Computing with MapReduce

Introduction to Analytics using Hadoop: Week Three Lecture

Many common algorithms can be implemented in MapReduce in order to perform computations across large or raw data sets. MapReduce algorithms are generally larger algorithms decomposed into smaller MapReduce tasks and chained together. Because of this, jobs rely on a set of commonly used Mappers or Reducers, and take advantage of the built in sort and shuffle in Hadoop's MapReduce API.

Although may of these computations become slightly more complex when computed with MapReduce rather than some other programming paradigm, the key feature is the ability to leverage a distributed computing environment to operate across big data. With this in mind, it becomes important to consider memory constraints, particularly in our Reducers, and this will be a common theme throughout this section.

Through this lecture, we'll cover some of the basic Mappers and Reducers, many of which you have already seen in examples. From there we'll discuss computational techniques for a few common MapReduce algorithms, and explore where to go to look for more advanced computations.

Table of Contents

Basic Map and Reduce 3
- **Combiners** 7
- **Partitioners** 8
- **Composite Keys** 9
- **Job Chaining** 12

MapReduce Patterns 15
- **The Identity Pattern** 16
- **Statistical Patterns** 16
- **Pairs vs. Stripes** 21

Some Common MapReduce Algorithms 24
- **Sorting** 24
- **Searching** 25
- **Classification** 26
- **TF-IDF** 26

© Benjamin Bengfort and Statistics.com, LLC

Conclusion **29**
Assignments **31**

As we move into more formal programming and start writing MapReduce applications, it's important to note that the code samples in this section will be pseudo-code representations of the important design concepts for creating mappers and reducers, except where implementation specifics are noted. On request, executable code samples in various languages can be provided, and I would highly encourage students to discuss implementations in their various languages on the discussion board! However, I will attempt to be as general as possible in this discussion to include a wide range of skills and backgrounds.

Just as a reminder- the core MapReduce API for Hadoop is implemented in Java. Most Hadoop jobs are compiled as JARs and directly submitted to the cluster. Jobs written in Java can take advantage of managing counters, partitioners, and other execution contexts through the API. However, there is a powerful option for other programming languages – Hadoop Streaming. Hadoop Streaming allows you to write MapReduce jobs as any executable that can read from standard input and write to standard output. For many tasks, this is all that is needed, and there are many libraries available for various languages to allow you to rapidly develop jobs for the cluster.

The goal of this section is to introduce basic MapReduce design patterns and algorithms, without relying on one method of executing jobs or the other. However, there are some concepts that must be explained relative to the Java API. Hadoop streaming users should still take note of these concepts, as they will be important to understand when leveraging Hadoop's internal tools through options passed to Hadoop streaming JAR, and to understand when Hadoop Streaming is simply not enough for a particular job.

Basic Map and Reduce

The basic template for producing any MapReduce job consists of a minimum of the following code components:

1. A Mapper
2. A Reducer
3. A Driver

For review, mappers and reducers can be defined as follows:

```
def map(inputKey, inputValue) {
   // Compute based on inputs
   emit(outputKey, outputValue)
}
```

```
def reduce(outputKey, iteratorValues) {
   // Reduce by outputKey for all values
   emit(finalKey, finalValue)
}
```

The mapper takes a single input key/value pair, performs some computation, and then outputs one or more output key/value pairs. These outputs are aggregated on the output key and the values for each key are sent to a reducer function as a key/iterable values pair. The reducer then computes the final result for the given key and outputs one or more final key/value pairs. Hadoop's implementation of MapReduce also has an intermediary sort and shuffle between the map and reduce phases, which can be leveraged in many algorithms.

The final piece of a MapReduce job is the driver. The driver is the code that runs on the client to configure and submit the job to the cluster. For the most part the driver's job is to specify the following items:

1. The `InputFormat`: the definition of the location and type of input data
2. The Mapper(s), including the input and output key/value types
3. The Reducer(s), and reducer input and output key/value types
4. The `OutputFormat`: the definition of the location and type of output data

Drivers can also be used to accept command line arguments to pass to the job, for instance the location of a stopwords file, or other optional settings. Users of the Java API must specify drivers as a particular class that includes a main method. This class is also the first argument when passing the JAR as a job to the cluster.

For Hadoop Streaming, the streaming JAR allows you to specify the mapper, reducer, and even the input and output formats when executing the job. For many purposes, executing the streaming JAR with options can be considered the driver component of the job, however, there are frameworks that wrap Hadoop streaming that may implement a more formal driver pattern.

By default, the InputFormat is the `TextInputFormat`, which treats each `\n`-terminated line of the file as a value (the key is the byte offset within the file). However, there are some other common InputFormats including:

- `KeyValueTextInputFormat`: maps `\n`-terminated lines as "key SEP value" where tab (`\t`) is the key by default, but can be specified.
- `SequenceFileInputFormat`: a binary file of key, value pairs with additional metadata included.
- `CompressionFileFormat`: decompresses inputs appropriately.

Another thing to consider with the MapReduce Java API is that keys and values are `Objects`, which means they need to implement the `Writable` interface in order to be serialized between processes. Furthermore, keys need to implement `WritableComparable` so that they can be sorted and shuffled correctly. There are many default `Writable` classes including `IntWritable`, `LongWritable`, `DoubleWritable`, and `Text` for various types in Java.

Although it seems like Hadoop Streaming users might not have to consider Java API definitions, it is important to take notice. For instance, passing keys and values between standard input and standard output means that all streaming keys and values are `Text` writable and serialized as such. Processes

that leverage Hadoop streaming will have to correctly cast values with no knowledge of the incoming format, which can lead to errors. Additionally, Hadoop Streaming can supplement various options with built-in Java classes for partitioners, etc. (but only the mapper, reducer, and combiner can be implemented with a different programming language).

To get a feel for how drivers are implemented with Java (for both Java API users and streaming users), consider the following driver for the airport average arrival delays:

```
public class OnTime {

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

        // Set up the Job
        Job job = new Job();
        job.setJarByClass(OnTime.class);
        job.setJobName("Flight On Time Performance");

        // Sepcify input and output on HDFS
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // Set the Mapper and Reducer
        job.setMapperClass(OnTimeMapper.class);
        job.setReducerClass(OnTimeReducer.class);

        // Set output Key/Value classes
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

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

First, we check to make sure we received two arguments from the command line- we're expecting the first argument to be the input path and the second argument to be the output path. We create the job with a few options, then add the input and output paths from the arguments. Next we set the Mapper

and Reducer classes, and set the output key and output value classes (leaving the map inputs the default). Finally, we execute the job.

A similar inspection of the command line method for executing a Hadoop Streaming job reveals that similar configuration options are being passed to the streaming JAR, which is acting as the streaming driver.

```
$ hadoop jar \
  $HADOOP_PREFIX/contrib/streaming/hadoop-*streaming*.jar \
    -file $MAPPER -file $REDUCER \
    -mapper $MAPPER -reducer $REDUCER -combiner $REDUCER \
    -input $1 -output $2
```

The hadoop-streaming JAR accepts command line arguments specifying the mapper, reducer, and the combiner (more on this in a bit) as well as the input and output directives. The input and output formats are TextFormats, and the keys are also Text by default with streaming. Also note that any files that are used in Hadoop Streaming must be specified by the `-file` option, which tells Hadoop to transmit those files to each node in the cluster for execution.

Combiners

I've briefly mentioned combiners, as a specification for MapReduce jobs; let's delve into this a bit more. Combiners are intended as intermediary reducers associated with one particular Map task. Consider the fact that Mappers produce a large amount of intermediary data that must be sent to the Reducers over the network, this can result in delays as well as memory bottlenecks. Combiners ease this problem by performing a mapper-local reduce-like task before sending the output the Reducer. Consider the following:

Mapper one output:

```
(IAD, 14.4), (SFO, 3.9), (JFK, 3.9), (IAD, 12.2), (JFK,
5.8)
```

Mapper two output:

```
(SFO, 4.7), (IAD, 2.3), (BWI, 4.4), (IAD, 1.2)
```

Intended sum reduce output:

```
(IAD, 29.1), (JFK, 9.7), (SFO, 8.6), (BWI, 4.4)
```

If a sum combiner is applied to each Mapper output, the sums of the duplicate IAD and JFK will be computed before being sent across the network to the Reducer, therefore reducing the amount of traffic as well as making the shuffle and sort operations that much faster.

It is extremely common for the combiner and the reducer to be identical, which is possible if the operation is commutative and associative, but this is not always the case. However, the input and output data types for the combiner and the reducer must be identical. The combiner must output zero or more key, value pairs, and is specified by a `reduce` method which is called to perform this operation.

One caveat, however, is that code should never be put in the Combiner that must be run as part of the MapReduce job. The combiner may not be run on the output for all Mappers, and is intended only as an intermediary reducer to increase performance and optimize jobs.

Partitioners

Partitioners control how keys and their values get sent to individual Reducers by dividing up the keyspace. The default behavior is the `HashPartitioner`, which is often all that is needed. This partitioner allocates keys evenly to each Reducer by computing the hash of the key and assigning the key to a keyspace determined by the number of reducers.

However, there are times where developers must write their own partitioners. Consider the case of composite keys, `(a, b)` where it is desired that each reducer not only sees all the values for the key `(a, b)` but further, it sees all keys that start with `a`. In this case, even if (a, b) is hashable (a requirement in Java for custom `WritableComparable`) it doesn't guarantee that every key that starts with `a` goes to the same partitioner.

Additionally, partitioners can also be used to improve performance (e.g. if there are many key/value pairs for a particular keyspace and a significant portion of the Reducers are underworked). Custom partitioners are also required for some MapReduce algorithms, notably during joins, or to produce sectioned outputs on a particular key pattern.

Unfortunately, custom Partitioners can only be created with the Java API. However, Hadoop Streaming users can still specify a partitioner java class either from the Hadoop library, or by writing their own Java partitioner and submitting it with their streaming job. The pseudocode for a partitioner is as follows:

```
def getPartition(key, value, numReducers) {
   // Compute keyspace based on key and value
   return 0 <= int < numReducers;
}
```

Composite Keys

Although Java users are used to strict typing, it is worth considering further that keys and values passed to and from Mappers and Reducers must implement the `Writable` interface, and keys further must implement the `WritableComparable` interface. This is intended to provide a very fast and extensible serialization for data structures to be passed across the network to various task processes, as well as provide fast shuffle and sorting.

One common paradigm for many MapReduce key/value inputs and outputs is that of composite keys and values. In fact, we've already seen an example of this in the LogReader exercise, where we grouped our key not just by the hour of the day, but also by the source of the request (e.g. remote or local). Additionally, computation may rely on composite values, not simply a single value. Unfortunately, it seems that in the Java API there are only primitive Writable types like `IntWritable` and `Text`!

For Hadoop Streaming users, the keys and values are always Text, unless specified by using a library that has a special serialization (like Dumbo) or

providing a different serialization method to the hadoop-streaming JAR. Hadoop streaming users can still specify composite keys and values by emitting a string representation of the composite with some sort of delimiter, e.g.:

```
emit key[0], key[1]    value
```

Typical delimiters include the TAB character ("\t") and COMMA (",") as in other textual data representations like TSV and CSV files. Hadoop streaming users are therefore also required to cast the values from strings to their appropriate format, which is not always onerous, particularly if the data is already in a text format.

Java API users can also use the String method of creating composite keys and values. However, this is at the cost of the performance boost with Hadoop-native serialization. Indeed, concerns about quoting (if part of the value contains the separation character), and difficulties with binary objects means that they should take advantage of the API by writing a custom `WritableComparable`. Example code is below for a 2D point:

```
import org.apache.hadoop.io.*

private IntWritable x;
private IntWritable y;

public class Point() {
   set(new IntWritable(), new  IntWritable())
}

public class Point(int x, int y) {
   set(new IntWritable(x), new IntWritable(y));
}

public void set(IntWritable x, IntWritable y) {
   this.x = x;
   this.y = y;
}

public IntWritable getX()  { return this.x; }

public intWritable getY() { return this.y; }
```

```
@Override
public void write(DataOutput out) throws IOException {
   x.write(out);
   y.write(out);
}



@Ovverride
public void readFields(DataInput in) throws IOException {
   x.readFields(in);
   y.readFields(in);
}

@Override
public int hashCode() {
   // Some spatial hashing algorithm like Z-Curve
}

@Override
public boolean equals(Object o) {
   Point p = (Point) o;
   return p.x == this.x && p.y == this.y;
}

@Override
public Boolean compareTo(Point p) {
   // some Euclidean comparison like distance to origin
}

@Override
public String toString() {
   return String.format("(%d, %d)", this.x, this.y);
}
```

Objects that implement `Writable` must provide definitions for the `write` and `readFields` methods, which are essentially ordered reads off the input and output buffer of lengths specific to the type. Be aware- *all* Writable objects must supply a default constructor so that the framework can instantiate them and populate values with `readFields`. Additionally, Writable objects should specify the `toString` method so that they can be written to text output files. Objects that implement `WritableComparable` must further implement `compareTo`, `equals`, and `hashCode`. `hashCode` is used in the paritioner and `equals` and `compareTo` are used during sorting.

For composite values, it may not be necessary to write a custom `Writable` as there are several Writable collection types including `ArrayWritable` and `MapWritable`. These collections, however, do not implement the `WritableComparable` interface, and therefore cannot be used as intermediate or output keys. Even so, it may be more meaningful to your applications to have a specific `Writable` data structure that is passed between jobs.

It seems that Hadoop-streaming users have gotten off easy with text based composite keys and the cost of some performance. However, performance boosts with native serialization could mean job differences in hours. If you're using a streaming library, it is likely that they have included JAR files for improved serialization. Hadoop Streaming also allows for byte streaming, which can be useful to leverage object serialization in the language of your choice.

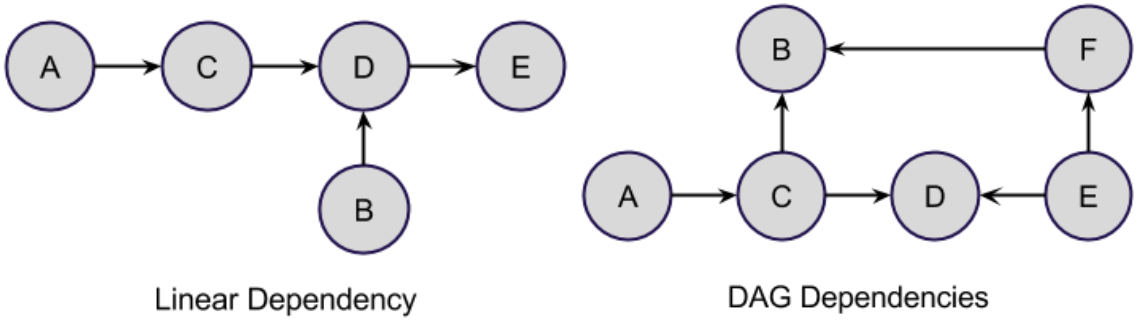
It is also generally useful to consider other data structures for serialization when using either Hadoop Streaming or the Java API. One common structured data representation is to use Base 64 radix encoded JSON strings. However, for the most part, the fastest job implementations will simply use Tuples or custom objects for serialization.

Job Chaining

Many algorithms implemented in MapReduce are decomposed into several smaller tasks and chained together as a sequence of several small jobs whose combined output leads to the full algorithms. Implementing algorithms like this requires the developer to think about how each individual step of a computation can be reduced to intermediary values not just between Mappers and Reducers but also between MapReduce jobs.

There are two types of job chaining. Typically the output from the first job is sent to the second job and so on until the data is written to some final output; this is called linear job chaining. Linear dependencies mean each

MapReduce job has dependencies on one more previous jobs. More complex job dependencies can be expressed as directed acyclic graphs (DAGs)



Consider the following simple problem: determine the top 100 flights (airport to airport) that have the best on time performance. The first job is to compute the average delay per flight for the data period; the second job then sorts the job by delay and filters out the top 100.

```
def firstMapper(key, value) {
   emit flight, (delay, 1)
}

def firstReducer(key, values) {
   total = 0;
   count = 0;
   for (value in values) {
         total += value[0];
         count += value[1];
   }
   emit key, total/count;
}



def secondMapper(key, value) {
   emit value, key
}

def secondReducer(key, values) {
   count = 0;
   while (count < 100) {
       for (value in values) {
           count += 1;
           emit value, key
       }
   }
}
```

Note that the second mapper utilizes the built-in sort and inverts the key and value to ensure that the reducers receive the flights in highest mean order. To ensure that this particular job will work, only a single Reducer can be used.

There are a couple of ways to chain jobs together in this fashion. The first is to utilize a `JobControl` object can be created to represent each job directly in your drivers. Use `JobControl.addDependingJob()` and `JobControl.addJob()` Many Hadoop-Streaming libraries also support this kind of job chaining.

However, as jobs get more complex, I would recommend using Apache Oozie- a Hadoop workflow control system that comes packaged with many Hadoop distributions. Although we didn't install Oozie as part of our installation on the cluster, it is worth a brief mention. Workflow definitions in Oozie are written using a DSL (domain specific language), an XML format called Hadoop Process Definition Language (HPDF). Workflows define a start and end node, and workflows proceed through transitions by determining the success and failure of intermediary jobs. You collect your jobs into HPDF, and submit the HPDF to Oozie, which will execute and monitor the jobs for you.

Map-Only Jobs

When considering job chains and job dependencies, it's helpful to note that the possibility of Map-only jobs exists. Map only jobs fall into two categories: those where you would require no aggregation and those where you are actively trying to avoid the shuffle and sort phase- either to maintain data order or to optimize the execution of the job.

To execute a Map-only job, simply set the number of Reducers to 0. In a Java driver this would be done with `job.setNumReduceTasks(0);` with Hadoop Streaming, you can specify the number of reducers via the `-numReduceTasks` flag. Reduce-only jobs are also possible, by using the `IdentityMapper`, as

discussed in the next section. "Map-only" jobs that require sorting can be executed with a Reducer, in this case the `IdentityReducer`.

MapReduce Patterns

Now that we've looked at some of the critical parts of MapReduce jobs, let's take a look at a few example algorithms and patterns that are used on a regular basis to construct larger patterns or algorithms. For the most part, these Mappers and Reducers will become extremely familiar to you, and it's useful to know them by heart.

The Explode Mapper

This mapper divides a value into its constituent parts:

```
def map(key, value) {
   foreach v in value:
       emit (key, v);
}
```

The Filter Mapper

This mapper only outputs certain values if they meet a particular criterion. For example, this mapper only yields prime numbers:

```
def map(key, value) {
   if (isPrime(value)) {
       emit (key, value);
   }
}
```

It is perfectly acceptable for Mappers to not emit keys or values.

Keyspace Change

Key/Value inversion or changing the space of the key or the key type is also a very common mapper pattern, particularly in chained jobs. We've already seen an example where we inverted the key in the mapper to leverage the built in sort of a Hadoop job, then reinverted the key in the Reducer.

```
def map(key, value) {
   emit (v.length(), v);
```

```
}
```

Reducers typically fall into more typical aggregation patterns, and we'll discuss those more in the following section.

The Identity Pattern

One important pattern is called the Identity pattern. This is simply a pass-through pattern that is typically implemented on Reducers, which makes the job equivalent to a sort. In functional terms, identity relations return the same value as their input, e.g. `f(x) = x`. Although `IdentityReducers` are more common, `IdentityMappers` are also important to many chained MapReduce jobs where the output of one Reducer must immediately be sent to a secondary Reducer.

```
def identityMapper(key, value) {
   emit (key, value);
}

def identityReducer(key, values) {
   foreach (value in values) {
       emit (key, value);
   }
}
```

Statistical Patterns

Like the Identity pattern, many statistical aggregations like Sum, Mean, Minimum, and Maximum, also make up smaller components of larger jobs and algorithms. The `SumReducer` combined with a `CountMapper` is the foundation of many probability algorithms (by creating frequency distributions) as well as other patterns that involve search and significance.

Count Mapper

The count mapper simply yields a one for every value that it sees. This is often a combination of explode and keyspace change patterns, e.g. count all the items in a particular value. We have already seen this effectively used in our word count analysis.

```
def countMapper(key, value) {
```

```
   for (v in value) {
       emit (v, 1)
   }
}
```

SumReducer

In order to effectively utilize a Count Mapper, we need sum way to sum the values. Again, from the word count example, we see the very popular `SumReducer`.

```
def sumReducer(key, values) {
   sum = 0;
   for (int i in vals) {
       sum += 1;
   }
   emit (key, sum);
}
```

This reducer can also be used as a combiner.

MaxReducer and MinReducer

Finding the largest and smallest values for a key is also an important task, particularly as memory can be at a premium for large Jobs. Sampling the largest or smallest values is often a way to reduce task computational space, or to provide smaller sets to dependent jobs.

A similar task that I will leave as an exercise to the reader is to determine the N largest or N smallest values for a key. To do this, the values iterable must be added to an ordered, in-memory data structure, the most popular choice being the `heap` data structure.

```
def maxReducer(key, values) {
   maximum = null;
   for (val in sorted(values) {
       if (maximum == null || val > maximum) {
           maximum = val;
       }
   }
   emit (key, maximum)
}

def minReducer(key, values) {
```

```
   minimum = null;
   for (val in sorted(values) {
       if (minimum == null || val < minimum) {
            minimum = val;
       }
   }
   emit (key, minimum)
}
```

AverageReducer

Computing the mean of values for a particular key is another simple task that involves a simple extension of the `SumReducer`. We've already seen this computation at work in our top 100 most delayed flights example in the jobs section, but more formally computing the mean for a set of keys is as follows:

```
def map(key, value) {
   emit (key, value);
}

def reduce(key, value) {
   total = 0;
   count = 0;
   for (value in values) {
       total += value;
       count += 1
   }
   emit (key, total/count);
}
```

However, in order to use a combiner with the mapper, we can't use the same reducer without a loss in precision. A slightly better implementation of the Mean of Values computation is therefore to use an intermediary `SumCombiner`. Since the combiner must accept and output the same intermediary key and values as the mapper's output key/value types, we have to update the mapper slightly to output count values, rather than simply guessing our count in the reducer.

```
def map(key, value) {
   emit (key, (value, 1));
}
```

```
def combine(key, values) {
   total = 0;
   count = 0;
   for (val, num in values) {
       total += val;
       count += num;
   }
   emit (key, (total, count))
}

def reduce(key, values) {
   total = 0;
   count = 0;
   for (val, num in values) {
       total += val;
       count += num;
   }
   emit (key, total/count);
}
```

Computing Statistical Metrics

At this point, we've discussed some very simple mappers and reducers in order to give a sense of how to implement common computations in a MapReduce format, and also to ensure that we understand basic mapping and aggregation computations that will make up our larger computations. From here, I could provide individual MapReduce jobs to go farther to compute variance, then standard deviation, etc. However, it should be clear by now how to formulate these individual computations.

However, if every key from our map job can be viewed as a sub data set, it can be useful to compute the statistical metrics: count, sum, mean, standard deviation, and range (minimum and maximum) on a per key basis. For instance, consider the On Time Performance data set, if we map each sub data set to the origination airport, we can compute these metrics on a per-airport basis and get a feel for how airports compare to each other. Rather than run six jobs to compute all these values, we can simply use the Statistics Job below:

```
def map(key, value) {
   emit(key, (1, value, Math.pow(value, 2), value, value));
}
```

```
def combine(key, values) {
   total   = 0;
   count   = 0;
   square  = 0;
   minimum = null;
   maximum = null;
   for ( (cnt, val, sqr, min, max) in values ) {
       total += val;
       count += cnt;
       square += sqr;
       if (minimum == null || min < minimum) {
           minimum = min;
       }
       if (maximum == null || max > maximum) {
           maximum = max;
       }
   }
   emit (key, (count, total, square, minimum, maximum));
}
```

The mapper emits a value for each column of our computation, the count, the value, the square of the value, and then emits the value twice again in order to create placeholders for the minimum and maximum value computations. The combiner assists the reducer by computing the total count, the sum of values, the sum of the squares, and determining the minimum and maximum values for each Mapper. The Reducer finally outputs the count, the total sum, the mean, the standard deviation, and the range as follows:

```
def reduce(key, values) {
   total   = 0;
   count   = 0;
   square  = 0;
   minimum = null;
   maximum = null;
   for ( (cnt, val, sqr, min, max) in values ) {
       total += val;
       count += cnt;
       square += sqr;
       if (minimum == null || min < minimum) {
           minimum = min;
       }
       if (maximum == null || max > maximum) {
           maximum = max;
       }
   }
```

```
   mean   = total / count;
   stddev = Math.sqrt((square-Math.pow(total, 2)/count)
                      /count-1);

   emit(key, (count, total, mean, stddev, minimum, maximum))
}
```

This type of job is our first example of a complex data type as an output value. Developers of jobs like these must consider how to serialize the intermediary and output keys from this job, as from our previous discussion about `Writable` objects and serializing complex data structures. The output value must also be considered, as this Job is certainly the preliminary component to other statistical evaluations that rely on these metrics.

Pairs vs. Stripes

The final pattern we'll discuss before we get into some common MapReduce algorithms is matrix representations via MapReduce. Matrix computations are common for larger algorithms, and rely on jobs that produce either pair or stripe based representations of matrices.

Consider the problem of building a word co-occurrence matrix for a particular text based corpus[1]. MapReduce is well suited for linguistic and natural language processing tasks, as NLP is embarrassingly parallel- one of the features of MapReduce jobs. Word co-occurrences (or word N-Grams as they're called in NLP) create a statistical model of language that can be used in many language applications including sentence generation, translation, etc.

The co-occurrence matrix of a corpus is a square matrix of size `NxN`, where `N` is the vocabulary (the number of unique words) in the corpus. Each cell `mij` contains the number of times word `wi` co-occurs with word `wj` within a specific context (either a sentence, a paragraph, or a document). In order to

[1] Although we won't cover this algorithm in detail, see the following blog post for more: http://chandramanitiwary.wordpress.com/2012/08/19/map-reduce-design-patterns-pairs-stripes/

process the MapReduce job to compute this matrix, the final output can be accomplished via two approaches: pairs or stripes.

In general the pair approach is to emit every cell in the matrix from the mapper; the combiner and reducer work on the cell values in order to produce a final, per-cell computation. This reasonable approach yields a data output where each `mij` pair are computed and stored separately.

```
def map(key, value) {
   for (i, word in enumerate(value)) {
       for (j, term in enumerate(value)) {
           if (i==j) { continue; }
           emit ((word, term) 1);
   }
}
```

For the input:

```
"See Spot run, run Spot, run!"
```

The expected output would be:

```
(run, run), 6
(run, see), 3
(run, spot), 6
(see, run), 3
(see, spot), 2
(spot, run), 6
(spot, see), 2
(spot, spot), 1
```

The omitted value from this matrix `(see, see)` would be 0.

The stripes approach is a very common pattern for collecting per-term vectors of n-dimensional arrays rather than outputting n by m cells in a pairwise fashion. This creates an output data structure with faster, ordered lookup, particularly when the vector is part of a chained computation, but the caveat is that vector must be able to be held in memory of a reducer.

The stripes pattern leverages the fact that each reducer is guaranteed to receive all values for a particular key. With this in mind, the mapper outputs an associative array of `(wj, count)` values for each word, `wi`.

```
def map(key, value) {
   for (i, word in enumerate(value)) {
       for (j, term in enumerate(value)) {
           output = {};
           if (i==j) { continue; }
           if (term in output) {
               output[term] += 1;
           } else {
               output[term] = 1
           }
       emit(word, tuple(output));
   }
}

def reduce(key, values) {
   output = {};
   for (value in values) {
       if (value[0] in output) {
           output[value[0]] += value[1];
       } else {
           output[value[0]] = value[1];
       }
   }
   emit(key, tuple(output));
}
```

For the same input, the output is now the more compact:

```
run,  ((run, 6), (see, 3), (spot, 6))
see,  ((run, 3), (spot, 2))
spot, ((run, 6), (see, 2), (spot, 1))
```

The pairs approach generates far more key/value pairs than the stripes approach. The stripes approach is not only more compact in its representation, but also generates fewer and simpler intermediary keys, thus optimizing the sort and shuffle. However, the deserialization and serialization of complex values will require more overhead, and Hadoop Streaming users need to pay special attention to how they are handling the stripes representation.

In the end, the stripes pattern tends to be more valuable for matrices whose square dimension can fit into memory, and is often produced in later stages of complex MapReduce jobs when the computation space has already been reduced. Stripes jobs also leverage filter mappers to omit values of no importance, like stop words in the case of word co-occurrence (he, she, it, those, with, etc.). The pairs pattern is used in the initial stages of those jobs to ensure that there are no memory out of bounds exceptions since it does not need to hold intermediary data in memory.

Some Common MapReduce Algorithms

Now that we've seen some of the building blocks for MapReduce jobs, let's explore some common applications and algorithms that use these blocks. As you've already noted, MapReduce jobs tend to be very small in terms of the number of lines of code and complexity, where identity functions are common to leverage the Hadoop architecture. Typically computations are composed of a number of utility jobs that run in sequence with various dependencies.

Because of the nature of MapReduce algorithms, it's useful to think of them as data flows rather than procedures or calculations. When performing analytics on the scale that Hadoop allows you to, this abstraction is extremely useful in plotting an end or application result from raw or rich data sets in the back end. Several of these common MapReduce algorithms therefore can be scheduled on your data set to improve flow and data awareness. MapReduce makes your data a living, breathing thing- rather than something that is simply "mined".

Sorting

Sorting is a relatively simple algorithm, given that MapReduce has an extremely efficient sort and shuffle built into the intermediary stage between mapping and reducing. Many distributed applications are routinely graded on

the "terasort" metric- the speed at which it can sort a terabyte of data. Hadoop holds the current record, able to sort 102.5 TB in 4,328 seconds[2].

Since reducers process keys in order and reducers are themselves ordered, the combination of a keyspace change plus the identity reducer is all that is required.

```
def map(lineno, value) {
   emit (value, _)
}

def reduce(value, _) {
   emit (value, _)
}
```

This algorithm assumes that the values you are trying to sort are one per line specified in the default `TextInput` format. The output will be sorted lines.

Searching

Consider a giant set of files containing lines of text along with some pattern that you would like to find in the text. MapReduce search algorithms leverage Job configurations that accept additional inputs on the command line- namely what you're trying to search for. A filter mapper is used to compare the line to the search pattern, and the identity reducer is used to output ordered results.

```
pattern = regex(input_pattern);

def map((filename, lineno), value) {
   if (pattern.search(value) == True) {
       emit (filename, _);
   }
}

def combine(filename, _) {
   emit (filename, _);
}

def reduce((filename, lineno), _) {
```

[2] http://sortbenchmark.org/

```
   emit (filename, _);
}
```

The combiner is used to ensure that only a single filename, if more than one lines are matched, are sent to the reducer, improving the efficiency of the search. Note also that with this pattern, the many small files problem can come into play. To deal with this, concatenate the many small files into one larger text file, and delineate them somehow. In the Reuters corpus, which you will see in the homework, a delimiter separates each document. In the Shakespeare corpus, lines are prepended by filename@lineno then tab delimited from the text.

Search algorithms tend to yield many results, so some ranking algorithm should be implemented alongside the simple pattern search. Ranking algorithms like PageRank from Google enjoy implementations as MapReduce algorithms; after all, it was this computation that inspired MapReduce in the first place!

Classification

Bayesian Classification is another typical algorithm performed with MapReduce. MapReduce algorithms not only train a Bayesian feature set, but can also be used to perform the classification on the test set. Given an input set of instances, the mapper will map the instance to the class, and once again the identity reducer is used to sort the classes.

```
def map(filename, instance) {
   emit(instance, class);
}
```

The mapper in this case will use some library like WEKA, Scikit, or e1071 R to build the classifier in the Map task memory and classify the instances.

TF-IDF

The final common algorithm we will discuss using MapReduce in this section is the Term Fequency-Inverse Document Frequency algorithm. This is another

canonical MapReduce example that is generally taught shortly after word count to demonstrate chained MapReduce jobs. This algorithm is used in natural language processing for larger algorithms like topic modeling and to determine the relevance of a term to a particular document. It is also extremely common in social network analysis, web analysis, and other analyses where there is unstructured or raw language data.

The algorithm can be expressed with the following equations computing term frequency, inverse document frequency, and finally the TF-IDF, which is the product of the first two equations:

$$tf_i = \frac{n_i}{\sum_k n_k}$$

$$idf_i = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$tfidf = tf \times idf$$

The term frequency is the relationship between the number of times a term, i appears in a document and the number of terms in the document. The inverse document frequency is the logarithmic relationship between the total number of documents, D and the number of documents the term appears in. To compute this, we require three map reduce jobs chained together:

```
// Word Frequency Per Document

def mapper(docid, document) {
   for (word in document) {
       emit ((word, docid), 1)
   }
}

def reducer((word, docid), counts) {
   emit((word, docid), sum(counts);
}
```

```
combiner = reducer




// Document Word Counts

def map((word, docid), tf) {
   emit(word, (docid, tf, 1));
}

def reduce(word, values) {
   terms = 0;
   for (docid, tf, num) in values {
       terms += num;
   }
   for (docid, tf, num) in values {
       emit((word, docid), (tf, terms));
   }
}

// Compute IDF

def map((word, docid), (tf, n)) {
   // Assume N is known or computed before hand
   // N is the number of documents.
   idf = Math.log(N/n);
   emit((word, docid), idf*tf);
}

// Identity Reducer
def reduce(key, values) {
   for value in values {
       emit(key, value);
   }
}
```

The first job computes the term frequency per document with a simple Word Count job that also maintains the document id for the term. A `SumReducer` and `SumCombiner` are used to optimize this job. The second job computes the total number of documents the word appears in. Finally, the last job computes the TF-IDF using the information populated through by the first two jobs.

Note, however, the two `for` loops in the second job's reducer. This reducer must buffer all `(doc, n, N)` counts while summing the value one into the

`terms` variable before emitting its computation. If there are many documents containing the word (think "the") this computation might not fit into memory. Possible solutions are to use a stop words filter to ignore high-frequency words, to use temporary disk storage for the intermediary data, or to write another in-between MapReduce job.

Although this job seems complex, envision the execution as a data flow, as pieces of the computation are produced, they are added to the flow of data. The key/value choices are motivated by the next step in the computation. And, crucially, the original input is only traversed a single time by this computation, allowing for linear dependency in the job. As you write more complex MapReduce jobs, keep these qualities in mind to solve larger problems.

Conclusion

Typical MapReduce algorithms are decomposed into constituent Map and Reduce functional chains that take advantage of the distributed computing infrastructure provided by the Hadoop architecture. Because of this, there are several key pieces of the architecture that we need to understand, including how to build jobs with Drivers, optimize Reducing and improve network traffic with Combiners, understand how data is partitioned to reducers, and finally how data is serialized between these various tasks and components.

Once these core pieces of the architecture are understood, we can leverage them to build chained, dependent Jobs that together make up a whole computation. Because Jobs are chained in this manner, there tend to be a few common patterns that we can take advantage of to compose the parts of the Job. We looked at a few of these patterns- including some interesting mapping and aggregation patterns, as well as the identity pattern, and other statistical patterns.

With these pieces we were able to explore more fully some common algorithms with MapReduce, including sorting, searching, and classification. As a final project we explored the computation of TF-IDF, an exemplary algorithm that utilizes many of the building blocks that we discussed in a chained job. This led us to the idea of composing our computations as data flows- stepwise computations that added to flow of data through each job as well as through each mapper and reducer.

Assignments

Please post your answers to the following discussion questions by Wednesday. See Assignment pages for more details.

Discussion

1. Consider a dataset of flights, where each record in the data set is the originating airport, the destination airport, the schedule arrival and departure times, and the actual arrival and departure times (in terms of wheels up and wheels down), along with distance in air miles. Discuss the data flow of designing a system that attempts to predict possible delays given this statistical history. What MapReduce jobs would be needed? Are there any concerns for individual tasks? What building blocks would be required? Are there any special implementations of partitioners or other internals that can be taken advantage of?

Exercises

There will be more details on the assignments, along with point breakdowns on the assignments page in the course materials.

1. Compute the Term Frequency-Inverse Document Frequency for words in the dataset of Reuter's newswire articles that is included in the course resources section.
2. Last week we computed the average flights per day and the mean arrival delay using the On Time Performance data set for January 2013. This week, let's go farther and compute all statistical metrics for each airport's arrival delay data set. Including sample size, total arrival delay, mean arrival delay, standard deviation, and the range.