Hadoop and Map Reduce

Terminology

Google calls it:	Hadoop equivalent:
MapReduce	Hadoop
GFS	HDFS
Bigtable	HBase
Chubby	Zookeeper

Cloud Resources

- Hadoop on your local machine
- Hadoop in a virtual machine on your local machine (Pseudo-Distributed on **Ubuntu**)
- Hadoop in the clouds with Amazon EC2

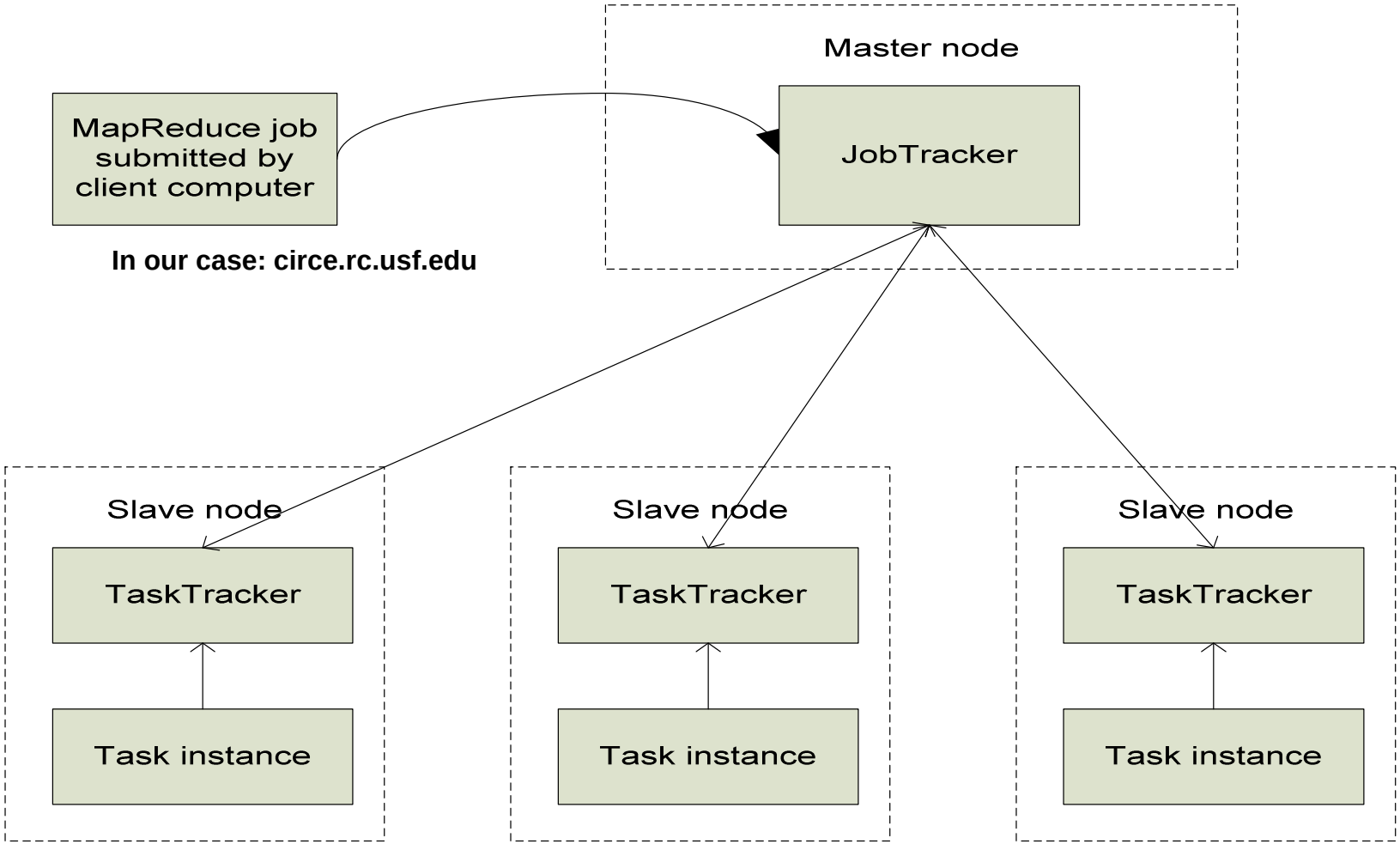
Some MapReduce Terminology

- *Job* – A “full program” - an execution of a Mapper and Reducer across a data set
- *Task* – An execution of a Mapper or a Reducer on a slice of data
  - a.k.a. Task-In-Progress (TIP)
- Task Attempt – A particular instance of an attempt to execute a task on a machine

Task Attempts

- A particular task will be attempted at least once, possibly more times if it crashes
  - If the same input causes crashes over and over, that input will eventually be abandoned
- Multiple attempts at one task may occur in parallel with speculative execution turned on
  - Task ID from *TaskInProgress* is not a unique identifier; don't use it that way

MapReduce: High Level



Nodes, Trackers, Tasks

- Master node runs *JobTracker* instance, which accepts *Job* requests from clients
- *TaskTracker* instances run on slave nodes
- TaskTracker forks separate Java process for task instances

Job Distribution

- MapReduce programs are contained in a Java “jar” file + an XML file containing serialized program configuration options
- Running a MapReduce job places these files into the HDFS and notifies TaskTrackers where to retrieve the relevant program code

Data Distribution

- Implicit in design of MapReduce!
  - All mappers are equivalent; so map whatever data is local to a particular node in HDFS
- If lots of data does happen to pile up on the same node, nearby nodes will map instead
  - Data transfer is handled implicitly by HDFS

Job and Task Tracker

- The JobTracker is the service within Hadoop that farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack.

Steps

- Client applications submit jobs to the Job tracker.
- The JobTracker talks to the NameNode to determine the location of the data
- The JobTracker locates TaskTracker nodes with available slots at or near the data
- The JobTracker submits the work to the chosen TaskTracker nodes.

Job and Task Tracker

- The TaskTracker nodes are monitored. If they do not submit heartbeat signals often enough, they are deemed to have failed and the work is scheduled on a different TaskTracker.
- A TaskTracker will notify the JobTracker when a task fails. The JobTracker decides what to do then:
  - it may resubmit the job elsewhere,
  - it may mark that specific record as something to avoid
  - it may may even blacklist the TaskTracker as unreliable.
- When the work is completed, the JobTracker updates its status.

Job and Task Tracker

- Client applications can poll the JobTracker for information.
- The JobTracker is a point of failure for the Hadoop MapReduce service. If it goes down, all running jobs are halted.

MapReduce

- Programmers specify two functions:

  map (k, v) → [(k', v')]

  reduce (k', [v']) → [(k', v'')]

  – All values with the same key (k') are sent to the same reducer, in k' order for each reducer

  – Here [] means a sequence

- The execution framework handles everything else…

“Hello World”: Word Count

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
    Emit(term, sum);
```

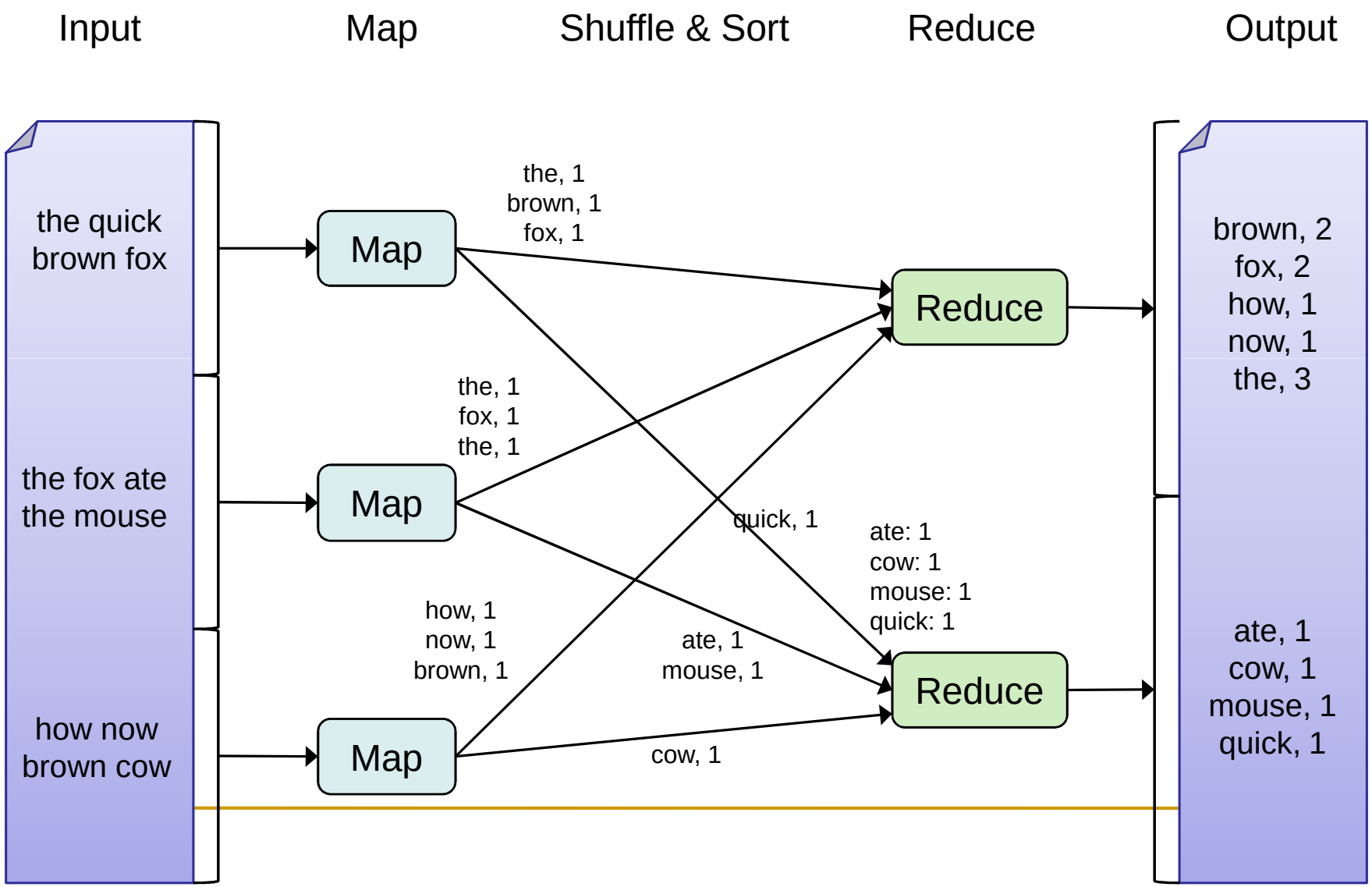
MapReduce "Runtime"

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

MapReduce

- Programmers specify two functions:
  map (k, v) → [(k', v')]
  reduce (k', [v']) → [(k', v'')]
  – All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:
  partition (k', number of partitions) → partition for k'
  – Often a simple hash of the key, e.g., hash(k') mod n
  – Divides up key space for parallel reduce operations
    - and eventual delivery of results to certain partitions
  combine (k', [v']) → [(k', v'')]
  – Mini-reducers that run in memory after the map phase
  – Used as an optimization to reduce network traffic

Word Count Execution



What Happens In Hadoop? Depth First

Job Launch Process: Client

- Client program creates a *JobConf*
  - Identify classes implementing *Mapper* and *Reducer* interfaces
    - JobConf.setMapperClass(), setReducerClass()
  - Specify inputs, outputs
    - FileInputFormat.setInputPath(),
    - FileOutputFormat.setOutputPath()
  - Optionally, other options too:
    - JobConf.setNumReduceTasks(), JobConf.setOutputFormat()…

Job Launch Process: *JobClient*

- Pass JobConf to JobClient.runJob() or submitJob()
  - runJob() blocks, submitJob() does not
- *JobClient*:
  - Determines proper division of input into *InputSplits*
  - Sends job data to master *JobTracker* server

Job Launch Process: *JobTracker*

- *JobTracker*:
  - Inserts jar and JobConf (serialized to XML) in shared location
  - Posts a *JobInProgress* to its run queue

Job Launch Process: *TaskTracker*

- *TaskTrackers* running on slave nodes periodically query *JobTracker* for work
- Retrieve job-specific jar and config
- Launch task in separate instance of Java
  - main() is provided by Hadoop

Job Launch Process: Task

- TaskTracker.Child.main():
  - Sets up the child *TaskInProgress* attempt
  - Reads XML configuration
  - Connects back to necessary MapReduce components via RPC
  - Uses *TaskRunner* to launch user process

Job Launch Process: *TaskRunner*

- *TaskRunner*, *MapTaskRunner*, *MapRunner* work in a daisy-chain to launch your *Mapper*
  - Task knows ahead of time which *InputSplits* it should be mapping
  - Calls *Mapper* once for each record retrieved from the InputSplit
- Running the *Reducer* is much the same

Creating the *Mapper*

- You provide the instance of *Mapper*
  - Should extend *MapReduceBase*
- One instance of your Mapper is initialized by the *MapTaskRunner* for a *TaskInProgress*
  - Exists in separate process from all other instances of Mapper – no data sharing!

Mapper

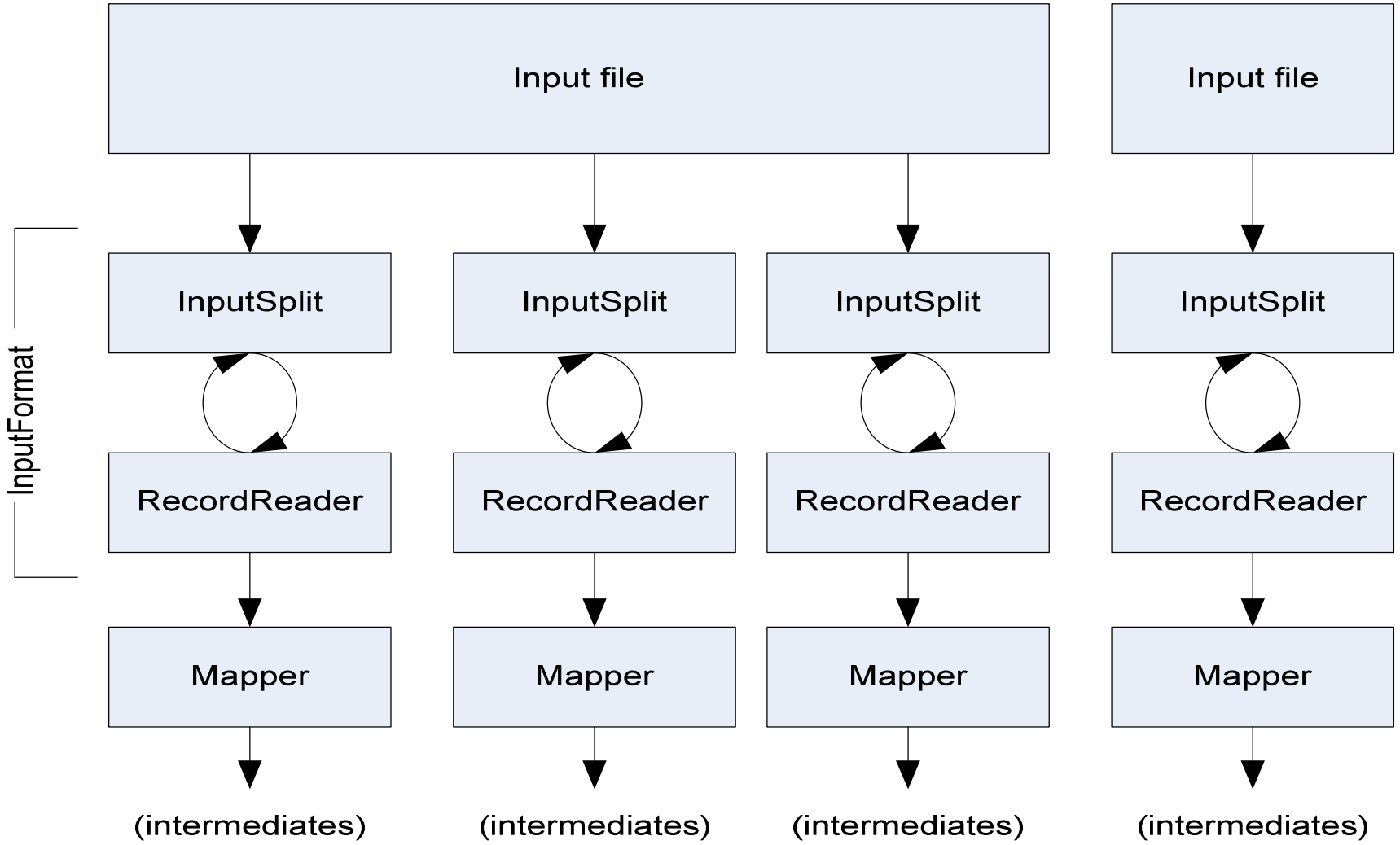
- void map(K1 key,
  V1 value,
  OutputCollector<K2, V2> output,
  Reporter reporter)

- *K* types implement *WritableComparable*
- *V* types implement *Writable*

What is Writable?

- Hadoop defines its own “box” classes for strings *(Text),* integers *(IntWritable)*, etc.
- All values are instances of *Writable*
- All keys are instances of *WritableComparable*

Getting Data To The Mapper



Reading Data

- Data sets are specified by *InputFormats*
  - Defines input data (e.g., a directory)
  - Identifies partitions of the data that form an *InputSplit*
  - Factory for *RecordReader* objects to extract (k, v) records from the input source

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())

Filtering File Inputs

- *FileInputFormat* will read all files out of a specified directory and send them to the mapper
- Delegates filtering this file list to a method subclasses may override
  - *e.g.,* Create your own “xyzFileInputFormat” to read *.xyz from directory list

Record Readers

- Each *InputFormat* provides its own *RecordReader* implementation
  - Provides (unused?) capability multiplexing
- *LineRecordReader* – Reads a line from a text file
- *KeyValueRecordReader* – Used by *KeyValueTextInputFormat*

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”

Sending Data To Reducers

- Map function receives *OutputCollector* object
  - OutputCollector.collect() takes (k, v) elements
- Any *(WritableComparable, Writable)* can be used
- By default, mapper output type assumed to be same as reducer output type

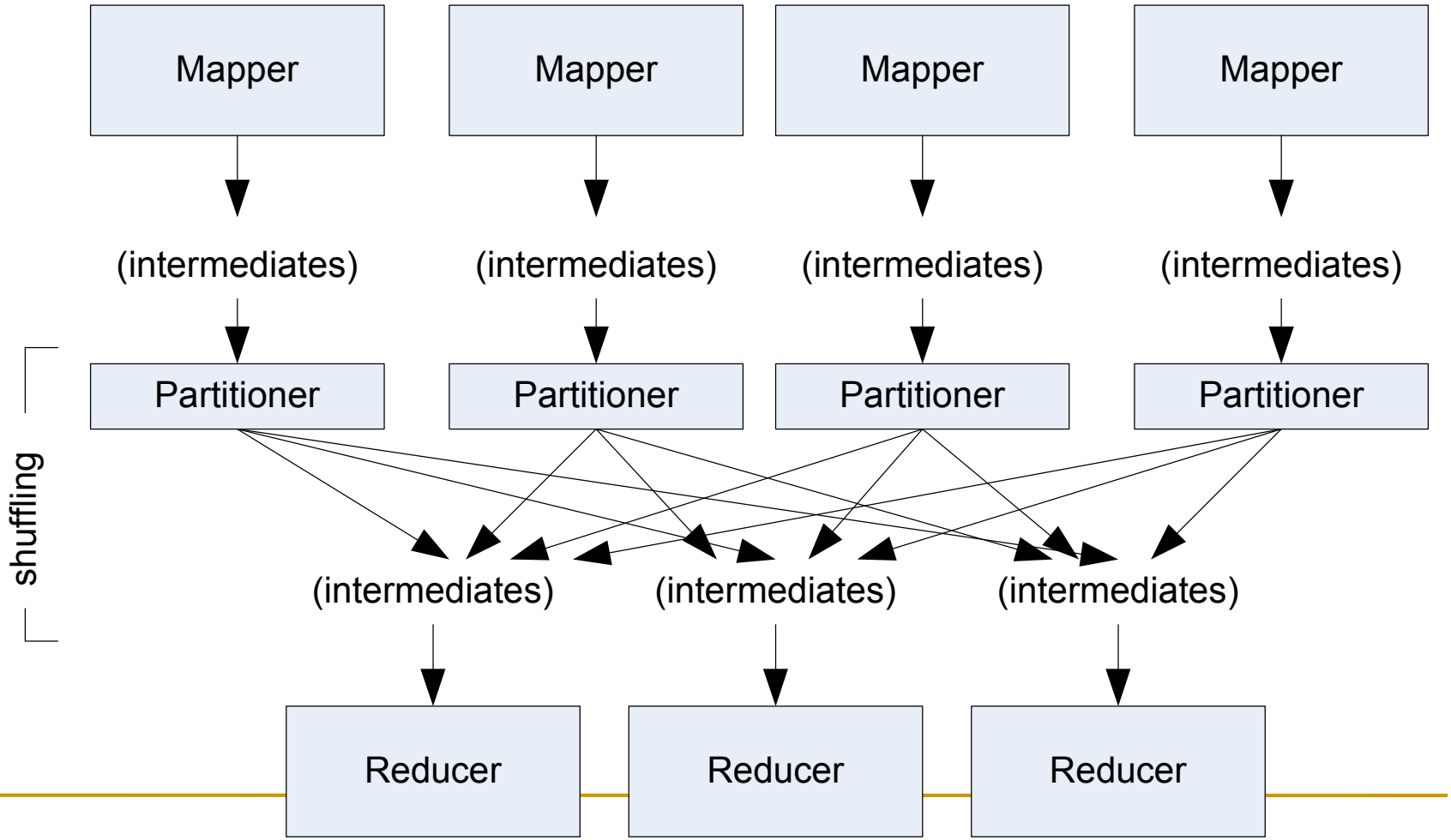
WritableComparator

- Compares WritableComparable data
  - Will call WritableComparable.compare()
  - Can provide fast path for serialized data
- *JobConf.setOutputValueGroupingComparator()*

Sending Data To The Client

- *Reporter* object sent to Mapper allows simple asynchronous feedback
  - incrCounter(Enum key, long amount)
  - setStatus(String msg)
- Allows self-identification of input
  - InputSplit getInputSplit()

Partition And Shuffle



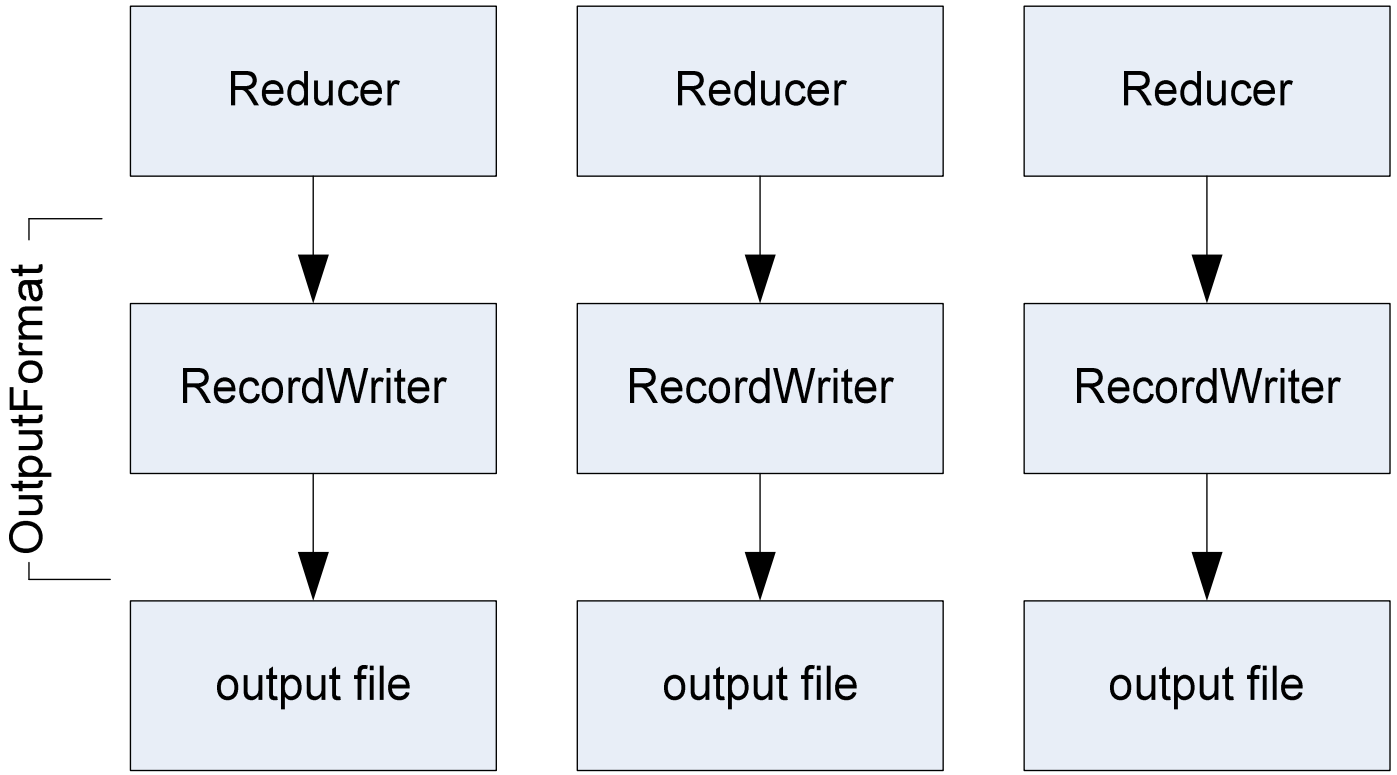
Partitioner

- 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

Reduction

- reduce(K2 key,
  Iterator<V2> values,
  OutputCollector<K3, V3> output,
  Reporter reporter)
- Keys & values sent to one partition all go to the same reduce task
- Calls are sorted by key – “earlier” keys are reduced and output before “later” keys

Finally: Writing The Output



OutputFormat

- Analogous to *InputFormat*
- *TextOutputFormat* – Writes “key val\n” strings to output file
- *SequenceFileOutputFormat* – Uses a binary format to pack (k, v) pairs
- *NullOutputFormat* – Discards output
  - Only useful if defining own output methods within reduce()

Example Program - Wordcount

- map()
  - Receives a chunk of text
  - Outputs a set of word/count pairs
- reduce()
  - Receives a key and all its associated values
  - Outputs the key and the sum of the values

```
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {
```

Wordcount – main( )

```
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.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);
}
```

Wordcount – map( )

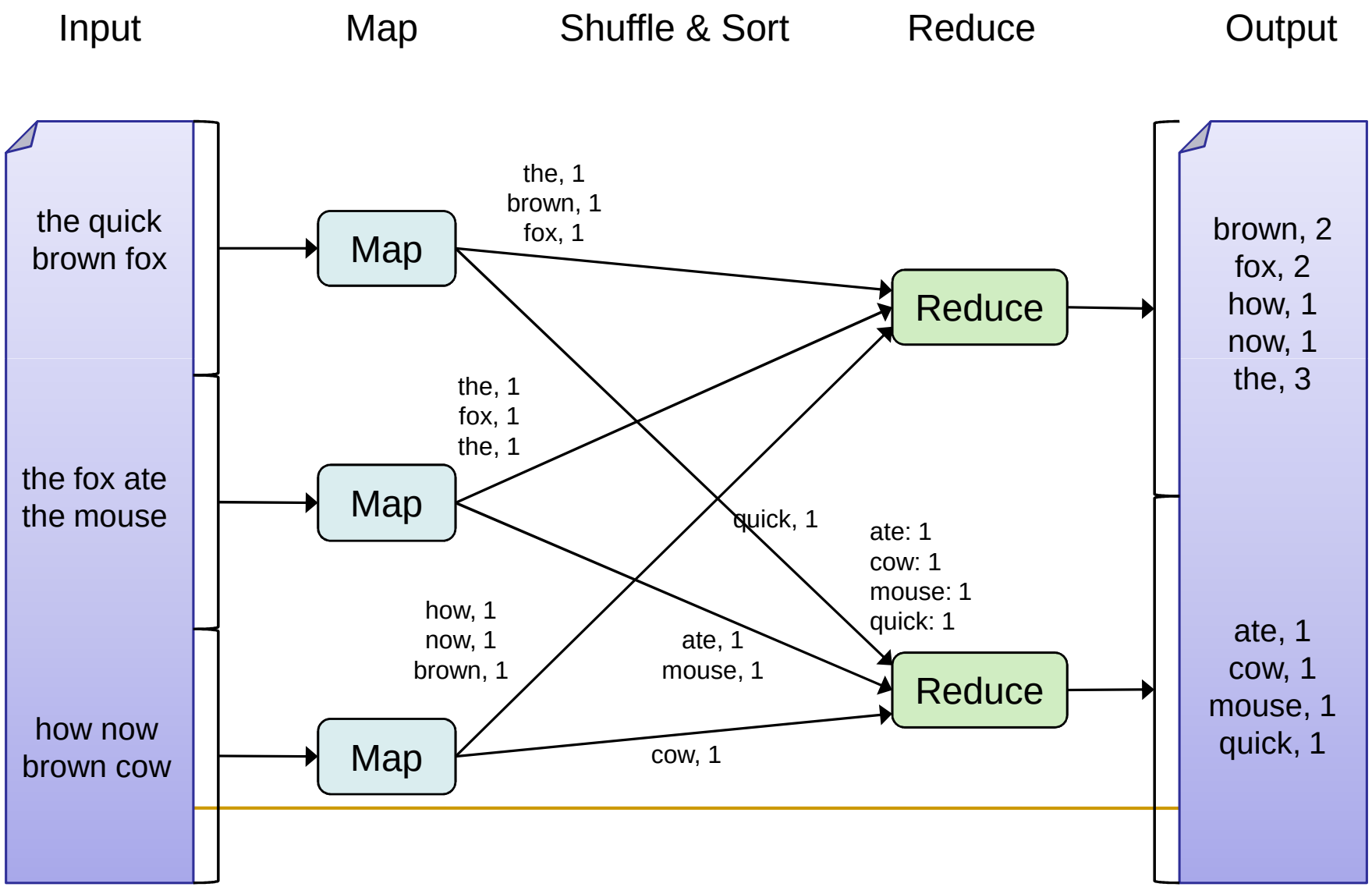
```
public static class Map extends MapReduceBase … {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

Wordcount – reduce( )

```
public static class Reduce extends MapReduceBase … {
    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output, …) … {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
}
```

Word Count Execution



Hadoop Streaming

- Allows you to create and run map/reduce jobs with any executable
- Similar to unix pipes, e.g.:
  - format is: Input | Mapper | Reducer
  - echo “this sentence has five lines” | cat | wc

Hadoop Streaming

- Mapper and Reducer receive data from stdin and output to stdout
- Hadoop takes care of the transmission of data between the map/reduce tasks
  - It is still the programmer's responsibility to set the correct key/value
  - Default format: "key \t value\n"
- Let's look at a Python example of a MapReduce word count program…

Streaming_Mapper.py

```
# read in one line of input at a time from stdin
for line in sys.stdin:
    line = line.strip()          # string
    words = line.split()         # list of strings

    # write data on stdout
    for word in words:
        print '%s\t%i' % (word, 1)
```

Hadoop Streaming

- What are we outputting?
  - Example output: “the 1”
  - By default, “the” is the key, and “1” is the value
- Hadoop Streaming handles delivering this key/value pair to a Reducer
  - Able to send similar keys to the same Reducer or to an intermediary Combiner

Streaming_Reducer.py

```
wordcount = { }          # empty dictionary
# read in one line of input at a time from stdin
for line in sys.stdin:
    line = line.strip()          # string
    key,value = line.split()
    wordcount[key] = wordcount.get(key, 0) + value

    # write data on stdout
    for word, count in sorted(wordcount.items()):
        print '%s\t%i' % (word, count)
```

Hadoop Streaming Gotcha

- Streaming Reducer receives single lines (which are key/value pairs) from stdin
  - Regular Reducer receives a collection of all the values for a particular key
  - It is still the case that all the values for a particular key will go to a single Reducer

Using Hadoop Distributed File System (HDFS)

- Can access HDFS through various shell commands (see Further Resources slide for link to documentation)
  - hadoop –put <localsrc> … <dst>
  - hadoop –get <src> <localdst>
  - hadoop –ls
  - hadoop –rm file

Configuring Number of Tasks

- Normal method
  - jobConf.setNumMapTasks(400)
  - jobConf.setNumReduceTasks(4)
- Hadoop Streaming method
  - -jobconf mapred.map.tasks=400
  - -jobconf mapred.reduce.tasks=4
- Note: # of map tasks is only a hint to the framework. Actual number depends on the number of InputSplits generated

Running a Hadoop Job

- Place input file into HDFS:
  - hadoop fs –put ./input-file input-file
- Run either normal or streaming version:
  - hadoop jar Wordcount.jar org.myorg.Wordcount input-file output-file
  - hadoop jar hadoop-streaming.jar \
    -input input-file \
    -output output-file \
    -file Streaming_Mapper.py \
    -mapper python Streaming_Mapper.py \
    -file Streaming_Reducer.py \
    -reducer python Streaming_Reducer.py \

Submitting to RC's GridEngine

- Add appropriate modules
  - module add apps/jdk/1.6.0_22.x86_64 apps/hadoop/0.20.2
- Use the submit script posted in the Further Resources slide
  - Script calls internal functions hadoop_start and hadoop_end
- Adjust the lines for transferring the input file to HDFS and starting the hadoop job using the commands on the previous slide
- Adjust the expected runtime (generally good practice to overshoot your estimate)
  - #$ -l h_rt=02:00:00

- NOTICE: "All jobs are required to have a hard run-time specification. Jobs that do not have this specification will have a default run-time of 10 minutes and will be stopped at that point."

Output Parsing

- Output of the reduce tasks must be retrieved:
  - hadoop fs –get output-file hadoop-output
- This creates a directory of output files, 1 per reduce task
  - Output files numbered part-00000, part-00001, etc.
- Sample output of Wordcount
  - head –n5 part-00000

    “’tis 1
    “come 2
    “coming 1
    “edwin 1
    “found 1

Extra Output

- The stdout/stderr streams of Hadoop itself will be stored in an output file (whichever one is named in the startup script)
  - #$ -o output.$job_id

```
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = svc-3024-8-10.rc.usf.edu/10.250.4.205
...
11/03/02 18:28:47 INFO mapred.FileInputFormat: Total input paths to process : 1
11/03/02 18:28:47 INFO mapred.JobClient: Running job: job_local_0001
...
11/03/02 18:28:48 INFO mapred.MapTask: numReduceTasks: 1
...
11/03/02 18:28:48 INFO mapred.TaskRunner: Task 'attempt_local_0001_m_000000_0' done.
11/03/02 18:28:48 INFO mapred.Merger: Merging 1 sorted segments
11/03/02 18:28:48 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of total
    size: 43927 bytes
11/03/02 18:28:48 INFO mapred.JobClient:  map 100% reduce 0%
...
11/03/02 18:28:49 INFO mapred.TaskRunner: Task 'attempt_local_0001_r_000000_0' done.
11/03/02 18:28:49 INFO mapred.JobClient: Job complete: job_local_0001
```

Further Resources

- GridEngine User's Guide: **http://rc.usf.edu/trac/doc/wiki/gridEngineUsers**
- GridEngine Hadoop Submission Script: http://rc.usf.edu/trac/doc/wiki/Hadoop
- Hadoop Tutorial: http://developer.yahoo.com/hadoop/tutorial/module1.html
- Hadoop Streaming: http://hadoop.apache.org/common/docs/r0.15.2/streaming.html
- Hadoop API: http://hadoop.apache.org/common/docs/current/api
- HDFS Commands Reference: http://hadoop.apache.org/hdfs/docs/current/file_system_shell.html