Hadoop and assignment 3

Wikipedia clustering

Multiple tasks (jobs) w/ Hadoop

1. Documents → Word count
   – (word, count) list

2. Sort words by count
   – Dictionary; (index, word)

3. Compute sparse matrix representation
   – Break the matrix into partitions

MapReduce Programming Model

- Data type: key-value *records*

- Map function:

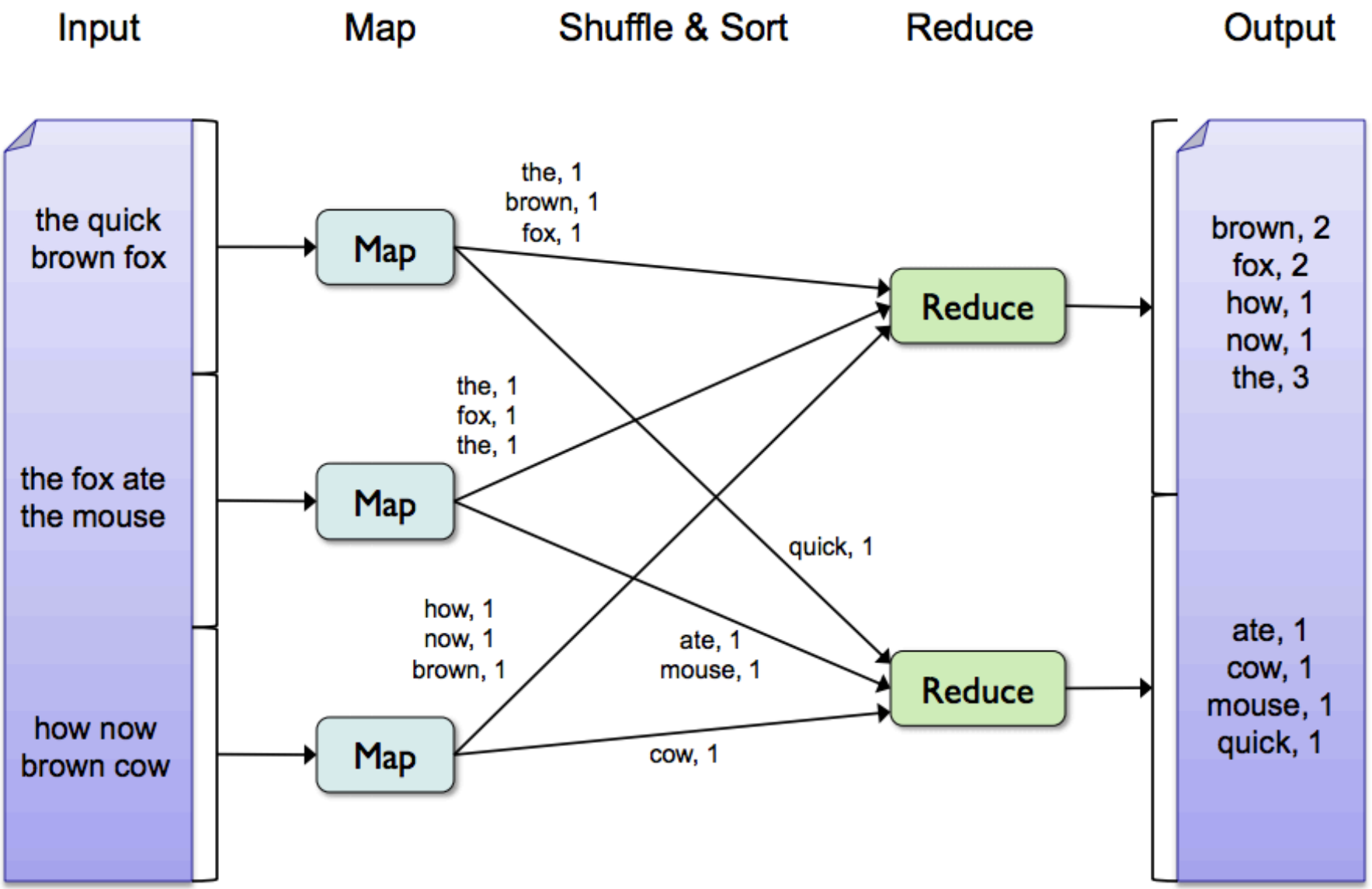
$$(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})$$

- Reduce function:

$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$

WORD COUNT AND COMBINER

Word Count Execution



Word Count in Java

```
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable ONE = new IntWritable(1);

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        output.collect(new text(itr.nextToken()), ONE);
      }
    }
 }
```

Word Count in Java

```
public static class Reduce extends MapReduceBase
   implements Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
     int sum = 0;
     while (values.hasNext()) {
       sum += values.next().get();
     }
     output.collect(key, new IntWritable(sum));
   }
 }
```

Job

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;

public class WordCount{
        public static void main(String[] args) throws IOException {
                if (args.length != 2) {
                        System.err.println("Usage: WordCount <input path> <output path>");
                        System.exit(-1);
                }
                JobConf conf = new JobConf(WordCount.class);
                conf.setJobName(”Word Count");
                FileInputFormat.addInputPath(conf, new Path(args[0]));
                FileOutputFormat.setOutputPath(conf, new Path(args[1]));
                conf.setMapperClass(MapClass.class);
                conf.setReducerClass(Reduce.class);
                conf.setOutputKeyClass(Text.class);
                conf.setOutputValueClass(IntWritable.class);
                JobClient.runJob(conf);
        }
}
```

To run the job

> export HADOOP_CLASSPATH=_some_path_

> hadoop WordCount sample.txt output

Instructions to run on iclusters

- http://inst.eecs.berkeley.edu/cgi-bin/pub.cgi?file=hadoop.help

Job status web page

- http://icluster1.eecs.berkeley.edu:50030/jobtracker.jsp

Combiner

- Stop words often have order of magnitude more appearance than others
  - Require high bandwidth from mapper to reducer
  - Hot spot in reducers

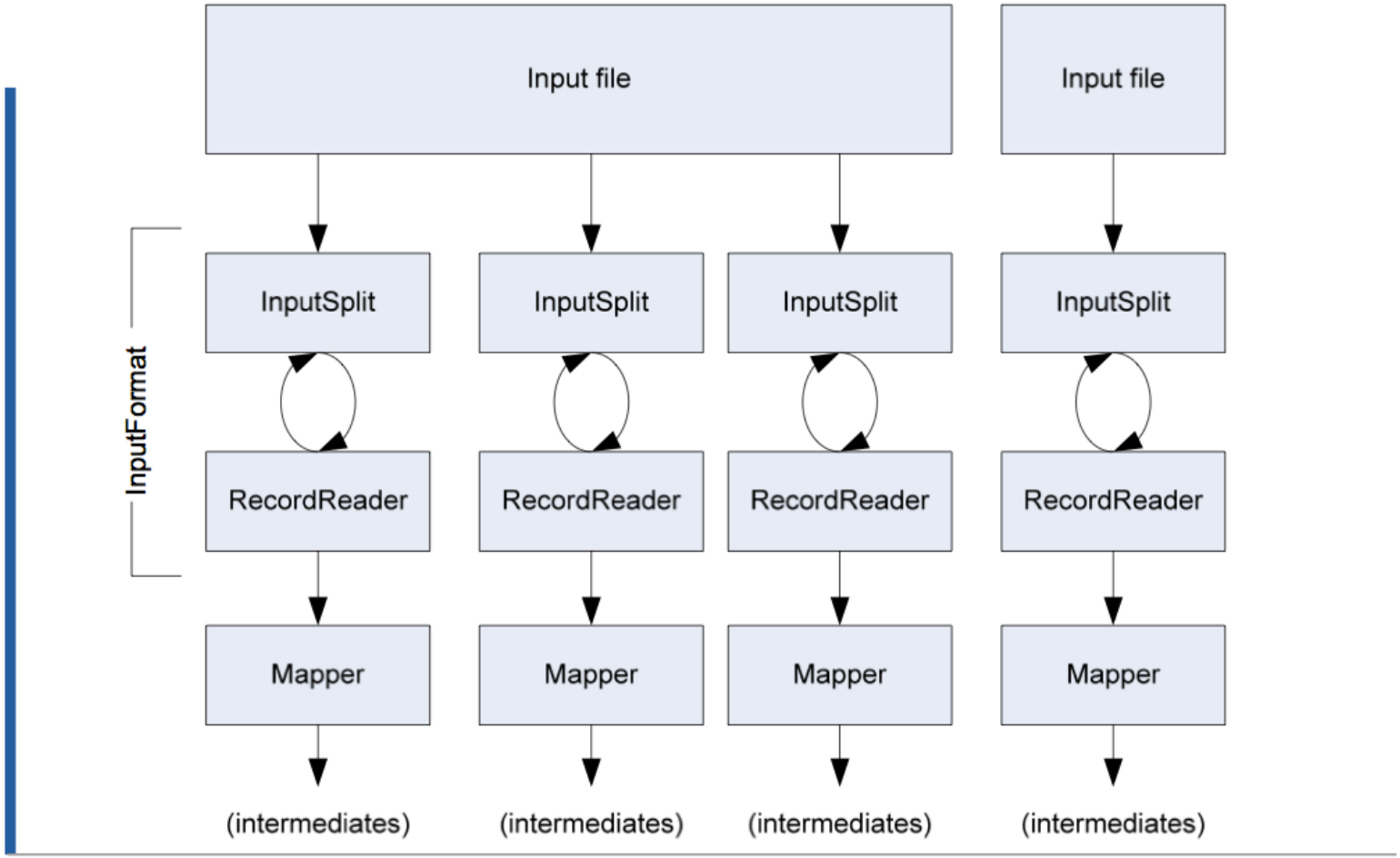
- So, combine multiple (key, value)'s of the same key before sending out from mapper

Use Reducer for Combiner

```
public class WordCount{
        public static void main(String[] args) throws IOException {
                if (args.length != 2) {
                        System.err.println("Usage: WordCount <input path> <output path>");
                        System.exit(-1);
                }
                JobConf conf = new JobConf(WordCount.class);
                conf.setJobName(”Word Count");
                FileInputFormat.addInputPath(conf, new Path(args[0]));
                FileOutputFormat.setOutputPath(conf, new Path(args[1]));
                conf.setMapperClass(MapClass.class);
                conf.setCombinerClass(Reduce.class);
                conf.setReducerClass(Reduce.class);
                conf.setOutputKeyClass(Text.class);
                conf.setOutputValueClass(IntWritable.class);
                JobClient.runJob(conf);
        }
}
```

Note: does not work for all problems

Getting Data To The Mapper



Input Split Size

- *FileInputFormat* will divide large files into chunks
  - Exact size controlled by mapred.min.split.size
- RecordReaders receive file, offset, and length of chunk
- Custom *InputFormat* implementations may override split size – e.g., "NeverChunkFile"

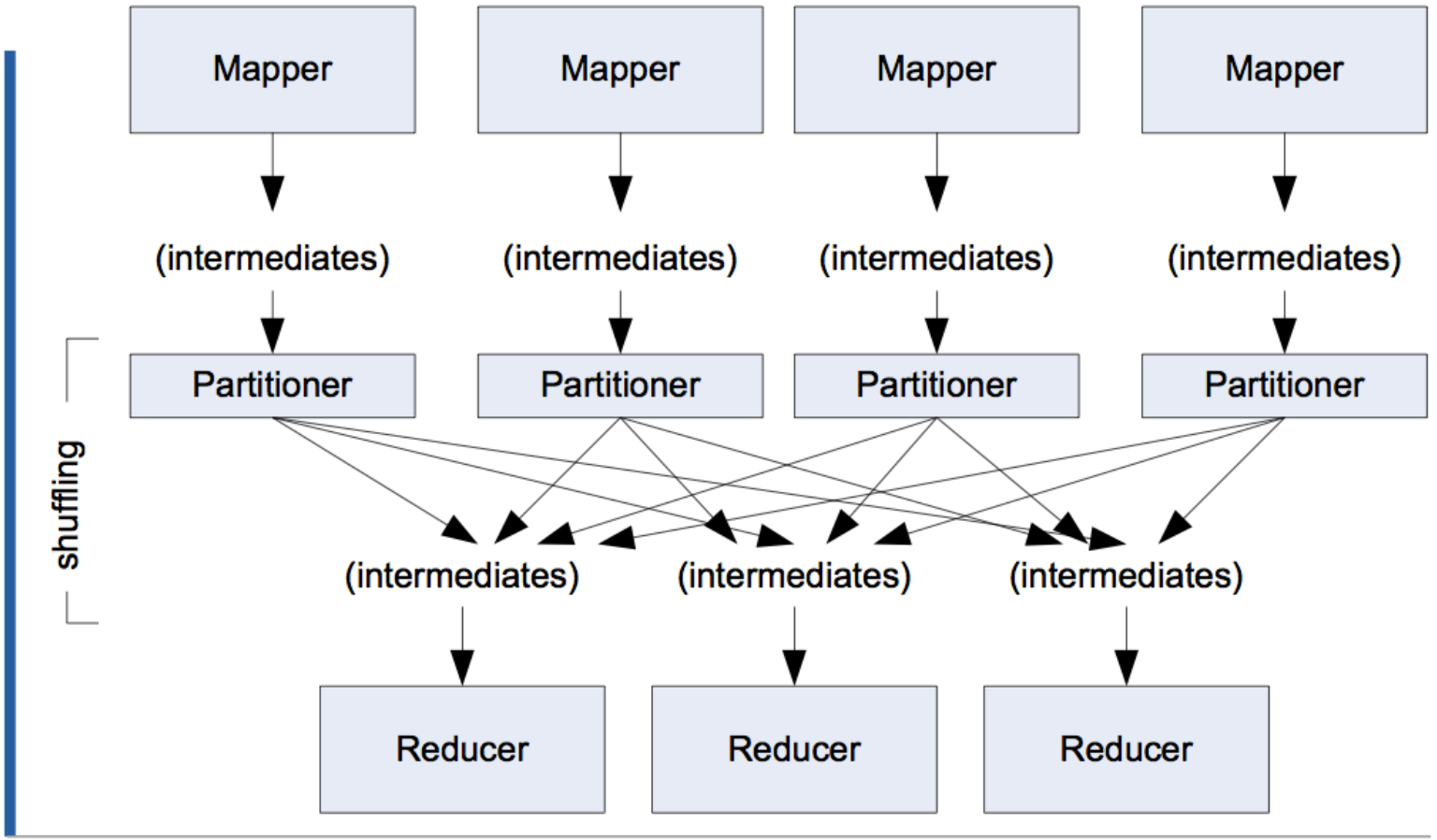
FileInputFormat and Friends

- *TextInputFormat* – Treats each '\n'-terminated line of a file as a value
- *KeyValueTextInputFormat* – Maps '\n'-terminated text lines of "k SEP v"
- *SequenceFileInputFormat* – Binary file of (k, v) pairs with some add'l metadata
- *SequenceFileAsTextInputFormat* – Same, but maps (k.toString(), v.toString())

Sending Data To Reducers

- Map function receives *OutputCollector* object
  - OutputCollector.collect() takes (k, v) elements
- Any *(WritableComparable, Writable)* can be used

Partition And Shuffle



Partitioner

- int getPartition(key, val, numPartitions)
  - Outputs the partition number for a given key
  - One partition == values sent to one Reduce task
- *HashPartitioner* used by default
  - Uses key.hashCode() to return partition num
- *JobConf* sets *Partitioner* implementation

SORT WORD BY COUNT

Sort (create dictionary)

- Input: (word, count) pairs
- Output: words sorted by count, marked with index

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered
- Moreover, we only need one output file (one reducer)

- Mapper: with count as key
  - (word, count)→(count, word)
- Reducer: output word with index
  - **// words with the same count will be sent to the same reducer**
  - **// words will appear in the order of count magnitude**
  - (count, word's)→(index, word)

COMPUTE SPARSE MATRIX

Compute sparse matrix

- Input: documents
  - dictionary loaded in memory
- Output: spare vectors
  - Count word frequency per document

- Mapper
  - convert word to index
  - Output (document id, (word index, 1))
- Reducer
  - **//values of the same document will be sent to the same reducer**
  - Count the frequency of all the words in a document
  - Output (document id, (word index, frequency in the doc))

CLUSTERING WITH MAP-REDUCE?

Some idea

- Given K centroids μ_k, N documents i/vectors f

- Mapper: calculate distance to all centroids
  - <i, f> → <i, d_k>
- Reducer: pick the best $\underline{k}$ for document I
  - **// distances of the same document will be sent to the same reducer**
  - <i, d_k's> → $\underline{k}$ → <i, $\underline{k}$>

- Mapper: identity with $\underline{k}$ as index
  - <f, $\underline{k}$> → <$\underline{k}$, f>
- Reducer: calculate new centroids
  - **// features of the same cluster will be sent to the same reducer**
  - <$\underline{k}$, f's> → <k, $\mu_{\underline{k}}$>

Scala on Hadoop

```scala
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.Path
import org.apache.hadoop.io.{IntWritable,Text}
import org.apache.hadoop.mapreduce.{Job,Mapper,Reducer}
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
import org.apache.hadoop.util.GenericOptionsParser
import scala.collection.JavaConversions._

class TokenizerMapper extends Mapper[Object, Text, Text, IntWritable] {

  val one = new IntWritable(1)
  var word = new Text

  override
  def map (key: Object, value: Text, context: Mapper[Object,Text,Text,IntWritable]#Context) {
    value.toString.split("\\s").foreach { token => word.set(token); context.write(word, one) }
  }
}

class IntSumReducer extends Reducer[Text,IntWritable,Text,IntWritable] {

  val result = new IntWritable()

  override
  def reduce (key: Text, values: java.lang.Iterable[IntWritable],
              context: Reducer[Text,IntWritable,Text,IntWritable]#Context) {
    result set(values.foldLeft(0) { _ + _.get })
    context write(key, result)
  }
}

object WordCountScala {

  def main (args: Array[String]) {
    val conf = new Configuration()
    val job = new Job(conf, "word count")
    job.setJarByClass(classOf[TokenizerMapper])
    job.setMapperClass(classOf[TokenizerMapper])
    job.setCombinerClass(classOf[IntSumReducer])
    job.setReducerClass(classOf[IntSumReducer])
    job.setOutputKeyClass(classOf[Text])
    job.setOutputValueClass(classOf[IntWritable])
    FileInputFormat.addInputPath(job, new Path(args(0)))
    FileOutputFormat.setOutputPath(job, new Path(args(1)))
    System.exit(if(job.waitForCompletion(true)) 0 else 1)
  }

}
```

References

- Cloudera- Programming with Hadoop
  - http://www.cloudera.com/resource/programming_with_hadoop/
- Cloudera – MapReduce Algorithms
  - http://www.cloudera.com/resource/mapreduce_algorithms/
- “Hadoop: The Definitive Guide”, Tom White, O’Reilly 2010