Cloud Computing with MapReduce and Hadoop

Matei Zaharia

UC Berkeley AMP Lab

matei@berkeley.edu

What is Cloud Computing?

- "Cloud" refers to large Internet services running on 10,000s of machines (Google, Facebook, etc)

- "Cloud computing" refers to services by these companies that let <u>external customers</u> rent cycles
  - Amazon EC2: virtual machines at 8¢/hour, billed hourly
  - Amazon S3: storage at 12.5¢/GB/month
  - Windows Azure: applications using Azure API

- Attractive features:
  - Scale: 100s of nodes available in minutes
  - Fine-grained billing: pay only for what you use
  - Ease of use: sign up with credit card, get root access

What is MapReduce?

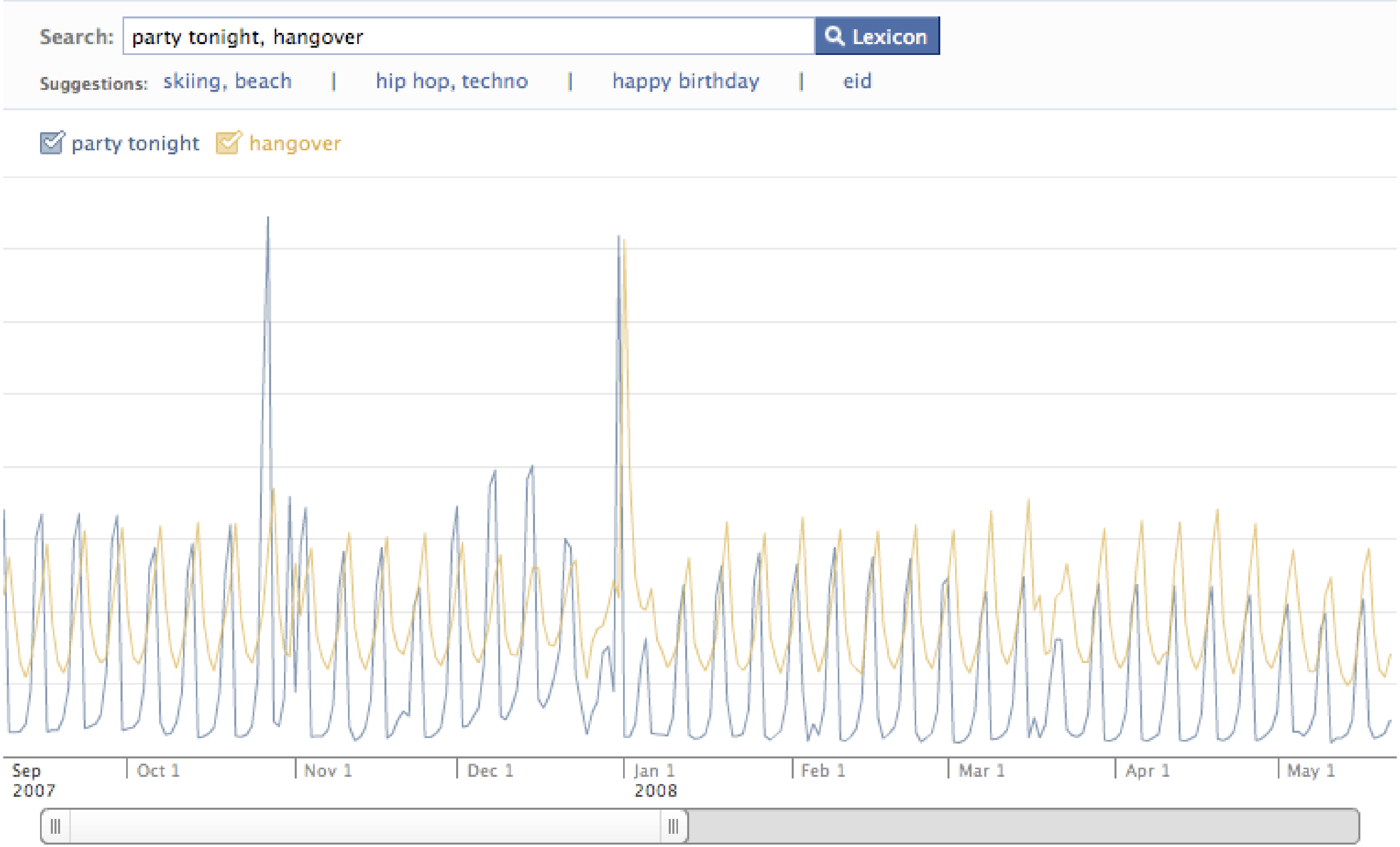
- Programming model for data-intensive computing on commodity clusters

- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache 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



Search: party tonight, hangover

🔍 Lexicon
Suggestions: skiing, beach | hip hop, techno | happy birthday | eid
☑ party tonight ☑ hangover
Sep 2007
Oct 1
Nov 1
Dec 1
Jan 1 2008
Feb 1
Mar 1
Apr 1
May 1

www.facebook.com/lexicon

What is MapReduce Used For?

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

Outline

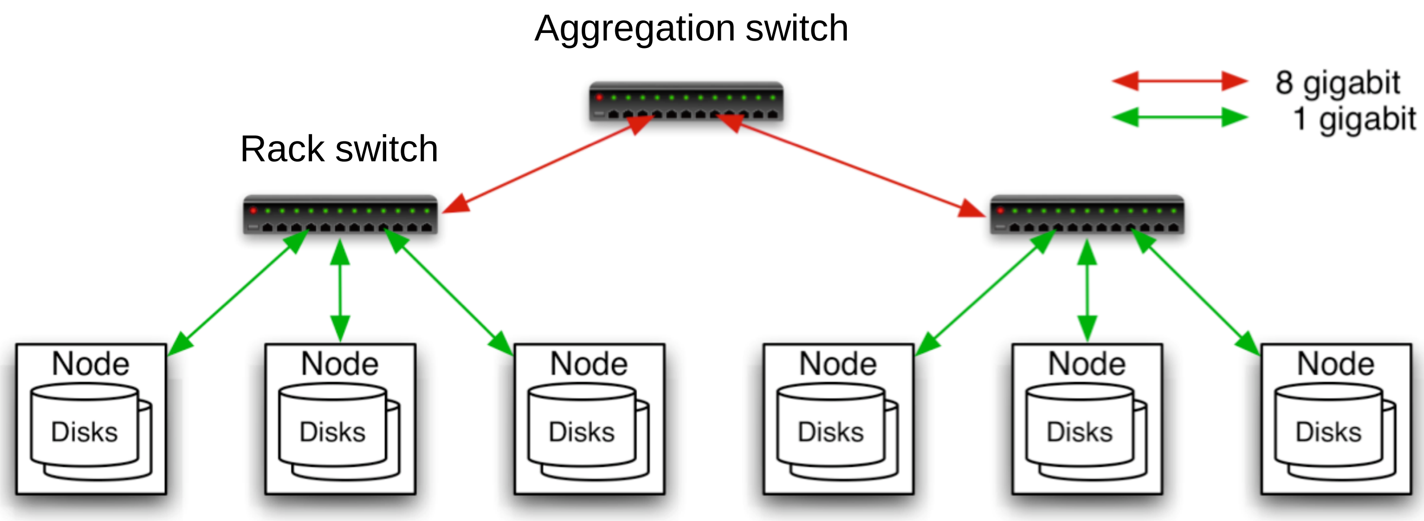
- MapReduce architecture
- Sample applications
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

MapReduce Goals

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

- **Cost-efficiency:**
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - 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 (Facebook):
  8-16 cores, 32 GB RAM, 8 × 1.5 TB disks, no RAID

Typical Hadoop Cluster

Challenges of Cloud Environment

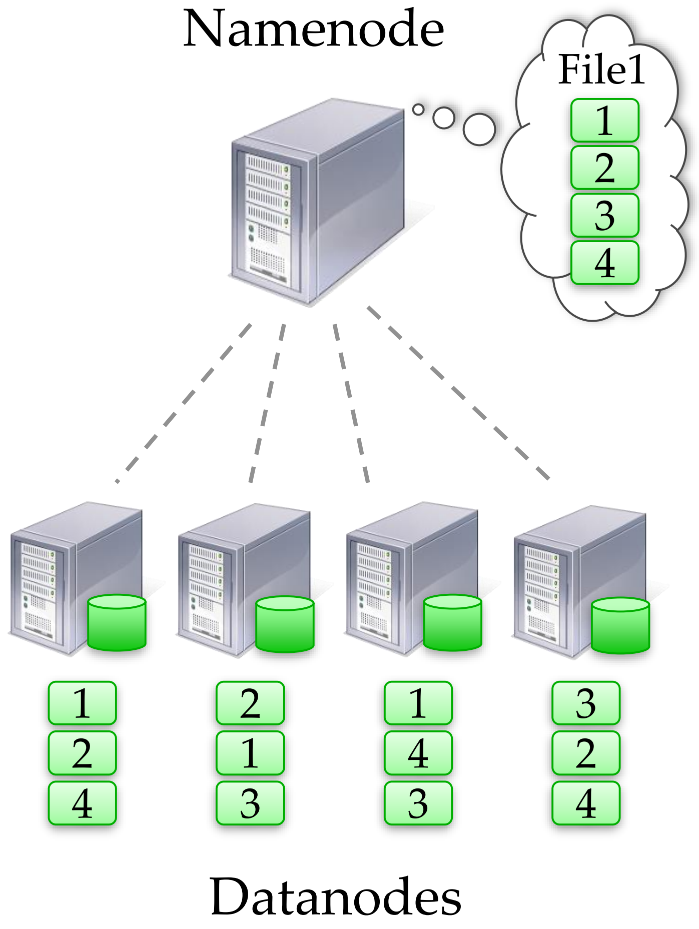
- Cheap nodes fail, especially when you have many
  - Mean time between failures for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - **Solution:** Build fault tolerance into system
- Commodity network = low bandwidth
  - **Solution:** Push computation to the data
- Programming distributed systems is hard
  - **Solution:** Restricted programming model: users write data-parallel "map" and "reduce" functions, system handles work distribution and failures

Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- MapReduce framework
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Colocated with file system

Hadoop Distributed File System

- Files split into 128MB blocks
- Blocks replicated across several datanodes (often 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:

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

- Reduce function:

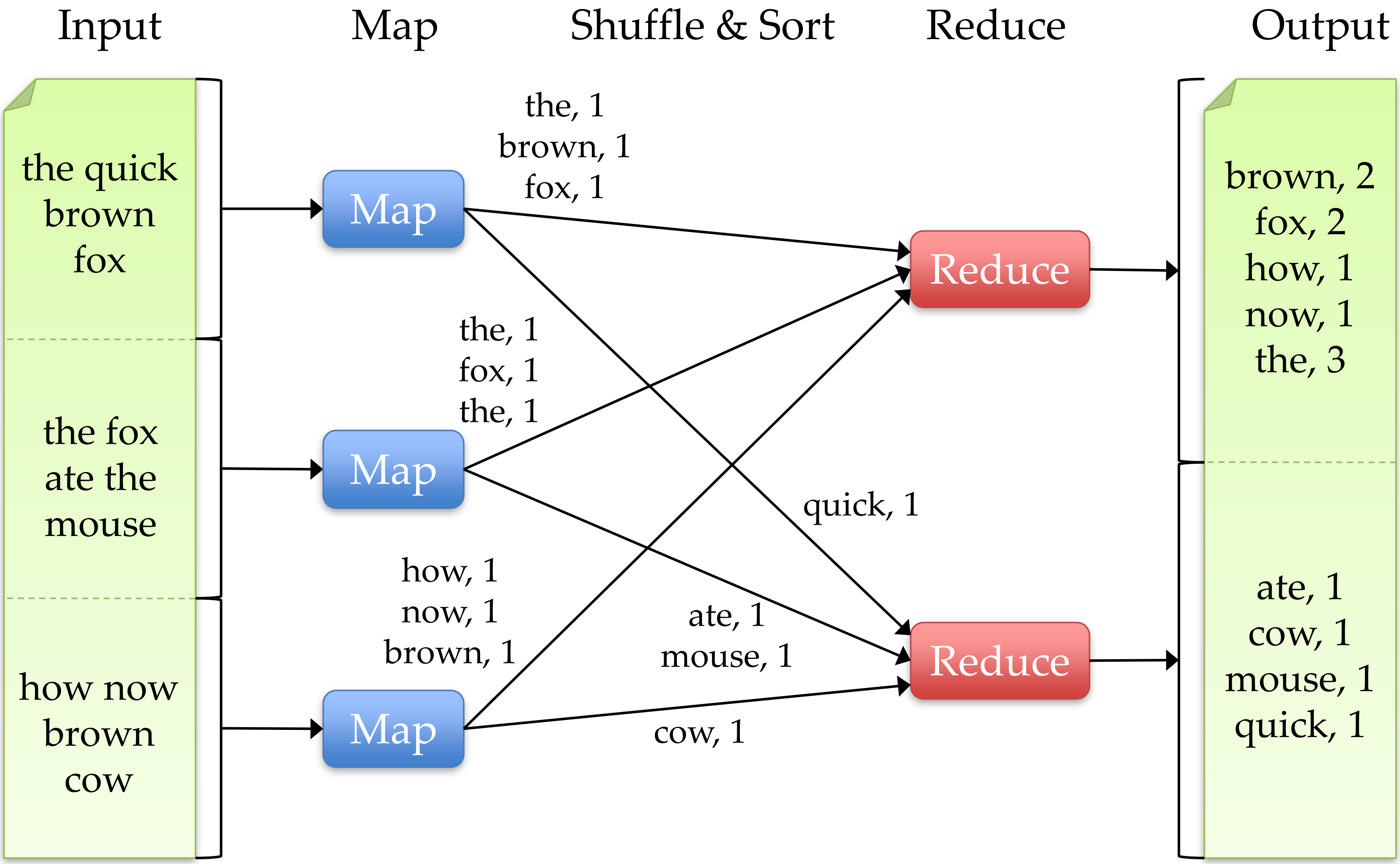
$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

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



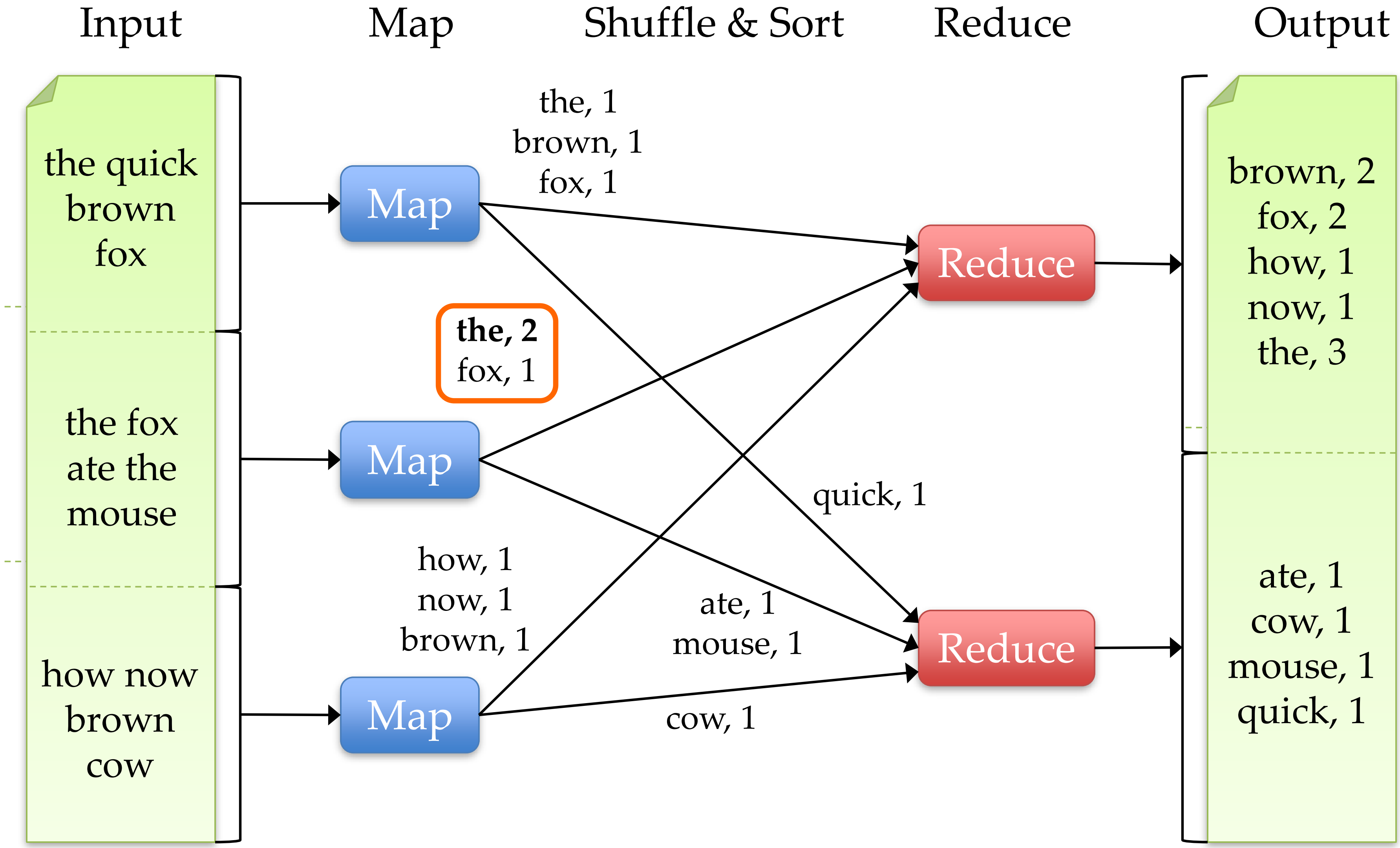
An Optimization: The Combiner

- Local reduce function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases amount 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 scheduled on same node or same rack as their input block
  - Minimize network use to improve performance

- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

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 the fault tolerance to work, *user tasks must be deterministic and 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 (many possible causes of stragglers)

Takeaways

- By providing a restricted data-parallel programming model, MapReduce can control job execution 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
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

1. Search

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

- **Map:**

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

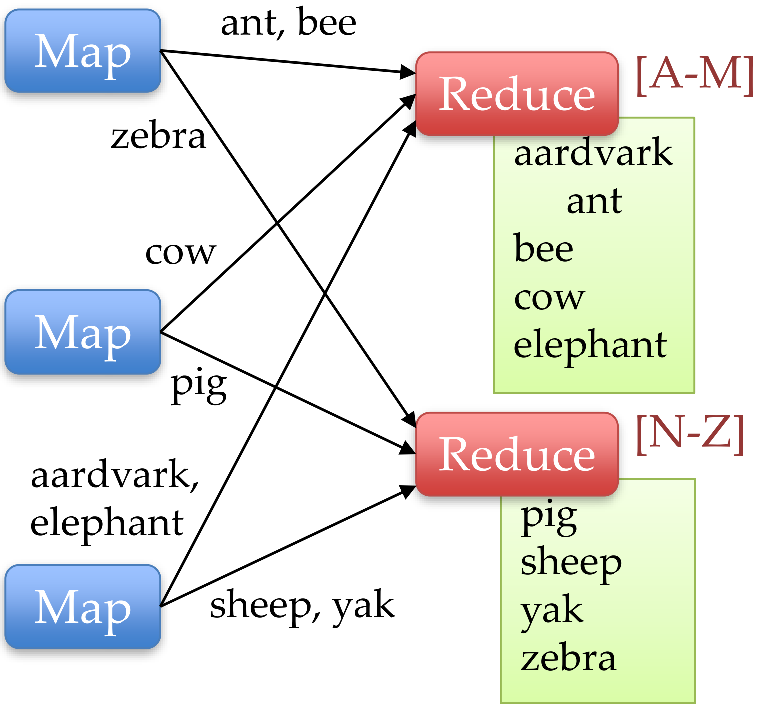
- **Reduce:** identity function
  - – Alternative: no reducer (map-only job)

2. Sort

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

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

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



3. Inverted Index

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

- **Map:**

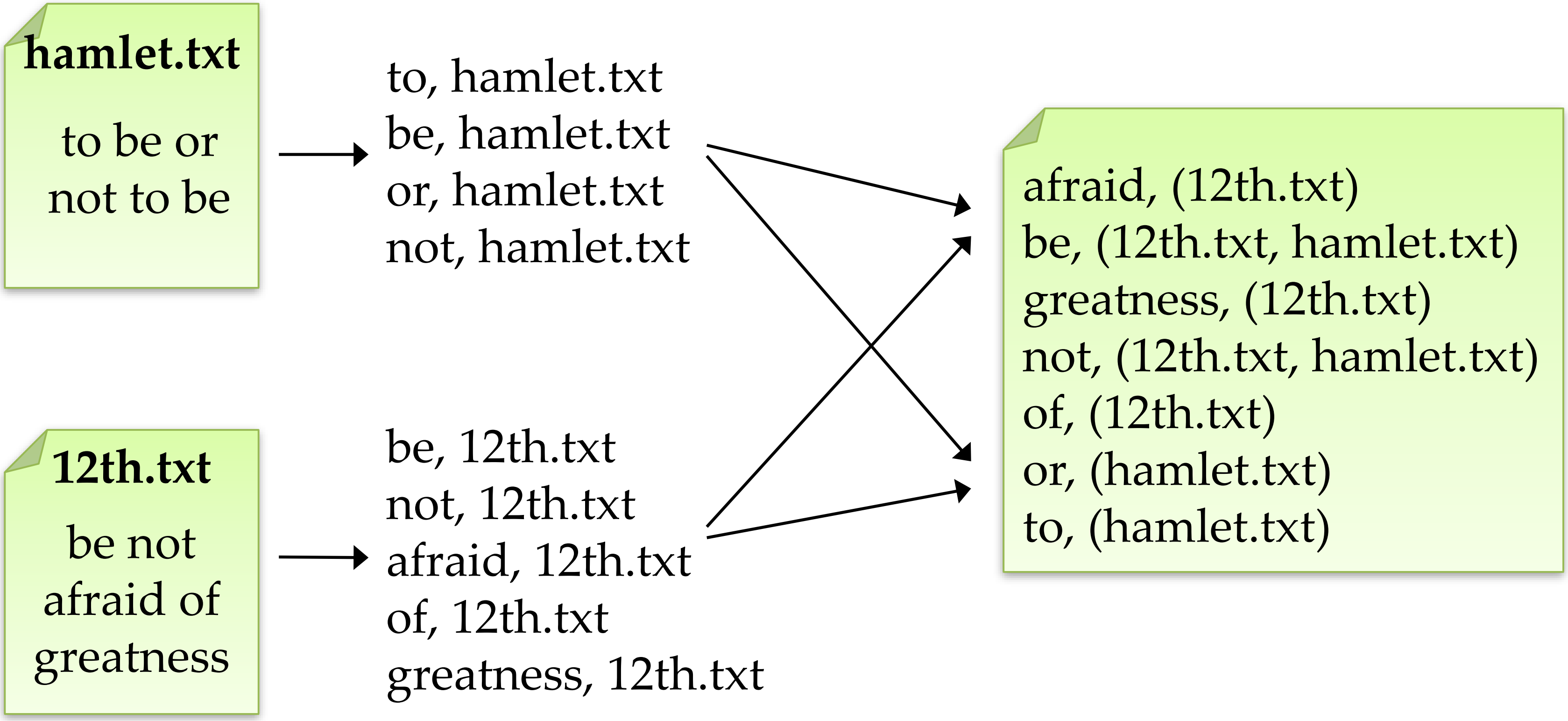
```
foreach word in text.split():
   output(word, filename)
```

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

- **Reduce:**

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

Inverted Index Example



hamlet.txt

to be or
not to be

to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

12th.txt

be not
afraid of
greatness

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

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
  - Can implement using custom InputFormat
- **Output:** integral of *f*(*x*) 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
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

Introduction to Hadoop

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

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

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)
```

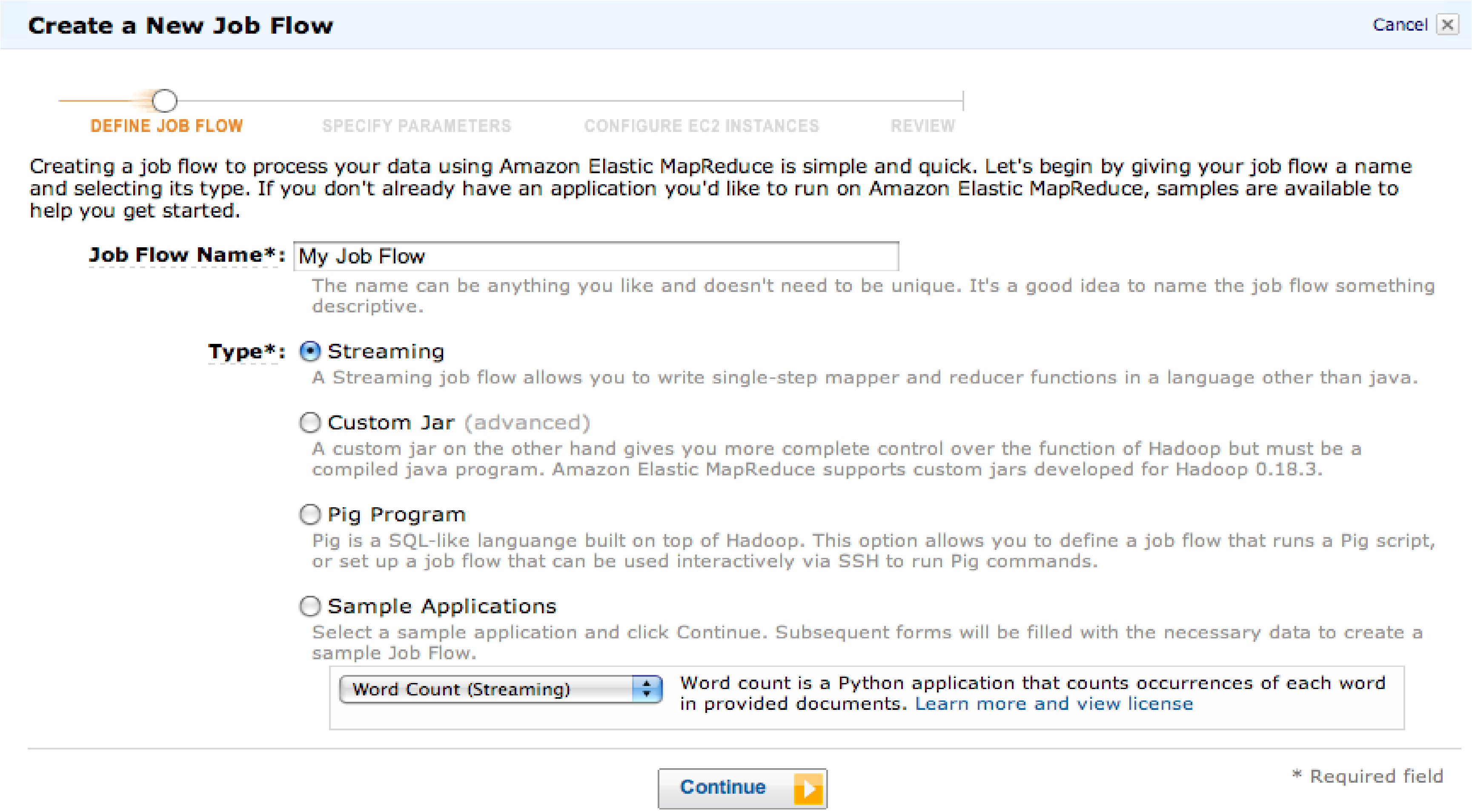
Reducer.py:

```
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

Elastic MapReduce UI



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

* Required field

Outline

- MapReduce architecture
- Sample applications
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

Motivation

- MapReduce is powerful: many algorithms can be expressed as a series of MR jobs

- But it’s fairly low-level: must think about keys, values, partitioning, etc.

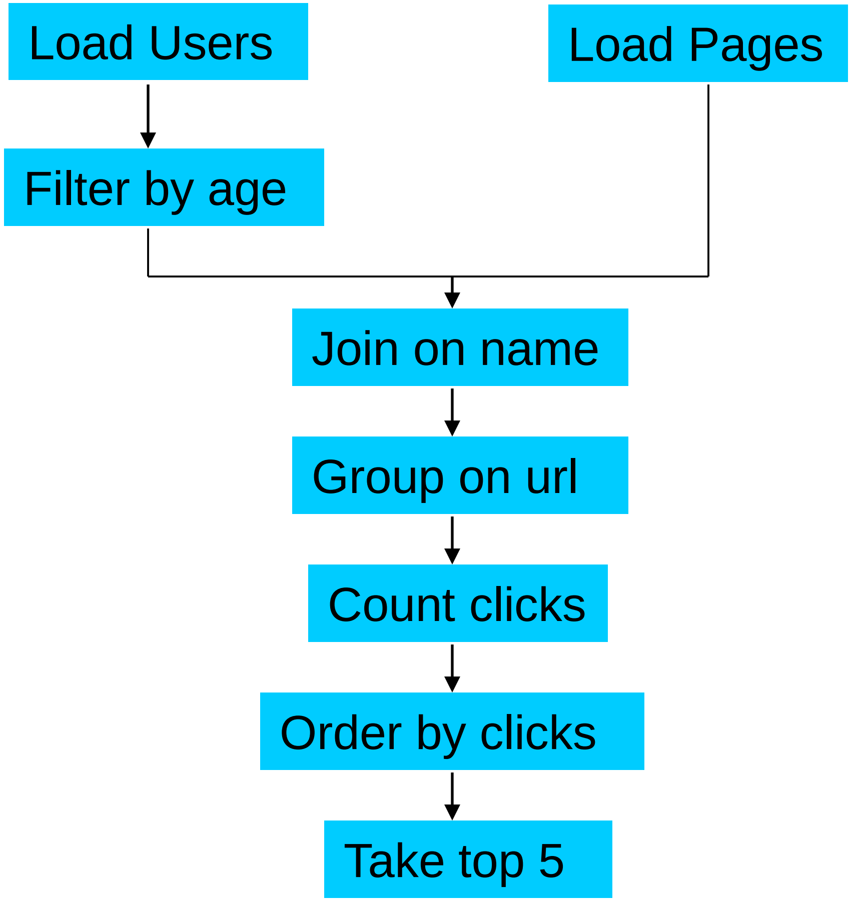
- Can we capture common “job patterns”?

Pig

- Started at Yahoo! Research
- Runs about 50% 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

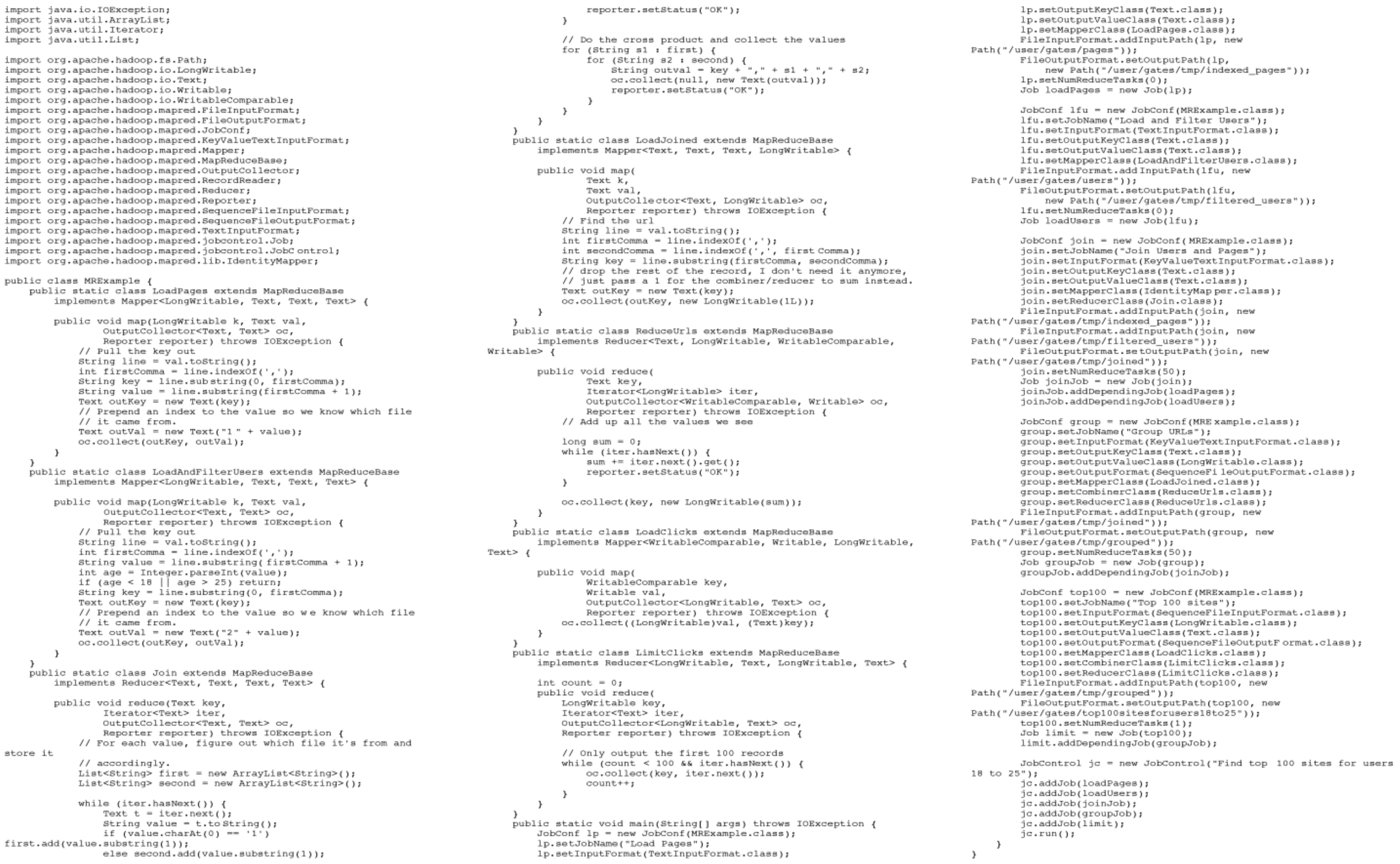
An Example Problem

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.JobC ontrol;
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.sub string(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 w e 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.to String();
                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(',', first Comma);
            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.add InputPath(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(IdentityMap per.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.se tOutputPath(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(MRE xample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFi leOutputFormat.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(SequenceFileOutputF ormat.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

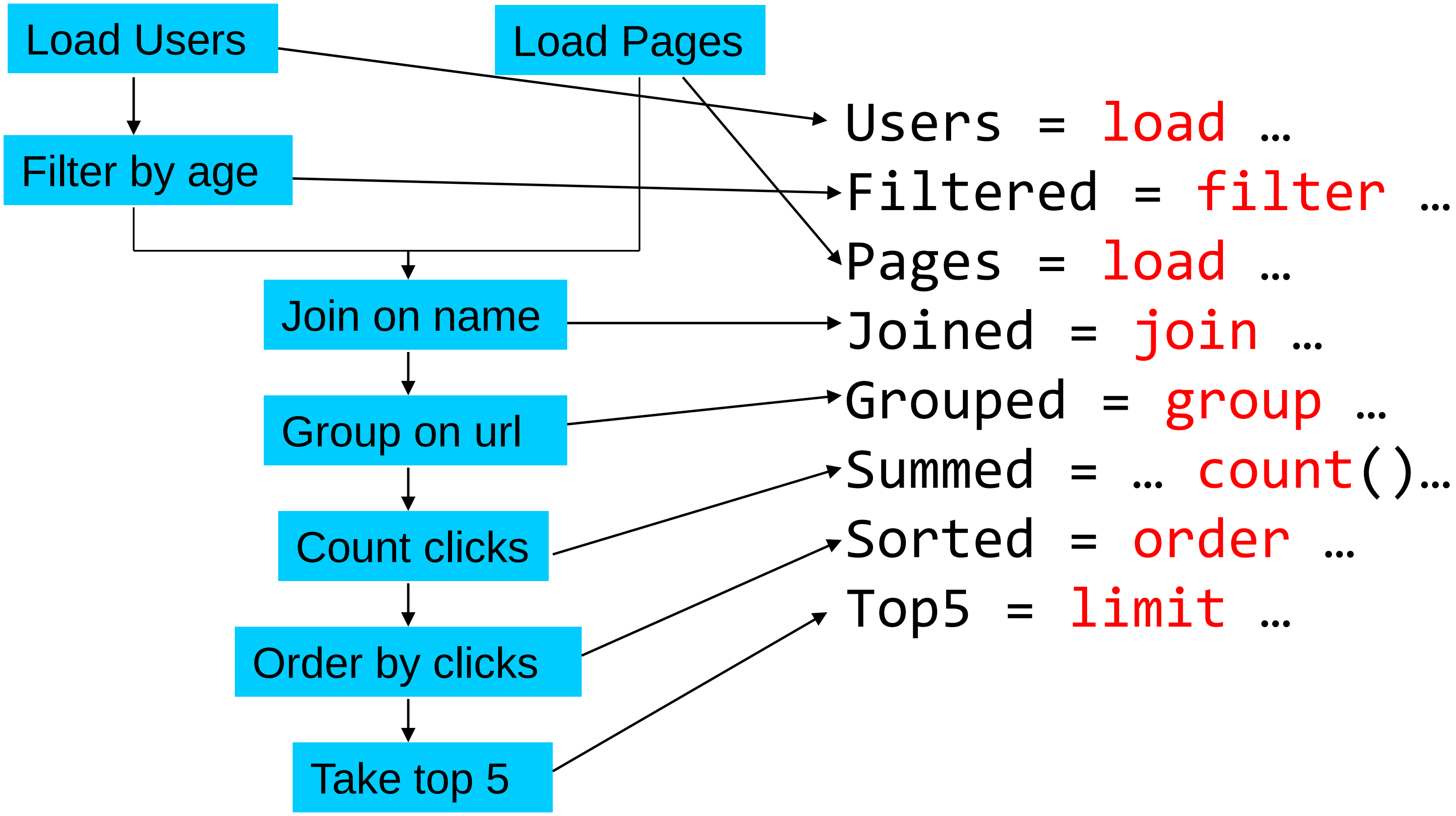
```
Users    = load ‘users’ as (name, age);
Filtered = filter Users by
                  age >= 18 and age <= 25;
Pages    = load ‘pages’ as (user, url);
Joined   = join Filtered by name, Pages by user;
Grouped  = group Joined by url;
Summed   = foreach Grouped generate group,
                   count(Joined)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



Load Users

Load Pages

Filter by age

Join on name

Group on url

Count clicks

Order by clicks

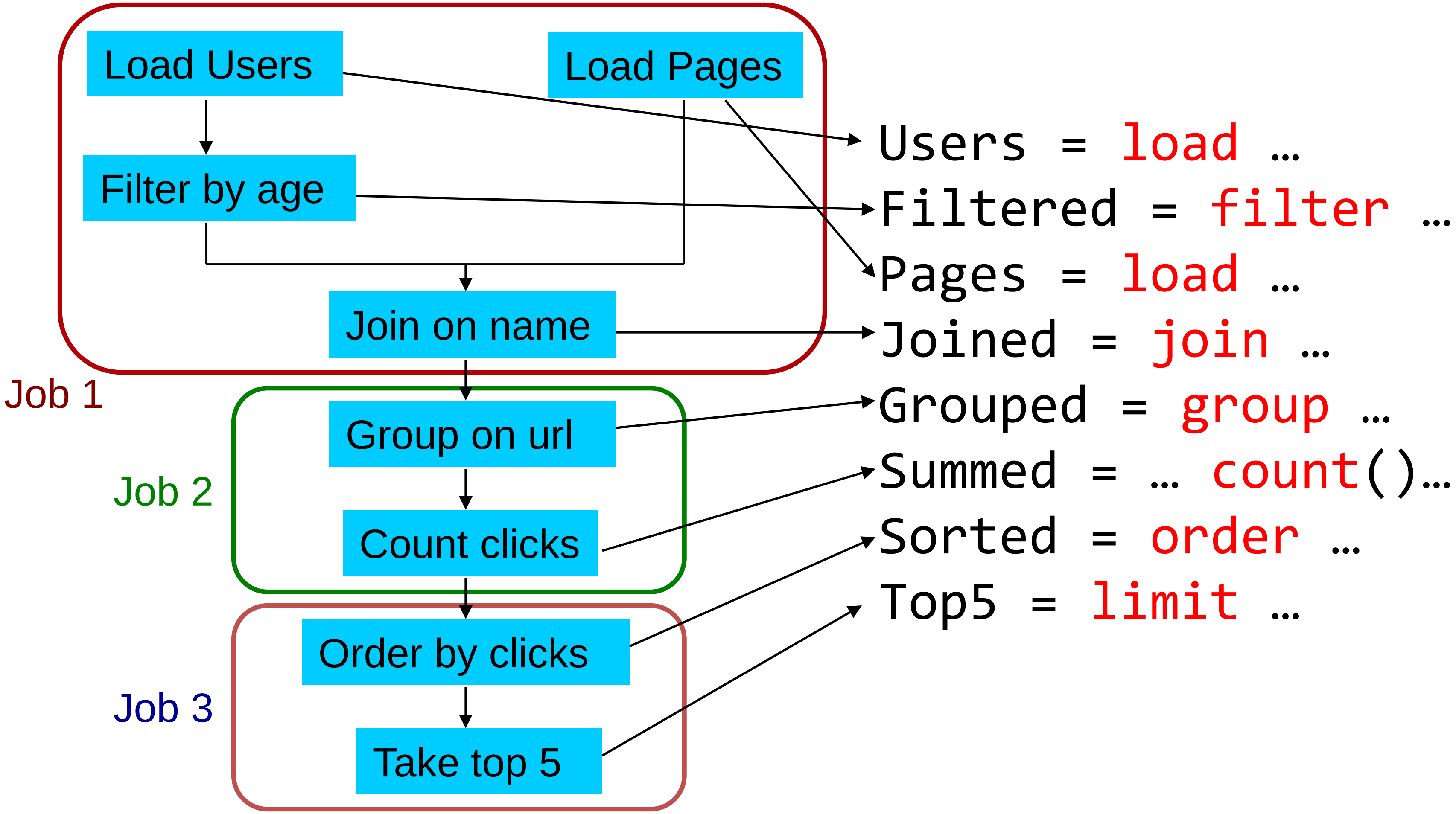
Take top 5

```
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

Translation to MapReduce

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



Load Users

Load Pages

Filter by age

Join on name

Job 1

Group on url

Count clicks

Job 2

Order by clicks

Take top 5

Job 3

```
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

Summary

- 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 necessitating 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
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

Cloud Programming Research

- More general execution engines
  - **Dryad** (Microsoft): general task DAG
  - **S4** (Yahoo!): streaming computation
  - **Pregel** (Google): in-memory iterative graph algs.
  - **Spark** (Berkeley): general in-memory computing

- Language-integrated interfaces
  - Run computations directly from host language
  - **DryadLINQ** (MS), **FlumeJava** (Google), **Spark**

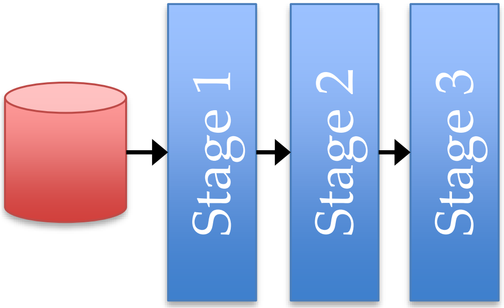
Spark Motivation

- MapReduce simplified "big data" analysis on large, unreliable clusters
- But as soon as organizations started using it widely, users wanted more:
  - More *complex*, multi-stage applications
  - More *interactive* queries
  - More *low-latency* online processing

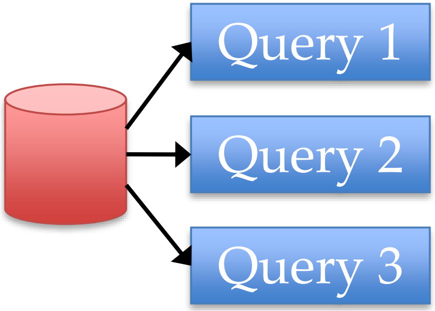
Spark Motivation

Complex jobs, interactive queries and online processing all need one thing that MR lacks:

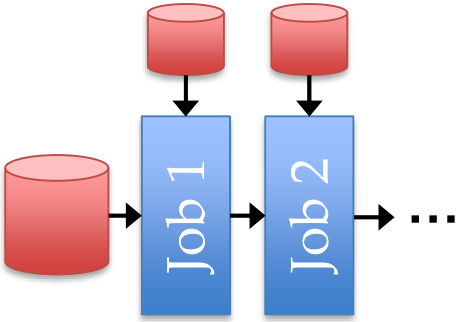
Efficient primitives for **data sharing**



Iterative job



Interactive mining

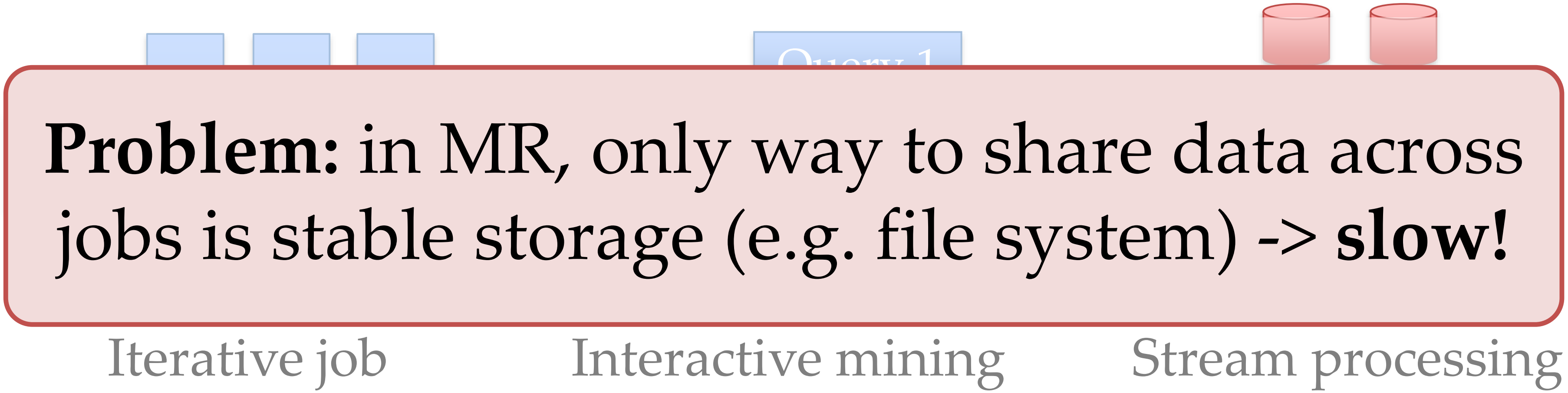


Stream processing

Spark Motivation

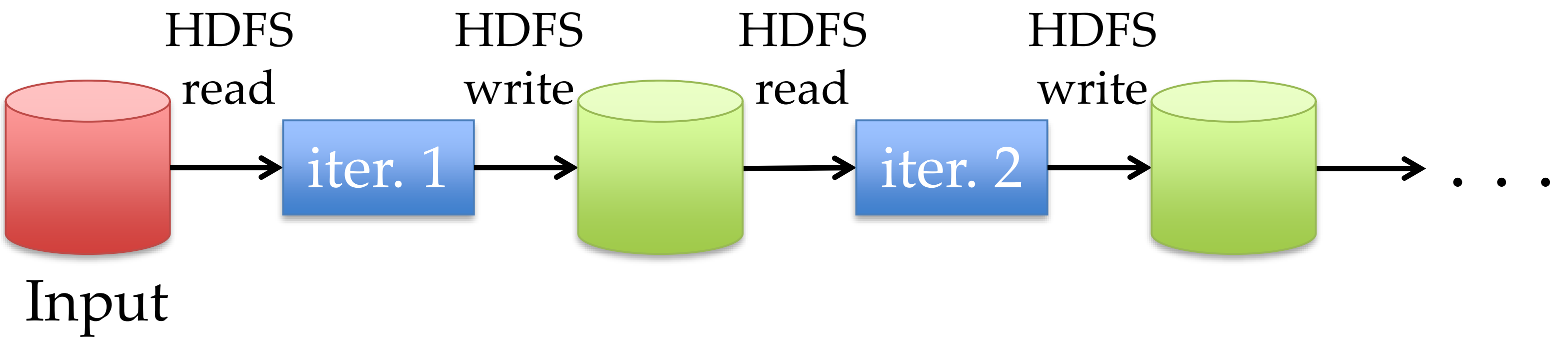
Complex jobs, interactive queries and online processing all need one thing that MR lacks:

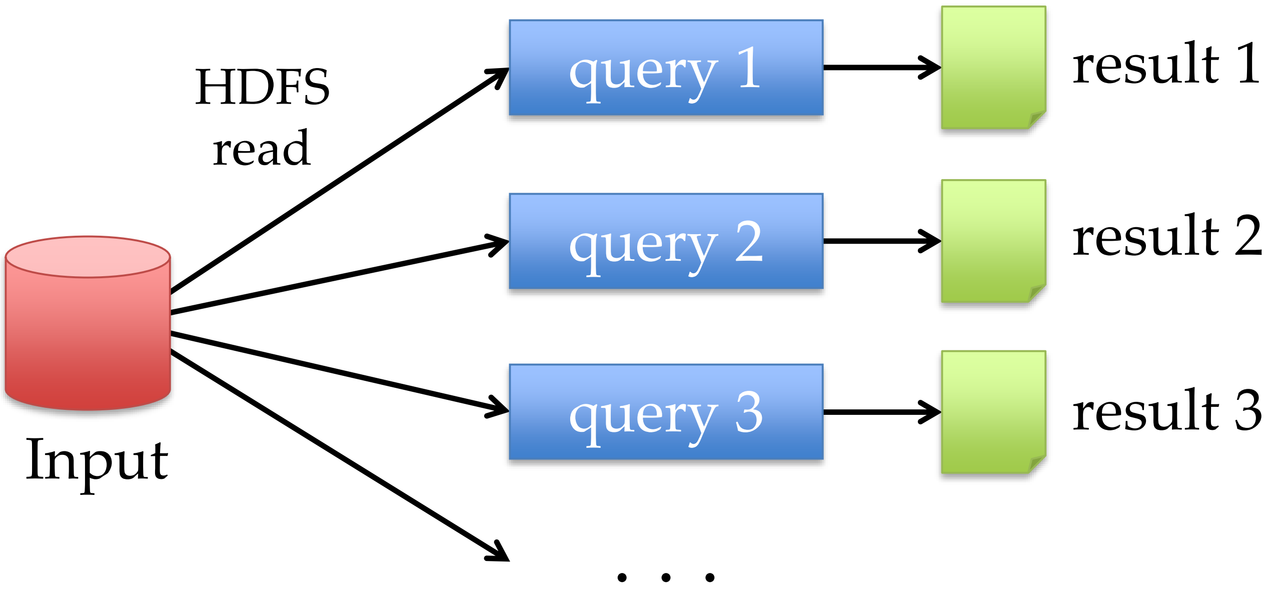
Efficient primitives for **data sharing**



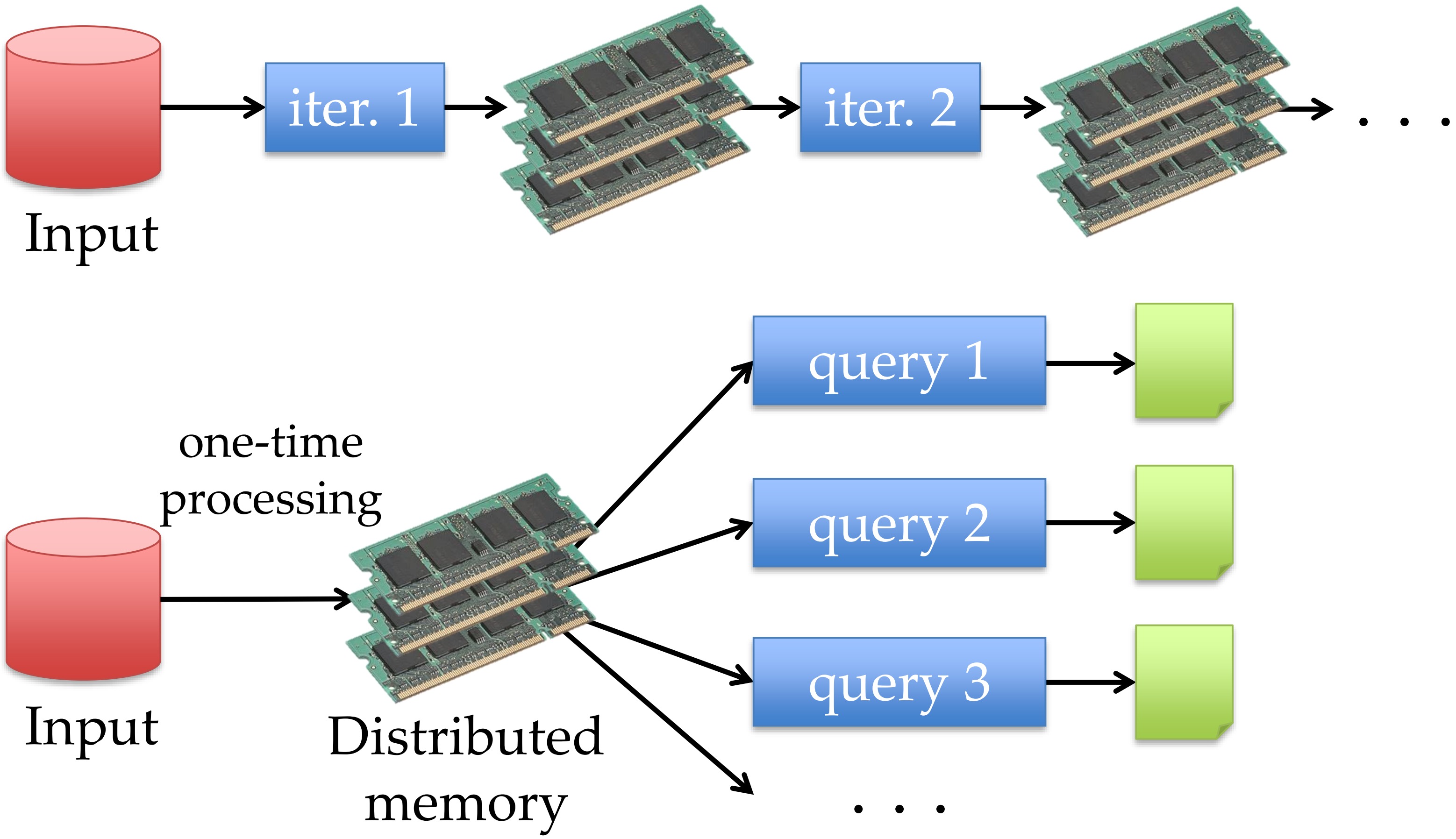
Problem: in MR, only way to share data across jobs is stable storage (e.g. file system) -> **slow!**

Examples


HDFS read
HDFS write
HDFS read
HDFS write
iter. 1
iter. 2
. . .
Input


HDFS read
query 1
query 2
query 3
result 1
result 2
result 3
Input
. . .

Goal: In-Memory Data Sharing



10-100× faster than network and disk

Solution: Resilient Distributed Datasets (RDDs)

- Partitioned collections of records that can be stored in memory across the cluster

- Manipulated through a diverse set of transformations (*map, filter, join,* etc)

- Fault recovery without costly replication
  - Remember the series of transformations that built an RDD (its *lineage*) to *recompute* lost data

Example: Log Mining

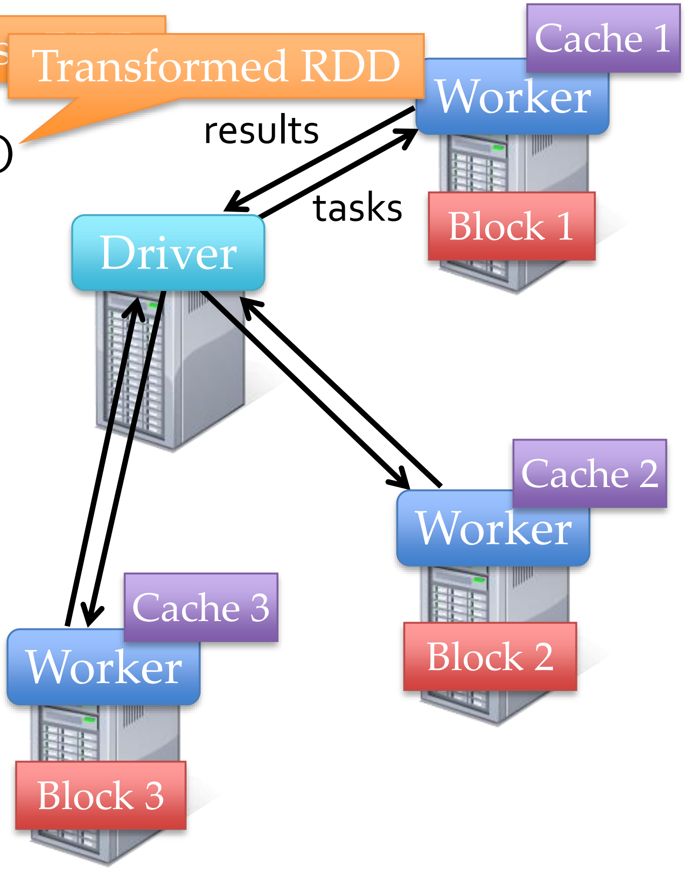
Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
messages.cache()

messages.filter(_.contains(“foo”)).count
messages.filter(_.contains(“bar”)).count
. . .
```

Bas

Transformed RDD



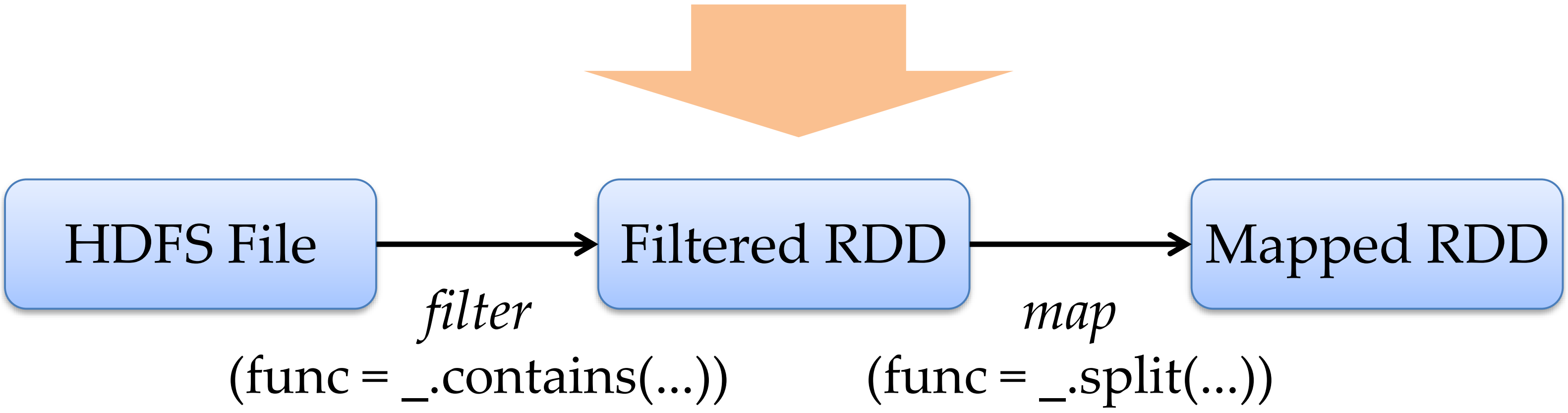
Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

Scala programming language

Fault Recovery

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex:
```
messages = textFile(...).filter(_.startsWith(“ERROR”))
                        .map(_.split(‘\t’)(2))
```



HDFS File

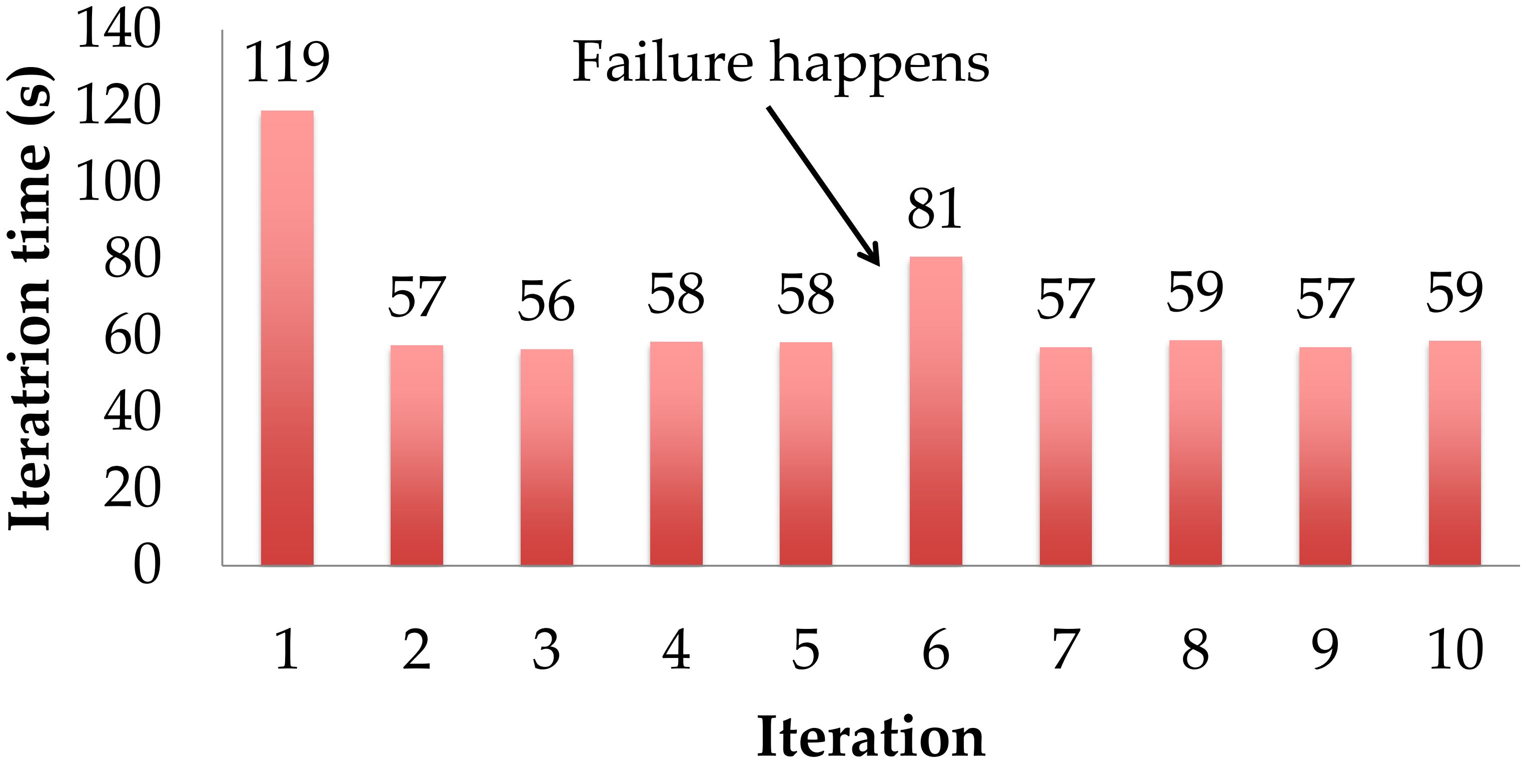
Filtered RDD

Mapped RDD

filter
(func = _.contains(...))

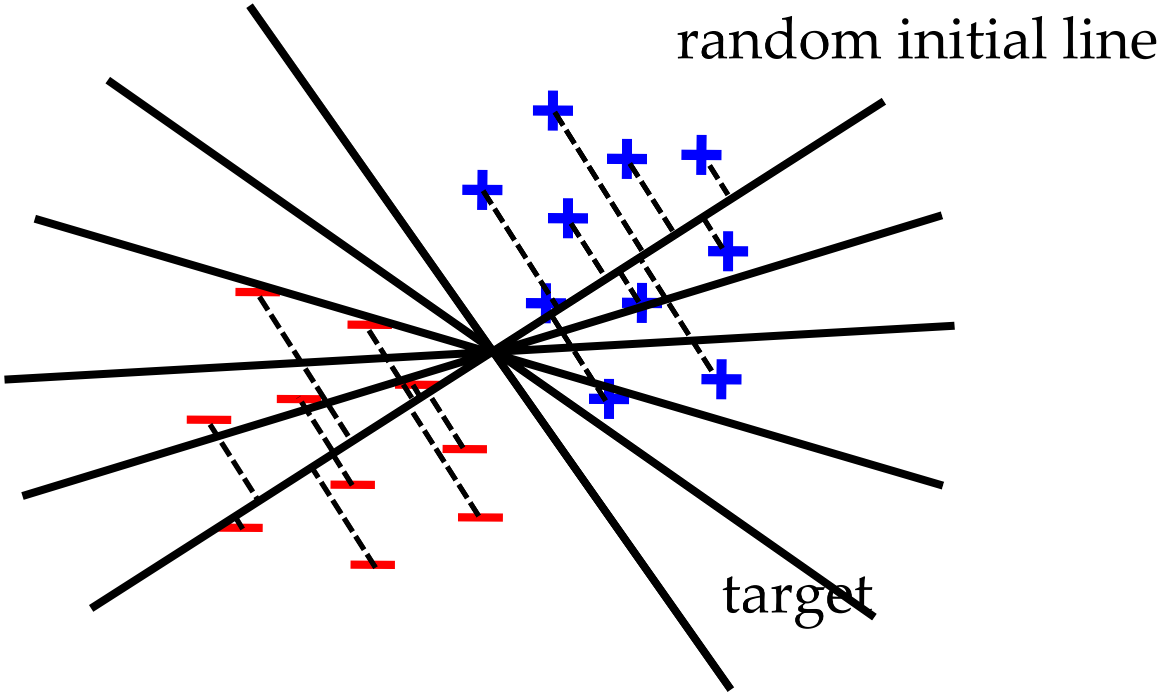
map
(func = _.split(...))

Fault Recovery Results



Example: Logistic Regression

Find best line separating two sets of points



Logistic Regression Code

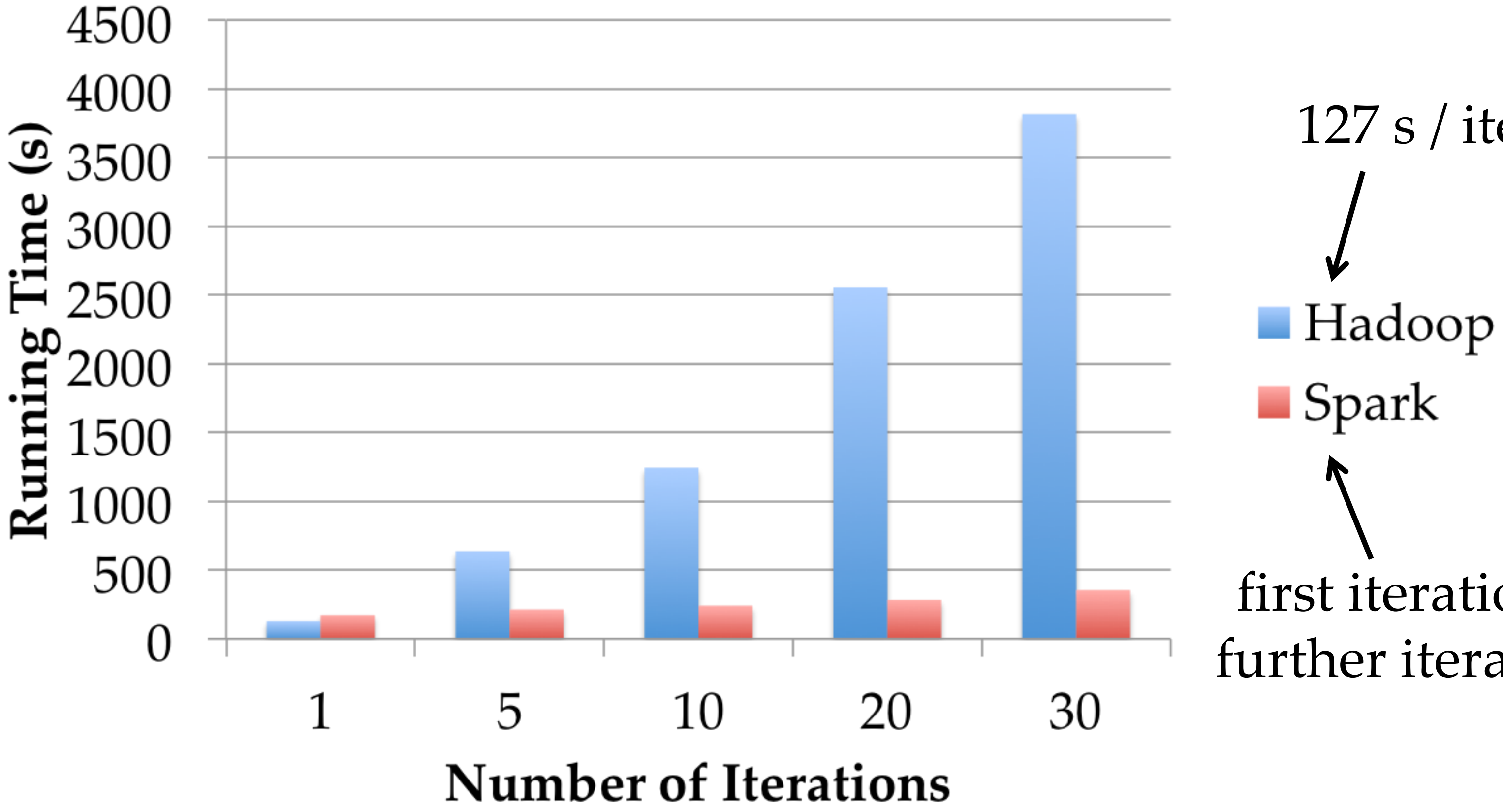
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

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

Logistic Regression Performance



127 s / iteration

first iteration 174 s
further iterations 6 s

If You Want to Try It Out

- www.spark-project.org
- To run locally, just need Java installed
- Easy scripts for launching on Amazon EC2
- Can call into any Java library from Scala

Other Resources

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

- Hadoop video tutorials: www.cloudera.com/hadoop-training

- Amazon Elastic MapReduce: http://aws.amazon.com/elasticmapreduce/