Big Data Analysis using Hadoop
Map-Reduce – An Introduction
Lecture 2

Last Week - Recap

What is Big Data?

O'Reilly/Reactor definition:
- Big data is when the size of the data itself becomes part of the problem

MDM/Gartner definition:
- Big Data technologies describe a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high velocity capture, discovery and/or analysis

Myth #1: Big Data has a clear definition

Fact:
- The term is used so often and in so many contexts that its meaning has become vague and ambiguous
- Industry experts and scientists often disagree

---

1. Create one file in the directory
2. Execute the script
3. View the output
4. Return to the copied file
5. Return to the copied directory
6. Return to the copied directory
7. Return to the copied directory
In this class

• Examine the Map-Reduce Framework

• What work each of the MR stages does
  • Mapper
  • Shuffle and Sort
  • Reducer

• Work through an Example illustrating what data is created and processed
  • Driver Class
  • Mapper Class
  • Reducer Class

• Create your first Map-Reduce job

Hadoop – Map-Reduce

MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster.

• It's a Framework!
• A set of code
  • With core parts
  • With customizable parts
  • With code you have specifically written to process the data

• Know and understand the Framework and you can process any data
• On Peta Bytes of data!
HDFS Architecture

MapReduce

- A batch based, distributed computing framework modelled on Google’s paper on MapReduce [http://research.google.com/archive/mapreduce.html]

- MapReduce decomposes work into small parallelised map and reduce tasks which are scheduled for remote execution on slave nodes

- Terminology
  - A **job** is a full programme
  - A **task** is the execution of a single map or reduce task over a slice of data called a **split**
  - A **Mapper** is a map task
  - A **Reducer** is a reduce task

- MapReduce works by manipulating key/value pairs in the general format

\[
\text{map}((\text{key}_1, \text{value}_1)) \rightarrow \text{list}(\text{key}_2, \text{value}_2) \\
\text{reduce}(\text{key}_2, \text{list}(\text{value}_2)) \rightarrow (\text{key}_3, \text{value}_3)
\]
A MapReduce Job

The input is divided into fixed-size pieces called input splits.

A map task is created for each split.

[from Hadoop in Practice, Alex Holmes]
A MapReduce Job

The role of the programmer is to define the Map and Reduce functions.

A MapReduce Job

The Shuffle & Sort phases between the Map and the Reduce phases combines map outputs and sorts them for the Reducers...

[from Hadoop in Practice, Alex Holmes]
A MapReduce Job

The Shuffle & Sort phases between the Map and the Reduce phases combines map outputs and sorts them for the Reducers...

The Reduce phase merges the data, as defined by the programmer to produce the outputs.

Map

- The Map function
  - The Mapper takes as input a key/value pair which represents a logical record from the input data source (e.g. a line in a file)
  - It produces zero or more outputs key/value pairs for each input pair
    - e.g. a filtering function may only produce output if a certain condition is met
    - e.g. a counting function may produce multiple key/value pairs, one per element being counted

\[
\text{map}(\text{in\_key}, \text{in\_value}) \rightarrow \text{list}(\text{temp\_key}, \text{temp\_value})
\]
Reduce

- The Reducer(s)
  - A single Reducer handles all the map output for a unique map output key
  - A Reducer outputs zero to many key/value pairs
  - The output is written to HDFS files, to external DBs, or to any data sink...

\[ \text{reduce(temp_key, list(temp_values)} \rightarrow \text{list(out_key, out_value)} \]

MapReduce

- JobTracker - (Master)
  - Controls MapReduce jobs
  - Assigns Map & Reduce tasks to the other nodes on the cluster
  - Monitors the tasks as they are running
  - Relaunches failed tasks on other nodes in the cluster

- TaskTracker - (Slave)
  - A single TaskTracker per slave node
  - Manage the execution of the individual tasks on the node
  - Can instantiate many JVMs to handle tasks in parallel
  - Communicates back to the JobTracker (via a heartbeat)
A MapReduce Job

[From Hadoop in Practice, Alex Holmes]
Monitoring progress

YARN (Yet Another Resource Negotiator) Framework
Data Locality

“This is a local node for local Data”

- Whenever possible Hadoop will attempt to ensure that a Mapper on a node is working on a block of data stored locally on that node via HDFS

- If this is not possible, the Mapper will have to transfer the data across the network as it accesses the data

- Once all the Map tasks are finished, the map output data is transferred across the network to the Reducers

- Although Reducers may run on the same node (physical machine) as the Mappers there is no concept of data locality for Reducers

Bottlenecks?

- Reducers cannot start until all Mappers are finished and the output has been transferred to the Reducers and sorted

- To alleviate bottlenecks in Shuffle & Sort - Hadoop starts to transfer data to the Reducers as the Mappers finish
  - The percentage of Mappers which should finish before the Reducers start retrieving data is configurable

- To alleviate bottlenecks caused by slow Mappers - Hadoop uses speculative execution
  - If a Mapper appears to be running significantly slower than the others, a new instance of the Mapper will be started on another machine, operating on the same data (remember replication)
  - The results of the first Mapper to finish will be used
  - The Mapper which is still running will be terminated by Hadoop
The MapReduce Job

Let us build up an example
The Scenario

- Build a Word Counter
- Using the Shakespeare Poems
- Count the number of times a word appears in the data set
- Use Map-Reduce to do this work
- Step-by-Step of creating the MR process
Setting up the MapReduce Job

- A Job object forms the specification for the job

- Job needs to know:
  - the jar file that the code is in which will be distributed around the cluster: `setJarByClass()`
  - the input path(s) (in HDFS) for the job: `FileInputFormat.addInputPath()`
  - the output path(s) (in HDFS) for the job: `FileOutputFormat.setOutputPath()`
  - the Mapper and Reducer classes: `setMapperClass()` `setReducerClass()`
  - the output key and value classes: `setOutputKeyClass()` `setOutputValueClass()`
  - the Mapper output key and value classes if they are different from the Reducer: `setMapOutputKeyClass()` `setMapOutputValueClass()`
  - the name of the job, default is the name of the jar file: `setJobName()`

- The default input considers the file as lines of text
  - The default key input is `LongWritable` (the byte offset into the file)
  - The default value input is `Text` (the contents of the line read from the file)

Driver Code

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

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

        Job job = Job.getInstance();
        job.setJarByClass(WordCount.class);
        job.setJobName("WordCount");
        job.setMapperClass(WordMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```
Driver Code

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

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

You will typically import these classes into every MapReduce job you write. We will omit the import statements in future slides for brevity.

Driver Code

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

        Job job = Job.getInstance();
        job.setJarByClass(WordCount.class);
        job.setJobName("WordCount");
        job.setMapperClass(WordMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
Driver Code

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

        Job job = Job.getInstance();
        job.setJarByClass(WordCount.class);
        job.setJobName("WordCount");
        job.setMapperClass(WordMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```
Driver Code

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

        Job job = Job.getInstance();
        job.setJarByClass(WordCount.class);
        job.setJobName("WordCount");
        job.setMapperClass(WordMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Specify the input directory (where the data will be read from) and the output directory where the data will be written.

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
File formats - Inputs

• The default InputFormat (TextInputFormat) will be used unless you specify otherwise

• To use an InputFormat other than the default, use e.g.
  conf.setInputFormat(KeyValueTextInputFormat.class)

• By default, FileInputFormat.setInputPaths() will read all files from a specified directory and send them to Mappers
  • Exceptions: items whose names begin with a period (.) or underscore (_)
  • Globs can be specified to restrict input
    • For example, /2010/*/01/*

File formats - Outputs

• FileOutputFormat.setOutputPath() specifies the directory to which the Reducers will write their final output

• The driver can also specify the format of the output data
  • Default is a plain text file
  • Could be explicitly written as
    conf.setOutputFormat(TextOutputFormat.class);
Driver Code

```java
public class WordCount {

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

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

        job.setMapperClass(WordMapper.class);
        job.setReducerClass(SumReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

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

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

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

- The Mapper takes as input a key/value pair which represents a logical record from the input data source (e.g. a line in a file).
- The Mapper may use or ignore the input key:
  - E.g. a standard pattern is to read a file one line at a time
  - Key = byte offset into the file where the line starts
  - Value = contents of the line in the file
  - Typically the key can be considered irrelevant
- It produces zero or more outputs key/value pairs for each input pair:
  - e.g. a filtering function may only produce output if a certain condition is met
  - e.g. a counting function may produce multiple key/value pairs, one per element being counted

Mapper Class

- extends the Mapper <K1, V1, K2, V2> class
- key and value classes implement the WriteableComparable and Writeable interfaces
- most Mappers override the map method which is called once for every key/value pair in the input
- void map (K1 key, V1 value, Context context) throws IOException, InterruptedException
- the default map method is the Identity mapper - maps the inputs directly to the outputs
- in general the map input types K1, V1 are different from the map output types K2, V2
Mapper Class

- Hadoop provides a number of Mapper implementations:

  **InverseMapper** - swaps the keys and values

  **TokenCounterMapper** - tokenises the input and outputs each token with a count of 1

  **RegexMapper** - extracts text matching a regular expression

Example:

```java
job.setMapperClass(TokenCounterMapper.class);
```

**Mapper Code**

```java
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String s = value.toString();
        for (String word : s.split("\W+")) {
            if (word.length() > 0) {
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

**Inputs**

Processes the input text

**Outputs**

Writes the outputs
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String s = value.toString();
        for (String word : s.split("\W+")) {
            if (word.length() > 0) {
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```java
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String s = value.toString();
        for (String word : s.split("\\W+")) {
            if (word.length() > 0) {
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

Processes the input text

```java
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String s = value.toString();
        for (String word : s.split("\\W+")) {
            if (word.length() > 0) {
                context.write(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

Writes the outputs
What the mapper does

• Input to the Mapper:

  {"this one I think is called a yink"}
  {"he likes to wink, he likes to drink"}
  {"he likes to drink and drink and drink"}

• Output from the Mapper:

  {"this, 1"
   "one, 1"
   "I, 1"
   "think, 1"
   "is, 1"
   "called, 1"
   "a, 1"
   "yink, 1"
   "he, 1"
   "likes, 1"
   "to, 1"
   "wink, 1"
   "he, 1"
   "likes, 1"
   "to, 1"
   "drink, 1"
   "he, 1"
   "likes, 1"
   "to, 1"
   "drink, 1"
   "and, 1"
   "drink, 1"
   "and, 1"
   "drink, 1"}

Shuffle and sort

• Shuffle
  • Integrates the data (key/value pairs) from outputs of each mapper
  • For now, integrates into 1 file

• Sort
  • The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer
  • Sorted within key

• Determines what subset of data goes to which Reducer
Reducer Class

- extends the Reducer <K2, V2, K3, V3> class
  - key and value classes implement the WriteableComparable and Writeable interfaces
    - void reduce (K2 key, 
      Iterable<V2> values, 
      Context context) throws IOException, InterruptedException
    - called once for each input key
    - generates a list of output key/values pairs by iterating over the values associated with the input key
      - the reduce input types K2, V2 must be the same types as the map output types
      - the reduce output types K3, V3 can be different from the reduce input types
      - the default reduce method is the Identity reducer - outputs each input/value pair directly
  - getConfiguration() - access the Configuration for a Job
  - void setup (Context context) - called once at the beginning of the reduce task
  - void cleanup (Context context) - called at the end of the task to wrap up any loose ends, closes files, db connections etc.
    - Default number of reducers = 1
Reducer Class

- Hadoop provides some Reducer implementations

**IntSumReducer** - sums the values (integers) for a given key

**LongSumReducer** - sums the values (longs) for a given key

Example:

```java
job.setReducerClass(IntSumReducer.class);
```

http://hadooptutorial.info/predefined-mapper-and-reducer-classes/


Reducer Code

```java
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {  
    public void reduce(Text key, Iterable<IntWritable> values, Context context)  
        throws IOException, InterruptedException {  
        int wordCount = 0;  
        for (IntWritable value : values) {  
            wordCount += value.get();  
        }  
        context.write(key, new IntWritable(wordCount));  
    }  
}
```
Reducer Code

```java
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int wordCount = 0;
        for (IntWritable value : values) {
            wordCount += value.get();
        }
        context.write(key, new IntWritable(wordCount));
    }
}
```

Processes the input text
Reducer Code

public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int wordCount = 0;
        for (IntWritable value : values) {
            wordCount += value.get();
        }
        context.write(key, new IntWritable(wordCount));
    }
}

Writes the outputs
Map-Reduce – Example 2

Map-Reduce – Example 3
Lab work

- Take the map-reduce code from these notes and get it running on your Hadoop VM
  - Driver Code
  - Mapper Code
  - Reducer Code

- Explore the results in HDFS and on the web interface.

- Additional exercises.

- Complete before next week.

- More MR labs next week and Assignment Handout.