Map-Reduce with Hadoop for Large Scale Data Mining

Marissa Hollingsworth and *Amit Jain*[*]

Department of Computer Science
College of Engineering
Boise State University

[*]Chief Science Officer
Boise Computing Partners

Big Data, Big Disks, Cheap Computers

- *"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."* Rear Admiral Grace Hopper.
- *"More data usually beats better algorithms."* Anand Rajaraman.
- *"The good news is that Big Data is here. The bad news is that we are struggling to store and analyze it."* Tom White.

Introduction

- MapReduce is a programming model and an associated implementation for processing and generating large data sets.
- Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.
- Allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Introduced by Google. Used internally for all major computations on over 100k servers. Yahoo is running on over 36,000 Linux servers with 5 Petabytes of data. Amazon is leasing servers to run map reduce computations (EC2 and S3 programs). Microsoft is developing Dryad (a super set of Map-Reduce).

MapReduce Programming Model

- Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the Reduce function.
- The Reduce function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.
- MapReduce Specification Object. Contains names of input/output files and optional tuning parameters. The user then invokes the MapReduce function, passing it the specification object. The user's code is linked together with the MapReduce library.

A MapReduce Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
     EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
     result += ParseInt(v);
  Emit(key, AsString(result));
```

MapReduce Examples

- **Distributed Grep**: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.
- **Count of URL Access Frequency**: The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.
- **Reverse Web-Link Graph**: The map function outputs <target, source> pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(source)>

More MapReduce Examples

- **Inverted Index**: The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.
- **Distributed Sort**: The map function extracts the key from each record, and emits a <key, record> pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning and ordering facilities that are provided in a MapReduce implementation.

MapReduce Exercises

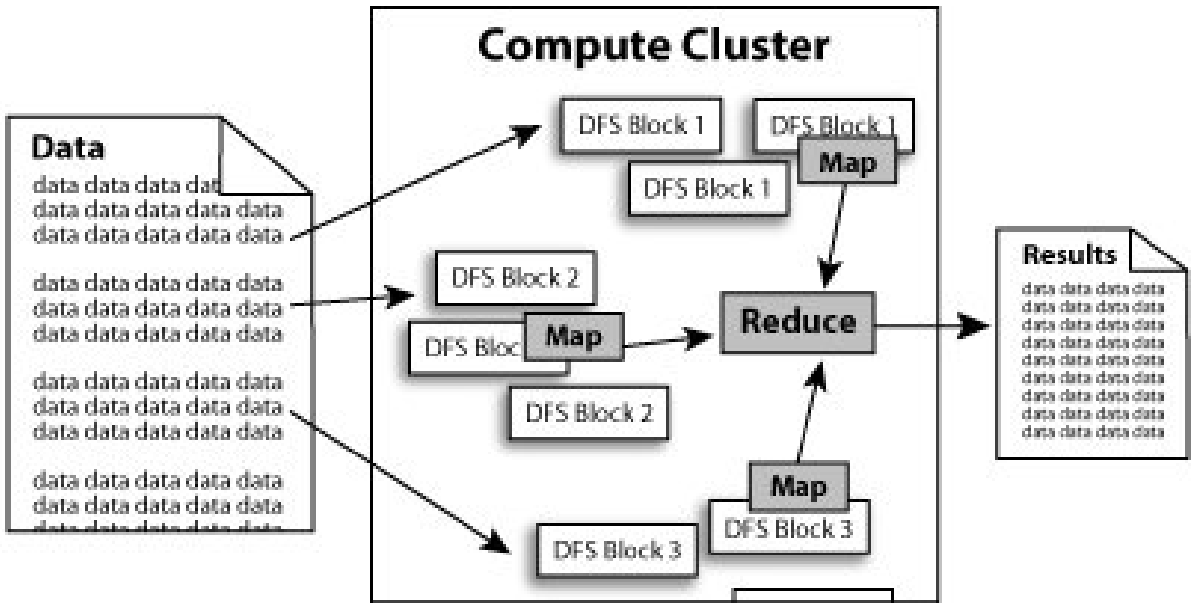
- **Capitalization Probability**: In a collection of text documents, find the percentage capitalization for each letter of the alphabet.
- **Term-Frequency and Inverse-Document-Frequency**. Given a corpus of text, we want to calculate tf-idf for every document and every token. We need to calculate over the corpus the following: number of tokens, number of unique terms, number of documents, number of occurrences of every term in every document and number of documents containing each term. This is useful in relevance ranking of a search–used in information retrieval and text mining. (`http://en.wikipedia.org/wiki/Tf-idf`)

Hadoop is a software platform that lets one easily write and run applications that process vast amounts of data. Features of Hadoop:

- Scalable: Hadoop can reliably store and process Petabytes.
- Economical: It distributes the data and processing across clusters of commonly available computers. These clusters can number into the thousands of nodes.
- Efficient: By distributing the data, Hadoop can process it in parallel on the nodes where the data is located. This makes it extremely efficient.
- Reliable: Hadoop automatically maintains multiple copies of data and automatically redeploys computing tasks based on failures.

Hadoop Implementation

Hadoop implements MapReduce, using the Hadoop Distributed File System (HDFS) (see figure below.) MapReduce divides applications into many small blocks of work. HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster. MapReduce can then process the data where it is located.



Hadoop History

- Hadoop is sub-project of the Apache foundation. Receives sponsorship from Google, Yahoo, Microsoft, HP and others.
- Hadoop is written in Java. Hadoop MapReduce programs can be written in Java as well as several other languages.
- Hadoop grew out of the Nutch Web Crawler project.
- Hadoop programs can be developed using Eclipse/NetBeans on MS Windows or Linux platforms. To use MS Windows requires Cygwin package. MS Windows is recommended for development but not as a full production Hadoop cluster.
- Used by Yahoo, Facebook, Amazon, RackSpace, Twitter, eBay, LinkedIn, New York Times, Last.fm, E-Harmony, Microsoft (via acquisition of Powerset) and many others. Several cloud consulting companies like Cloudera.
- New York Times article on Hadoop (3/17/2009) `http://www.nytimes.com/2009/03/17/technology/business-computing/17cloud.html`

Hadoop Map-Reduce Inputs and Outputs

- The Map/Reduce framework operates exclusively on $< key, value >$ pairs, that is, the framework views the input to the job as a set of $< key, value >$ pairs and produces a set of $< key, value >$ pairs as the output of the job, conceivably of different types.
- The key and value classes have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.
- The user needs to implement a `Mapper` class as well as a `Reducer` class. Optionally, the user can also write a `Combiner` class.

$$(\text{input}) < k1, v1 > \rightarrow \text{map} \rightarrow < k2, v2 > \rightarrow \text{combine} \rightarrow < k2, v2 >$$

$$\rightarrow \text{reduce} \rightarrow < k3, v3 > (\text{output})$$

How to write a Hadoop Map class

Subclass from `MapReduceBase` and implement the `Mapper` interface.

```
public class MyMapper<K extends WritableComparable, V extends Writable>
 extends MapReduceBase implements Mapper<K, V, K, V> { ... }
```

The `Mapper` interface provides a single method:

```
public void map(K key, V val, OutputCollector<K, V> output,
Reporter reporter)
```

- `WriteableComparable` key:
- `Writeable` value:
- `OutputCollector` output: this has the collect method to output a <key, value> pair
- `Reporter` reporter: allows the application code to permit alteration of status

The Hadoop system divides the input data into logical "records" and then calls `map()` once for each record. For text files, a record is one line of text. The key then is the byte-offset and the value is a line from the text file. For other input types, it can be defined differently. The main method is responsible for setting output key values and value types.

How to write a Hadoop Reduce class

Subclass from `MapReduceBase` and implement the `Reducer` interface.

```
public class MyReducer<K extends WritableComparable, V extends Writable>
 extends MapReduceBase implements Reducer<K, V, K, V> {...}
```

The `Reducer` interface provides a single method:

```
public void reduce(K key, Iterator<V> values,
OutputCollector<K, V> output, Reporter reporter)
```

- `WriteableComparable` key:
- `Iterator` values:
- `OutputCollector` output:
- `Reporter` reporter:

Given all the values for the key, the Reduce code typically iterates over all the values and either concatenates the values together in some way to make a large summary object, or combines and reduces the values in some way to yield a short summary value.

WordCount example in Java with Hadoop

Problem: To count the number of occurrences of each word in a large collection of documents.

```
/**
 * Counts the words in each line.
 * For each line of input, break the line into words and emit them as (word, 1)
 */
public static class Map extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
            Reporter reporter) throws IOException
    {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```

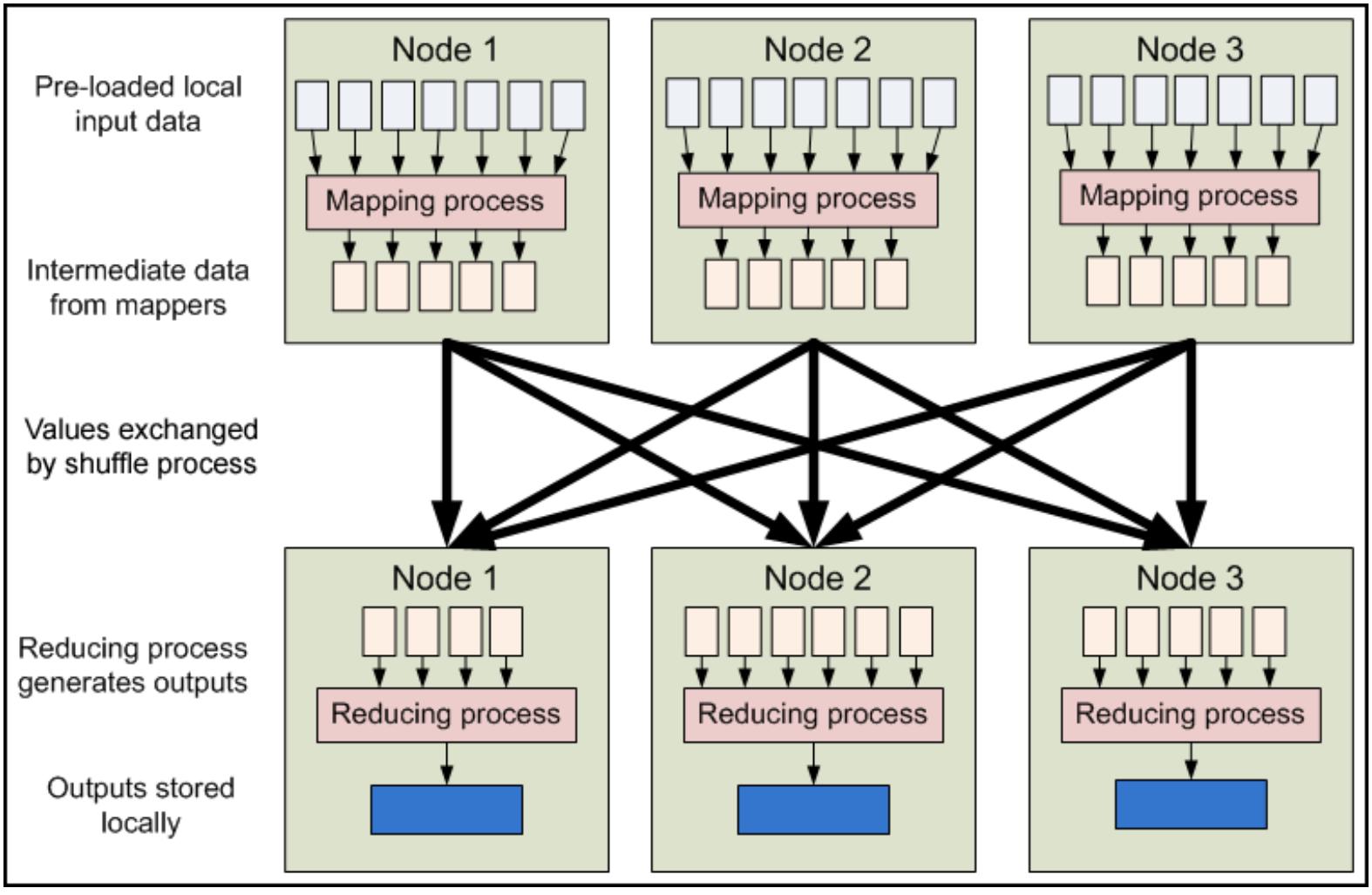
WordCount Example continued

```
/**
 * A reducer class that just emits the sum of the input values.
 */
public static class Reduce extends MapReduceBase
     implements Reducer<Text, IntWritable, Text, IntWritable>
{
    public void reduce(Text key, Iterator<IntWritable> values,
         OutputCollector<Text, IntWritable> output, Reporter reporter)
         throws IOException
    {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

WordCount Example continued

```
public static void main(String[] args) throws Exception
{
    if (args.length != 2) {
        printUsage();
        System.exit(1);
    }
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

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

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(new Path(args[0]));
    FileOutputFormat.setOutputPath(new Path(args[1]));

    JobClient.runJob(conf);
}
```

MapReduce: High-Level Data Flow



MapReduce Optimizations

- Overlap of maps, shuffle, and sort
- Mapper locality
  - Schedule mappers close to the data.
- Combiner
  - Mappers may generate duplicate keys
  - Side-effect free reducer can be run on mapper node
  - Minimizes data size before transfer
  - Reducer is still run
- Speculative execution to help with load-balancing
  - Some nodes may be slower
  - Run duplicate task on another node, take first answer as correct and abandon other duplicate tasks
  - Only done as we start getting toward the end of the tasks

Inverted Index Example

Given an input text, an inverted index program uses Mapreduce to produce an index of all the words in the text. For each word, the index has a list of all the files where the word appears.
The map method is shown below.

```
public static class InvertedIndexerMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, Text>
{
    private final static Text word = new Text();
    private final static Text location = new Text();

    public void map(LongWritable key, Text val,
            OutputCollector<Text, Text> output, Reporter reporter)
            throws IOException
    {
        FileSplit fileSplit = (FileSplit) reporter.getInputSplit();
        String fileName = fileSplit.getPath().getName();
        location.set(fileName);

        String line = val.toString();
        StringTokenizer itr = new StringTokenizer(line.toLowerCase());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, location);
        }
    }
}
```

Inverted Index Example (contd.)

The reduce method is shown below.

```
public static class InvertedIndexerReducer extends MapReduceBase
    implements Reducer<Text, Text, Text, Text>
{
    public void reduce(Text key, Iterator<Text> values,
            OutputCollector<Text, Text> output,
            Reporter reporter) throws IOException
    {
        boolean first = true;
        StringBuilder toReturn = new StringBuilder();
        while (values.hasNext()) {
            if (!first)
                toReturn.append(", ");
            first = false;
            toReturn.append(values.next().toString());
        }
        output.collect(key, new Text(toReturn.toString()));
    }
}
```

Inverted Index Example (contd)

```
public static void main(String[] args) throws IOException
{
    if (args.length < 2) {
    System.out
    println("Usage: InvertedIndex <input path> <output path>");
    System.exit(1);
    }
    JobConf conf = new JobConf(InvertedIndex.class);
    conf.setJobName("InvertedIndex");

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

    conf.setMapperClass(InvertedIndexerMapper.class);
    conf.setReducerClass(InvertedIndexerReducer.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    try {
        JobClient.runJob(conf);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Setting up Hadoop

- Download the latest stable version of Hadoop from `http://hadoop.apache.org/`.
- Unpack the tarball that you downloaded in previous step.
  `tar xzvf hadoop-0.20.2.tar.gz`
- Edit `conf/hadoop-env.sh` and at least set `JAVA_HOME` to point to Java installation folder. You will need Java version 1.6 or higher.
- Now run `bin/hadoop` to test Java setup. You should get output similar to shown below.

```
[amit@kohinoor hadoop-0.20.2]$ bin/hadoop
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
...
```

- We can use hadoop in three modes:
  - *Standalone mode*: Everything runs in a single process. Useful for debugging.
  - *Pseudo-distributed mode*: Multiple processes as in distributed mode but they all run on one host. Again useful for debugging distributed mode of operation before unleashing it on a real cluster.
  - *Distributed mode*: "The real thing!" Multiple processes running on multiple machines.

Standalone and Pseudo-Distributed Mode

- Hadoop comes ready to run in standalone mode out of the box. Try the following to test it.

```
mkdir input
cp conf/*.xml input
bin/hadoop jar wordcount.jar input output
cat output/*
```

- To run in pseudo-distributed mode, we need to specify the following:
  - The NameNode (Distributed Filesystem master) host and port. This is specified with the configuration property `fs.default.name`.
  - The JobTracker (MapReduce master) host and port. This is specified with the configuration property `mapred.job.tracker`.
  - The Replication Factor should be set to 1 with the property `dfs.replication`.
  - A slaves file that lists the names of all the hosts in the cluster. The default slaves file is `conf/slaves` it should contain just one hostname: `localhost`.
- Make sure that you can run `ssh localhost` command without a password. If you cannot, then set it up as follows:

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Pseudo-Distributed Mode

- To run everything in one node, edit three config files so that they contain the configuration shown below:

```
--> conf/core-site.xml
<configuration>
<property> <name>fs.default.name</name>
           <value>hdfs://localhost:9000</value> </property>
</configuration>
--> conf/hdfs-site.xml
<configuration>
<property> <name>dfs.replication</name>
           <value>1</value> </property>
</configuration>
--> conf/mapred-site.xml
<configuration>
<property> <name>mapred.job.tracker</name>
           <value>localhost:9001</value> </property>
</configuration>
```

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:
  `bin/hadoop namenode -format`
- Start the Hadoop daemons.
  `bin/start-all.sh`
- Put input files into the Distributed file system.
  `bin/hadoop dfs -put input input`
- Now run the distributed job and copy output back to local file system.
  `bin/hadoop jar wordcount.jar input output`
  `bin/hadoop dfs -get output output`
- Point your web browser to `localhost:50030` to watch the Hadoop job tracker and to `localhost:50070` to be able to browse the Hadoop DFS and get its status.
- When you are done, stop the Hadoop daemons as follows.
  `bin/stop-all.sh`

Fully Distributed Hadoop

Normally Hadoop runs on a dedicated cluster. In that case, the setup is a bit more complex than for the pseudo-distributed case.

- Specify hostname or IP address of the master server in the values for `fs.default.name` in core-site.xml and and `mapred.job.tracker` in mapred-site.xml file. These are specified as host:port pairs. The default ports are 9000 and 9001.
- Specify directories for `dfs.name.dir` and `dfs.data.dir` and dfs.replication in `conf/hdfs-site.xml`. These are used to hold distributed file system data on the master node and slave nodes respectively. Note that `dfs.data.dir` may contain a space- or comma-separated list of directory names, so that data may be stored on multiple devices.
- Specify `mapred.system.dir` and `mapred.local.dir` in `conf/hadoop-site.xml`. The system directory must be accessible by server and clients. The local directory determines where temporary MapReduce data is written. It also may be a list of directories.

Fully Distributed Hadoop (contd.)

- Specify `mapred.map.tasks.maximum` (default value: 2) and `mapred.reduce.tasks.maximum` (default value: 2) in `conf/mapred-site.xml`. This is suitable for local or pseudo-distributed mode only.
- Specify `mapred.map.tasks` (default value: 2) and `mapred.reduce.tasks` (default value: 1) in `conf/mapred-site.xml`.This is suitable for local or pseudo-distributed mode only. Choosing the right number of map and reduce tasks has significant effects on performance.
- Default Java memory size if 1000MB. This can be changed in `conf/hadoop-env.sh`. This is related to the parameters discussed above. See the Hadoop book by Tom White for further discussion.
- List all slave host names or IP addresses in your `conf/slaves` file, one per line. List name of master node in `conf/master`.

Sample Config Files

- Sample `core-site.xml` file.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>fs.default.name</name> <value>hdfs://node00:9000</value>
  <description>The name of the default filesystem.</description>
</property>
</configuration>
```

- Sample `hdfs-site.xml` file.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property> <name>dfs.replication</name> <value>1</value> </property>
<property>
  <name>dfs.name.dir</name><value>/tmp/hadoop-amit/hdfs/name</value>
</property>
<property>
  <name>dfs.data.dir</name> <value>/tmp/hadoop-amit/hdfs/data</value>
</property>
</configuration>
```

Sample Config Files (contd.)

- Sample `mapred-site.xml` file.

```
<?xml version="1.0"?> <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
<property>
  <name>mapred.job.tracker</name> <value>hdfs://node00:9001</value>
</property>
<property>
  <name>mapred.system.dir</name> <value>/tmp/hadoop-amit/mapred/system</value>
</property>
<property>
  <name>mapred.local.dir</name> <value>/tmp/hadoop-amit/mapred/local</value>
</property>
<property>
   <name>mapred.map.tasks.maximum</name> <value>17</value>
   <description>
      The maximum number of map tasks which are run simultaneously
      on a given TaskTracker individually. This should be a prime
      number larger than multiple of the number of slave hosts.
   </description>
</property>
<property>
   <name>mapred.reduce.tasks.maximum</name> <value>16</value>
</property>
<property>
   <name>mapred.reduce.tasks</name> <value>7</value>
</property>
</configuration>
```

Sample Config Files (contd.)

- Sample `hadoop-env.sh` file. Only need two things to be defined here.

```
# Set Hadoop-specific environment variables here.
...
export HADOOP_HOME=/home/faculty/amit/hadoop-install/hadoop
# The java implementation to use.  Required.
export JAVA_HOME=/usr/local/java/
...

# The maximum amount of heap to use, in MB. Default is 1000.
# export HADOOP_HEAPSIZE=1000

# Extra Java runtime options.  Empty by default.
# export HADOOP_OPTS=-server

...
```

References

- *MapReduce: Simplified Data Processing on Large Clusters* by Jeffrey Dean and Sanjay Ghemawat, Google Inc. Appeared in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- *Hadoop: An open source implementation of MapReduce*. The main website: http://hadoop.apache.org/.
- *Can Your Programming Language Do This?* by Joel Spolsky. http://www.joelonsoftware.com/items/2006/08/01.html
- *Hadoop: The Definitive Guide.* (2nd ed.) Tom White, October 2010, O'Reilly.
- *MapReduce tutorial at Yahoo*. http://developer.yahoo.com/hadoop/tutorial/
- *Hadoop Eclipse Plugin*: Jar file packaged with Hadoop in `contrib/eclipse-plugin` folder.
- *Data Clustering using MapReduce* by Makho Ngazimbi (supervised by Amit Jain). Masters in Computer Science project report, Boise State University, 2009.

New Hadoop API

The biggest change in 0.20 onwards is a large refactoring of the core MapReduce classes.

- All of the methods take *Context* objects that allow us to add new methods without breaking compatibility.
- *Mapper* and *Reducer* now have a "run" method that is called once and contains the control loop for the task, which lets applications replace it.
- *Mapper* and *Reducer* by default are Identity Mapper and Reducer.
- The *FileOutputFormats* use part-r-00000 for the output of reduce 0 and part-m-00000 for the output of map 0.
- The reduce grouping comparator now uses the raw compare instead of object compare.
- The number of maps in *FileInputFormat* is controlled by min and max split size rather than min size and the desired number of maps.

Hadoop added the classes in a new package at *org.apache.hadoop.mapreduce*.

WordCount example with new Hadoop API

Problem: To count the number of occurrences of each word in a large collection of documents.

```
/**
 * Counts the words in each line.
 * For each line of input, break the line into words
 * and emit them as (word, 1).
 */
public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
   }
}
```

WordCount Example with new Hadoop API (contd.)

```
/**
 * A reducer class that just emits the sum of the input values.
 */
 public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
 }
```

WordCount Example with new Hadoop API (contd.)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf,
                                    args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```