CS 5450

Spark

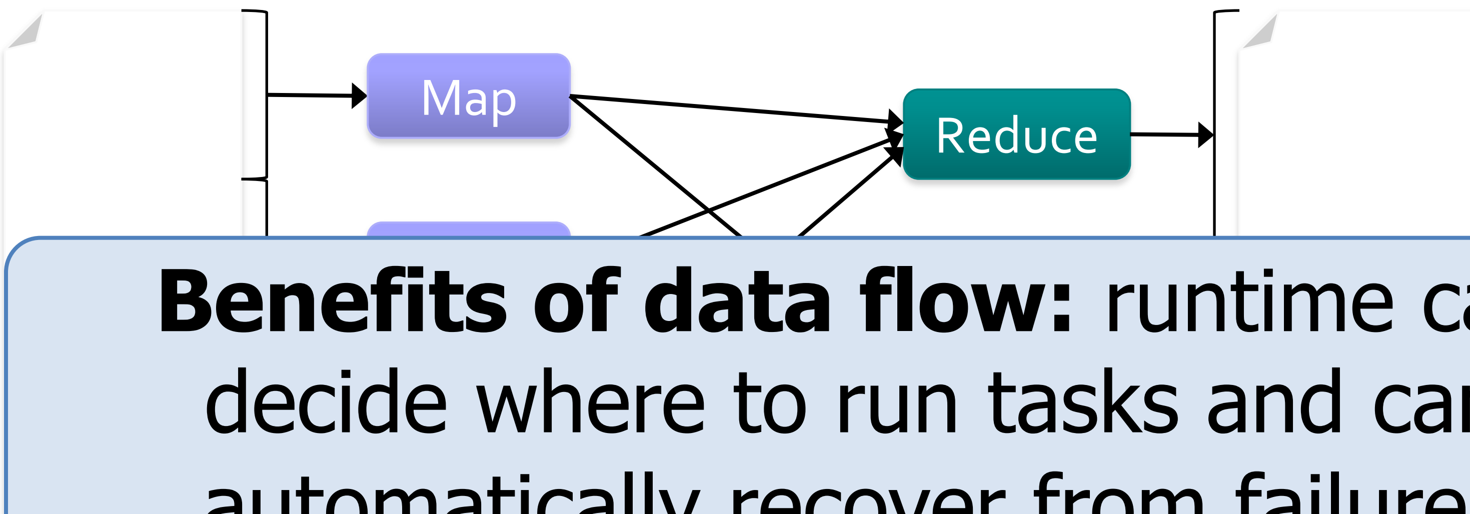
Slides from Matei Zaharia
and Databricks

Goals

- Extend the MapReduce model to better support two common classes of analytics apps
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter
  - Also support for Java, Python…

Cluster Programming Models

Most current cluster programming models are based on acyclic data flow from stable storage to stable storage



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Acyclic Data Flow Inefficient ...

... for applications that repeatedly reuse a working set of data

- Iterative algorithms (machine learning, graphs)
- Interactive data mining (R, Excel, Python)

because apps have to reload data from stable storage on every query

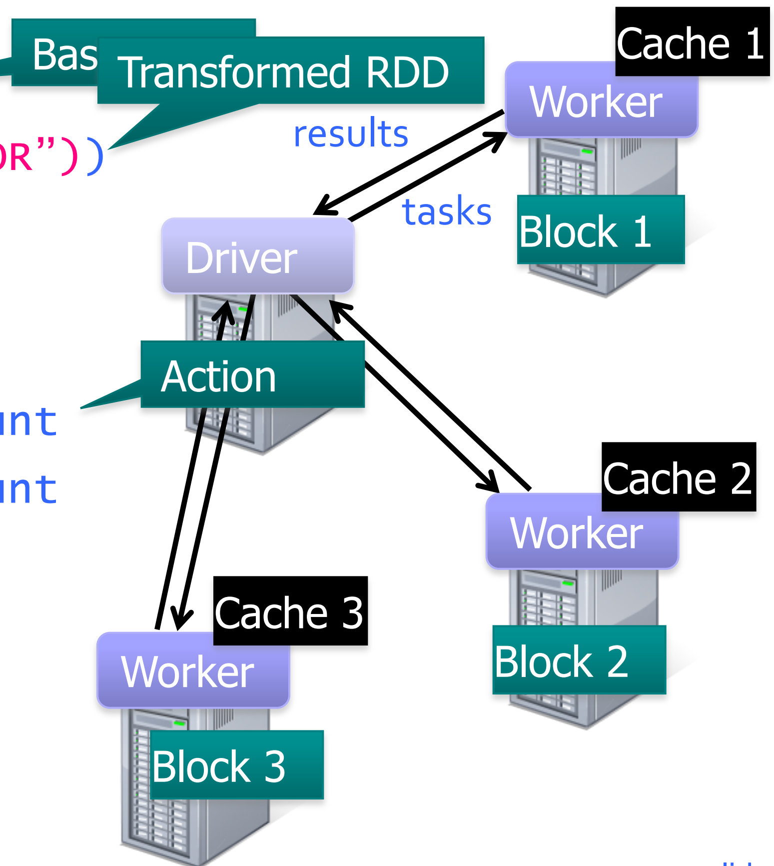
Resilient Distributed Datasets

- Resilient distributed datasets (RDDs)
  - Immutable, partitioned collections of objects spread across a cluster, stored in RAM or on disk
  - Created through parallel transformations (map, filter, groupBy, join, …) on data in stable storage
- Allow apps to cache working sets in memory for efficient reuse
- Retain the attractive properties of MapReduce
  - Fault tolerance, data locality, scalability
- Actions on RDDs support many applications
  - Count, reduce, collect, save…

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))
cachedMsgs = messages.cache()

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



Full-text search of Wikipedia

60GB on 20 EC2 machine

0.5 sec vs. 20s for on-disk

Spark Operations

Transformations define a new RDD	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions return a result to driver program	collect reduce count save lookupKey	

Creating RDDs

```
# Turn a Python collection into an RDD
>sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
>sc.textFile("file.txt")
>sc.textFile("directory/*.txt")
>sc.textFile("hdfs://namenode:9000/path/file")

# Use existing Hadoop InputFormat (Java/Scala only)
>sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

```
> nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
> squares = nums.map(lambda x: x*x)   // {1, 4, 9}

# Keep elements passing a predicate
> even = squares.filter(lambda x: x % 2 == 0) // {4}

# Map each element to zero or more others
> nums.flatMap(lambda x: => range(x))
   > # => {0, 0, 1, 0, 1, 2}
```

Range object
(sequence of numbers 0, 1, ..., x-1)

Basic Actions

```
> nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
> nums.collect() # => [1, 2, 3]

# Return first K elements
> nums.take(2)   # => [1, 2]

# Count number of elements
> nums.count()   # => 3

# Merge elements with an associative function
> nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
> nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's "distributed reduce" transformations operate on RDDs of key-value pairs

Python:

```python
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

Scala:

```scala
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

Java:

```java
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

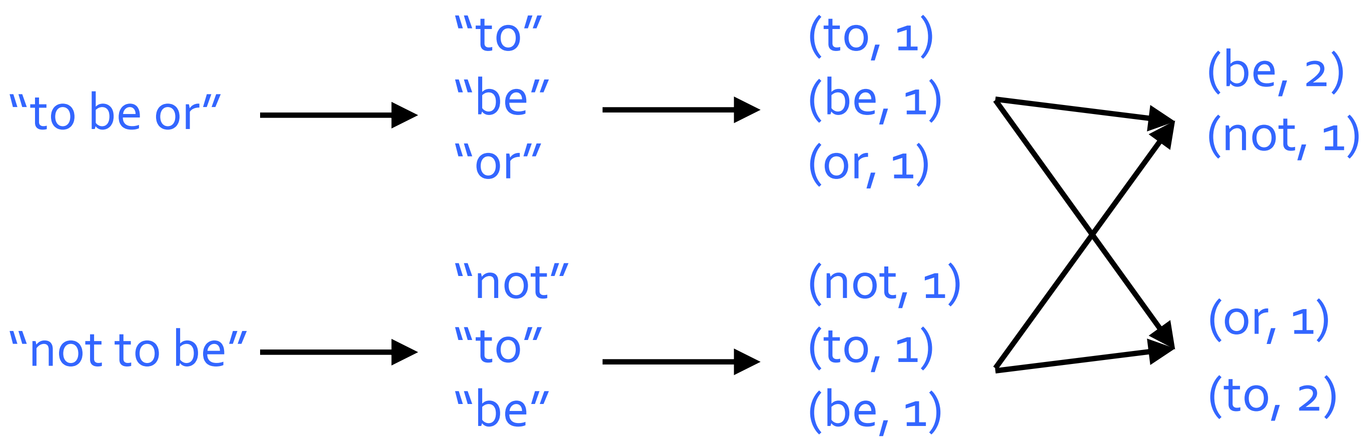
Some Key-Value Operations

```
> pets = sc.parallelize(
    [("cat", 1), ("dog", 1), ("cat", 2)])
> pets.reduceByKey(lambda x, y: x + y)
                        # => {(cat, 3), (dog, 1)}
> pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
> pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

`reduceByKey` also automatically implements combiners on the map side

Example: Word Count

```
> lines = sc.textFile(“hamlet.txt”)
> counts = lines.flatMap(lambda line: line.split(“ ”))
                .map(lambda word => (word, 1))
                .reduceByKey(lambda x, y: x + y)
```

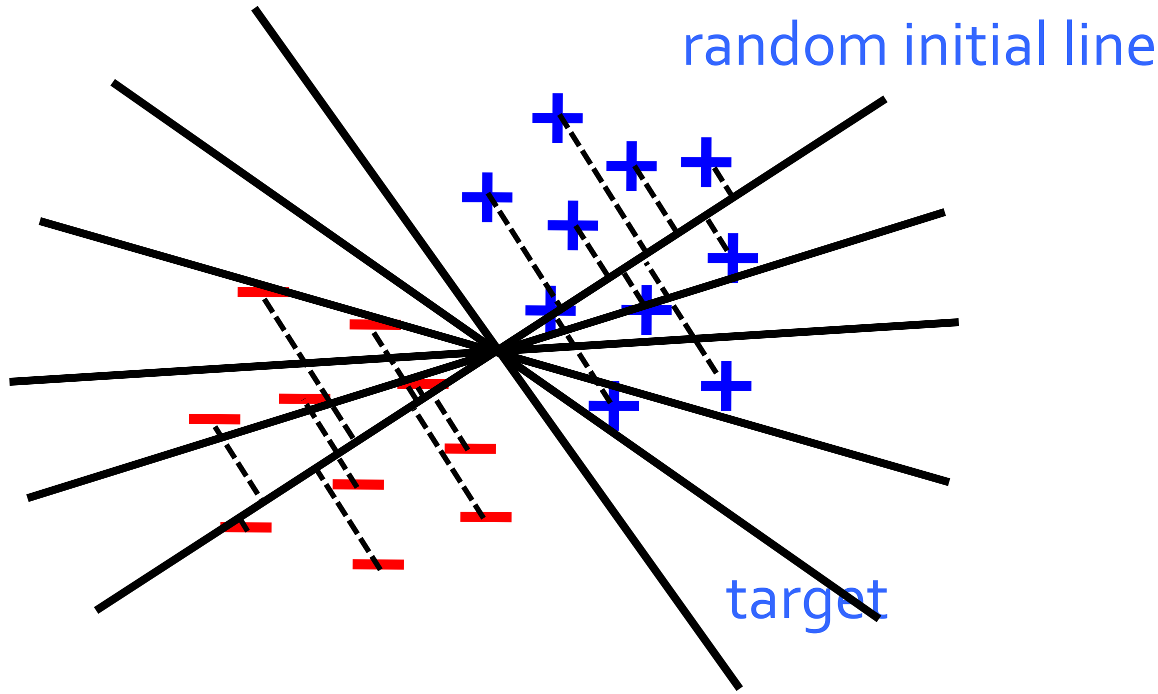


Other Key-Value Operations

```
> visits = sc.parallelize([ ("index.html", "1.2.3.4"),
                            ("about.html", "3.4.5.6"),
                            ("index.html", "1.3.3.1") ])

> pageNames = sc.parallelize([ ("index.html", "Home"),
                               ("about.html", "About") ])

> visits.join(pageNames)
  # ("index.html", ("1.2.3.4", "Home"))
  # ("index.html", ("1.3.3.1", "Home"))
  # ("about.html", ("3.4.5.6", "About"))

> visits.cogroup(pageNames)
  # ("index.html", (["1.2.3.4", "1.3.3.1"], ["Home"]))
  # ("about.html", (["3.4.5.6"], ["About"]))
```

Example: Logistic Regression

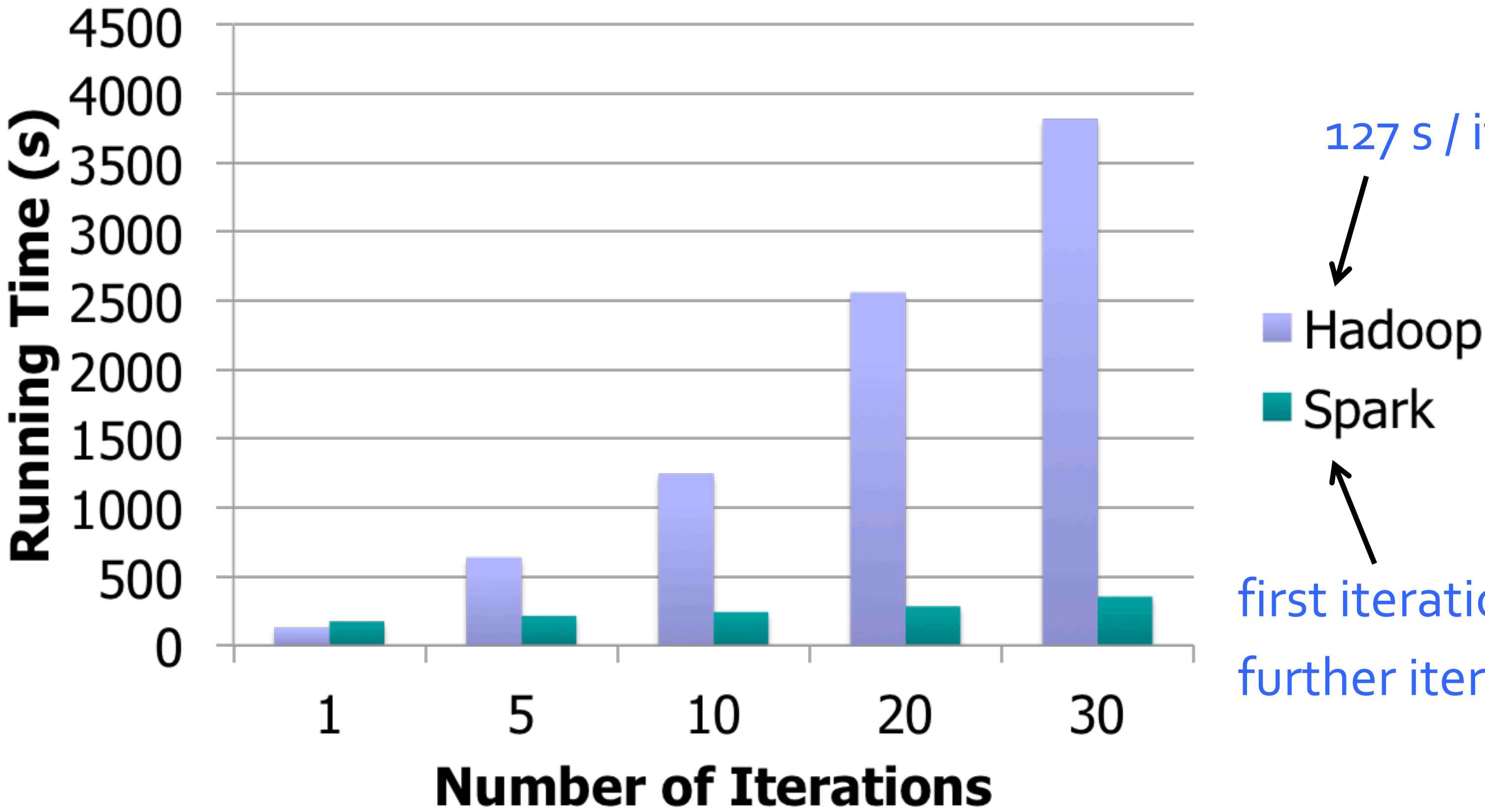
Goal: find best line separating two sets of points



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)
```

Logistic Regression Performance



127 s / iteration

first iteration 174 s

further iterations 6 s

Setting the Level of Parallelism

All pair RDD operations take an optional second parameter for the number of tasks

```
> words.reduceByKey(lambda x, y: x + y, 5)
> words.groupByKey(5)
> visits.join(pageViews, 5)
```

Using Local Variables

Any external variables used in a closure are automatically shipped to the cluster

```
> query = sys.stdin.readline()
> pages.filter(lambda x: query in x).count()
```

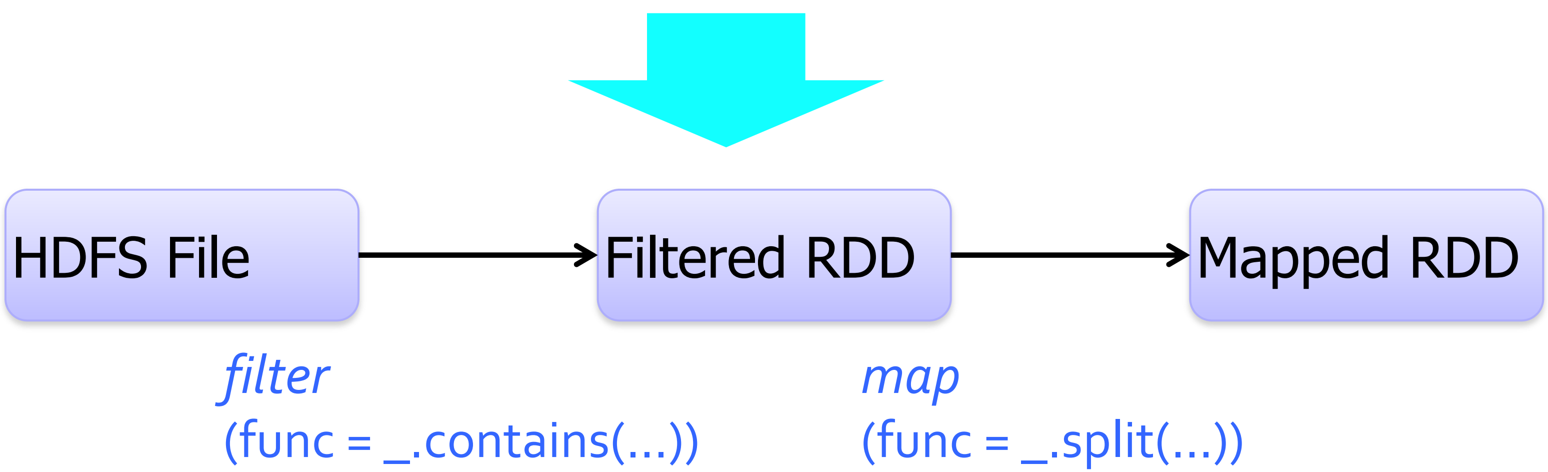
Some caveats:

- Each task gets a new copy (updates aren't sent back)
- Variable must be serializable / pickle-able
- Don't use fields of an outer object (ships all of it!)

RDD Fault Tolerance

RDDs maintain lineage information that can be used to reconstruct lost partitions

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

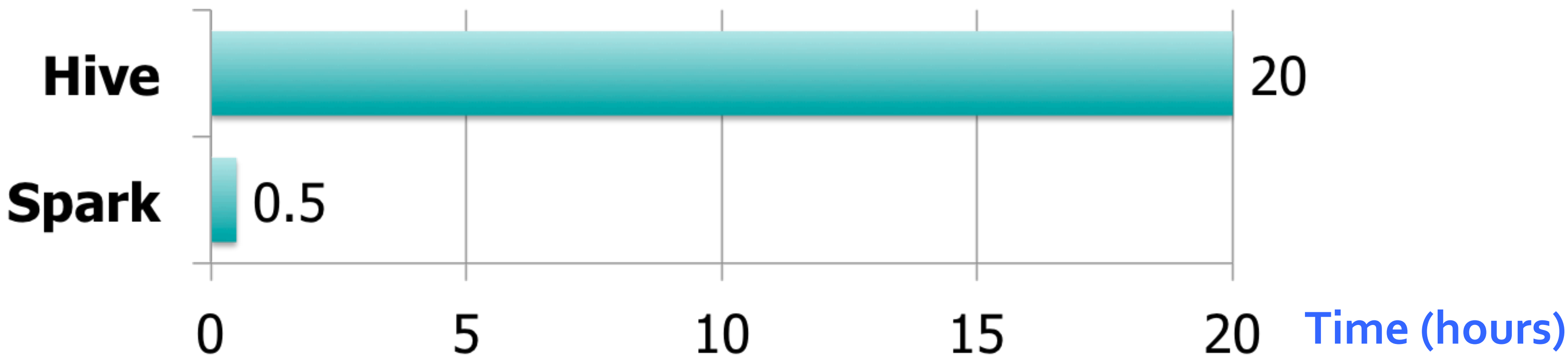


Spark Applications

- In-memory data mining on Hive data (Conviva)
- Predictive analytics (Quantifind)
- City traffic prediction (Mobile Millennium)
- Twitter spam classification (Monarch)

... many others

Conviva GeoReport



- Aggregations on many keys w/ same WHERE clause
- 40× gain comes from:
  - Not re-reading unused columns or filtered records
  - Avoiding repeated decompression
  - In-memory storage of de-serialized objects

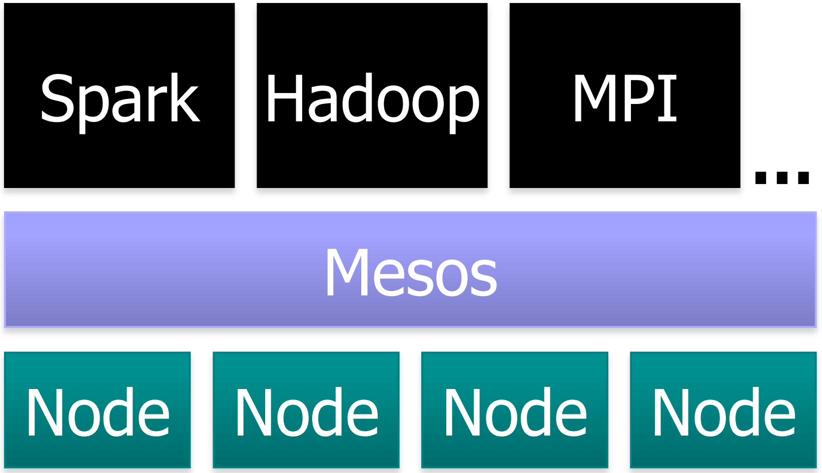
Frameworks Built on Spark

- Pregel on Spark (Bagel)
  - Google message passing model for graph computation
  - 200 lines of code
- Hive on Spark (Shark)
  - 3000 lines of code
  - Compatible with Apache Hive
  - ML operators in Scala

Implementation

Runs on Apache Mesos to share resources with Hadoop & other apps

Can read from any Hadoop input source (e.g. HDFS)



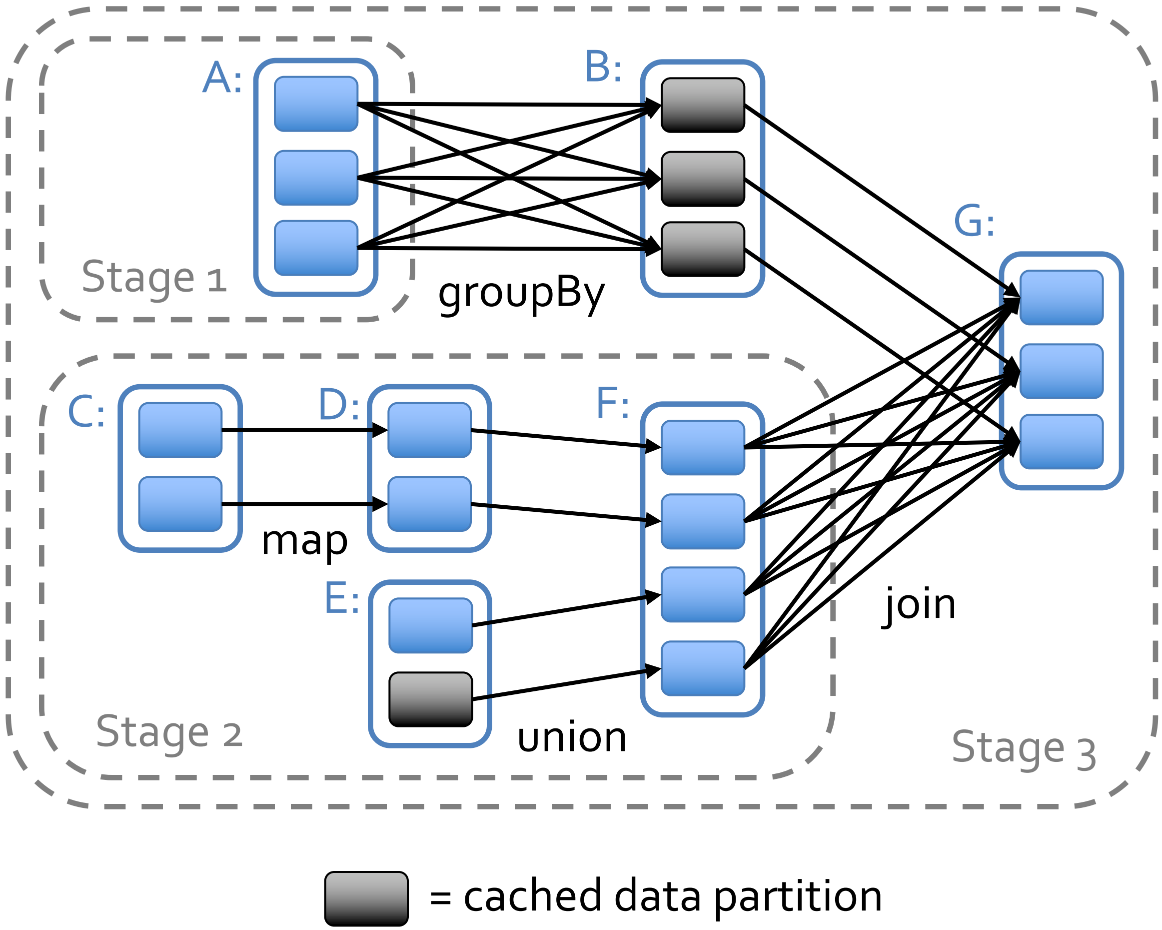
Spark Scheduler

Dryad-like DAGs

Pipelines functions within a stage

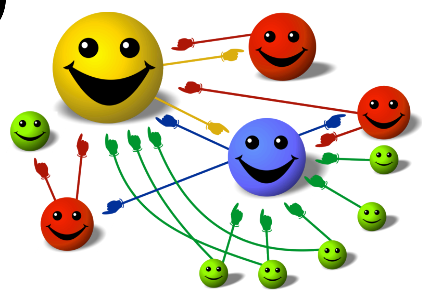
Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles



Example: PageRank

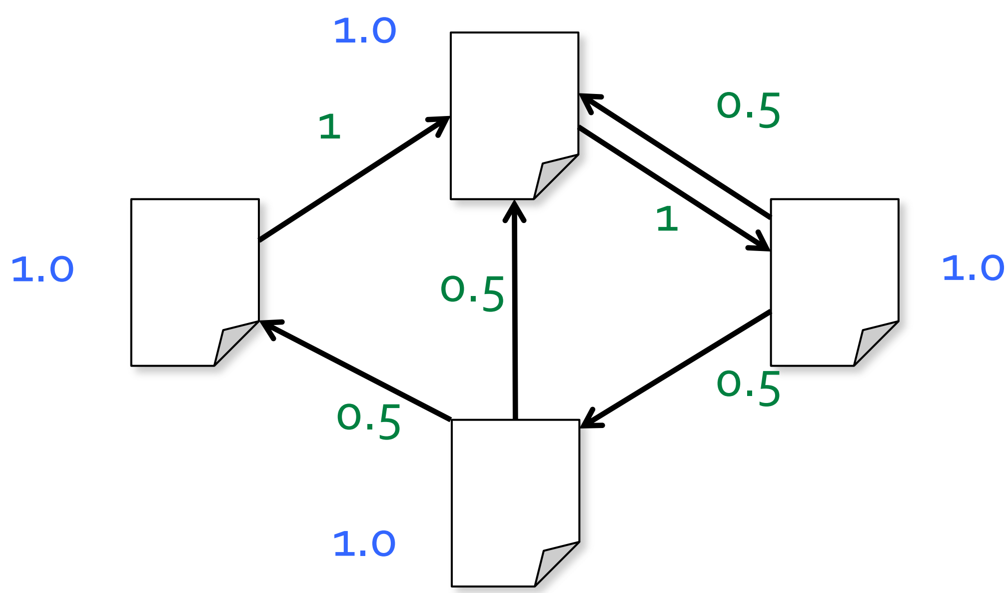
- Basic idea: gives pages ranks (scores) based on links to them
  - Links from many pages ➔ high rank
  - Link from a high-rank page ➔ high rank



- Good example of a more complex algorithm
  - Multiple stages of map & reduce
- Benefits from Spark's in-memory caching
  - Multiple iterations over the same data

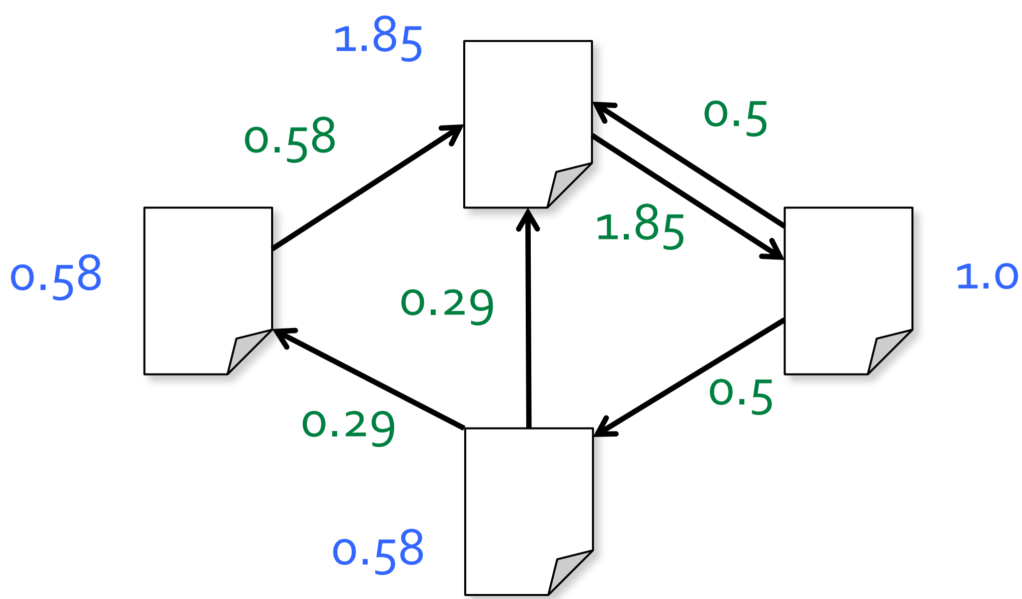
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs



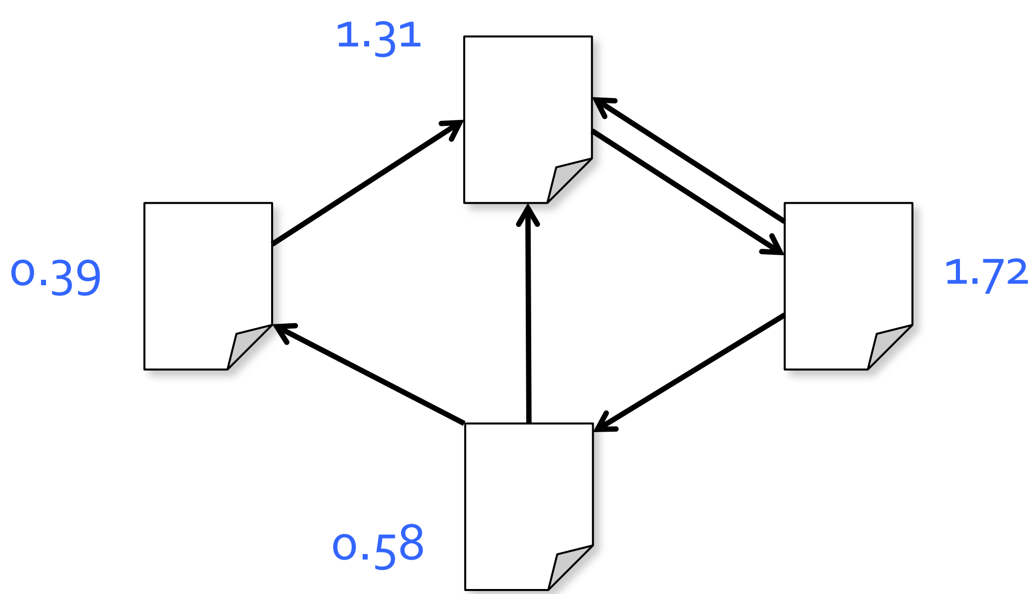
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs



Algorithm

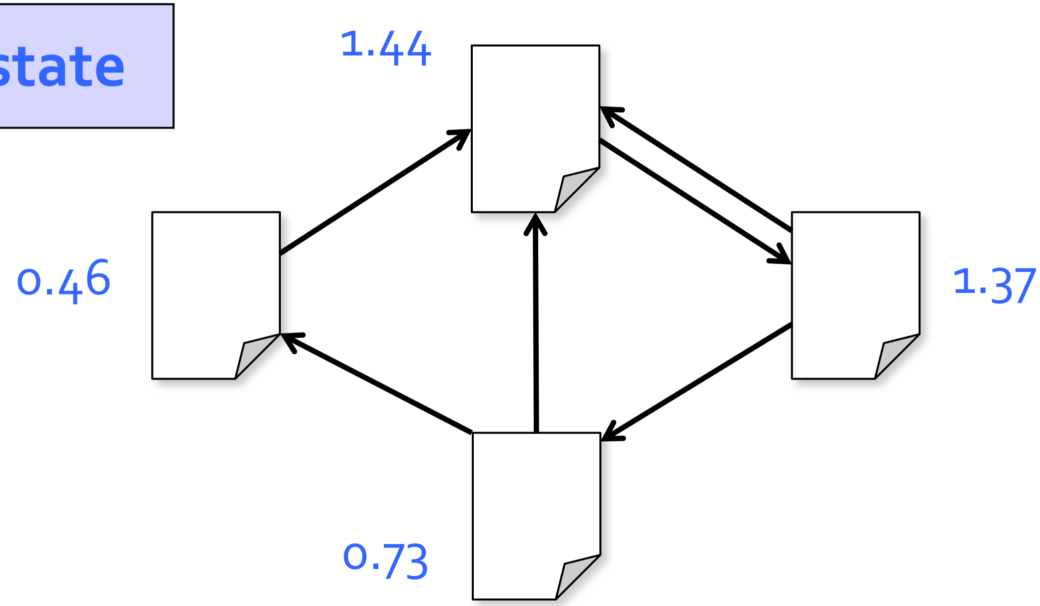
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs



Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs

Final state



Spark Implementation (in Scala)

```scala
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
         links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
                  .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

PageRank Performance