



Programming with Hadoop



© 2009 Cloudera, Inc.

Overview

- How to use Hadoop
 - Hadoop MapReduce
 - Hadoop Streaming

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

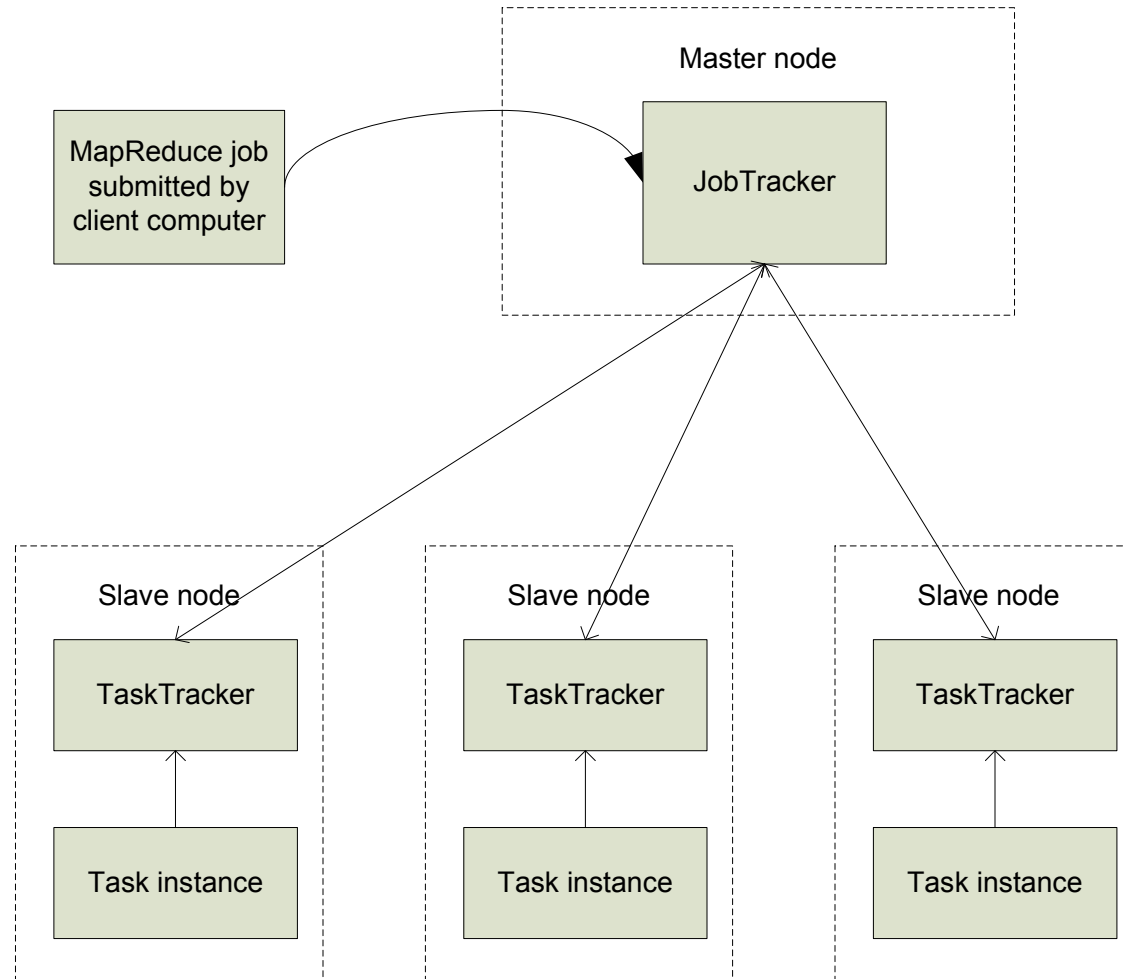
Terminology Example

- 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



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
 - `FileInputFormat.addInputPath(conf)`
 - `FileOutputFormat.setOutputPath(conf)`
 - 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* launches 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 per task
 - 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;  
}
```

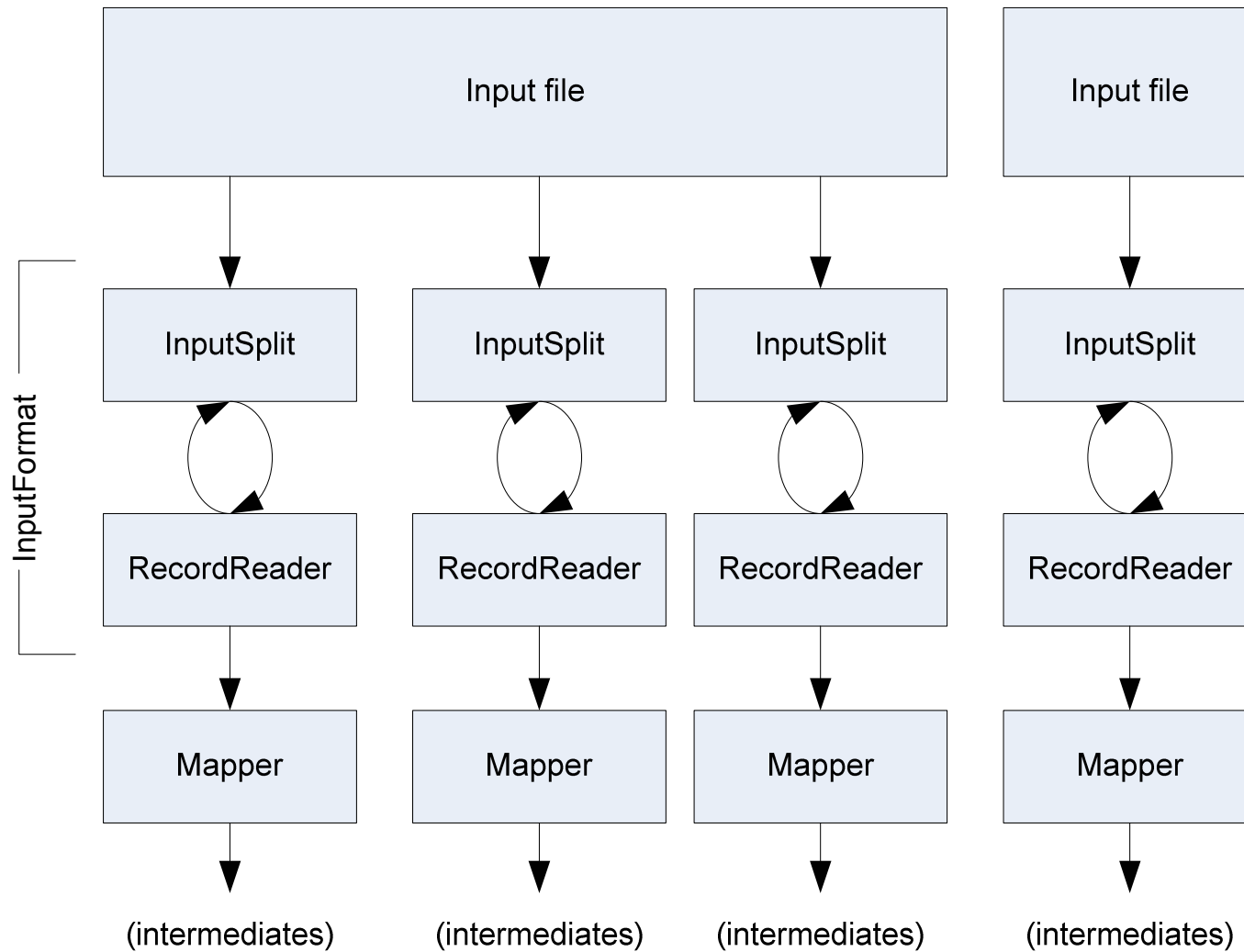
Writing For Cache Coherency

```
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 (up to 2x performance benefit)
- All keys and values given to you by Hadoop use this model (share container objects)

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”

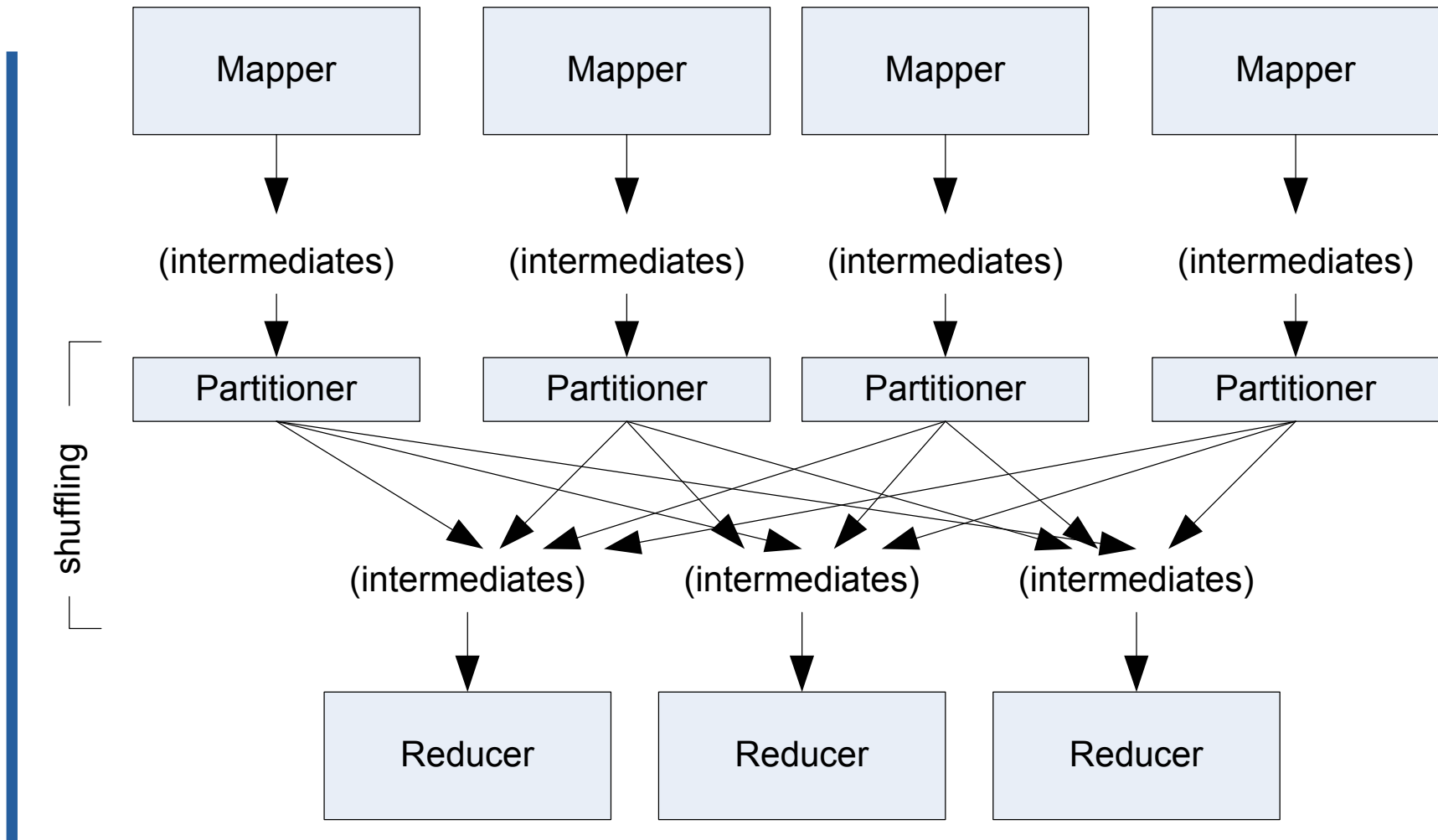
Sending Data To Reducers

- Map function receives *OutputCollector* object
 - `OutputCollector.collect()` takes (k, v) elements
- Any (*WritableComparable*, *Writable*) can be used

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



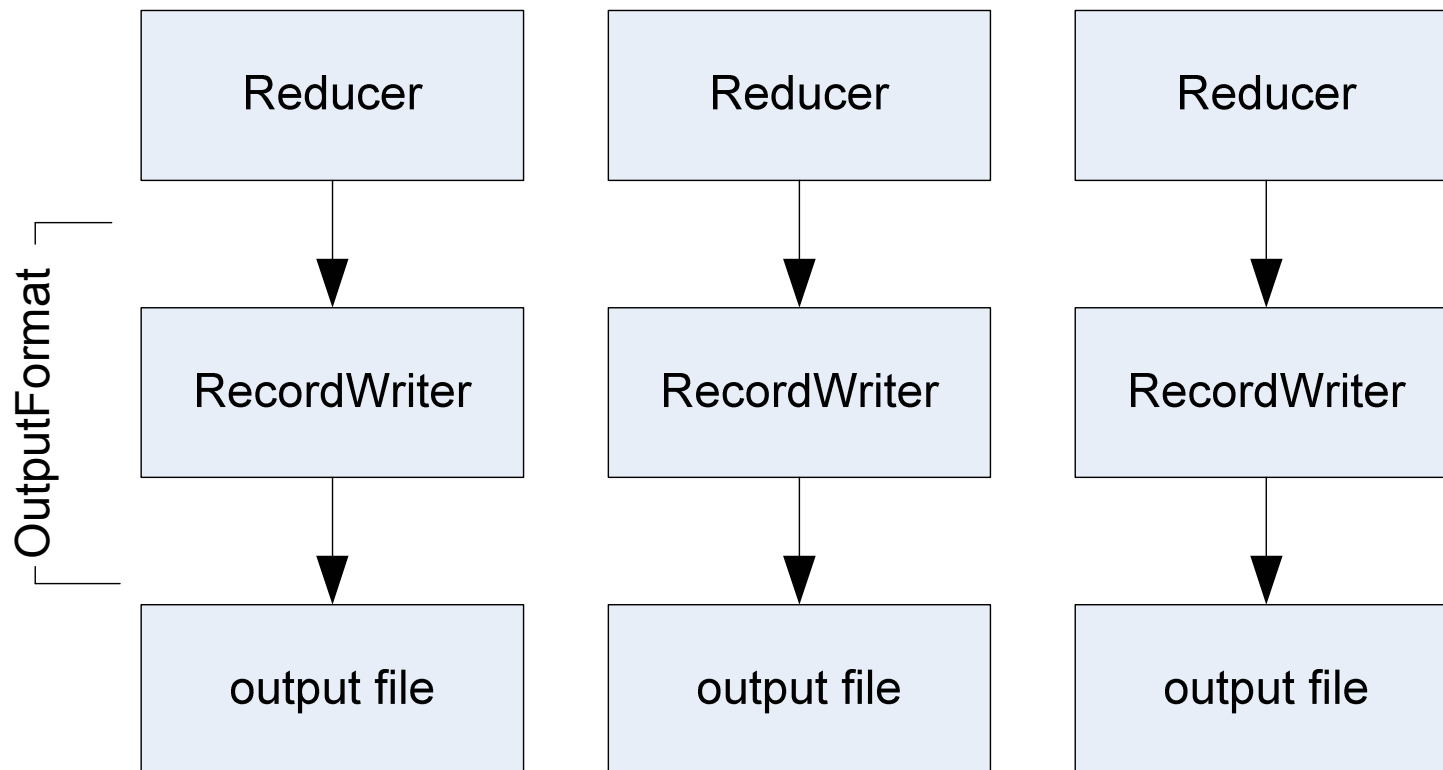
Partitioner

- `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
- Remember – values.next() always returns the same object, different data!

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

Conclusions

- That's the Hadoop flow!
- Lots of flexibility to override components, customize inputs and outputs
- Using custom-built binary formats allows high-speed data movement



Hadoop Streaming

Motivation

- You want to use a scripting language
 - Faster development time
 - Easier to read, debug
 - Use existing libraries
- You (still) have lots of data

HadoopStreaming

- Interfaces Hadoop MapReduce with arbitrary program code
- Uses `stdin` and `stdout` for data flow
- You define a separate program for each of mapper, reducer

Data format

- Input (key, val) pairs sent in as lines of input
key (tab) val (newline)
- Data naturally transmitted as text
- You emit lines of the same form on stdout for output (key, val) pairs.

Example: $\text{map } (k, v) \rightarrow (v, k)$

```
#!/usr/bin/env python
import sys
while True:
    line = sys.stdin.readline()
    if len(line) == 0:
        break
    (k, v) = line.strip().split("\t")
    print v + "\t" + k
```

Launching Streaming Jobs

- Special jar contains streaming “job”
- Arguments select mapper, reducer, format...
- Can also specify Java classes
 - Note: must be in Hadoop “internal” library

Reusing programs

- Identity mapper/reducer: `cat`
- Summing: `wc`
- Field selection: `cut`
- Filtering: `awk`

Streaming Conclusions

- Fast, simple, powerful
- Low-overhead way to get started with Hadoop
- Resources:
 - <http://wiki.apache.org/hadoop/HadoopStreaming>
 - <http://hadoop.apache.org/core/docs/current/streaming.html>

