totals).

YARN separates these two roles into two independent daemons:
I. Resource Manager
II. Application master

a resource manager to manage the use of resources across the cluster, and an application master to manage the lifecycle of applications running on the cluster.

The idea is that an application master negotiates with the resource manager for cluster resources—described in terms of a number of containers each with a certain memory limit—then runs applicationspecific processes in those containers. The containers are overseen by node managers running on cluster nodes, which ensure that the application does not use more resources than it has been allocated. In contrast to the jobtracker, each instance of an application—here a MapReduce job —has a dedicated application master, which runs for the duration of the application.
The beauty of YARN's design is that different YARN applications can co-exist on the same cluster—so a MapReduce application can run at the same time as an MPI application, for example—which brings great benefits for managability and cluster utilization. Furthermore, it is even possible for users to run different versions of MapReduce on the same YARN cluster, which makes the process of upgrading MapReduce more manageable.

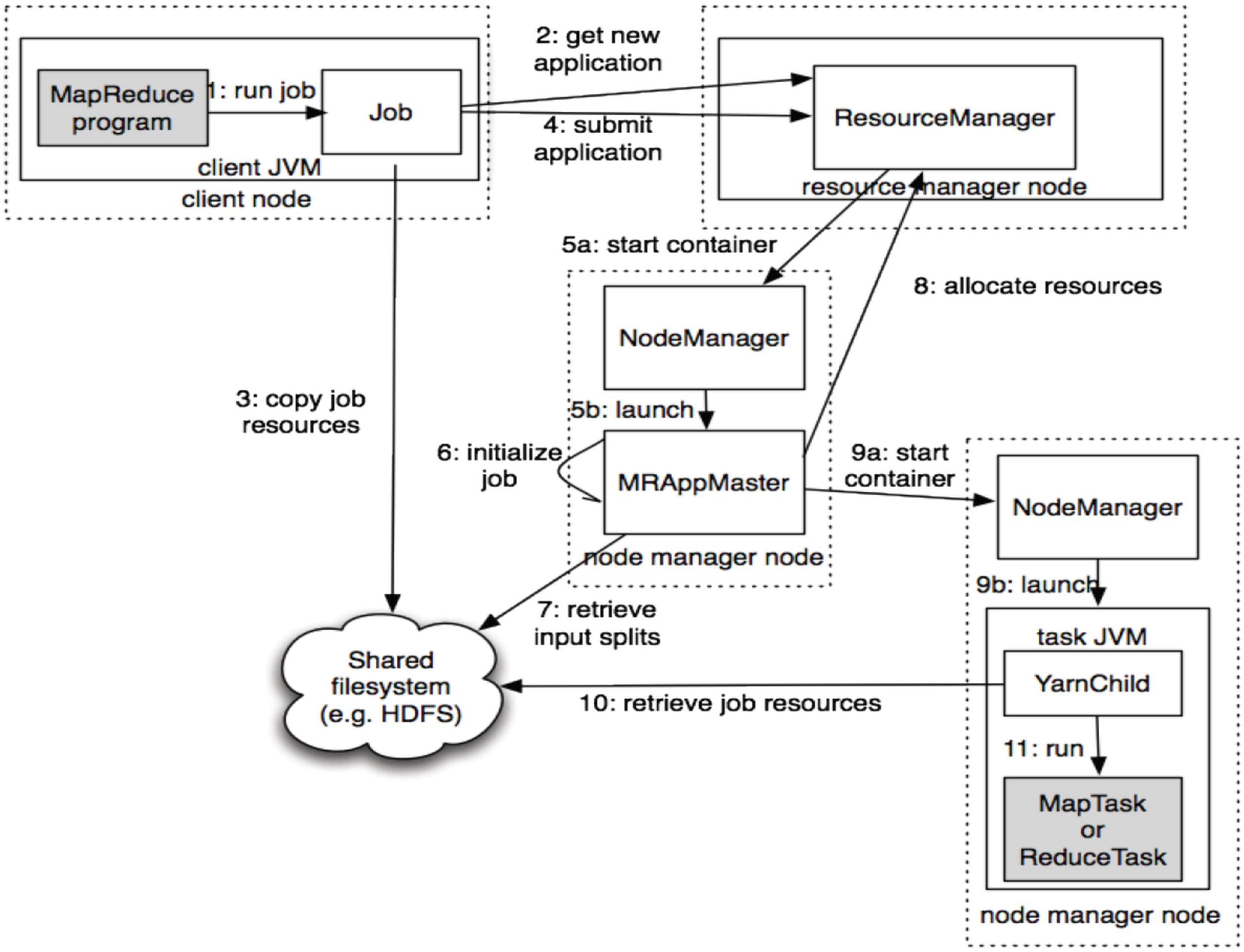

Figure 4. How Hadoop runs a MapReduce job using YARN

MapReduce on YARN involves more entities than classic MapReduce. They are:

• The client, which submits the MapReduce job.

• The YARN resource manager, which coordinates the allocation of compute resources on the cluster.

• The YARN node managers, which launch and monitor the compute containers on machines in the cluster.

• The MapReduce application master, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager, and managed by the node managers.

• The distributed filesystem (normally HDFS, covered in Chapter 3), which is used for sharing job files between the other entities. The process of running a job is shown in Figure 2, and described in the following sections.

Job Submission

Jobs are submitted in MapReduce 2 using the same user API as MapReduce 1 (step 1). MapReduce 2 has an implementation of ClientProtocol that is activated when mapreduce.framework.name is set to yarn. The submission process is very similar to the classic implementation. The new job ID is retrieved from the resource manager rather than the jobtracker.

Job Initialization

When the resource manager receives a call to its submitApplication(), it hands off the request to the scheduler. The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management (steps 5a and 5b).

The application master initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks (step 6). Next, it retrieves the input splits computed in the client from the shared filesystem (step 7). It then creates a map task object for each split, and a number of reduce task objects determined by the mapreduce.job.reduces property.

Task Assignment

The application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8). Each request, which are piggybacked on heartbeat calls, includes information about each map task's data locality, in particular the hosts and corresponding racks that the input split resides on.

Task Execution

Once a task has been assigned a container by the resource manager's scheduler, the application master starts the container by contacting the node manager (steps 9a and 9b). The task is executed by a Java application whose main class is YarnChild. Before it can run the task it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache (step 10). Finally, it runs the map or reduce task (step 11).

Understanding Hadoop API for MapReduce Framework (Old and New)

Hadoop provides two Java MapReduce APIs named as old and new respectively. There are several notable differences between the two APIs:

1. The new API favors abstract classes over interfaces, since these are easier to evolve. For example, you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class2. For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.

2. The new API is in the org.apache.hadoop.mapreduce package (and subpackages). The old API can still be found in org.apache.hadoop.mapred.

3. The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The new Context, for example, essentially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.

4. In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method. In the old API this is possible for mappers by writing a MapRunnable, but no equivalent exists for reducers.

5. Configuration has been unified. The old API has a special JobConf object for job configuration. In the new API, this distinction is dropped, so job configuration is done through a Configuration.

6. Job control is performed through the Job class in the new API, rather than the old JobClient, which no longer exists in the new API.
7. Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, while in the new API map outputs are named partm-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integerdesignating the part number, starting from zero).

8. In the new API the reduce() method passes values as a java.lang.Iterable, rather than a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java's for-each loop construct: for (VALUEIN value : values) { ...}

<u>Basic programs of Hadoop MapReduce:</u>

<u>*Driver code*</u>

A Job object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the Job's setJarByClass() method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static addInputPath() method on FileInputFormat, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern.
The output path (of which there is only one) is specified by the static setOutputPath() method on FileOutputFormat. It specifies a directory where the output files from the reducer functions are written.

Next, we specify the map and reduce types to use via the setMapperClass() and setReducerClass() methods.

The setOutputKeyClass() and setOutputValueClass() methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, then the map output types can be set using the methods
setMapOutputKeyClass() and setMapOutputValueClass().

The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat. After setting the classes that define the map and reduce functions, we are ready to run the job. The waitForCompletion() method on Job submits the job and waits for it to finish.

The return value of the waitForCompletion() method is a boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1. The driver code for weather program is specified below.

```
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);
}
}
```

Mapper code

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. For the present example, the 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 (an integer). The following example shows the implementation of our map method.

```
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) == '+')
 {
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 map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in.

The map() method also provides an instance of Context to write the output to. In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable. We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

Reducer code

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

```
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));
}
}
```

Record Reader

RecordReader is responsible for creating key / value pair which has been fed to Map task to process. Each InputFormat has to provide its own RecordReader implementation to generate key / value pairs.
For example, the default TextInputFormat provides LineRecordReader which generates byte offset of the file as key and n separated line in the input file as value.

Combiner code

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. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer. The combiner function doesn't replace the reduce function. But it can help cut down the amount of data shuffled between the maps and reduces.

```
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);
}
}
```

Partitioner code

The partitioning phase takes place after the map phase and before the reduce phase. The number of partitions is equal to the number of reducers. The data gets partitioned across the reducers according to the partitioning function. The difference between a partitioner and a combiner is that the partitioner divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, the combiner functions similar to the reducer and processes the data in each partition. The combiner is an optimization to the reducer. The default partitioning function is the hash partitioning function where the hashing is done on the key. However it might be useful to partition the data according to some other function of the key or the value.