# Google Cluster Computing Faculty Training Workshop

## Module V: Hadoop Technical Review



# Overview

- Hadoop Technical Walkthrough
- HDFS
- Databases
- Using Hadoop in an Academic Environment
- Performance tips and other tools



# You Say, “tomato…”

| Google calls it: | Hadoop equivalent: |
| --- | --- |
| MapReduce | Hadoop |
| GFS | HDFS |
| Bigtable | HBase |
| Chubby | (nothing yet… but planned) |

# 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



# Terminology Example

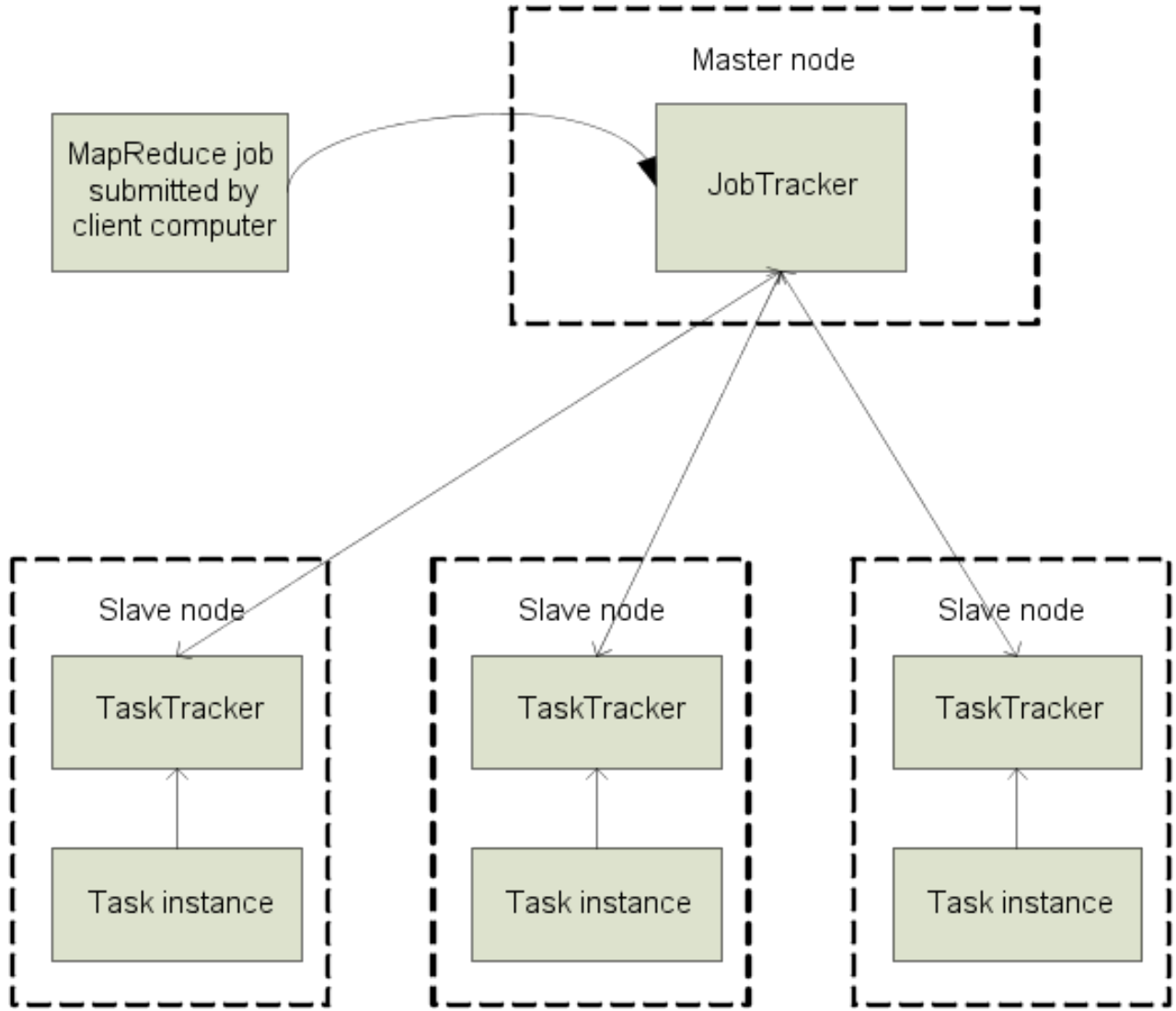
- Running “Word Count” across 20 files is one *job*
- 20 files to be mapped imply 20 *map tasks* + some number of *reduce tasks*
- At least 20 map *task attempts* will be performed… more if a machine crashes, etc.

# 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






# Node-to-Node Communication

- Hadoop uses its own RPC protocol
- All communication begins in slave nodes
  - Prevents circular-wait deadlock
  - Slaves periodically poll for “status” message
- Classes must provide explicit serialization



# 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

- … 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

# Configuring With JobConf

- MR Programs have many configurable options
- *JobConf* objects hold (key, value) components mapping String • ’a
  - e.g., “mapred.map.tasks” • 20
  - JobConf is serialized and distributed before running the job
- Objects implementing *JobConfigurable* can retrieve elements from a JobConf

# What Happens In MapReduce? Depth First



# Job Launch Process: Client

- Client program creates a *JobConf*
  - Identify classes implementing *Mapper* and *Reducer* interfaces
    - JobConf.setMapperClass(), setReducerClass()
  - Specify inputs, outputs
    - JobConf.setInputPath(), 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(WritableComparable key,
  Writable value,
  OutputCollector output,
  Reporter reporter)



# 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*



# Writing For Cache Coherency

```
while (more input exists) {
myIntermediate = new intermediate(input);
myIntermediate.process();
export outputs;
}
```



# Writing For Cache Coherency

```
myIntermediate = new intermediate (junk);
while (more input exists) {
myIntermediate.setupState(input);
myIntermediate.process();
export outputs;
}
```

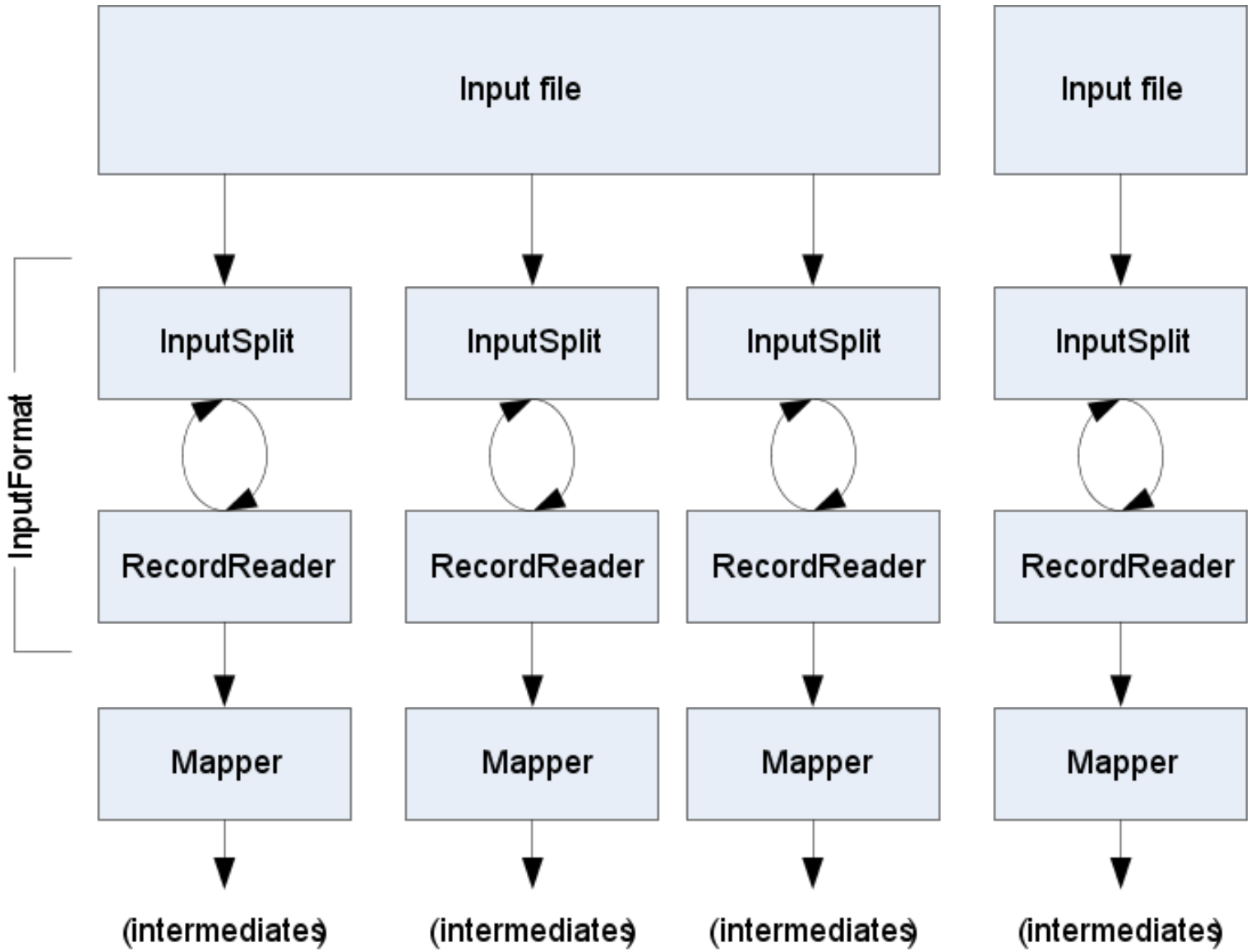


# Writing For Cache Coherency

- Running the GC takes time
- Reusing locations allows better cache usage
- Speedup can be as much as two-fold
- All serializable types must be *Writable* anyway, so make use of the interface



# 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

# *WritableComparator*

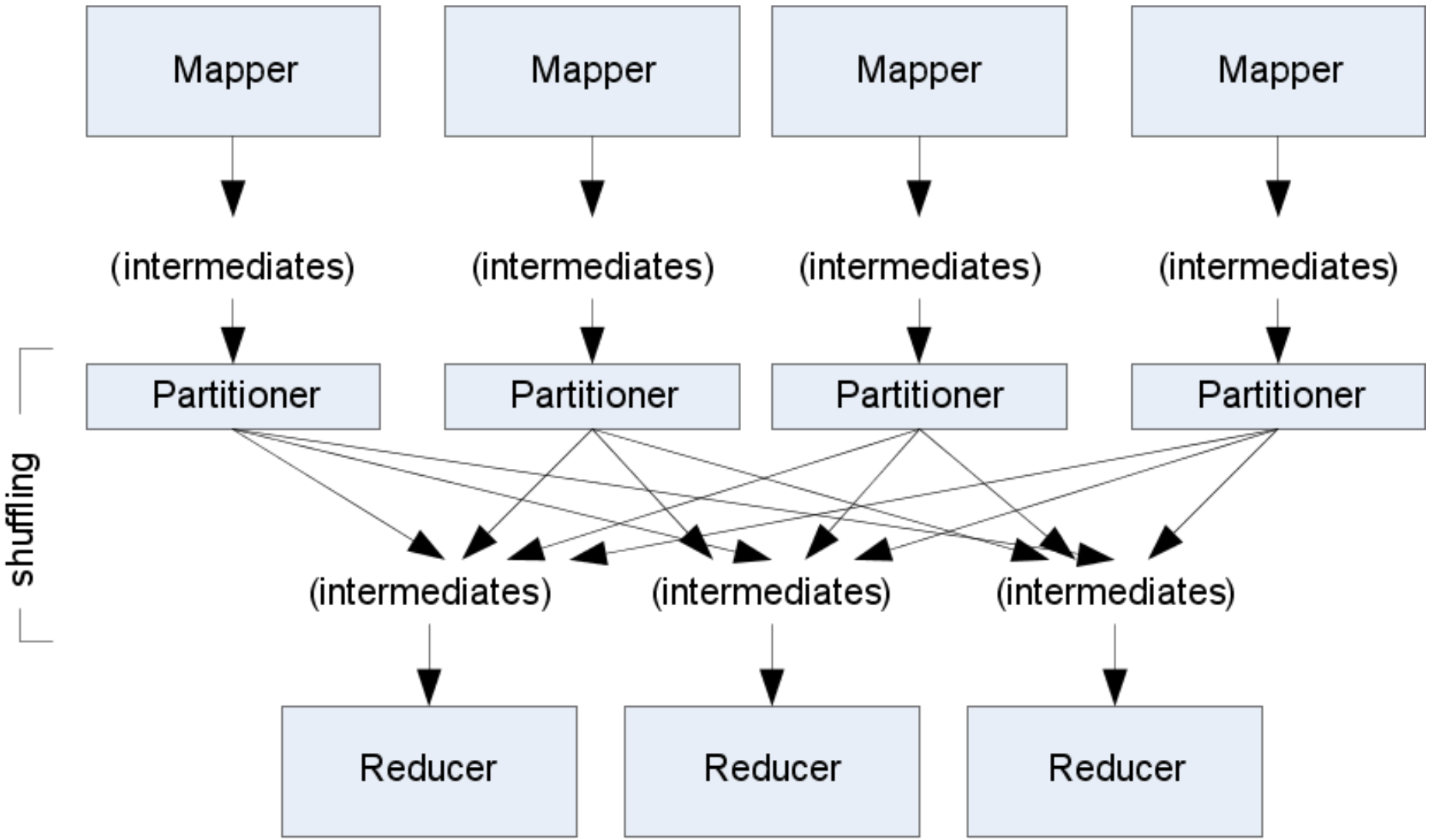
- Compares WritableComparable data
  - Will call WritableComparable.compare()
  - Can provide fast path for serialized data
- *JobConf.setOutputValueGroupingComparator()*



# 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*

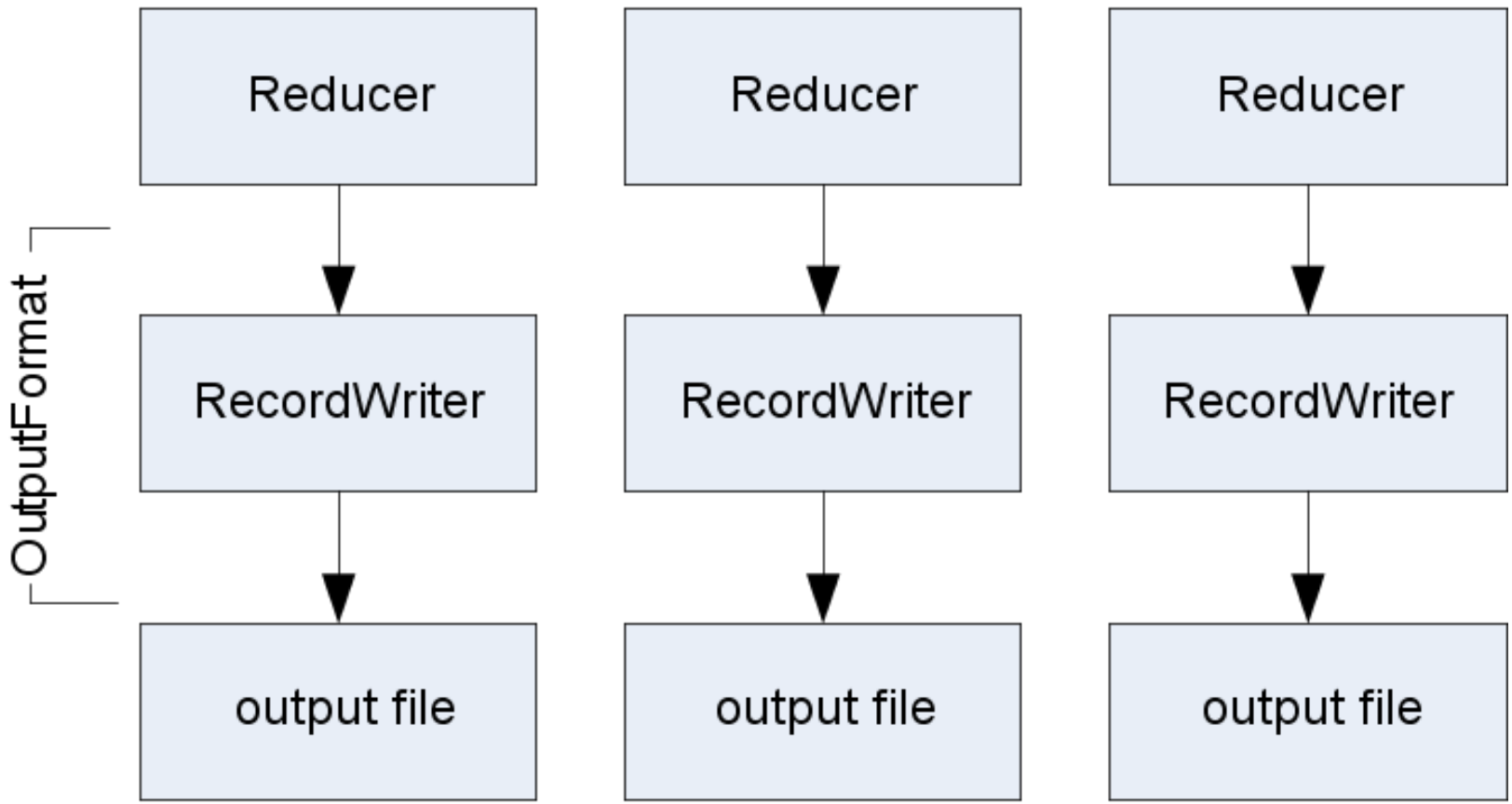
- int getPartition(key, val, numPartitions)
  - Outputs the partition number for a given key
  - One partition == values sent to one Reduce task
- *HashPartitioner* used by default
  - Uses key.hashCode() to return partition num
- *JobConf* sets *Partitioner* implementation

# Reduction

- reduce( WritableComparable key,
  Iterator values,
  OutputCollector 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



# HDFS



# HDFS Limitations

- "Almost" GFS
  - No file update options (record append, etc); all files are write-once
- Does not implement demand replication
- Designed for streaming
  - Random seeks devastate performance



# NameNode

- “Head” interface to HDFS cluster
- Records all global metadata



# Secondary NameNode

- Not a failover NameNode!
- Records metadata snapshots from “real” NameNode
  - Can merge update logs in flight
  - Can upload snapshot back to primary



# NameNode Death

- No new requests can be served while NameNode is down
  - Secondary *will not* fail over as new primary

- So why have a secondary at all?



# NameNode Death, cont'd

- If NameNode dies from software glitch, just reboot
- But if machine is hosed, metadata for cluster is irretrievable!



# Bringing the Cluster Back

- If original NameNode can be restored, secondary can re-establish the most current metadata snapshot
- If not, create a new NameNode, use secondary to copy metadata to new primary, restart whole cluster ( ☹ )
- Is there another way…?

# Keeping the Cluster Up

- Problem: DataNodes “fix” the address of the NameNode in memory, can’t switch in flight
- Solution: Bring new NameNode up, but use DNS to make cluster believe it’s the original one
  - Secondary can be the “new” one



# Further Reliability Measures

- Namenode can output multiple copies of metadata files to different directories
  - Including an NFS mounted one
  - May degrade performance; watch for NFS locks



# Databases



# Life After GFS

- Straight GFS files are not the only storage option
- HBase (on top of GFS) provides column-oriented storage
- mySQL and other db engines still relevant



# HBase

- Can interface directly with Hadoop
- Provides its own Input- and OutputFormat classes; sends rows directly to mapper, receives new rows from reducer

- … But might not be ready for classroom use (least stable component)

# MySQL Clustering

- MySQL database can be sharded on multiple servers
  - For fast IO, use same machines as Hadoop
- Tables can be split across machines by row key range
  - Multiple replicas can serve same table



# Sharding & Hadoop *Partitioners*

- For best performance, Reducer should go straight to local mysql instance
  - Get all data in the right machine in one copy
- Implement custom Partitioner to ensure particular key range goes to mysql-aware Reducer



# Academic Hadoop Requirements



# Server Profile

- UW cluster:
  - 40 nodes, 80 processors total
  - 2 GB ram / processor
  - 24 TB raw storage space (8 TB replicated)

- One node reserved for JobTracker/NameNode
- Two more wouldn't cooperate
- … But still vastly overpowered



# Setup & Maintenance

- Took about two days to setup and configure
  - Mostly hardware-related issues
  - Hadoop setup was only a couple hours
- Maintenance: only a few hours/week
  - Mostly rebooting the cluster when jobs got stuck

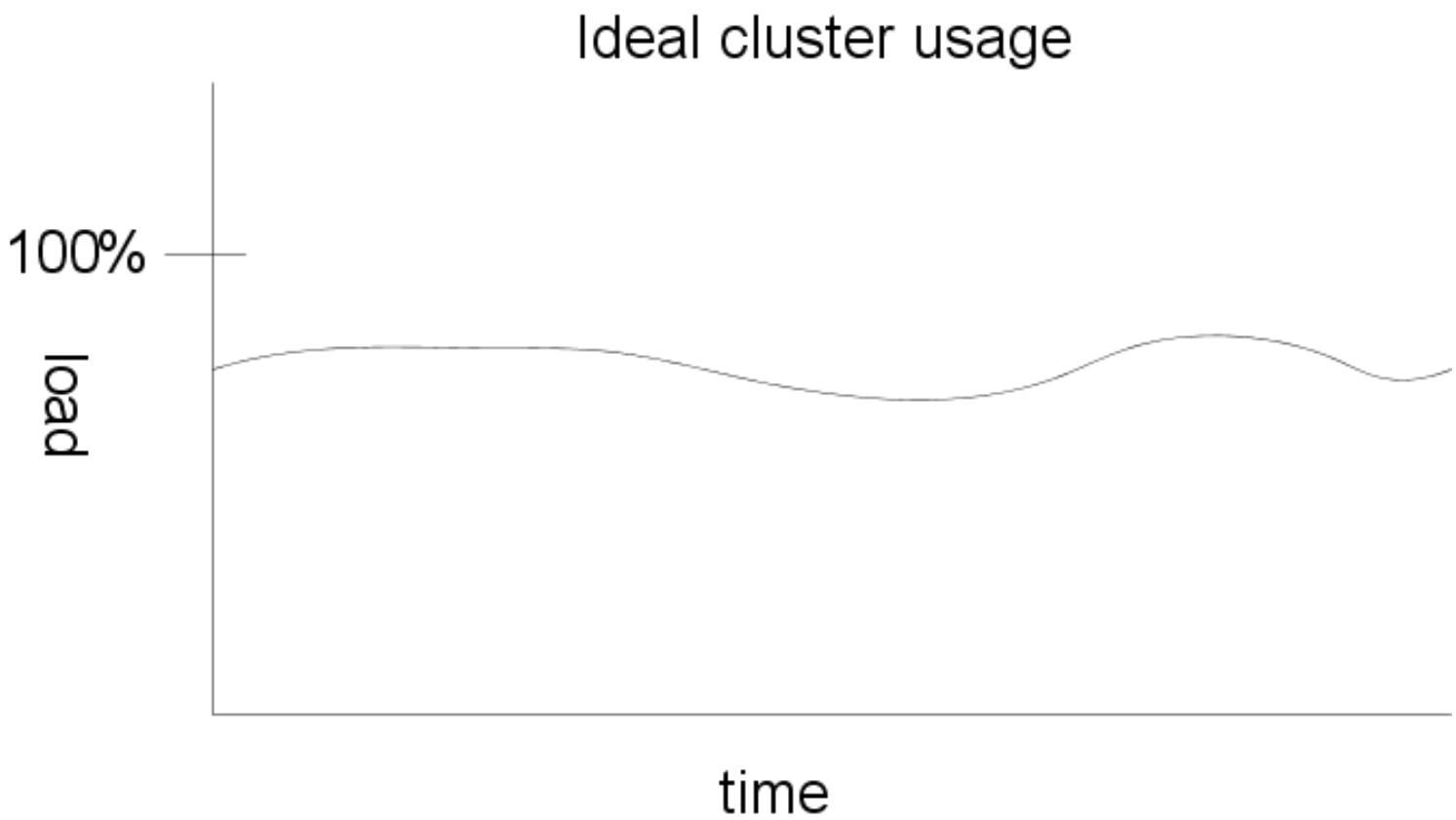
# Total Usage

- About 15,000 CPU-hours consumed by 20 students
- … Out of 130,000 available over quarter

- Average load is about 12%



# Analyzing student usage patterns



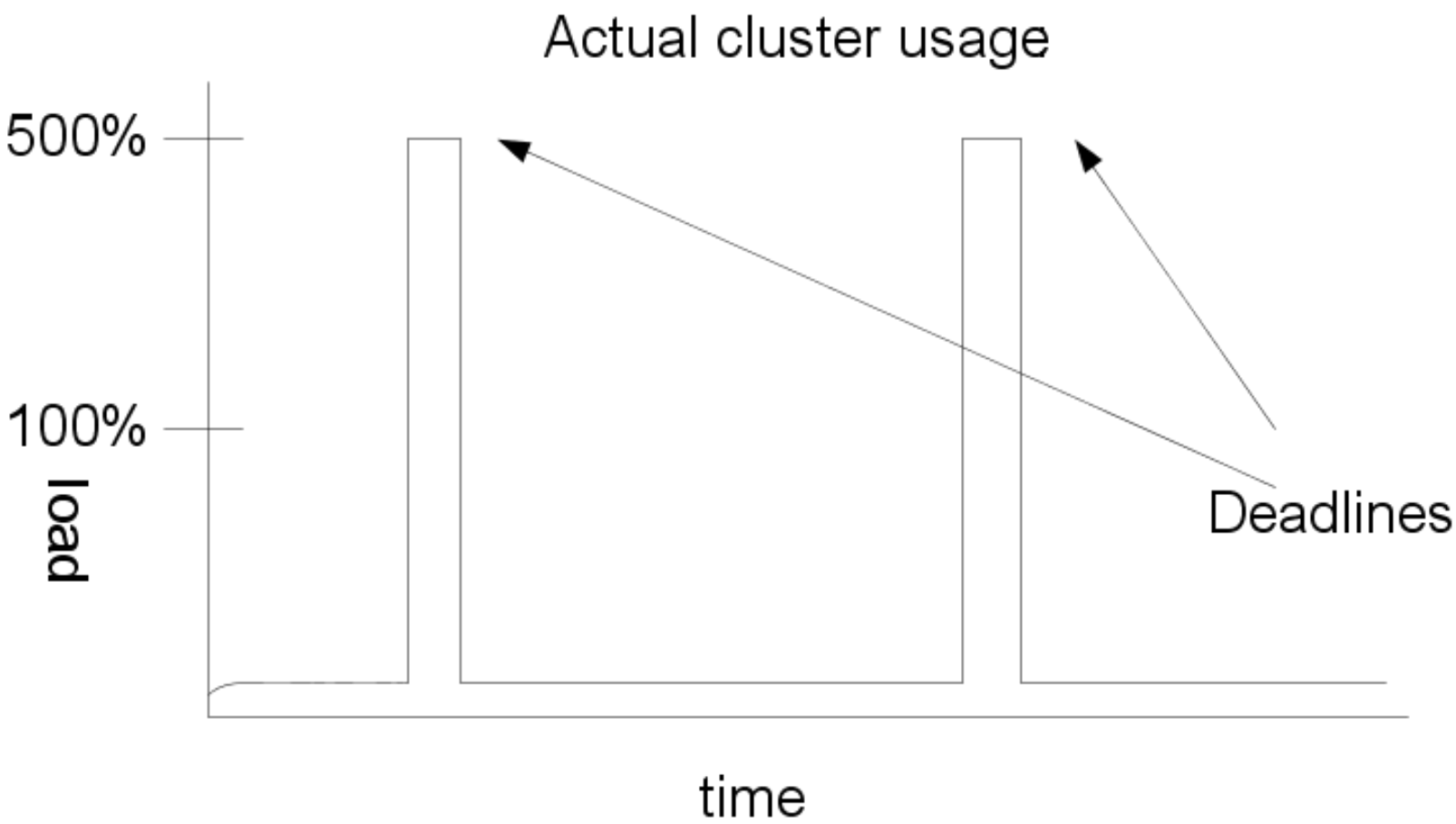



# Not Quite the Whole Story

- Realistically, students did most work very close to deadline
  - Cluster sat unused for a few days, followed by overloading for two days straight



# Analyzing student usage patterns




Lesson: Resource demands are NOT constant!



# Hadoop Job Scheduling

- FIFO queue matches incoming jobs to available nodes
  - No notion of fairness
  - Never switches out running job
  - Run-away tasks could starve other student jobs



# Hadoop Security

- But on the bright (?) side:
  - No security system for jobs
  - Anyone can start a job; but they can also cancel other jobs
- Realistically, students did not cancel other student jobs, even when they should



# Hadoop Security: The Dark Side

- No permissions in HDFS either
  - Just now added in 0.16
- One student deleted the common data set for a project
  - Email subject: “Oops…”
  - No students could test their code until data set restored from backup



# Job Scheduling Lessons

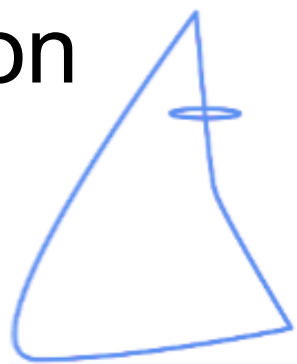
- Getting students to “play nice” is hard
  - No incentive
  - Just plain bad/buggy code
- Cluster contention caused problems at deadlines
  - Work in groups
  - Stagger deadlines



# Another Possibility

- Amazon EC2 provides on-demand servers
- May be able to have students use these for jobs
  - “Lab fee” would be ~$150/student
- Simple web-based interfaces exist
  - Rightscale.com
- HadoopOnDemand (HOD) coming soon
  - Injects new nodes into live clusters





# More Performance & Scalability



# Number of Tasks

- Mappers = 10 * nodes (or 3/2 * cores)
- Reducers = 2 * nodes (or 1.05 * cores)

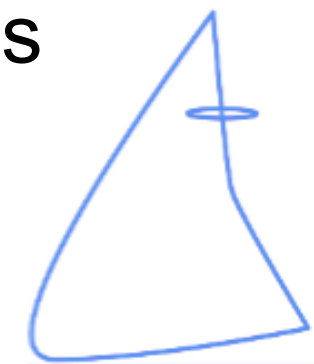
- Two degrees of freedom in mapper run time: Number of tasks/node, and size of InputSplits

- See http://wiki.apache.org/lucene-hadoop/HowManyMapsAndReduces



# More Performance Tweaks

- Hadoop defaults to heap cap of 200 MB
  - Set: mapred.child.java.opts = -Xmx512m
  - 1024 MB / process may also be appropriate
- DFS block size is 64 MB
  - For huge files, set dfs.block.size = 134217728
- mapred.reduce.parallel.copies
  - Set to 15—50; more data => more copies



# Dead Tasks

- Student jobs would “run away”, admin restart needed
- Very often stuck in huge shuffle process
  - Students did not know about Partitioner class, may have had non-uniform distribution
  - Did not use many Reducer tasks
  - Lesson: Design algorithms to use Combiners where possible



# Working With the Scheduler

- Remember: Hadoop has a FIFO job scheduler
  - No notion of fairness, round-robin
- Design your tasks to “play well” with one another
  - Decompose long tasks into several smaller ones which can be interleaved at Job level



# Additional Languages & Components



# Hadoop and C++

- Hadoop Pipes
  - Library of bindings for native C++ code
  - Operates over local socket connection
- Straight computation performance may be faster
- Downside: Kernel involvement and context switches



# Hadoop and Python

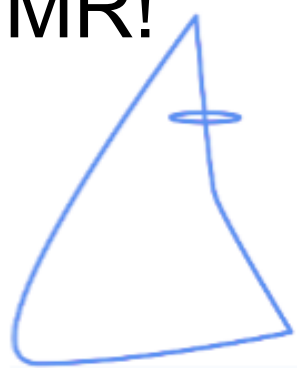
- Option 1: Use Jython
  - Caveat: Jython is a subset of full Python
- Option 2: HadoopStreaming



# HadoopStreaming

- Allows shell pipe '|' operator to be used with Hadoop
- You specify two programs for *map* and *reduce*
  - (+) stdin and stdout do the rest
  - (-) Requires serialization to text, context switches…
  - (+) "cat | grep | sort | uniq" is now a valid MR!





# Eclipse Plugin

- Support for Hadoop in Eclipse IDE
  - Allows MapReduce job dispatch
  - Panel tracks live and recent jobs
- Included in Hadoop since 0.14
  - (But works with older versions)
  - Contributed by IBM



# Conclusions

- Hadoop systems will put up with reasonable amounts of student abuse
- Biggest pitfall is deadlines
- HBase may not be ready for this quarter's students; next year almost certainly
- Other tools provide student design projects with additional options