Chapter 6. Hadoop MapReduce

You write MapReduce jobs in Java when you need low-level control and want to optimize or streamline your big data pipeline. Using MapReduce is not required, but it is rewarding, because it is a beautifully designed system and API. Learning the basics can get you very far, very quickly, but before you embark on writing a customized MapReduce job, don't overlook the fact that tools such as Apache Drill enable you to write standard SQL queries on Hadoop.

This chapter assumes you have a running Hadoop Distributed File System (HDFS) on your local machine or have access to a Hadoop cluster. To simulate how a real MapReduce job would run, we can run Hadoop in pseudodistributed mode on one node, either your localhost or a remote machine. Considering how much CPU, RAM, and storage resources we can fit on one box (laptop) these days, you can, in essence, create a mini supercomputer capable of running fairly massive distributed jobs. You can get pretty far on your localhost (on a subset of data) and then scale up to a full cluster when your application is ready.

If the Hadoop client is properly installed, you can get a complete listing of all available Hadoop operations by simply typing the following:

```
bash$ hadoop
```

Hadoop Distributed File System

Apache Hadoop comes with a command-line tool useful for accessing the Hadoop filesystem and launching MapReduce jobs. The filesystem access command `fs` is invoked as follows:

```
bash$ hadoop fs <command> <args>
```

The command is any number of standard Unix filesystem commands such as `ls`, `cd`, or `mkdir` preceded by a hyphen. For example, to list all the items in the HDFS root directory, type this:

```
bash$ hadoop fs -ls /
```

Note the inclusion of the `/` for root. If it were not included at all, the command would return nothing and might fool you into thinking that your HDFS is empty! Typing `hadoop fs` will print out all the filesystem operations available. Some of the more useful operations involve copying data to and from HDFS, deleting directories, and merging data in a directory.

To copy local files into a Hadoop filesystem:

```
bash$ hadoop fs -copyFromLocal <localSrc> <dest>
```

To copy a file from HDFS to your local drive:

```
bash$ hadoop fs -copyToLocal <hdfsSrc> <localDest>
```

After a MapReduce job, there will most likely be many files contained in the output directory of the job. Instead of retrieving these one by one, Hadoop has a convenient operation for merging the files into one and then storing the results locally:

```
bash$ hadoop fs -getmerge <hdfs_output_dir> <my_local_dir>
```

One essential operation for running MapReduce jobs is to first remove the output directory if it already exists, because MapReduce will fail, almost immediately, if it detects the output directory:

```
bash$ hadoop fs -rm rf <hdfs_dir>
```

MapReduce Architecture

MapReduce invokes the *embarrassingly parallel* paradigm of distributed computing. Initially, the data is broken into chunks, and portions are sent to identical mapper classes that extract key-value pairs from the data, line by line. The key-value pairs are then partitioned into key-list pairs where the lists are sorted. Typically, the number of partitions is the number of reduce jobs, but this is not required. In fact, multiple key-list groups can be in the same partition and reducer, but each key-list group is guaranteed not to be split across partitions or reducers. The general flow of data through a MapReduce framework is displayed in Figure 6-1.

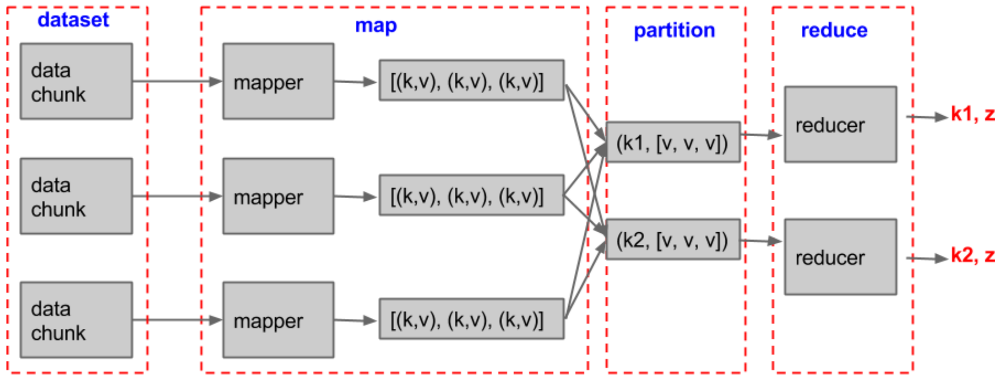

Figure 6-1. MapReduce schema

Say, for example, we have data like this:

```
San Francisco, 2012
New York, 2012
San Francisco, 2017
New York, 2015
New York, 2016
```

The mapper could output key-value pairs such as (San Francisco, 2012) for each line in the dataset. Then the partitioner would collect the data by key and sort the list of values:

```
(San Francisco, [2012, 2017])
(New York, [2012, 2015, 2016])
```

We could designate the reducer's function to output the maximum year such that the final output (written to the output directory) would look like this:

```
San Francisco, 2017
New York, 2016
```

It is important to consider that the Hadoop MapReduce API allows compound keys and customizable comparators for partitioning keys and sorting values.

Writing MapReduce Applications

Although there are more than a few ways to store and shuttle around data in the Hadoop ecosystem, we will focus on plain old text files. Whether the underlying data is stored as a string, CSV, TSV, or JSON data string, we easily read, share, and manipulate the data. Hadoop also provides resources for reading and writing its own Sequence and Map file formats, and you may want to explore various third-party serialization formats such as Apache Avro, Apache Thrift, Google Protobuf, Apache Parquet, and others. All of these provide operational and efficiency advantages. However, they do add a layer of complexity that you must consider.

Anatomy of a MapReduce Job

A basic MapReduce job has just a few essential features. The main guts of the overriden `run()` method contain a singleton instance of the `Job` class:

```
public class BasicMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new BasicMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {

        Job job = Job.getInstance(getConf());
        job.setJarByClass(BasicMapReduceExample.class);
        job.setJobName("BasicMapReduceExample");

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

        return job.waitForCompletion(true) ? 0 : 1;
    }

}
```

Note that because we have not defined any `Mapper` or `Reducer` classes, this job will use the default classes that copy the input files to the output directory, unchanged. Before we delve into customizing `Mapper` and `Reducer` classes, we must first understand the exclusive data types that are required by Hadoop MapReduce.

Hadoop Data Types

Data must be shuttled around through the MapReduce universe in a format that is both reliable and efficient. Unfortunately (according to the authors of Hadoop), the native Java primitive types (e.g., `boolean`, `int`, `double`) and the more complex types (e.g., `String`, `Map`) do not travel well! For this reason, the Hadoop ecosystem has its own version of serializable types that are required in all MapReduce applications. Note that all the regular Java types are perfectly fine inside our MapReduce code. It is only for the connections between MapReduce components (between mapper and reducers) where we need to convert native Java types to Hadoop types.

WRITABLE AND WRITABLECOMPARABLE TYPES

The Java primitive types are all represented, but the most useful ones are `BooleanWritable`, `IntWritable`, `LongWritable`, and `DoubleWritable`. A Java `String` type is represented by `Text`. A null is `NullWritable`, which comes in handy when we have no data to pass through a particular key or value in a MapReduce job. There is even an `MD5Hash` type, which could be used, among other things, when we are using a key that is a hash corresponding to `userid` or some other unique identifier. There is also a

`MapWritable` for creating a `Writable` comparable `HashMap`. All of these types are comparable (e.g., they have `hash()` and `equals()` methods that enable comparison and sorting events in the MapReduce job). Of course, there are more types, but these are a few of the more useful ones. In general, a Hadoop type takes the Java primitive as an argument in the constructor:

```
Int count = 42;
IntWritable countWritable = new IntWritable(count);

String data = "The is a test string";
Text text = new Text(data);
```

Note that Java types are used inside your code for `Mapper` and `Reducer` classes. *Only* the key and value inputs and outputs for those instances must use the Hadoop writable (and writable comparable if a key) types, because this is how data is shuttled between the MapReduce components.

CUSTOM WRITABLE AND WRITABLECOMPARABLE TYPES

At times we need a specialized type not covered by Hadoop. In general, a Hadoop type must implement `Writable`, which handles the object's serialization with a `write()` method and deserialization with a `read()` method. However, if the object will be used as a key, it must implement `WritableComparable`, because the `compareTo()` and `hashCode()` methods will be required during partitioning and sorting.

Writable

Because the `Writable` interface has only two methods, `write()` and `readFields()`, a basic custom writable needs to override only these methods. However, we can add a constructor that takes arguments so that we can instantiate the object in the same way we created `IntWritable` and `Text` instances in the previous example. In addition, if we add a static `read()` method, we will require a no-argument constructor:

```
public class CustomWritable implements Writable {

    private int id;
    private long timestamp;

    public CustomWritable() {
    }

    public CustomWritable(int id, long timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    public void write(DataOutput out) throws IOException {
        out.writeInt(id);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
        timestamp = in.readLong();
    }

    public static CustomWritable read(DataInput in) throws IOException {
        CustomWritable w = new CustomWritable();
        w.readFields(in);
        return w;
    }
}
```

WritableComparable

If our custom writable will be used a key, we will need `hashCode()` and `compareTo()` methods in addition to the `write()` and `readField()` methods:

```
public class CustomWritableComparable implements WritableComparable {

    private int id;
    private long timestamp;

    public CustomWritable() {
    }

    public CustomWritable(int id, long timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    public void write(DataOutput out) throws IOException {
        out.writeInt(id);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
        timestamp = in.readLong();
    }

    public int compareTo(CustomWritableComparable o) {
        int thisValue = this.value;
        int thatValue = o.value;
        return (thisValue < thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
    }

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + (int) (timestamp ^ (timestamp >>> 32));
        return result
    }
}
```

Mappers

The `Mapper` class is what maps the raw input data into a new and typically smaller sized data structure. In general, you do not need every piece of data from each line of the input files, but rather a select few items. In some cases, the line may be discarded entirely. This is your chance to decide what data will go into the next round of processing. Think of this step as a way of transforming and filtering the raw data into only the parts we actually need. If you do not include a `Mapper` instance in a MapReduce job, the `IdentityMapper` will be assumed, which just passes all the data directly through to the reducer. And if there is no reducer, input will be essentially copied to output.

GENERIC MAPPERS

Several common mappers that are already included with Hadoop can be designated in the MapReduce job. The default is the `IdentityMapper`, which outputs the exact data it inputs. The `InverseMapper` switches the key and value. There is also `TokenCounterMapper`, which outputs each token and its count as a `Text`, `IntWritable` key-value pair. The `RegexMapper` outputs a regex match as the key and constant value of 1. If none of these work for your application, consider writing your own customized mapper instance.

CUSTOMIZING A MAPPER

Parsing text files within a `Mapper` class is much the same as parsing lines from a regular text file, as in Chapter 1. The only required method is the `map()` method. The fundamental purpose of this `map()` method is to parse one line of input and output a key-value pair via the `context.write()` method:

```
public class ProductMapper extends
    Mapper<LongWritable, Text, IntWritable, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
```

```
            throws IOException, InterruptedException {
            try {

                /* each line of file is <userID>, <productID>, <timestamp> */

                String[] items = value.toString().split(",");
                int userID = Integer.parseInt(items[0]);
                String productID = items[1];
                context.write(new IntWritable(userID), new Text(productID));

            } catch (NumberFormatException | IOException | InterruptedException e) {

                context.getCounter("mapperErrors", e.getMessage()).increment(1L);
            }
        }

    }
```

There are also `startup()` and `cleanup()` methods. The `startup()` method is run once when the `Mapper` class is instantiated. You probably won't need it, but it comes in handy, for example, when you need a data structure that needs to be used by each call to the `map()` method. Likewise, you probably won't need the `cleanup()` method, but it is called once after the last call to `map()` and is used to do any cleanup actions. There is also a `run()` method, which does the actual business of mapping the data. There is no real reason to override this method, and it's best to leave it alone unless you have a good reason to implement your own `run()` method. In "MapReduce Examples", we show how to utilize the `setup()` method for some unique computations.

To use a custom mapper, you must designate it in the MapReduce application and set the map output key and value types:

```
job.setMapperClass(ProductMapper.class);
job.setMapOutputKeyClass(IntWritable.class);
job.setMapOutputValueClass(Text.class);
```

Reducers

The role of the `Reducer` is to iterate over the list of values associated with a key and calculate a singular output value. Of course, we can customize the output type of the `Reducer` to return anything we would like as long as it implements `Writable`. It is important to note that each reducer will process at least one key and all its values, so you do not need to worry that some values belonging to a key have been sent somewhere else. The number of reducers is also the number of output files.

GENERIC REDUCERS

If a `Reducer` instance is not specified, the MapReduce job sends mapped data directly to the output. There are some useful reducers in the Hadoop library that come in handy. The `IntSumReducer` and `LongSumReducer` take respective `IntWritable` and `LongWritable` integers as values in the `reduce()` method. The outputs are then the sum of all the values. Counting is such a common use case for MapReduce that these classes are purely convenient.

CUSTOMIZING A REDUCER

The code for a reducer has a similar structure to the mapper. We usually need to override only `reduce()` with our own code, and on occasion we'll use the `setup()` method when we are building a specific data structure or file-based resource that must be utilized by all reducers. Note that the reducer signature takes an `Iterable` of the value type because after the mapper phase, all the data for a particular key is grouped and sorted into a list (`Iterable`) and input here:

```
public class CustomReducer extends
    Reducer<IntWritable, Text, IntWritable, IntWritable>{

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

        int someValue = 0;

        /* iterate over the values and do something */
        for (Text value : values) {
            // use value to augment someValue
        }

        context.write(key, new IntWritable(someValue));

    }
```

The `Reducer` class and its key and value output types need to be specified in the MapReduce job:

```
job.setReducerClass(CustomReducer.class);
job.setOutputKeyClass(IntWritable.class);
job.setOutputValueClass(IntWritable.class);
```

The Simplicity of a JSON String as Text

JSON data (where each row of a file is a separate JSON string) is everywhere, and for good reason. Many tools are capable of ingesting JSON data, and its human readability and built-in schema are really helpful. In the MapReduce world, using JSON data as input data eliminates the need for custom writables because the JSON string can be serialized in the Hadoop `Text` type. This process can be as simple as using `JSONObject` right in the `map()` method. Or you can create a class to consume the `value.toString()` for more complicated mapping schemas.

```
public class JSONMapper extends Mapper<LongWritable, Text, Text, Text> {

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

        JSONParser parser = new JSONParser();
        try {
            JSONObject obj = (JSONObject) parser.parse(value.toString());

            // get what you need from this object
            String userID = obj.get("user_id").toString();
            String productID = obj.get("product_id").toString();
            int numUnits = Integer.parseInt(obj.get("num_units").toString());

            JSONObject output = new JSONObject();
            output.put("productID", productID);
            output.put("numUnits", numUnits);

            /* many more key value pairs, including arrays, can be added here */

            context.write(new Text(userID), new Text(output.toString()));


        } catch (ParseException ex) {
            //error parsing json
        }

    }
}
```

This also works great for outputting the data from the final reducer as a `Text` object. The final data file will be in JSON data format to enable efficient use down the rest of your pipeline. Now the reducer can input a `Text`, `Text` key-value pair and process the JSON with `JSONObject`. The advantage is that we did not have to create a complicated custom `WritableComparable` for this data structure.

Deployment Wizardry

There are many options and command-line switches for running a MapReduce job. Remember that before you run a job, the output directory needs to be deleted first:

```
bash$ hadoop fs -rm -r <path>/output
```

RUNNING A STANDALONE PROGRAM

You will certainly see (and probably write yourself) one file that contains the entire MapReduce job. The only real difference is that you must define any custom `Mapper`, `Reducer`, `Writable`, and so forth, as static. Otherwise, the mechanics are all the same. The obvious advantage is that you have a completely self-contained job without any worry of dependencies, and as such, you don't have to worry about JARs, and so forth. Just build the Java file (at the command line with `javac`) and run the class like this:

```
bash$ hadoop BasicMapReduceExample input output
```

DEPLOYING A JAR APPLICATION

If the MapReduce job is part of a larger project that has become a JAR possibly containing many such jobs, you will need to deploy from the JAR and designate the full URI of the job:

```
hadoop jar MyApp.jar com.datascience.BasicMapReduceExample input output
```

INCLUDING DEPENDENCIES

Include a comma-separated list of files that must be used in the MapReduce job as follows:

```
-files file.dat, otherFile.txt, myDat.json
```

Any JARs required can be added with a comma-separated list:

```
-libjars myJar.jar, yourJar.jar, math.jar
```

Note that command-line switches such as `-files` and `-libjars` must be placed before any command arguments such as input and output.

SIMPLIFYING WITH A BASH SCRIPT

At some point, typing all this text in the command line is error prone and laborious. So is scrolling through your bash history to find that command you launched last week. You can create custom scripts for specific tasks that take command-line arguments, like the input and output directories, or even which class to run. Consider putting it all in an executable bash script like this:

```
#!/bin/bash

# process command-line input and output dirs
INPUT=$1
OUTPUT=$2

# these are hardcoded for this script
LIBJARS=/opt/math3.jar, morejars.jar
FILES=/usr/local/share/meta-data.csv, morefiles.txt
APP_JAR=/usr/local/share/myApp.jar
APP_CLASS=com.myPackage.MyMapReduceJob

# clean the output dir
hadoop fs -rm -r $OUTPUT

# launch the job
hadoop jar $APP_JAR $APP_CLASS -files $FILES -libjars $LIBJARS $INPUT $OUTPUT
```

Then you have to remember to make the script executable (just once):

```
bash$ chmod +x runMapReduceJob.sh
```

And then run it like this:

```
bash$ myJobs/runMapReduceJob.sh inputDirGoesHere outputDirGoesHere
```

Or if you are running it from the same directory that the script is located in, use this:

```
bash$ ./runMapReduceJob.sh inputDirGoesHere outputDirGoesHere
```

MapReduce Examples

To really master MapReduce, you need to practice. There is no better way to understand how it all works than to jump in and start solving problems. Although the system may seem complex and cumbersome at first, its beauty will reveal itself as you have some successes. Here are some typical examples and some insightful computations.

Word Count

Here we use the built-in mapper class for counting tokens, `TokenCounterMapper`, and the built-in reducer class for summing integers, `IntSumReducer`:

```
public class WordCountMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new WordCountMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(WordCountMapReduceExample.class);
        job.setJobName("WordCountMapReduceExample");

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

        job.setMapperClass(TokenCounterMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setNumReduceTasks(1);

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

The job can be run on an input directory that has any type of text file:

```
hadoop jar MyApp.jar \\
com.datascience.WordCountMapReduceExample input output
```

The output can be viewed with the following:

```
hadoop fs -cat output/part-r-00000
```

Custom Word Count

We may notice the built-in `TokenCounterMapper` class is not producing the results we like. We can always use our `SimpleTokenizer` class from Chapter 4:

```
public class SimpleTokenMapper extends
    Mapper<LongWritable, Text, Text, LongWritable> {

    SimpleTokenizer tokenizer;

    @Override
    protected void setup(Context context) throws IOException {
        // only keep words greater than three chars
        tokenizer = new SimpleTokenizer(3);
    }

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

        String[] tokens = tokenizer.getTokens(value.toString());
        for (String token : tokens) {
            context.write(new Text(token), new LongWritable(1L));
        }

    }
}
```

Just be sure to set the appropriate changes in the job:

```
/* mapper settings */
job.setMapperClass(SimpleTokenMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

/* reducer settings */
job.setReducerClass(LongSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
```

Sparse Linear Algebra

Imagine that we have a large matrix (either sparse or dense) in which the `i,j` coordinates and corresponding value are stored in each line of the file in the format `<i,j,value>`. This matrix is so large that it is not practical to load it into RAM for further linear algebra routines. Our goal is to perform the matrix vector multiplication with an input vector we provide. The vector has been serialized so that the file can be included in the MapReduce job.

Imagine we have stored text files of comma- (or tab-) separated values across many nodes in our distributed filesystem. If the data is stored as a literal `i,j,value` text string (e.g., 34, 290, 1.2362) in each line of the file, then we can write a simple mapper to parse each line. In this case, we will do matrix multiplication, and as you may recall, that process requires multiplying each row of the matrix by the column vector of the same length. Each position `i` of the output vector will then take the same index as the corresponding matrix row. So we will use the matrix row i as the key. We will create a custom writable `SparseMatrixWritable` that contains the row index, column index, and value for each entry in the matrix:

```
public class SparseMatrixWritable implements Writable {
    int rowIndex; // i
    int columnIndex; // j
    double entry; // the value at i,j

    public SparseMatrixWritable() {
    }

    public SparseMatrixWritable(int rowIndex, int columnIndex, double entry) {
        this.rowIndex = rowIndex;
        this.columnIndex = columnIndex;
        this.entry = entry;
    }

    @Override
    public void write(DataOutput d) throws IOException {
        d.writeInt(rowIndex);
        d.writeInt(rowIndex);
        d.writeDouble(entry);
    }

    @Override
    public void readFields(DataInput di) throws IOException {
        rowIndex = di.readInt();
        columnIndex = di.readInt();
        entry = di.readDouble();
    }

}
```

A custom mapper will read in each line of text and parse the three values, using the row index as the key and the `SparseMatrixWritable` as the value:

```
public class SparseMatrixMultiplicationMapper
 extends Mapper<LongWritable, Text, IntWritable, SparseMatrixWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        try {
            String[] items = value.toString().split(",");
            int rowIndex = Integer.parseInt(items[0]);
            int columnIndex = Integer.parseInt(items[1]);
            double entry = Double.parseDouble(items[2]);
            SparseMatrixWritable smw = new SparseMatrixWritable(
            rowIndex, columnIndex, entry);
            context.write(new IntWritable(rowIndex), smw);
            // NOTE can add another context.write() for
            // e.g., a symmetric matrix entry if matrix is sparse upper triag
        } catch (NumberFormatException | IOException | InterruptedException e) {
            context.getCounter("mapperErrors", e.getMessage()).increment(1L);
        }
    }
}
```

The reducer must load in the input vector in the `setup()` method, and then in the `reduce()` method we extract column indices from the list of `SparseMatrixWritable`, adding them to a sparse vector. The dot product of the input vector and sparse vector give the value for the output for that key (e.g., the value of the resultant vector at that index).

```
public class SparseMatrixMultiplicationReducer extends Reducer<IntWritable,
                     SparseMatrixWritable, IntWritable, DoubleWritable>{

    private RealVector vector;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {

        /* unserialize the RealVector object */

        // NOTE this is just the filename
        // please include the resource itself in the dist cache
        // via -files at runtime
        // set the filename in Job conf with
        // set("vectorFileName", "actual file name here")

        String vectorFileName = context.getConfiguration().get("vectorFileName");
        try (ObjectInputStream in = new ObjectInputStream(
        new FileInputStream(vectorFileName))) {
            vector = (RealVector) in.readObject();
        } catch(ClassNotFoundException e) {
            // err
        }
    }

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

        /* rely on the fact that rowVector dim == input vector dim */
        RealVector rowVector = new OpenMapRealVector(vector.getDimension());

        for (SparseMatrixWritable value : values) {
            rowVector.setEntry(value.columnIndex, value.entry);
        }

        double dotProduct = rowVector.dotProduct(vector);

        /* only write the nonzero outputs,
        since the Matrix-Vector product is probably sparse */
        if(dotProduct != 0.0) {
            /* this outputs the vector index and its value */
            context.write(key, new DoubleWritable(dotProduct));
        }
    }
}
```

The job can be set up to run like this:

```
public class SparseAlgebraMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new SparseAlgebraMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(SparseAlgebraMapReduceExample.class);
        job.setJobName("SparseAlgebraMapReduceExample");

        // third command-line arg is the filepath to the serialized vector file
        job.getConfiguration().set("vectorFileName", args[2]);

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

        job.setMapperClass(SparseMatrixMultiplicationMapper.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(SparseMatrixWritable.class);
        job.setReducerClass(SparseMatrixMultiplicationReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(DoubleWritable.class);
        job.setNumReduceTasks(1);

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

This can be run with the following command:

```
hadoop jar MyApp.jar \\
com.datascience.SparseAlgebraMapReduceExample \\
-files /<path>/RandomVector.ser input output RandomVector.ser
```

You can view the output with this:

```
hadoop fs -cat output/part-r-00000
```

PREV
5. Learning and Prediction

NEXT
A. Datasets