MAPREDUCE & HADOOP

INTRODUCTION TO DATA SCIENCE

CARSTEN BINNIG

BROWN UNIVERSITY

WHAT IF WE HAVE TO MUCH DATA?

Computations that need the power of many computers

To analyze large datasets

Using nodes (i.e. servers) in a cluster in parallel

WHAT IS MAPREDUCE?

Simple programming model for data-intensive computing on large commodity clusters

Pioneered by Google

- Processes PB's of data of per day (e.g., process user logs, web crawls, …)

Popularized by Apache Hadoop project (Yahoo)

- Used by Yahoo!, Facebook, Amazon, …

Many other MapReduce-based frameworks today

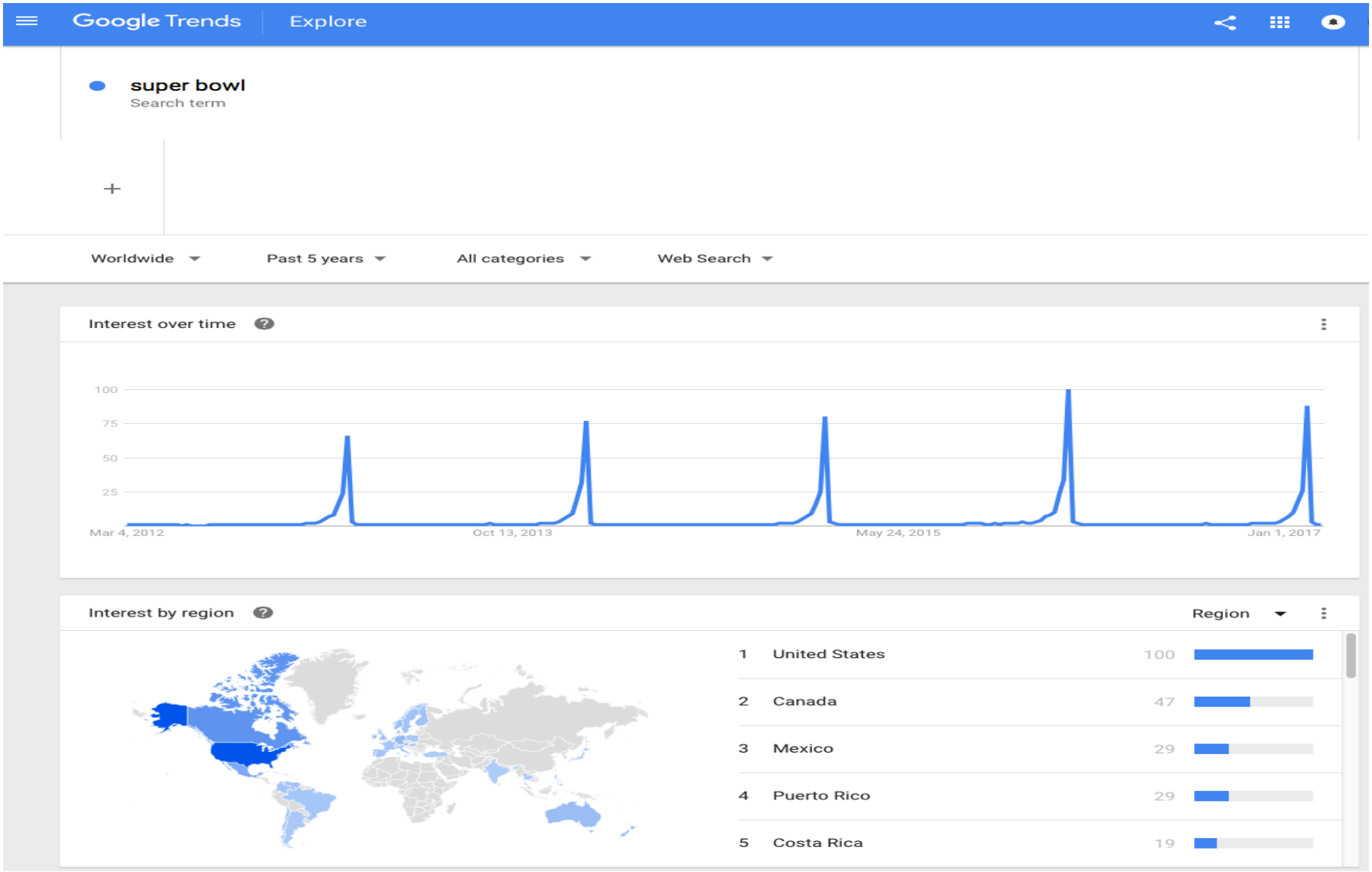
WHAT IS MAPREDUCE USED FOR?

- **At Google:**
  - Index building for Google Search
  - Analyzing search logs (Google Trends)
  - Article clustering for Google News
  - Statistical language translation

- **At Facebook:**
  - Data mining
  - Ad optimization
  - Machine learning (e.g., spam detection)

…

EXAMPLE: GOOGLE TRENDS



https://www.google.com/trends/

WHAT ELSE IS MAPREDUCE USED FOR?

In research:

- Topic distribution in Wikipedia (PARC)
- Natural language processing (CMU)
- Climate simulation (UW)
- Bioinformatics (Maryland)
- Particle physics (Nebraska)
- **<Your application here>**

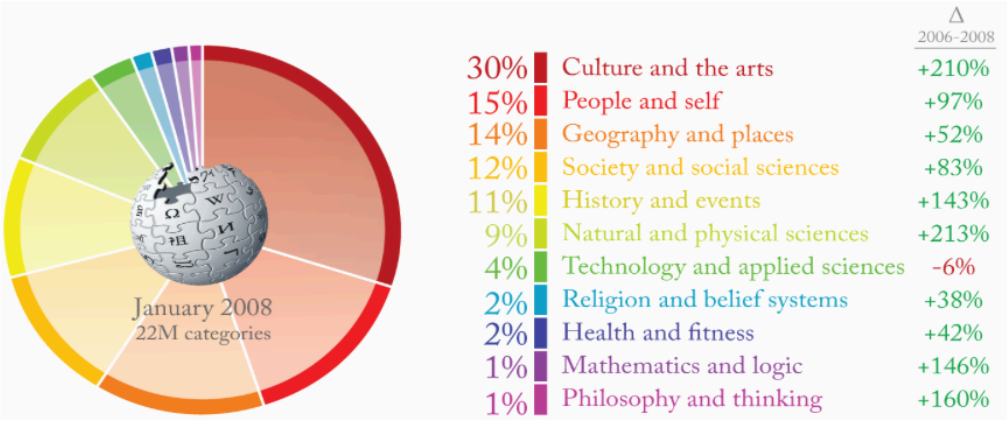


Figure 1. Distribution of topics in Wikipedia from January 2008 along with change since July 2006.

MAPREDUCE GOALS

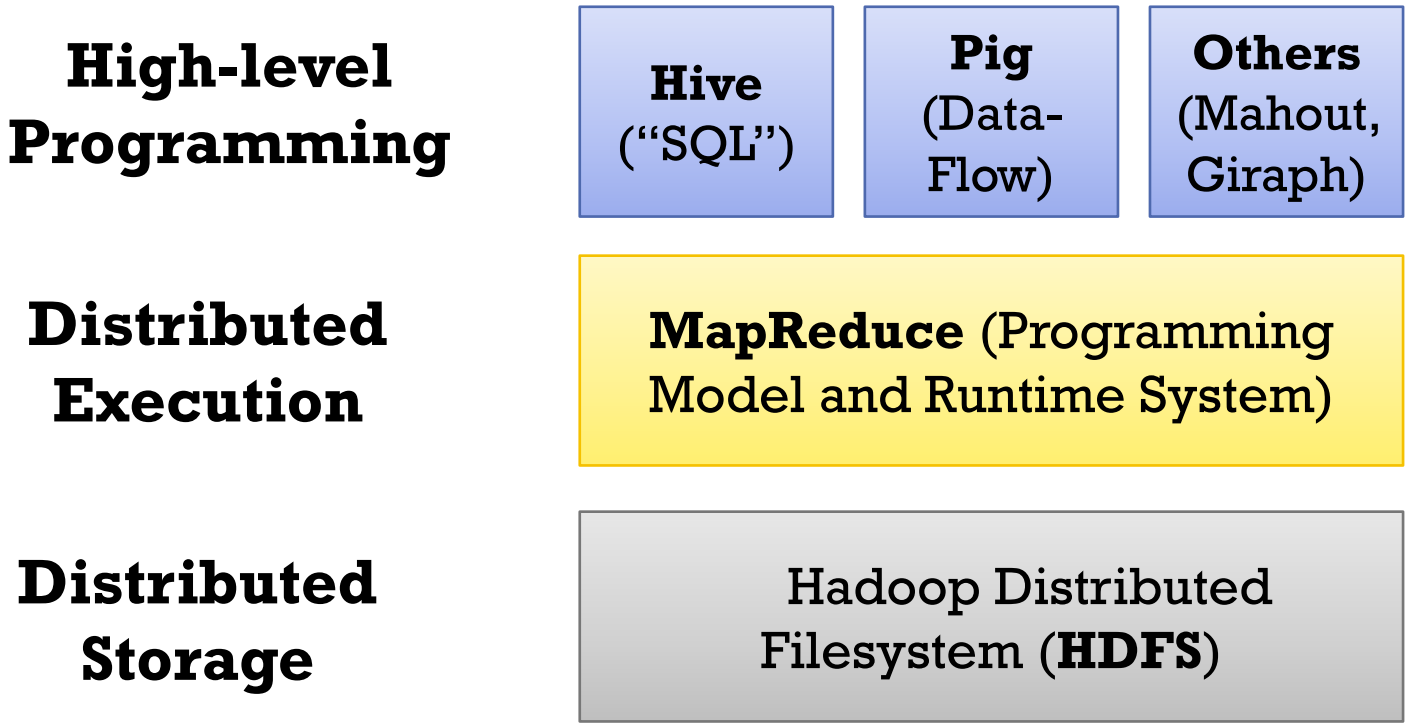
- **Scalability to large data volumes:**
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

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

ARCHITECTURE & PROGRAMMING MODEL

HADOOP 1.0: ARCHITECTURE

Open-Source MapReduce System:



MAIN HADOOP COMPONENTS

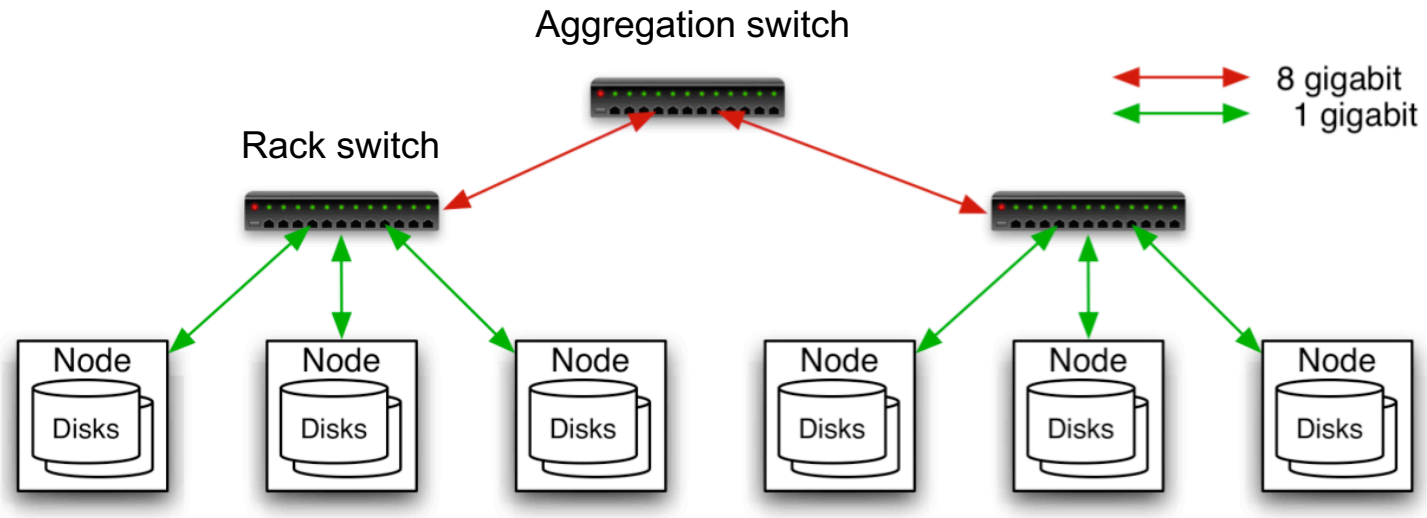
HDFS = Hadoop Distributed File System

- Single namespace for entire cluster
- Replicates data 3x for fault-tolerance

MapReduce framework

- Runs jobs submitted by users
- Manages work distribution & fault-tolerance
- Co-locates work with file system

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 COMMODITY CLUSTERS

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

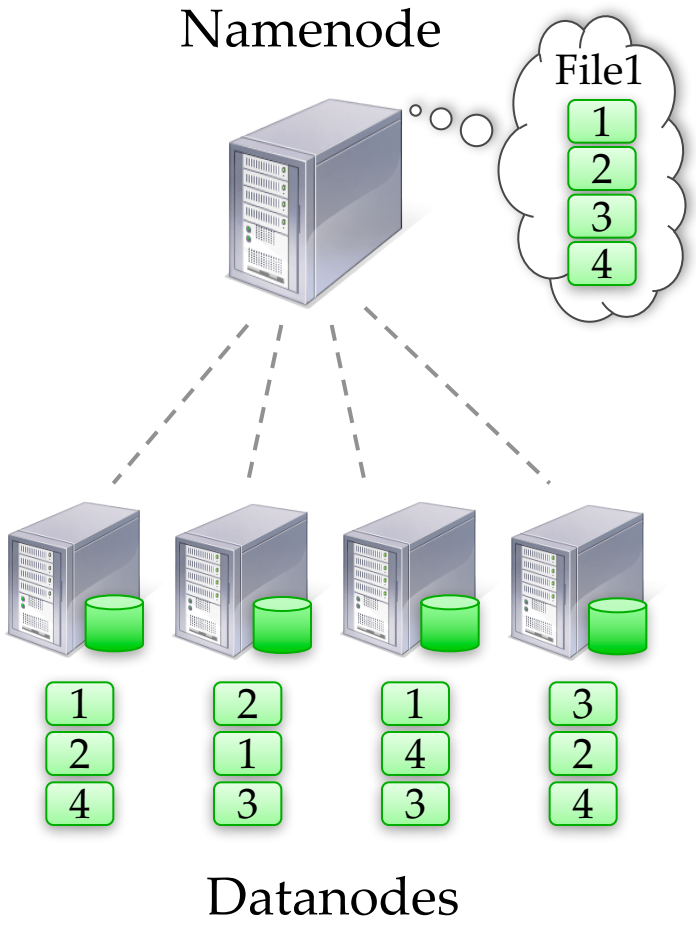
- **Solution:** Push computation to the data

Programming in a cluster is hard

- Parallel programming is hard
- Distributed parallel programming is even harder
- **Solution:** Restricted programming model: Users write data-parallel "map" and "reduce" functions, system handles work distribution and failures

HADOOP DISTRIBUTED FILE SYSTEM

- Files split into **64 MB 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:

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

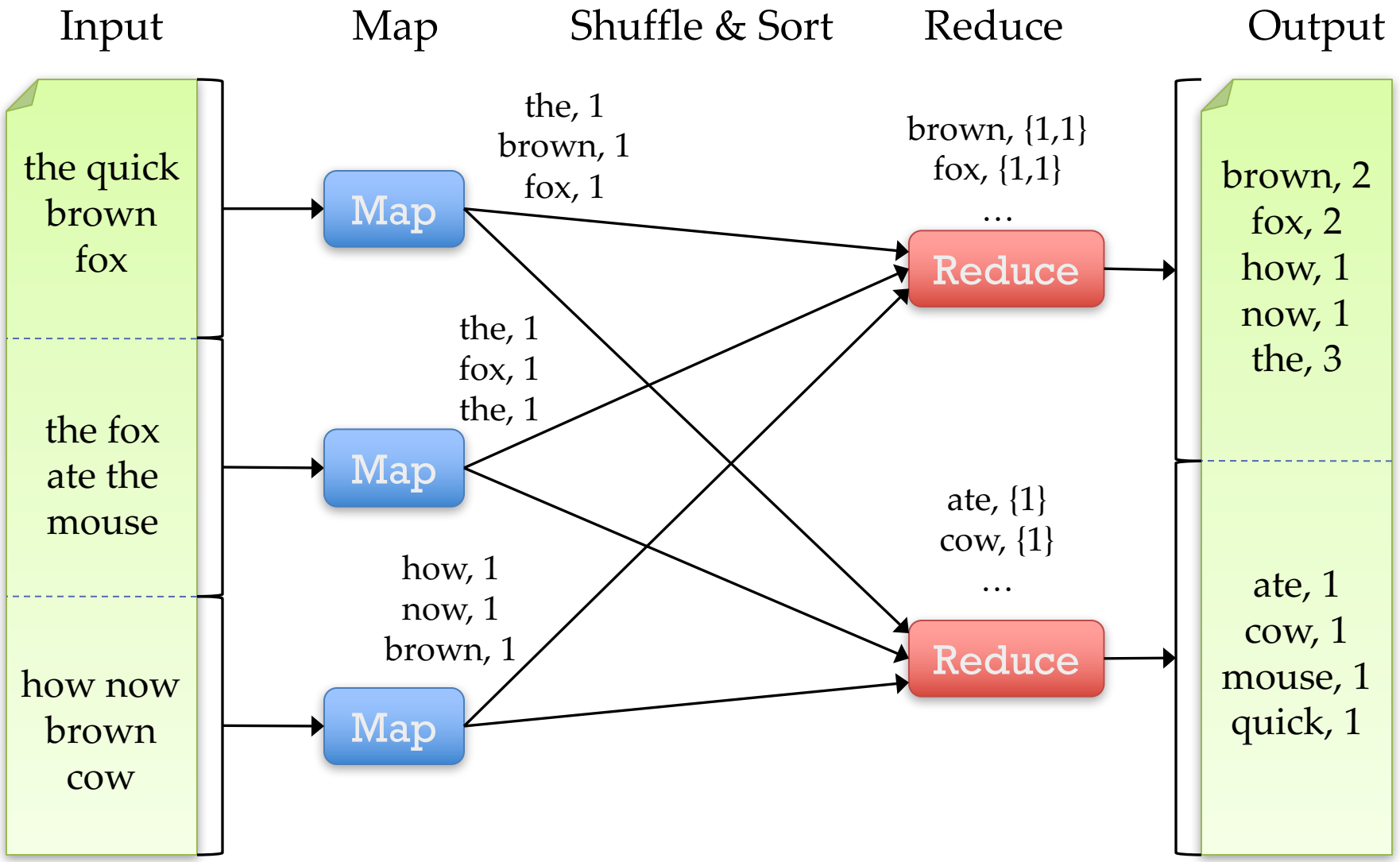
Reduce function:

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

EXAMPLE: WORD COUNT

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

def reducer(key, values): //values={1,1,1,…}
    emit(key, count(values))
```

WORD COUNT EXECUTION



AN OPTIMIZATION: THE COMBINER

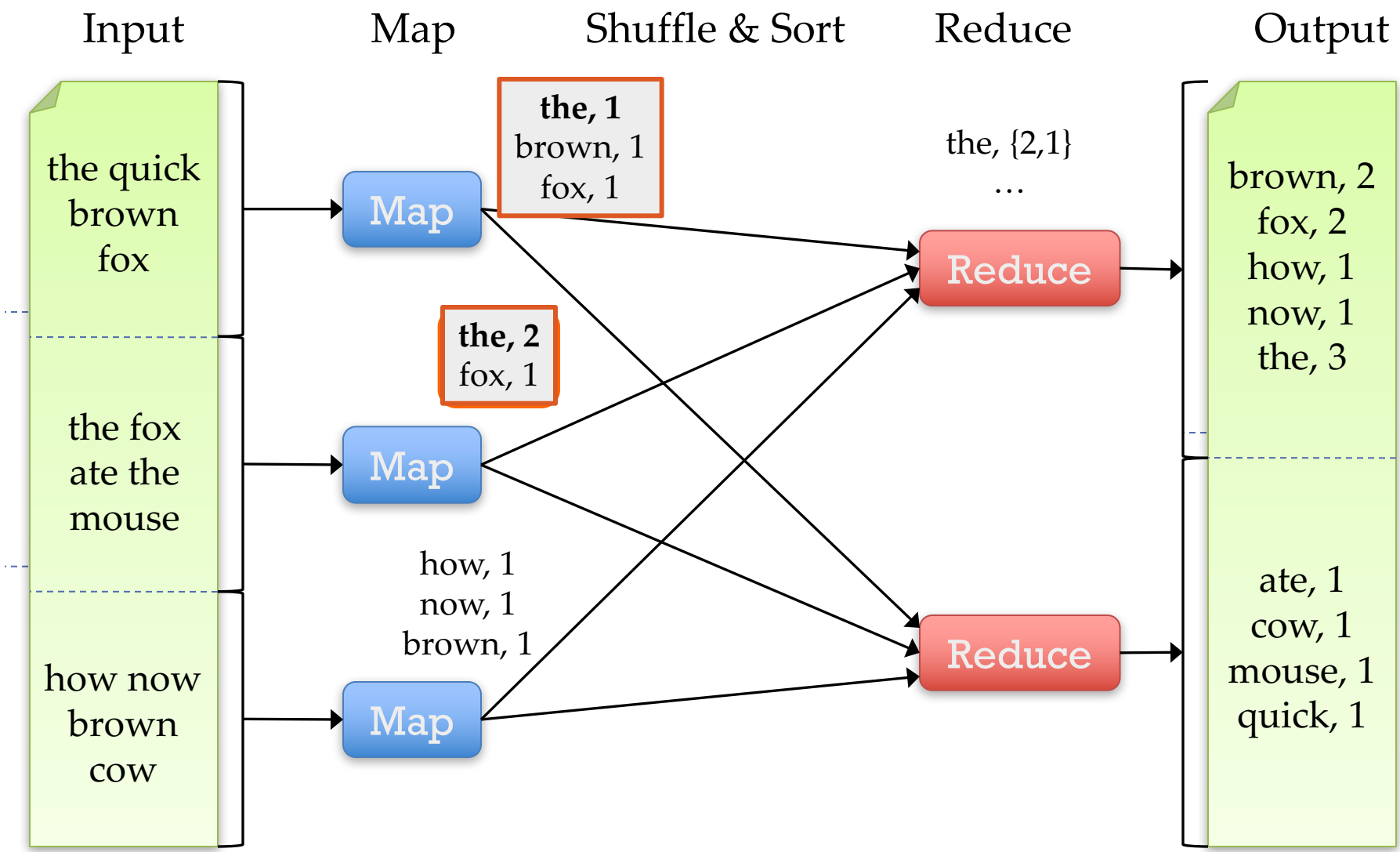
Local reduce function for **repeated keys produced by same map**

- For associative operations like sum, count, max
- Decreases amount of intermediate data

Example: local counting for Word Count:

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

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

WORD COUNT WITH COMBINER



CLICKER QUESTION

You want to implement the dot product $a_1*b_1 + a_2*b_2 + \ldots + a_n*b_n$ of two large vectors $a=[a_1, a_2,\ldots,a_n]$ and $b=[b_1, b_2,\ldots,b_n]$. The input to the mapper is a pair (a_i,b_i).

Which of the following implementations is correct?

A)

```
def mapper(a,b):
  emit(a*b, 1)

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

B)

```
def mapper(a,b):
  emit(1, a+b)

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

C)

```
def mapper(a,b):
  emit(1, a*b)

def reducer(key, values):
  emit(key, mult(values))
```

→ D)

```
def mapper(a,b):
  emit(1, a*b)

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

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

Reducers save outputs to HDFS

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 also 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

SAMPLE APPLICATIONS

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

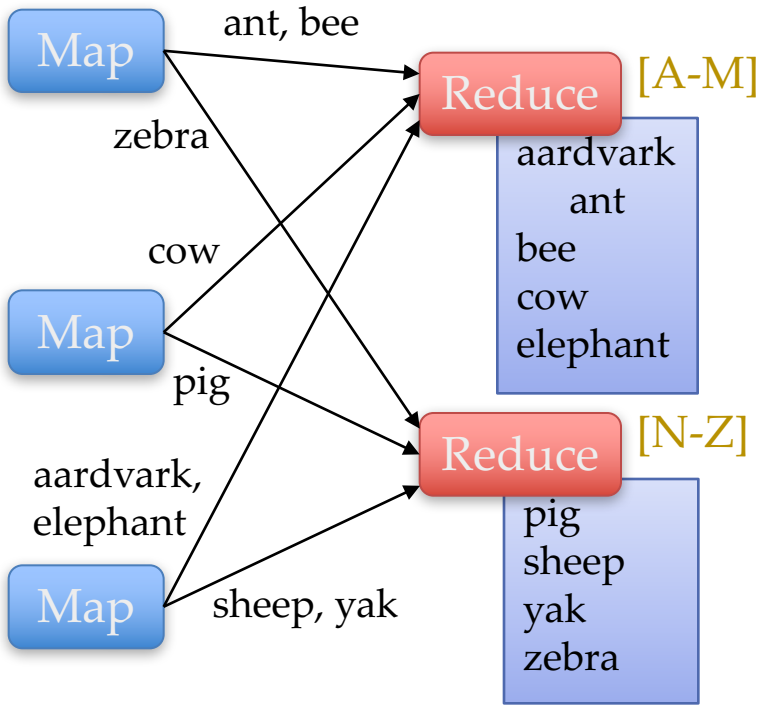
Input: (key, value) records

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:

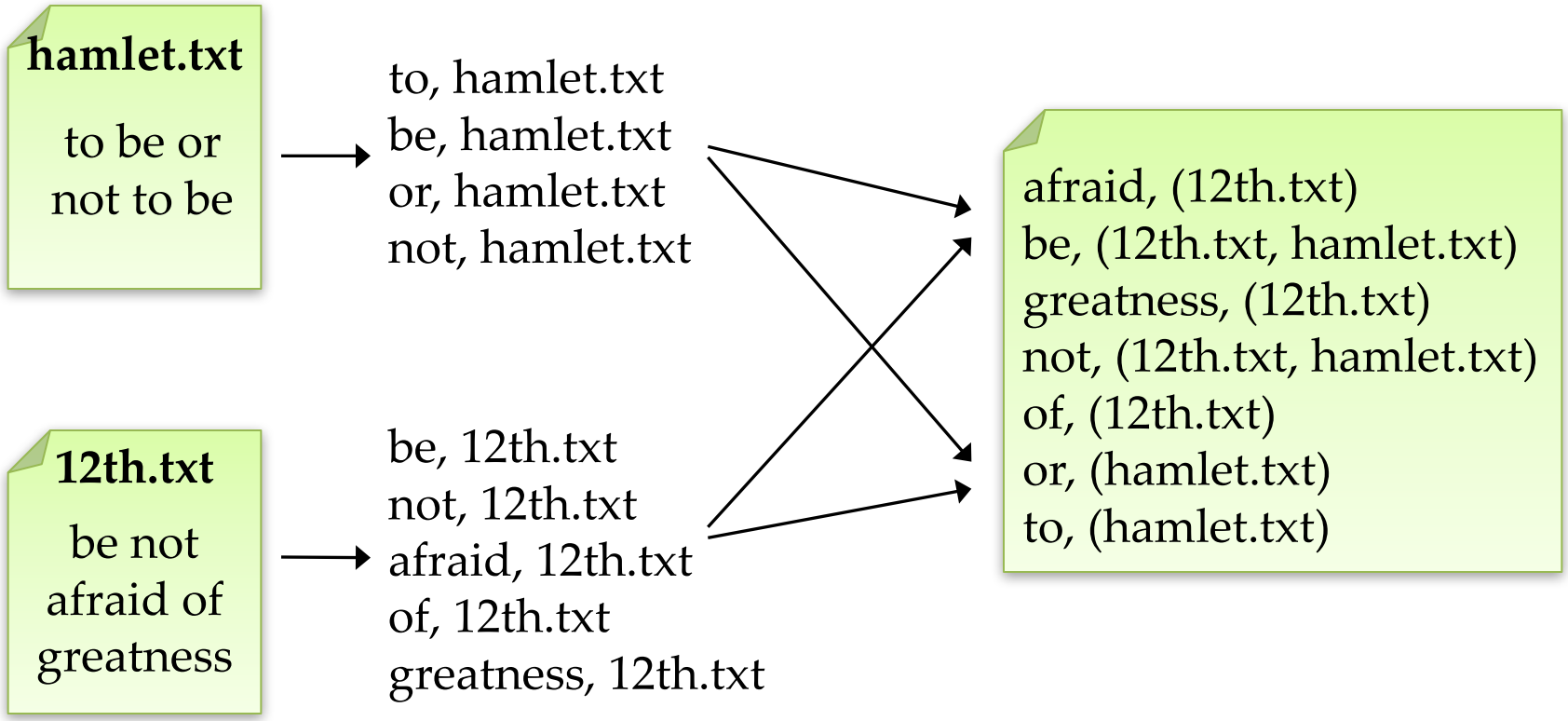
```
foreach word in text.split():
  output(word, filename)
```

Combine: uniquify filenames for each word

Reduce:

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

INVERTED INDEX EXAMPLE



4. MOST POPULAR WORDS

Input: (filename, text) records

Output: the 100 words occurring in most files

Two-stage solution:

- **MapReduce Job 1:**

  Create inverted index, giving (word, list(file)) records

- **MapReduce Job 2:**

  Map each (word, list(file)) to (count, word)

  Sort these records by count as in sort job

5. NUMERICAL INTEGRATION

Input: (start, end) records for sub-ranges to integrate

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
  emit(1, sum)
```

Reduce:

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

HADOOP APIS

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.): Uses std.in and out
- 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);
}
```

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)
```

MAPREDUCE & HADOOP

PIG & HIVE

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

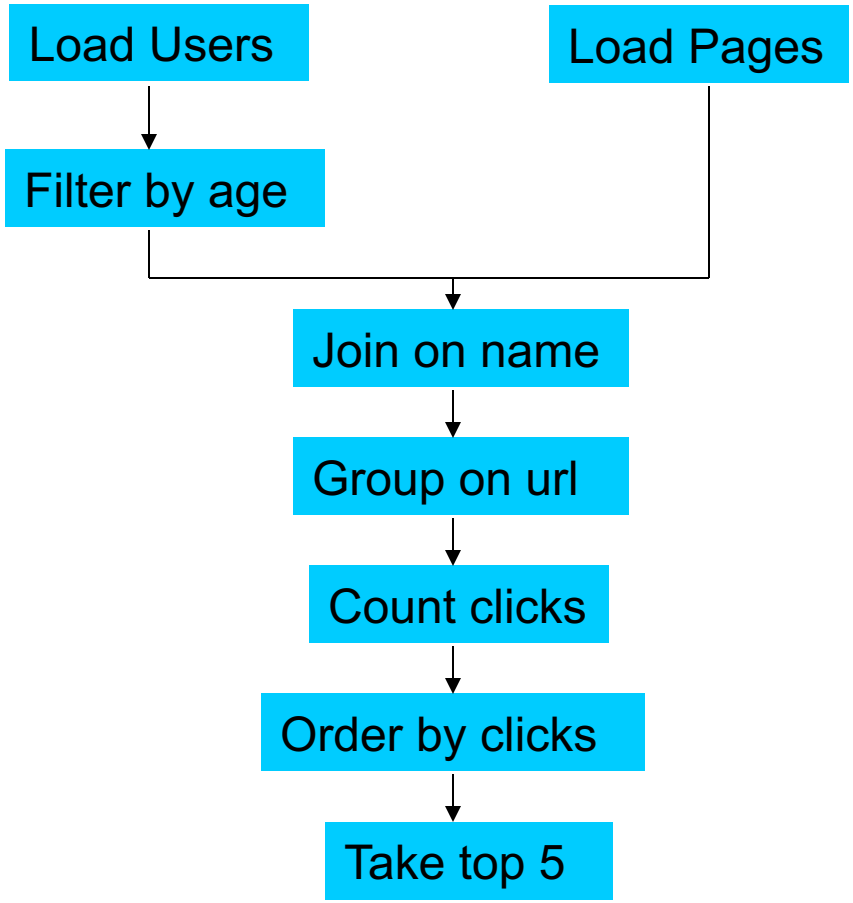
Runs about 50% of Yahoo!'s jobs

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.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(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.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(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

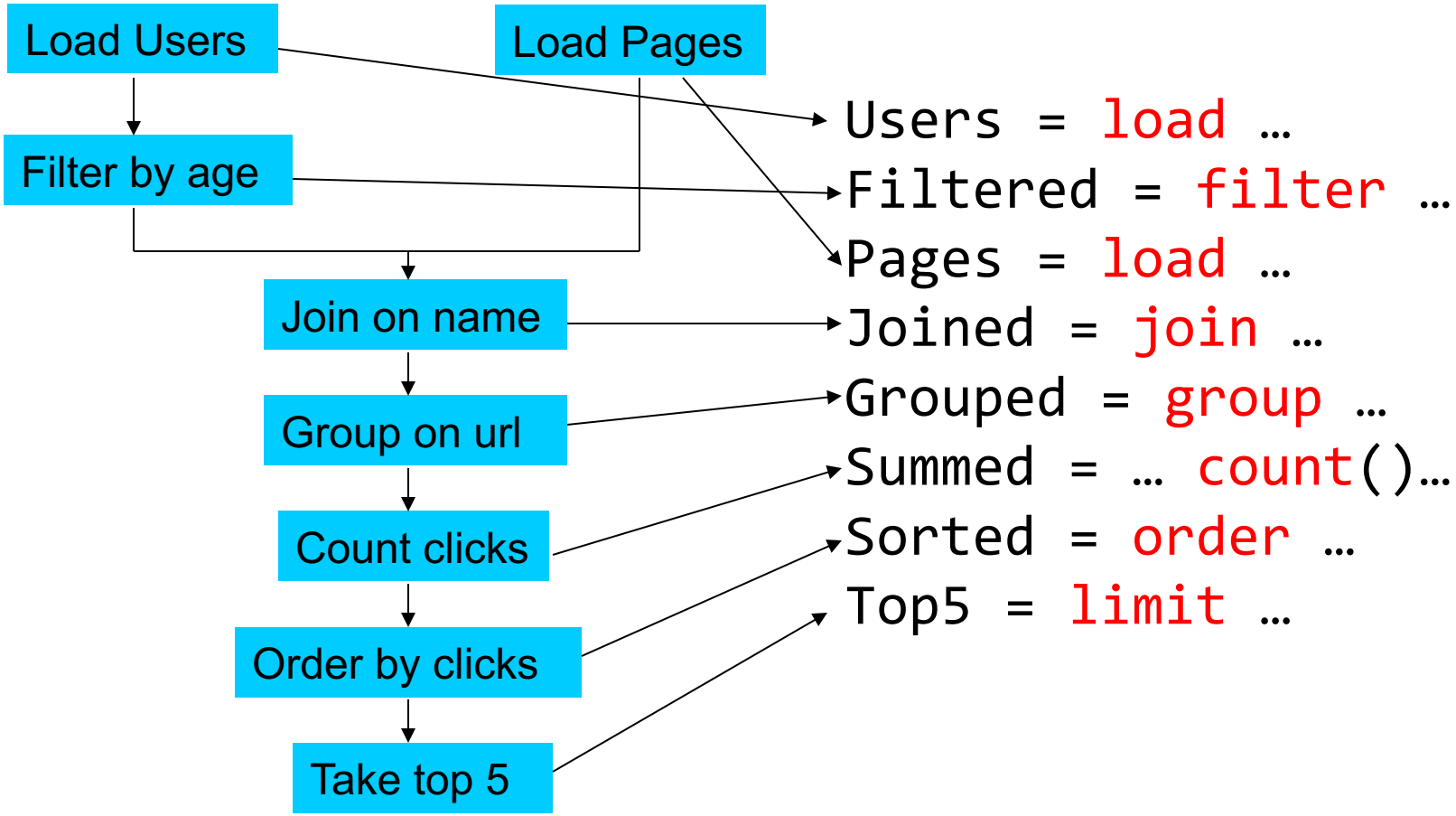
IN PIG LATIN

```
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

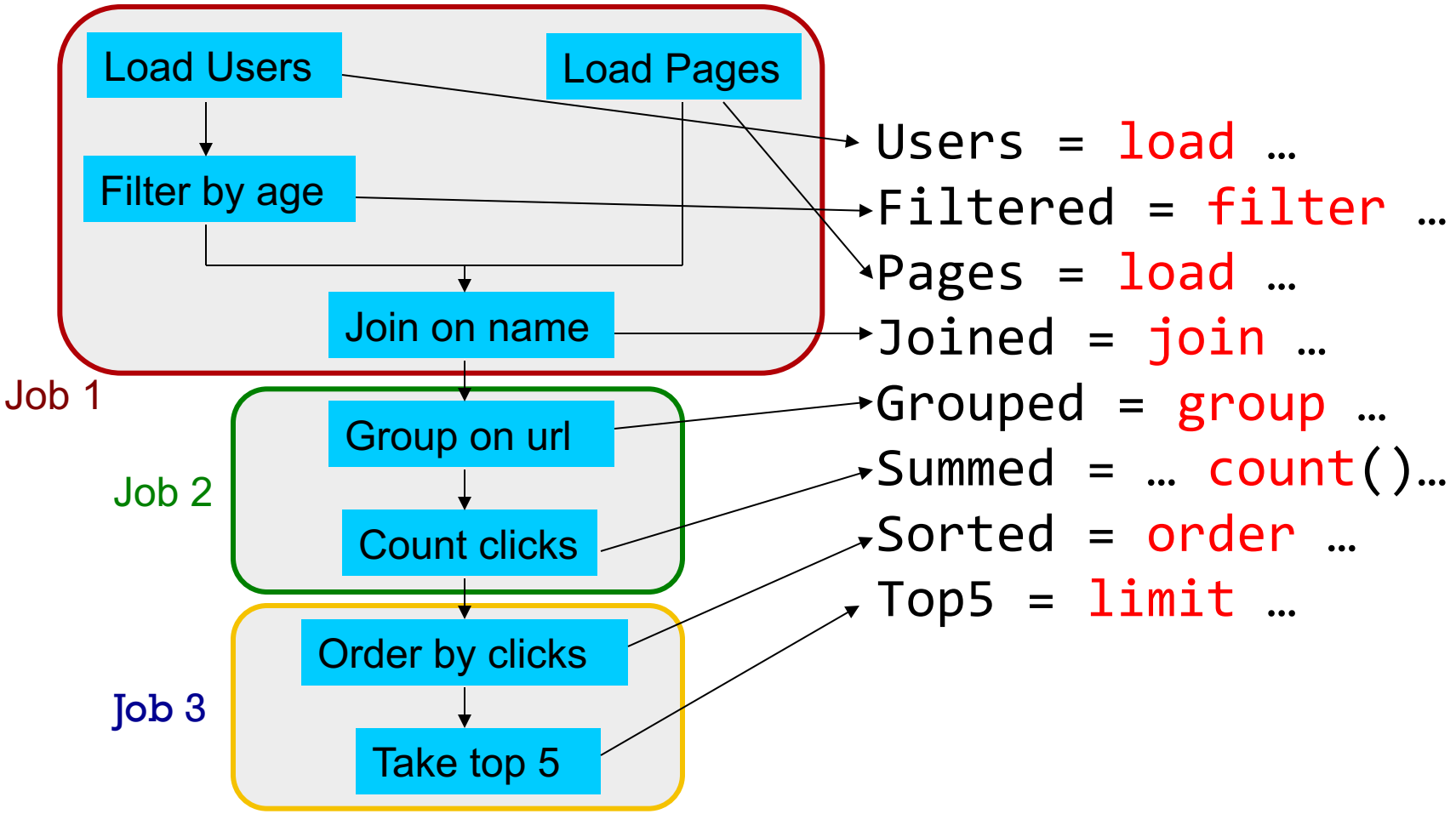
Notice how naturally the components of the job translate into Pig Latin.



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

SQL-like interface 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 query optimization

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

RECENT TRENDS

OTHER SYSTEMS

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

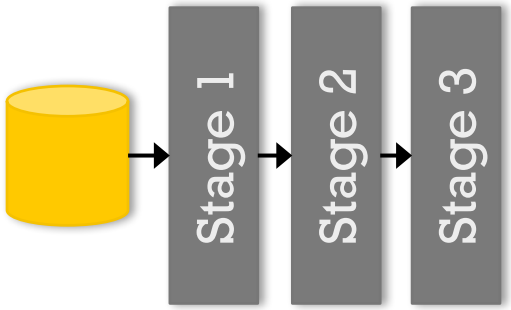
But as soon as organizations started using it widely, users wanted more:

- More *complex*, multi-stage applications
- More *interactive* queries
- More *low-latency* online processing

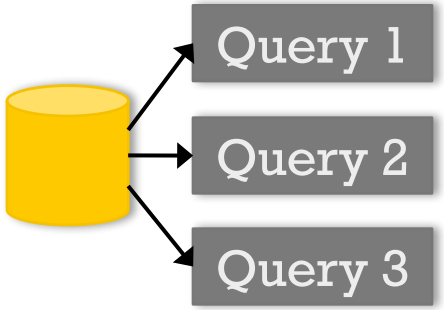
SPARK MOTIVATION

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

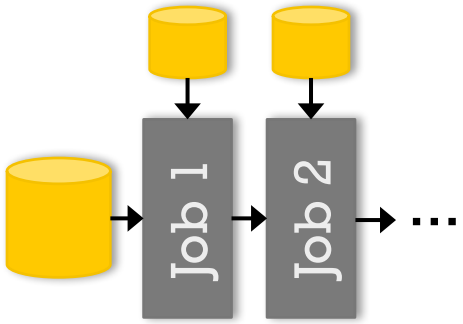
Efficient primitives for data sharing



Iterative job

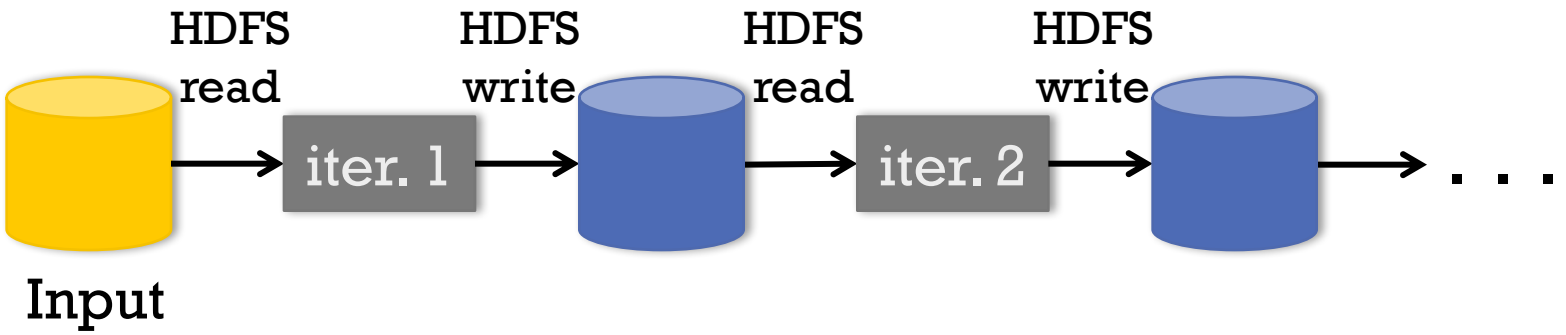


Interactive mining



Stream processing

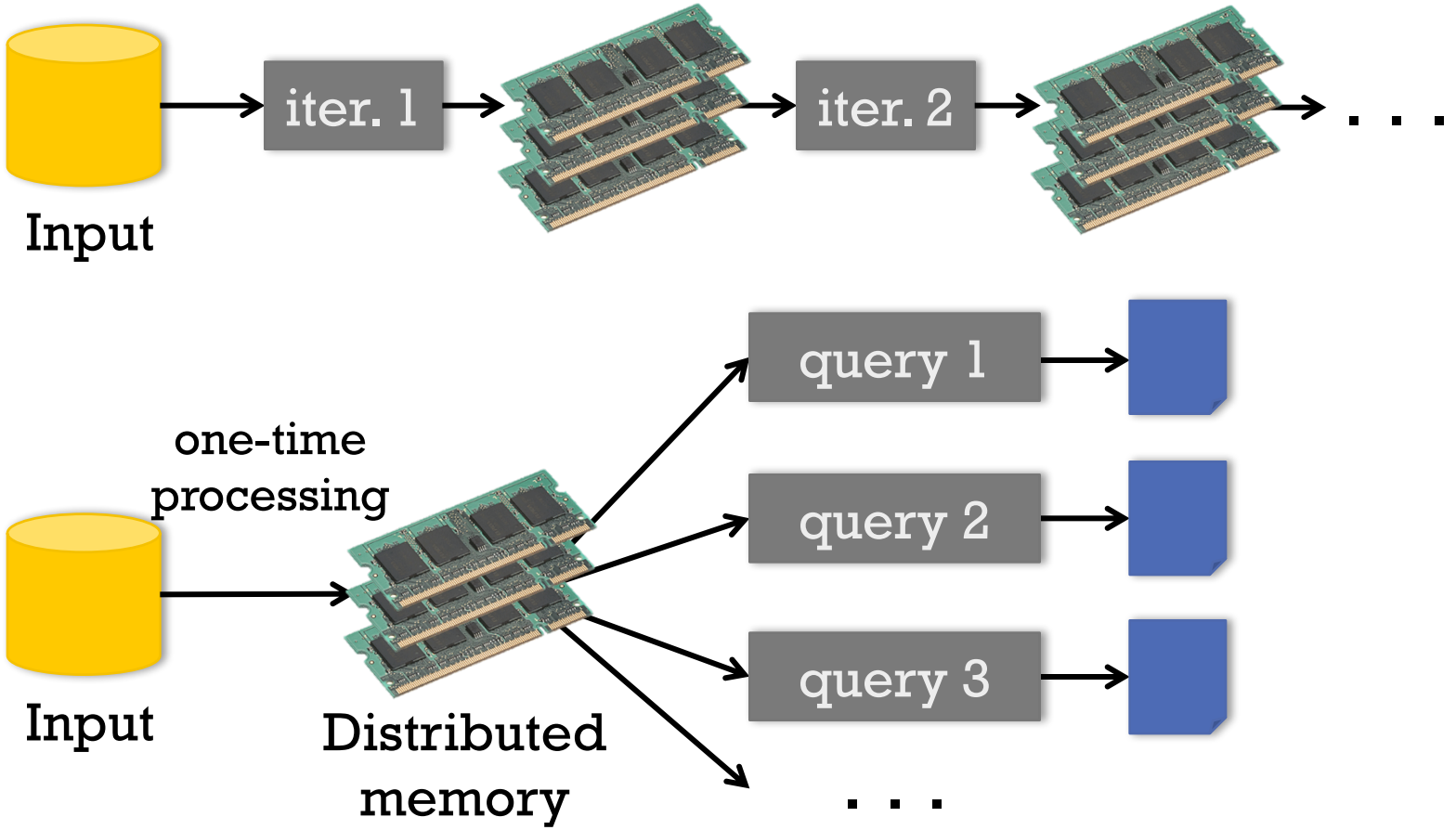
EXAMPLES



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

. . .

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)

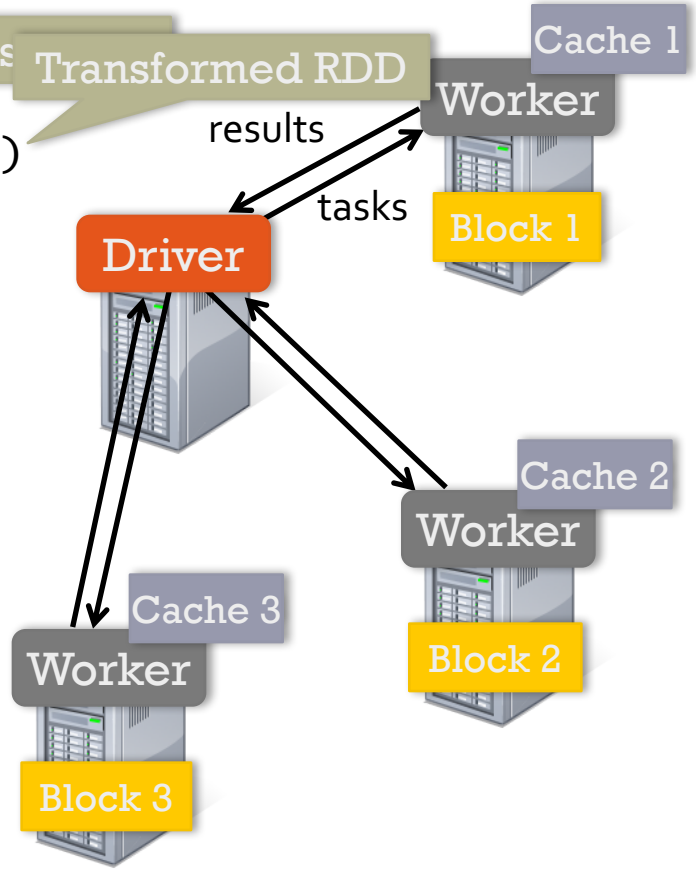
Fault recovery without costly replication

- 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

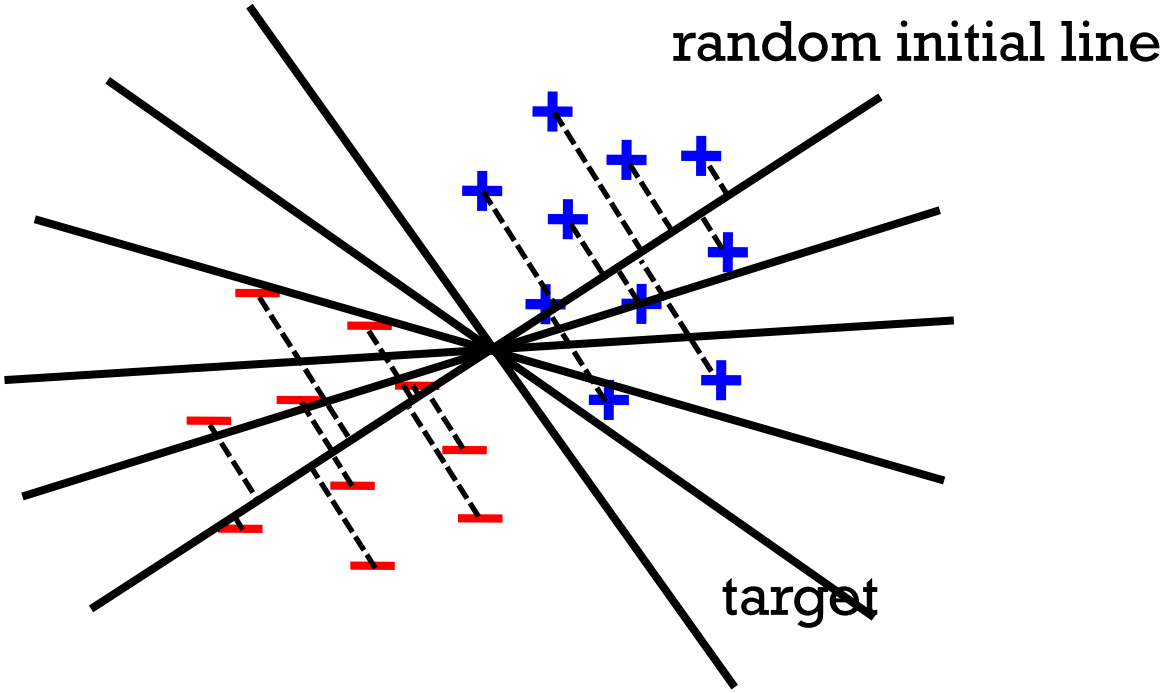
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
. . .
```



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

EXAMPLE: LINEAR REGRESSION

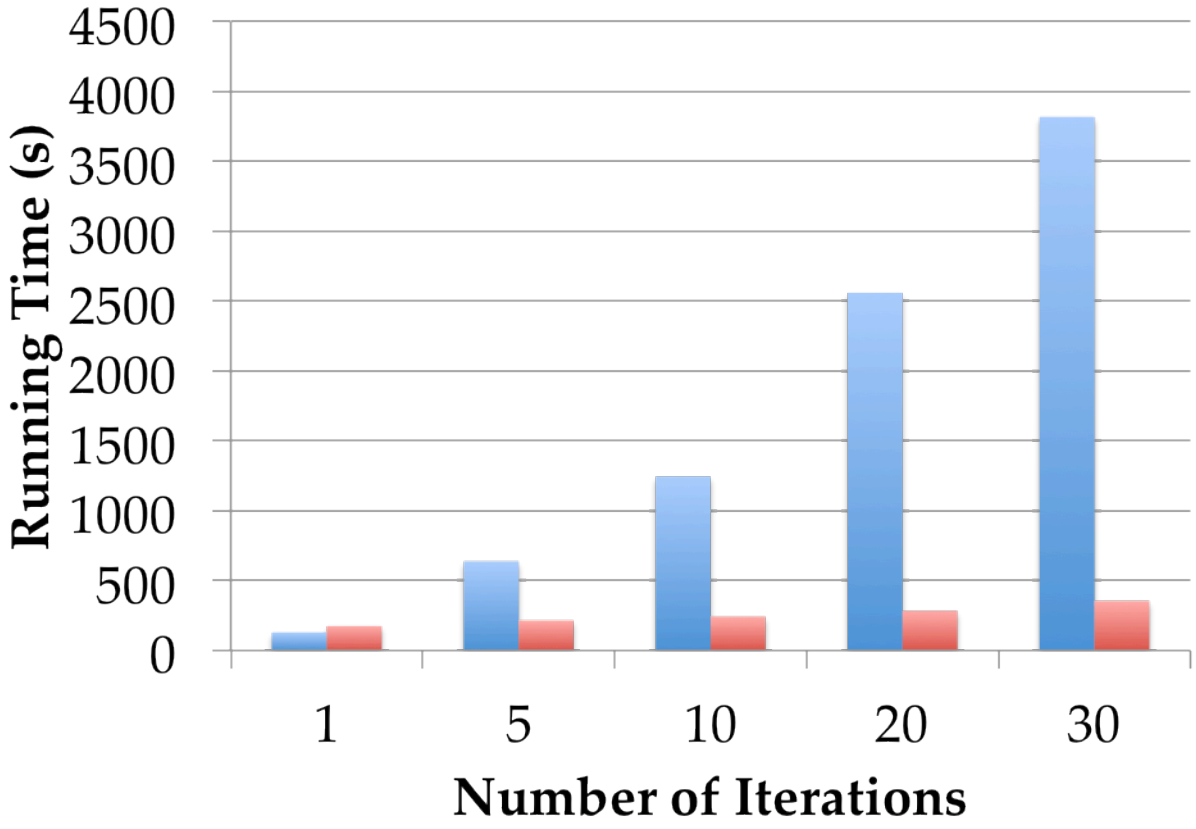
Find best line separating two sets of points



LINEAR 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)
```

LINEAR REGRESSION PERFORMANCE



127 s / iteration

Hadoop

Spark

first iteration 174 s
further iterations 6 s

OTHER PROJECTS

Hive on Spark (SparkSQL): SQL engine

Spark Streaming: incremental processing with in-memory state

MLLib: Machine learning library

GraphX: Graph processing on top of Spark

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/