CS 378 – Big Data Programming

Lecture 3
Anatomy of a Hadoop Map-Reduce Program
Assignment 1 Update

• JAR file build issues?
• What’s in pom.xml

• Running the example on AWS
  – The cluster and job monitor page
  – Log files: controller, syslog

• Questions?
Map-Reduce Code

• main() method

• Job object - Collects up all the specs for the job
  – Where is the JAR file to distribute?
  – Type of the output pair
  – Mapper and Reducer classes
  – Input and output file formats
  – Input file(s), output directory

• Configuration object – forwarded to map(), reduce()
  – Job level parameters communicated via this object
Map-Reduce Code

• MapClass
  – Extends Mapper, declaring the input and output pair types for the map() method

• map() method
  – Arguments: input pair, and the Context
  – Output done via the context object
Map-Reduce Code

• ReduceClass
  – Extends Reducer, declaring the input and output pair types for the reduce() method

• reduce() method
  – Arguments: input pair, and the Context
  – Output done via the context object
Map-Reduce Code

• map() and reduce() input pair and output pair types
• Derived from Writable
  – readFields(DataInput in)
  – write(DataOutput out)

• Text, IntWritable, LongWritable all implement Writable
  – As do many other types, some of which we will use

• Possible to design your own class that implements Writable
Map-Reduce Code

• Combiner – combines multiple outputs from a Mapper before shuffle

• Input and output pair types must be the same.
  – Why?

• When can a combiner be used?
  – Map output can be processed (“combined”) even through we do not see all values associated with the key
  – Combiner output can be interpreted by reducer
  – Word count, and many other counting applications can use a combiner.
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).
Map-Reduce Code

• For WordCount, suppose we used a hash table to collect word counts over multiple input records.

• Why wouldn’t this work?