Frequently asked questions from the previous class survey

- Computing Economics
  - Is 10,000 instructions per-byte still the limit now?
- MapReduce
  - Variants provide less or more constraints?
  - Can we breach the synchronization barrier somehow?
  - Difference between HDFS and GFS?
  - Difference between Hadoop and Storm?
  - Shuffle phase? And OutOfHeapSpace errors?
  - Optimal ratio of Mappers to Reducers?
  - What is Spark?

Topics covered in this lecture

- MapReduce
- Hadoop
  - Phases of Map
  - Phases of Reducers
  - Examples

Implementation of the Runtime

Execution Overview

Execution Overview: Step III
Workers that are assigned a map task

- Read contents of their input split
- Parses <key, value> pairs out of input data
- Pass each pair to user-defined Map function
- Intermediate <key, value> pairs from Maps
  - Buffered in Memory
Execution Overview: Step IV
Writing to disk
- Periodically, buffered pairs are written to disk
- These writes are partitioned
  - By the partitioning function
- Locations of buffered pairs on local disk
  - Reported to back to Master
  - Master forwards these locations to reduce workers

Execution Overview: Step V
Reading Intermediate data
- Master notifies Reduce worker about locations
- Reduce worker reads buffered data from the local disks of Maps
- Read all intermediate data; sort by intermediate key
  - All occurrences of same key grouped together
  - Many different keys map to the same Reduce task

Execution Overview: Step VI
Processing data at the Reduce worker
- Iterate over sorted intermediate data
- For each unique key pass
  - Key + set of intermediate values to Reduce function
- Output of Reduce function is appended
  - To output file of reduce partition

Execution Overview: Step VII
Waking up the user
- After all Map & Reduce tasks have been completed
- Control returns to the user code

Task Granularity
- Subdivide map phase into M pieces
- Subdivide reduce phase into R pieces
- M, R >> number of worker machines
- Each worker performing many different tasks
  - Improves dynamic load balancing
  - Speeds up recovery during failures
Master Data Structures

- For each Map and Reduce task
  - **State**: (idle, in-progress, completed)
  - Worker machine identity
- For each completed Map task store
  - **Location** and sizes of R intermediate file regions
- Information pushed incrementally to in-progress Reduce tasks

Practical bounds on how large M and R can be

- Master must make $O(M + R)$ scheduling decisions
- Keep $O(MR)$ state in memory

Hadoop

- Java-based open-source implementation of MapReduce
- Created by Doug Cutting
- Origins of the name Hadoop
  - Stuffed yellow elephant
- Includes HDFS [Hadoop Distributed File System]

Hadoop: MapReduce Dataflow

In Hadoop a Map task has 4 phases

- Record reader
- Mapper
- Combiner
- Partitioner

October 1, 2015
Map task phases: **Record Reader**

- Translates input splits into records
- Parses data into records, but does not parse the record itself
- Passes the data to the mapper in the form of a key/value pair
  - key in this context is positional information
  - value is the chunk of data that composes a record

Map task phases: **Map**

- User-provided code is executed on each key/value pair from the record reader
- This user-code produces zero or more new key/value pairs, called the intermediate pairs
  - key is what the data will be grouped on and value is the information pertinent to the analysis in the reducer
  - Choice of key/value pairs is critical and not arbitrary

Map task phases: **Combiner**

- Can group data in the map phase
- Takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper
- Significantly reduces the amount of data that has to move over the network.
  - Sending ("hello", 3) requires fewer bytes than sending ("hello", 1) three times over the network

Combiner function

- No guarantees on how many times Hadoop will call this on a map output record
  - The combiner should, however, result in the same output from the reducer
- Combiners must be commutative and associative
  - Sometimes they are also called distributive
  - Commutative: Order of operands \((5+2) + 1 = 5 + (2+1)\)
  - Associative: Order of operators \(5 \times (5 \times 3) = (5 \times 5) \times 3\)
  - Non-associative and non-commutative: Vector cross products and matrix multiplication \((AB) \neq (BA)\) respectively

Map task phases: **Partitioner**

- Takes the intermediate key/value pairs from the mapper (or combiner) and splits them up into shards, one shard per reducer
- Default: key.hashCode() % (number of reducers)
  - Randomly distributes the keyspace evenly over the reducers
  - But still ensures that keys with the same value in different mappers end up at the same reducer

Map task phases: **Partitioner**

- Partitioner can be customized (e.g. for sorting)
  - Changing the partitioner is rarely necessary
- The partitioned data is written to the local file system for each map and waits to be pulled by its respective reducer
In Hadoop a Reduce task has 4 phases

- Shuffle
- Sort
- Reducer
- Output format

Reduce task phases: Shuffle and sort

- Shuffle
  - Takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running
- Sort
  - Individual data pieces are then **sorted by key** into one larger data list
  - Groups equivalent keys together so that their values can be iterated over easily in the reduce task.

Reduce task phases: Reducer

- This phase is **not customizable** and the framework handles everything automatically
- The only control a developer has is how the keys are sorted and grouped by specifying a custom Comparator object

Reduce task phases: Output format

- Translates the final key/value pair from the reduce function and writes it out to a file using a record writer
- By default:
  - Separate the key and value with a tab
  - Separates records with a newline character
- Can typically be customized to provide richer output formats
  - But in the end, the data is written out to HDFS, regardless of format

**Reduce task phases:**

- **Shuffle and sort**
- **Reducer**
- **Output format**

**MapReduce example**
### Word Count

- Word count over user-submitted comments on StackOverflow
- Content of the `Text` field will be retrieved and preprocessed
- Then count how many times we see each word
- Example record from this data set is:

  ```
  <row Id = "8189677" PostId = "6881722" Text = "Have you looked at Hadoop?" CreationDate = "2011-07-30T07:34:33.343" UserId = "831878" />
  ```
  - This record is the 8,189,677th comment on Stack Overflow, and is associated with post number 6,881,722, and is by user number 831,878.

### Driver class for the example

```java
public class CommentWordCount {
    public static void main( String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job( conf, "StackOverflow Comment Word Count");
        job.set JarByClass( CommentWordCount.class);
        job.setMapperClass( WordCountMapper.class);
        job.setCombinerClass( IntSumReducer.class);
        job.setOutputKeyClass( Text.class);
        job.setOutputValueClass( IntWritable.class);
        FileInputFormat.addInputPath( job, new Path( args[0]));
        System.exit( job.waitForCompletion( true) ? 0 : 1);
    }
}
```

### Waiting for the job to complete

- The `waitForCompletion()` method on Job submits the job and waits for it to finish
- The single parameter is a flag indicating whether verbose output is generated. When `true` the job writes information about its progress to the console
- The return value of the `waitForCompletion()` method is a Boolean indicating success (`true`) or failure (`false`), which we translate into the program's exit code of 0 or 1

### The Mapper class

```java
public class WordCountMapper
    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 {
        Map <String,String> parsed = MRDPUtils.transformXmlToMap(value.toString());
        String txt = parsed.get("Text");
        StringTokenizer itr = new StringTokenizer( txt);
        while (itr.hasMoreTokens()) {
            word.set( itr.nextToken());
            context.write( word, one);
        }
    }
}
```

### Some details about the Mapper class

- Notice the type of the parent class: `Mapper <Object, Text, Text, IntWritable>`
- Maps to the types of the input key, input value, output key, and output value, respectively.
- The key of the input in this case is not useful, so we use `Object`
- Data coming in is `Text` (Hadoop’s special `String` type) because we are reading the data as a line-by-line text document
- Our output key and value are `Text` and `IntWritable` because we will be using the word as the key and the count as the value.

### The word count Reducer

```java
public 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);
    }
}
```

---

**CS555: Distributed Systems [Fall 2015]**
**Dept. Of Computer Science, Colorado State University**
Reducer class

- As in the mapper, we specify the input and output types via the template parent class
- Types correspond to the same things: input key, input value, output key, and output value
- The input key and input value data types must match the output key/value types from the mapper
- The output key and output value data types must match the types that the job’s FileOutputFormat is expecting

The reduce function has a different signature from map

- Gives you an Iterator over values instead of just a single value
- We iterate over all values that have that key, instead of just one at a time
- key is very important in the reducer of pretty much every MapReduce job
- Unlike the input key in the map.

More about reducer outputs

- Anything passed to context.write will get written out to a file
- Each reducer will create one file

Another example (Averages)

- Given a list of user’s comment determine the average comment length per-hour
- To calculate average we need two things:
  - Sum values that we want to average
  - Number of values that went into the sum
- Reducer can do this very easily by iterating through each value in the set and adding to a running sum while keeping count
- But if you do this you cannot use the reducer as your combiner!
  - Calculating an average is not an associative operation
  - You cannot change the order of the operators
  - mean(0, 20, 10, 25, 15) = 14 BUT ...
  - mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15
Approach to ensuring code reuse at the combiner

- Mapper will output two columns of data
  - Count and average
- Reducer will multiply "count" field by the "average" field to add to a running count and add "count" to the running count
- Then divide the running sum with running count
- Output the count with the calculated average

Mapper code

```java
public static class AverageMapper extends Mapper<Object, Text, IntWritable, CountAverageTuple> {
  private CountAverageTuple outCountAverage = new CountAverageTuple();
  public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
    String strDate = parsed.get("CreationDate");
    String text = parsed.get("Text");
    Date creationDate = frmt.parse(strDate);
    outHour.set(creationDate.getHours());
    outCountAverage.setCount(1);
    outCountAverage.setAverage(text.length());
    context.write(outHour, outCountAverage);
  }
}
```

Reducer code

```java
public class AverageReducer extends Reducer<IntWritable, CountAverageTuple, IntWritable, CountAverageTuple> {
  private CountAverageTuple result = new CountAverageTuple();
  public void reduce(IntWritable key, Iterable<CountAverageTuple> values, Context context) throws IOException, InterruptedException {
    float sum = 0; float count = 0;
    // Iterate through all input values for this key
    for (CountAverageTuple val : values) {
      sum += val.getCount() * val.getAverage();
      count += val.getCount();
    }
    result.setCount(count);
    result.setAverage(sum / count);
    context.write(key, result);
  }
}
```

Data flow for the average example

<table>
<thead>
<tr>
<th>Hour</th>
<th>Count</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Hour Count Average

<table>
<thead>
<tr>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Group 1

Group 2

Setting:
- Combiner executes over Groups 1 and 2
- DOES NOT execute on the last two rows

Stragglers

- Machine that takes an unusually long time to complete a map or reduce operation
- Can slow down entire computation
How stragglers arise

- Machine with a bad disk
  - Frequent, correctable errors
  - Read performance drops from 30 MB/s to 1 MB/s
- Over scheduling
  - Many tasks executing on the same machine
  - Competition for CPU, memory, disk or network cycles
- Bug in machine initialization code
  - Processor caches may be disabled

Alleviating the problem of stragglers

- When a MapReduce operation is close to completion
- Schedule backup executions of remaining in-progress tasks
- Task completed when
  - Primary or back finishes execution
- Significantly reduces time to complete large MapReduce operations

Skipping Bad Records

- Bugs in user code cause Map or Reduce functions to crash
  - Deterministically: On certain records
- Fix the bug?
  - Yes, but not always feasible
- Acceptable to ignore a few records

Skipping bad records

- Optional mode of operation
  1. Detect records that cause deterministic crashes
  2. Skip them
- Each worker installs signal handler to catch segmentation violations and bus errors

The contents of this slide-set are based on the following references

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004: 137-150