Introduction to MapReduce and Hadoop

Matei Zaharia

UC Berkeley RAD Lab

matei@eecs.berkeley.edu

What is MapReduce?

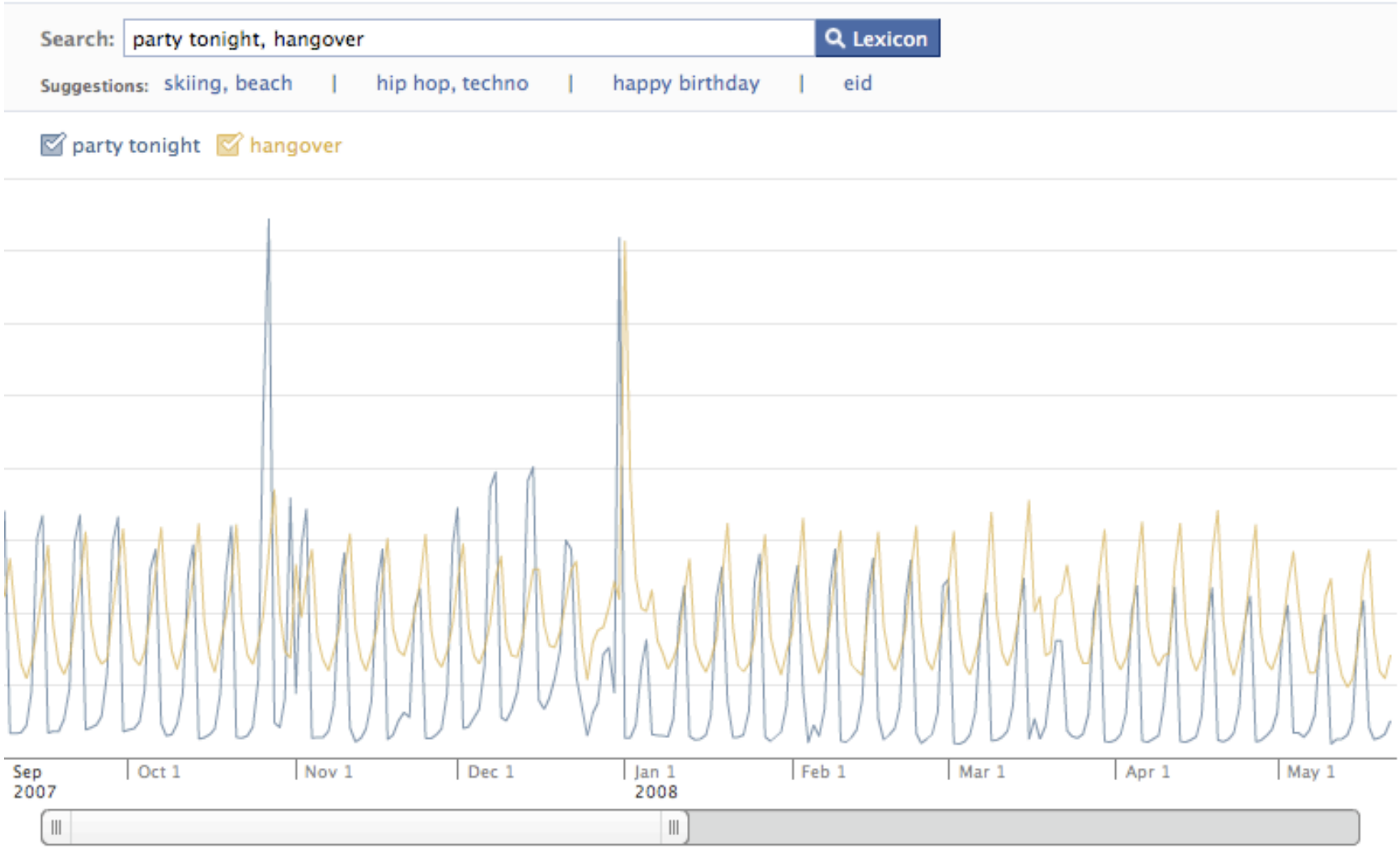
- Data-parallel programming model for clusters of commodity machines

- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by open-source Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

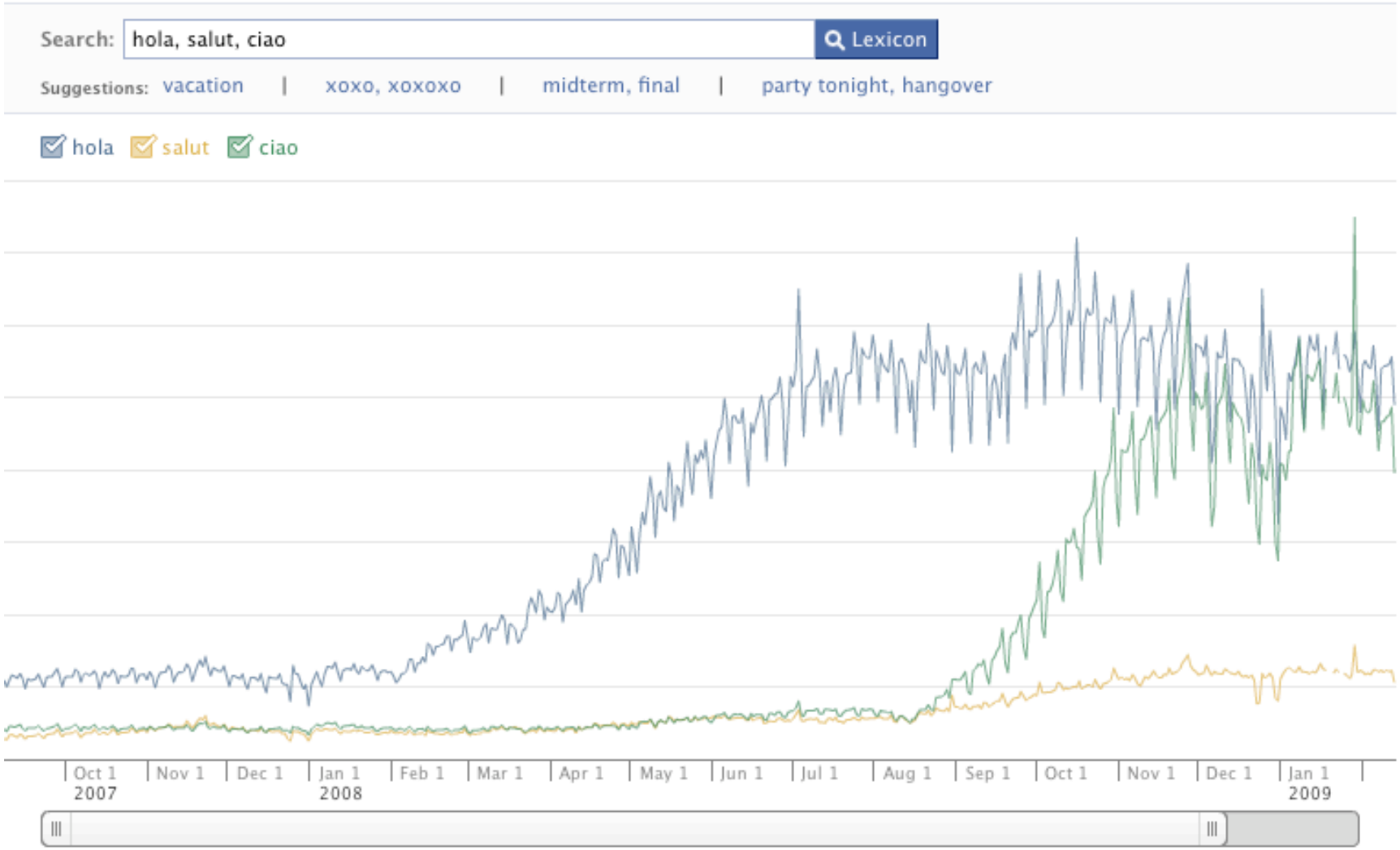
- At Google:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Example: Facebook Lexicon



www.facebook.com/lexicon

Example: Facebook Lexicon



www.facebook.com/lexicon

What is MapReduce used for?

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

Outline

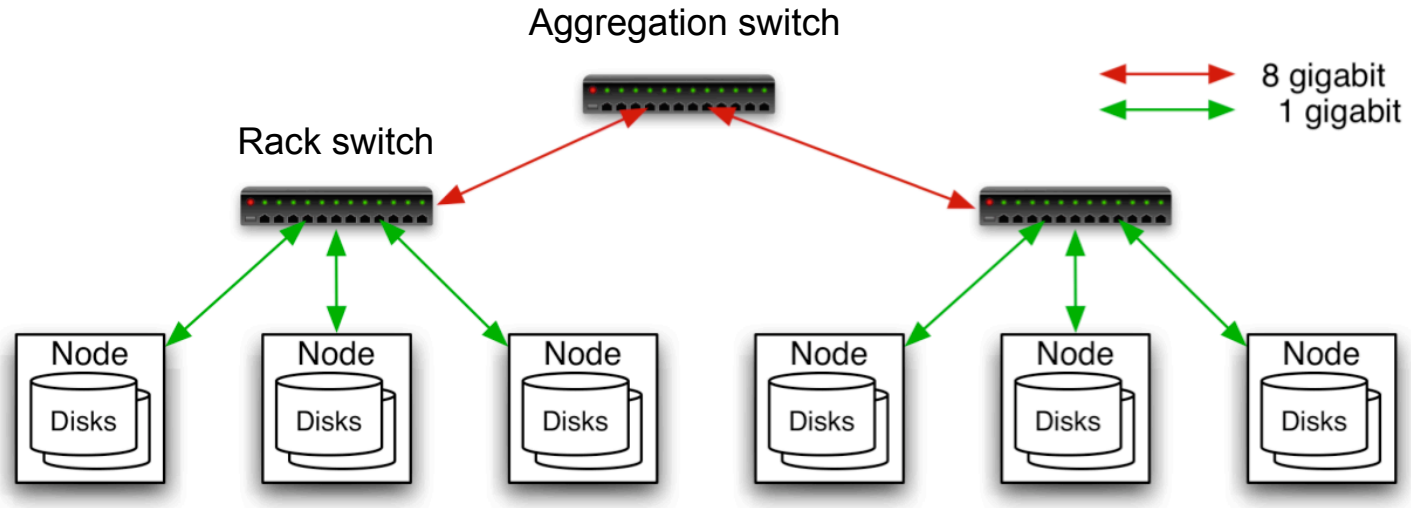
- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level queries with Pig & Hive
- Current research

MapReduce Goals

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 50 MB/s = 24 days
   - Scan on 1000-node cluster = 35 minutes

2. **Cost-efficiency:**
   - Commodity nodes (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo! terasort):
  8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Typical Hadoop Cluster

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

Challenges

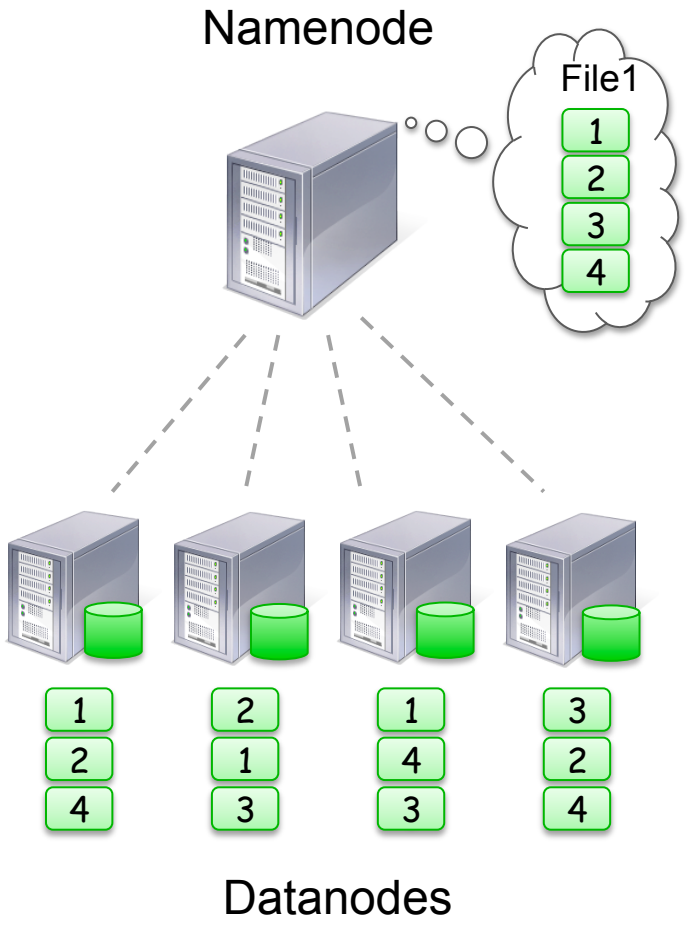
- **Cheap nodes fail, especially if you have many**
  - Mean time between failures for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - Solution: Build fault-tolerance into system
- **Commodity network = low bandwidth**
  - Solution: Push computation to the data
- **Programming distributed systems is hard**
  - Solution: Users write data-parallel “map” and “reduce” functions, system handles work distribution and 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)
- *Namenode* stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



MapReduce Programming Model

- Data type: key-value *records*

- Map function:

$$(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})$$

- Reduce function:

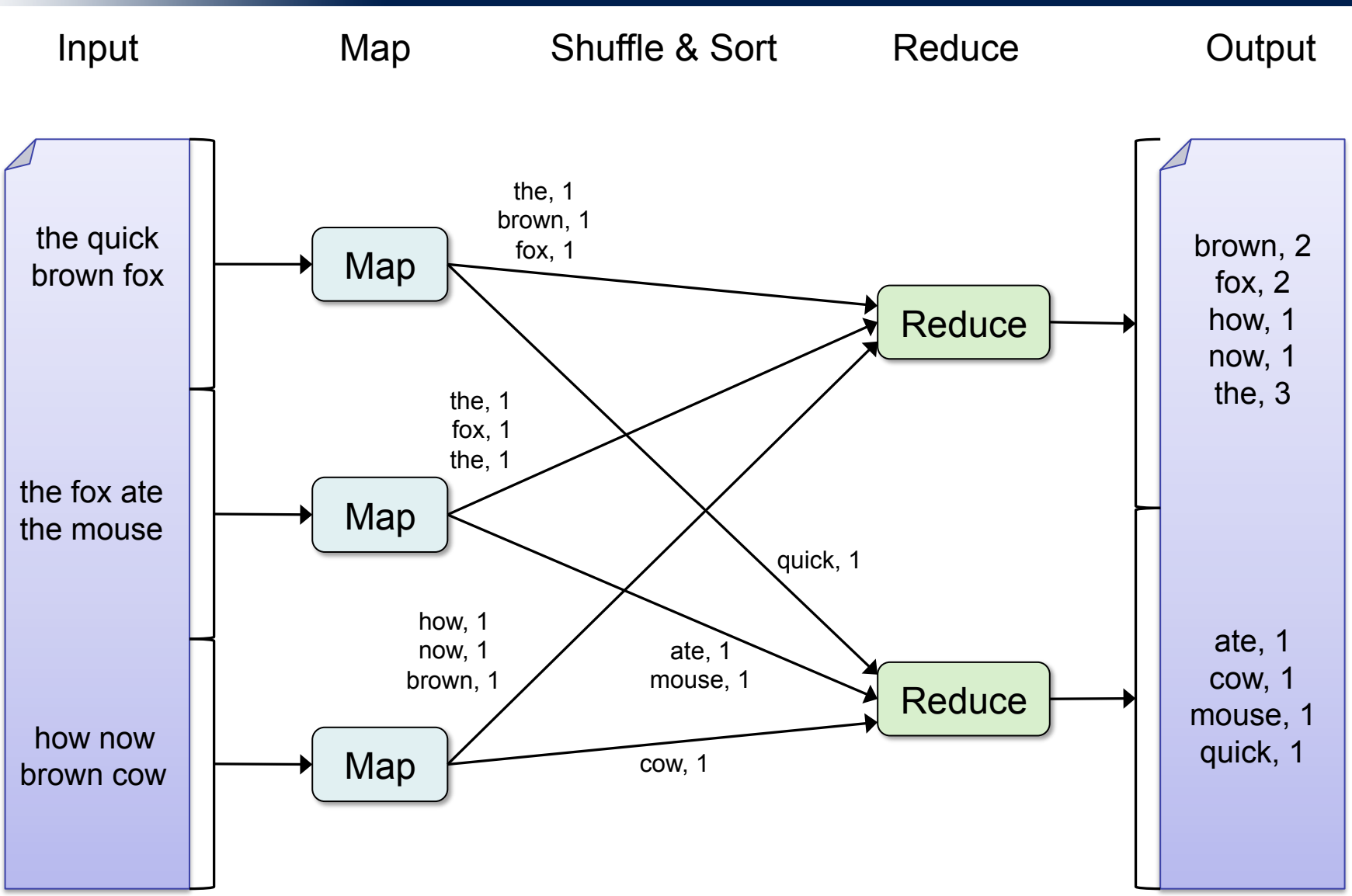
$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$

Example: Word Count

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

def reducer(key, values):
  output(key, sum(values))
```

Word Count Execution



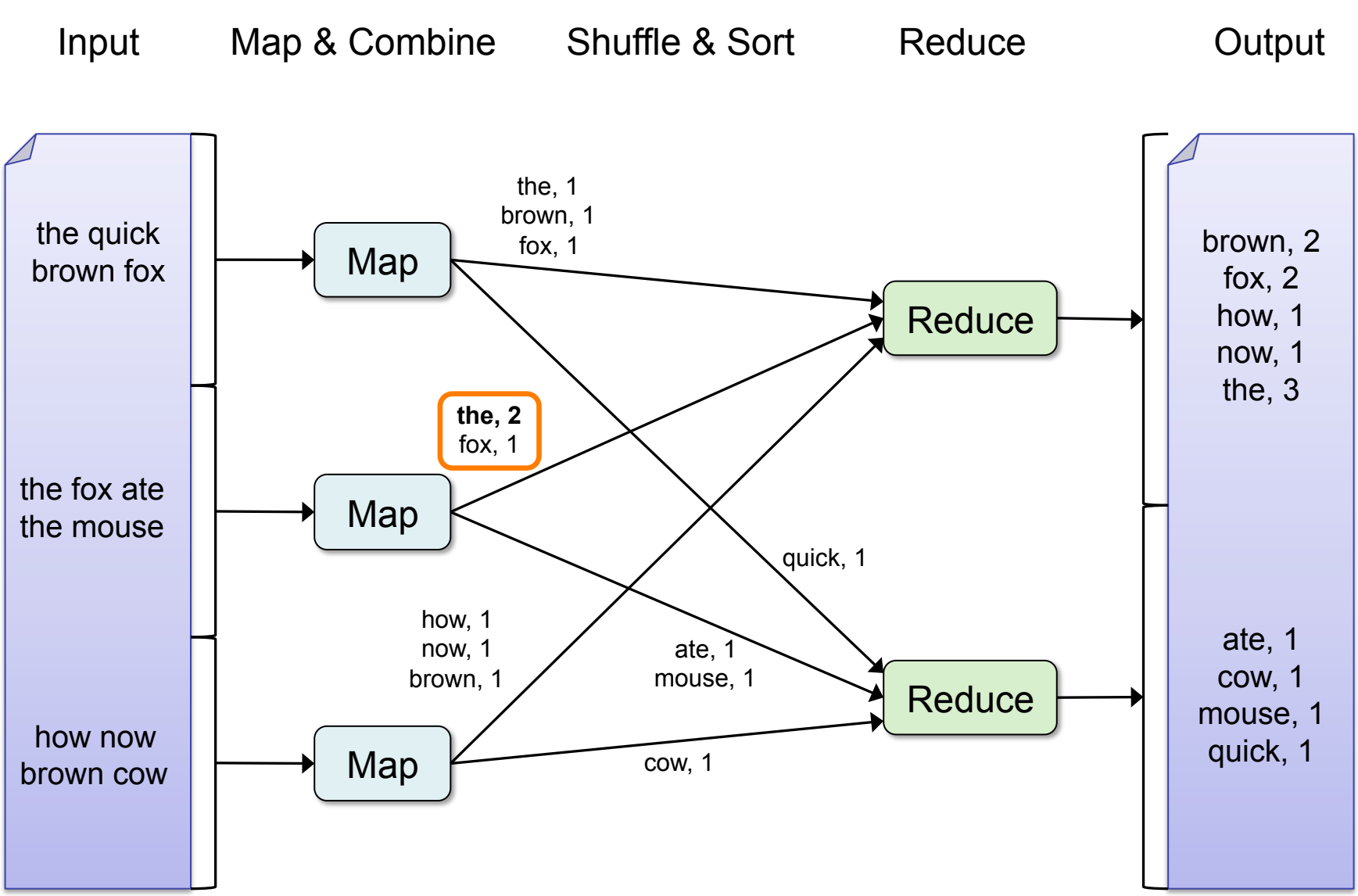
An Optimization: The Combiner

- Local aggregation function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases size of intermediate data

- Example: local counting for Word Count:

```
def combiner(key, values):
  output(key, sum(values))
```

Word Count with Combiner



MapReduce Execution Details

- Mappers preferentially placed on same node or same rack as their input block
  - Push computation to data, minimize network use
- Mappers save outputs to local disk before serving to reducers
  - Allows having more reducers than nodes
  - Allows recovery if a reducer crashes

Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     - OK for a map because it had no dependencies
     - OK for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block

➢ Note: For fault tolerance to work, *your map and reduce tasks must be side-effect-free*

Fault Tolerance in MapReduce

2. If a node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     - Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and kill the other one

- Critical for performance in large clusters ("everything that can go wrong will")

Takeaways

- By providing a data-parallel programming model, MapReduce can control job execution under the hood in useful ways:
  - Automatic division of job into tasks
  - Placement of computation near data
  - Load balancing
  - Recovery from failures & stragglers

Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level queries with Pig & Hive
- Current research

1. Search

- **Input:** (lineNumber, line) records
- **Output:** lines matching a given pattern

- **Map:**

```
if(line matches pattern):
    output(line)
```

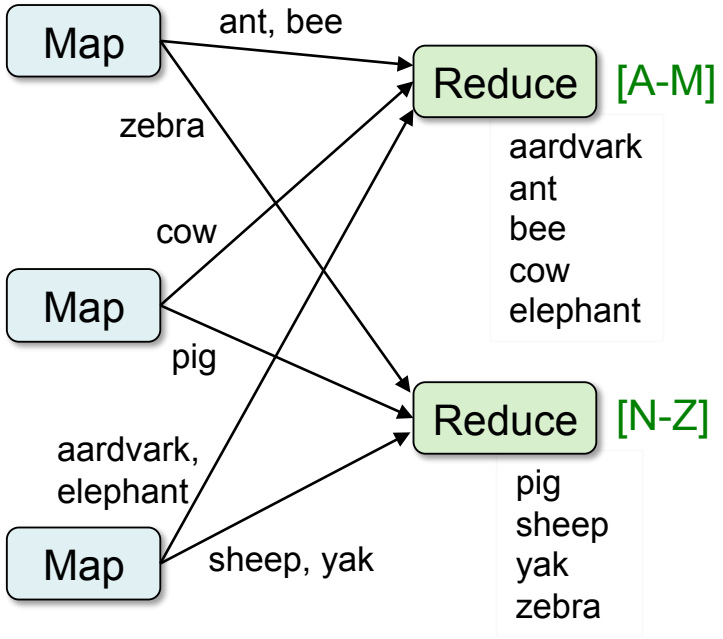
- **Reduce:** identify function
  - Alternative: no reducer (map-only job)

2. Sort

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick partitioning function h such that $k_1<k_2 \Rightarrow h(k_1)<h(k_2)$



3. Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

- **Map:**

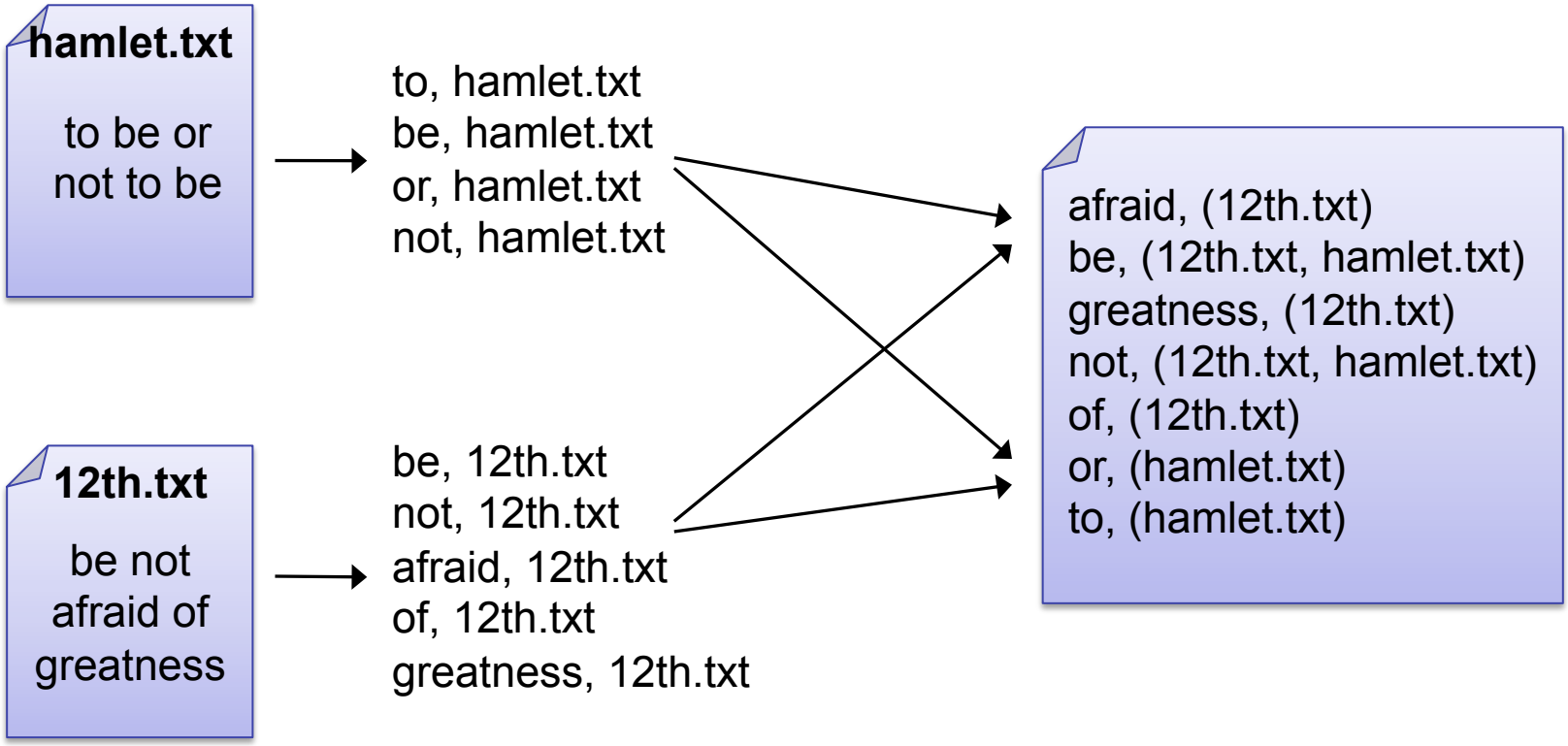
```
foreach word in text.split():
    output(word, filename)
```

- **Combine:** uniquify filenames for each word

- **Reduce:**

```
def reduce(word, filenames):
    output(word, sort(filenames))
```

Inverted Index Example



4. Most Popular Words

- **Input:** (filename, text) records
- **Output:** the 100 words occurring in most files

- Two-stage solution:
  - **Job 1:**
    - Create inverted index, giving (word, list(file)) records
  - **Job 2:**
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job

- Optimizations:
  - Map to (word, 1) instead of (word, file) in Job 1
  - Estimate count distribution in advance by sampling

5. Numerical Integration

- **Input:** (start, end) records for sub-ranges to integrate
  - Doable using custom InputFormat
- **Output:** integral of f(x) dx over entire range

- **Map:**

```
def map(start, end):
    sum = 0
    for(x = start; x < end; x += step):
        sum += f(x) * step
    output(“”, sum)
```

- **Reduce:**

```
def reduce(key, values):
    output(key, sum(values))
```

Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level queries with Pig & Hive
- Current research

Getting Started with Hadoop

- Download from hadoop.apache.org
- To install locally, unzip and set `JAVA_HOME`
- Guide: hadoop.apache.org/common/docs/current/quickstart.html

- Three ways to write jobs:
  - Java API
  - Hadoop Streaming (for Python, Perl, etc)
  - Pipes API (C++)

Word Count in Java

```
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable ONE = new IntWritable(1);

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        output.collect(new text(itr.nextToken()), ONE);
      }
    }
  }
```

Word Count in Java

```
public static class Reduce extends MapReduceBase
   implements Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
     int sum = 0;
     while (values.hasNext()) {
       sum += values.next().get();
     }
     output.collect(key, new IntWritable(sum));
   }
}
```

Word Count in Java

```
public static void main(String[] args) throws Exception {
   JobConf conf = new JobConf(WordCount.class);
   conf.setJobName("wordcount");

   conf.setMapperClass(MapClass.class);
   conf.setCombinerClass(Reduce.class);
   conf.setReducerClass(Reduce.class);

   FileInputFormat.setInputPaths(conf, args[0]);
   FileOutputFormat.setOutputPath(conf, new Path(args[1]));

   conf.setOutputKeyClass(Text.class); // out keys are words (strings)
   conf.setOutputValueClass(IntWritable.class); // values are counts

   JobClient.runJob(conf);
 }
```

Word Count in Python with Hadoop Streaming

Mapper.py:

```
import sys
for line in sys.stdin:
  for word in line.split():
    print(word.lower() + "\t" + 1)
```

Reducer.py:

```
import sys
counts = {}
for line in sys.stdin:
  word, count = line.split("\t")
    dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
  print(word.lower() + "\t" + 1)
```

Amazon Elastic MapReduce

- Web interface and command-line tools for running Hadoop jobs on EC2
- Data stored in Amazon S3
- Monitors job and shuts machines after use

- If you want more control, you can launch a Hadoop cluster manually using scripts in `src/contrib/ec2`

Elastic MapReduce UI

Create a New Job Flow Cancel

DEFINE JOB FLOW — SPECIFY PARAMETERS — CONFIGURE EC2 INSTANCES — REVIEW

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

Continue

* Required field

Elastic MapReduce UI

Create a New Job Flow Cancel

DEFINE JOB FLOW ✓ — SPECIFY PARAMETERS ✓ — CONFIGURE EC2 INSTANCES — REVIEW

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.

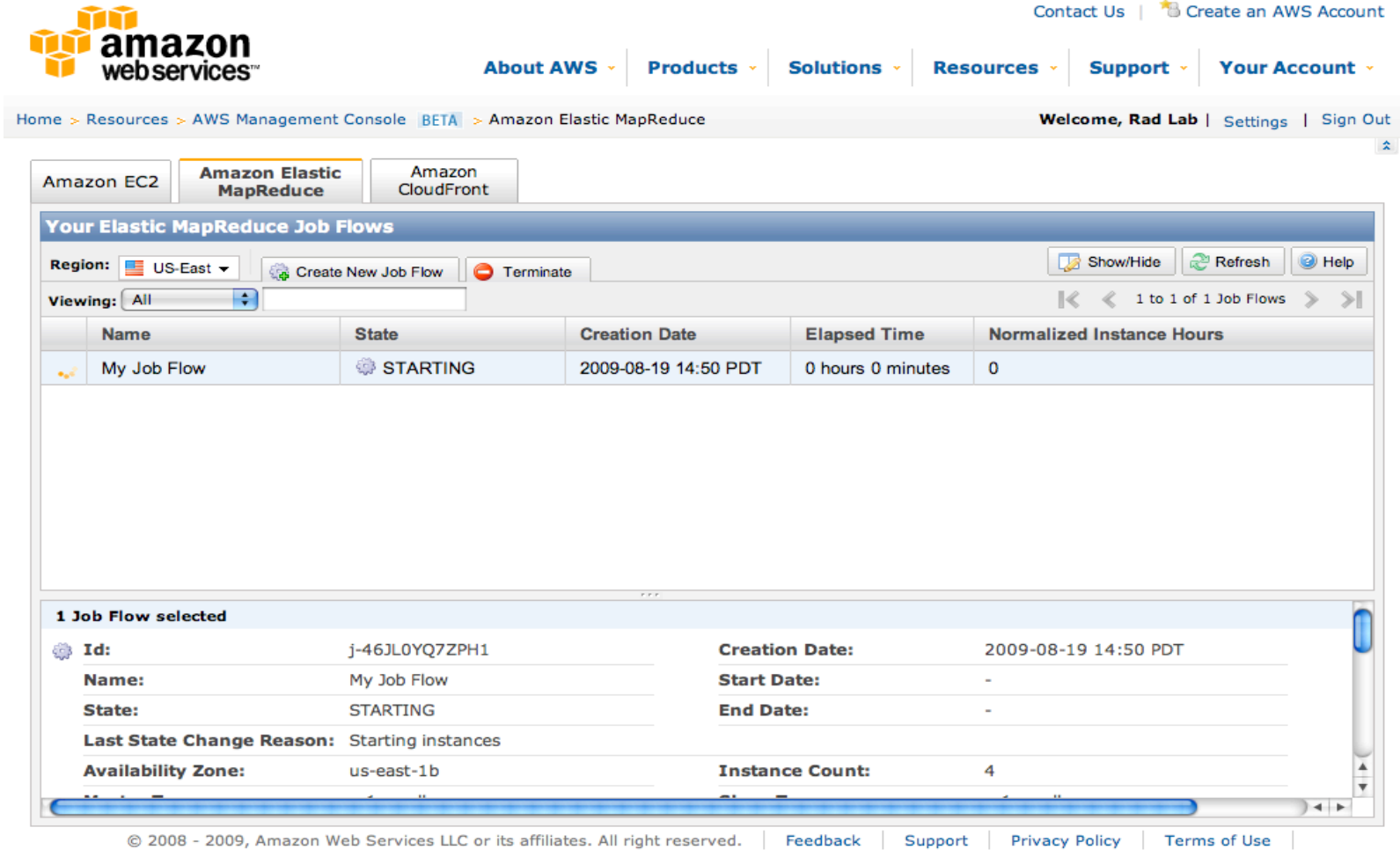
Type of Instance*: Small (m1.small)

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

Show advanced options

‹ Back Continue * Required field

Elastic MapReduce UI



Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level queries with Pig & Hive
- Current research

Motivation

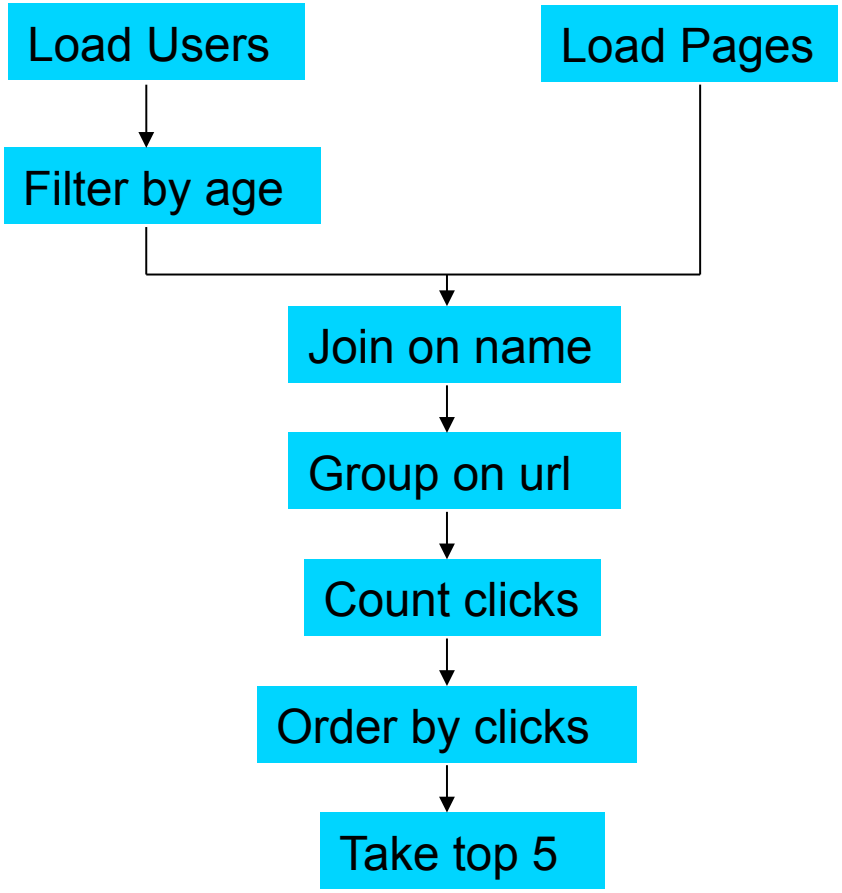
- MapReduce is great, as many algorithms can be expressed by a series of MR jobs
- But it's low-level: must think about keys, values, partitioning, etc
- Can we capture common "job patterns"?

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

An Example Problem

Suppose you have user data in one file, website 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.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
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(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        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

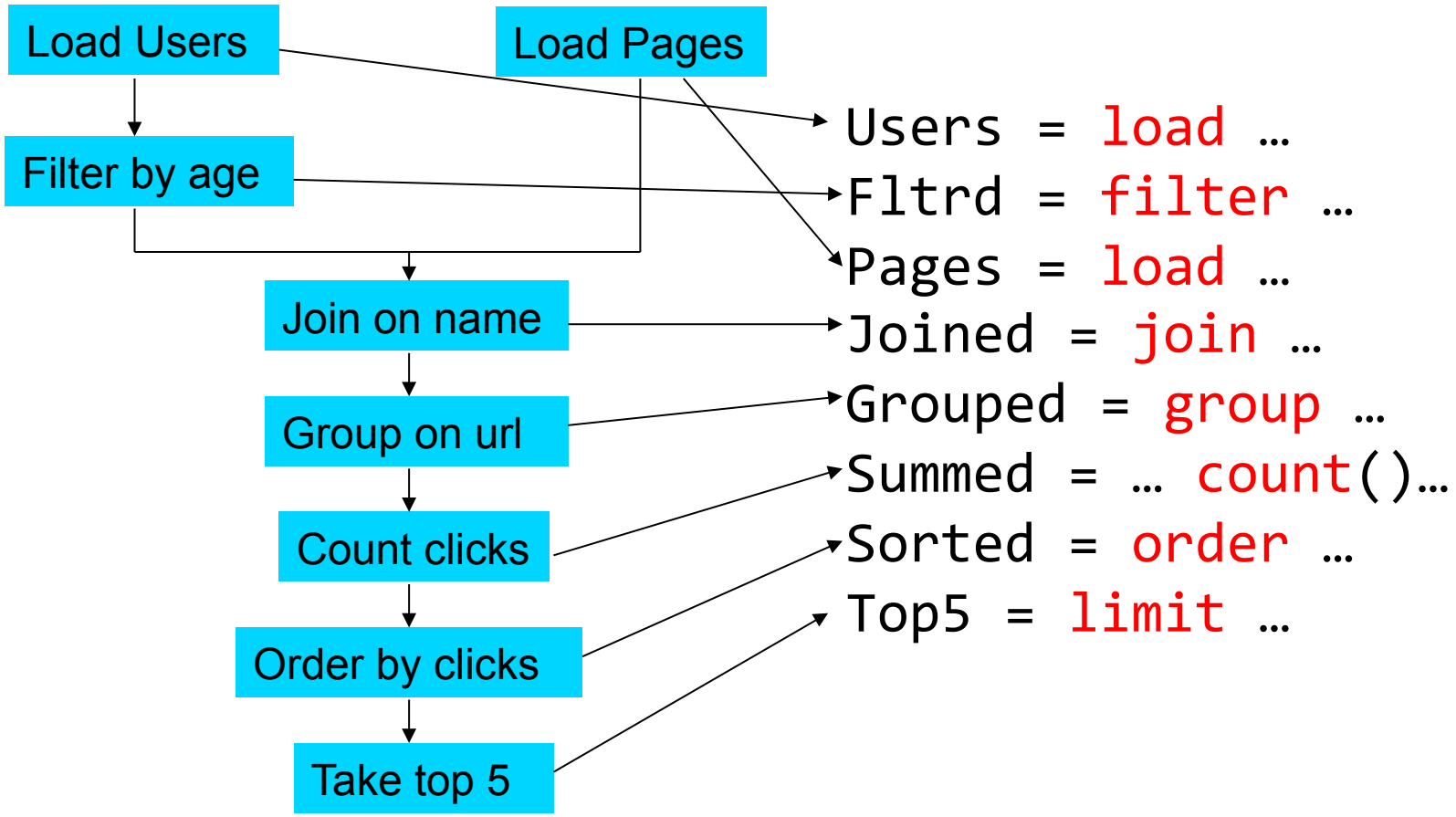
```
Users    = load ‘users’ as (name, age);
Filtered = filter Users by
                  age >= 18 and age <= 25;
Pages    = load ‘pages’ as (user, url);
Joined   = join Filtered by name, Pages by user;
Grouped  = group Joined by url;
Summed   = foreach Grouped generate group,
                   count(Joined)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

Translation to MapReduce

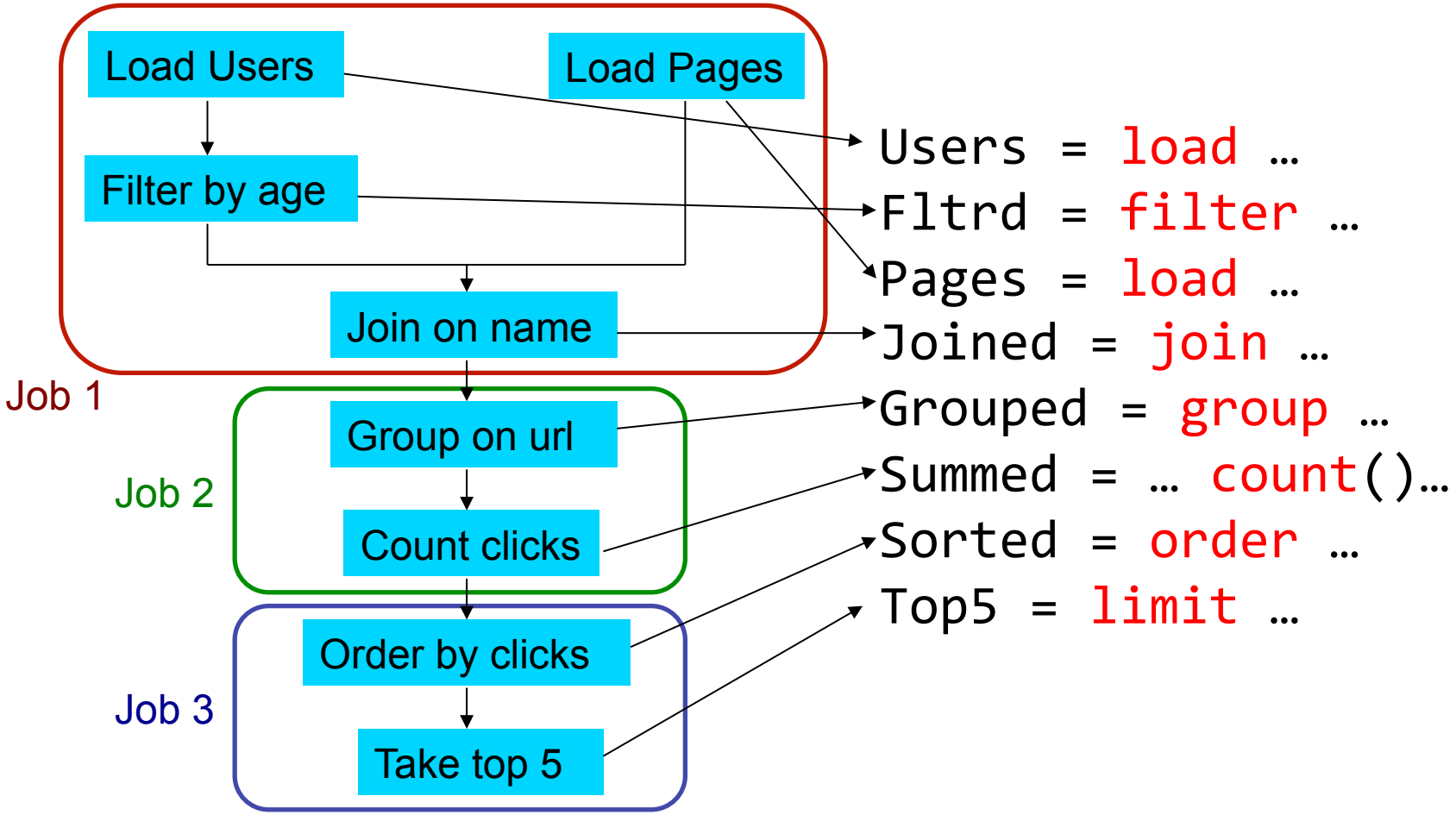
Notice how naturally the components of the job translate into Pig Latin.



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

Translation to MapReduce

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



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

Hive

- Developed at Facebook
- Used for most Facebook jobs
- “Relational database” built on Hadoop
  - Maintains table schemas
  - SQL-like query language (which can also call Hadoop Streaming scripts)
  - Supports table partitioning, complex data types, sampling, 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;
```

Conclusions

- MapReduce's data-parallel programming model hides complexity of distribution and fault tolerance
- Principal philosophies:
  - *Make it scale*, so you can throw hardware at problems
  - *Make it cheap*, saving hardware, programmer and administration costs (but requiring fault tolerance)
- Hive and Pig further simplify programming
- MapReduce is not suitable for all problems, but when it works, it may save you a lot of time

Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level queries with Pig & Hive
- Current research

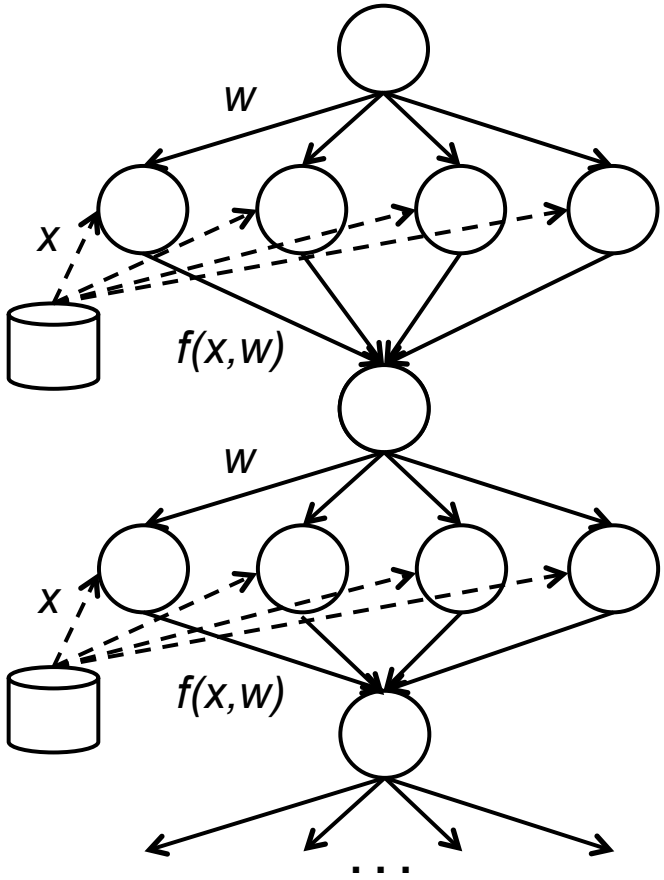
Cluster Computing Research

- New execution models
  - Dryad (Microsoft): DAG of tasks
  - Pregel (Google): bulk synchronous processes
  - MapReduce Online (Berkeley): streaming
- Easier programming
  - DryadLINQ (MSR): language-integrated queries
  - SEJITS (Berkeley): specializing Python/Ruby
- Improving efficiency/scheduling/etc

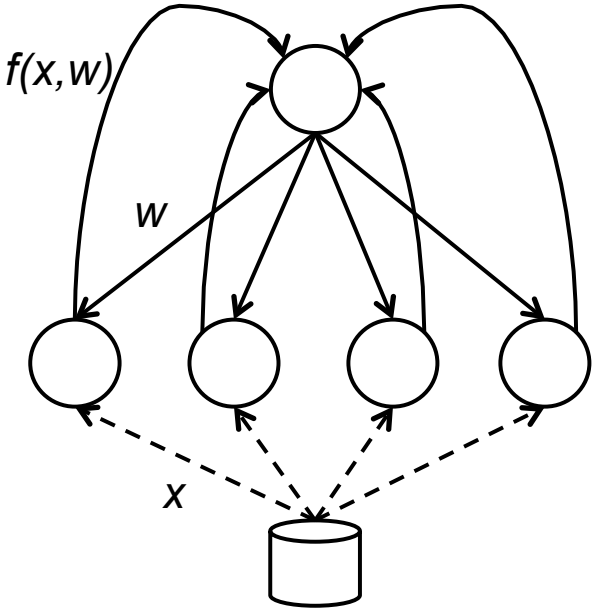
Self-Serving Example: Spark

- **Motivation:** iterative jobs (common in machine learning, optimization, etc)
- **Problem:** iterative jobs reuse the same data over and over, but MapReduce / Dryad / etc require acyclic data flows
- **Solution:** support “caching” data between parallel operations.. but remain fault-tolerant
- Also experiment with language integration etc

Data Flow



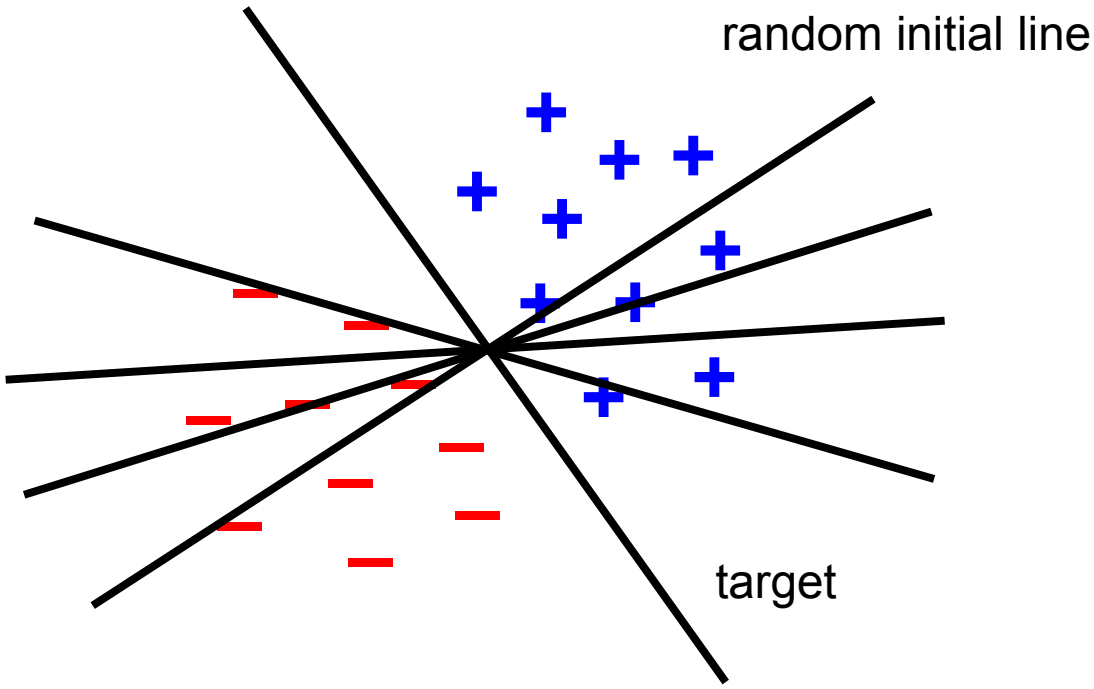
MapReduce



Spark

Example: Logistic Regression

Goal: find best line separating 2 datasets



Serial Version

```
val data = readData(...)

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = Vector.zeros(D)
  for (p <- data) {
    val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
    gradient += scale * p.x
  }
  w -= gradient
}

println("Final w: " + w)
```

Spark Version

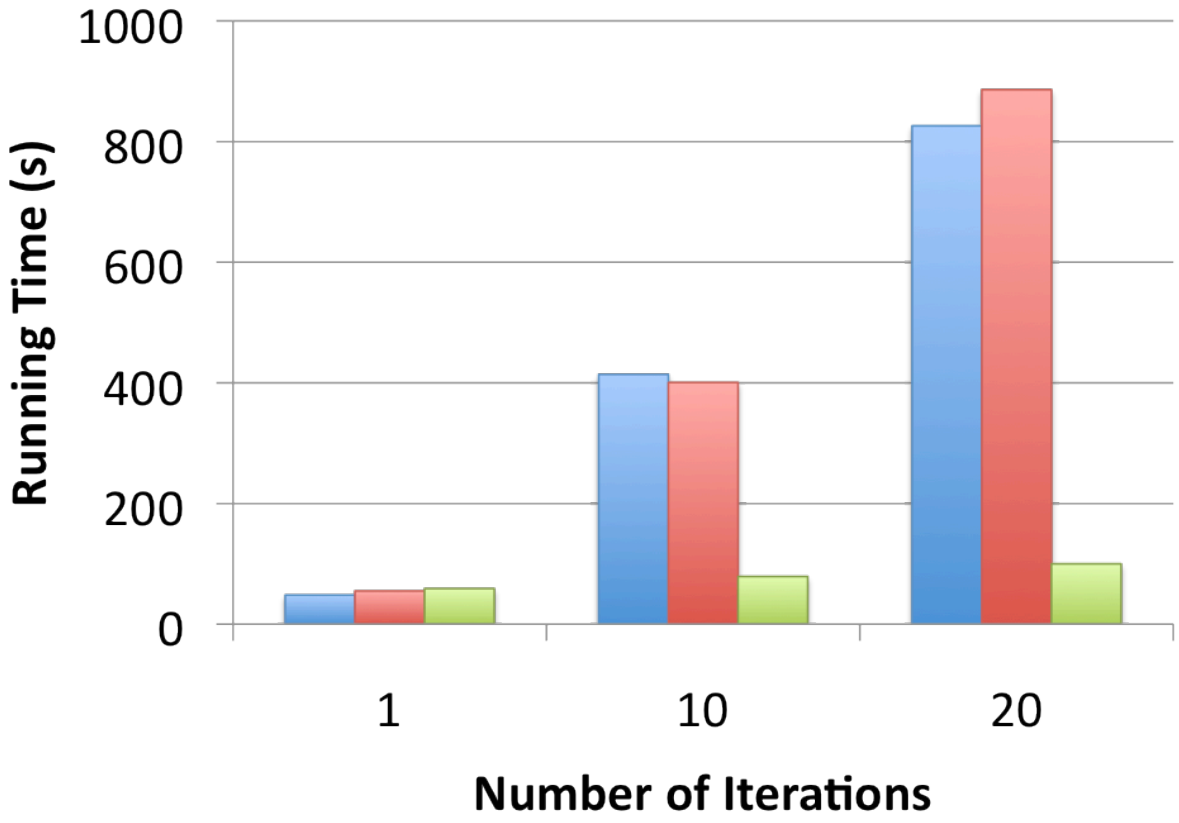
```
val data = spark.hdfsTextFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator(Vector.zeros(D))
  for (p <- data) {
    val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
    gradient += scale * p.x
  }
  w -= gradient.value
}

println("Final w: " + w)
```

Performance



40s / iteration

Hadoop

Hadoop on Nexus

Spark on Nexus

first iteration 60s
further iterations 2s

Crazy Idea: Interactive Spark

- Being able to cache datasets in memory is great for interactive analysis: extract a working set, cache it, query it repeatedly
- Modified Scala interpreter to support interactive use of Spark
- Result: can search Wikipedia in ~0.5s after a ~20-second initial load
- Still figuring out how this should evolve

Resources

- Hadoop: http://hadoop.apache.org/common
- Pig: http://hadoop.apache.org/pig
- Hive: http://hadoop.apache.org/hive

- Video tutorials: www.cloudera.com/hadoop-training

- Amazon Elastic MapReduce: http://docs.amazonwebservices.com/ElasticMapReduce/latest/GettingStartedGuide/