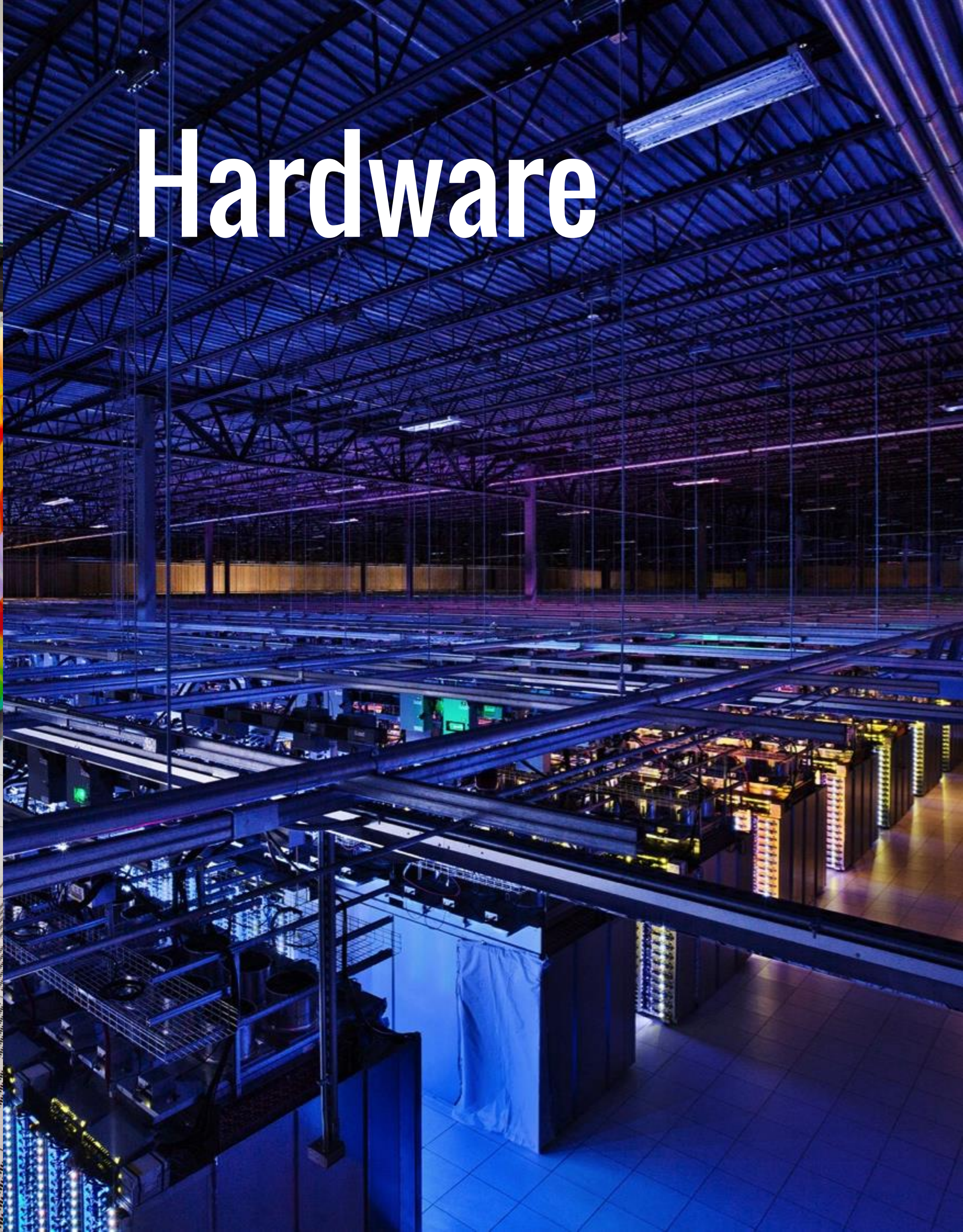
Cloud Computing and Big Data Processing

Shivaram Venkataraman
UC Berkeley, AMP Lab

Slides from Matei Zaharia

Cloud Computing, Big Data

Hardware



Software

Open MPI

UPC

hadoop

Spark

Google 1997

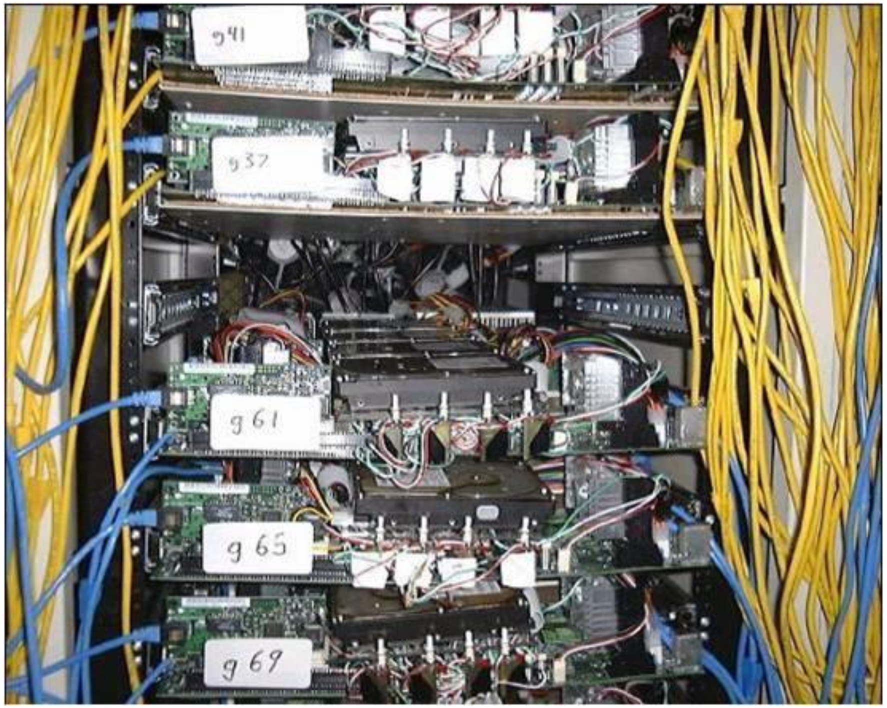
Data, Data, Data

"…Storage space must be used efficiently to store indices and, optionally, the documents themselves. The indexing system must process hundreds of gigabytes of data efficiently…"

The Anatomy of a Large-Scale Hypertextual Web Search Engine

Sergey Brin and Lawrence Page

Google 2001



Commodity CPUs

Lots of disks

Low bandwidth network

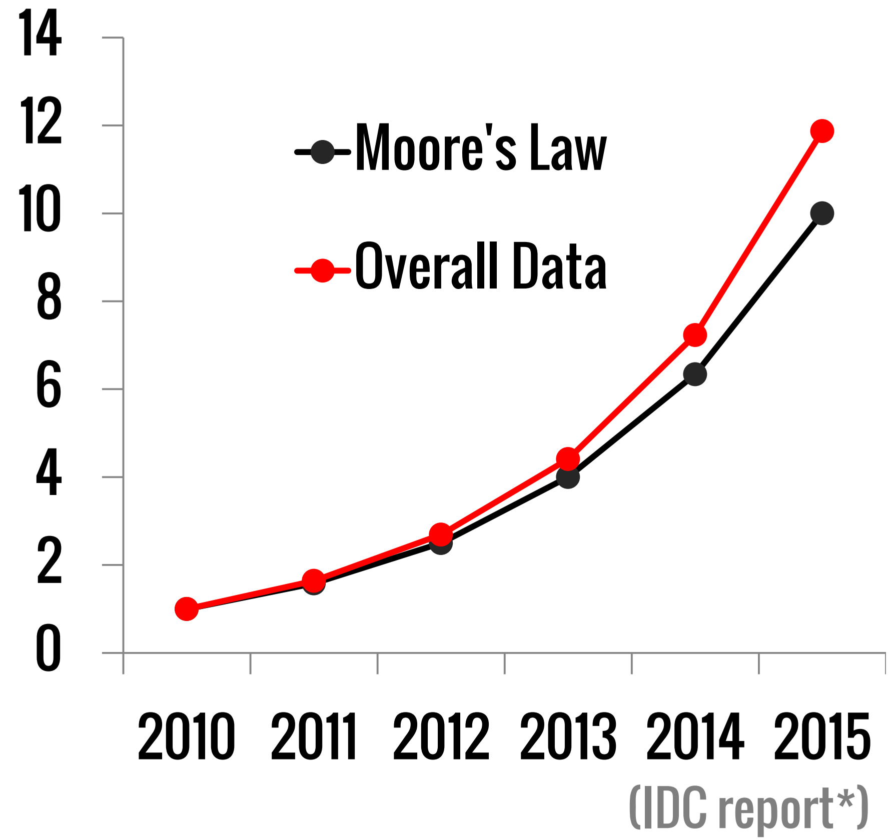
Cheap !

Datacenter evolution

Facebook's daily logs: 60 TB

1000 genomes project: 200 TB

Google web index: 10+ PB



Slide from Ion Stoica

Datacenter Evolution

Google data centers in The Dalles, Oregon

Datacenter Evolution

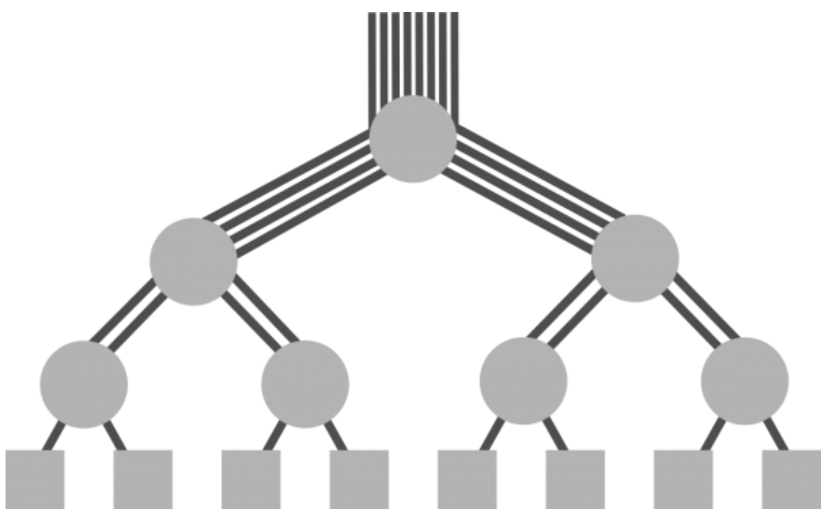
Capacity:
~10000 machines

Bandwidth:
12-24 disks per node

Latency:
256GB RAM cache

Datacenter Networking

Initially tree topology
Over subscribed links



Fat tree, Bcube, VL2 etc.

Lots of research to get
full bisection bandwidth

Datacenter Design

Goals

Power usage effectiveness (PUE)

Cost-efficiency

Custom machine design

Open Compute Project
(Facebook)

Datacenters → Cloud Computing

Above the Clouds: A Berkeley View of Cloud Computing

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz,
Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia
(Comments should be addressed to abovetheclouds@cs.berkeley.edu)

UC Berkeley Reliable Adaptive Distributed Systems Laboratory *
http://radlab.cs.berkeley.edu/

RAD Lab

"…long-held dream of computing as a utility…"

From Mid 2006

Rent virtual computers in the "Cloud"

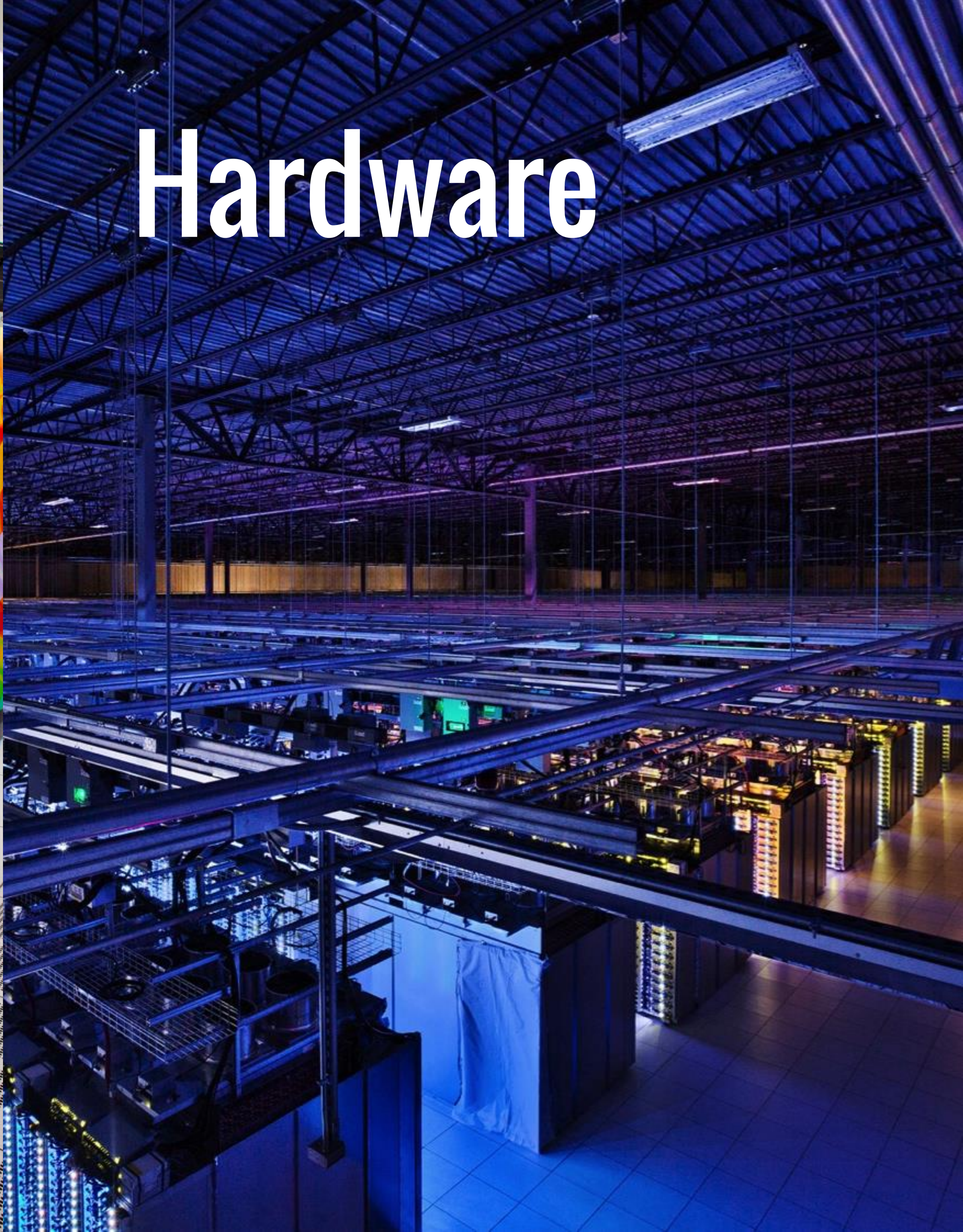
On-demand machines, spot pricing

Amazon EC2

Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t1.micro	0.615	2	0	$0.02
m1.xlarge	15	8	1680	$0.48
cc2.8xlarge	60.5	88 (Xeon 2670)	3360	$2.40

1 ECU = CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor

Hardware



Hopper vs. Datacenter

	Hopper	Datacenter[2]
Nodes	6384	1000s to 10000s
CPUs (per node)	2x12 cores	~2x6 cores
Memory (per node)	32-64GB	~48-128GB
Storage (overall)	~4 PB	120-480 PB
Interconnect	~ 66.4 Gbps	~10Gbps

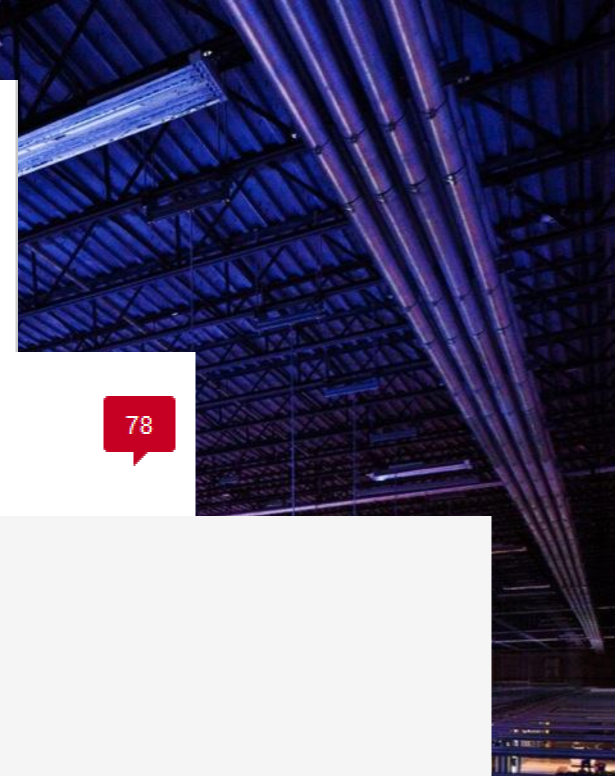
[2]http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/

Summary

Focus on Storage vs. FLOPS

Scale out with commodity components

Pay-as-you-go model



Outage in Dublin Knocks Amazon, Microsoft Data Centers Offline

By: Rich Miller

August 7th, 2011

557

Like

A lightning s

for Amazon

many sites

Microsoft's I

Dallas-Fort Worth Data Center Update

78

Filed in

on July 9th, 200

Tweet 0

Message from Ra

July 9, 2009

Rackspace Commu

Some of our custo

Worth Data Cente

interruption like thi

such incidents fror

Official Gmail Blog

News, tips and tricks from Google's Gmail team and friends.

Sign U

More

Posted:

Posted

Gmail's

people r

problem

and we'

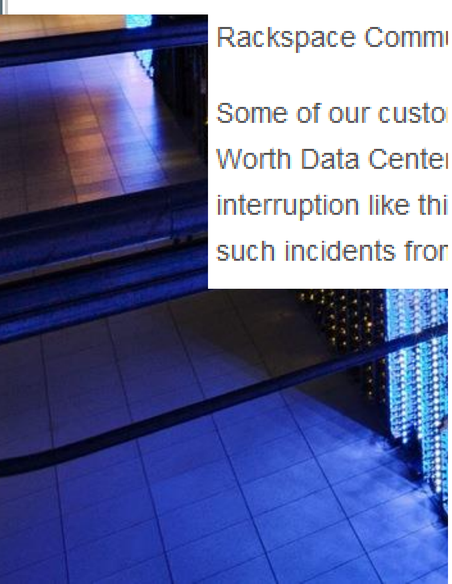
a list of

Entire Site

Amazon EC2 and Amazon RDS Service Disruption

nctionality to all affected services, we would like to share more details with our customers about the events t

our efforts to restore the services, and what we are doing to prevent this sort of issue from happening agai



The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)

~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures

~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Jeff Dean @ Google


How do we program this ?

Programming Models

Message Passing Models (MPI)

Fine-grained messages + computation

Hard to deal with disk locality, failures, stragglers

1 server fails every 3 years →

10K nodes see 10 faults/day

Programming Models

Data Parallel Models

Restrict the programming interface

Automatically handle failures, locality etc.

"Here's an operation, run it on all of the data"

- I don't care *where* it runs (you schedule that)
- In fact, feel free to run it *retry* on different nodes

MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

ce is a programming model and an associ-
entation for processing and generating large
sers specify a *map* function that processes a
ir to generate a set of intermediate key/value
educe function that merges all intermediate
ated with the same intermediate key. Many
sks are expressible in this model, as shown

given day, etc. Most such computations a
ally straightforward. However, the input da
large and the computations have to be distri
hundreds or thousands of machines in orde
a reasonable amount of time. The issues of
allelize the computation, distribute the data
failures conspire to obscure the original sir
tation with large amounts of complex code
these issues.

Google 2004

Build search index
Compute PageRank

Hadoop: Open-source
at Yahoo, Facebook

MapReduce Programming Model

Data type: Each record is (key, value)

Map function:

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

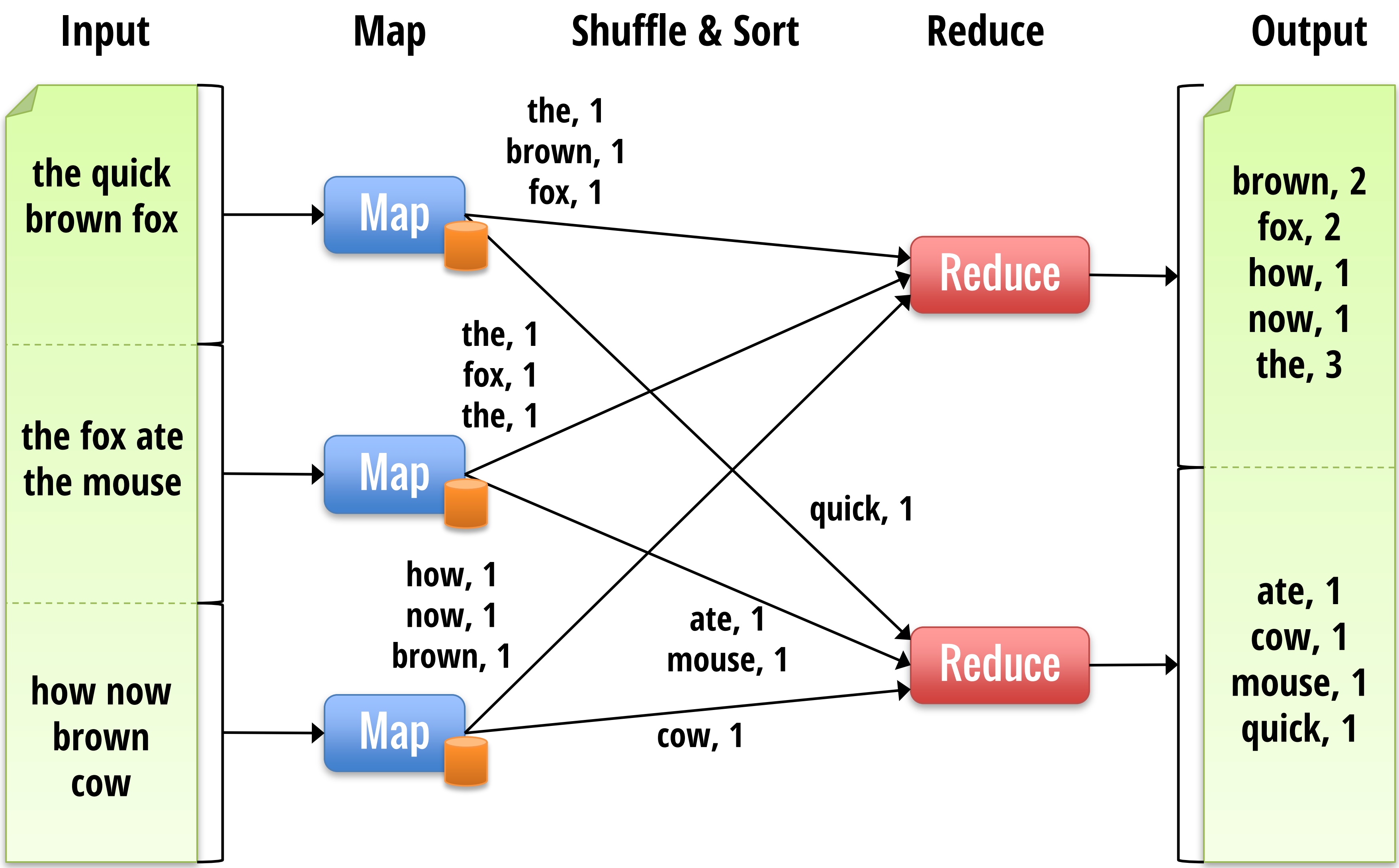
Reduce function:

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

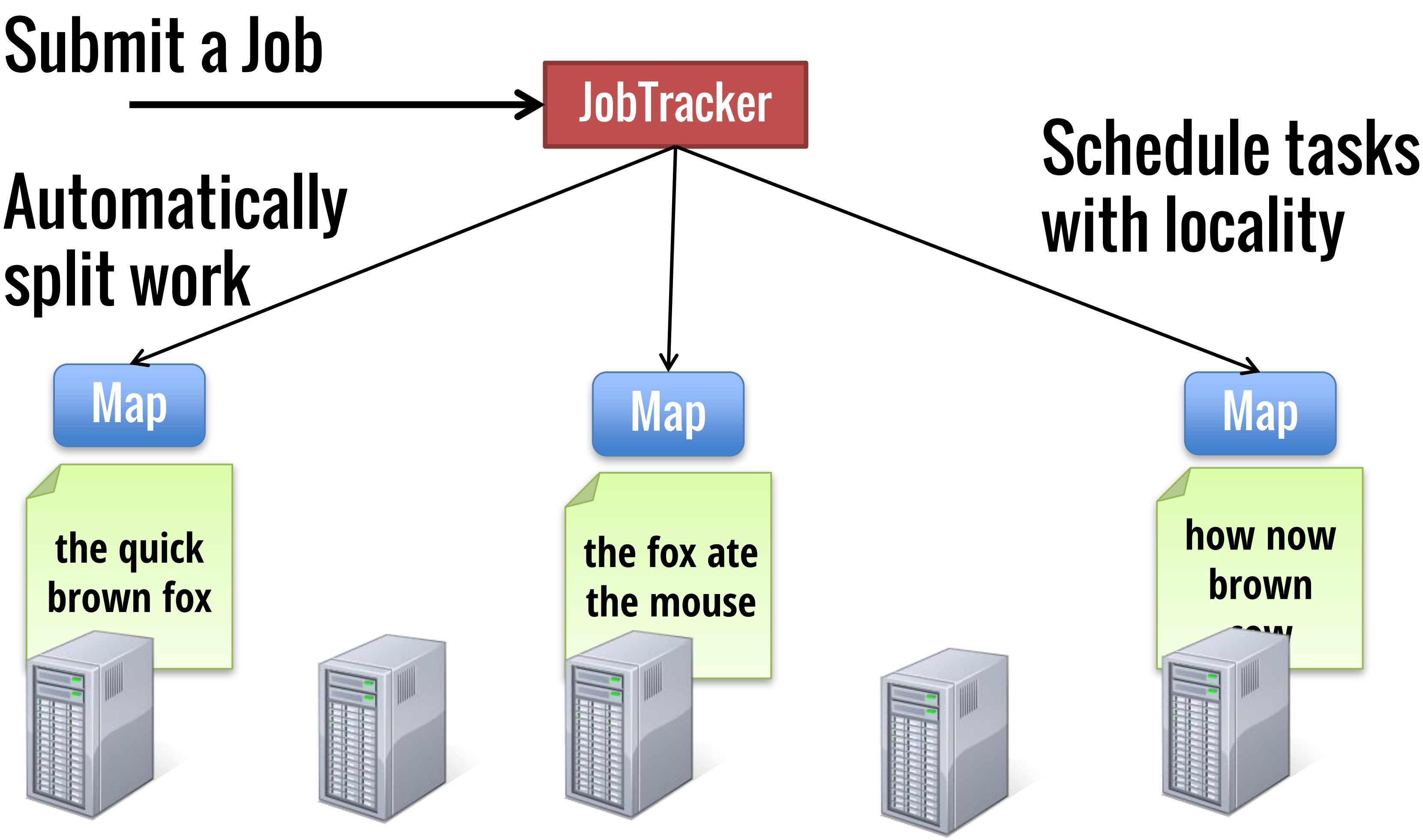
Example: Word Count

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


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

Word Count Execution



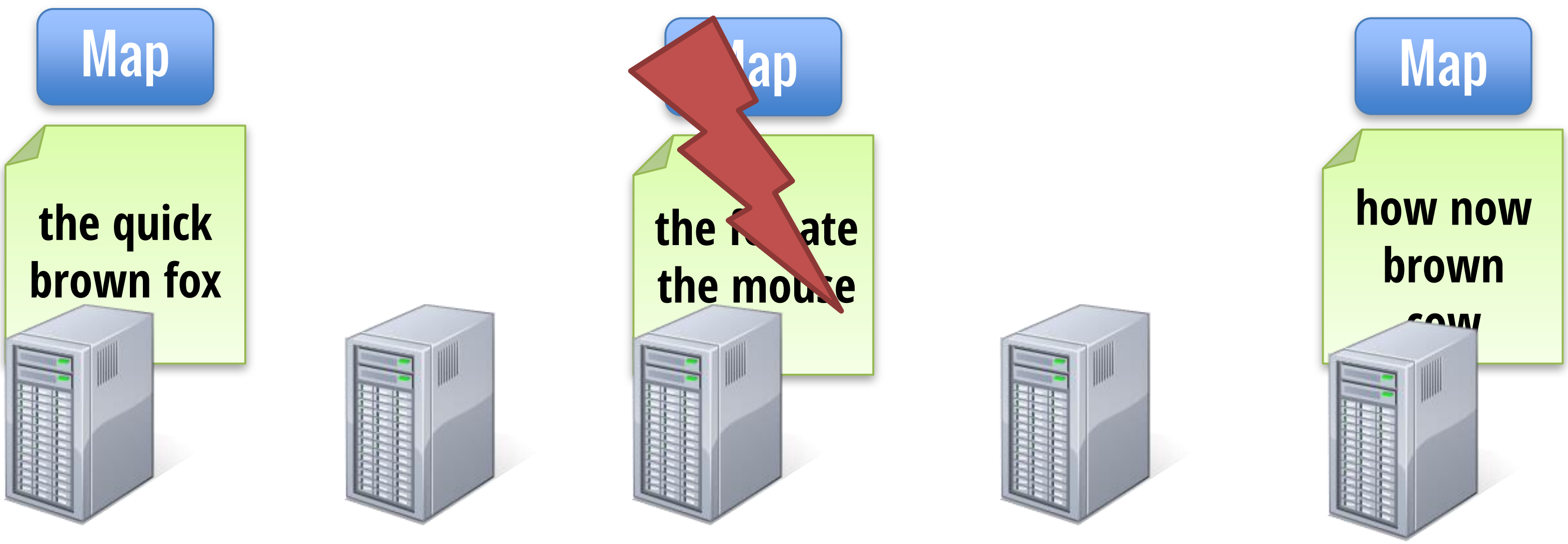
Word Count Execution



Fault Recovery

If a task crashes:

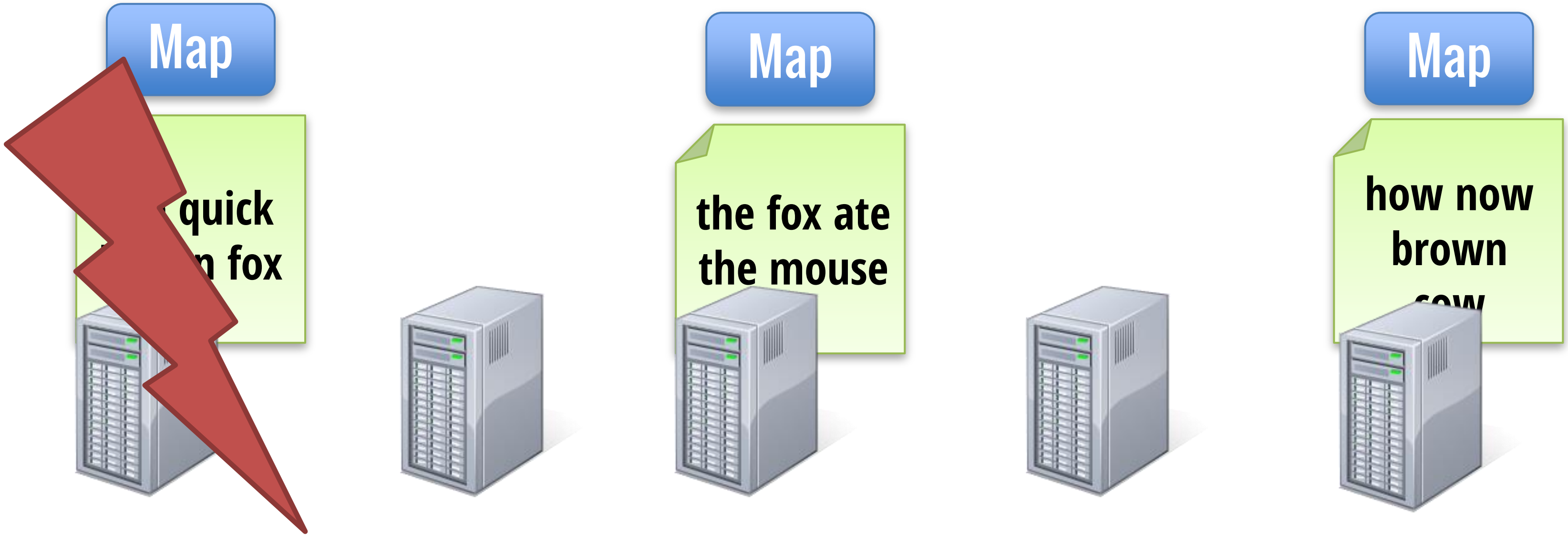
- Retry on another node
- If the same task repeatedly fails, end the job



Requires user code to be **deterministic**

Fault Recovery

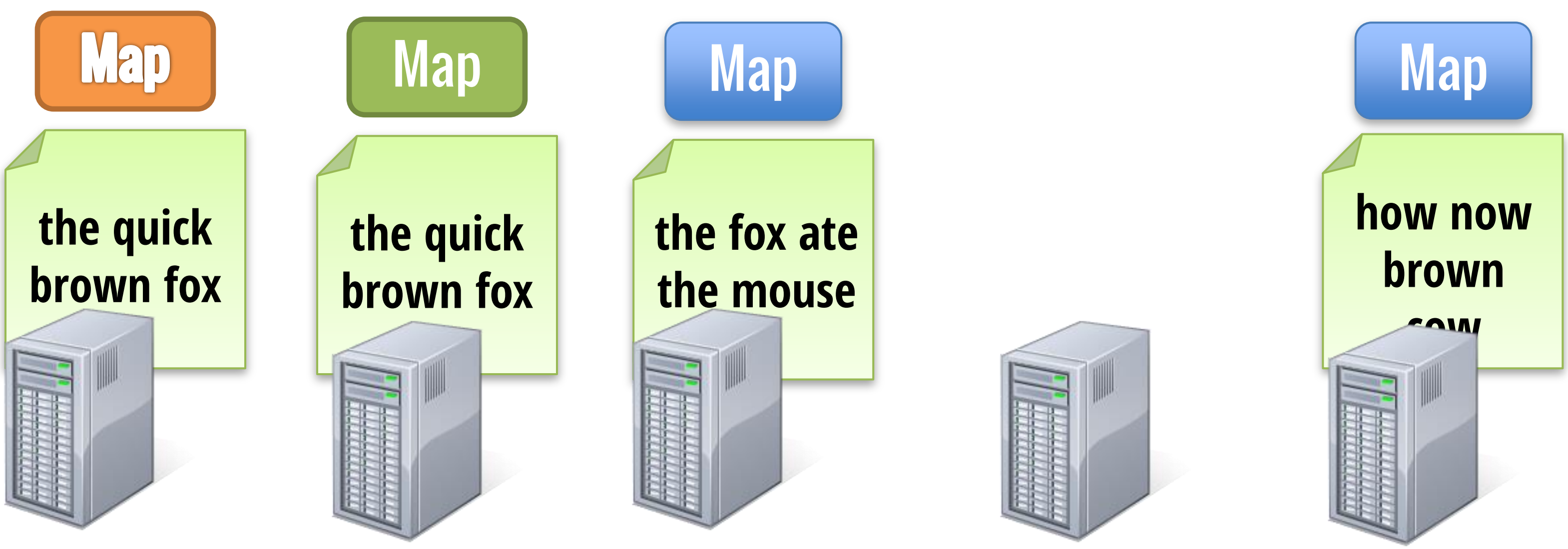
If a node crashes:
- Relaunch its current tasks on other nodes
- Relaunch tasks whose outputs were lost



Fault Recovery

If a task is going slowly (straggler):

- Launch second copy of task on another node
- Take the output of whichever finishes first



Applications

1. Search

Input: (lineNumber, line) records

Output: lines matching a given pattern

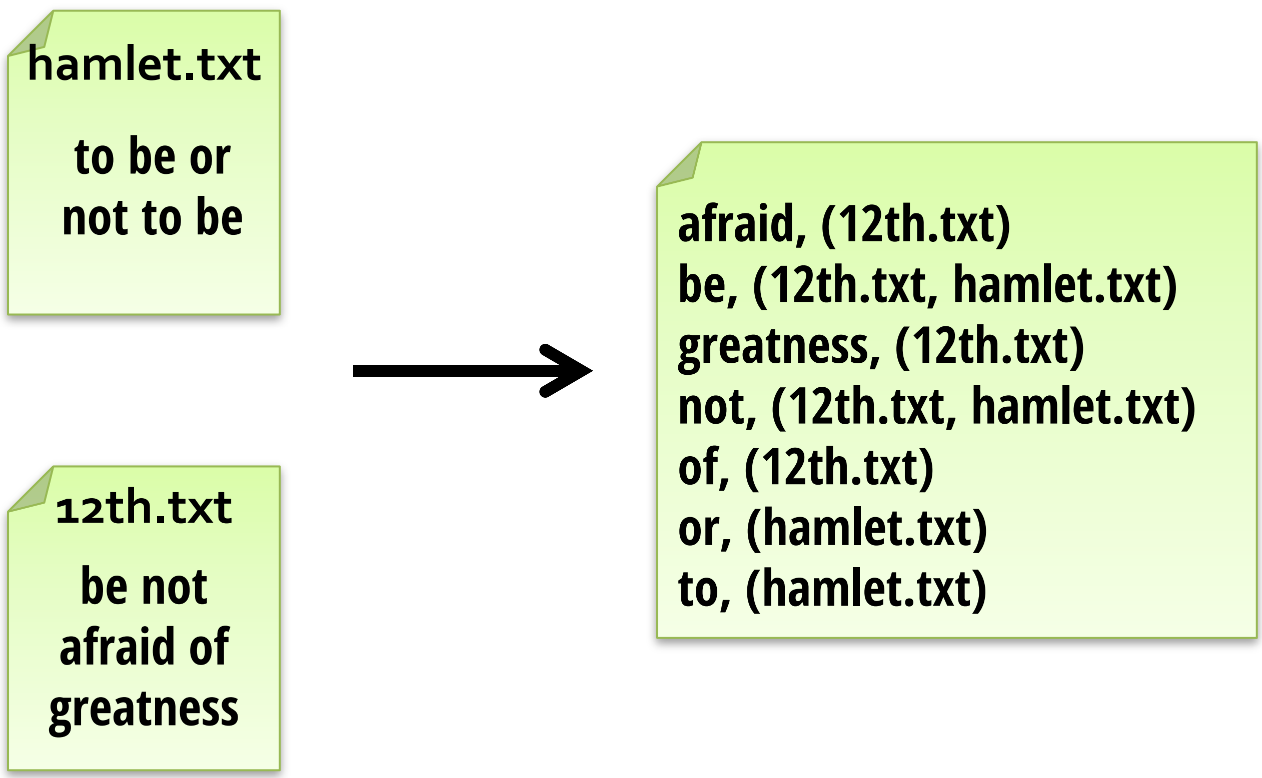
Map:

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

Reduce: Identity function

- Alternative: no reducer (map-only job)

2. Inverted Index



2. Inverted Index

Input: (filename, text) records

Output: list of files containing each word

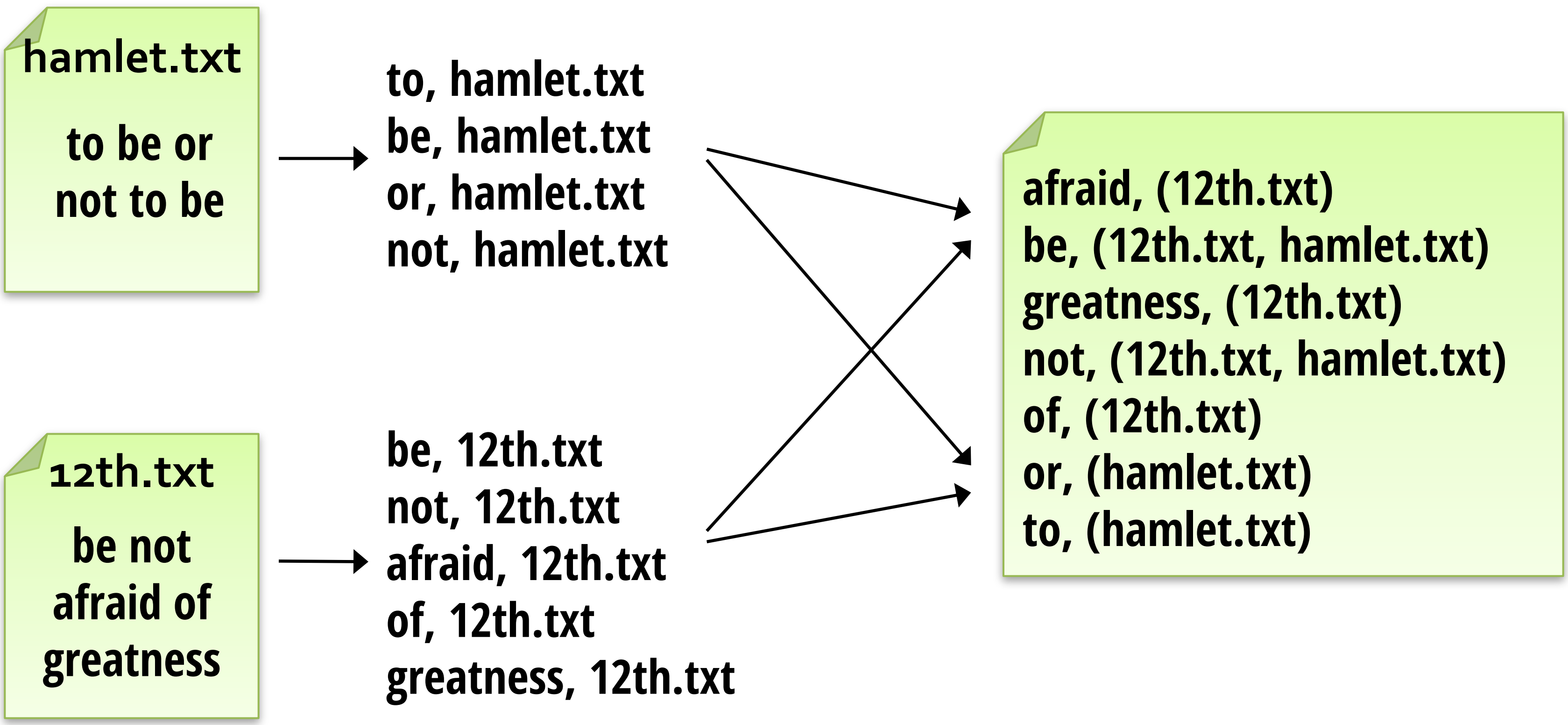
Map:

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

Reduce:

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

2.Inverted Index



MPI

- Parallel process model
- Fine grain control
- High Performance

MapReduce

- High level data-parallel
- Automate locality, data transfers
- Focus on fault tolerance

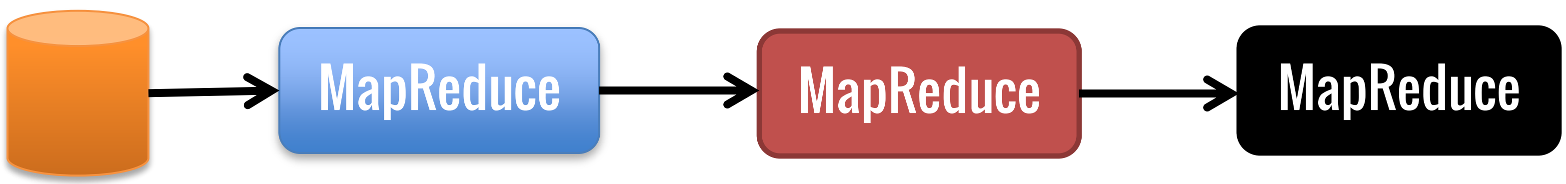
Summary

MapReduce data-parallel model
Simplified cluster programming

Automates
- Division of job into tasks
- Locality-aware scheduling
- Load balancing
- Recovery from failures & stragglers

When an Abstraction is Useful…

People want to compose it!



Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. sessions, top K): 2-5 steps
- Iterative algorithms (e.g. PageRank): 10's of steps

Programmability

Multi-step jobs create spaghetti code

- 21 MR steps → 21 mapper and reducer classes

Lots of boilerplate wrapper code per step

API doesn't provide type safety

Performance

MR only provides one pass of computation

- Must write out data to file system in-between

Expensive for apps that need to *reuse* data

- Multi-step algorithms (e.g. PageRank)
- Interactive data mining

Spark

Programmability: clean, functional API

- Parallel transformations on collections
- 5-10x less code than MR
- Available in Scala, Java, Python and R

Performance

- In-memory computing primitives
- Optimization across operators

Spark Programmability

Google MapReduce WordCount:

```
#include "mapreduce/mapreduce.h"

// User’s map function
class SplitWords: public Mapper {
  public:
  virtual void Map(const MapInput&
input)
  {
    const string& text =
input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while (i < n &&
isspace(text[i]))
        i++;
      // Find word end
      int start = i;
      while (i < n &&
!isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(
            start,i-start),"1");
    }
  }
};

REGISTER_MAPPER(SplitWords);
```

```
// User’s reduce function
class Sum: public Reducer {
  public:
  virtual void Reduce(ReduceInput*
input)
  {
    // Iterate over all entries with
the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(
                input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Sum);
```

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;
  for (int i = 1; i < argc; i++) {
    MapReduceInput* in=
spec.add_input();
    in->set_format("text");
    in->set_filepattern(argv[i]);
    in-
>set_mapper_class("SplitWords");
  }

  // Specify the output files
  MapReduceOutput* out =
spec.output();
  out-
>set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Sum");

  // Do partial sums within map
  out->set_combiner_class("Sum");

  // Tuning parameters
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result))
abort();
  return 0;
 }
```

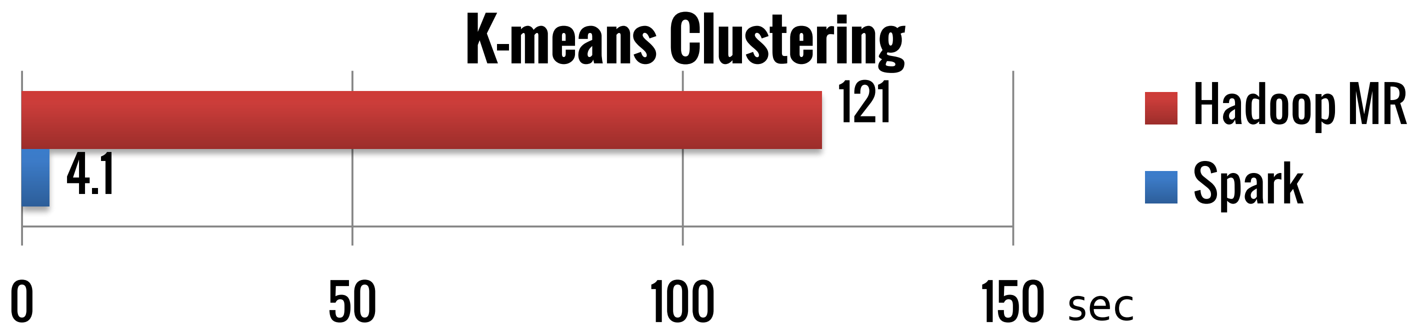
Spark Programmability

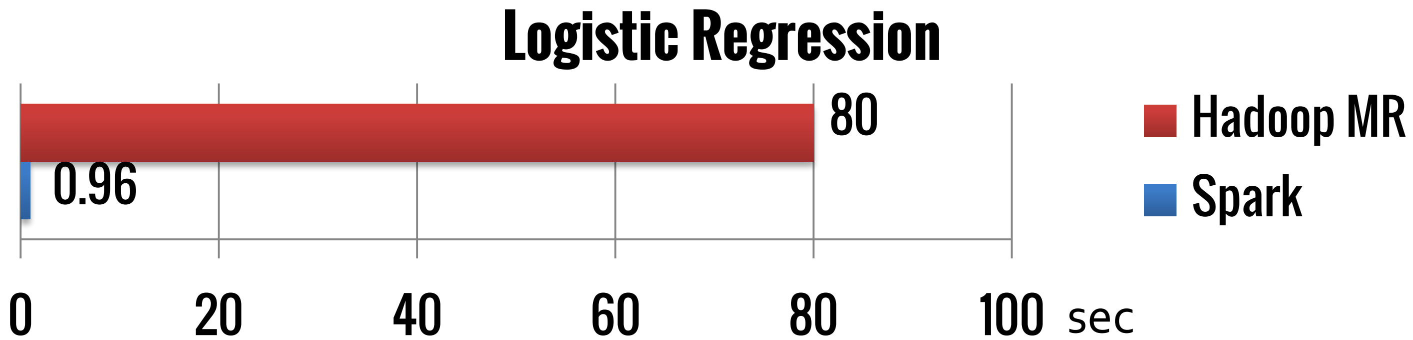
Spark WordCount:

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.save("out.txt")
```

Spark Performance

Iterative algorithms:





Spark Concepts

Resilient distributed datasets (RDDs)

- Immutable, partitioned collections of objects
- May be cached in memory for fast reuse

Operations on RDDs

- *Transformations* (build RDDs)
- *Actions* (compute results)

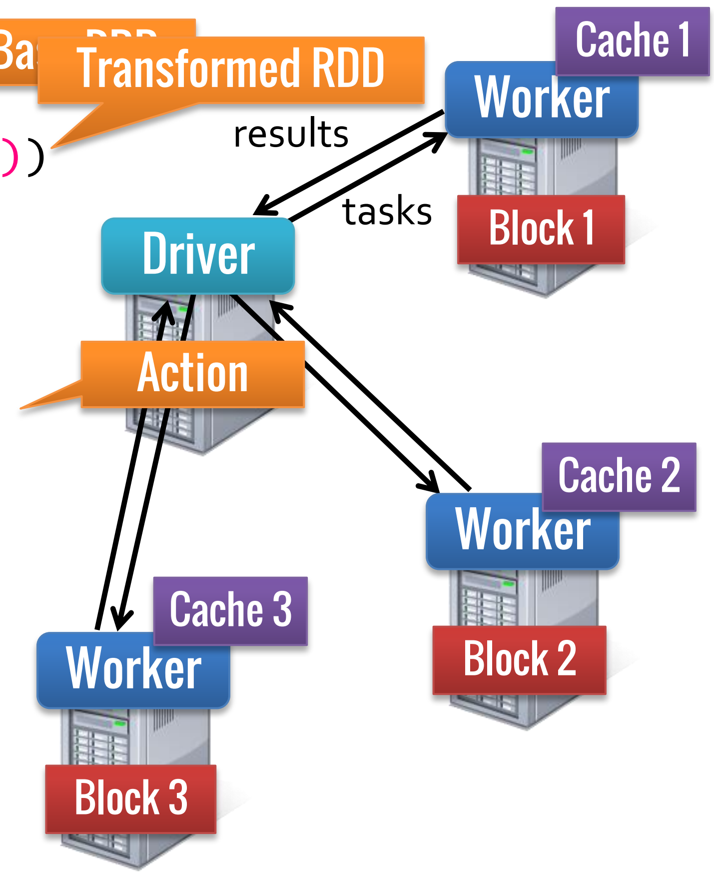
Restricted shared variables

- Broadcast, accumulators

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
. . .
```

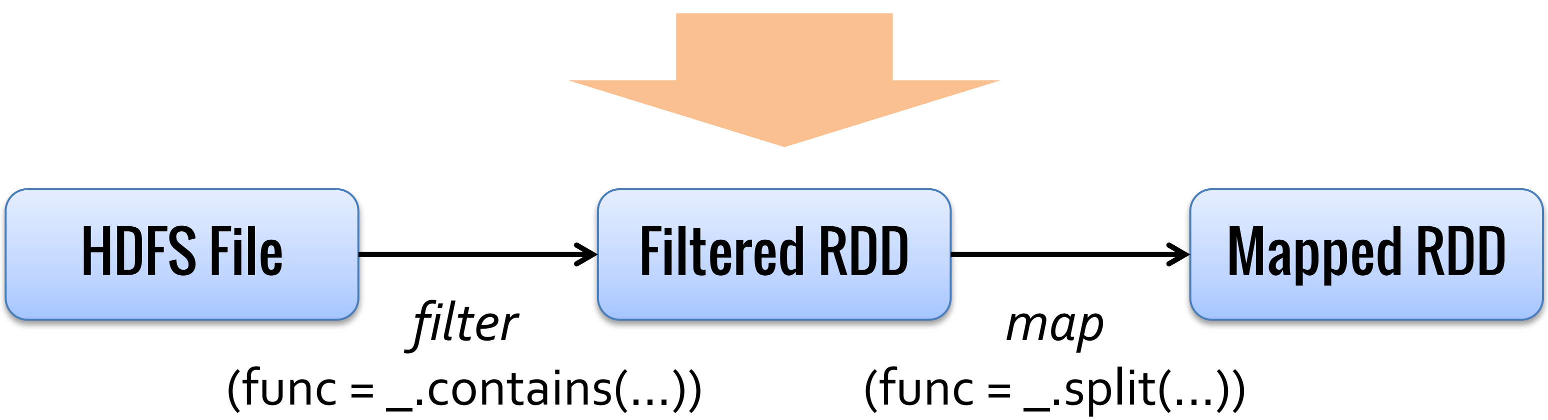


Result: search 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Fault Recovery

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

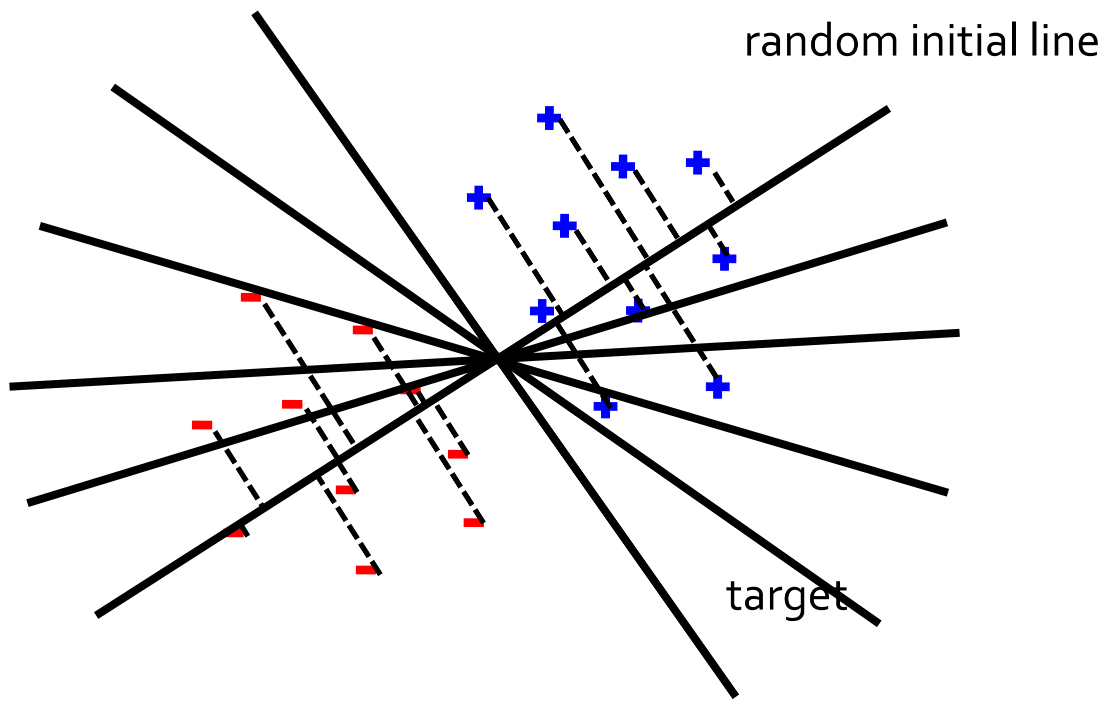
Ex:
```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



Demo

Example: Logistic Regression

Goal: find best line separating two sets of points

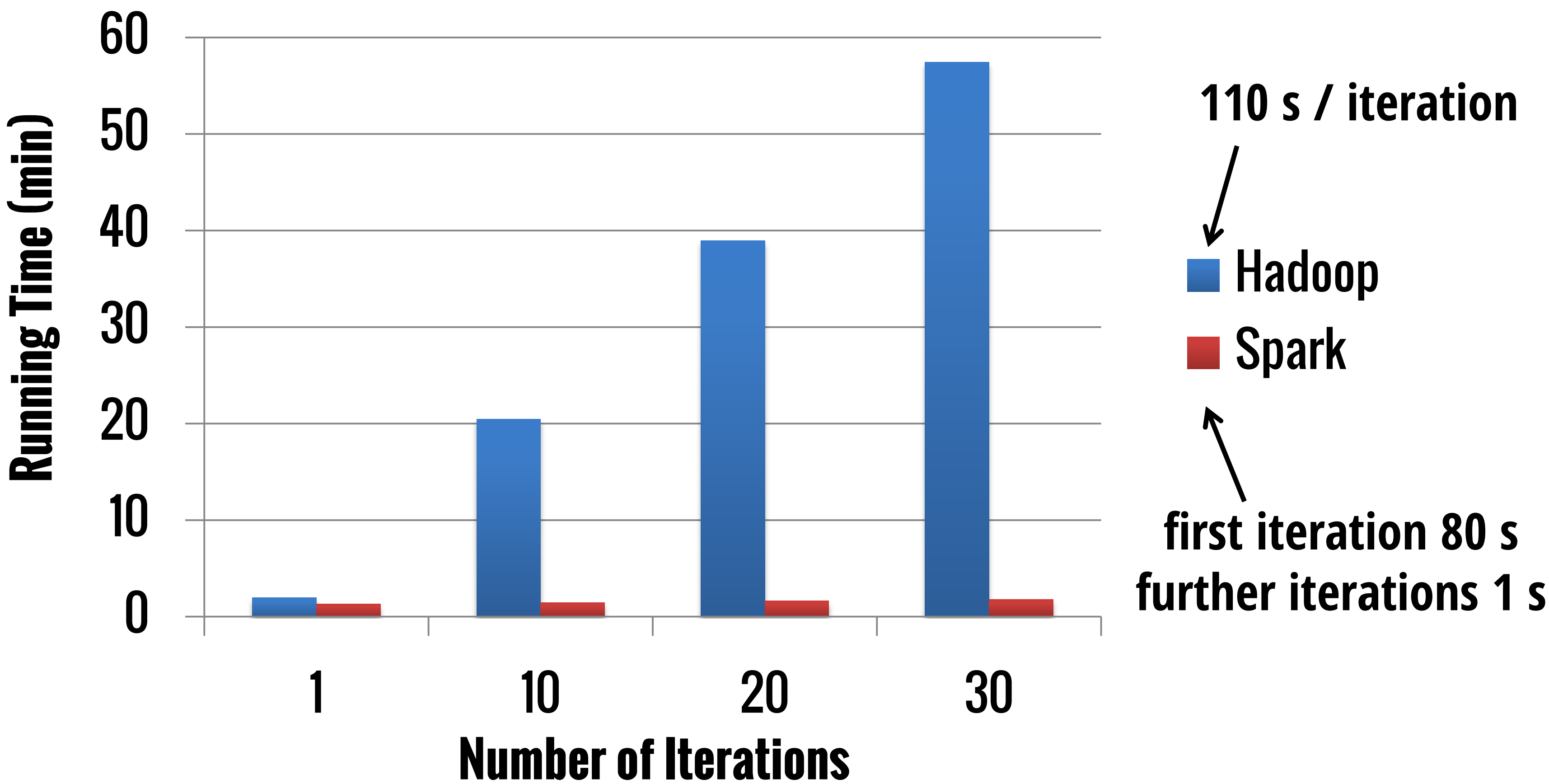


Example: Logistic Regression

```
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)
```

w automatically shipped to cluster

Logistic Regression Performance



Shared Variables

RDD operations: use local variables from scope

Two other kinds of shared variables:

- Broadcast Variables
- Accumulators

Broadcast Variables

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

// Random Projection
val M = Matrix.random(N)

var w = Vector.random(D)

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

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

Large Matrix

Problem:
M re-sent to all nodes in each iteration

Broadcast Variables

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

// Random Projection
Val M = spark.broadcast(Matrix.random(N))

var w = Vector.random(D)

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

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

Solution: mark M as broadcast variable

Other RDD Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey cogroup	flatMap union join cross mapValues ...
Actions (output a result)	collect reduce take fold	count saveAsTextFile saveAsHadoopFile ...

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Python

```
lines = sc.textFile(...)
lines.filter(lambda x: "error" in x).count()
```

R

```
lines <- textFile(sc, ...)
filter(lines, function(x) grepl("error", x))
```

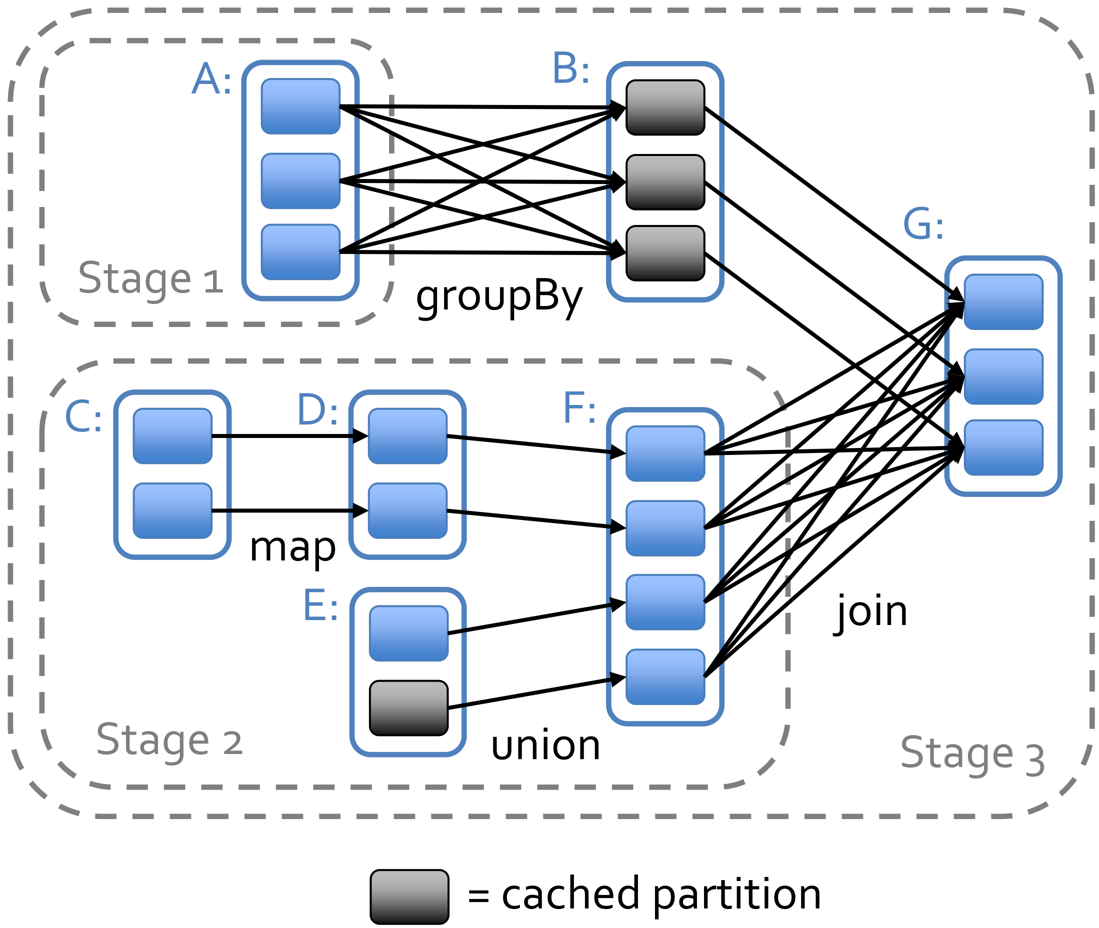
Job Scheduler

Captures RDD dependency graph

Pipelines functions into "stages"

Cache-aware for data reuse & locality

Partitioning-aware to avoid shuffles



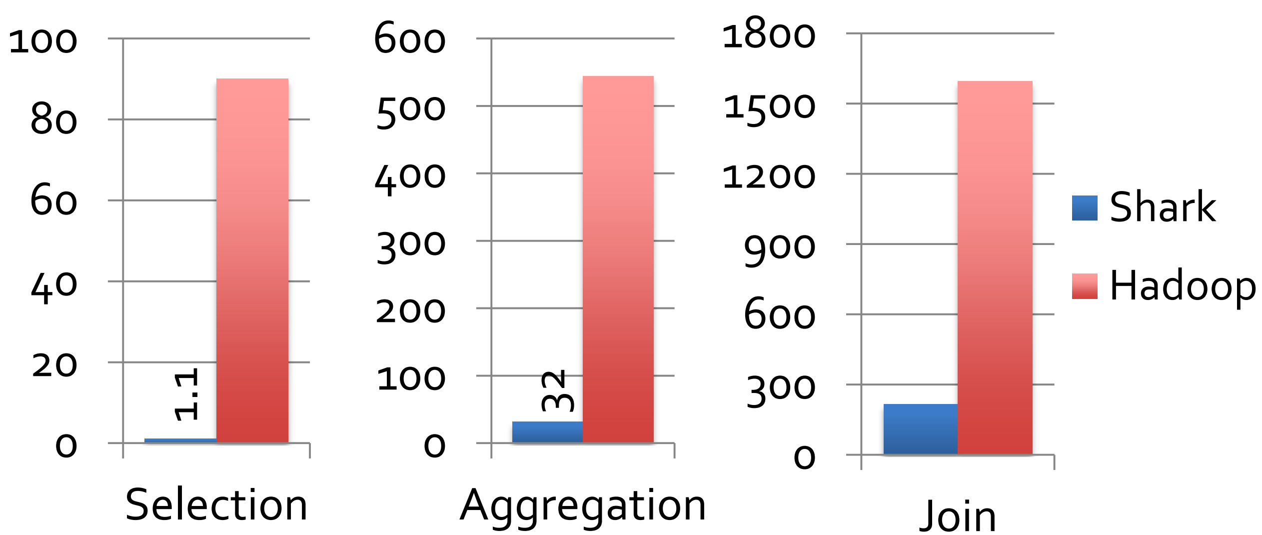
Higher-Level Abstractions

SparkStreaming: API for streaming data

GraphX: Graph processing model

MLLib: Machine learning library

Shark: SQL queries



ampcamp

Big Data Bootcamp

Hands-on Exercises using Spark, Shark etc.

~250 in person

3000 online

http://ampcamp.berkeley.edu

Course Project Ideas

Linear Algebra on commodity clusters

- Optimizing algorithms
- Cost model for datacenter topology

Measurement studies

- Comparing EC2 vs Hopper
- Optimizing BLAS for virtual machines

Conclusion

Commodity clusters needed for big data

Key challenges: Fault tolerance, stragglers

Data-parallel models: MapReduce and Spark

- Simplify programming
- Handle faults automatically