Distributed and Cloud Computing

K. Hwang, G. Fox and J. Dongarra

Chapter 6: Cloud Programming and Software Environments

Part 1

Adapted from Kai Hwang, University of Southern California
with additions from
Matei Zaharia, EECS, UC Berkeley

November 25, 2012

Parallel Computing and Programming Enviroments

- MapReduce
- Hadoop
- Amazon Web Services

What is MapReduce?

- Simple data-parallel programming model
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability
- Pioneered by Google
  - Processes 20 petabytes of data per day
- Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - “Web map” powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

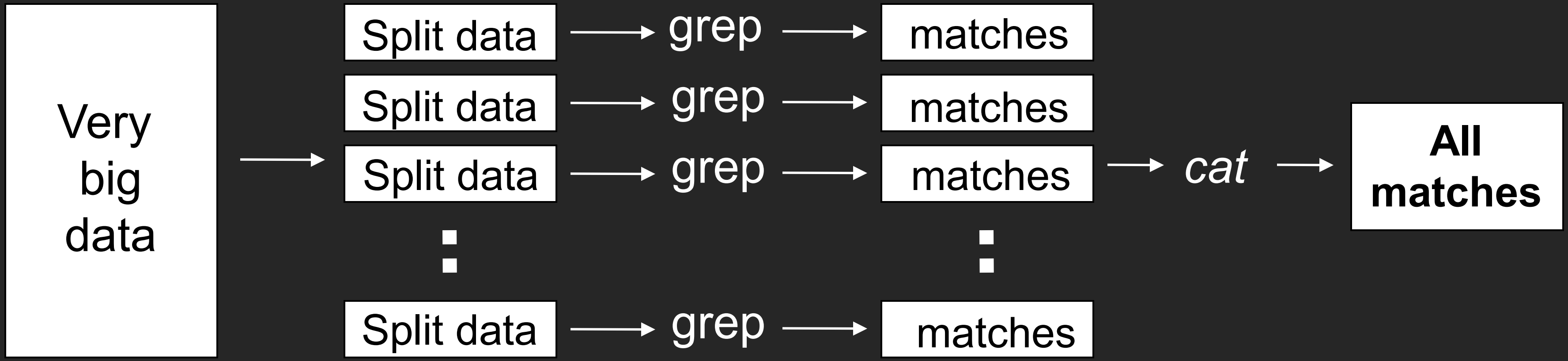
Motivation: Large Scale Data Processing

- Many tasks composed of processing lots of data to produce lots of other data
- Want to use hundreds or thousands of CPUs ... but this needs to be easy!
- MapReduce provides
  - User-defined functions
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

What is MapReduce used for?

- In research:
  - Astronomical image analysis (Washington)
  - Bioinformatics (Maryland)
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Particle physics (Nebraska)
  - Ocean climate simulation (Washington)
  - <Your application here>

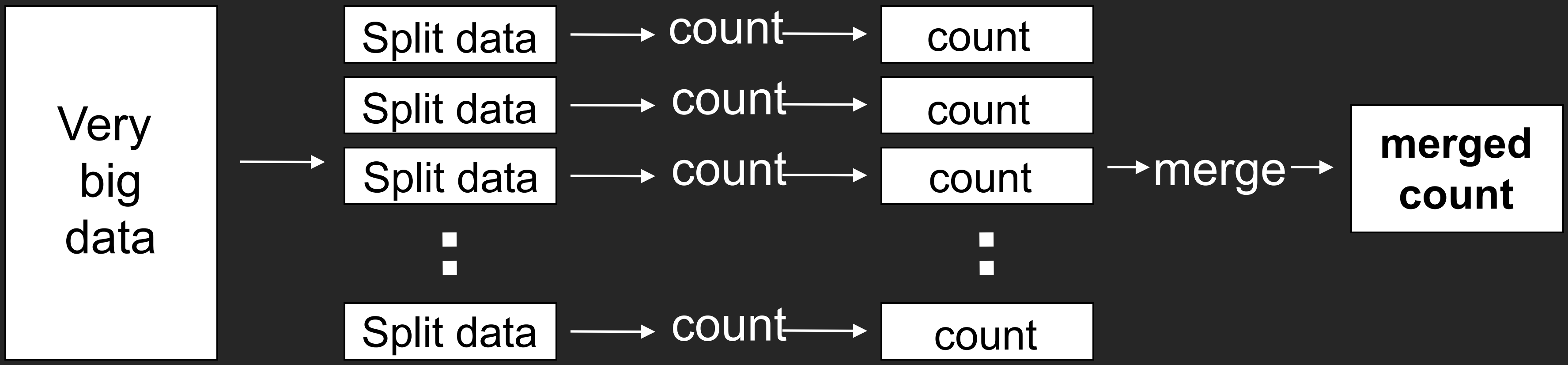
Distributed Grep



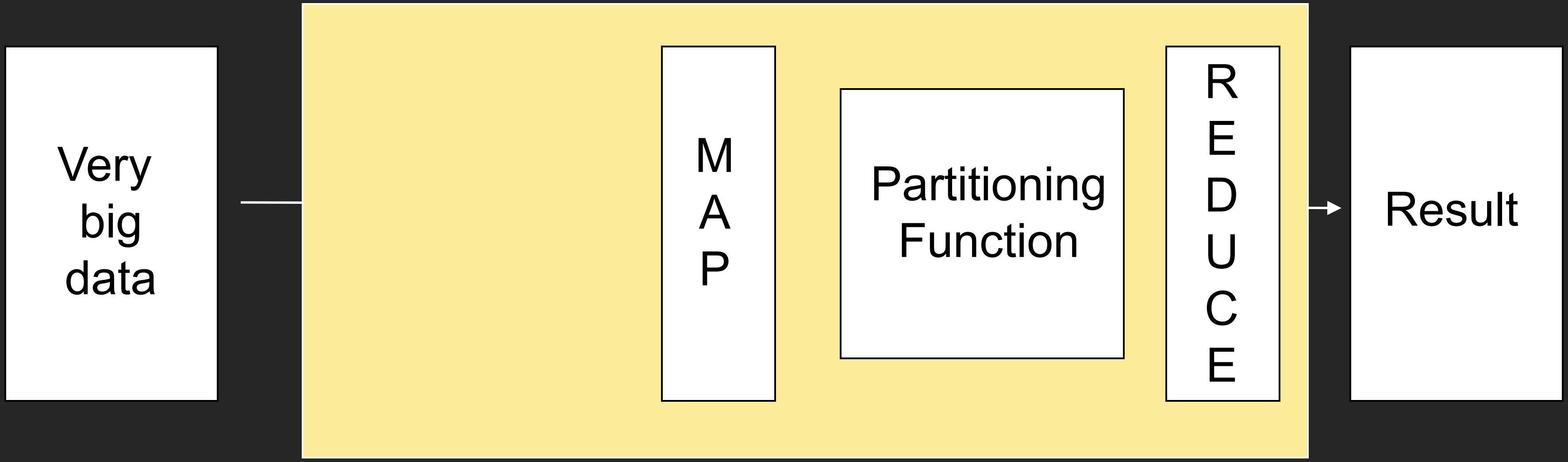
grep is a command-line utility for searching plain-text data sets for lines matching a regular expression.

cat is a standard Unix utility that concatenates and lists files

Distributed Word Count



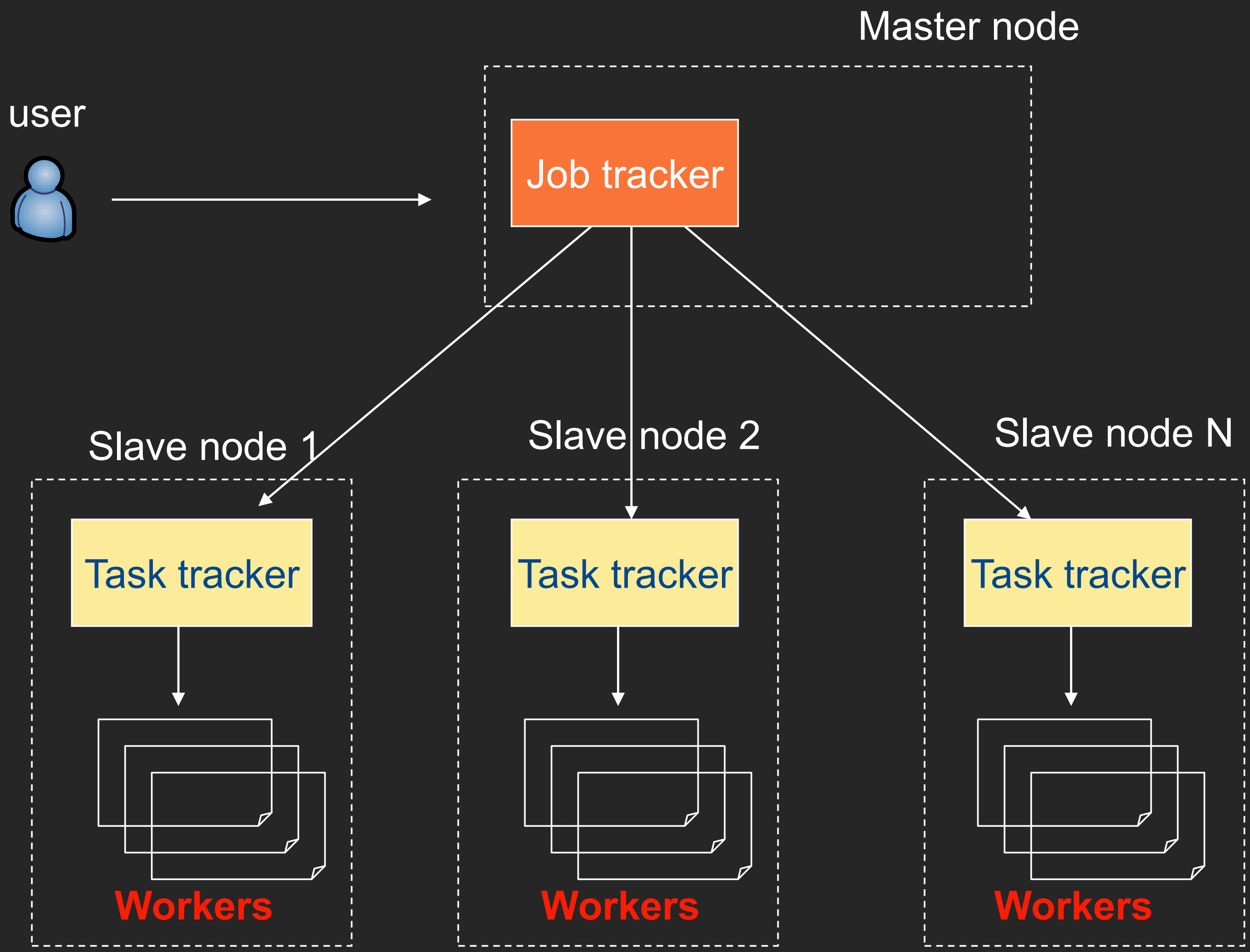
Map+Reduce



- Map:
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair

- Reduce :
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair

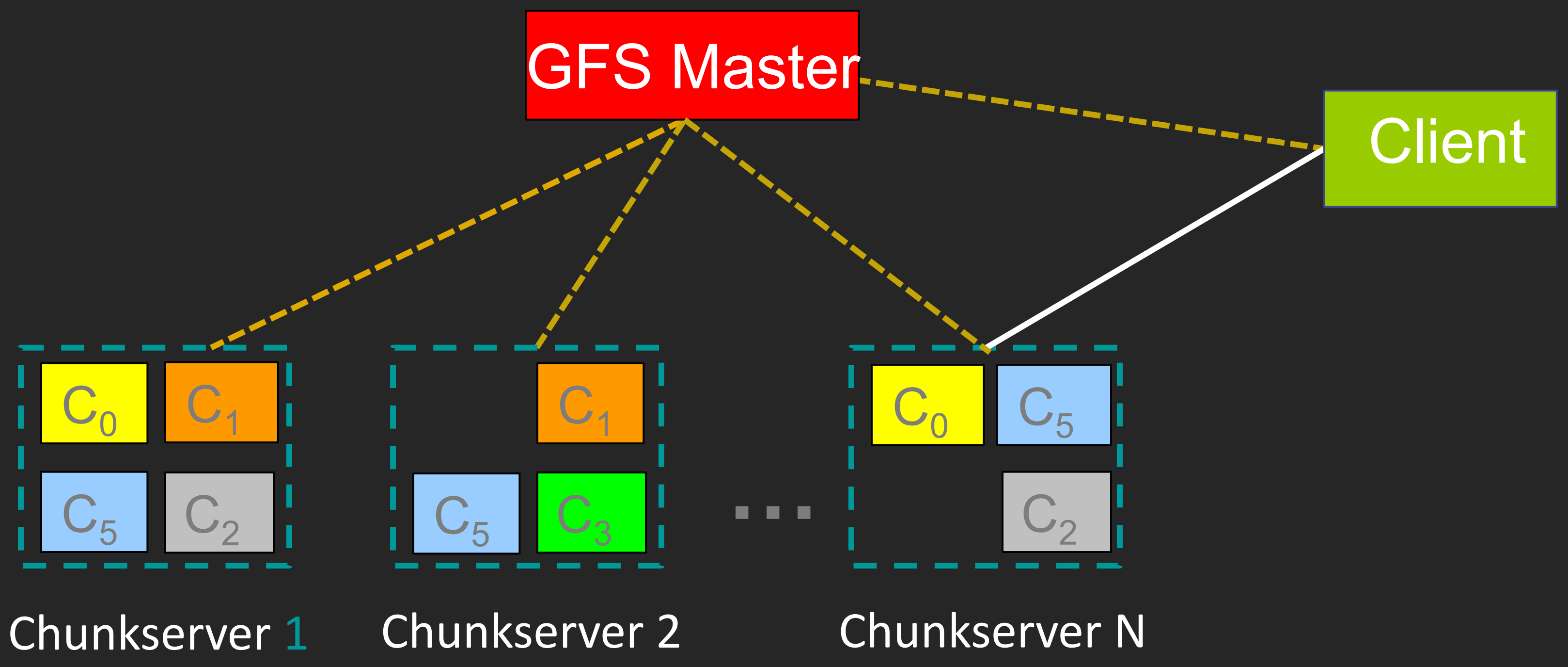
Architecture overview



GFS: underlying storage system

- Goal
  - global view
  - make huge files available in the face of node failures
- Master Node (meta server)
  - Centralized, index all chunks on data servers
- Chunk server (data server)
  - File is split into contiguous chunks, typically 16-64MB.
  - Each chunk replicated (usually 2x or 3x).
    - Try to keep replicas in different racks.

GFS architecture

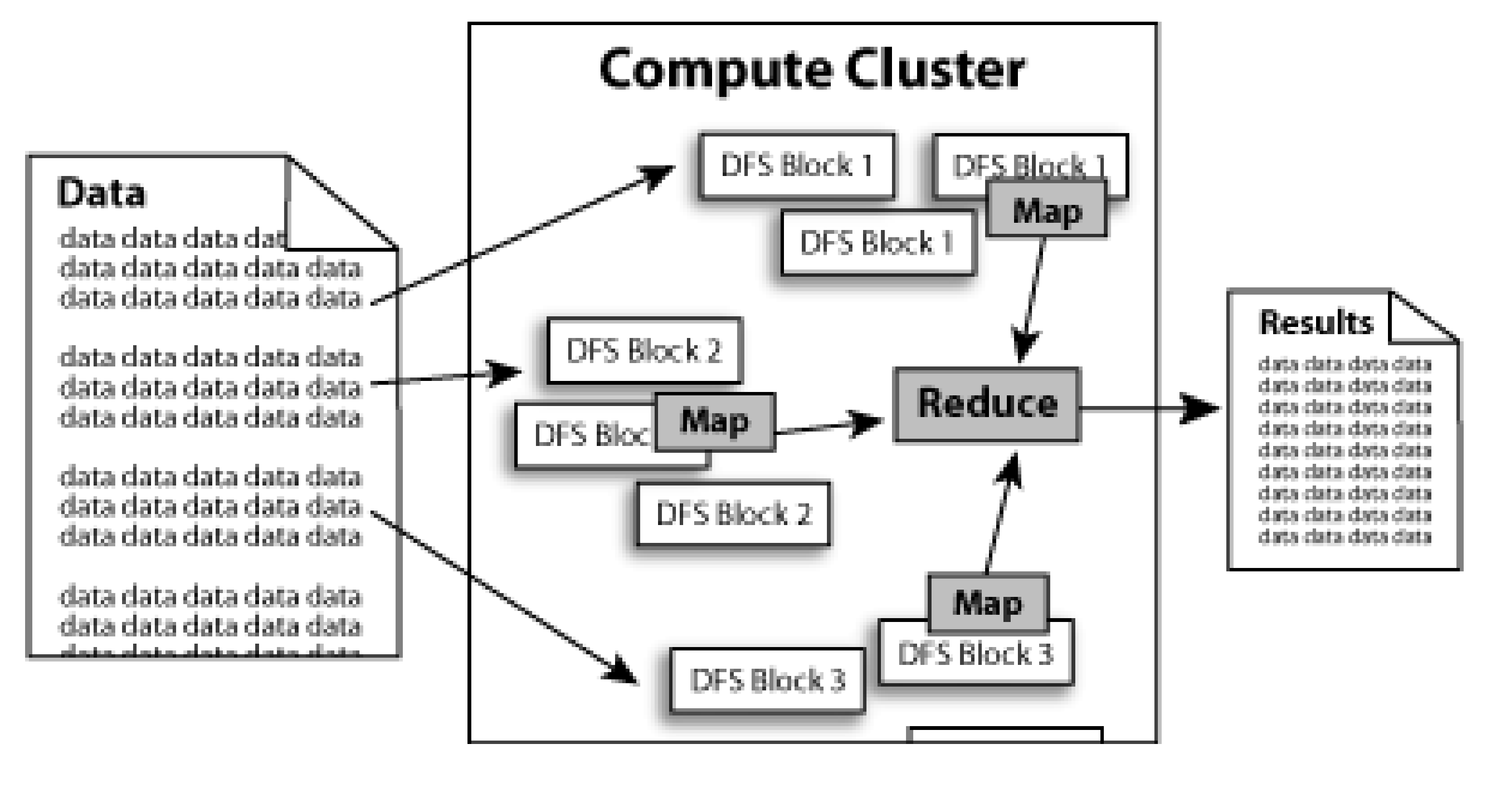


Functions in the Model

- Map
  - Process a key/value pair to generate intermediate key/value pairs
- Reduce
  - Merge all intermediate values associated with the same key
- Partition
  - By default : `hash(key) mod R`
  - Well balanced

Programming Concept

- Map
  - **Perform** a function on **individual values** in a data set to create a **new list** of values
  - Example: square x = x * x
    map square [1,2,3,4,5]
    returns [1,4,9,16,25]
- Reduce
  - **Combine** values in a data set to create a new **value**
  - Example: sum = (each elem in arr, total +=)
    reduce [1,2,3,4,5]
    returns 15 (the sum of the elements)


Compute Cluster
Data
data data data dat
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
DFS Block 1
DFS Block 1
DFS Block 1
Map
DFS Block 2
DFS Bloc
Map
DFS Block 2
Reduce
Map
DFS Block 3
DFS Block 3
Results
data data data data

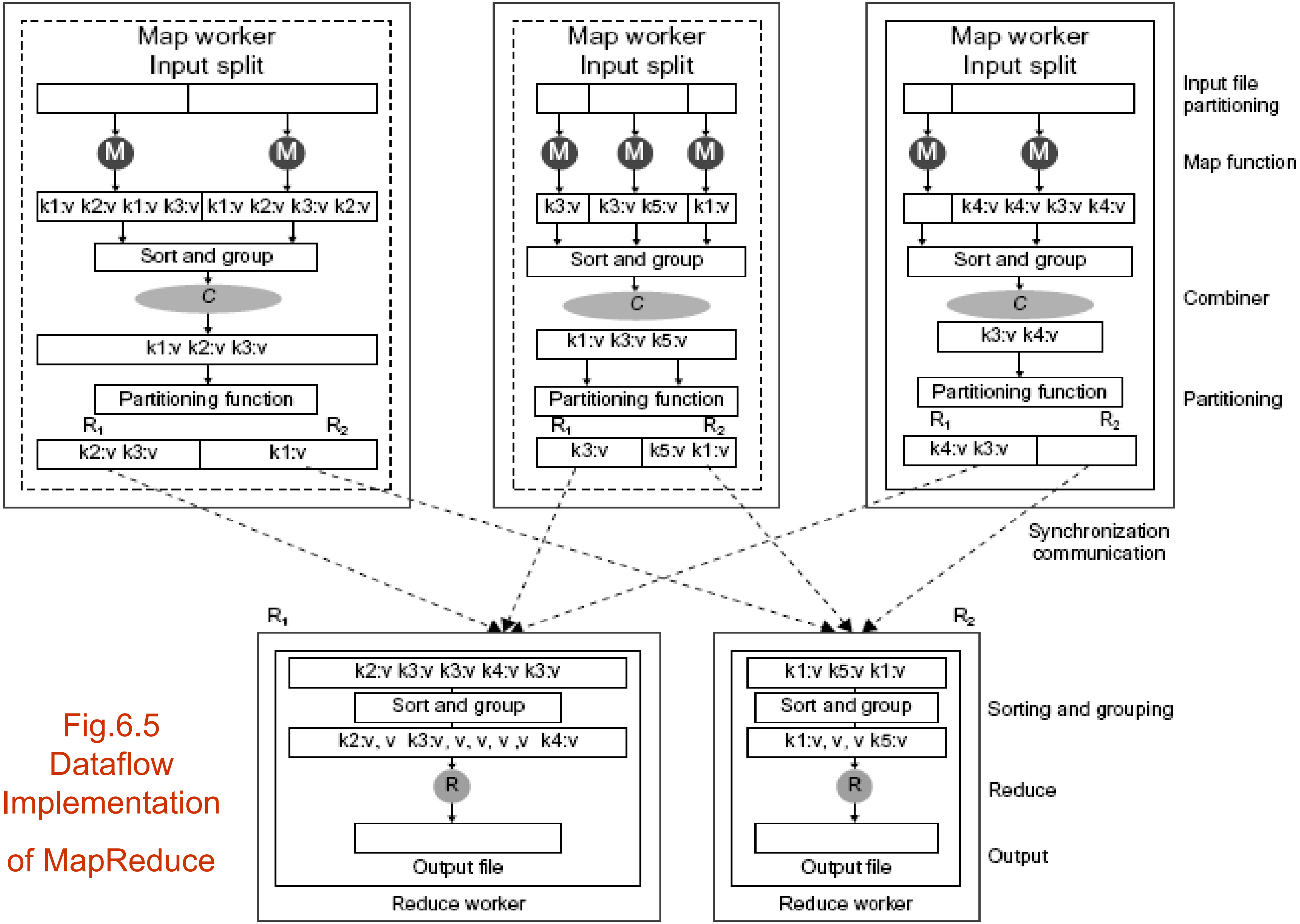


Fig.6.5 Dataflow Implementation

of MapReduce

A Simple Example

- Counting words in a large set of documents

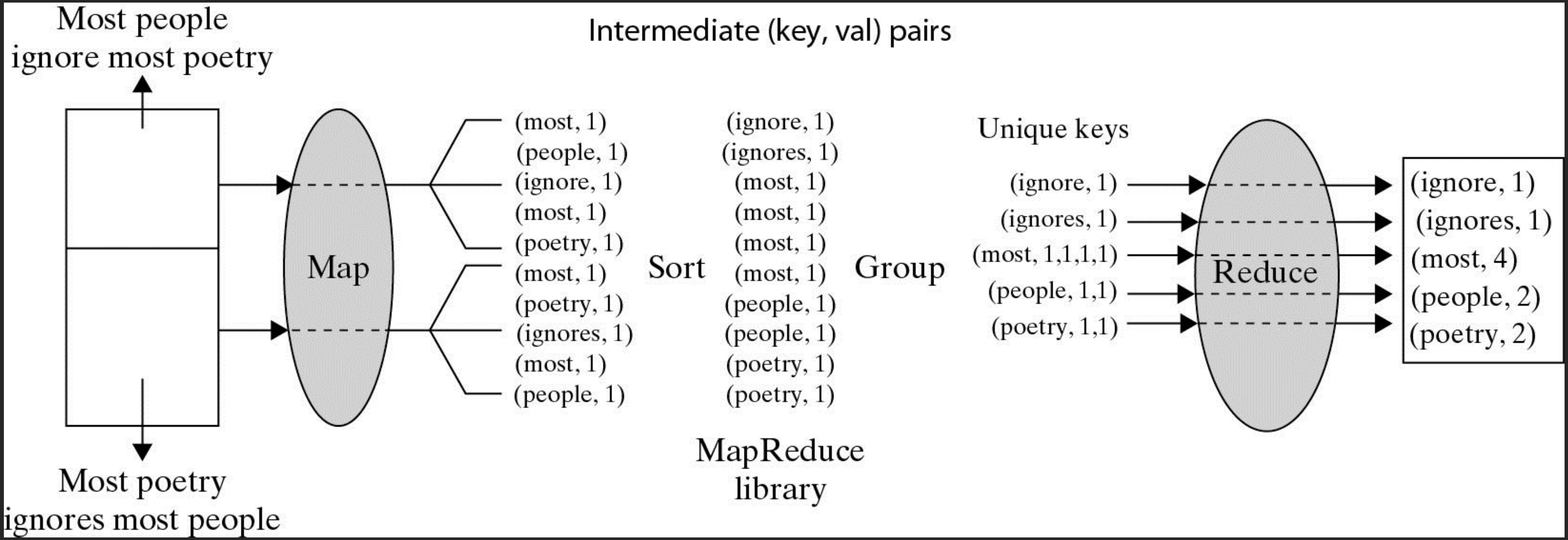
```
map(string value)
    //key: document name
    //value: document contents
    for each word w in value
        EmitIntermediate(w, “1”);

reduce(string key, iterator values)
    //key: word
    //values: list of counts
    int results = 0;
    for each v in values
        result += ParseInt(v);
    Emit(AsString(result));
```

The map function emits each word *w* plus an associated count of occurrences (just a “1” is recorded in this pseudo-code)

The reduce function sums together all counts emitted for a particular word

A Word Counting Example on <Key, Count> Distribution



How Does it work?

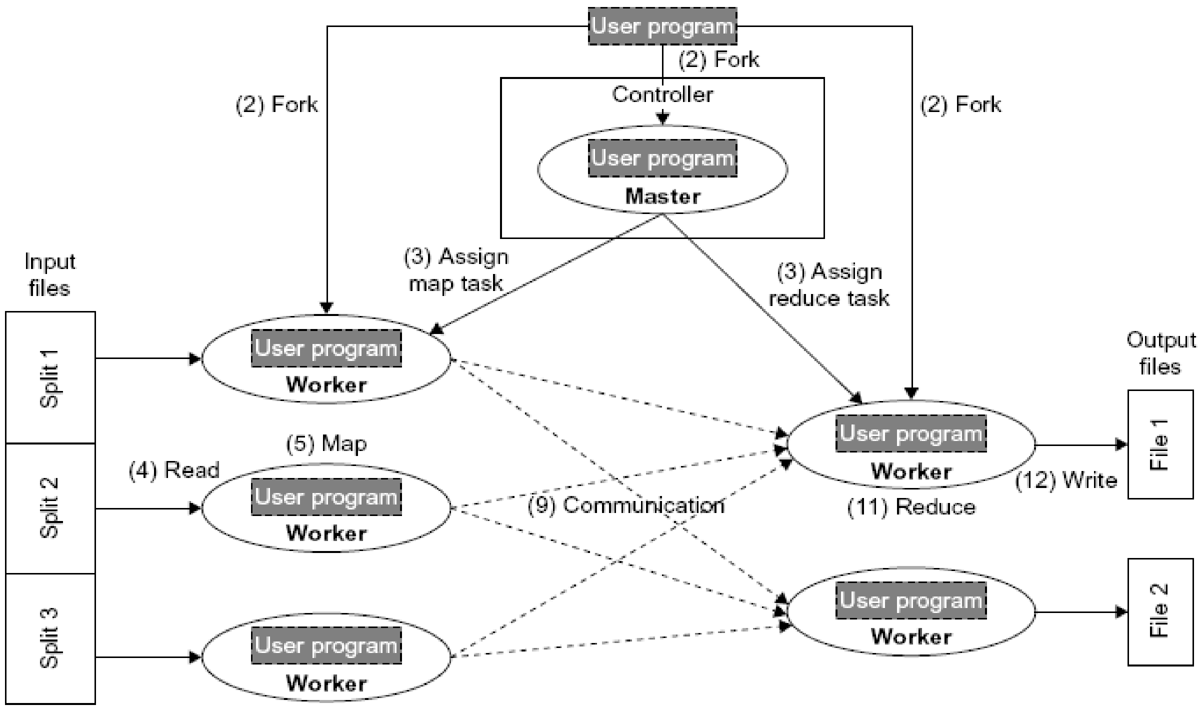


FIGURE 6.6

Control flow implementation of MapReduce.

(Courtesy of Yahoo! Pig Tutorial [54])

- Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits.
- Reduce invocations are distributed by paritioning the intermediate key space into R pieces using a hash function: hash(key) mod R.
  - R and the partitioning function are specified by the programmer.

MapReduce : Operation Steps

When the user program calls the MapReduce function, the following sequence of actions occurs :

1) The MapReduce library in the user program first splits the input files into M pieces – 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of program on a cluster of machines.

2) One of the copies of program is master. The rest are workers that are assigned work by the master.

MapReduce : Operation Steps

3) A worker who is assigned a map task :

- reads the contents of the corresponding input split
- parses key/value pairs out of the input data and passes each pair to the user - defined Map function.

The intermediate key/value pairs produced by the Map function are buffered in memory.

4) The buffered pairs are written to local disk, partitioned into R regions by the partitioning function.

The location of these buffered pairs on the local disk are passed back to the master, who forwards these locations to the reduce workers.

MapReduce : Operation Steps

5) When a reduce worker is notified by the master about these locations, it reads the buffered data from the local disks of the map workers.

When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

6) The reduce worker iterates over the sorted intermediate data and for each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user's Reduce function.

The output of the Reduce function is appended to a final output file.

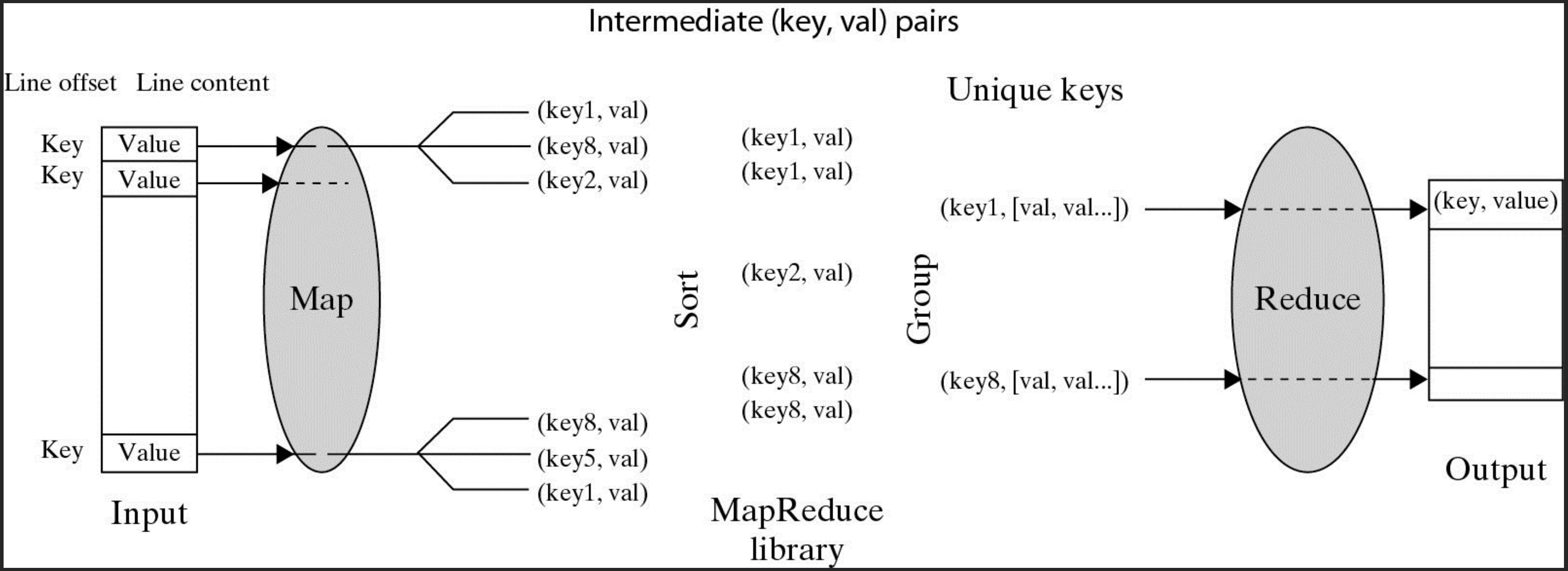
MapReduce : Operation Steps

7) When all map tasks and reduce tasks have been completed, the master wakes up the user program.

At this point, MapReduce call in the user program returns back to the user code.

After successful completion, output of the mapreduce execution is available in the R output files.

Logical Data Flow in 5 Processing Steps in MapReduce Process



(Key, Value) Pairs are generated by the Map function over multiple available Map Workers (VM instances). These pairs are then sorted and group based on key ordering. Different key-groups are then processed by multiple Reduce Workers in parallel.

Locality issue

- Master scheduling policy
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (== GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

Fault Tolerance

- Reactive way
  - Worker failure
    - Heartbeat, Workers are periodically pinged by master
      - NO response = failed worker
    - If the processor of a worker fails, the tasks of that worker are reassigned to another worker.
  - Master failure
    - Master writes periodic checkpoints
    - Another master can be started from the last checkpointed state
    - If eventually the master dies, the job will be aborted

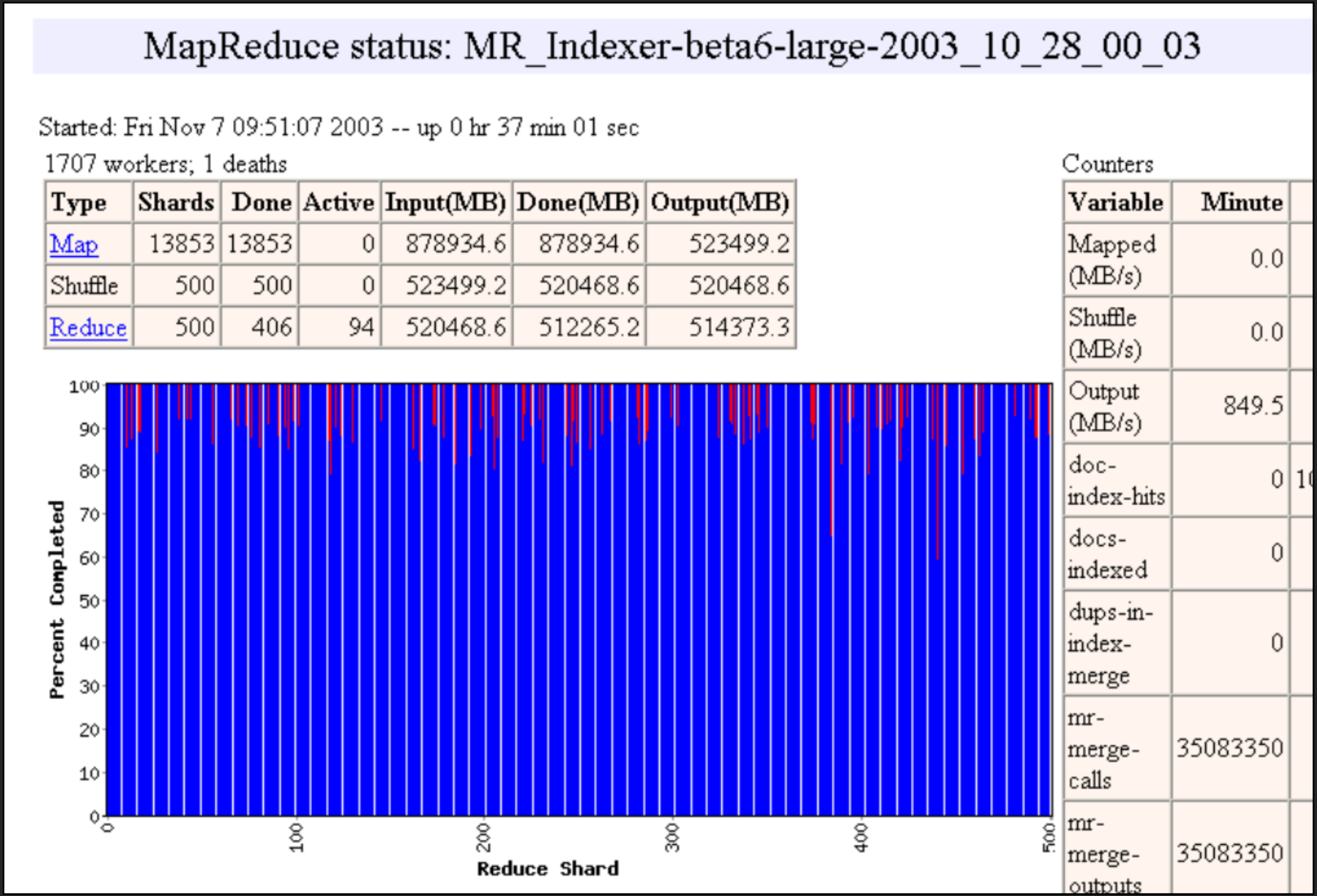
Fault Tolerance

- Proactive way (**Redundant Execution**)
  - The problem of “stragglers” (slow workers)
    - Other jobs consuming resources on machine
    - Bad disks with soft errors transfer data very slowly
    - Weird things: processor caches disabled (!!)
  - When computation almost done, reschedule in-progress tasks
  - Whenever either the primary or the backup executions finishes, mark it as completed

Fault Tolerance

- Input error: bad records
  - Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix, but not always possible
  - On segment fault
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - Skip bad records
    - If master sees two failures for same record, next worker is told to skip the record

Status monitor



MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 37 min 01 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	520468.6	520468.6
Reduce	500	406	94	520468.6	512265.2	514373.3

Counters

Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	849.5	
doc-index-hits	0	1
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	35083350	
mr-merge-outputs	35083350	

Points need to be emphasized

- No *reduce* can begin until *map* is complete
- Master must communicate locations of intermediate files
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- MapReduce library does most of the hard work for us!

Other Examples

- Distributed Grep:
  - Map function emits a line if it matches a supplied pattern.
  - Reduce function is an identity function that copies the supplied intermediate data to the output.
- Count of URL accesses:
  - Map function processes logs of web page requests and outputs <URL, 1>,
  - Reduce function adds together all values for the same URL, emitting <URL, total count> pairs.
- Reverse Web-Link graph; e.g., all URLs with reference to http://dblab.usc.edu:
  - Map function outputs <tgt, src> for each link to a tgt in a page named src,
  - Reduce concatenates the list of all src URLS associated with a given tgt URL and emits the pair: <tgt, list(src)>.
- Inverted Index; e.g., all URLs with 585 as a word:
  - Map function parses each document, emitting a sequence of <word, doc_ID>,
  - Reduce accepts all pairs for a given word, sorts the corresponding doc_IDs and emits a <word, list(doc_ID)> pair.
  - Set of all output pairs forms a simple inverted index.

MapReduce Implementations

MapReduce

Cluster,
1, Google
2, Apache Hadoop

Multicore CPU,
Phoenix @ stanford

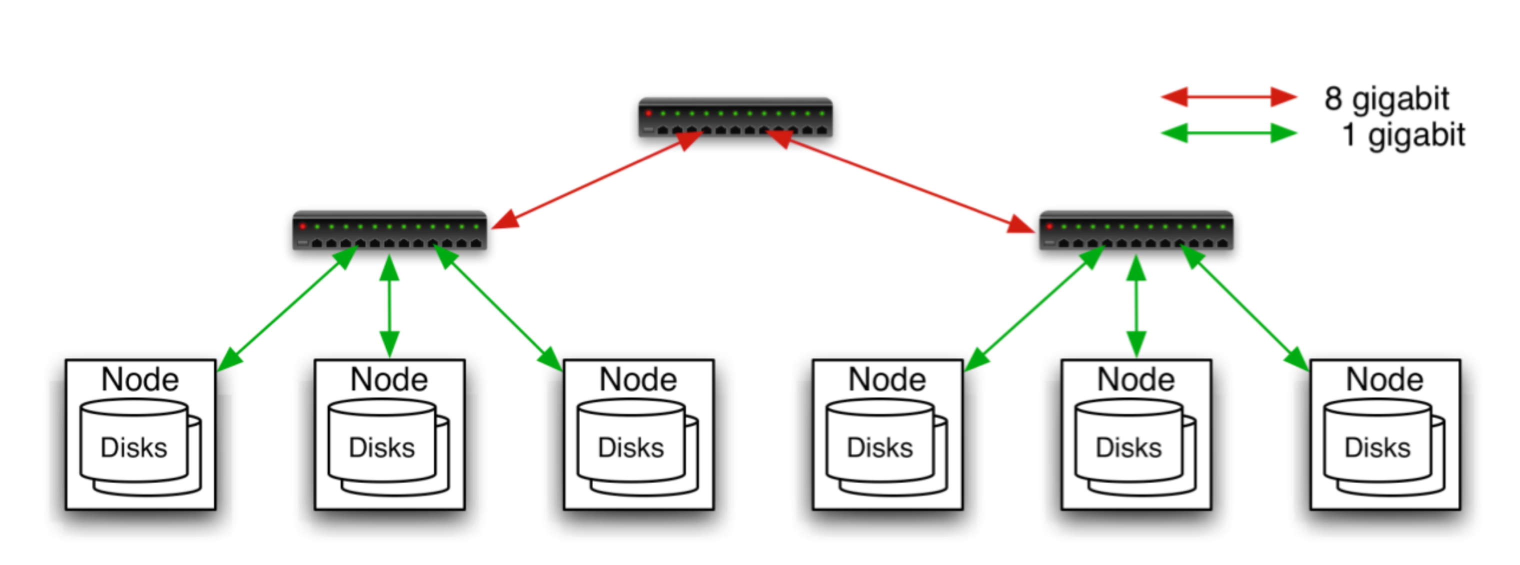
GPU,
Mars@HKUST

Hadoop : software platform originally developed by Yahoo enabling users to write and run applications over vast distributed data.

Attractive Features in Hadoop :

- Scalable : can easily scale to store and process petabytes of data in the Web space
- Economical : An open-source MapReduce minimizes the overheads in task spawning and massive data communication.
- Efficient: Processing data with high-degree of parallelism across a large number of commodity nodes
- Reliable : Automatically maintains multiple copies of data to facilitate redeployment of computing tasks on failures

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf

Typical Hadoop Cluster

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

Challenges

Cheap nodes fail, especially if you have many

1. Mean time between failures for 1 node = 3 years
2. Mean time between failures for 1000 nodes = 1 day
3. Solution: Build fault-tolerance into system

Commodity network = low bandwidth

1. Solution: Push computation to the data

Programming distributed systems is hard

1. Solution: Data-parallel programming model: users write “map” & “reduce” functions, system distributes work and handles faults

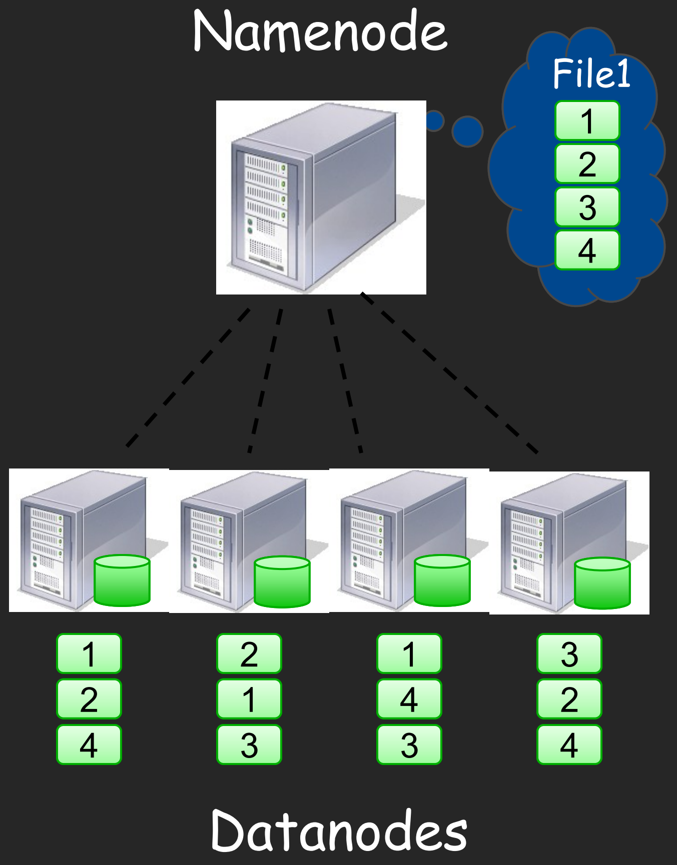
Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance

- MapReduce framework
  - Executes user jobs specified as “map” and “reduce” functions
  - Manages work distribution & fault-tolerance

Hadoop Distributed File System

- Files split into 128MB *blocks*
- Blocks replicated across several *datanodes* (usually 3)
- Single *namenode* stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



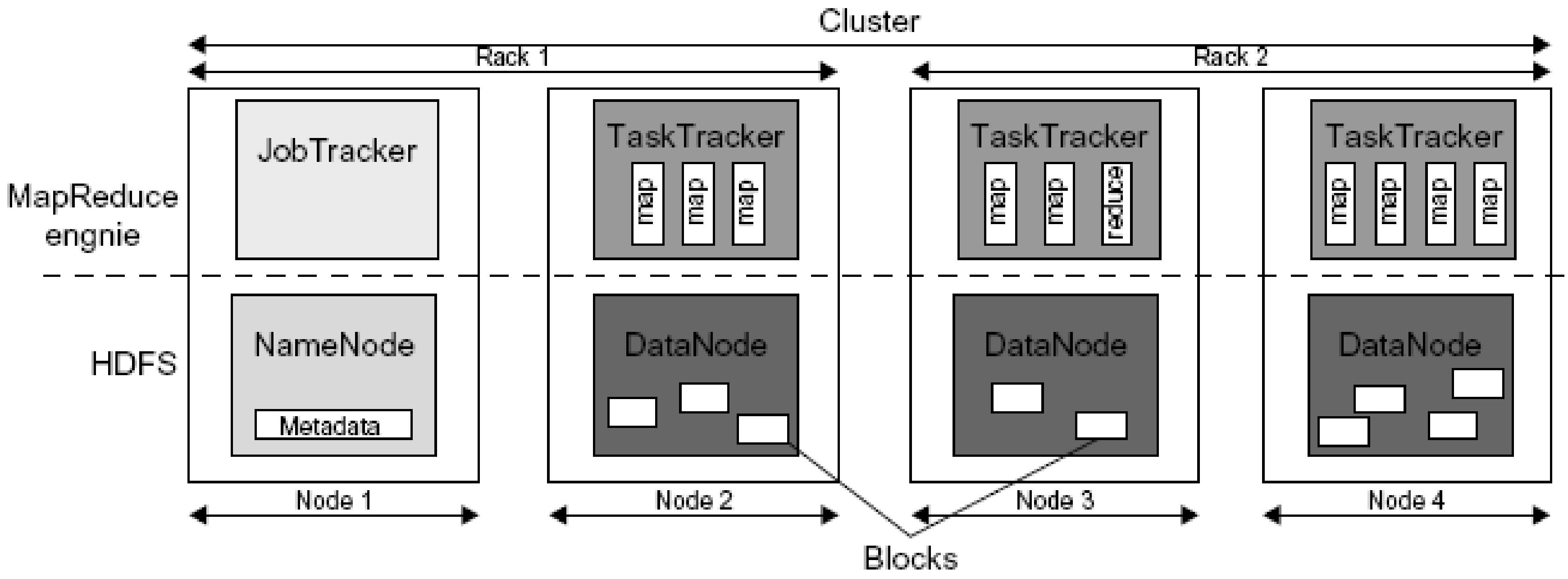


FIGURE 6.11

HDFS and MapReduce architecture in Hadoop.

Secure Query Processing with Hadoop/MapReduce

- Query Rewriting and Optimization Principles defined and implemented for two types of data
- (i) Relational data: Secure query processing with HIVE
- (ii) RDF Data: Secure query processing with SPARQL
- Demonstrated with XACML Policies (content, temporal, association)
- Joint demonstration with Kings College and U. of Insubria
  - First demo (2010): Each party submits their data and policies
  - Our cloud will manage the data and policies
  - Second demo (2011): Multiple clouds

Higher-level languages over Hadoop: Pig and Hive

Motivation

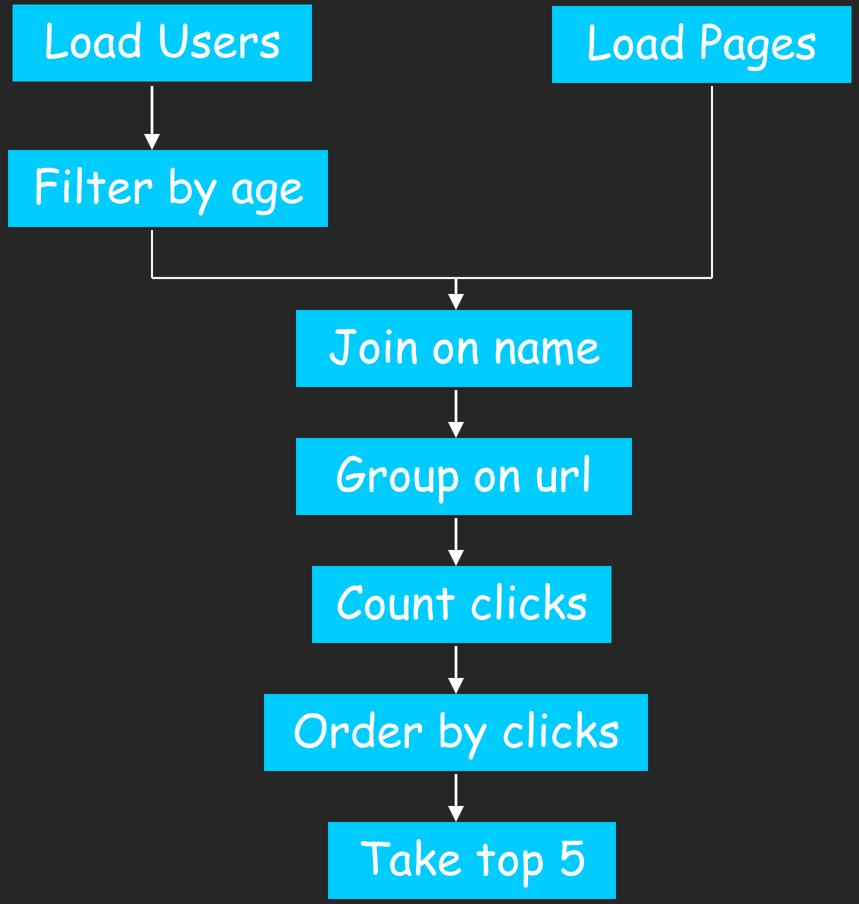
- Many parallel algorithms can be expressed by a series of MapReduce jobs
- But MapReduce is fairly low-level: must think about keys, values, partitioning, etc
- Can we capture common “job building blocks”?

Pig

- Started at Yahoo! Research
- Runs about 30% of Yahoo!'s jobs
- Features:
  - Expresses sequences of MapReduce jobs
  - Data model: nested “bags” of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  - Easy to plug in Java functions
  - Pig Pen development environment for Eclipse

An Example Problem

Suppose you have user data in one file, page view data in another, and you need to find the top 5 most visited pages by users aged 18 - 25.



Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

In MapReduce

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
           OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
```

```
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
```

```
        lp.setOutputKeyClass(Text.cl
        lp.setOutputValueClass(Text.
        lp.setMapperClass(LoadPages.
        FileInputFormat.addInputPath
new Path("/user/gates/pages"));
        FileOutputFormat.setOutputPa
            new Path("/user/gates/tm
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MR
        lfu.setJobName("Load and Filter Use
        lfu.setInputFormat(TextInput
        lfu.setOutputKeyClass(Text.c
        lfu.setOutputValueClass(Text
        lfu.setMapperClass(LoadAndFi
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPa
            new Path("/user/gates/tm
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu)

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users
        join.setInputFormat(KeyValue
        join.setOutputKeyClass(Text.
        join.setOutputValueClass(Tex
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.cl
        FileInputFormat.addInputPath
Path("/user/gates/tmp/indexed_pages"
        FileInputFormat.addInputPath
Path("/user/gates/tmp/filtered_users
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(load
        joinJob.addDependingJob(load

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs
        group.setInputFormat(KeyValu
        group.setOutputKeyClass(Text
        group.setOutputValueClass(Lo
        group.setOutputFormat(Sequence
        group.setMapperClass(LoadJoi
        group.setCombinerClass(Reduc
        group.setReducerClass(Reduce
        FileInputFormat.addInputPath
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group
        groupJob.addDependingJob(joi

        JobConf top100 = new JobConf
        top100.setJobName("Top 100 si
        top100.setInputFormat(Sequen
        top100.setOutputKeyClass(Lon
        top100.setOutputValueClass(T
        top100.setOutputFormat(Seque
        top100.setMapperClass(LoadCl
        top100.setCombinerClass(Limi
        top100.setReducerClass(Limit
        FileInputFormat.addInputPath
Path("/user/gates/tmp/grouped"));
    FileOutputFormat.setOutputPath(t
Path("/user/gates/top100sitesforuser
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJ

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

In Pig Latin

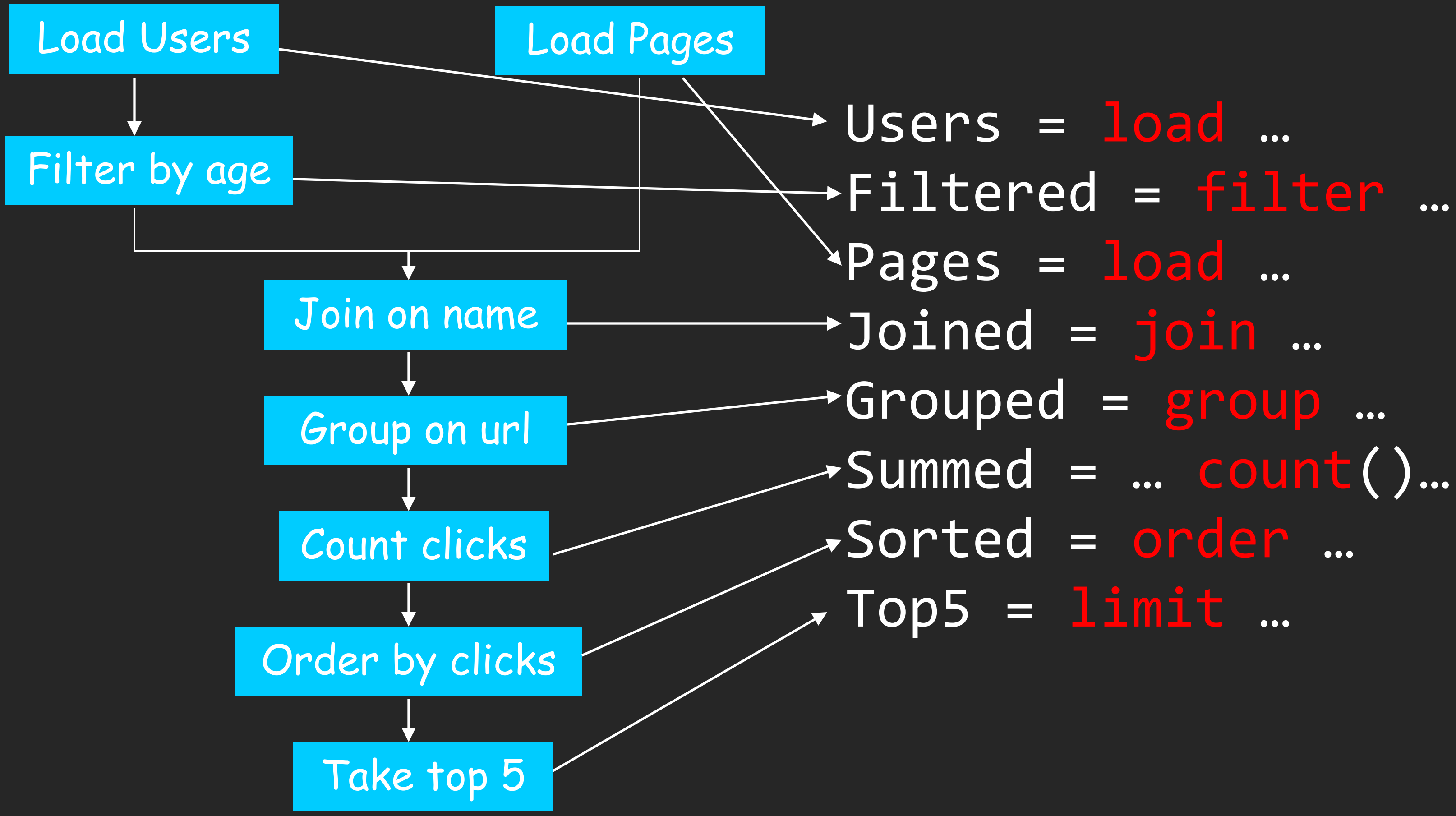
```
Users     = load ‘users’ as (name, age);
Filtered = filter Users by
                  age >= 18 and age <= 25;
Pages     = load ‘pages’ as (user, url);
Joined    = join Filtered by name, Pages by user;
Grouped   = group Joined by url;
Summed    = foreach Grouped generate group,
                   count(Joined) as clicks;
Sorted    = order Summed by clicks desc;
Top5      = limit Sorted 5;

store Top5 into ‘top5sites’;
```

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

Ease of Translation

Notice how naturally the components of the job translate into Pig Latin.

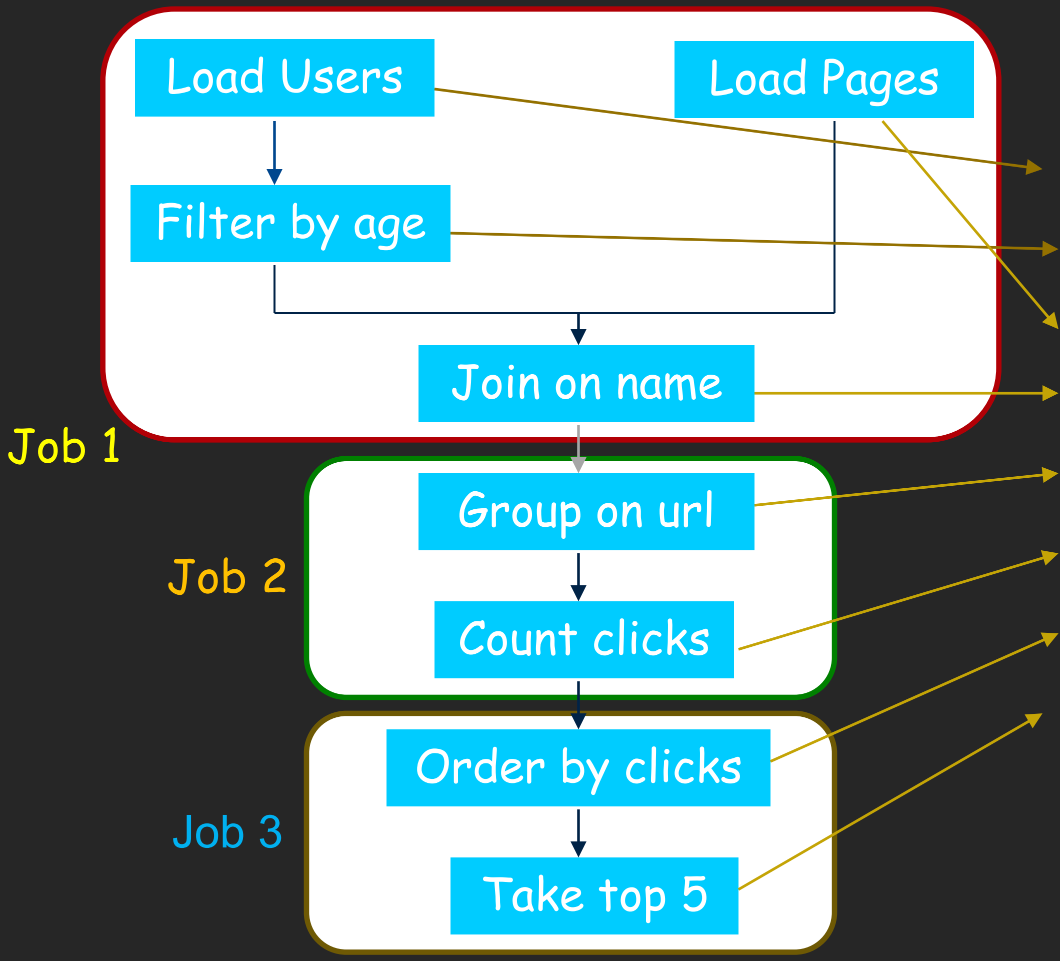


```
Users = load …
Filtered = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …
```

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

Ease of Translation

Notice how naturally the components of the job translate into Pig Latin.



```
Users = load …
Filtered = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …
```

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

Hive

- Developed at Facebook
- Used for majority of Facebook jobs
- “Relational database” built on Hadoop
  - Maintains list of table schemas
  - SQL-like query language (HQL)
  - Can call Hadoop Streaming scripts from HQL
  - Supports table partitioning, clustering, complex data types, some optimizations

Sample Hive Queries

- Find top 5 pages visited by users aged 18-25:

```
SELECT p.url, COUNT(1) as clicks
FROM users u JOIN page_views p ON (u.name = p.user)
WHERE u.age >= 18 AND u.age <= 25
GROUP BY p.url
ORDER BY clicks
LIMIT 5;
```

- Filter page views through Python script:

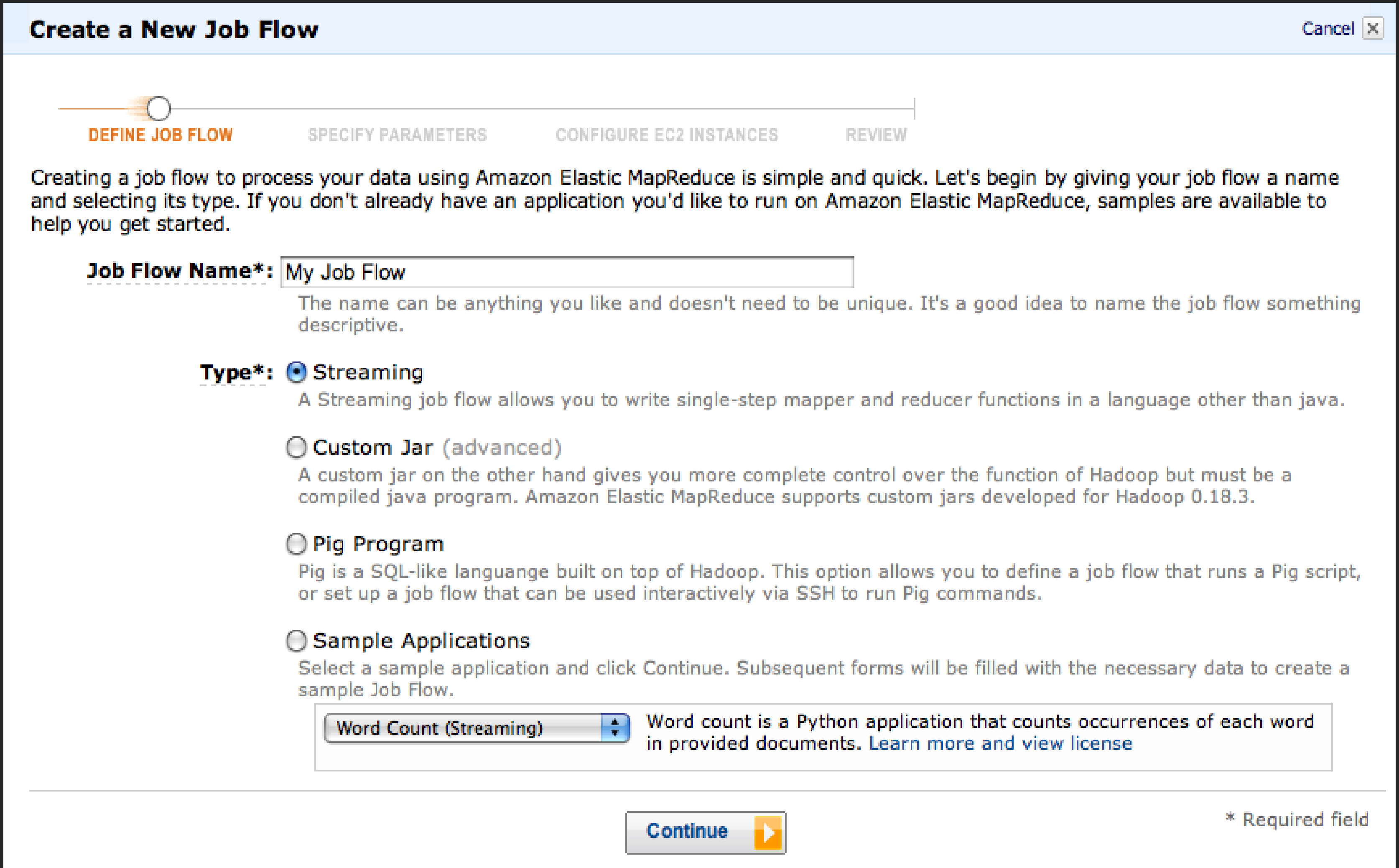
```
SELECT TRANSFORM(p.user, p.date)
USING 'map_script.py'
AS dt, uid CLUSTER BY dt
FROM page_views p;
```

Amazon Elastic MapReduce

- Provides a web-based interface and command-line tools for running Hadoop jobs on Amazon EC2
- Data stored in Amazon S3
- Monitors job and shuts down machines after use
- Small extra charge on top of EC2 pricing

- If you want more control over how you Hadoop runs, you can launch a Hadoop cluster on EC2 manually using the scripts in `src/contrib/ec2`

Elastic MapReduce Workflow



Create a New Job Flow

Creating a job flow to process your data using Amazon Elastic MapReduce is simple and quick. Let's begin by giving your job flow a name and selecting its type. If you don't already have an application you'd like to run on Amazon Elastic MapReduce, samples are available to help you get started.

Job Flow Name*: My Job Flow

The name can be anything you like and doesn't need to be unique. It's a good idea to name the job flow something descriptive.

Type*: ● Streaming

A Streaming job flow allows you to write single-step mapper and reducer functions in a language other than java.

○ Custom Jar (advanced)

A custom jar on the other hand gives you more complete control over the function of Hadoop but must be a compiled java program. Amazon Elastic MapReduce supports custom jars developed for Hadoop 0.18.3.

○ Pig Program

Pig is a SQL-like languange built on top of Hadoop. This option allows you to define a job flow that runs a Pig script, or set up a job flow that can be used interactively via SSH to run Pig commands.

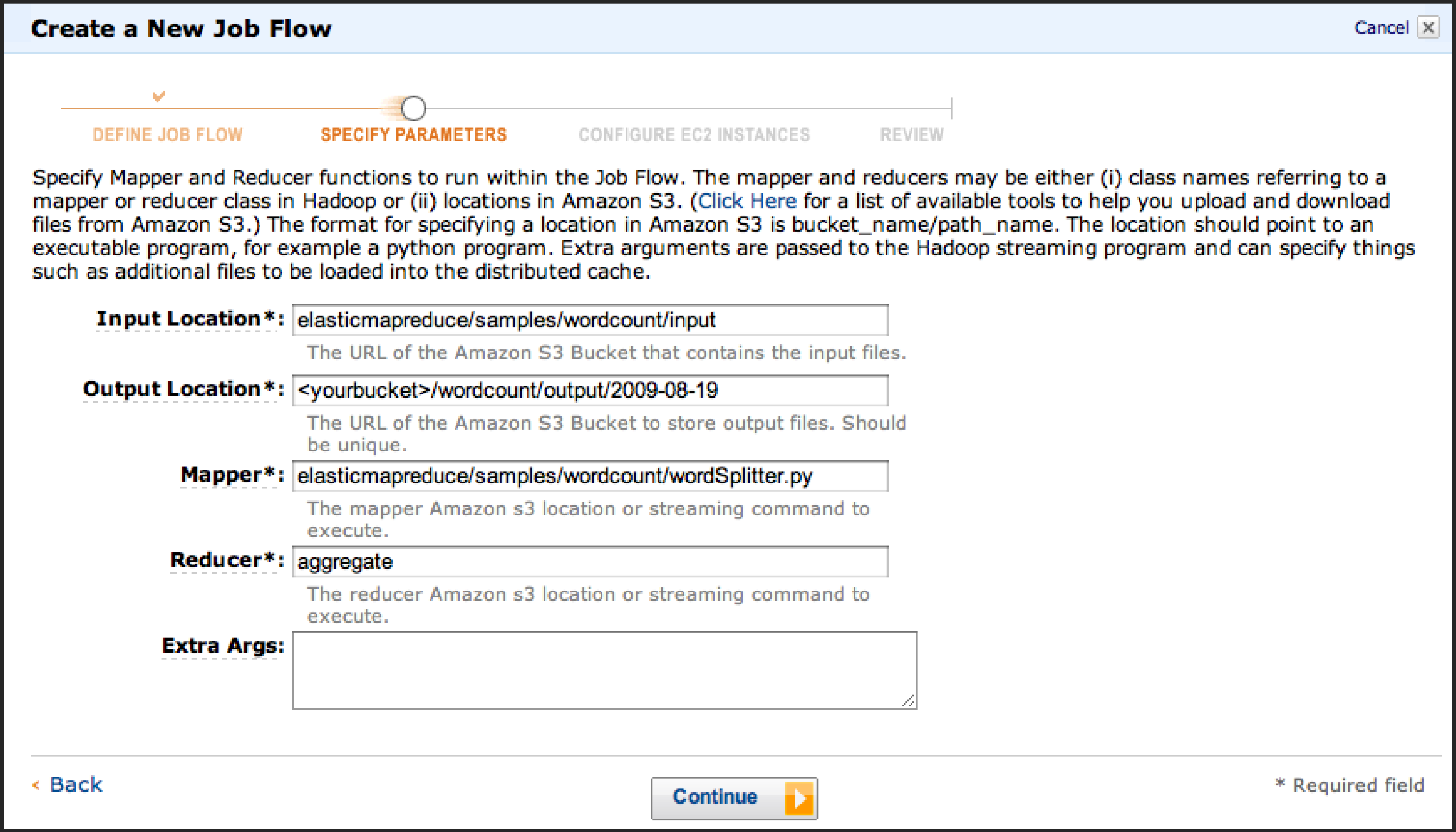
○ Sample Applications

Select a sample application and click Continue. Subsequent forms will be filled with the necessary data to create a sample Job Flow.

Word Count (Streaming)

Word count is a Python application that counts occurrences of each word in provided documents. Learn more and view license

Elastic MapReduce Workflow



Create a New Job Flow

Specify Mapper and Reducer functions to run within the Job Flow. The mapper and reducers may be either (i) class names referring to a mapper or reducer class in Hadoop or (ii) locations in Amazon S3. (Click Here for a list of available tools to help you upload and download files from Amazon S3.) The format for specifying a location in Amazon S3 is bucket_name/path_name. The location should point to an executable program, for example a python program. Extra arguments are passed to the Hadoop streaming program and can specify things such as additional files to be loaded into the distributed cache.

Input Location*: elasticmapreduce/samples/wordcount/input
The URL of the Amazon S3 Bucket that contains the input files.

Output Location*: <yourbucket>/wordcount/output/2009-08-19
The URL of the Amazon S3 Bucket to store output files. Should be unique.

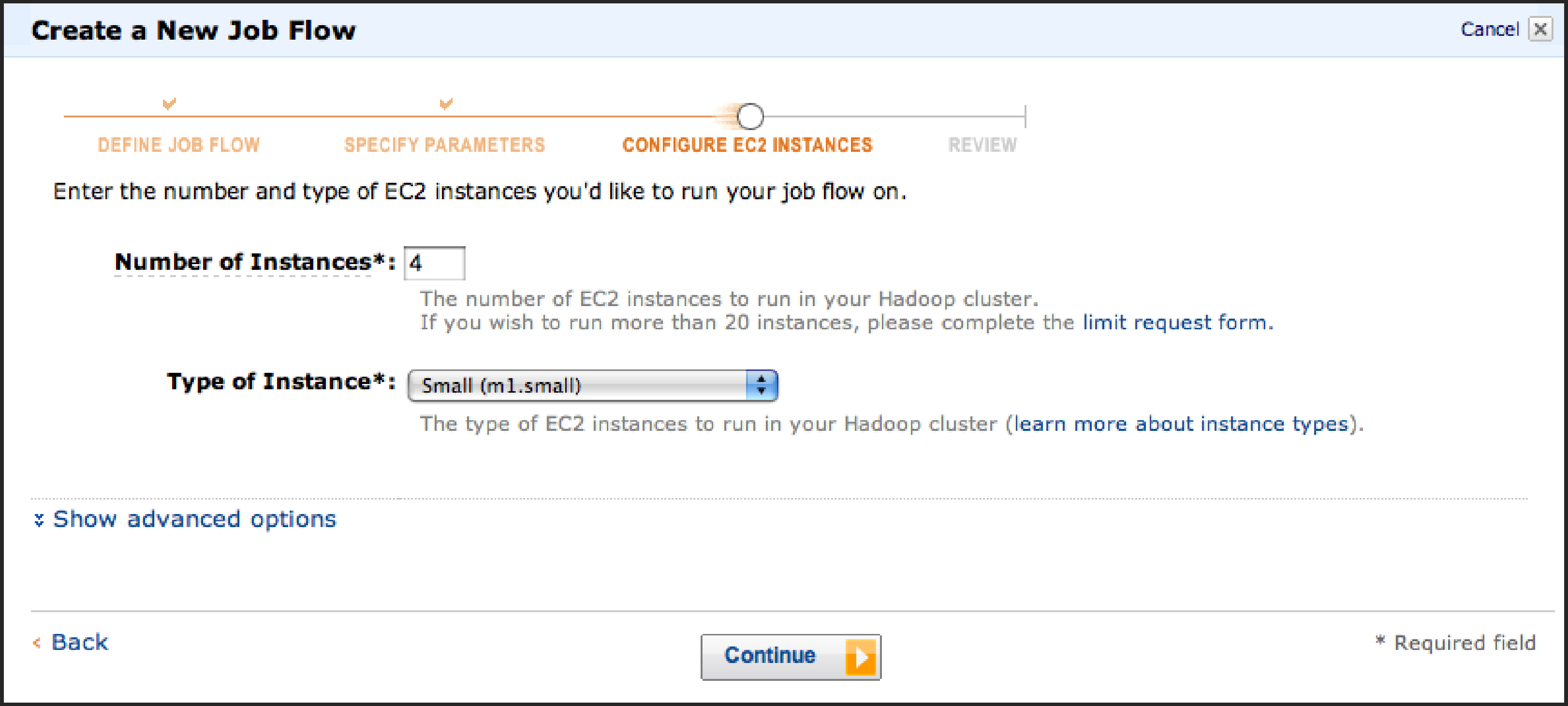
Mapper*: elasticmapreduce/samples/wordcount/wordSplitter.py
The mapper Amazon s3 location or streaming command to execute.

Reducer*: aggregate
The reducer Amazon s3 location or streaming command to execute.

Extra Args:

* Required field

Elastic MapReduce Workflow



Create a New Job Flow

Enter the number and type of EC2 instances you'd like to run your job flow on.

Number of Instances*: 4

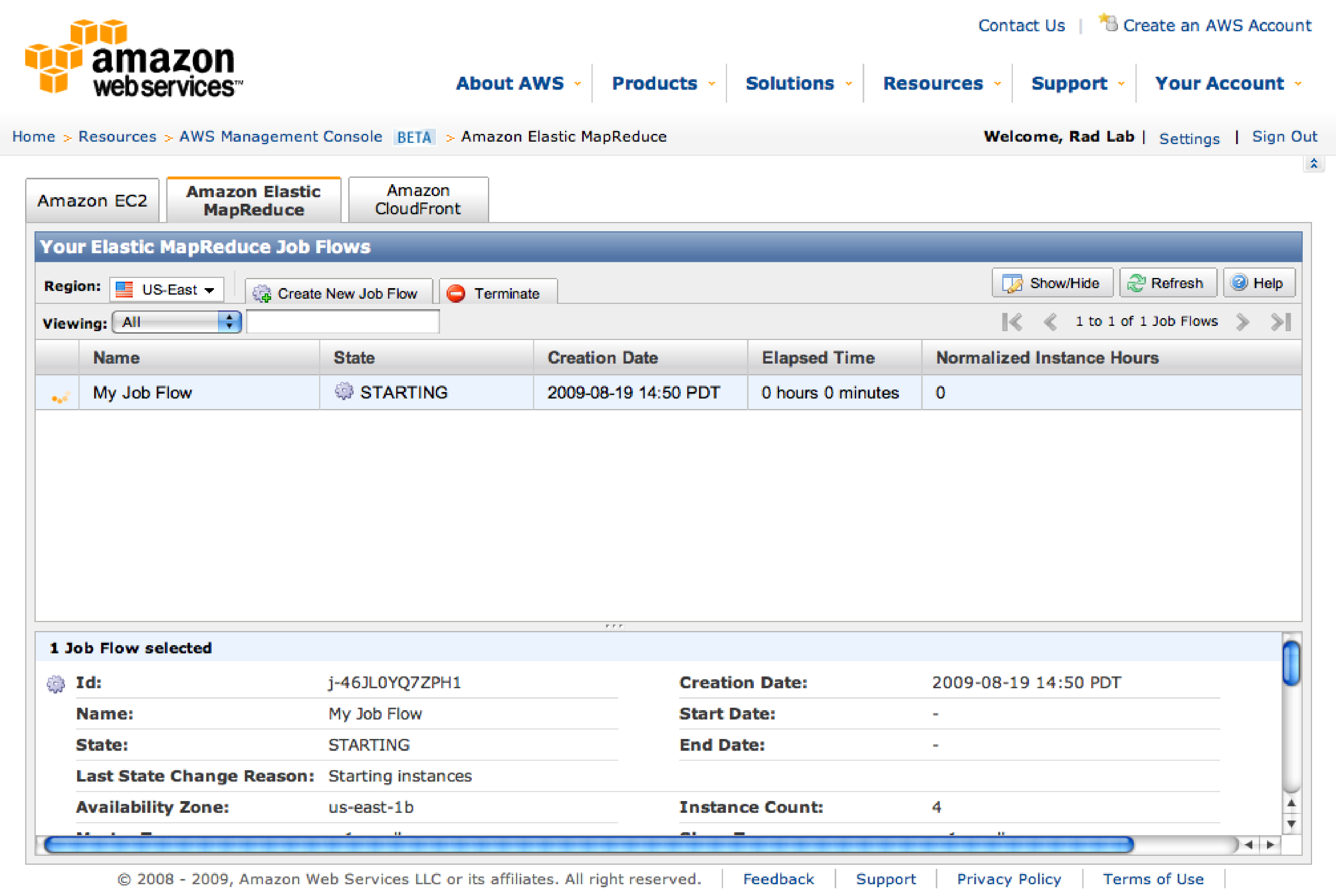
The number of EC2 instances to run in your Hadoop cluster.
If you wish to run more than 20 instances, please complete the limit request form.

Type of Instance*: Small (m1.small)

The type of EC2 instances to run in your Hadoop cluster (learn more about instance types).

* Required field

Elastic MapReduce Workflow



Your Elastic MapReduce Job Flows

Region: US-East

Viewing: All

Name	State	Creation Date	Elapsed Time	Normalized Instance Hours
My Job Flow	STARTING	2009-08-19 14:50 PDT	0 hours 0 minutes	0

1 Job Flow selected

Id:	j-46JL0YQ7ZPH1	**Creation Date:**	2009-08-19 14:50 PDT
Name:	My Job Flow	**Start Date:**	-
State:	STARTING	**End Date:**	-
Last State Change Reason:	Starting instances		
Availability Zone:	us-east-1b	**Instance Count:**	4

© 2008 - 2009, Amazon Web Services LLC or its affiliates. All right reserved. | Feedback | Support | Privacy Policy | Terms of Use

Conclusions

- MapReduce programming model hides the complexity of work distribution and fault tolerance

- Principal design philosophies:
  - *Make it scalable*, so you can throw hardware at problems
  - *Make it cheap*, lowering hardware, programming and admin costs

- MapReduce is not suitable for all problems, but when it works, it may save you quite a bit of time

- Cloud computing makes it straightforward to start using Hadoop (or other parallel software) at scale

Resources

- Hadoop: http://hadoop.apache.org/core/
- Pig: http://hadoop.apache.org/pig
- Hive: http://hadoop.apache.org/hive
- Video tutorials: http://www.cloudera.com/hadoop-training

- Amazon Web Services: http://aws.amazon.com/
- Amazon Elastic MapReduce guide: http://docs.amazonwebservices.com/ElasticMapReduce/latest/GettingStartedGuide/