CMSC 433 Fall 2014
Section 0101
Mike Hicks
With slides due to Rance Cleaveland
and Shivnath Babu

UNIVERSITY OF MARYLAND

Lecture 22
Hadoop

Hadoop

- An open-source implementation of MapReduce
- Design desiderata
  - Performance: support processing of huge data sets (millions of files, GB to TB of total data) by exploiting parallelism, memory within computing clusters
  - Economics: control costs by using commodity computing hardware
  - Scalability: a larger the cluster should yield better performance
  - Fault-tolerance: node failure does not cause computation failure
  - Data parallelism: same computation performed on all data

Cluster?

- Hadoop is designed to run on a cluster
  - Multiple machines, typically running Linux
  - Machines connected by high-speed local-area network (e.g. 10-gigabit Ethernet)
- Hardware is:
  - High-end (fast, lots of memory)
  - Commodity (cheaper than specialized equipment)

Principles of Hadoop Design

- *Data is distributed* around network
  - No centralized data server
  - Every node in cluster can host data
  - Data is *replicated* to support fault tolerance
- *Computation is sent to data*, rather than vice versa
  - Code to be run is sent to nodes
  - Results of computations are aggregated as tend
- *Basic architecture is master/worker*
  - Master, aka JobNode, launches application
  - Workers, aka WorkerNodes, perform bulk of computation

Components of Hadoop

- MapReduce
  - Basic APIs in Java supporting MapReduce programming model
- Hadoop Distributed File System (HDFS)
  - Applications see files
  - Behind the scenes: HDFS handles distribution, replication of data on cluster, reading, writing, etc.

MapReduce in Hadoop

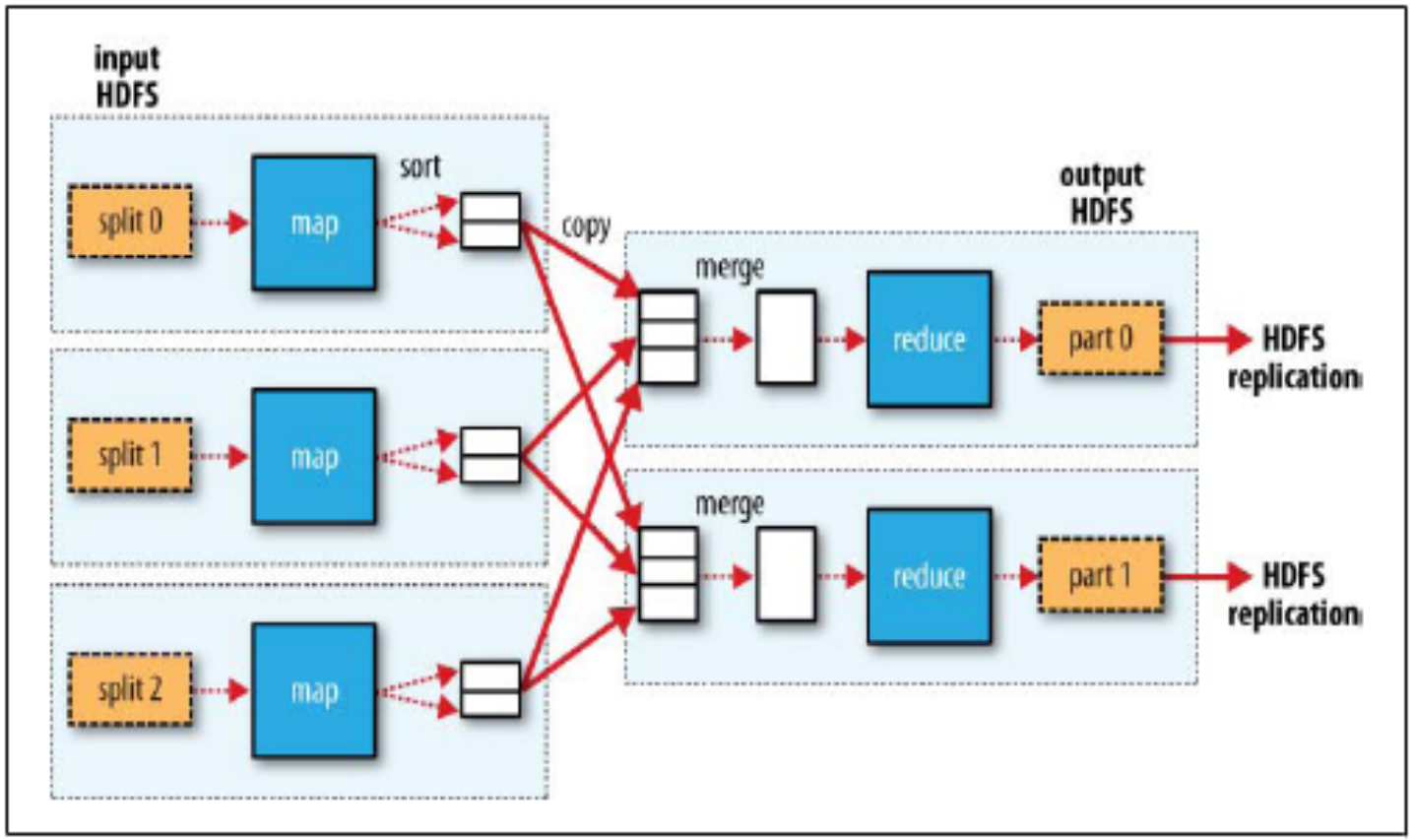


Figure 2-3. MapReduce data flow with multiple reduce tasks

Hadoop Execution: Startup

1. MapReduce library in user program splits input files into pieces (typically 16-64 MB), starts multiple copies of program on cluster
2. One copy is master; rest are workers. Work consists of map, reduce tasks
3. Master keeps track of idle workers, assigns them map / reduce tasks

[Discussion adapted from Ravi Mukkamala, "Hadoop: A Software Framework for Data Intensive Computing Applications"; Hadoop 1.2.1 "MapReduce Tutorial"]

Hadoop Execution: Map Task

1. Read contents of assigned input split
   Master will try to ensure that input split is "close by"
2. Parse input into key/value pairs
3. *Apply map operation* to each key/value pair; store resulting intermediate key/value pairs on local disk
4. File is sorted on output key, then partitioned based on key values
5. Locations of these files forwarded back to Master
6. Master forwards locations of files to relevant reduce workers
   - Which reduce workers get which files depends on *partition*
   - Partition assigns different key values to different reduce tasks

Hadoop Execution: Reduce Task

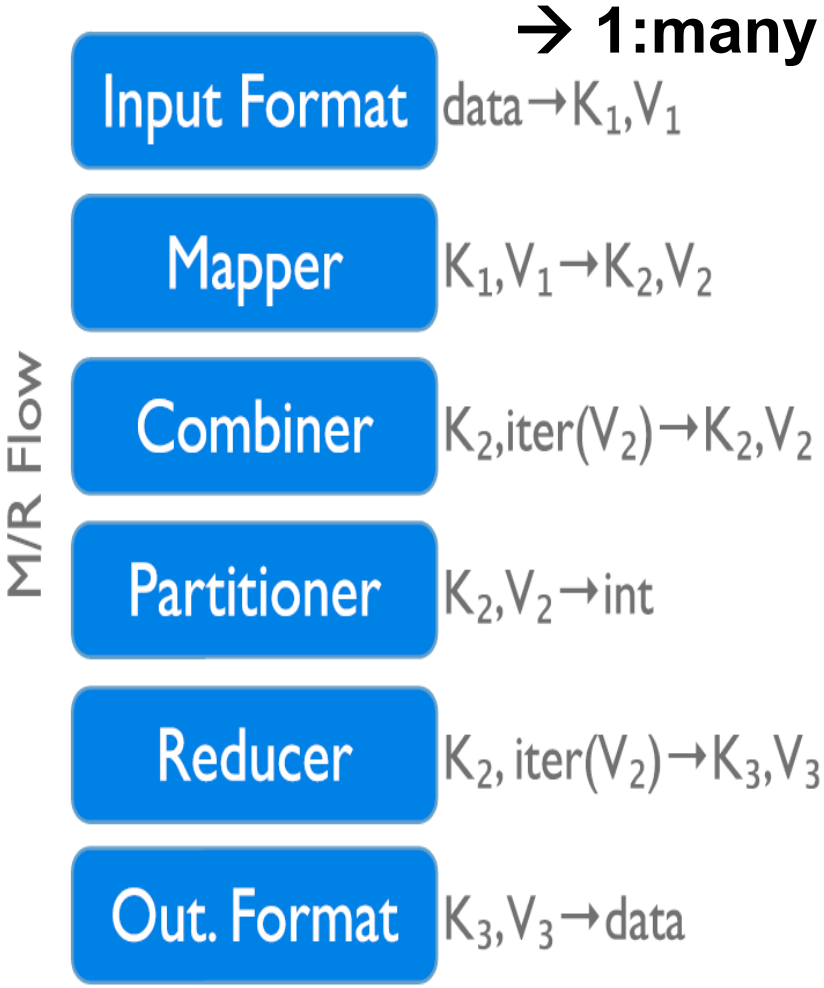
1. Fetch input (files produced by map tasks and sent by master)
2. Sort input data by key
3. For each key, *apply reduce operation* to key / list of values associated with key
4. Write result in file (one output file / key, often, but configurable)
5. Return location of result files to Master

Fault-Tolerance

- Big clusters = increased possibility of hardware failure
  - Disk crashes
  - Overheating
- Worker failure
  - Master pings worker periodically: no response = worker marked as failed
  - Tasks assigned to failed worker added back into task pool for re-assignment
  - This works because *functional nature* of MapReduce ensures no shared state, while HDFS ensures *data is replicated* (so data hosted by failed node is still available)
- Master failure
  - Masters write checkpoint files showing intermediate progress
  - If master fails, a new master can be started from the last checkpoint
  - In practice: job generally restarted

Data Flow in Hadoop

- InputFormat
- Map function
- Partitioner
- Sorting & Merging
- Combiner
- Shuffling
- Merging
- Reduce function
- OutputFormat



Lifecycle of a MapReduce Job

```
File  Edit  Options  Buffers  Tools  Java  Help

public class WordCount {

  public static class Map extends MapReduceBase implements
                Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

  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));
  }}

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}

--:---  mapreduce.java   All L9     (Java/l Abbrev)----------
 Wrote /home/shivnath/Desktop/mapreduce.java
```

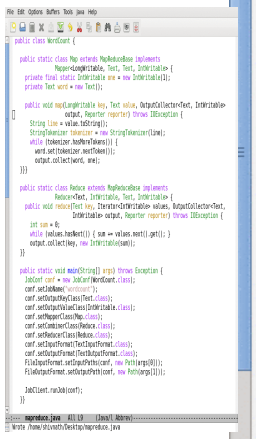
Map function

Reduce function

Run this program as a MapReduce job

Note that this code is for an older Hadoop API than the one we're using for the project

Lifecycle of a MapReduce Job



File Edit Options Buffers Tools Java Help

```
public class WordCount {

  public static class Map extends MapReduceBase implements
                Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

  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));
    }}

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}
```

Map function

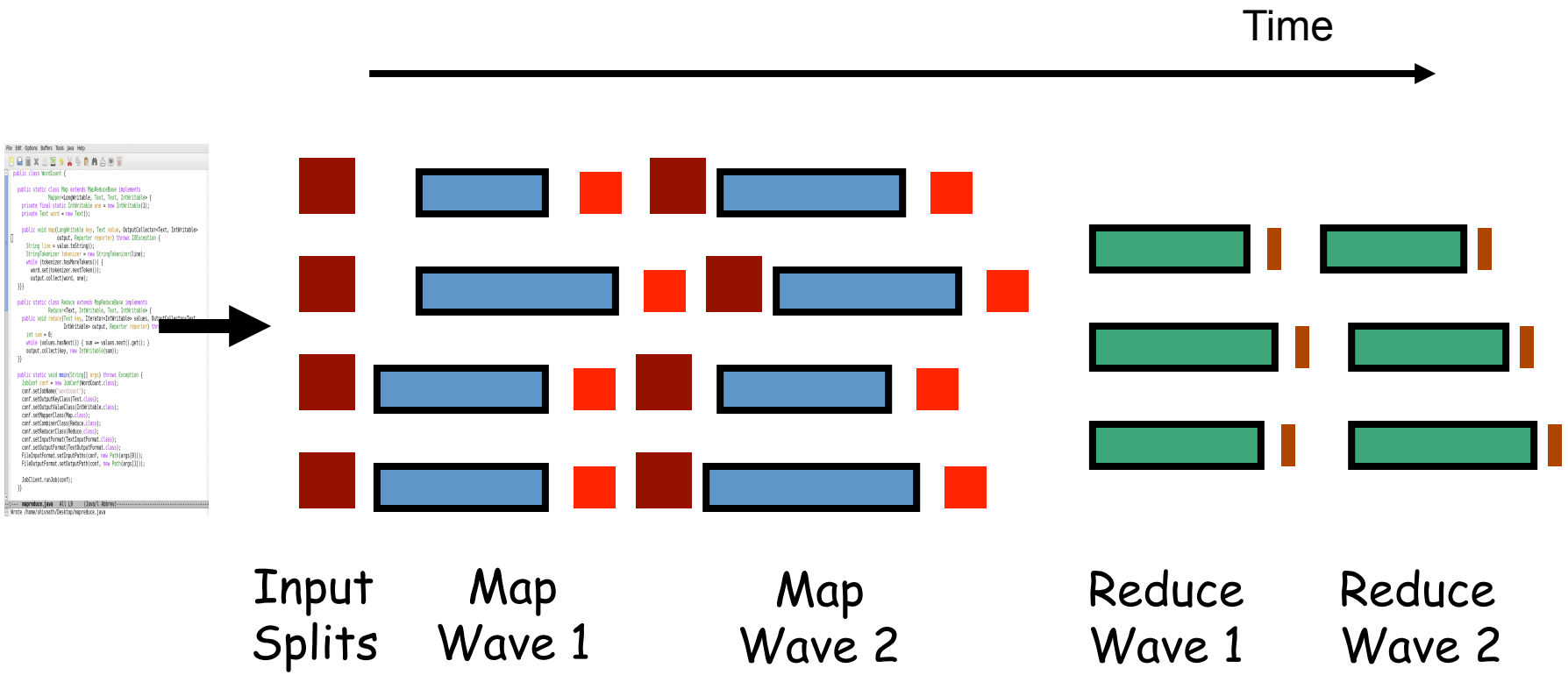
Reduce function

Run this program as a MapReduce job

--:--- **mapreduce.java** All L9 (Java/l Abbrev)----------

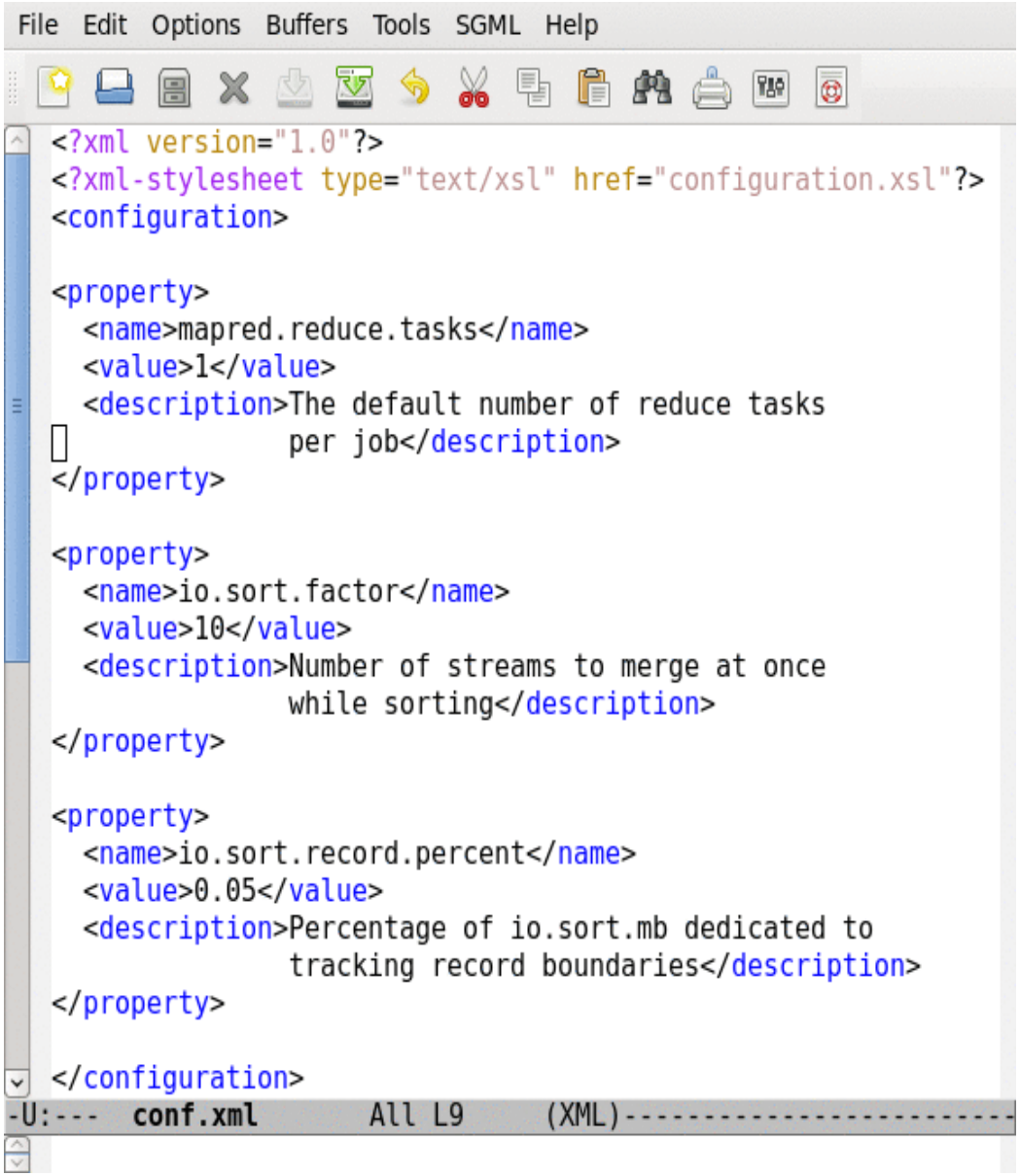
Wrote /home/shivnath/Desktop/mapreduce.java

Lifecycle of a MapReduce Job



How are the number of splits, number of map and reduce tasks, memory allocation to tasks, etc., determined?

Configuring MapReduce Execution



- 190+ parameters in Hadoop
- Set manually or defaults are used

Configuring MapReduce Execution

- Many configuration parameters to tune performance!
  - Number of maps
  - Number of reduces
  - Splitting of input
  - Sorting, partitioning
  - Etc.
- Hadoop MapReduce tutorial gives a starting point

  http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Setting Up Hadoop

- Three possibilities
  - Local standalone (everything run in one process)
  - Pseudo-distributed (tasks run as separate processes on same machine)
  - Fully distributed (cluster computing)
- ???
  - Standalone usually used for development, debugging
  - Pseudo-distributed to analyze performance bottlenecks
  - Fully-distributed used for final job runs

Installing, Running Stand-alone Hadoop

- Download 0.20.2, extract files (remember where!)
  - There are multiple "development streams" in Hadoop: 0.xx, 1.xx, 2.xx
  - Why? Who knows
  - 0.20.2 is one guaranteed to work with Windows / Macs / etc.
- Follow instructions in P5 (basically, need to tell Eclipse where Hadoop libraries are that you just downloaded)
  - Download Hadoop-0.20.2.tar.gz
  - Unpack
  - Add .jar files listed in P5 to build path

Writing a Hadoop Application

- MapReduce
  - One class should extend Mapper<K1,V1,K2,V2>
    - K1, V1 are key/value classes for input
    - K2, V2 are key/value classes for output
  - Another should extend Reducer<K2,V2,K3,V3>
    - K2, V2 are key/value classes for inputs to reduce operation
    - K3, V3 are output key/value classes
- Main driver
  - Need to create an object in Job (Hadoop class) containing configuration settings for Hadoop application
  - Settings include input / output file formats for job, input file-slice size, key/value types, etc.
  - To run the job: invoke `job.waitForCompletion(true)`

Implementing Mapper<K1,V1,K2,V2>

- Key function to implement:

```
public void map(K1 key, V1 value, Context c)
```

  - First two inputs are key / value pair, which map should convert into key2 / value2 pairs
  - "Context"?
    - Used to store key2 / value2 pairs produced by map
    - Context is a Hadoop class
    - To store a newly created key2 / value2 pair, invoke:

```
c.write (key2, value2);
```

- Hadoop takes care of ensuring that pairs written into context are provided to Reducer!

Implementing Reducer<K2,V2,K3,V3>

- Key function to implement:

```
public void reduce(K2 key, Iterable<V2> values, Context c)
```

  - First args are key / list-of-values pair, which map should convert into (usually at most one) key3 / value3 pairs
  - Context argument used to store these key3 / value3 pairs!
    - Idea is same as for Mapper implementation!
    - To store a newly created key2 / value2 pair, invoke:

```
c.write (key2, value2);
```

- Hadoop takes care of ensuring that pairs written into context are made available for post-processing (i.e. sorting, writing into a file)

Implementing main()

- Must create a Job object (Hadoop class)
  - Job constructor typically requires a Context argument
  - E.g.:

```
Configuration conf = new Configuration ();
Job job = new Job(conf);
```

- Job object must be configured!
  - Key, Value classes must be set
  - Mapper, Reducer classes (your implementation!) must be specified
  - Input / output formatting must be given
  - Paths to input files, output files must be given

Sample main() (from WordCount.java)

```
public static void main(String[] args) throws Exception {

  // Set up and configure MapReduce job.
  Configuration conf = new Configuration ();
  Job job = new Job(conf);
  job.setJobName("WordCount");
  job.setJarByClass(WordCount.class);  // In Eclipse this will not create JAR file

  // Set key, output classes for the job (same as output classes for Reducer)
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  // Set Mapper and Reducer classes for the job. (Combiner often not needed.)
  job.setMapperClass(MapClass.class);
  job.setReducerClass(ReduceClass.class);
```

Sample main() (cont.)

```
  // Sets format of input files.  "TextInputFormat" views files as a sequence of lines.
  job.setInputFormatClass(TextInputFormat.class);

  // Sets format of output files:  here, lines of text.
  job.setOutputFormatClass(TextOutputFormat.class);

  // Set paths for location of input, output.  Note former is assumed to be
  // initial command-line argument, while latter is second.  No error-checking
  // is performed on this, so there is a GenericOptionsParser warning when run.

  TextInputFormat.setInputPaths(job, new Path(args[0]));
  TextOutputFormat.setOutputPath(job, new Path(args[1]));

  // Run job
  Date startTime = new Date();
  System.out.println("Job started: " + startTime);
  boolean success = job.waitForCompletion(true);
  if (success) {
    Date end_time = new Date();
    System.out.println("Job ended: " + end_time);
    System.out.println("The job took " + (end_time.getTime() - startTime.getTime()) /1000 + " seconds.");
  }
  else { System.out.println ("Job failed."); }
}
```

Running a Hadoop Application

- Arguments configuring job given as command-line arguments
  - To parse these arguments, can use GenericOptionsParser class provided by Hadoop.
  - Can also "roll your own" command-line parser, but Hadoop will give a warning
- Need to have "chmod" installed in order to run!
  - Unix command for changing permissions on files and directories.
  - This command is not native to Windows!
  - Easiest way to obtain a chmod.exe:
    - Install Cygwin ("Unix for Windows")
    - In Cygwin installation, locate directory containing "chmod.exe" (e.g., "C:\cygwin\bin").
    - Add this directory to your Windows path by editing Path system variable
      - Right-click on Computer, select "Properties"
      - Select "Advanced system settings"
      - Click on "Environment Variables"
      - Select "Path" variable in "System variables" pane, then Edit… button
      - Go to end, add ";" then path where chmod.exe is