RogueWave SOFTWARE

Accelerating Great Code

USING JMSL IN HADOOP MAPREDUCE APPLICATIONS

Big Data has become an industry buzzword over the last decade, even though it can mean different things to different people. To be sure, the size and complexity of data has exploded. Some estimates [CIO Insight, 2014] put the size of the digital universe by the year 2020 to 44 zettabytes[1]. Thanks to the internet, smart phones, and other devices making up the Internet of Things, plus the availability of inexpensive commodity hardware, more data and newer types of data are being collected, processed, stored, and analyzed.

While the scale today may be unprecedented, research into solving large computational problems has been going on for some time. The canonical strategy is to break the problem down first into parts of manageable size and then send the parts out to independent workers to process them in parallel. Each worker reports results to some centralized controller, which in turn combines the information into an actionable form. Distributed computing is the term generally used for breaking the problem down into parts and the communication of those parts across a network. Parallel computing, on the other hand, refers to the simultaneous processing of parts of the larger problem[2].

MapReduce is a programming model for just such a strategy. MapReduce was developed by Google scientists in 2004[3] in response to their own challenges in processing large scale data. The two basic steps in a MapReduce program are map and reduce. The map step reads raw data and collates it for the reduce step, which combines the data in a meaningful way. One of the most widely used implementations of the MapReduce model is Apache Hadoop. Apache Hadoop is a Java open source project for distributed computing. Hadoop has two main components: the Hadoop Distributed File System (HDFS) and the MapReduce programming and job management framework.

The JMSL Numerical Library is a pure Java numerical library, providing a broad range of advanced mathematics, statistics, and charting for the Java environment. It extends core Java numerics and allows developers to seamlessly integrate advanced analytics into their Java applications.

In a distributed computing environment, JMSL can be made available to each node in a cluster to process a section of the data and produce output in a form that can be gathered and combined in a logical fashion. The following sections illustrate using JMSL in Hadoop MapReduce applications.

[1] A zettabyte is 1,000 exabytes = 1M petabytes = 1B terabytes. 44 zettabytes is 44 billion terabytes!
[2] Some authors define parallel computing as the use of multiple threads on the same CPU.
[3] http:research.google.com/archive/mapreduce-osdi04-slides/index-auto-0006.html

THE STRUCTURE OF A HADOOP MAPREDUCE APPLICATION

The prototypical Hadoop MapReduce application has at least one `Mapper` class, one `Reducer` class, and one `driver` class. The `Mapper` class reads the input data and creates intermediate data. The `Reducer` class accumulates intermediate information and emits a combined result. The `Driver` class extends the Hadoop class `Configured` and implements the Hadoop `Tool` interface, and is used to configure and execute the job.

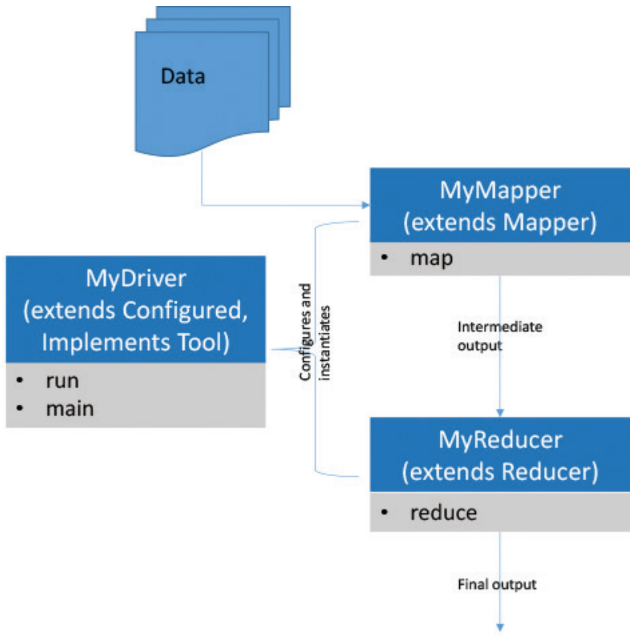

Data
MyMapper (extends Mapper)
• map
MyDriver (extends Configured, Implements Tool)
• run
• main
Configures and instantiates
Intermediate output
MyReducer (extends Reducer)
• reduce
Final output

Figure 1: Components of a simple Hadoop application

The programmer must write (override) the methods to handle the format of the data, the instructions for how the data should be processed or combined, and the desired form of the output. All of this depends on the nature of the data analysis problem that the programmer needs to solve. For our first example we begin with a very simple problem: summarizing a numeric value by key.

Example 1: JMSL summary

MapReduce operates on data organized into <key, Item> pairs. It assumes that every data "item" belongs to one and only one "key" and this is how the `Mapper` expects the raw data. In the first example, the key is text and the item is a scalar value, such as <ABC, 10>. A sample of the contents of a comma delimited input file may be:

BDK,	20
ABC,	10
EAF,	50
AGC,	10
IJK,	90
DAL,	40

The JMSL `Summary` class produces summary statistical information such as counts, means, variances, and other descriptive statistics. To summarize data by key on a Hadoop cluster, we write three classes, `SummaryDriver`, `SummaryMapper`, and `SummaryReducer`.

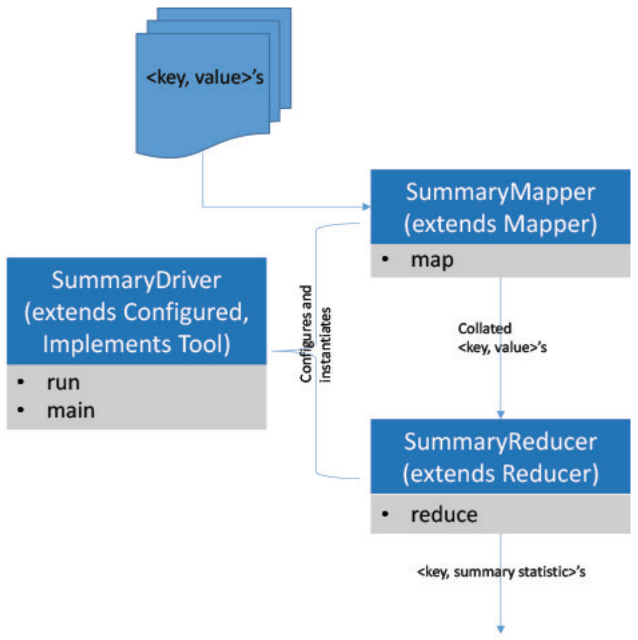

<key, value>'s
SummaryMapper (extends Mapper)
• map
SummaryDriver (extends Configured, Implements Tool)
• run
• main
Configures and instantiates
Collated <key, value>'s
SummaryReducer (extends Reducer)
• reduce
<key, summary statistic>'s

Figure 2: Summary components

`SummaryMapper` extends the Hadoop class `Mapper` and overrides the `map()` method. The `map()` method reads the input data and writes intermediate output that is sent to the `SummaryReducer`. This intermediate output is managed by Hadoop and is written to the local file system.

```
public static class SummaryMapper extends
            Mapper<Object, Text, Text, DoubleWritable> {

    Text word = new Text("key");
    DoubleWritable val = new DoubleWritable();

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

        String columns[] = value.toString().split(",");
        word.set(columns[0]);
        val.set(Double.parseDouble(columns[1]));
        context.write(word, val);
    }
}
```

`SummaryReducer` extends the Hadoop class `Reducer` and overrides the method, `reduce()`. The `reduce()` method combines all the intermediate values for each unique key and merges the output. It is in the `reduce()` method that we use the JMSL `Summary` class to define how the items should be combined.

```
public static class SummaryReducer extends
        Reducer<Text, DoubleWritable, Text, DoubleWritable> {

    Text out = new Text();
    DoubleWritable result = new DoubleWritable();

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

        // JMSL
        Summary summary = new Summary();
        // For manual calculation.
        double sum = 0;
        double count = 0;

        for (DoubleWritable val : values) {
            // JMSL
            summary.update(val.get());
            // Manual calculation, just for validation.
            sum += val.get();
            count += 1;
        }
        sum = sum / count;
        out.set(key);
        result.set(summary.getMean());
        System.out.println(key + “,” + sum);

        context.write(out, result);
    }
}
```

On a large cluster, data for a specific key may be reside on several different nodes. The map tasks take care of pulling all the data for a specific key together, before calling the `reduce()` method. Within the `reduce()` method `summary.update()` accepts one value at a time, so we don’t need to create additional data arrays.

Of course, calculating the mean is simple; we could do it manually and Hadoop would keep track of the results by key for us. The advantage comes when we can leverage JMSL methods for more advanced analyses on distributed networks, knowing that Hadoop manages the communication overhead for us.

Next, the `run()` method in `SummaryDriver` handles the set up and configuration of the MapReduce job. First instantiate a new job with this line:

```
      Job job = new Job(super.getConf(), “calculate mean”);
```

The following lines set the input and output values for the `Mapper` and `Reducer` and the input and output paths.

```
    @Override
    public int run(String[] args) throws IOException, InterruptedException,
            ClassNotFoundException {


        // Set up the configuration for the MapReduce job.
        Job job = new Job(super.getConf(), "calculate mean");
        job.setJarByClass(getClass());
        job.setMapperClass(SummaryMapper.class);
        job.setReducerClass(SummaryReducer.class);

        // Set the input and output data types.
        // Note that Mapper output key and value must match
        // reducer input key and value.
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(DoubleWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(DoubleWritable.class);

        // Set the input path and output path.
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

The statement to run the job is:

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

The `main()` method in `SummaryDriver` calls the `run()` method.

```
public static void main(String[] args) throws IOException,
            InterruptedException, ClassNotFoundException,
            Exception {

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
                  args).getRemainingArgs();

        if (otherArgs.length != 2) {
            System.err.println("Usage: SummaryDriver <in> <out>");
            System.exit(2);
        }

    int res = ToolRunner.run(conf, new SummaryDriver(), args);
}
```

Once on the cluster, to run the code from the command line the format is:

```
$hadoop jar <nameOfJarFile>.jar <class_directory/nameOfDriverClass> /
<Hadoop options> <args to driver class>
```

We use the Hadoop option `libjars` to make JMSL available to the nodes in the cluster. For example, the command for our first example will look something like:

```
$hadoop jar HadoopJMSLExamples.jar <class_directory>/SummaryDriver /
-libjars /lib/jmsl.jar /user/hdfs/SummaryInput /user/hdfs/SummaryOutput
```

See the appendix for a few other considerations when running the job on the cluster.

Example 2: JMSL linear regression

Multiple linear regression is one of the most widely used predictive models. The model assumes that the variation in one variable, the dependent or response variable, can be explained by the variation in a set of independent variables, the explanatory or predictor variables, and that the relationship is linear in the coefficients[4].

For example, in order to plan for new store locations a products company may set up a regression model with sales as a dependent variable and predictors such as population size, per capita income, and other demographic variables. Large companies, with thousands of products and customers and potentially millions of transactions a day, may need a distributed solution, such as Hadoop MapReduce, to first accumulate the data to a relevant level of detail and then to fit the regression model.

In this example, we illustrate using the JMSL linear regression in a Hadoop application, similar to what might be used for a business planning for new stores.

There are usually many steps to organize the raw data before observations are ready for the regression model. Assuming these steps have been done (Hadoop MapReduce may be leveraged for this purpose as well), the observations when ready for the regression model are of the form:

$$
\begin{array}{l}
Y_1, x_{11}, x_{12}, x_{13}, \ldots, x_{1m} \\
Y_2, x_{21}, x_{22}, x_{23}, \ldots, x_{2m} \\
Y_3, x_{31}, x_{32}, x_{33}, \ldots, x_{3m} \\
\ldots \\
Y_n, x_{n1}, x_{n2}, x_{n3}, \ldots, x_{nm}
\end{array}
$$

That is, the value of the dependent variable Y is paired with the observed values of the predictors. The linear regression algorithm must have knowledge of the complete (m+1)-Tuple (Y_i, x_{i1}, xi_2, x_{i3}, …, x_{im}) for each index, *i*.

While example 1 worked directly on the <key, Item> pairs, which is the form of data for the prototypical MapReduce job, the `Mapper` class in this example needs some additional sophistication so it can read and process <key, (Item1, …, ItemM)> type data, where different values of keys separate different regression data sets. One option is to treat the entire row as a single text object and then parse and re-cast the elements into `doubleValues`. For clarity, we prefer to group the response and predictor variables as one set so the observation is communicated by Hadoop as one piece of data.

[4] The technical assumptions behind the model should be considered and validated for a specific problem using diagnostic tests. See for example, Neter, et. al., for more details.

To begin with, we create the new type, `DoubleArrayWritable`, by extending two Hadoop data types, `DoubleWritable` and `ArrayWritable`.

```
public class DoubleArrayWritable extends ArrayWritable {

    public DoubleArrayWritable() {
        super(DoubleWritable.class);
    }

    public DoubleArrayWritable(DoubleWritable[] values) {
        super(DoubleWritable.class, values);
    }

    public void set(DoubleWritable[] values) {
        super.set(values);
    }

    public DoubleWritable get(int idx) {
        return (DoubleWritable) get()[idx];
    }

    public double[] getArray(int from, int to) {
        int sz = to - from + 1;
        double[] vector = new double[sz];
        for (int i = from; i <= to; i++) {
            vector[i - from] = get(i).get();
        }
        return vector;
    }
}
```

In addition to our new class, once again we need a driver, mapper, and reducer:

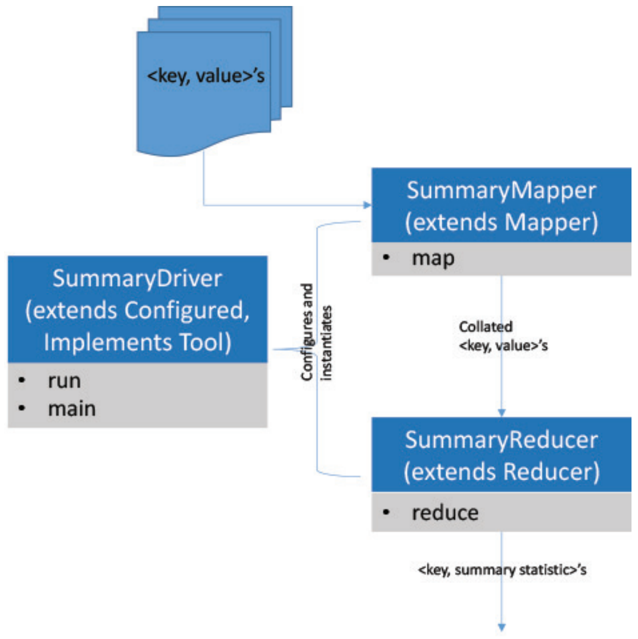

<key, value>'s
SummaryMapper (extends Mapper)
• map
SummaryDriver (extends Configured, Implements Tool)
• run
• main
Configures and instantiates
Collated <key, value>'s
SummaryReducer (extends Reducer)
• reduce
<key, summary statistic>'s

Figure 3: Linear regression components

The `Mapper` class handles the new input and creates the `DoubleArrayWritable` object. The `map()` method emits the key and the array for the reducer in the line, `context.write(word, val)`.

```
public static class LinearRegressionMapper extends
        Mapper<Object, Text, Text, DoubleArrayWritable> {

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

        String origColumns[] = value.toString().split(",");
        Text word = new Text("key");
        word.set(origColumns[0]);
        DoubleWritable[] x = new DoubleWritable[origColumns.length - 1];

        for (int i = 0; i < x.length; i++) {
            x[i] = new DoubleWritable();
            x[i].set(Double.parseDouble(origColumns[i + 1]));
        }
        DoubleArrayWritable val = new DoubleArrayWritable(x);
        // Emit (word, val) for the reducer where word
        // is the key and val is a DoubleArrayWritable.
        context.write(word, val);
    }
}
```

In `LinearRegressionReducer`, we extract the observations from the `DoubleArrayWritable` object, assign the response value and then the predictor values. (A check for the number of predictors allows for potentially different sized problems by different keys.) After extracting information from the observation an update is applied to the `LinearRegression` object, here named `lr`.

```
public static class LinearRegressionReducer extends
        Reducer<Text, DoubleArrayWritable, Text, Text> {

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

        int numPred = 0;
        double y = 0.0;
        double[] x = null;
        DoubleWritable[] dwa = null;
        LinearRegression lr = null;

        for (DoubleArrayWritable val : values) {
            dwa = (DoubleWritable[]) val.toArray();
            if (numPred == 0) {
                numPred = dwa.length - 1;
                //JMSL
                lr = new LinearRegression(numPred, false);
            }
            y = dwa[0].get();
            x = new double[numPred];
            for (int i = 0; i < numPred; i++) {
                x[i] = dwa[i + 1].get();
            }
            //JMSL
            lr.update(x, y);
        }

        // Now we emit the regression model object
            double[] coefs = lr.getCoefficients();
            StringBuilder val = new StringBuilder();
            String out = "Problem " + key.toString() + " Coefficients ";
            val.append("\n");

            for (int i = 0; i < coefs.length; i++) {
                if (i < coefs.length - 1) {
                    val.append(String.format("%5.4f,", coefs[i]));
                } else {
                    val.append(String.format("%5.4f", coefs[i]));
                }
            }
            val.append(String.format("%n"));
            context.write(new Text(out), new Text(val.toString()));
    }
}
```

There are many options for the format and content of the output. Here we extract the estimated coefficients and append them into a `Text` object. Note that the format of the output from the reducer must match the last two arguments in:

```
     Reducer<Text, DoubleArrayWritable, Text, Text>
```

The `run()` method in `LinearRegressionDriver` contains the analogous statements as in `SummaryDriver.run`, setting up the job configuration, and `LinearRegressionDriver.main` calls the `run()` method, as before.

```
    @Override
    public int run(String[] args) throws Exception {
        // Set up a new job.
        Job job = new Job(super.getConf(), "Linear Regression");
        // Sets the Jar file to this specific class.
        job.setJarByClass(getClass());
        // Sets the Mapper and Reducer to LinearRegressionMapper and
        // LinearRegressionReducer.
        job.setMapperClass(LinearRegressionMapper.class);
        job.setReducerClass(LinearRegressionReducer.class);
        // Set the Map output classes and the Reducer Output classes.
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(DoubleArrayWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        //Set the input and output paths.
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

    public static void main(String[] args) throws IOException,
            InterruptedException, ClassNotFoundException, Exception {

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
                                            args).getRemainingArgs();

        if (otherArgs.length != 2) {
            System.err.println("Usage: LinearRegressionDriver <in> <out>");
            System.exit(2);
        }

        int res = ToolRunner.run(conf, new LinearRegressionDriver(), args);
    }
```

Similarly, the command to run the job on the cluster is:

```
$hadoop jar HadoopJMSLExamples.jar / <class_directory>/LinearRegressionDriver -libjars
/lib/jmsl.jar / /user/hdfs/LinearRegressionInput /user/hdfs/LinearRegressionOutput
```

This Hadoop MapReduce application performs linear regression on distributed data, such as might arise when a large company wants to predict sales based on demographic data to help them plan for new store locations. As is, this code fits linear regression models by key assuming the input data is in the observation format we described earlier. The output, by key, is the estimated coefficients of each model. For example, we simulated sales and demographic data for three regions, R1, R2, and R3, and ran three regression problems using sales as the dependent variable and population and per capita disposable income as the independent variables. There is one output file with the estimated coefficients for each problem:

```
$ hadoop fs -cat LinearRegressionOutput/part-r-00000
Problem R1 Coefficients
4.5092, 0.0592, 0.0089

Problem R2 Coefficients
2.0779, 0.0640, 0.0095

Problem R3 Coefficients
14.7010, 0.1002, 0.0057
```

Thus for the region 1, the fitted regression line is:

$$Y = 4.5092 + 0.0592*X1 + 0.0089*X2$$

For a new marketing area within this region, having a population X1 = 302 (1000's) and mean disposable income of X2 = 4500 ($ per capita), the predicted sales ($1000's) are:

$$Y = 4.5092 + 0.0592*302 + 0.0089*4500 = 62.4376.$$

Similarly, we can obtain predicted sales for the other two regions. Other big data applications for linear regression include public survey data, social network analysis, and genome association analysis, to name just a few.

Example 3: Apriori for association rule discovery

Association rule discovery refers to the problem of detecting associations among discrete items. In market basket analysis, for example, the discrete items are the different products purchased together in individual transactions. Companies that sell many and varied products might use market basket analysis to help them make better promotional decisions. For example, knowing that two products are highly associated, a company would run a promotion on one or the other, but not both. There are applications for association rule discovery in the areas of text mining and bioinformatics as well.

The Apriori algorithm (Agrawal and Srikant, 1994) is one of the most popular algorithms for association rule discovery in transactional datasets. Apriori first mines the transactions for the frequent item sets. An item set (set of items) is frequent if it appears in more than a minimum number of transactions. The number of transactions containing an item set is known as its *support*, and the minimum support (as a percentage of transactions) is a control parameter in the algorithm. From the collection of frequent item sets, the algorithm then filters for significant associations amongst the items.

If an item set is frequent in one transaction data set, it does not necessarily mean that it is frequent in the entire collection of data. This is why Apriori needs two passes through the data when it is distributed. The main reference for this procedure is Savasere, Omiecinski, and Navathe (1995). The procedure is compared with alternative approaches in Rajaraman and Ullman (2011).

For distributed data, the procedure is:

1. Find the frequent item sets in each transaction partition. This first pass through the data sets and uses a MapReduce pair.
2. Find the union of all the frequent item sets. This step reads the output of the separate tasks in step 1 but does not need to re-read the transactions. This step is accomplished with a second MapReduce pair.
3. For every item set in the union, count the number of times it occurs in each transaction set. This requires a second pass through the transaction data sets. The outcome of step 3 is the collection of all the item sets which may be frequent for all of the data, i.e., the candidate frequent item sets. This step is performed by a `Mapper` which reads intermediate output from step 2. The `Mapper` in this step emits the itemset and the total count.
4. Filter through the collection of candidate item sets for those item sets meeting the minimum support threshold. This step is performed with a MapReduce pair identical to step 1.

While a different design pattern may be more or less efficient, we selected the following design for demonstration purposes. The `Driver` code manages the configuration of the three MapReduce jobs, ensuring that `Mapper2` can find `Reducer1` output, and `Reducer3` knows where to find `Reducer2` output, and so on. These relationships are shown in the diagram below.

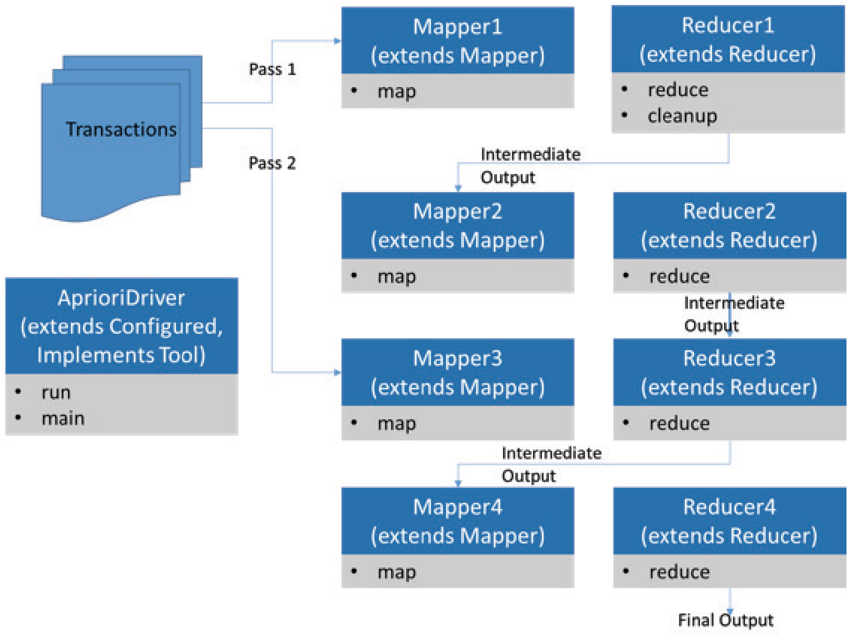
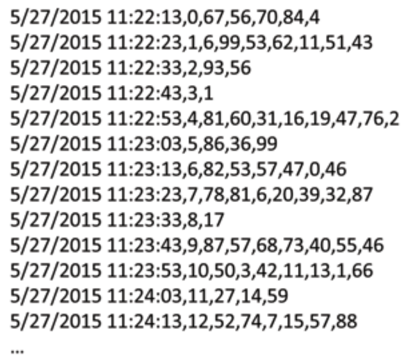


Figure 4: Apriori components

```
public class AprioriDriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws IOException, InterruptedException,
            ClassNotFoundException {

        // Job 1
        Configuration conf1 = super.getConf();
        conf1.set("minSupportPercentage", "0.05");
        conf1.set("maxNumProducts", "100");
        conf1.set("maxSetSize", "3");

        Job job1 = new Job(conf1, "Apriori Job 1");

        job1.setJarByClass(getClass());

        job1.setMapperClass(AprioriMapper1.class);
        job1.setReducerClass(AprioriReducer1.class);


        job1.setNumReduceTasks(4);

        job1.setMapOutputKeyClass(Text.class);
        job1.setMapOutputValueClass(IntArrayWritable.class);

        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(Text.class);
        job1.setInputFormatClass(TextInputFormat.class);
        job1.setOutputFormatClass(TextOutputFormat.class);

        TextInputFormat.addInputPath(job1, new Path(args[0]));
        TextOutputFormat.setOutputPath(job1, new Path(args[1]));

        // Job 2
        Configuration conf2 = super.getConf();
        Job job2 = new Job(conf2, " Apriori Map Task 2");

        job2.setJarByClass(getClass());

        job2.setMapperClass(AprioriMapper2.class);
        job2.setReducerClass(AprioriReducer2.class);

        job2.setMapOutputKeyClass(Text.class);
        job2.setMapOutputValueClass(DoubleWritable.class);

        job2.setOutputKeyClass(Text.class);
        job2.setOutputValueClass(Text.class);

        job2.setInputFormatClass(TextInputFormat.class);
        job2.setOutputFormatClass(TextOutputFormat.class);

        TextInputFormat.addInputPath(job2, new Path(args[1]));
        TextOutputFormat.setOutputPath(job2, new Path(args[2]));

        // Job 3
        Configuration conf3 = super.getConf();
        conf3.set("intermediateOutputPath", args[2]);
        conf3.set("maxNumProducts", "100");
        conf3.set("maxSetSize", "3");
        conf3.set("minSupportPercentage", "0.05");

        Job job3 = new Job(conf3, " Apriori Map Task 3");

        job3.setJarByClass(getClass());

        job3.setMapperClass(AprioriMapper3.class);
        job3.setReducerClass(AprioriReducer3.class);

        job3.setMapOutputKeyClass(Text.class);
        job3.setMapOutputValueClass(IntArrayWritable.class);

        job3.setOutputKeyClass(Text.class);
        job3.setOutputValueClass(Text.class);

        job3.setInputFormatClass(TextInputFormat.class);
        job3.setOutputFormatClass(TextOutputFormat.class);

        // Mapper3 reads the transactions again.
        TextInputFormat.addInputPath(job3, new Path(args[0]));
        TextOutputFormat.setOutputPath(job3, new Path(args[3]));
        // Job 4
        Configuration conf4 = super.getConf();

        Job job4 = new Job(conf4, " Apriori Map Task 4");

        job4.setJarByClass(getClass());

        job4.setMapperClass(AprioriMapper4.class);
        job4.setReducerClass(AprioriReducer4.class);

        job4.setMapOutputKeyClass(Text.class);
        job4.setMapOutputValueClass(Text.class);
        job4.setOutputKeyClass(Text.class);
        job4.setOutputValueClass(Text.class);

        job4.setInputFormatClass(TextInputFormat.class);
        job4.setOutputFormatClass(TextOutputFormat.class);

        TextInputFormat.addInputPath(job4, new Path(args[3]));
        TextOutputFormat.setOutputPath(job4, new Path(args[4]));

        // Run job1.
        job1.waitForCompletion(true);
        // Run job2.
        job2.waitForCompletion(true);
        // Run job3.
        job3.waitForCompletion(true);
        // Run job4.
        job4.waitForCompletion(true);
        return 0;
    }

public static void main(String[] args) throws IOException,
            InterruptedException, ClassNotFoundException,
            Exception {

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
            args).getRemainingArgs();
        if (otherArgs.length != 5) {
            System.err.println("Usage: AprioriDriver <input_dir> "
                    + "<intermediate_dir_1>"
                    + "<intermediate_dir_2"
                    + "<intermediate_dir_3"
                    + "<output_dir>");
            System.exit(2);
        }
        int res = ToolRunner.run(conf, new AprioriDriver(), args);
    }
}
```

The format of the input is shown below. Each row contains the date/time, transaction id, and the list of items purchased in that transaction. (The items are encoded with unique integer id's here, but may be in different alpha-numeric formats):



```
5/27/2015 11:22:13,0,67,56,70,84,4
5/27/2015 11:22:23,1,6,99,53,62,11,51,43
5/27/2015 11:22:33,2,93,56
5/27/2015 11:22:43,3,1
5/27/2015 11:22:53,4,81,60,31,16,19,47,76,2
5/27/2015 11:23:03,5,86,36,99
5/27/2015 11:23:13,6,82,53,57,47,0,46
5/27/2015 11:23:23,7,78,81,6,20,39,32,87
5/27/2015 11:23:33,8,17
5/27/2015 11:23:43,9,87,57,68,73,40,55,46
5/27/2015 11:23:53,10,50,3,42,11,13,1,66
5/27/2015 11:24:03,11,27,14,59
5/27/2015 11:24:13,12,52,74,7,15,57,88
...
```

Figure 5: Input format

`AprioriMapper1` reads transactions from the input directory and emits a <key, item> pair, where key is converted to `Text` and the item is the list of products purchased in a single transaction. The list of products is converted to an `IntArrayWritable`. See the line containing the method, `context.write()`.

```
public class AprioriMapper1
        extends Mapper<Object, Text, Text, IntArrayWritable> {

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

        SimpleDateFormat dateFormat = new SimpleDateFormat(“M/yy”);
        String origColumns[] = value.toString().split(“,”);

        try {
            Date trxDate = dateFormat.parse(origColumns[0]);
            String trxId = origColumns[1];

            if (origColumns.length > 1) {
                IntWritable products[] =
                             new IntWritable[origColumns.length - 2];

                for (int i = 0; i < products.length; i++) {
                    Integer prodId = Integer.parseInt(origColumns[i + 2]);
                    products[i] = new IntWritable(prodId);
                }

         context.write(new Text(dateFormat.format(trxDate)),
                                          new IntArrayWritable(products));
            }
        } catch (ParseException ex) {
           Logger.getLogger(AprioriMapper1.class.getName()).log(Level.SEVERE,
                                                                  null, ex);
        }
    }
}
```

Note that we are using month-year to define the key:

```
      SimpleDateFormat dateFormat = new SimpleDateFormat("M/yy");
      …
         context.write(new Text(dateFormat.format(trxDate)),
                                          new IntArrayWritable(products));
```

A Hadoop partition may contain records for multiple keys, but all records for a unique key must be in a single partition, and furthermore, the single partition must fit into memory. In our case, all transactions in a month must be in the same partition. If there were too many transactions in a month, we could choose a smaller unit (week or day, for example). Determining the proper key level is part of configuring a specific Hadoop application. `AprioriMapper1` collates all transactions by Month-Year and then `AprioriReducer1` processes each of these chunks by calling JMSL Apriori and emitting the list of frequent itemsets to an intermediate output directory.

```
public class AprioriReducer1 extends
        Reducer<Text, IntArrayWritable, Text, Text> {

    int totalNumberOfTrx =  0;

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

        ArrayList trxList = new ArrayList();

        int numberOfTrx = 0;
        int numRows = 0;
        IntWritable[] iwa;

        for (IntArrayWritable val : values) {
            iwa = (IntWritable[]) val.toArray();
            trxList.add(iwa);
            numRows += iwa.length;
            numberOfTrx++;
        }
        totalNumberOfTrx += numberOfTrx;

        String sMinSupportPct
                = context.getConfiguration().get("minSupportPercentage");
        double minSupportPct = Double.parseDouble(sMinSupportPct);
        String sMaxNumProducts
                = context.getConfiguration().get("maxNumProducts");
        int maxNumProducts = Integer.parseInt(sMaxNumProducts);
        String sMaxSetSize = context.getConfiguration().get("maxSetSize");
        int maxSetSize = Integer.parseInt(sMaxSetSize);

        int xint[][] = new int[numRows][2];
        int idx = 0;
        for (int i = 0; i < trxList.size(); i++) {
            IntWritable[] A = (IntWritable[]) trxList.get(i);
            for (IntWritable A1 : A) {
                xint[idx][0] = i + 1;
                xint[idx][1] = A1.get() + 1;
                idx++;
            }
        }


        Itemsets fis = Apriori.getFrequentItemsets(xint, maxNumProducts,
                maxSetSize, minSupportPct);
        int numberOfItemSets = fis.getNumberOfItemsets();

        DoubleWritable support;
        for (int i = 0; i < numberOfItemSets; i++) {
            String outString = "";
            int[] itemSet = fis.getItemset(i);
            support = new DoubleWritable(fis.getSupport(i));
            for (int j = 0; j < itemSet.length; j++) {
                outString += Integer.toString(itemSet[j]) + ",";
            }
            String outStringF = String.format("%s%n", support.toString());
            context.write(new Text(outString), new Text(outStringF));
        }
    }

     @Override
    public void cleanup(Context context) throws IOException,
            InterruptedException{
        context.write(new Text("TotalTrx,"), new
                              Text(Integer.toString(totalNumberOfTrx)));
    }
}
```

Note how `Apriori` parameters can be retrieved from the configurations, for example:

```
String sMinSupportPct =
    context.getConfiguration().get("minSupportPercentage");
```

`AprioriReducer1` emits the <itemset, support> to a second intermediate output directory. We override the `cleanup()` method in order to pass the total number of transactions to subsequent MapReduce jobs. The second MapReduce job finds the union of itemsets. `AprioriMapper2` is an identity mapper, in that it just reads the output of `AprioriReducer1` and sends it as-is to `AprioriReducer2`, where the `reduce()` method produces the union of the itemsets and sends the output to a third intermediate output directory. An itemset in the union is frequent in at least one of the chunks of transactions, but because it may not be frequent when considering all of the transactions, it has to be counted over the transactions again. The third MapReduce job manages this step. Note that the union or the *c*andidate frequent itemsets must fit into memory. Usually this is a reasonable assumption. If there are too many candidate itemsets, the minimum support percentage may be too small. The goal is to find the strong associations, not to enumerate every combination.

```
public class AprioriMapper2 extends Mapper<Object, Text, Text, DoubleWritable> {
   // This is an identity mapper.
    @Override
    public void map(Object key, Text value, Mapper.Context context)
            throws IOException, InterruptedException {

        String origColumns[] = value.toString().split(“,”);
        int len = origColumns.length;
        String outString=””;

        if(len >= 2){
        for(int i = 0; i < origColumns.length-2; i++){
            outString += (origColumns[i] + “,”);
        }

  outString += origColumns[origColumns.length-2];
        double outDouble =
            Double.parseDouble(origColumns[origColumns.length -1]);
        context.write(new Text(outString), new DoubleWritable(outDouble));
        }
   }
}
```

```
public class AprioriReducer2 extends Reducer<Text, DoubleWritable, Text, Text> {

    // Performs the union of the candidate itemsets.
    @Override
    public void reduce(Text key, Iterable<DoubleWritable> values,
            Context context) throws IOException, InterruptedException {

        String outString;
        int sumOfSupports = 0;
        Iterator<DoubleWritable> iterator = values.iterator();

        for (Iterator<DoubleWritable> it = values.iterator(); it.hasNext();) {
            double value = it.next().get();
            sumOfSupports += value;
        }
        outString = String.format(“,%s%n”, Integer.toString(sumOfSupports));
        context.write(key, new Text(outString));
    }
}
```

`AprioriMapper3` is a repeat of the map task in `AprioriMapper1`, except it is paired with `AprioriReducer3`.

```
public class AprioriMapper3 extends Mapper<Object, Text, Text, IntArrayWritable> {

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

        SimpleDateFormat dateFormat = new SimpleDateFormat("M/yy");
        String origColumns[] = value.toString().split(",");

        try {
            Date trxDate = dateFormat.parse(origColumns[0]);

            if (origColumns.length > 1) {
                String trxId = origColumns[1];
                IntWritable products[] =
                        new IntWritable[origColumns.length - 2];

                for (int i = 0; i < products.length; i++) {
                    Integer prodId = Integer.parseInt(origColumns[i + 2]);
                    products[i] = new IntWritable(prodId);
                }

                context.write(new Text(dateFormat.format(trxDate)),
                        new IntArrayWritable(products));
            }
        } catch (ParseException ex) {
        Logger.getLogger(AprioriMapper1.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

`AprioriReducer3` overrides `setup()` to read in the candidate itemsets from the intermediate output path. Then, it receives the transactions from `AprioriMapper3`.

```
public class AprioriReducer3 extends
        Reducer<Text, IntArrayWritable, Text, Text> {

    int totalTrx = 0;
    ArrayList listOfCandidateItemSets = new ArrayList();
    Itemset[] candidateItemSets;
    int numberOfSets;

    @Override
    public void setup(Context context) throws IOException {

        // Retrieve the intermediate output path.
        Configuration conf = context.getConfiguration();
        String intermediateOutputPath = conf.get("intermediateOutputPath");
        Path opPath = new Path(intermediateOutputPath);

        try {
            FileSystem fs = FileSystem.get(conf);
            FileStatus[] status = fs.listStatus(opPath);

            for (FileStatus statu : status) {
                BufferedReader d = new BufferedReader(new
                              InputStreamReader(fs.open(statu.getPath())));
                String line;
                String origColumns[];
                String regexp = "[\\s,;\\n\\t]+";

                int aRow[];

                // Read the candidate itemsets from the intermediate path.
                while (d.ready()) {
                    line = d.readLine();
                    origColumns = line.trim().split(regexp);
                    if (origColumns[0].contains("TotalTrx")) {
                        this.totalTrx = Integer.parseInt(origColumns[1]);
                    } else {
                        aRow = new int[origColumns.length];
                        for (int i = 0; i < origColumns.length; i++) {
                            aRow[i] =
                                    Integer.parseInt(origColumns[i].trim());
                        }
                        listOfCandidateItemSets.add(aRow);
                    }
                    d.readLine();
                }
                d.close();
            }
        } catch (IOException | NumberFormatException ex) {
            Logger.getLogger(AprioriReducer3.class.getName()).log(Level.SEVERE, null, ex);
        }
        numberOfSets = listOfCandidateItemSets.size();
        candidateItemSets = new Itemset[numberOfSets];

        for (int i = 0; i < numberOfSets; i++) {
            int[] arow = (int[]) listOfCandidateItemSets.get(i);
            int[] itemSet = new int[arow.length - 1];
            System.arraycopy(arow, 0, itemSet, 0, arow.length - 1);
            candidateItemSets[i]
                    = new Itemset(itemSet, arow[arow.length - 1]);
        }
    }

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

```
        if(numberOfSets > 0){
        // Process
        ArrayList trxList = new ArrayList();
        int numberOfTrx = 0;
        int numRows = 0;
        IntWritable[] iwa;

        for (IntArrayWritable val : values) {
            iwa = (IntWritable[]) val.toArray();
            trxList.add(iwa);
            numRows += iwa.length;
            numberOfTrx++;
        }


        String sMinSupportPct
                = context.getConfiguration().get("minSupportPercentage");
        double minSupportPct = Double.parseDouble(sMinSupportPct);
        String sMaxNumProducts
                = context.getConfiguration().get("maxNumProducts");
        int maxNumProducts = Integer.parseInt(sMaxNumProducts);
        String sMaxSetSize = context.getConfiguration().get("maxSetSize");
        int maxSetSize = Integer.parseInt(sMaxSetSize);

        int xint[][] = new int[numRows][2];
        int idx = 0;

        // Build input data from data in the partition.
        for (int i = 0; i < trxList.size(); i++) {
            IntWritable[] A = (IntWritable[]) trxList.get(i);
            for (IntWritable A1 : A) {
                xint[idx][0] = i + 1;
                xint[idx][1] = A1.get() + 1;
                idx++;
            }
        }

        int[] freq;

        Itemsets candSets = new Itemsets(candidateItemSets, numberOfTrx,
                maxNumProducts, minSupportPct, maxSetSize);

        // Count frequencies of the itemset in the current
        // segment of transactions and emit the result.
        freq = Apriori.countFrequency(candSets, xint);

        for (int i = 0; i < candSets.getNumberOfItemsets(); i++) {
            String keyString = "";
            int[] itemset = candSets.getItemset(i);

            for (int j = 0; j < itemset.length - 1; j++) {
                keyString += Integer.toString(itemset[j]) + ",";
            }
            // Include the total number of transactions so that
            // the final reducer will have the information.
            keyString += Integer.toString(itemset[itemset.length - 1]);
            String outString = String.format(",%s,%s%n",
                    Integer.toString(freq[i]),
                    Integer.toString(this.totalTrx));

            context.write(new Text(keyString), new Text(outString));
        }
    }}
}
```

The last reducer writes the frequent `Itemsets` to the final output directory. In this example, the final output is just the list of frequent itemsets. Optionally, the example could generate the association rules using the JMSL method `Apriori.getAssociationRules()` and then form the final output using the results.

```
public class AprioriReducer4 extends Reducer<Text, Text, Text, Text> {

    // Performs the final filtering for frequent itemsets.
    @Override
    public void reduce(Text key, Iterable<Text> values,
            Context context) throws IOException, InterruptedException {

        String sMinSupportPct
                = context.getConfiguration().get("minSupportPercentage");
        double minSupportPct = Double.parseDouble(sMinSupportPct);

        String outString;
        int sumOfSupports = 0;
        double minimumSupport = 0;
        Iterator<Text> iterator = values.iterator();

        for (Iterator<Text> it = values.iterator(); it.hasNext();) {
            String[] valueSplit = iterator.next().toString().split(",");
            sumOfSupports += Integer.parseInt(valueSplit[0]);
            minimumSupport = minSupportPct * Integer.parseInt(valueSplit[1]);
        }

        if (sumOfSupports >= minimumSupport) {
            outString = String.format(",%s%n",
                                    Integer.toString(sumOfSupports));
            context.write(key, new Text(outString));
        }
    }
}
```

This Hadoop application performs market basket analysis using the Apriori algorithm and the scheme for aggregating Apriori results over distributed data sets. The final output will show the frequent sets and the corresponding support, i.e., the number of transactions in which the sets occurred. Here are a few of the frequent itemsets in the final output of an example with roughly one million transactions:

```
0               ,718729
0,1             ,592565
0,1,10          ,522135
0,1,19          ,355607
0,1,2           ,521825
0,1,3           ,355753
0,1,4           ,521838
0,1,5           ,356492
0,1,6           ,356687
0,1,7           ,356325
0,1,8           ,356652
…
```

So we can see that product with Id {0} is very frequent (72% of the transactions), and several of its supersets, {0, 1}, {0, 1, 10}, etc., are also frequent in the transactions. A next step would be to add a method in our application to generate the association rules. And, before generating any reports, we would use post-processing steps to use product names instead of product Id's.

SUMMARY

In the first example, the JMSL class `Summary` calculates a simple average. `Summary.update` takes one input value and updates the mean and other statistics with the input value. It is straightforward to treat univariate data as collection of <key, value> pairs as expected by MapReduce. On the 3-node cluster, Hadoop uses the `map()` and `reduce()` methods and other classes to manage the data across the network, ultimately calling `Summary.update()` with each data element and collecting results (the average) by unique key.

JMSL `LinearRegression` requires multiple values per record of input: a designated dependent variable and a collection of independent variables. Putting this type of data into a format MapReduce can use properly was the challenge in the second example. The best solution was to write a custom extension of one of the Hadoop classes, `ArrayWritable`, into the class, `DoubleArrayWritable`. Using the new class, the <key, value > pairs become <key, `DoubleArrayValues`> pairs.

The third example tests JMSL `Apriori` on distributed data sets. `Apriori` requires two passes through the data to perform aggregation. By the nature of the problem, the third example featured four advances in using Hadoop:

- Multiple MapReduce jobs
- Reading from intermediate output directories
- Accumulating <key, value> pairs into arrays
- Passing configuration parameters to the mappers and reducers

The excitement surrounding big data is the expectation of better results and new insights because we are taking more and presumably more accurate measurements of our world. Among the many important considerations to help reach this potential, not least are highly scalable mathematical and statistical algorithms that can perform efficiently on large clusters as traditional way of putting all the data in one place is often no longer feasible, reasonable or possible. Fundamentally, this means adapting algorithms for parallelization and distributed computing.

These fundamentals drive recent releases and inform the product road map for Rogue Wave Software's IMSL Numerical Libraries. To see more information on JMSL algorithms discussed here and more, visit IMSL Numerical Libraries.

REFERENCES

Agrawal, R. and Srikant, R. (1994), "Fast algorithms for mining association rules," *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29 - September 1, 1994.

Neter, John; Wasserman, William; and Kutner, Michael H. (1990), *Applied Linear Statistical Models*, 3rd ed., Richard D. Irwin, Inc., Homewood, Ill.

Rajaraman Anand and Ullman, Jeffrey David (2011), *Mining of Massive Datasets*, Cambridge University Press, Cambridge, UK.

Savasere, Ashok; Omiecinski, Edward; and Navathe, Shamkant (1995), *"An Efficient Algorithm for Association Rules in Large Databases"*, Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995.

White, Tom (2012). *Hadoop: The Definitive Guide*. O'Reilly Media, 3rd edition.

APPENDIX

Maven project

We actually built our HadoopJMSLExamples.jar file in NetBeans using Maven. The following dependencies were added to the pom.xml:

```
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>2.7.0</version>
        <type>jar</type>
    </dependency>

    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-core</artifactId>
        <version>1.2.1</version>
        <type>jar</type>
    </dependency>
    <dependency>
      <groupId>jmsl</groupId>
      <artifactId>jmsl</artifactId>
      <version>8.0</version>
      <scope>system</scope>
      <systemPath>${basedir}\lib\jmsl-8.0.jar</systemPath>
    </dependency>
```

Incidentally, we had no luck running the examples in Windows.

Running the job on the cluster

Working from the command line to run the Hadoop application requires first to switch user to the HDFS. This very much depends on how the Hadoop installation is set up, but it may look something like:

```
<user1>@myHadoopServer <136> su hdfs
Password:
hdfs@myHadoopServer:<homedir>$   <****>
```

To list on the Hadoop file system, use the command `hadoop fs -ls <dirname>`:

```
hdfs@UbuntuServer1: <homedir>$ hadoop fs -ls /user/hdfs/
```

For example, to make sure the input data directory contains the input files, list the directory:

```
hdfs@UbuntuServer1: <homedir>$ hadoop fs -ls /user/hdfs/SummaryInput
```

If the data files are not in this directory, use the Hadoop file system command “put” to place them into the input directory (for testing purposes):

```
hdfs@UbuntuServer1: <homedir>$ hadoop fs -put Number* /user/hdfs/SummaryInput
```

To run the job from the command line, the general format is:

```
$ hadoop jar <nameOfJarFile>.jar <class_directory/nameOfMainClass> /
<Hadoop generic options> <args expected by the main class>
```

Where the <class_directory> depends on the organization of the Java project and the generic options and other arguments expected by the main class are separated by spaces. In our summary example, the command takes the form:

```
$ hadoop jar HadoopJMSLExamples.jar com.mycompany.hadoopjmsl/SummaryDriver /  -libjar /
lib/jmsl.jar /user/hdfs/SummaryInput /user/hdfs/SummaryOutput
```

For this to work, we must have a local directory `/lib/` containing `jmsl.jar` and also we must have added the jar file to `$HADOOP_CLASSPATH`:

```
hdfs@UbuntuServer1: <homedir> export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:jmsl.jar
```

Apriori example has several command line arguments:

```
$ hadoop jar HadoopJMSLExamples.jar com/mycompany/hadoopjmsl/AprioriDriver / -libjar /
lib/jmsl.jar /user/hdfs/InputDirectory /user/hdfs/IntOutput1 / /user/hdfs/IntOutput2 /
user/hdfs/IntOutput3 /user/hdfs/FinalOutput
```

If any of the output directories already exist before running a job, an error results.

To empty a directory:

```
$ hadoop fs -rm /user/hdfs/Summary_output/*
```

To remove the directory:

```
$ hadoop fs -rmdir /user/hdfs/Summary_output
```

If a job completes without any exceptions, Hadoop outputs to the screen something like:

```
15/06/26 10:21:27 INFO mapreduce.Job: Counters: 38
     File System Counters
             FILE: Number of bytes read=22885514
             FILE: Number of bytes written=26089332
             FILE: Number of read operations=0
             FILE: Number of large read operations=0
             FILE: Number of write operations=0
             HDFS: Number of bytes read=922622
             HDFS: Number of bytes written=4712
             HDFS: Number of read operations=177
             HDFS: Number of large read operations=0
             HDFS: Number of write operations=13
     Map-Reduce Framework
             Map input records=1396
             Map output records=1396
             Map output bytes=16618
             Map output materialized bytes=19470
             Input split bytes=1516
             Combine input records=0
             Combine output records=0
             Reduce input groups=453
             Reduce shuffle bytes=19470
             Reduce input records=1396
             Reduce output records=453
             Spilled Records=2792
             Shuffled Maps =10
             Failed Shuffles=0
             Merged Map outputs=10
             GC time elapsed (ms)=521
             CPU time spent (ms)=0
             Physical memory (bytes) snapshot=0
             Virtual memory (bytes) snapshot=0
             Total committed heap usage (bytes)=2825912320
```

```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=111378
        File Output Format Counters
                Bytes Written=4712
```

To inspect the output, again we can use the Hadoop `fs` commands, `-ls` and `-cat`.

```
$ hadoop fs -ls /user/hdfs/SummaryOutput

-rw-r--r--   3 hdfs supergroup          0 2015-06-26 10:21 SummaryOutput2/_SUCCE
SS
-rw-r--r--   3 hdfs supergroup       4712 2015-06-26 10:21 SummaryOutput2/part-r
-00000
$ Hadoop fs -cat /user/hdfs/SummaryOutput/part-r-00000
            BDK    184.0
            ABC    275.0
            EAF    177.0
            AGC    71.0
            IJK    47.0
            DAL    187.0
            MUZ    125.0
            TLK    258.0
            …
```

RogueWave SOFTWARE

Accelerating Great Code