Cloud Computing with MapReduce and Hadoop

Matei Zaharia

UC Berkeley AMP Lab

matei@berkeley.edu

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

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>**

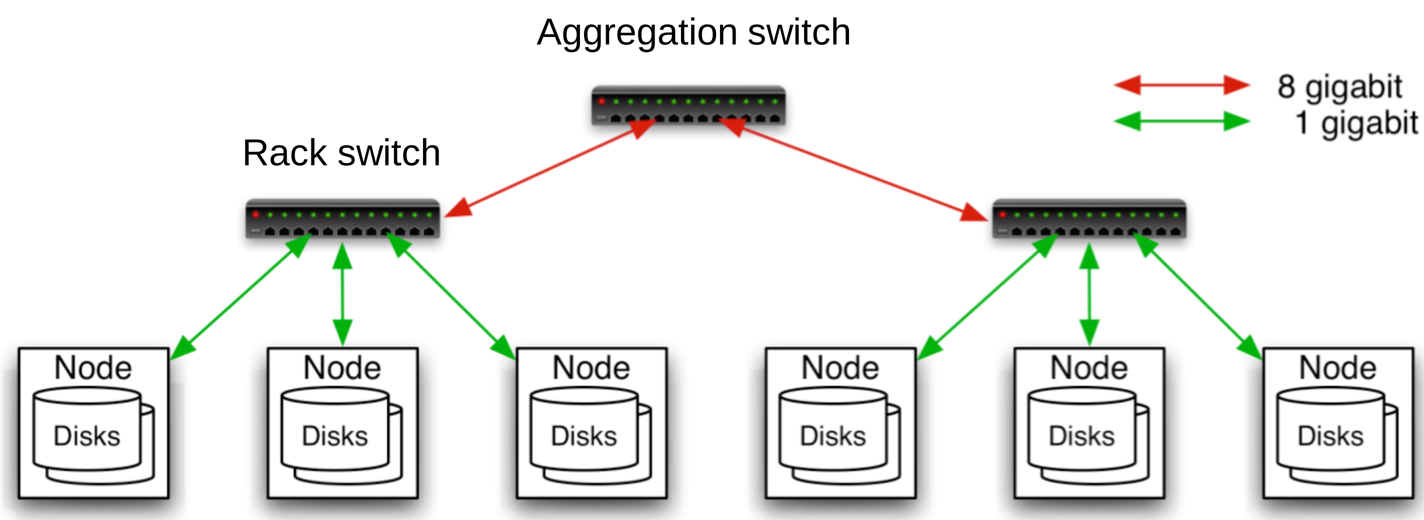
Outline

- 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

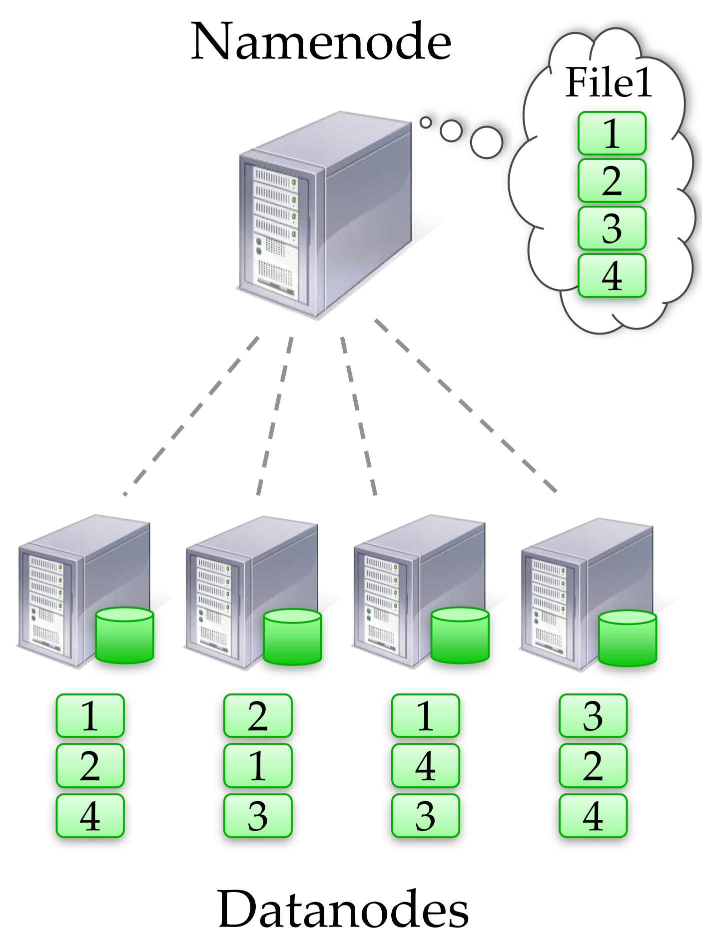
- Programming distributed systems is hard
  - **Solution:** Restricted programming model: users write data-parallel "map" and "reduce" functions, system handles work distribution and failures

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:

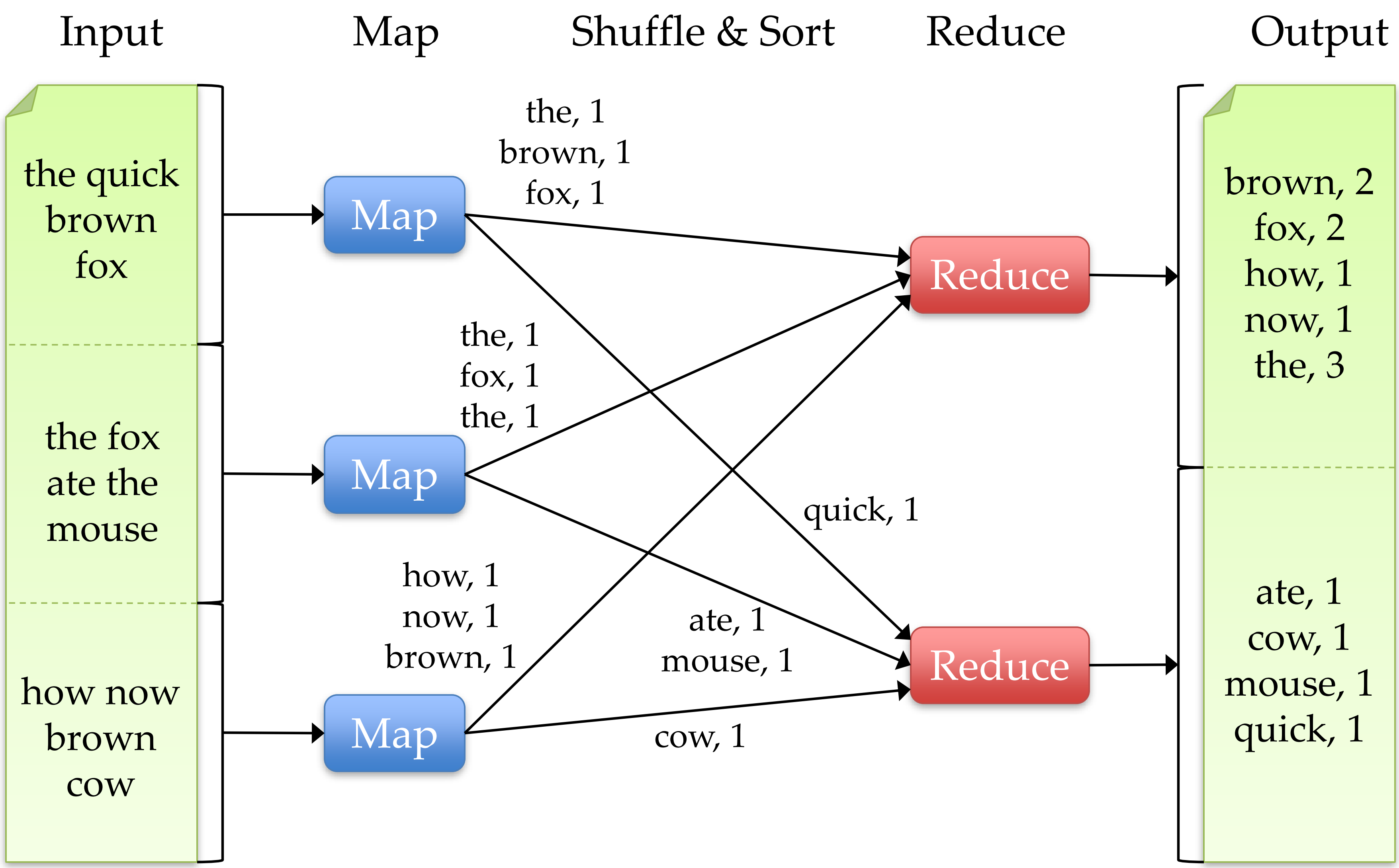
$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

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



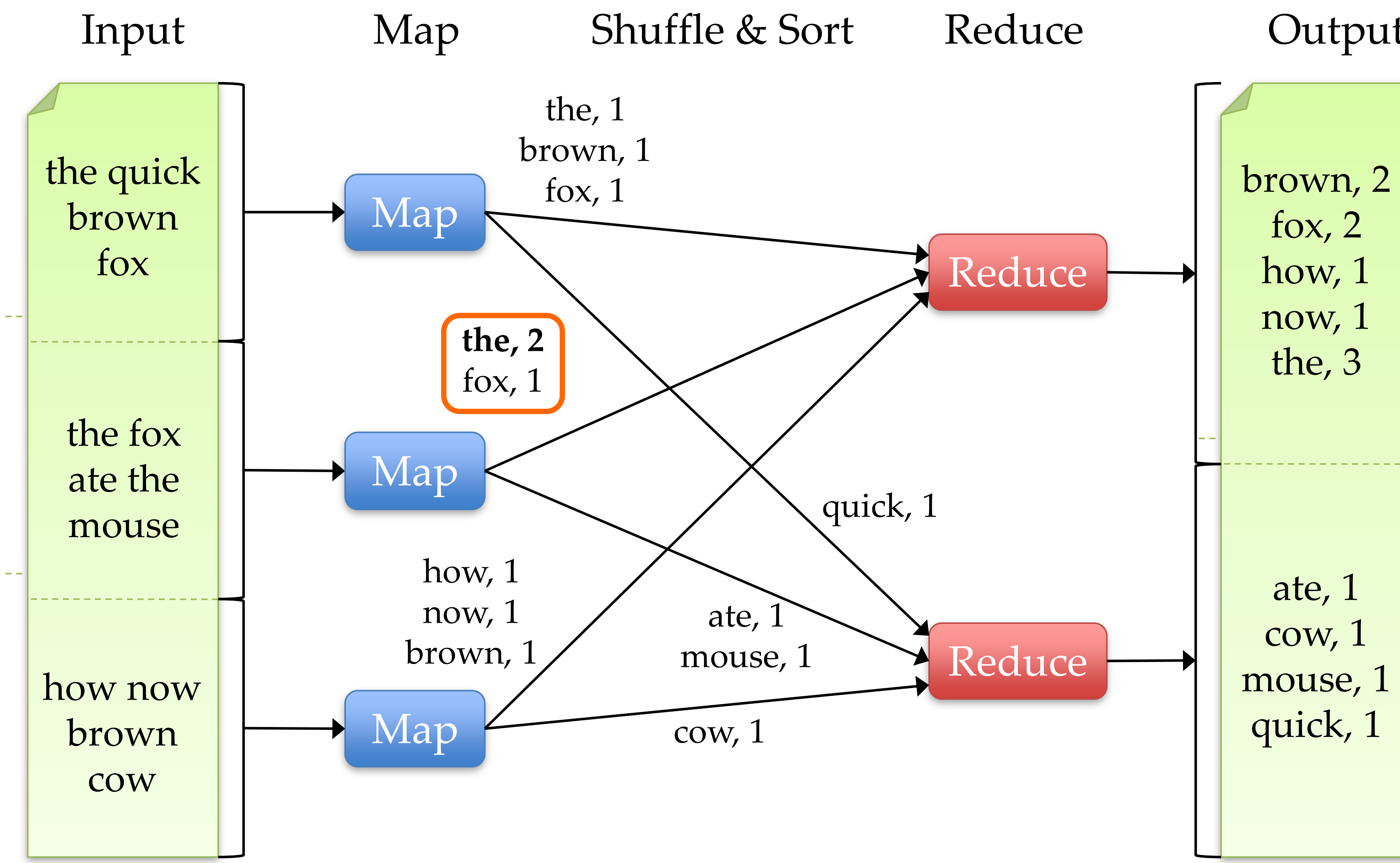
An Optimization: The Combiner

- 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)
```

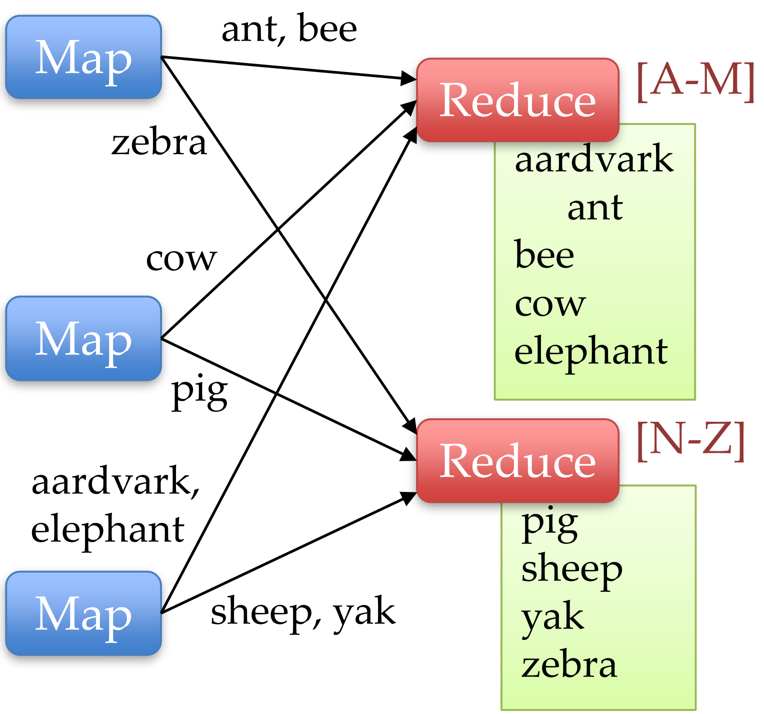
- **Reduce:** identity function
  - Alternative: no reducer (map-only job)

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:**

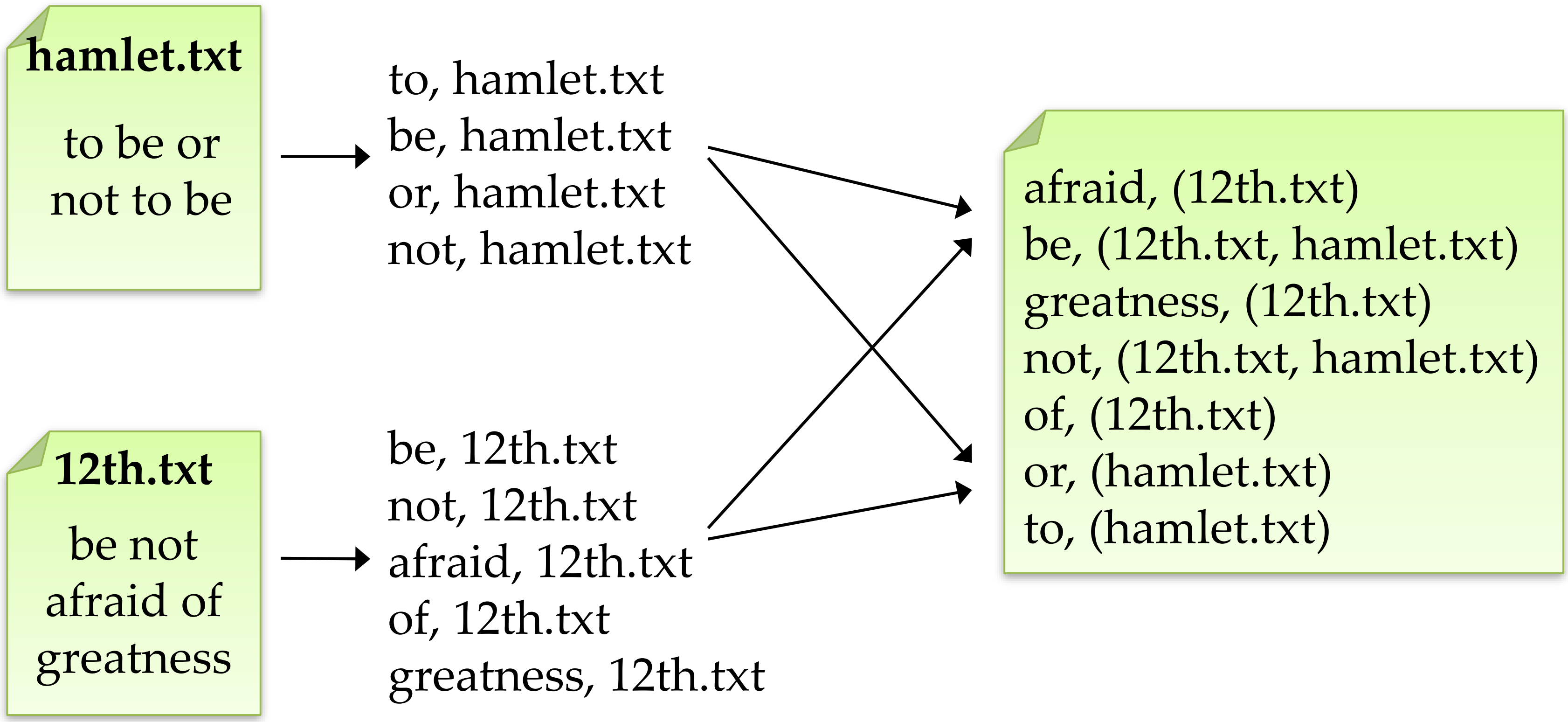
```
foreach word in text.split():
   output(word, filename)
```

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

- **Reduce:**

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

Inverted Index Example



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))
```

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**

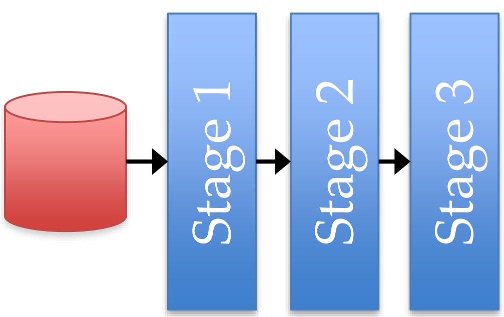
Spark Motivation

- 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

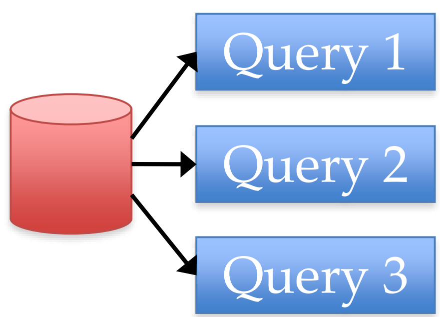
Spark Motivation

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

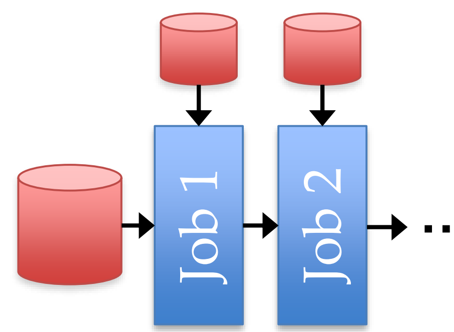
Efficient primitives for **data sharing**



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**

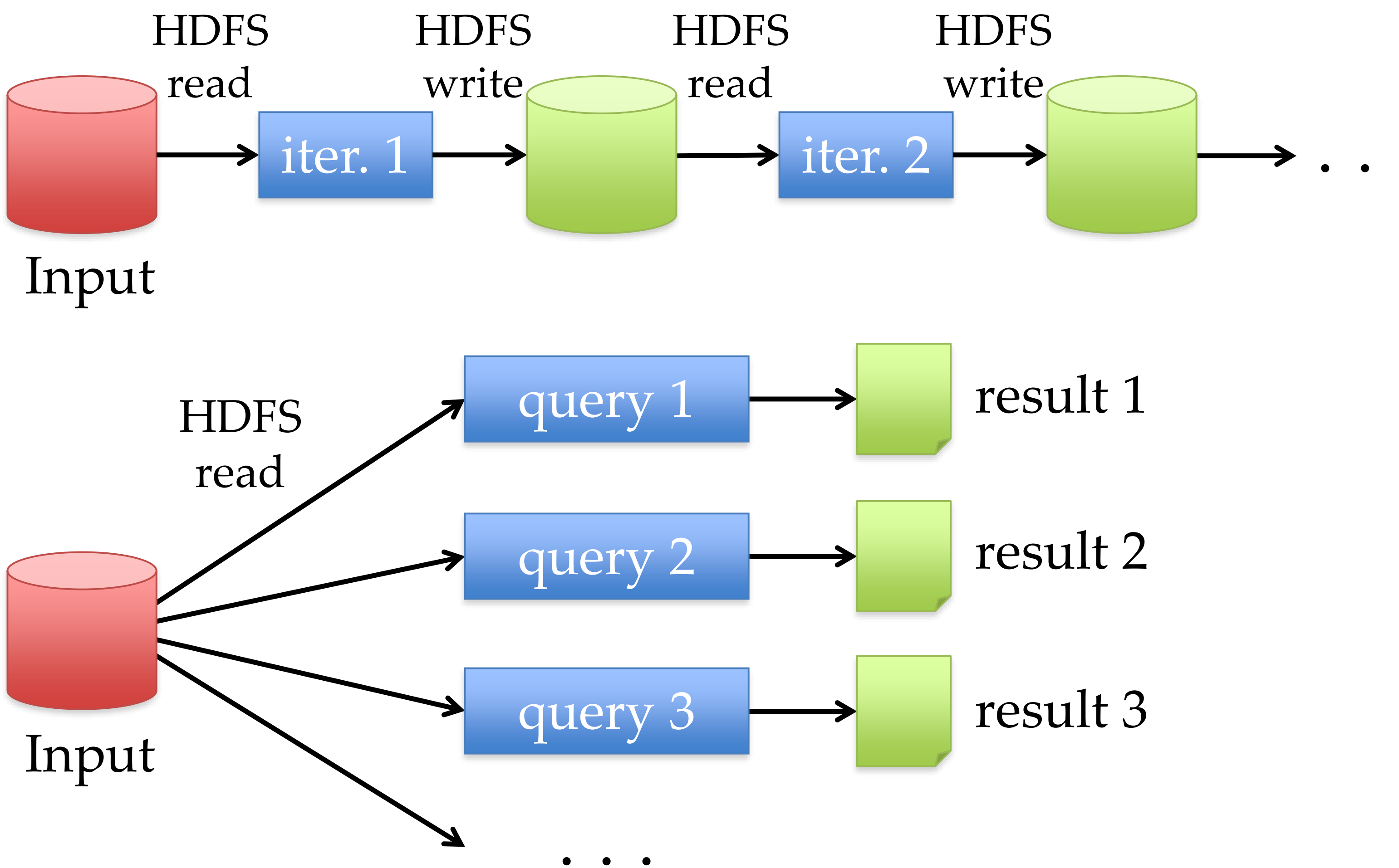
Problem: in MR, only way to share data across jobs is stable storage (e.g. file system) -> **slow!**

Iterative job

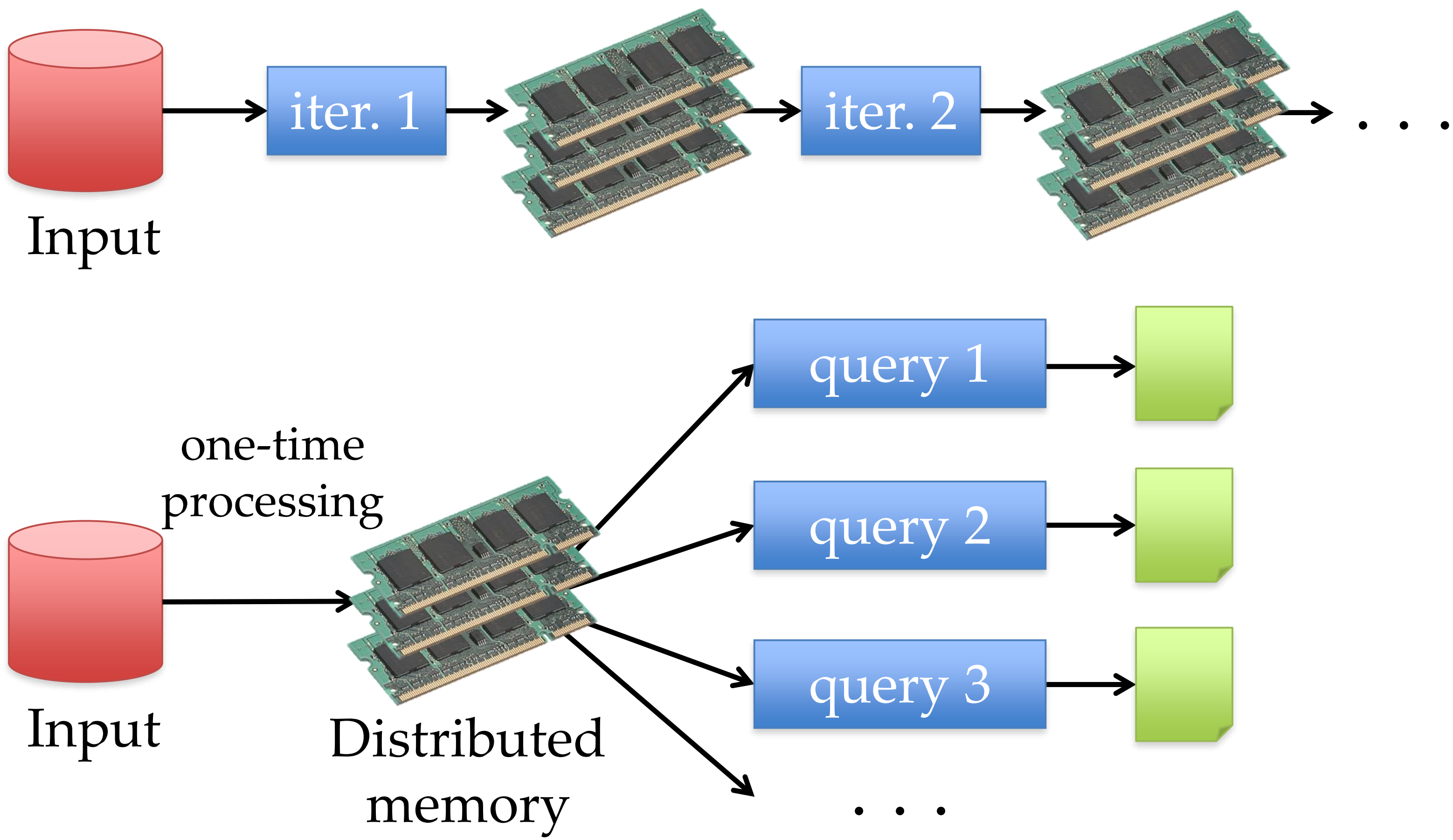
Interactive mining

Stream processing

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)

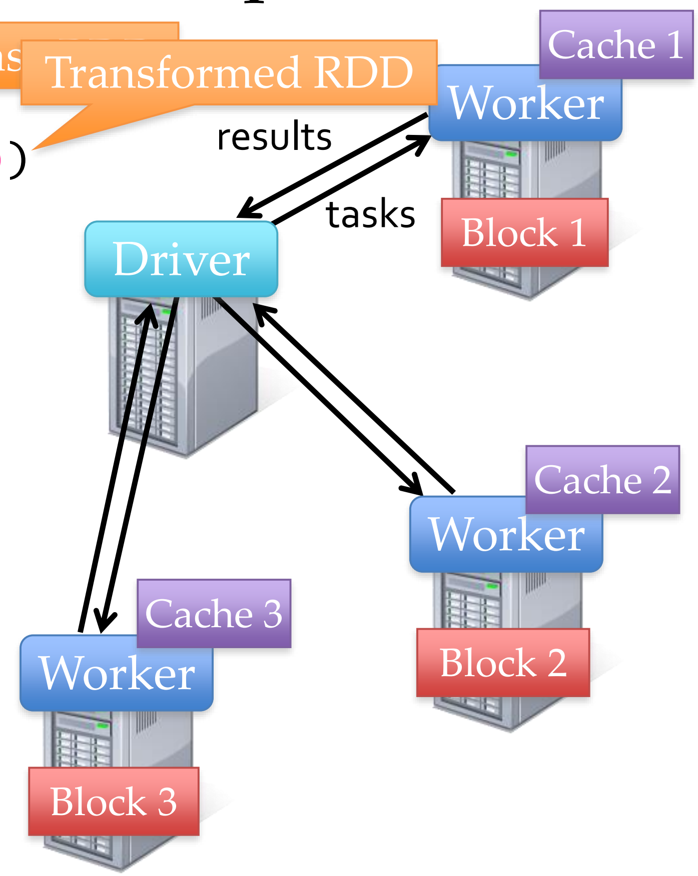
- Partitioned collections of records that can be stored in memory across the cluster
- Manipulated through a diverse set of transformations (*map, filter, join,* etc)
- Fault recovery without costly replication
  - Remember the series of transformations that built an RDD (its *lineage*) to *recompute* 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
. . .
```



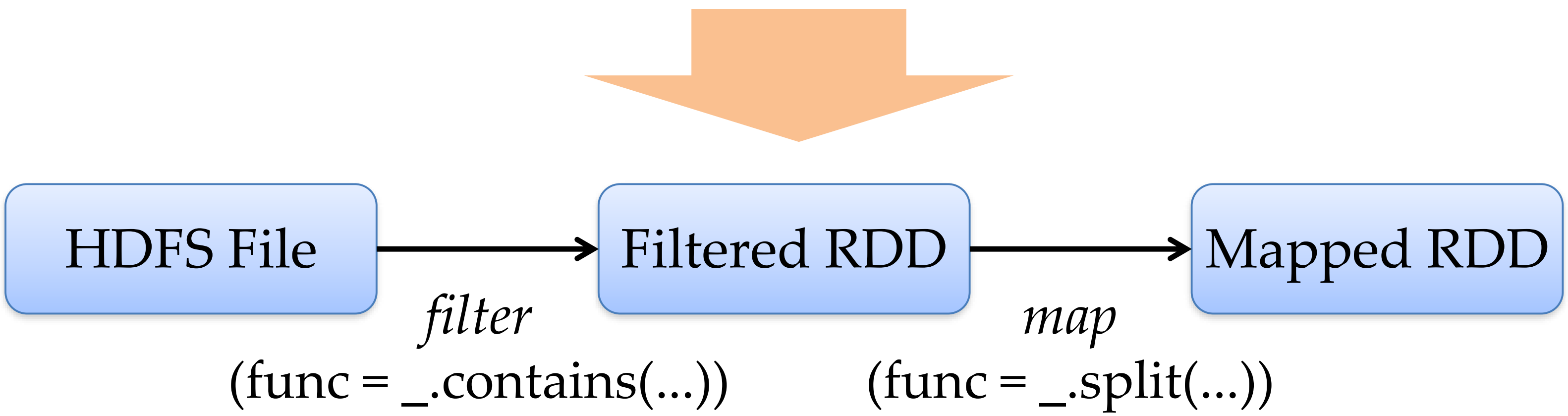
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Scala programming language

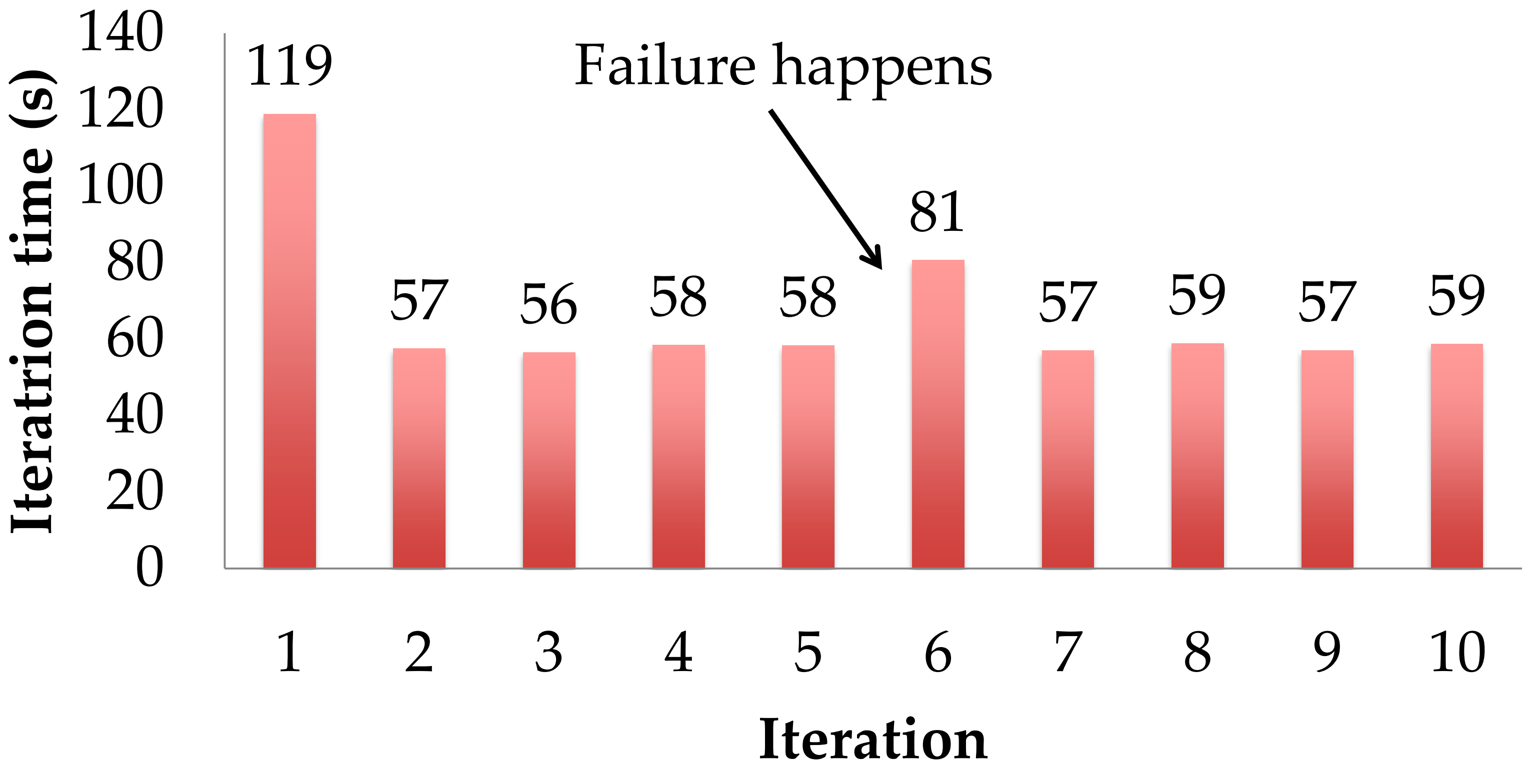
Fault Recovery

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

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

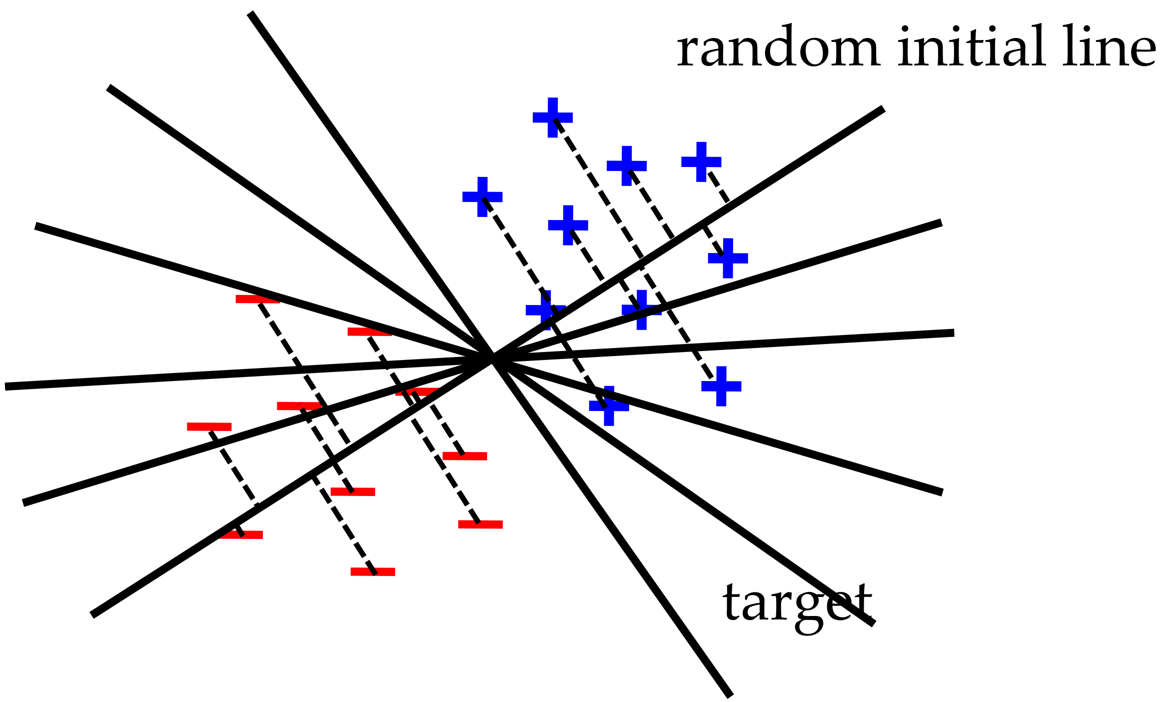


Fault Recovery Results



Example: Logistic Regression

Find best line separating two sets of points



Logistic Regression Code

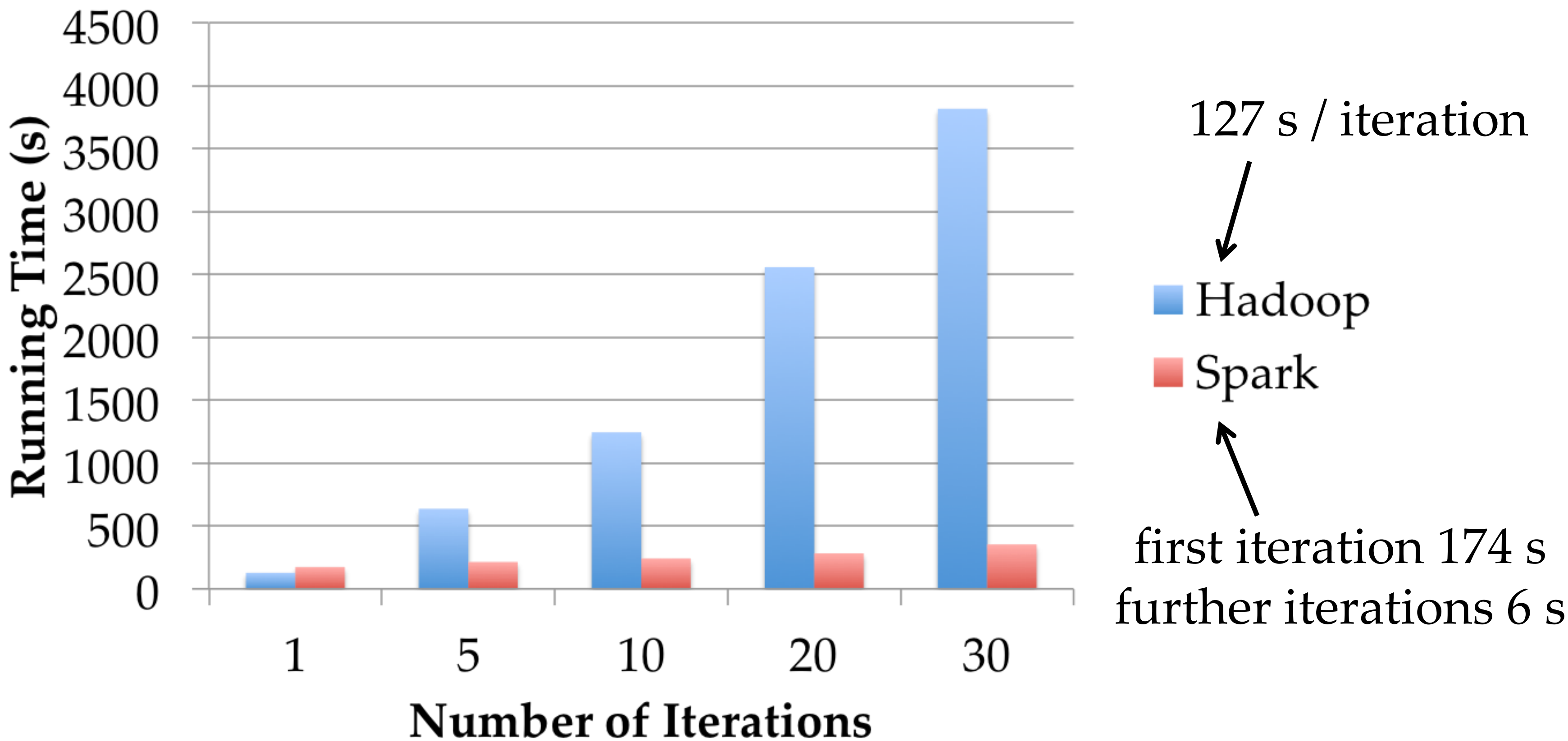
```
val data = spark.textFile(...).map(readPoint).cache()

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



Ongoing Projects

- Pregel on Spark (Bagel): graph processing programming model as a 200-line library

- Hive on Spark (Shark): SQL engine

- Spark Streaming: incremental processing with in-memory state

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/