Parallel Programming

Hadoop and MapReduce Programming

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Acknowledgement

- This set of slides is adapted from lectures by
  - Prof. Jimmy Lin (University of Maryland), "Introduction to Cloud Computing"
  - Prof. Anand Rajaraman (Stanford University), "Data Mining"
  - Prof. Adriana Iamnitchi (University of South Florida), "Basics of Parallel and Distributed Systems"

Outline

- Introduction to MapReduce
  - Google File System
- The Hadoop Framework

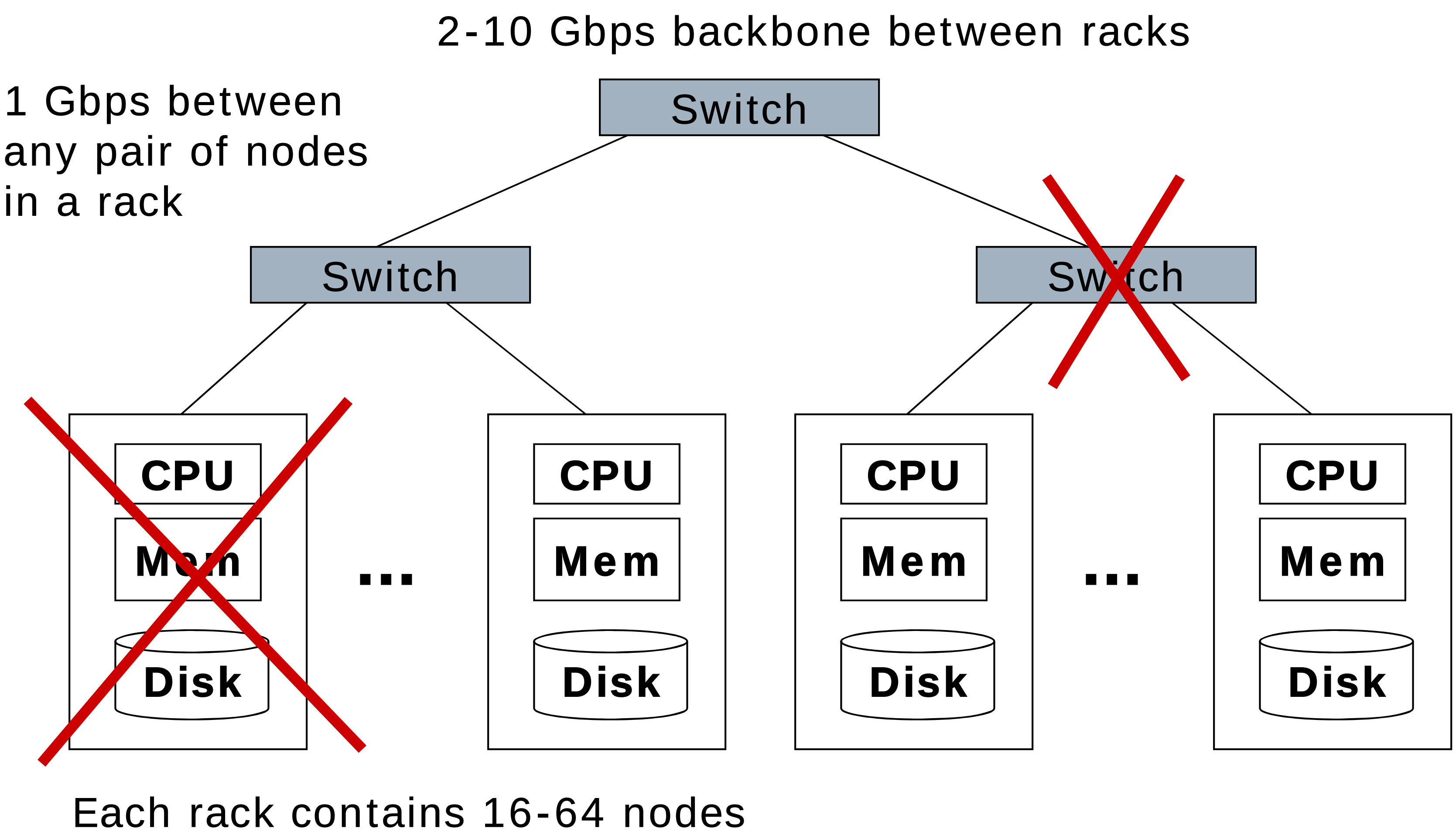
What is MapReduce?

- Data-parallel programming model for clusters of commodity machines
- Pioneered by Google (published in OSDI'04)
  - Processes 20 PB of data per day
    - Index building for Google Search
    - Article clustering for Google News
    - Statistical machine translation
- Popularized by Apache Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …
    - Index building for Yahoo! Search
    - Spam detection for Yahoo! Mail
    - Ad optimization for Facebook

Commodity Clusters

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- How to organize computations on this architecture?
  - Mask issues such as hardware failure

Cluster Architecture



Stable storage

- First order problem: if nodes can fail, how can we store data persistently?
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File System

- Chunk Servers
  - a.k.a. Data Nodes in HDFS
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - a.k.a. Name Node in HDFS
  - Stores metadata
  - Might be replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunkservers to access data

Motivation for MapReduce

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But don't want hassle of ***managing*** things
- MapReduce Architecture provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

MapReduce Goals

- Cloud Environment:
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - Automatic fault-tolerance (fewer admins)
- Scalability to large data volumes:
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

What is Map/Reduce

- Map/Reduce
  - Programming model from LISP
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics

Map in LISP (Scheme)

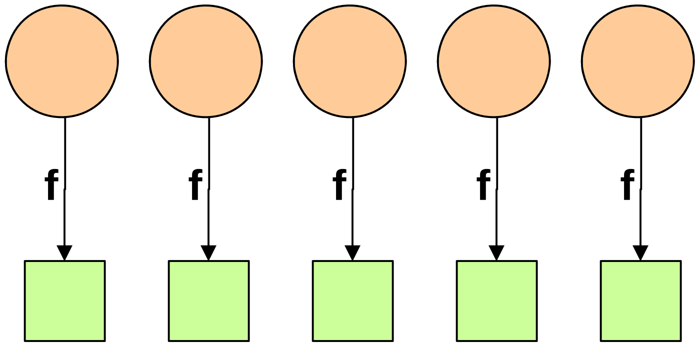
- (map ***f list [list$_2$ list$_3$ …]***)

Unary operator

- (map square ‘(1 2 3 4))
  - (1 4 9 16)

Map

- Map is a higher-order function
- How map works:
  - Function is applied to every element in a list
  - Result is a new list
- No limit to map parallelization since maps are independent



Reduce in LISP (Scheme)

- (reduce ***f id list***)

Binary operator

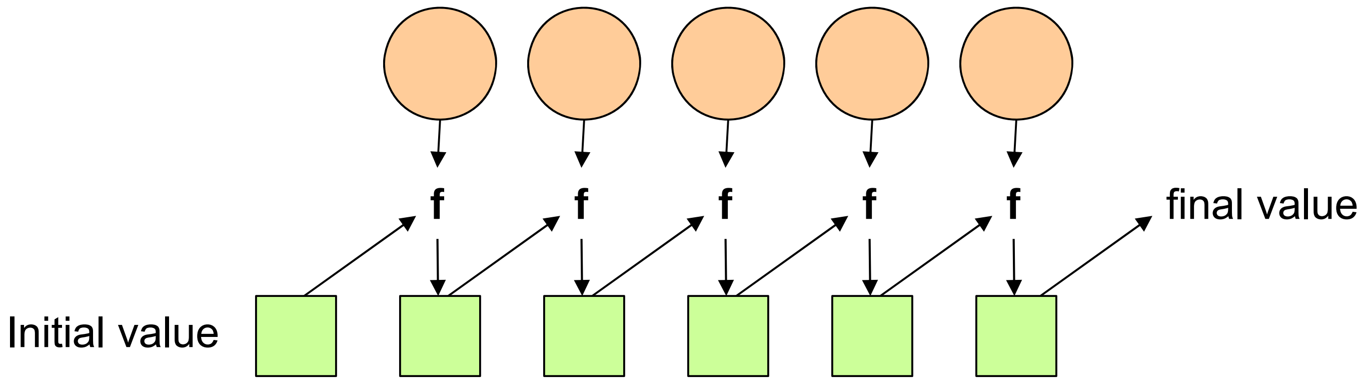
- (reduce + 0 ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 (+ 1 0))))
  - 30

- (reduce + 0
  (map square (l_1 l_2)))

Reduce

- Reduce is also a higher-order function
- How reduce works:
  - Accumulator set to initial value
  - Function applied to list element and the accumulator
  - Result stored in the accumulator
  - Repeated for every item in the list
  - Result is the final value in the accumulator

Reduce



- We can reorder folding if the reduce function is commutative and associative

Typical Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide an abstraction at the point of these two operations

MapReduce

- Programmers specify two functions:

  Map function:

  $$(K_{in}, V_{in}) \Rightarrow list<(K_{inter}, V_{inter})>$$

  Reduce function:

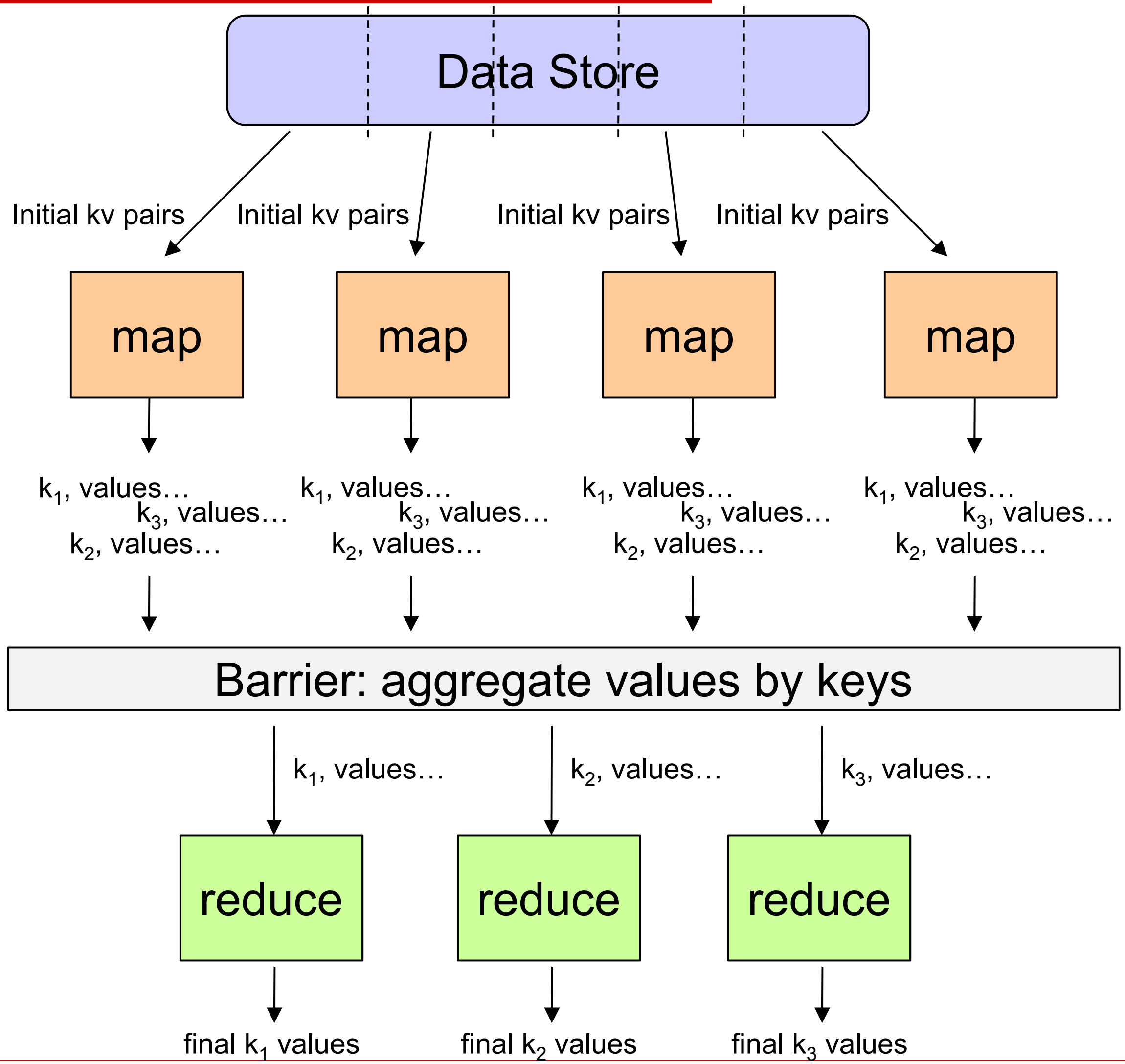
  $$(K_{inter}, list<V_{inter}>) \Rightarrow list<(K_{out}, V_{out})>$$

- 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
  - Allows reduce operations for different keys in parallel

It's just divide and conquer



“Hello World”: Word Count

- We have a large file of words
- Count the number of times each distinct word appears in the file
- *Sample application*: analyze web server logs to find popular URLs

Word Count (2)

- Case 1: Entire file fits in memory
- Case 2: File too large for mem, but all <word, count> pairs fit in mem
- Case 3: File on disk, too many distinct words to fit in memory
  - **`xargs –n1 < datafile | sort | uniq –c`**

Word Count (3)

- To make it slightly harder, suppose we have a large corpus of documents
- Count the number of times each distinct word occurs in the corpus

  `words(docs/*) | sort | uniq -c`

  where **`words`** takes a file and outputs the words in it, one to a line
- The above captures the essence of MapReduce
  - Great thing is that it is naturally parallelizable

MapReduce

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs

Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)


reduce(key, values):
// key: a word; values: an iterator over counts
        result = 0
        for each count v in values:
                result += v
        emit(key,result)
```

Count, Illustrated

map(key=url, val=contents):

For each word *w* in contents, emit (w,”1”)

reduce(key=word,
values=uniq_counts):

Sum all “1” s in values list

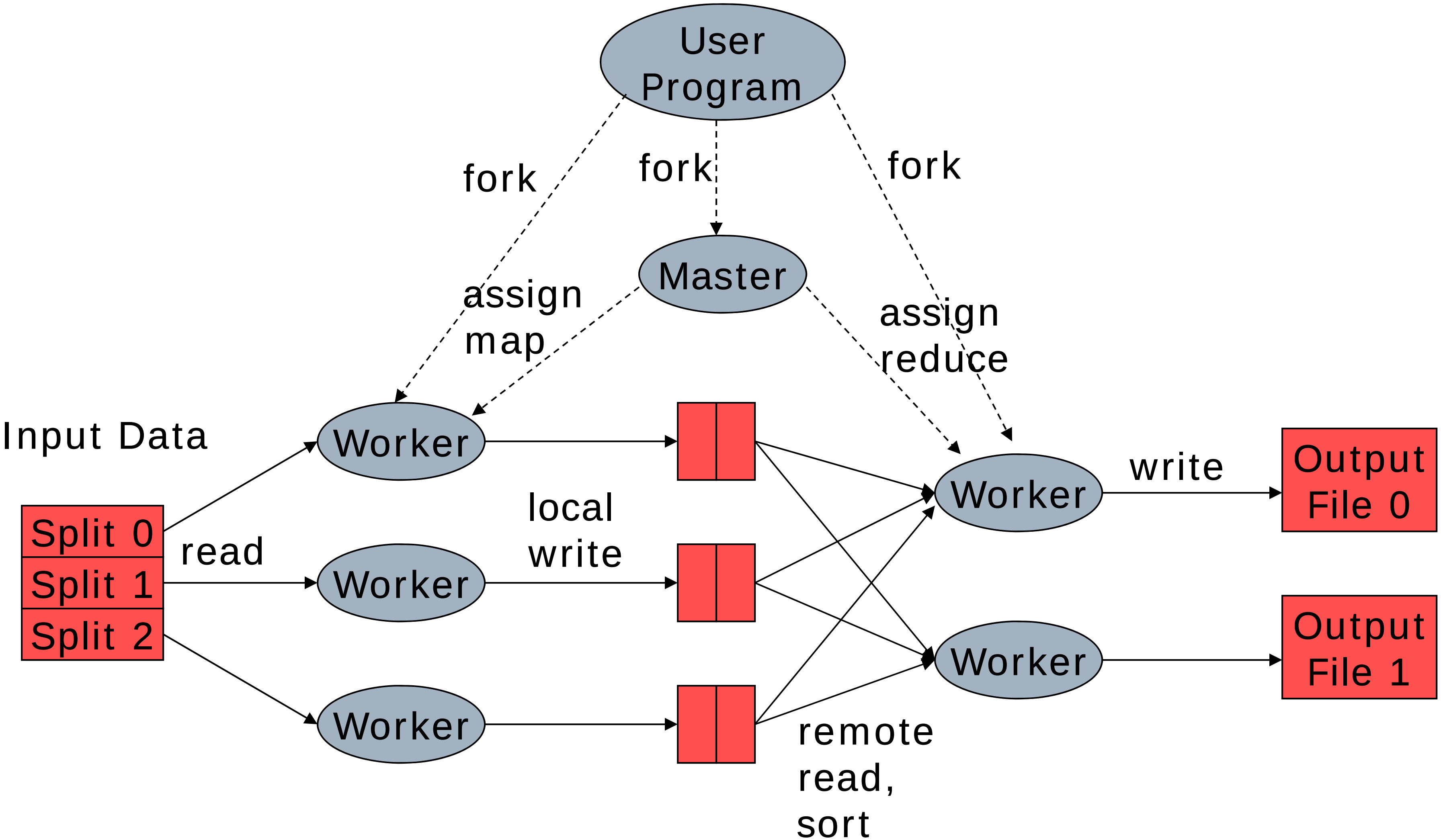
Emit result “(word, sum)”

see bob run
see spot throw

see	1
bob	1
run	1
see	1
spot	1
throw	1

bob	1
run	1
see	2
spot	1
throw	1

Behind the scenes...



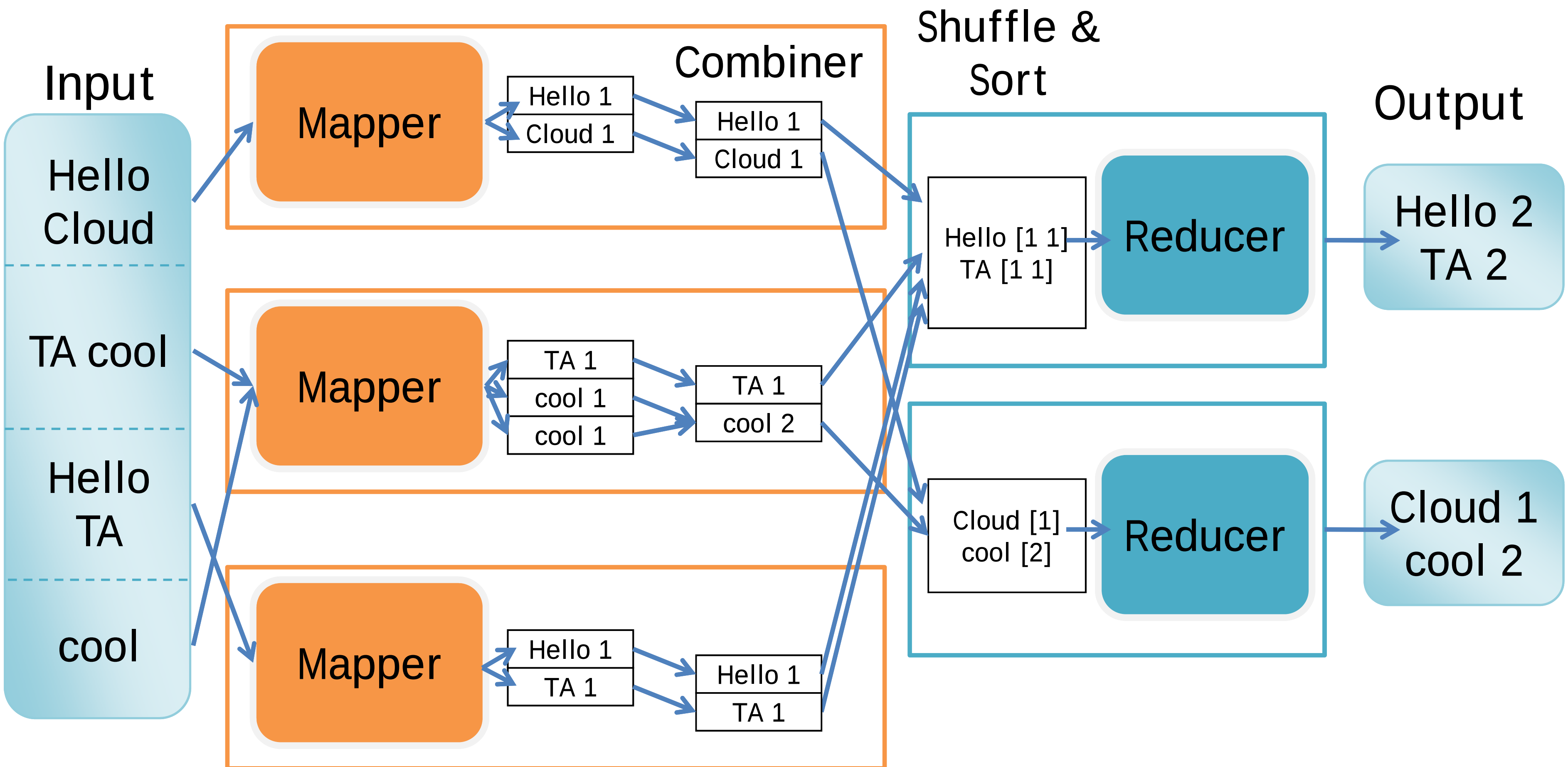
Bandwidth Optimization

- Issue: large number of key-value pairs
- Solution: use “Combiner” functions
  - Executed on same machine as mapper
  - Results in a “mini-reduce” right after the map phase
  - Reduces key-value pairs to save bandwidth

Example: Word Count

```
def map(line_num, line):
    foreach word in line.split():
        output(word, 1)
```

```
def reduce(word, counts):
    output(word, sum(counts))
```



Combiners

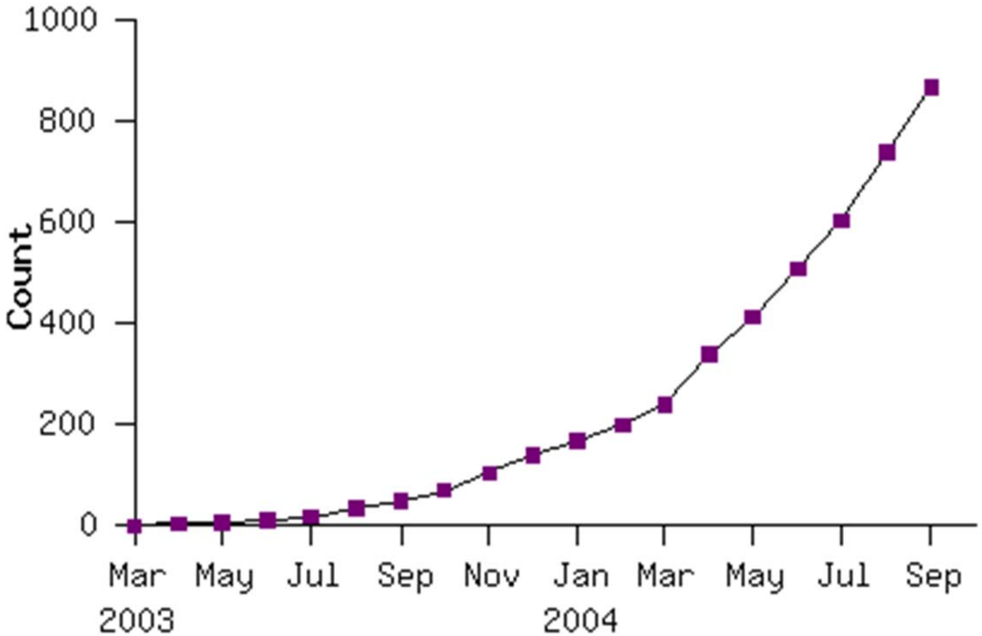
- Often a map task will produce many pairs of the form (k,v1), (k,v2), ... for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) $\rightarrow$ v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

Skew Problem

- Issue: reduce is only as fast as the slowest map
- Solution: redundantly execute map operations, use results of first to finish
  - Addresses hardware problems...
  - But not issues related to inherent distribution of data

Model is Widely Applicable

MapReduce Programs In Google Source Tree



Example uses:

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

Example: Reverse Web-Link Graph

- Find all pages that link to a certain page
- Map Function
  - Outputs <*target*, *source*> pairs for each link to a *target* URL found in a *source* page
  - For each page we know what pages it links to
- Reduce Function
  - Concatenates the list of all *source* URLs associated with a given *target* URL and emits the pair: <*target*, list(*source*)>
  - For a given web page, we know what pages link to it

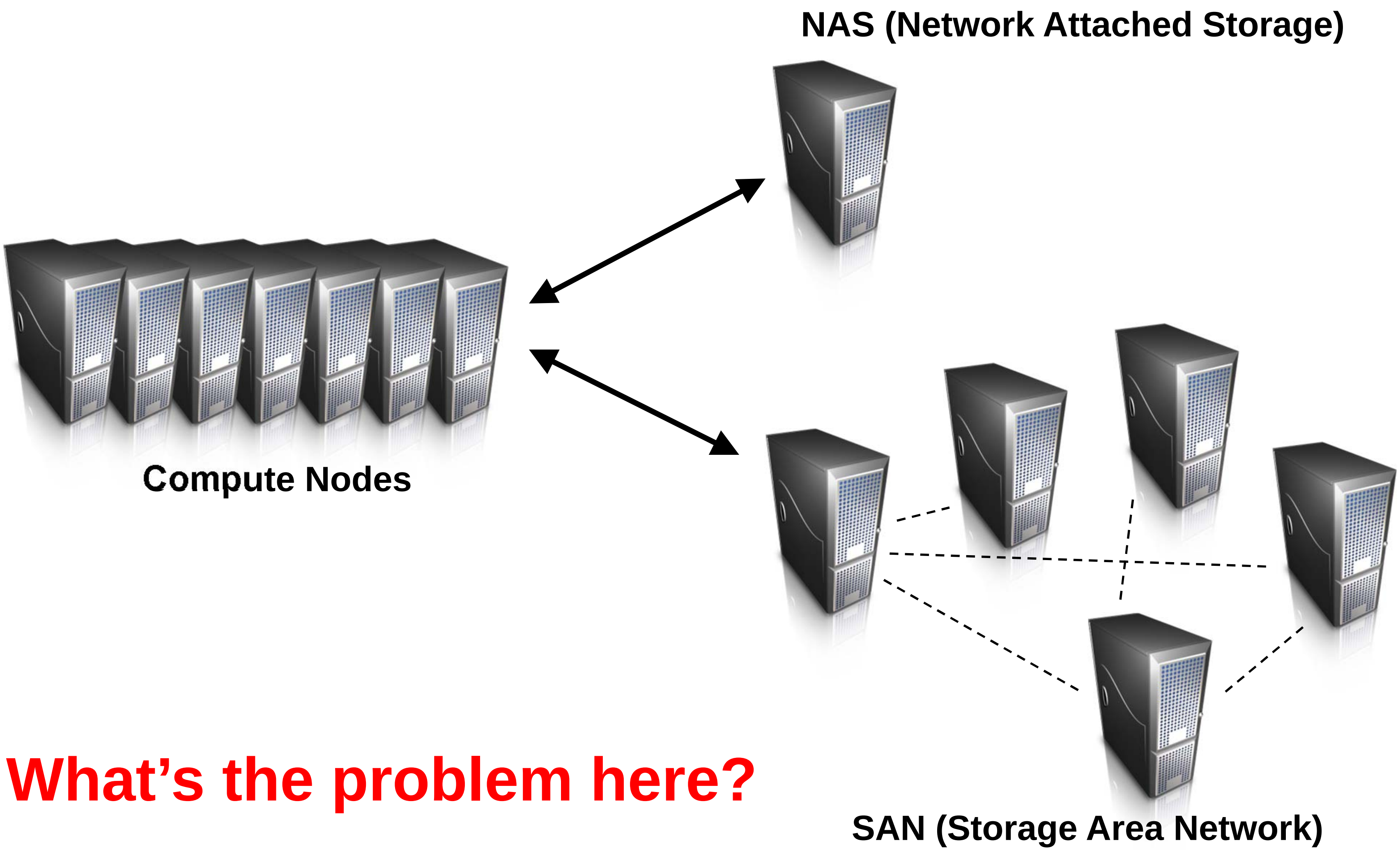
What about the following problems?

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

MapReduce Runtime

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves the process to the data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don't move data to workers… Move workers to the data!
  - Store data on the local disks for nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, disk throughput is good
- A distributed file system is the answer
  - GFS (Google File System)
  - HDFS for Hadoop

Outline

- Introduction to MapReduce
  - Google File System
- The Hadoop Framework

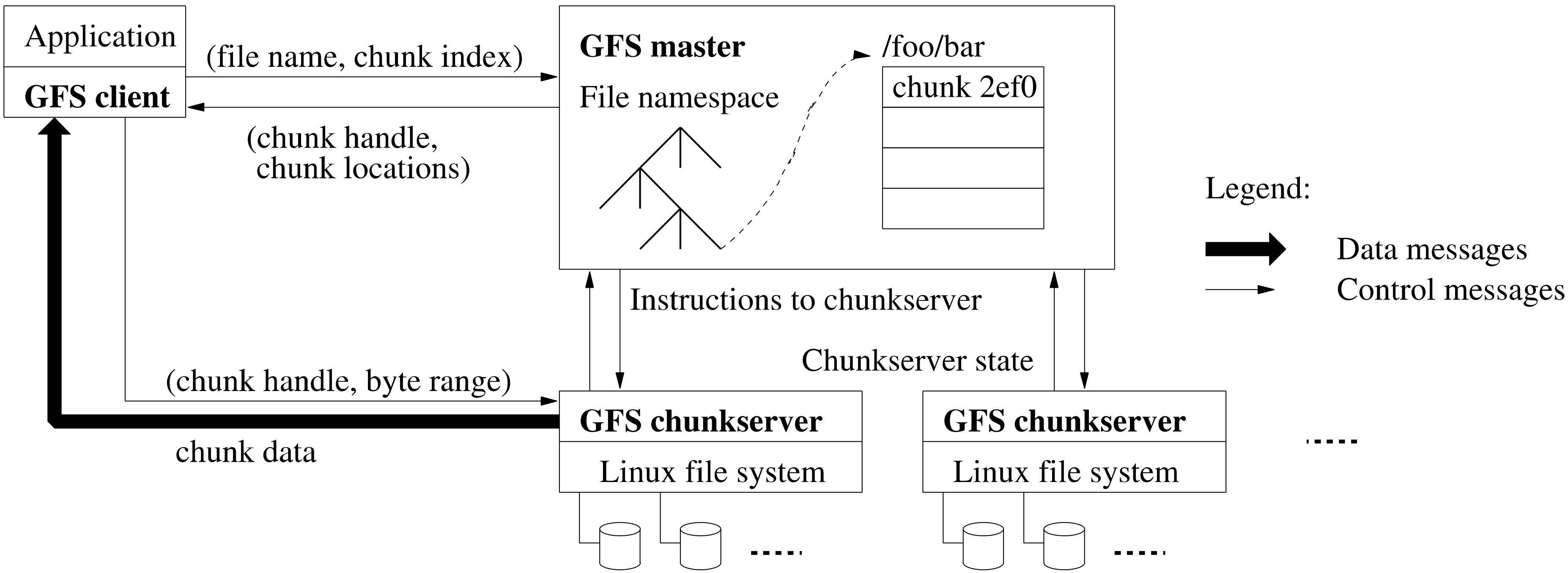
GFS: Assumptions

- Commodity hardware over "exotic" hardware
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of HUGE files
- Files are write-once, mostly appended to
- Perhaps concurrently
- Large streaming reads over random access
- High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large data sets, streaming reads
- Simplify the API
  - Push some of the issues onto the client

GFS Architecture



Source: Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system, SOSP 2003.

Single Master

- We know this is a:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - Shadow masters
  - Minimize master involvement
    - Never move data through it, use only for metadata (and cache metadata at clients)
    - Large chunk size
    - Master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough!

Master's Responsibilities (1/2)

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
  - Give instructions, collect state, track cluster health
- Chunk creation, re-replication, rebalancing
  - Balance space utilization and access speed
  - Spread replicas across racks to reduce correlated failures
  - Re-replicate data if redundancy falls below threshold
  - Rebalance data to smooth out storage and request load

Master's Responsibilities (2/2)

- Garbage Collection
  - Simpler, more reliable than traditional file delete
  - Master logs the deletion, renames the file to a hidden name
  - Lazily garbage collects hidden files
- Stale replica deletion
  - Detect "stale" replicas using chunk version numbers

Metadata

- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk’s replicas
- All in memory (64 Mbytes / chunk)
  - Fast
  - Easily accessible
- Master has an operation log for persistent logging of critical metadata updates
  - Persistent on local disk
  - Replicated
  - Checkpoints for faster recovery

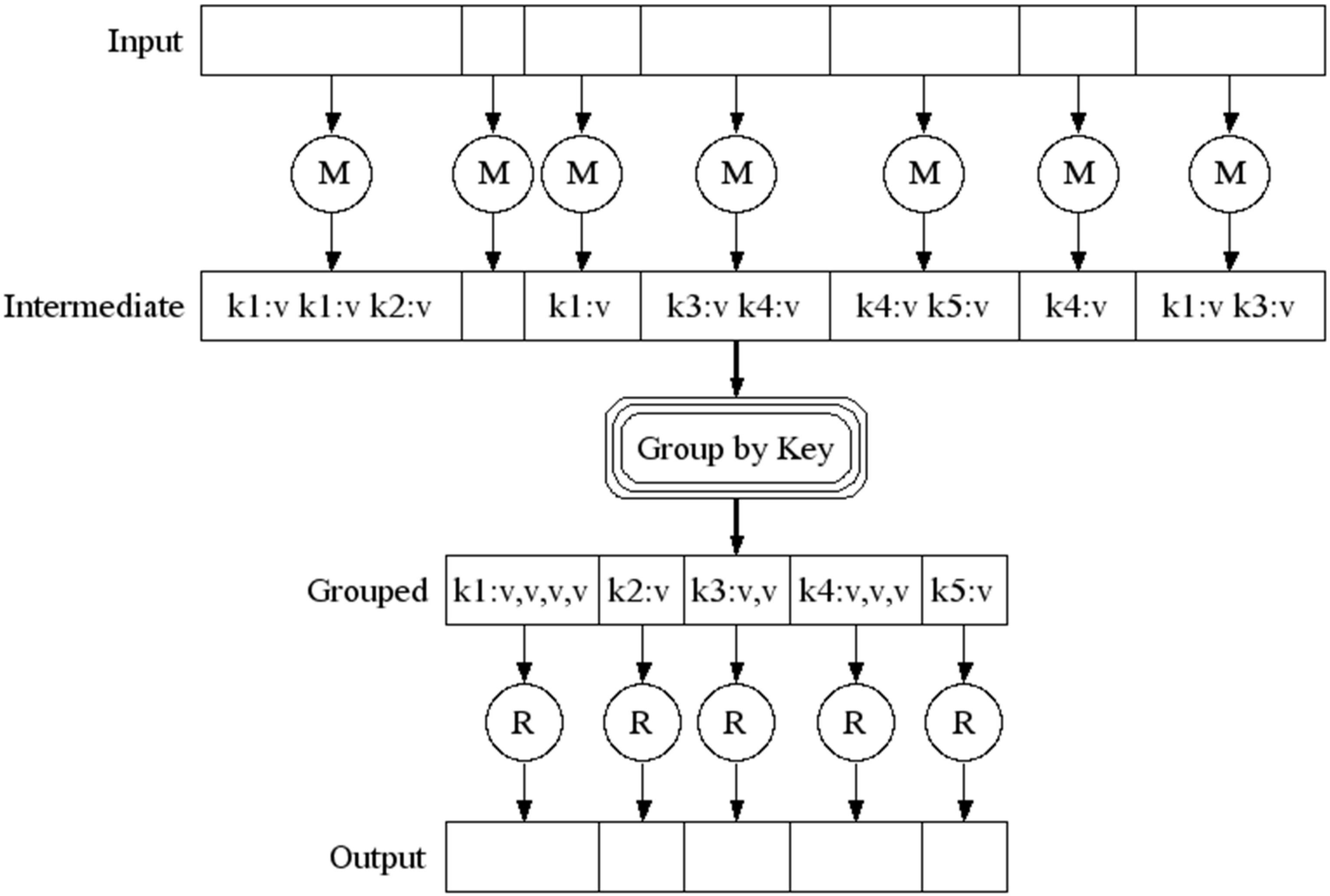
Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

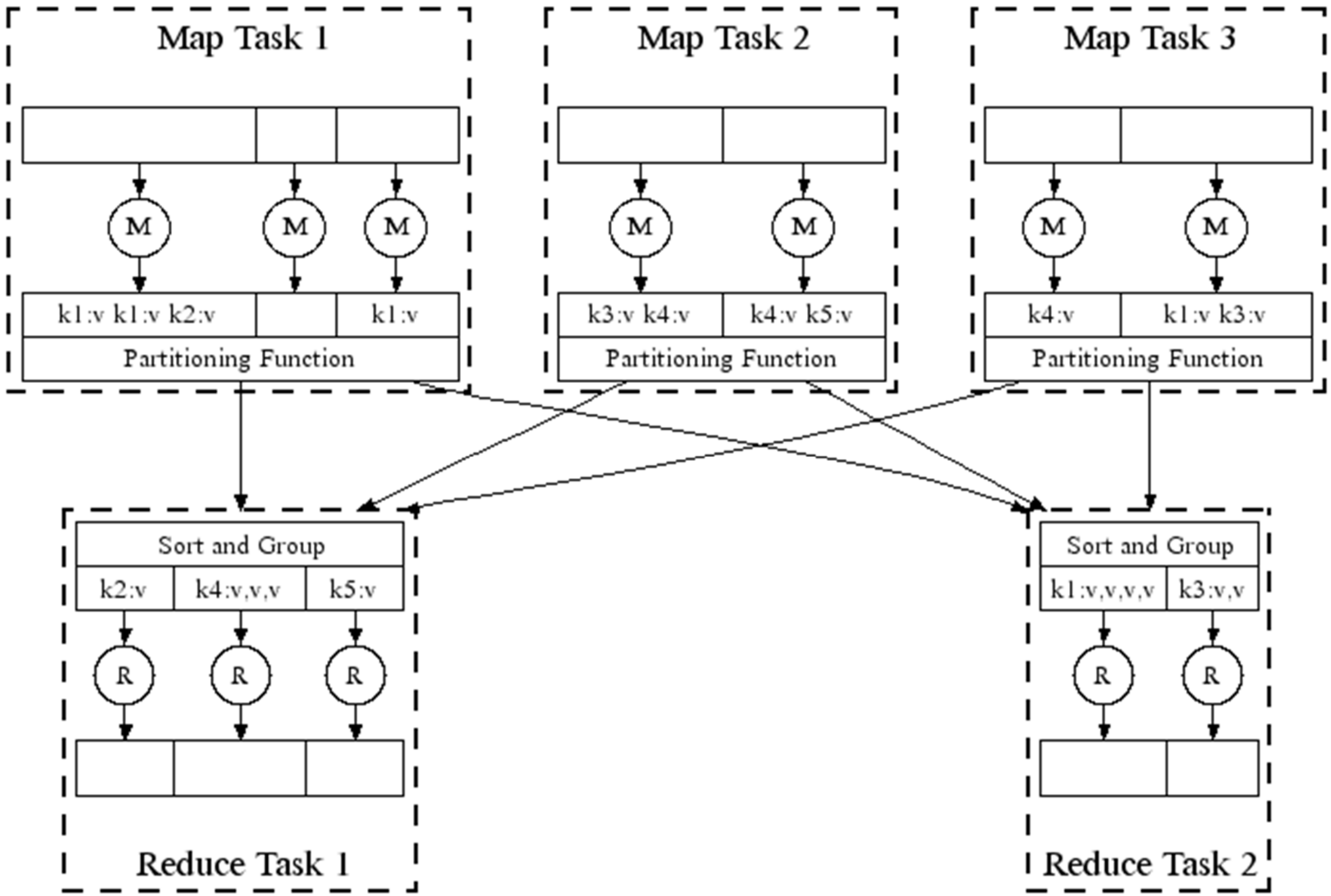
Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

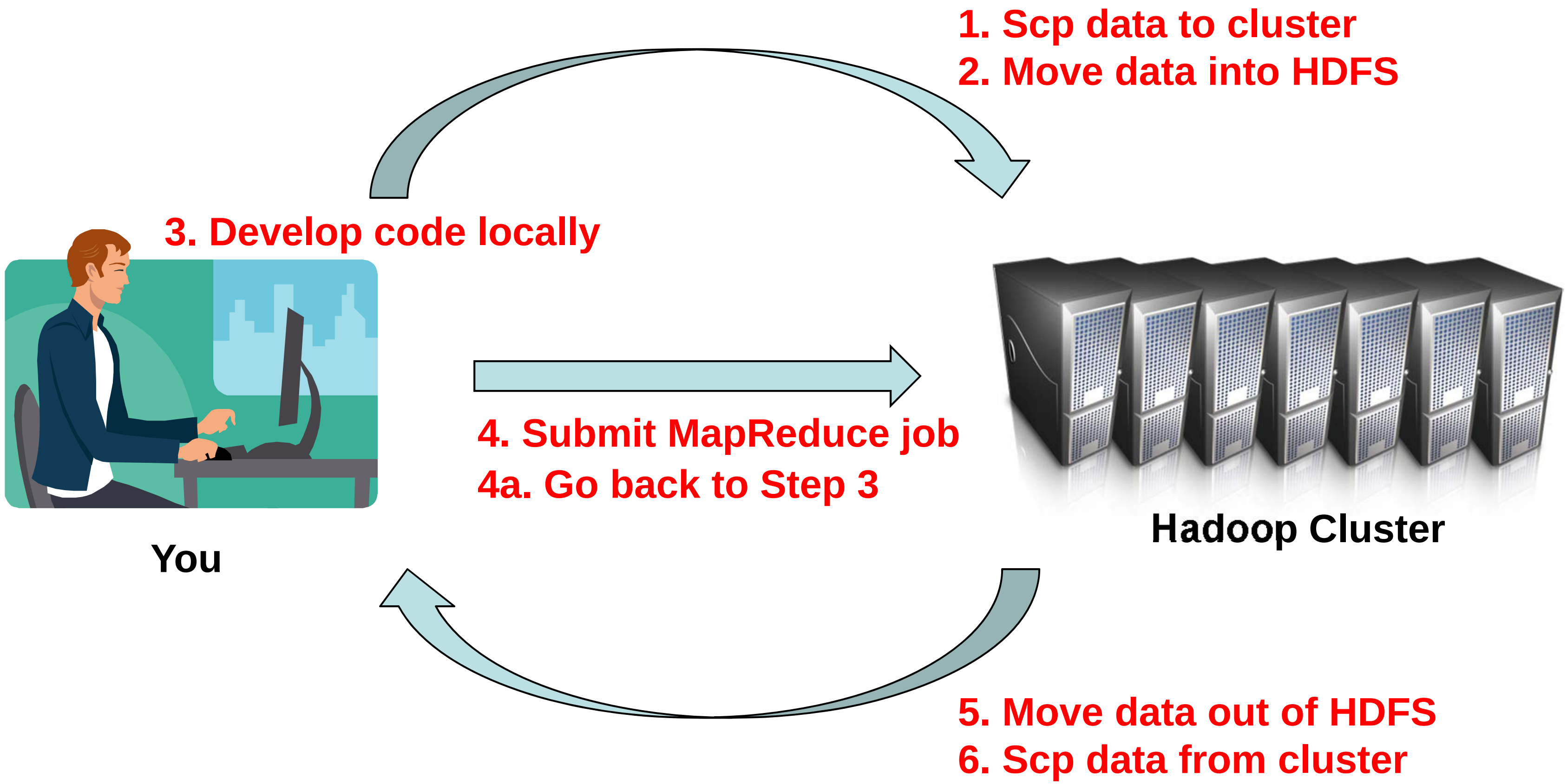
Parallel Execution



How many Map and Reduce jobs?

- M map tasks, R reduce tasks
- Rule of thumb:
  - Make M and R much larger than the number of nodes in cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
- Usually R is smaller than M, because output is spread across R files

From Theory to Practice



Execution Summary

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

Outline

- Introduction to MapReduce
  - Google File System
- The Hadoop Framework

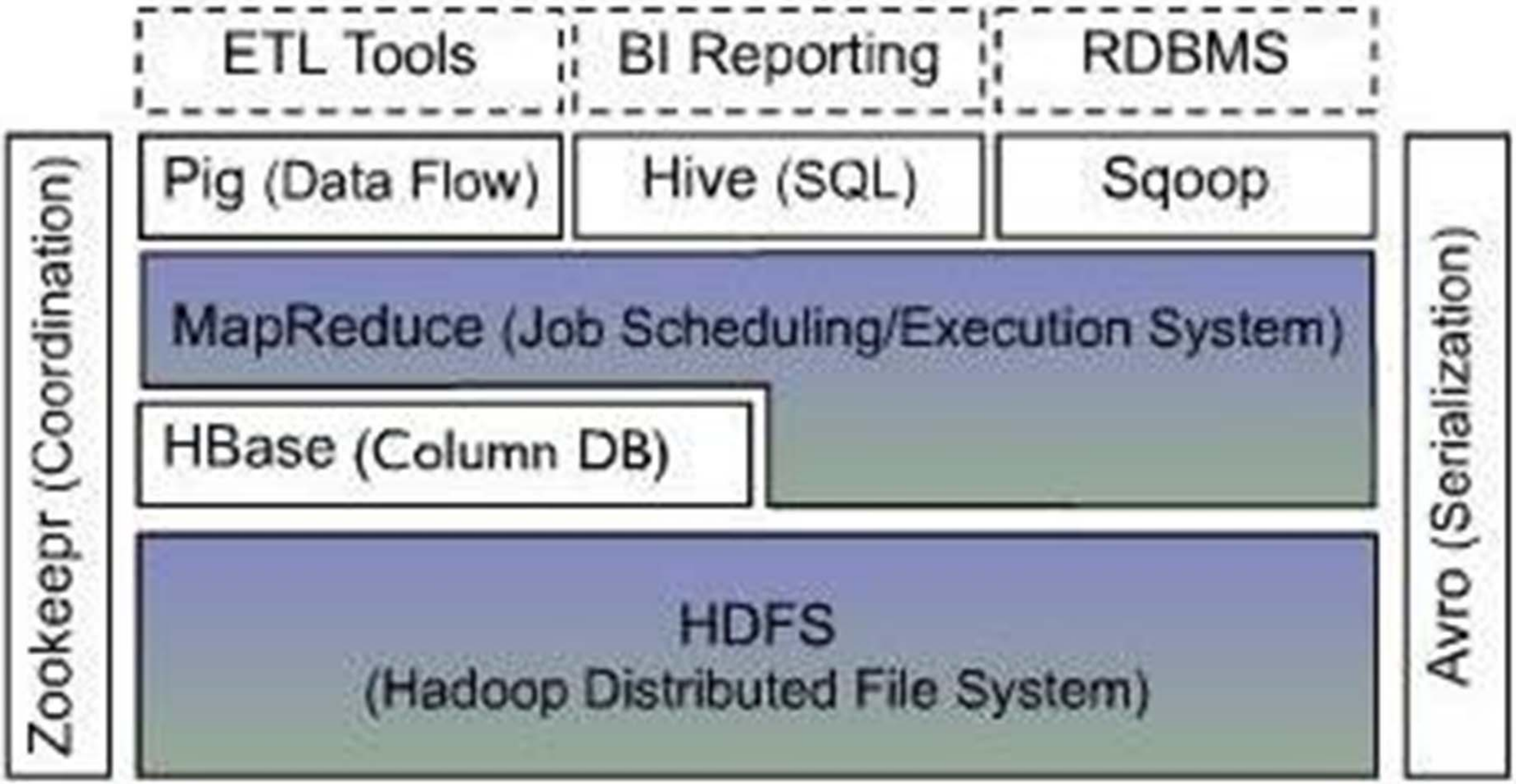
Hadoop

- Hadoop (High-availability distributed object-oriented platform)
- An open-source software framework that supports data-intensive distributed applications
- Distributed File System (HDFS)
  - Runs on same machines
  - Replicates data 3x for fault-tolerance
- It supports the running of applications on large clusters of commodity hardware

Hadoop

- Hadoop was derived from Google's MapReduce and Google File System (GFS) papers
- Hadoop is written in the Java programming language
  - Download from: http://hadoop.apache.org/

Hadoop Ecosystem



Source: Cloudera

Terminology

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

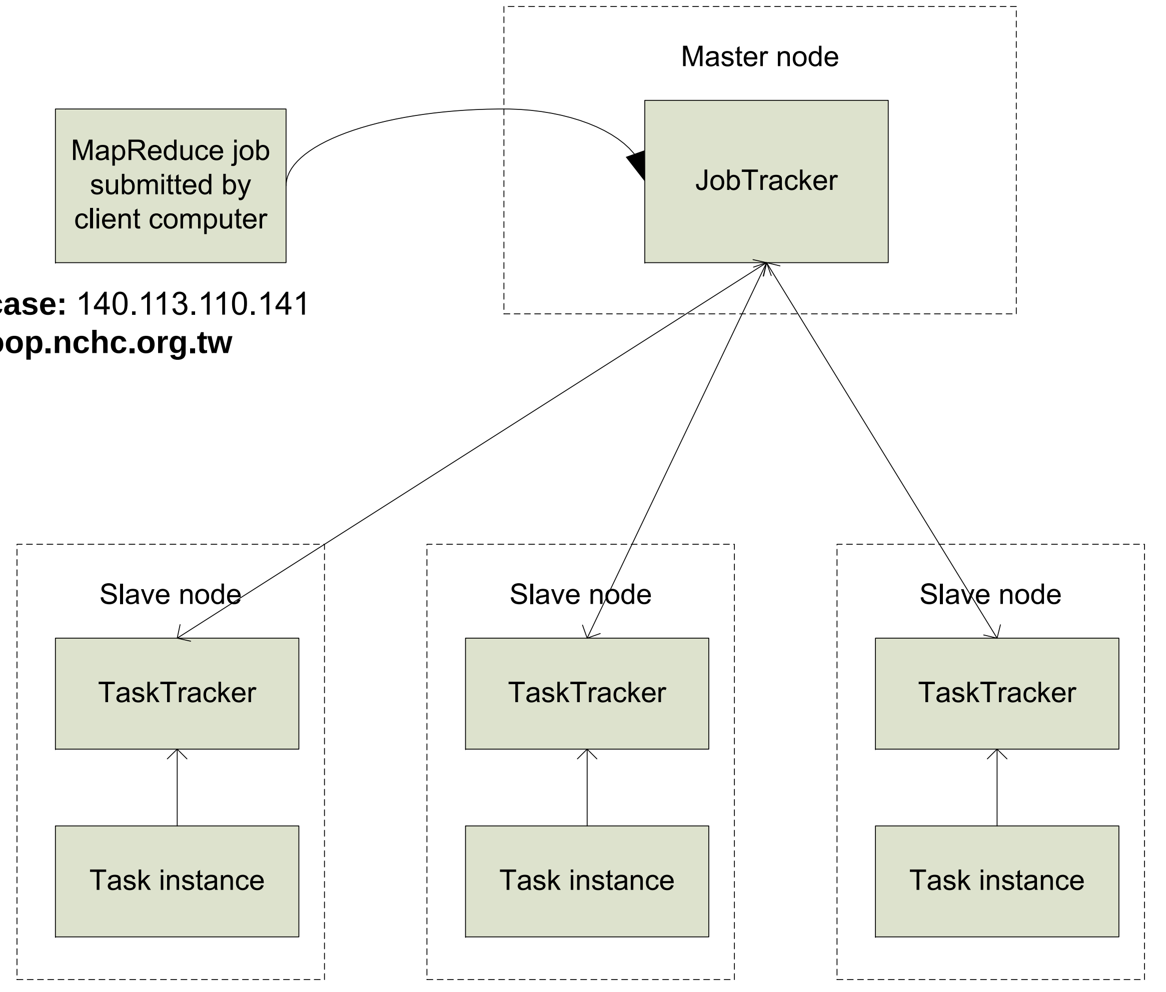
Some MapReduce Terminology

- *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



In our case: 140.113.110.141
or hadoop.nchc.org.tw

Nodes, Trackers, Tasks

- Master node runs *JobTracker* instance, which accepts *Job* requests from clients

- *TaskTracker* instances run on slave nodes

- TaskTracker forks separate Java processes 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

- … Where’s the data distribution?

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

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

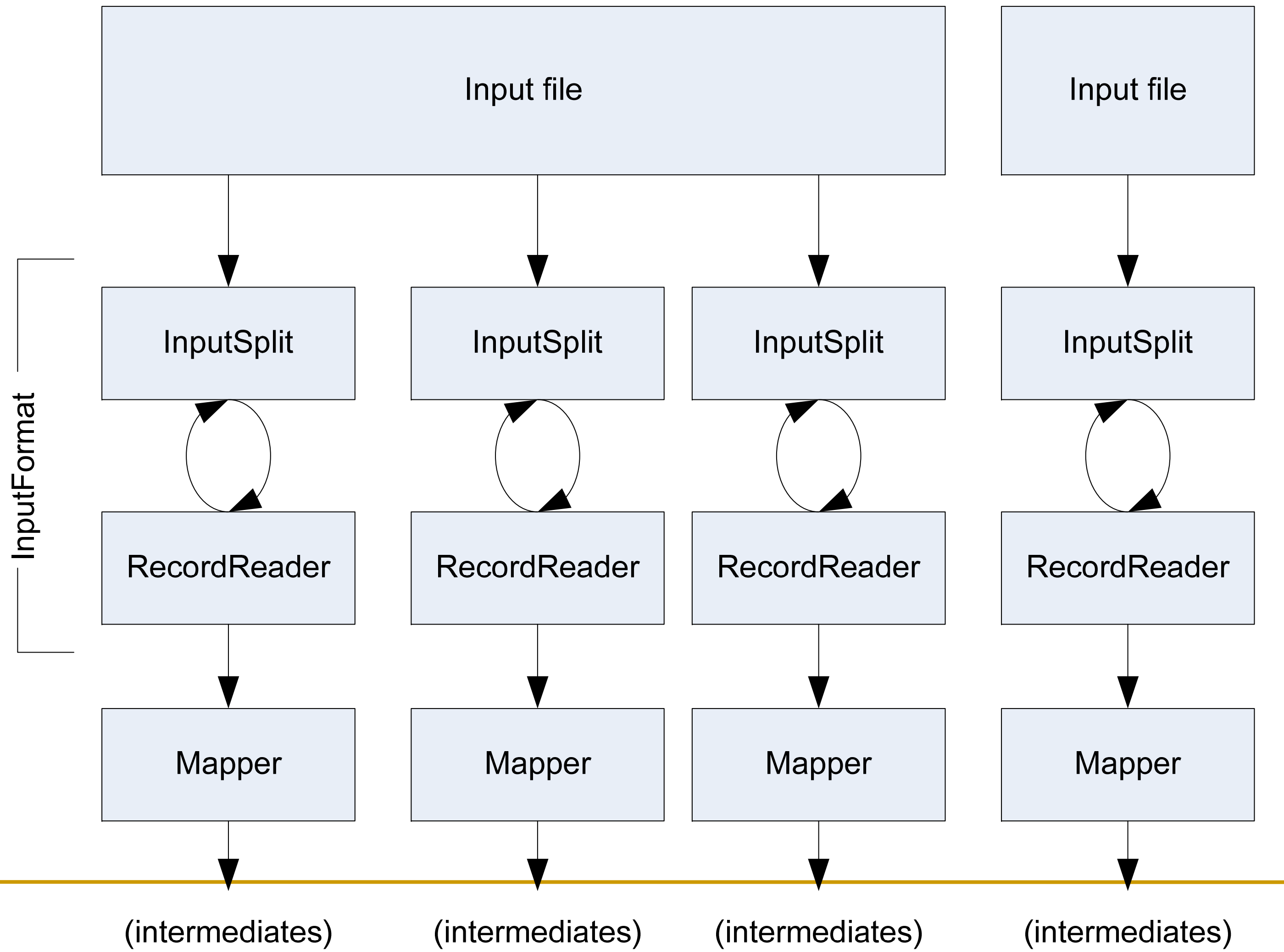
- void map(K1 key,
  V1 value,
  OutputCollector<K2, V2> output,
  Reporter reporter)

- *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

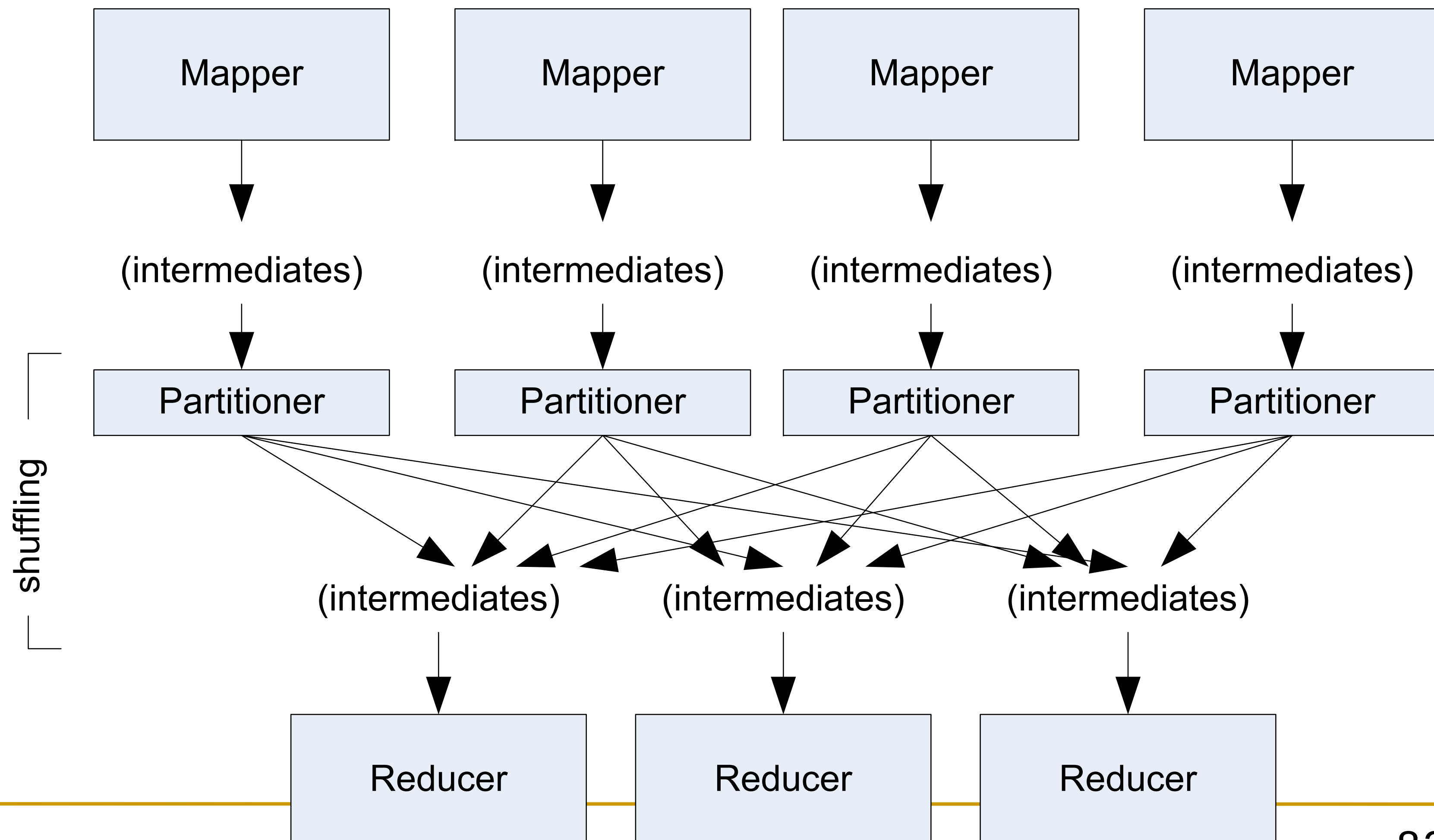
WritableComparator

- 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



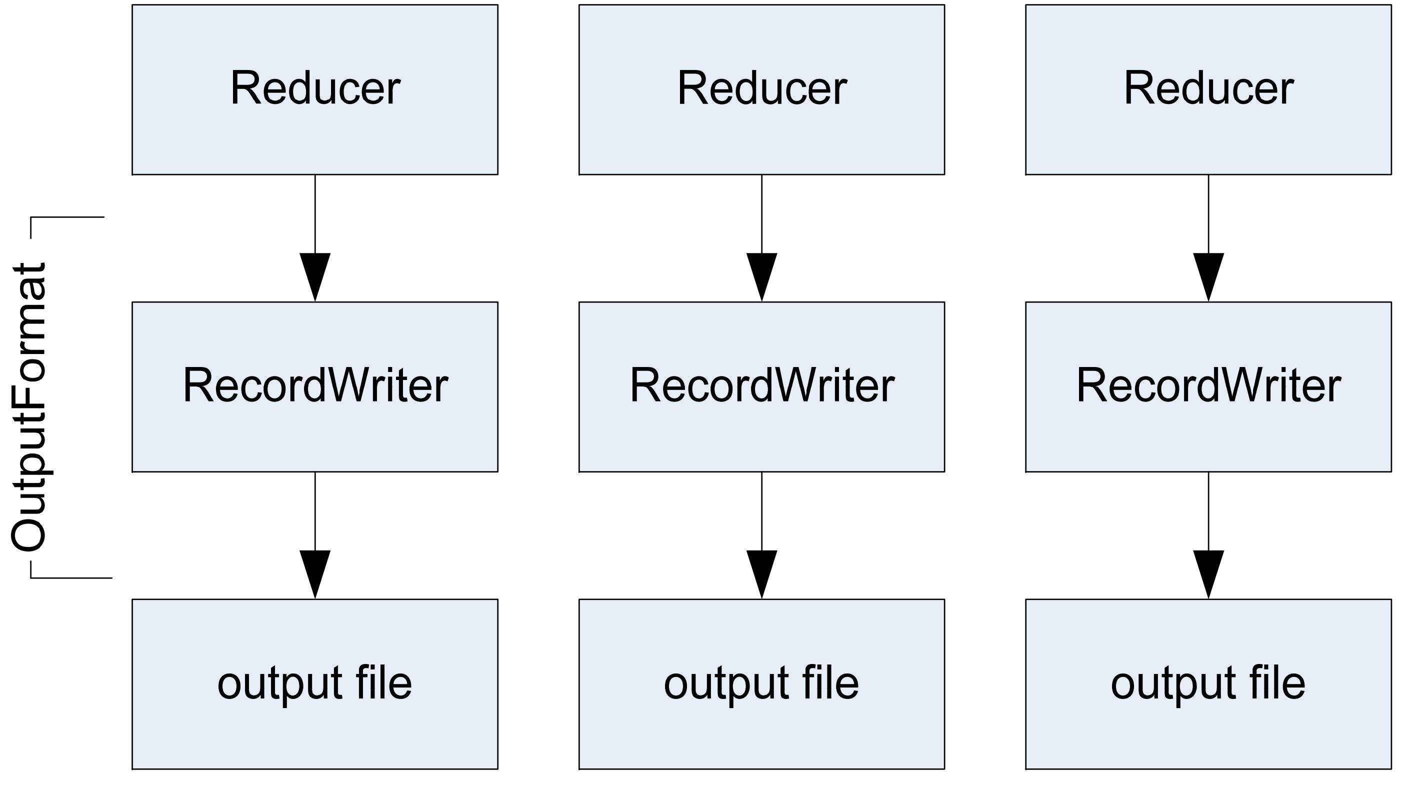
Partitioner

- 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()

Methods to write MapReduce Jobs

- Typical – usually written in Java
  - MapReduce 2.0 API
  - MapReduce 1.0 API
- Pipes
  - Often used with C++
- Streaming
  - Uses stdin and stdout
  - Can use any language to write Map and Reduce functions
- Abstraction libraries
  - Hive, Pig, etc.
  - Written in a higher level language, generate one or more MapReduce jobs

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));
    }
}
}
```

Hadoop Pipes: wordcount – main()

- Allows you to create map/reduce jobs in C++

```
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

int main(int argc, char *argv[]) {
  return
HadoopPipes::runTask(HadoopPipes::TemplateFactory<WordCountMap,
                          WordCountReduce>());
}
```

Hadoop Pipes: wordcount – map()

- Allows you to create map/reduce jobs in C++

```
class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};
```

Hadoop Pipes: wordcount – reduce()

- Allows you to create map/reduce jobs in C++

```
class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::toString(sum));
  }
};
```

Running with Hadoop Pipes

- Allows you to create map/reduce jobs in C++

```
g++ wordcount.cpp -o wordcount -lhadooputils \
-lhadooppipes -lpthread
```

```
hadoop pipes -D hadoop.pipes.java.recordreader=true \
-D hadoop.pipes.java.recordwriter=true \
-program examples/bin/wordcount \
-input examples/input -output examples/output
```

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 -w

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
        -input myInputDirs \
        -output myOutputDir \
        -mapper /bin/cat \
        -reducer /bin/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 C++/Python example of a MapReduce word count program…

wordcount_map.cpp

```
#include<iostream>
using namespace std;

int main()
{
    string word;
    while (cin >> word)
    {
        cout << word << "\t" << "1" << endl;
    }
    return 0;
}
```

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

wordcount_reduce.cpp

```
#include<vector>
#include<map>
#include<string>
#include<iostream>
using namespace std;
int main() {
    string key;
    string value;
    map<string,int> mapTemp;
    while(cin >> key >> value) {
        if(mapTemp.find(key) != mapTemp.end())
            mapTemp[key] += 1;
        else
            mapTemp[key] = 1;
    }
    map<string,int>::iterator it = mapTemp.begin();
    for(; it != mapTemp.end(); ++it) {
        cout << it->first << "\t" << it->second << endl;
    }

    return 0;
}
```

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 \

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

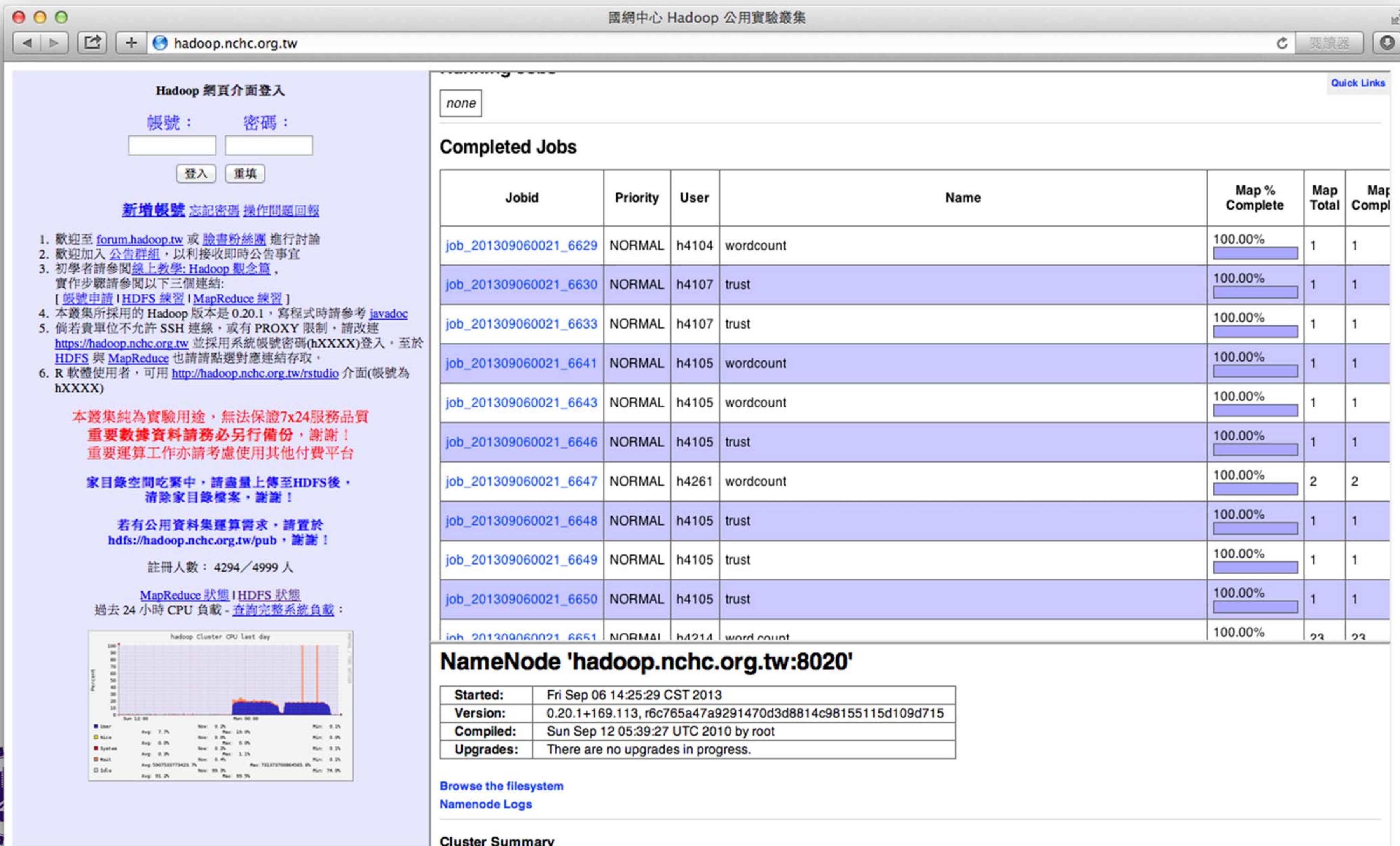
Further Resources

- 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

Public Hadoop Cluster

- http://hadoop.nchc.org.tw/

NARLabs 國家實驗研究院
國家高速網路與計算中心



Reference

- Running Hadoop on Ubuntu Linux (Single-Node Cluster)
  - http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/

Conclusions

- MapReduce proven to be useful abstraction
- Greatly simplifies large-scale computations
- Fun to use:
  - Focus on problem
  - Let library deal with messy details