COSC 6397
Big Data Analytics

Introduction to Map Reduce (I)

Edgar Gabriel
Spring 2015

Recommended Literature

• Original MapReduce paper by google
  http://research.google.com/archive/mapreduce-osdi04.pdf

• Fantastic resource for tutorials, examples etc:
  http://www.coreservlets.com/
Map Reduce Programming Model

- Input key/value pairs \(\rightarrow\) output a set of key/value pairs
- Map
  - Input pair \(\rightarrow\) intermediate key/value pair
  - \((k1, v1)\) \(\rightarrow\) list\((k2, v2)\)
- Reduce
  - One key all associated intermediate values
  - \((k2, \text{list}(v2))\) \(\rightarrow\) \text{list}(v3)

- Reduce stage starts when final map task is done

Map Reduce Programming Model (II)

- Mappers and Reducers are typically single threaded and deterministic
  - Determinism allows to restart failed tasks
  - Tasks can run on arbitrary number of nodes

- Mappers/Reducers run entirely independent of each other
  - In Hadoop, they run in separate JVMs
Map Reduce Frameworks

- **Hadoop:**
  - Widely used open source implementation
  - Java based
- **Google MapReduce framework**
  - Not publicly available
  - C++ based
- **MapReduce-MPI**
- **Disco**
  - Python based MapReduce framework
- ....

Map Reduce Framework

- Takes care of distributed processing and coordination
- Provides default implementations for certain operations of Map Reduce code, e.g.
  - Input splitting
  - Data transfer and sorting between map and reduce step
  - Writing output files
- Hadoop provides full functionality to manage compute resources
  - Scheduling and resource management
  - Parallel file system (HDFS)
Example: Word Count

- Count the number of occurrences of each word in a given text document

Mapper
- Input is text
- Tokenize the text
- Emit for each word a count of 1 - <word, 1>

Reducer
- Sum up counts for each word
- Write result

Configure the Job
- Specify Input, Output, Mapper, Reducer and Combiner

Run the job
Mapper

- **Mapper Class**
  - Class has 4 Java generics parameters
  - (1) input key (2) input value (3) output key (4) output value
- **map()** method injects Context object, used to write output
- A note to the code shown: removed a lot of Java stuff to focus on the essential aspects
  - No import of packages that are required to run the code
  - No exception handling
  - Parsing of command line arguments
- Code shown here available at: [http://www.cs.uh.edu/~gabriel/courses/cosc6397_s14/](http://www.cs.uh.edu/~gabriel/courses/cosc6397_s14/)

```java
public static class TokenizerMapper
extends 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, Context context){
// split the text into individual words
StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    // write each word with a counter of 1
    context.write(word, one);
}
}
```

Writables

- Types that can be serialized / deserialized to a stream
  - framework will serialize your data before writing it to disk
- Required to be input/output classes
- User can implement this interface, and use their own types for their input/output/intermediate values
- There are default for basic values, like
  - Strings: Text
  - Integers: IntWritable
  - Long: LongWritable
  - Float: FloatWritable

Reducer

- Similarly to Mapper - generic class with four types
  (1) input key (2) input value (3) output key (4) output value
  - The output types of map functions must match the input types of reduce function
  - In this case Text and IntWritable
- Map/Reduce framework groups key-value pairs produced by mapper by key
- For each key there is a set of one or more values

- Input into a reducer is sorted by key
  - Known as Shuffle and Sort
Reducer code in Hadoop

```java
public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

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

Assuming that count could be > 1 (e.g. if using a combiner)

Main code

```java
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    job.setInputFormatClass(TextInputFormat.class);
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Has to match the first two arguments of the map class

Has to match the last two arguments of the reduce class
Main code (II)

- **Job class**
  - Encapsulates information about a job
  - Controls execution of the job
- **A job is packaged within a jar file**
  - Hadoop Framework distributes the jar file on your behalf
  - Needs to know which jar file to distribute
  - Hadoop will locate the jar file that contains the provided class
- **Input data**
  - Can be a file or a directory
  - Directory is converted to a list of files as an input
  - Input is specified by implementation of `InputFormat`
  - `InputFormat` responsible for creating splits

Main code (III)

- **Define output path where reducer should place its output**
  - If path already exists then the job will fail
  - Writes key-value pair as plain text
  - Each reducer task writes to its own file
  - By default a job is configured to run with a single reducer.
    - Change with `job.setNumReduceTasks(10);`
- **Specify the output key and value types for both mapper and reducer functions**
  - Many times the same type
  - If types differ then use
    - `setMapOutputKeyClass(...);`
    - `setMapOutputValueClass(...);`
Main code (IV)

- `job.waitForCompletion(true)`
  - Submits and waits for completion
  - The boolean parameter flag specifies whether output should be written to console
  - If the job completes successfully ‘true’ is returned, otherwise ‘false’ is returned

Combiner

- Combine data per Mapper task to reduce amount of data transferred to reduce phase
- Reducer can very often serve as a combiner
  - Only works if reducer’s output key-value pair types are the same as mapper’s output types
- Combiners are not guaranteed to run
  - Optimization only
  - Not for critical logic
- Add to main file
  ```java
  job.setCombinerClass(IntSumReducer.class);
  ```
**Map Reduce Components**

- **User Components:**
  - **Mapper**
  - **Reducer**
  - **Combiner** (Optional)
  - **Writable(s)** (Optional)

- **System Components:**
  - **Input Splitter**: how to decompose input data to mappers
  - **Partitioner** (Shuffle): how to distribute data from mappers to reducers
  - **Output Committer**: how to write output data

Slide based on a lecture of Matei Zaharia: “Introduction to MapReduce and Hadoop”,
http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt
2nd example: K-means in MapReduce

- Mapper:
  - K= datapoint ID, V=datapoint coordinates
  - Calculate the distance for a datapoint to each centroid
  - Determine closest cluster
  - Write K’, V’ with K’ = cluster ID and V’=coordinates of the point

- Reducer:
  - Each reducer gets all entry associated with one cluster
  - Calculate sum of all point coordinates per cluster
  - Recalculate cluster mean

K-means in MapReduce

- One iteration of the k-means algorithm

- Multiple iterations within one job not supported by MapReduce
  - Hadoop has a way on how to specify a sequence of jobs with dependencies

- How to distribute Cluster centroids to all map tasks?
  - Hadoop distributed cache
K-means Implementation

- Simplified assumptions
  - 1 iteration
  - 2-D points, floating point coordinates
  - One data point per line of input file
  - 2 clusters
  - Initial cluster centroids provided by coordinates, one centroid per line
- Challenge:
  - Need a data structure to abstract a 2D point
  - Need to have access to cluster centroids on all mapper tasks

Creating your own Writable Datatype

- Writable is Hadoop’s serialization mechanism for reading and writing data to/from network or files
  - Optimized for network serialization
  - A set of basic types is provided
- Extends Writable interface
  - Framework’s serialization mechanisms
  - Defines how to read and write fields
  - org.apache.hadoop.io package

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
Creating your own Writable Datatype

- To define a writable type to be used as a Key
  - Keys must implement WritableComparable interface
  - Extends Writable and java.lang.Comparable<T>
  - Required because keys are sorted prior reduce phase

- Hadoop is shipped with many default implementations of WritableComparable<T>
  - Wrappers for primitives (String, Integer, etc...)
  - In fact, all primitive writables mentioned previously are WritableComparable

Implement your own Writable

- Implement 2 methods
  - write(DataOutput)
    - Serialize the content
  - readFields(DataInput)
    - Deserialize the content

- Additionally for WritableComparable
  - compareTo(T)
    - Has to return negative, zero, or positive number when comparing two elements to each other
public class TwoDPointWritable implements Writable {
    private FloatWritable x, y;
    public TwoDPointWritable() {
        this.x = new FloatWritable();
        this.y = new FloatWritable();
    }
    public void write(DataOutput out) {
        x.write(out);
        y.write(out);
    }
    public void readFields(DataInput input) {
        x.readFields(input);
        y.readFields(input);
    }
    Implement your own WritableComparable
    public void set(float a, float b) {
        this.x.set(a);
        this.y.set(b);
    }
    public float getx() {
        return x.get();
    }
    public float gety() {
        return y.get();
    }
}
InputFormat

- Specification for reading data
  - Creates Input Splits: Breaks up work into chunks
- Specifies how to read each split
  - For each Mapper instance a reader is retrieved by
    InputFormat.createRecordReader
- RecordReader generates key-value pairs
- map() method is called for each key-value pair

- If you want to use your own writable as Input to the mapper
  - Implement the createRecordReader interface
  - Return a RecordReader which consists of a key/value pair (i.e. LongWritable, TwoDPointWritable)

Predefine FileInputFormats

- Hadoop eco-system is packaged with many InputFormats
  - TextInputFormat
  - NLineInputFormat
  - DBInputFormat
  - TableInputFormat (HBASE)
  - StreamInputFormat
  - SequenceFileInputFormat

- Configure on a Job object
  - job.setInputFormatClass(XXXInputFormat.class);

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
public static class KmeansMapper
extends Mapper<LongWritable, TwoDPointWritable, IntWritable, TwoDPointWritable>
{
    public void map(LongWritable key, TwoDPointWritable value, Context context) {
        float distance=0.0f, mindistance=999999999.9f;
        int minindex=-1, i=0;
        for (i=0; i<2; i++) {
            float x = value.getx();
            float y = value.gety();
            distance = (x-centroids[i][0])*(x-centroids[i][0]) + (y-centroids[i][1])*(y-centroids[i][1]);
            if ( distance < mindistance ) {
                mindistance = distance;
                minindex=i;
            }
        }
        IntWritable closestCentroid = new IntWritable(minindex);
        context.write(closestCentroid, value);
    }
}

public static class KmeansReducer
extends Reducer<IntWritable, TwoDPointWritable, IntWritable, Text> {
    public void reduce(IntWritable clusterid, Iterable<TwoDPointWritable> points, Context context){
        int num = 0;
        float centerx=0.0f, centery=0.0f;
        for (TwoDPointWritable point : points) {
            num++;
            centerx += point.getx();
            centery += point.gety();
        }
        centerx = centerx/num;
        centery = centery/num;
        String preres = String.format("%f %f", centerx, centery);
        Text result = new Text(preres);
        context.write(clusterid, result);
    }
}
Output Value type

- In our Reducer: Text
- Could be a variant of TwoDPointWritable as well
  - Have to implement the class OutputFormat
  - Have to implement RecordWriter method
- Could we have used Text for input as well?
  - As long as split is done on a per line basis, yes
  - We still need the TwoDPointWritable abstraction for the intermediary output
    - Otherwise have constantly rewrite/parse floats to/from strings
      -> slow

Distributed Cache

- A mechanism to distribute files
  - Make them available to MapReduce task code
  - Has to be in HDFS - does not work for local file systems
- Hadoop yarn command provides several options to add distributed files
  - Can also use be configured in the main method directly
- Supports text files, jar files, etc...

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
Distributed Cache

- Simple life-cycle of Map and Reduce
  - Setup (Context) is called once
  - For each key/value pair in the split:
    map(Key, Value, Context)
  - Cleanup(Context) is called

- Distributed Cache operation are implemented in the setup method

---


---

e.g. for our kmeans Mapper

```java
public static class KmeansMapper
    extends Mapper<LongWritable, TwoDPPointWritable, IntWritable, TwoDPPointWritable>
{
    public final static String centerfile = "centers.txt";
    public float[][] centroids = new float[2][2];

    public void setup(Context context)
    {
        Scanner reader = new Scanner(new FileReader(centerfile));
        for (int i=0; i<2; i++)
        {
            int pos = reader.nextInt();
            centroids[pos][0] = reader.nextFloat();
            centroids[pos][1] = reader.nextFloat();
        }
    }

    public void map(LongWritable key, TwoDPPointWritable value, ...
```
Putting our main file together

```java
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

    Job job = new Job(conf, "kmeans");
    Path toCache = new Path("/centers/centers.txt");
    job.addCacheFile(toCache.toUri());
    job.createSymlink();

    job.setJarByClass(Kmeans.class);
    job.setMapperClass(KmeansMapper.class);
    job.setReducerClass(KmeansReducer.class);
    job.setInputFormatClass(TwoDPointFileInputFormat.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    job.setInputFormatClass(TwoDPointFileInputFormat.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));

    job.setMapOutputKeyClass(IntWritable.class);
    job.setMapOutputValueClass(TwoDPointWritable.class);
    FileOutputStream.setOutputStream(job, new Path(otherArgs[1]));
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(Text.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```
Hadoop MapReduce - additional features

- **Context**
  - provides possibility to store and retrieve counters
  - Could be used to store the iteration id for the k-means algorithm and read e.g. centers.txt.<iteration> id
- **Job dependencies using JobControl class**
  - Create simple workflows
  - Could be used to implement multiple iterations of the k-means algorithm
- **Streaming API:**
  - Support for non-java codes to be used as Mappers and Reducers

<table>
<thead>
<tr>
<th></th>
<th>Master/Worker</th>
<th>MPI</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Model &amp; Flexibility</strong></td>
<td>Limited Flexibility</td>
<td>Unlimited</td>
<td>Limited Flexibility</td>
</tr>
<tr>
<td></td>
<td>• Workers can not cooperate</td>
<td>• Generic data parallel model</td>
<td>• One map and one reduce step</td>
</tr>
<tr>
<td></td>
<td>• Best suited for bag of task applications</td>
<td></td>
<td>• Extensions for multi-job execution available</td>
</tr>
<tr>
<td><strong>Data Formats</strong></td>
<td>Unlimited</td>
<td>Limited:</td>
<td>Nearly unlimited</td>
</tr>
<tr>
<td></td>
<td>• Up to the code developer writing the master</td>
<td>• Flat data files only</td>
<td>• If you can express it as Writable and Split</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• E.g. no support for databases</td>
<td>• Support for DBs available</td>
</tr>
</tbody>
</table>

Comparison of Programming Models
## Comparison of Programming Models

<table>
<thead>
<tr>
<th></th>
<th>Master/Worker</th>
<th>MPI</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fault Tolerance</strong></td>
<td>Good</td>
<td>Bad</td>
<td>Very good</td>
</tr>
<tr>
<td></td>
<td>• User can easily add code to achieve reliability</td>
<td>• Not supported by current standard</td>
<td>• No user code required to deal with failures</td>
</tr>
<tr>
<td><strong>User support from programming environment</strong></td>
<td>• Tools such as CONDOR available for generating input tasks</td>
<td>Limited:</td>
<td>Good:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Support for data types and I/O</td>
<td>• Shuffle/sort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Large eco-infrastructure of libraries and tools available</td>
<td>• I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Load balancing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Increasing number of libraries and tools available</td>
</tr>
</tbody>
</table>