CS 378 – Big Data Programming

Lecture 4

Summarization Patterns
Review

• Assignment 1 – Questions?
  – Using maven
  – Using AWS
Simple Debugging

• Counters
  – controller
  – syslog

• Custom counters
  – `context.getCounter(group, counter).increment(1L);`
  – `group` and `counter` are strings
Summarization

• Counting things is a common map-reduce task
  – Word count was a simple example
  – Min, max, mean, median, variance, ...

• By making the “things” being counted keys, MapReduce is doing much of the work for us
  – Hadoop sorts and groups data by key

• In WordCount, the words counted are the keys
Summarization

• Simple and useful pattern
• Mappers do local counts, reducers sum up
• Combiners are very useful here
• Usually collecting multiple statistics
Assignment 2 – Word Statistics

• Input:
  – Each input record/value is a paragraph of document

• Output (similar to word count, but more numbers):
  – For each word in the document, output:
    – Number of paragraphs containing the word
    – Mean
      • In paragraphs where the word appears, what is the average number of times it appears
    – Variance
      • In paragraphs where the word appears, what is the variance
Word Statistics

• What do we need to calculate mean, variance?

• Mean is straightforward
  – Total number of occurrences of the word
  – Number of paragraphs containing the word

• Variance is less obvious
  – We can get there with a little algebra
  – “Mean of square minus square of mean”
Multiple Output Values

• If we are to output multiple values for each key
  – How do we do that?
  – WordCount output a single number as the value

• Remember, our object containing the values needs to implement the **Writable** interface

• We could use **Text**
  – Value is a string of comma separated values
  – Have to convert our counts to strings, build the full string
  – Have to parse the string on input (not hard)
Multiple Output Values

• Suppose we wanted to implement a custom class
• Call it: `LongArrayWritable`
  – How would we implement this class?
  – Needs to implement the `Writable` interface
  – `write()` method:
    • Output the length of the array
    • Output that many long values
  – `readFields()` method:
    • Read the length of the array
    • Read that many long values
Multiple Output Values

- Our `LongArrayWritable` class could use some other methods and instance data
  - An instance variable to hold the values.
    - What would its type be?
  - A method to set the values (an array)
  - A method to get the values (an array)

- A method to sum to instances?
  - What would the signature be?
Multiple Output Values

• Hadoop provides a class to facilitate this:
  • ArrayWritable

• In addition to write() and readFields():
  – Writable[] get()
  – Class getValueClass()
  – void setWritable(Writable[] values)
  – Object toArray()
  – String[] toString()
Multiple Output Values

- If our `ArrayWritable` object is input to a reducer, we need to tell Hadoop how to set the value to the proper type.

- To do this, we’ll extend this class to `LongArrayWritable`:

  ```java
  public class LongArrayWritable extends ArrayWritable {
      public LongArrayWritable() {
          super(LongWritable.class);
      }
  }
  ```
Multiple Output Values

• We can add methods to LongArrayWritable class to make it easier to use.

```java
public long[] getValueArray() {
    Writable[] wValues = get();
    long[] values = new long[wValues.length];
    for (int i = 0; i < values.length; i++) {
        values[i] = ((LongWritable)wValues[i]).get();
    }
    return values;
}
```
Word Statistics

• Mapper will output what values?

• Reducer will calculate non-integer values
  – Mean, variance

• So we’ll need to handle float/double values
  – Do we need to create DoubleArrayWritable for reduce output?
Word Statistics

• Combiner will be useful for computing word statistics

• Can we reuse the reducer class for the combiner?
  – What are the combiner inputs and outputs?
The number of reduce tasks is not governed by the size of the input, but instead is specified independently. In "The Default MapReduce Job" on page 227, you will see how to choose the number of reduce tasks for a given job.

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time, as you will see in "Shuffle and Sort" on page 208.

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel (a few examples are discussed in "NLineInputFormat" on page 247). In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).

Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner...