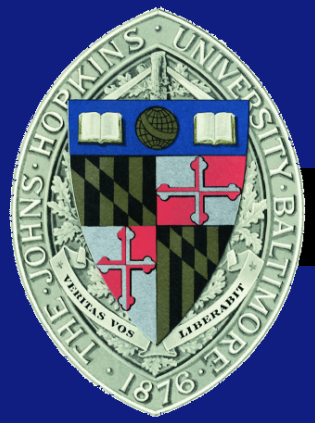
Lecture 9
Hadoop!

EN 600.320/420

Instructor: Randal Burns

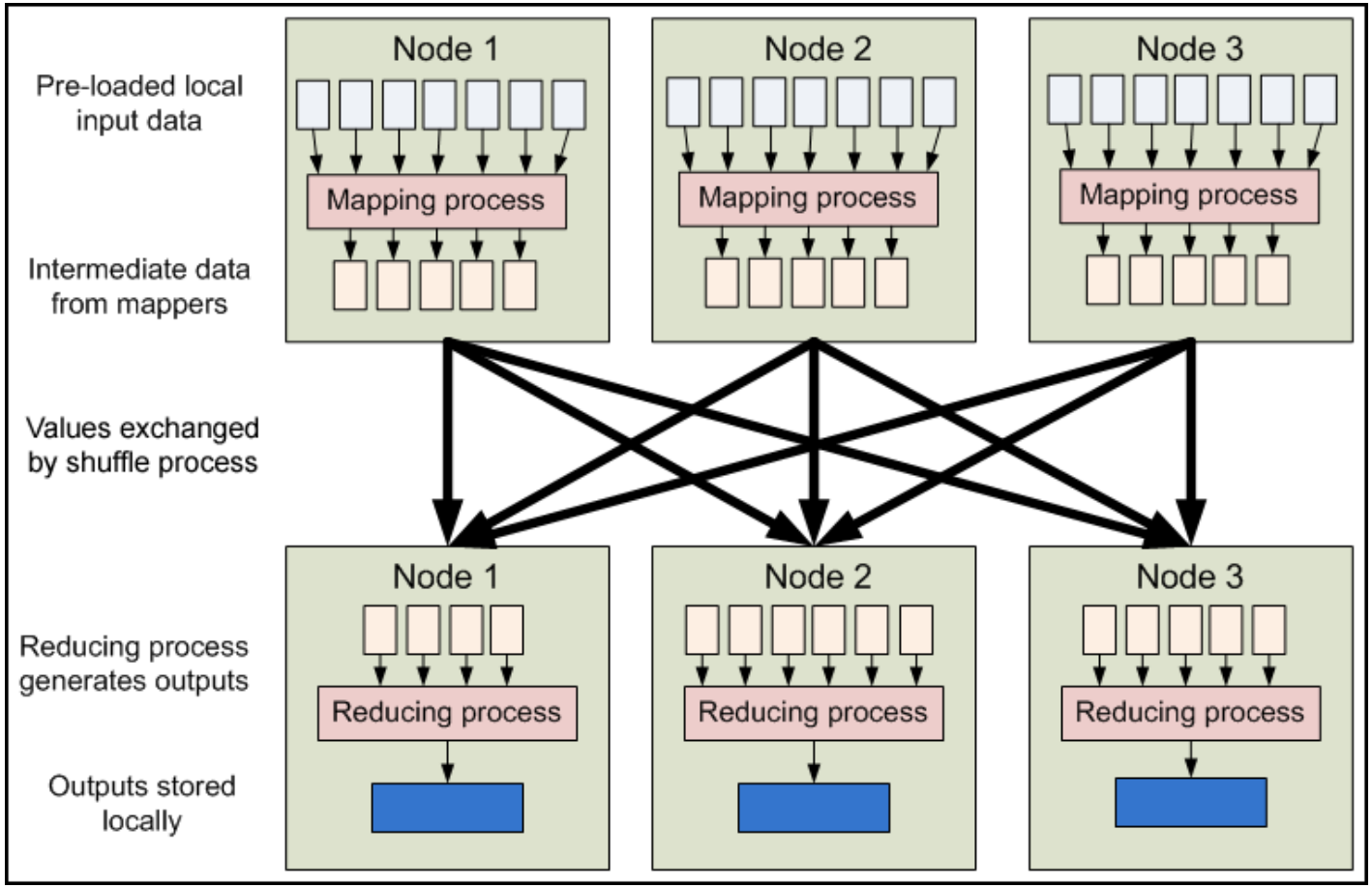
10 March 2014



Department of Computer Science, *Johns Hopkins University*

Hadoop!

- Open source reimplementation of Map/Reduce



New Tutorial

- At http://developer.yahoo.com/hadoop/tutorial/module3.html
- Good news
  - Eclipse support
  - Better examples
- Bad news
  - Long
- You all should do the tutorial (Module III)
  - We follow that module here

Define map() and reduce()

- Java package org.apache.hadoop.io.mapred:
  - MapReduceBase (is a M/R program)
  - Mapper (interface for mapper function)
  - Reducer (interface for reducer function)

- Paradigm
  - Extend MapReduceBase (to be Hadoop! executable)
  - Implement the interfaces

Mapper Code

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class WordCountMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

  private final IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(WritableComparable key, Writable value,
      OutputCollector output, Reporter reporter) throws IOException {

    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line.toLowerCase());
    while(itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
```

Observations

- Types:
  - Hadoop! wraps all types that are to be input/output
  - Must use IntWritable() for output, not int
  - Text is a Hadoop! class for strings

- Collector paradigm for I/O:
  - Output of the map, input to shuffle
  - Output of the reducer, into partitions

Reducer Code

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class WordCountReducer extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterator values,
      OutputCollector output, Reporter reporter) throws IOException {

    int sum = 0;
    while (values.hasNext()) {
      IntWritable value = (IntWritable) values.next();
      sum += value.get(); // process value
    }

    output.collect(key, new IntWritable(sum));
  }
}
```

Hadoop! has Schemas

- For type checking
- Mapper – specifies
  - Input key and value type
  - Output key and value type
- Reducer – specifies same
  - DANGER: reduces is not a transformation, so you cannot change the key type
  - Doing so will break the system (silently?? Used to be)
  - Seems like a poor design

Configure and Launch (Driver)

- Configure a job: a class with "public static void main(..)" entry point to be run by Hadoop!
- Assign, output types (seems redundant)
- Assign input and output directories
- Configure
  - Mapper, reducer, **combiner**
- Create a client to manipulate the running job
- LAUNCH!

Driver Code (i)

```
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) {
    JobClient client = new JobClient();
    JobConf conf = new JobConf(WordCount.class);

    // specify output types
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    // specify input and output dirs
    FileInputPath.addInputPath(conf, new Path("input"));
    FileOutputPath.addOutputPath(conf, new Path("output"));
```

Driver Code (ii)

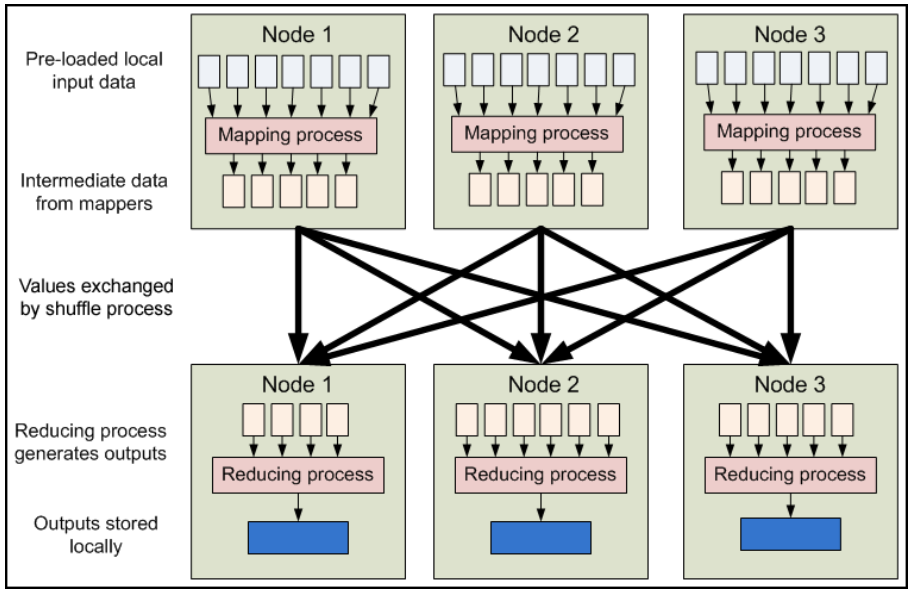
```
    // specify a mapper
    conf.setMapperClass(WordCountMapper.class);

    // specify a reducer
    conf.setReducerClass(WordCountReducer.class);
    conf.setCombinerClass(WordCountReducer.class);

    client.setConf(conf);
    try {
      JobClient.runJob(conf);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Hadoop! Terms

- Combiner
- Shuffle
- Partition

Hadoop! Terms

- Combiner
  - A reducer (like) function that runs at the mapper
  - Used mostly with reducers that are composable to run the reducer many times
  - Makes input to the shuffle smaller, reduce data handling
  - Combiner typically more parallel



Hadoop! Combiner

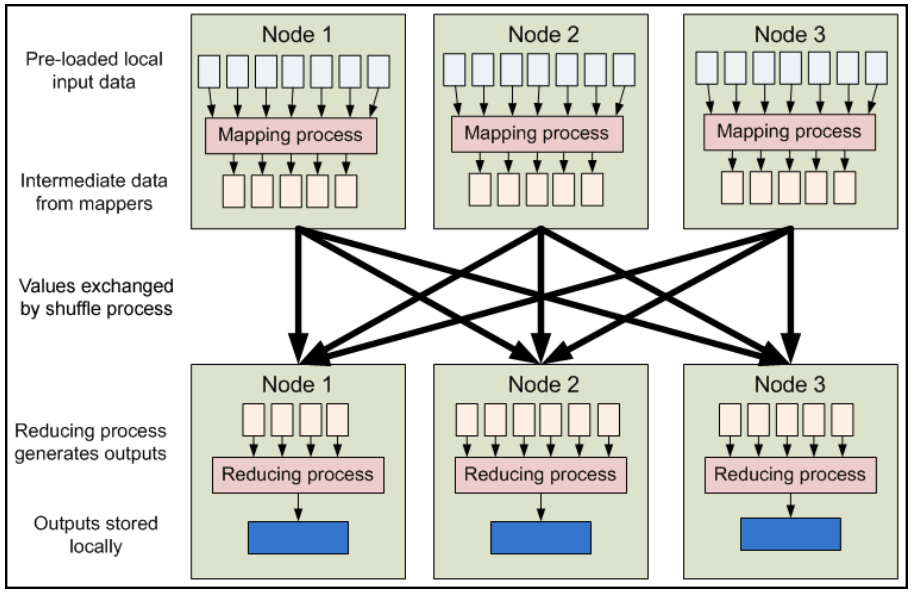
- Combiner executes on the mappers <key,value> output while in memory at the mapper
  - It is possible to write unique combiner and reduce classes
  - It is common to use the reducer as a combiner

- Typical use: pre-compute aggregates or extrema
  - Word count: compute sum output by mapper for each key and send a single aggregated value to reducers
  - Maximum: compute maximum value for each key. Reducer computes a maximum of maxima.

- Caution!!! Your homework assignment cannot use a combiner. I will ask you why?

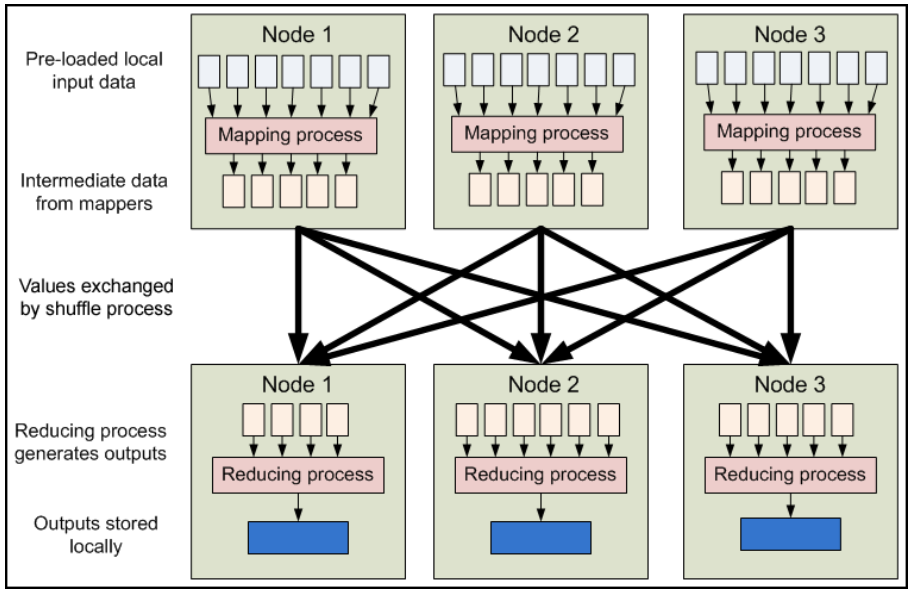
Hadoop! Terms

- Shuffle
  - The process of routing mapper outputs to the reducer inputs
  - This is a giant sort
  - We'll belabor the point



Hadoop! Terms

- Partition
  - The portion of data associated with a parallel execution unit
  - Input partition (to the mapper or reducer)
  - Output partition: the part of the result written by each reducer



The Hadoop Tool Chain

- The command line tool chain
  - Build files into directory
  - Construct java archive (jar)
  - Point Hadoop! at the jar
- I recommend you use Eclipse instead

Hadoop! Streaming

- Given arbitrary string processing functions to the Hadoop! Environment
  - A map script and a reduce script
- Equivalent to:
  - cat inputdir/* | mapper.py | sort | reducer.py

Hadoop! Configurations

- Hadoop! is a heterogeneous, distributed system
  - Many components: namenode, hdfs, reporting
  - Parallelization (mappers, reducers, shuffle, loading)
  - Typically involves managing a cluster
- But can run in several simpler ways
  - Pseudo-distributed (full runtime on one machine)
  - Fully distributed (on a cluster)
- Running on pre-configured clusters
  - Specify size and types of nodes
  - Launch a compiled JAVA jar file or streaming scripts
  - AWS, Azure, Joyent, IBM, RackSpace
  - Metaservices: Cloudera

Sorting by Key

- Map: extract a sorting key from the value
  - <key, value> -> <sort_key, value>
- Reduce: use shuffling properties of reduce
  - <sort_key, value> -> <sort_key, value>
- What's going on?

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

Observations about Sorting

- The ordering guarantees sort within partitions
- To sort completely:
  - All output to a single partition (use one reducer)
  - Customize the shuffle function (quite complex)
  - The default shuffle uses hashing (for load balance)

- The Google paper optimizes sort by customizing shuffle so that partitions are ordered, not randomized
  - Run a M/R job to learn the key distribution
  - Specify a shuffle based on evenly paritioning the key distribution
  - This is also how Hadoop!' s sort record worked

Streaming and Sorting

- Streaming mode in Hadoop! Gives a different sorting guarantee
  - Recall: cat inputdir/* | mapper.py | sort | reducer.py
- *Why?*
- *Same or different semantics?*
- *Any performance implications?*

Streaming and Sorting

- Streaming mode in Hadoop! Gives a different sorting guarantee
  - Recall: cat inputdir/* | mapper.py | sort | reducer.py
- Why?
  - There is no schema
  - So, it sorts the whole output of mapper.py as a key
  - This is more restrictive than the default sort
  - And, thus, less efficient

Map/Reduce Recast (3 y.o. #s)

- Scanning engine
  - Use massive parallelism to look at large data sets
- Performance on 100 TB data sets
  - 1 node @ 50 MB/s (STR of disk) = 23 days
  - 1000 nodes = 33 minutes
- Batch Processing
  - Not real-time/user facing
- Large production environments
  - Not useful on small scales
  - Too much overhead on small jobs