

Πανεπιστήμιο Πατρών


Προχωρημένα Θέματα σε
Κατανεμημένα Συστήματα

Spark


What is Spark?

- Fast, expressive cluster computing system compatible with Apache Hadoop
 - Works with any Hadoop-supported storage system (HDFS, S3, Avro, ...)

- Improves **efficiency** through:
 - In-memory computing primitives
 - General computation graphs

 Up to 100× faster

- Improves **usability** through:
 - Rich APIs in Java, Scala, Python
 - Interactive shell

 Often 2-10× less code

How to Run It

- ❑ Local multicore: just a library in your program
- ❑ EC2: scripts for launching a Spark cluster
- ❑ Private cluster: Mesos, YARN, Standalone Mode

Languages

- APIs in Java, Scala, Python, R
- Interactive shells in Scala and Python

Introduction to Spark

Key Idea

- **Work with distributed collections as you would with local ones**

- **Concept: Resilient Distributed Datasets (RDDs)**
 - Immutable collections of objects spread across a cluster
 - Built through parallel transformations (map, filter, etc)
 - Automatically rebuilt on failure
 - Controllable persistence (e.g. caching in RAM)

Operations

- Transformations (e.g. map, filter, groupBy, join)
 - Lazy operations to build RDDs from other RDDs

- Actions (e.g., count, collect, save)
 - Return a result or write it to storage

Example: Mining Console Logs

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

```
lines = sc.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('\t')[2])
messages.cache()

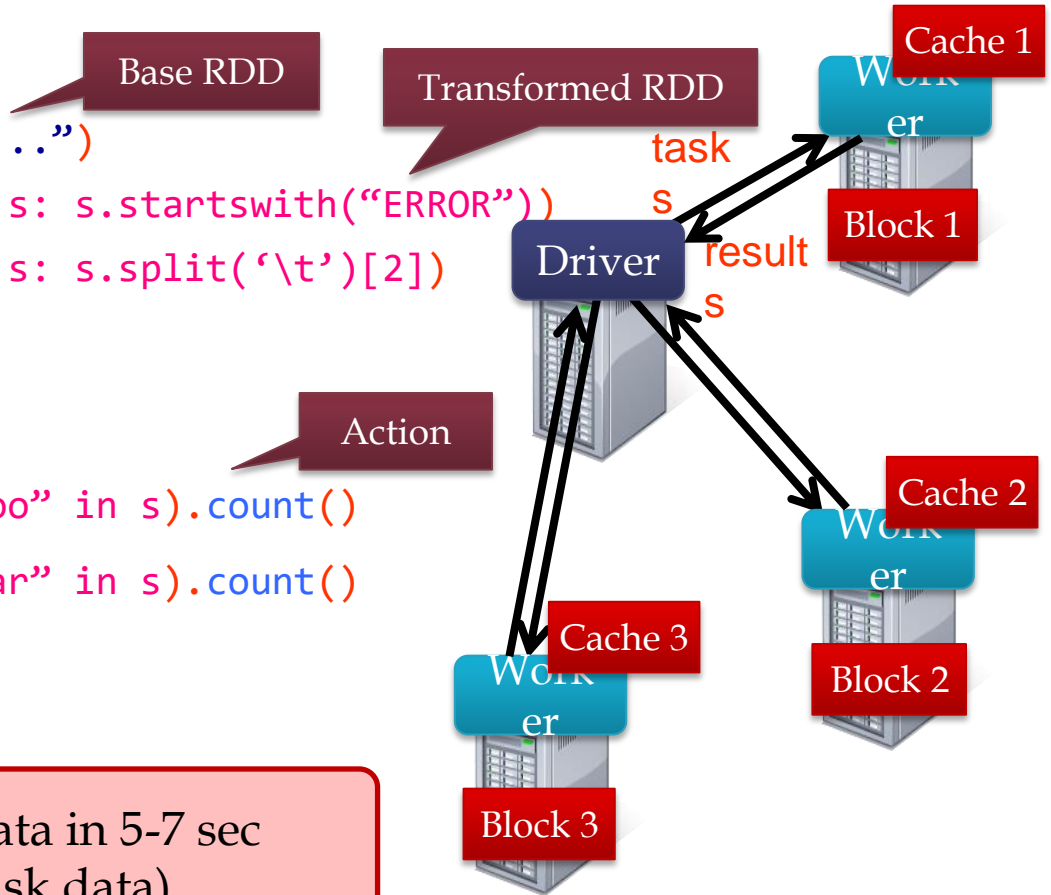
messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
...

```

```
messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
...

```

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

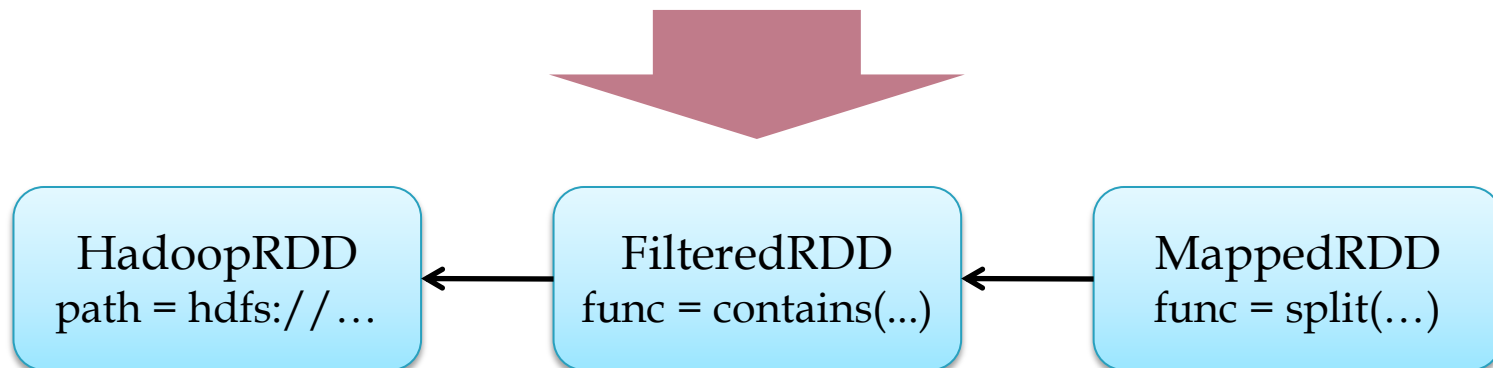


RDD Fault Tolerance

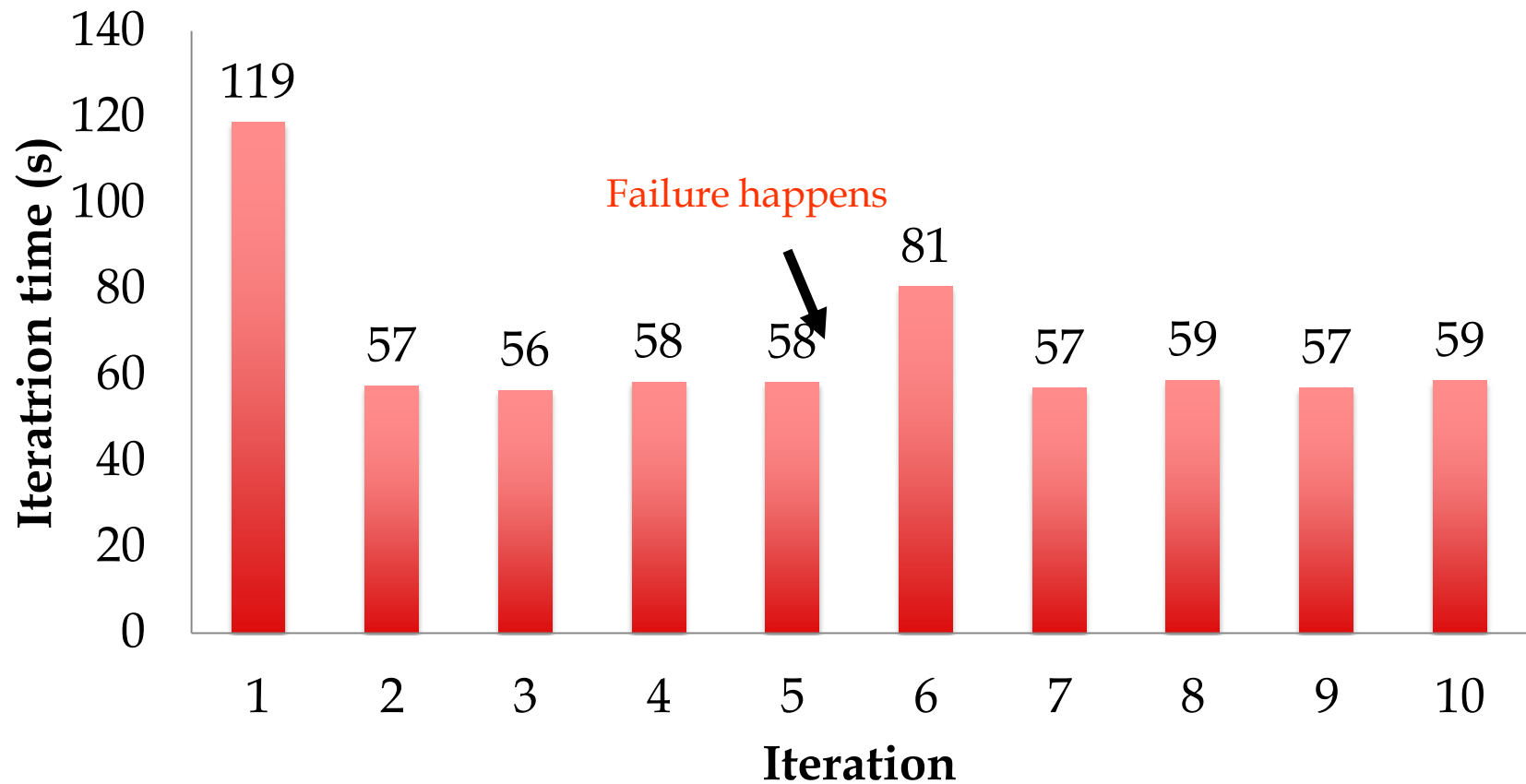
- RDDs track the transformations used to build them (their lineage) to recompute lost data

- E.g:

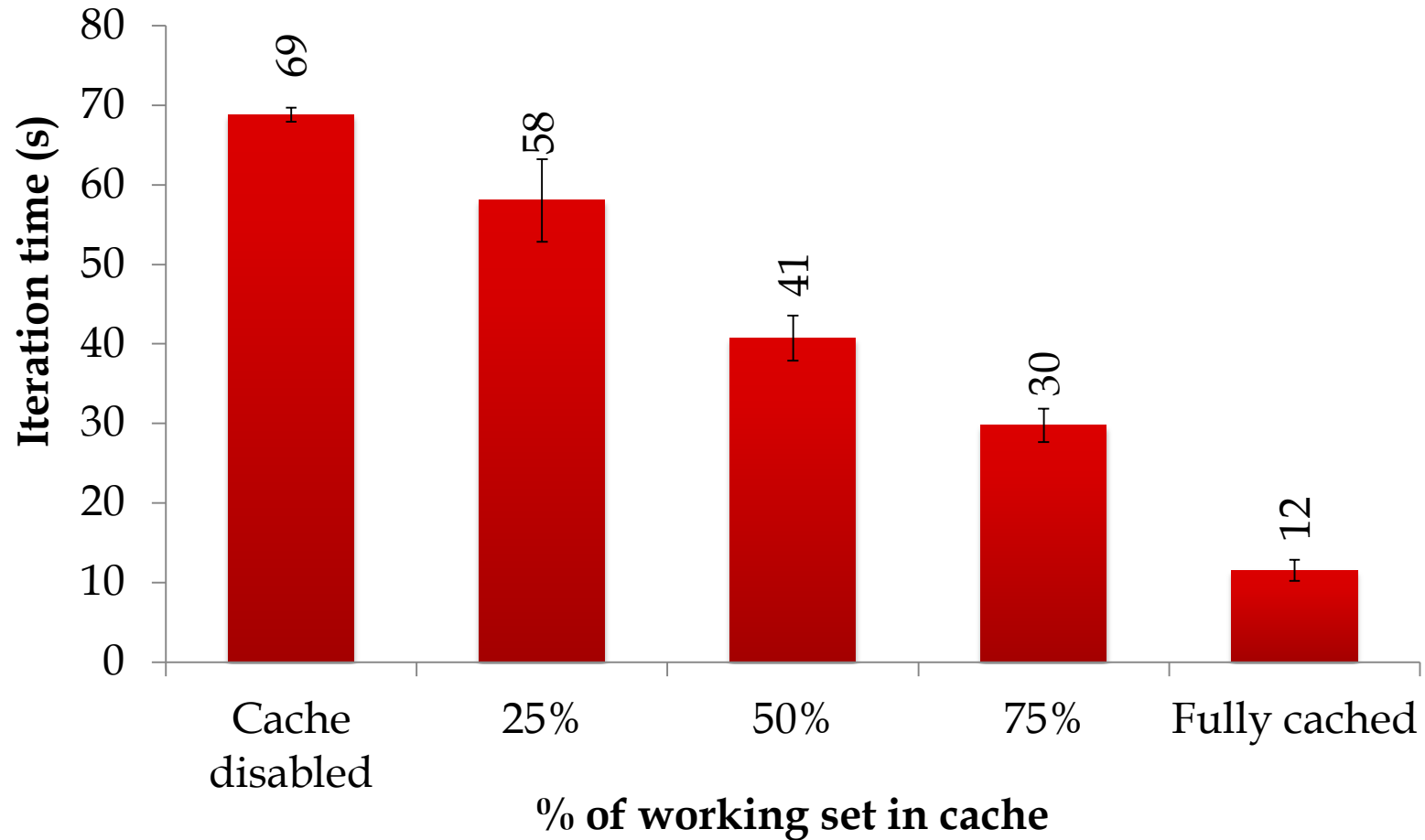
```
messages = textFile(...).filter(lambda s: s.contains("ERROR"))  
                        .map(lambda s: s.split('\t')[2])
```



Fault Recovery Test



Behavior with Less RAM



Spark in Java and Scala

Java API:

```
JavaRDD<String> lines = spark.textFile(...);

errors = lines.filter(
    new Function<String, Boolean>() {
        public Boolean call(String s) {
            return s.contains("ERROR");
        }
    });

errors.count()
```

Scala API:

```
val lines = spark.textFile(...)

errors = lines.filter(s =>
    s.contains("ERROR"))
// can also write
filter(_.contains("ERROR"))

errors.count
```

Which Language Should I Use?

- ❑ Standalone programs can be written in any, but console is only Python & Scala
- ❑ **Python developers:** can stay with Python for both
- ❑ **Java developers:** consider using Scala for console (to learn the API)

- ❑ Performance: Java / Scala will be faster (statically typed), but Python can do well for numerical work with NumPy

Spark Operations

Learning Spark

- Easiest way: Spark interpreter (spark-shell or pyspark)
 - Special Scala and Python consoles for cluster use

- Runs in local mode on 1 thread by default, but can control with MASTER environment var:

```
MASTER=local      ./spark-shell      # local, 1 thread
MASTER=local[2]   ./spark-shell      # local, 2 threads
MASTER=spark://host:port ./spark-shell # Spark standalone cluster
```

First Stop: SparkContext

- ❑ Main entry point to Spark functionality
- ❑ Created for you in Spark shells as variable `sc`
- ❑ In standalone programs, you'd make your own (see later for details)

Creating RDDs

```
# Turn a local 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 any existing Hadoop InputFormat
```

```
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

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

```
# Pass each element through a function
```

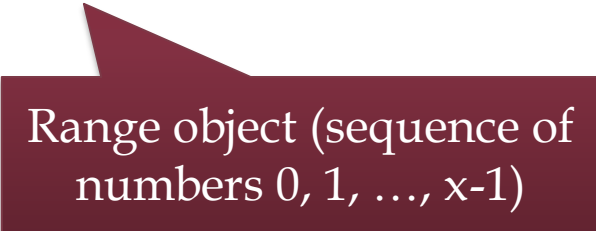
```
squares = nums.map(lambda x: x*x) # => {1, 4, 9}
```

```
# 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(0, 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 act on RDDs of *key-value pairs*
- ❑ Python:

```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```
- ❑ Scala:

```
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```
- ❑ Java:

```
Tuple2 pair = new Tuple2(a, b); // class scala.Tuple2
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)}
```

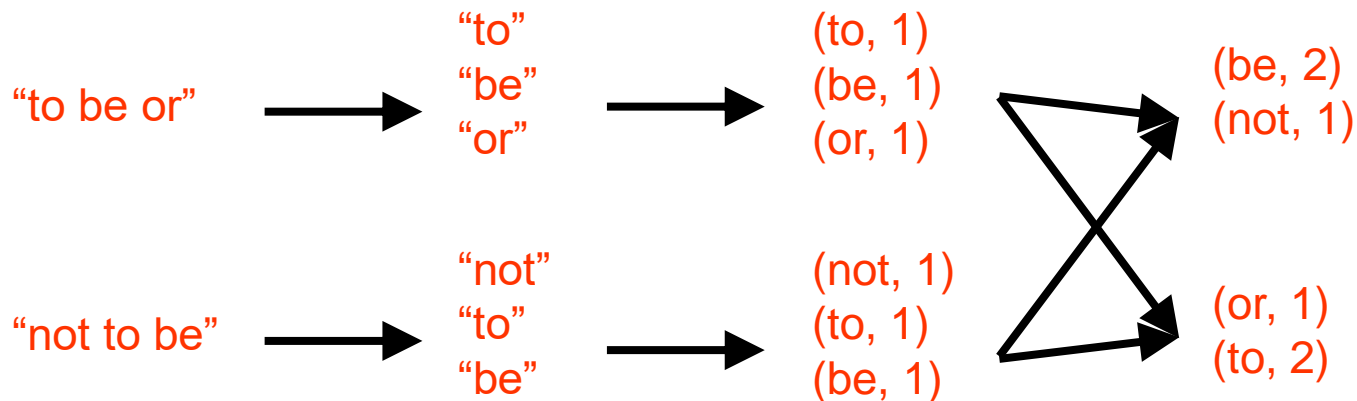
```
pets.groupByKey()  
# => {(cat, Seq(1, 2)), (dog, Seq(1))}
```

```
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)
```



Multiple Datasets

```
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”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))  
# (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Controlling the Level of Parallelism

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

```
words.reduceByKey(lambda x, y: x + y, 5)
```

```
words.groupByKey(5)
```

```
visits.join(pageViews, 5)
```


Using Local Variables

- External variables you use in a closure will automatically be shipped to the cluster:

```
query = raw_input("Enter a query:")  
pages.filter(lambda x: x.startswith(query)).count()
```

- Some caveats:

- Each task gets a new copy (updates aren't sent back)
- Variable must be Serializable (Java/Scala) or Pickle-able (Python)
- Don't use fields of an outer object (ships all of it!)

Closure Mishap Example

```
class MyCoolRddApp {  
  val param = 3.14  
  val log = new Log(...)  
  ...  
  
  def work(rdd: RDD[Int]) {  
    rdd.map(x => x + param)  
        .reduce(...)  
  }  
}
```

NotSerializableException:
MyCoolRddApp (or Log)

How to get around it:

```
class MyCoolRddApp {  
  ...  
  
  def work(rdd: RDD[Int]) {  
    val param_ = param  
    rdd.map(x => x + param_)  
        .reduce(...)  
  }  
}
```

References only local variable
instead of this.param

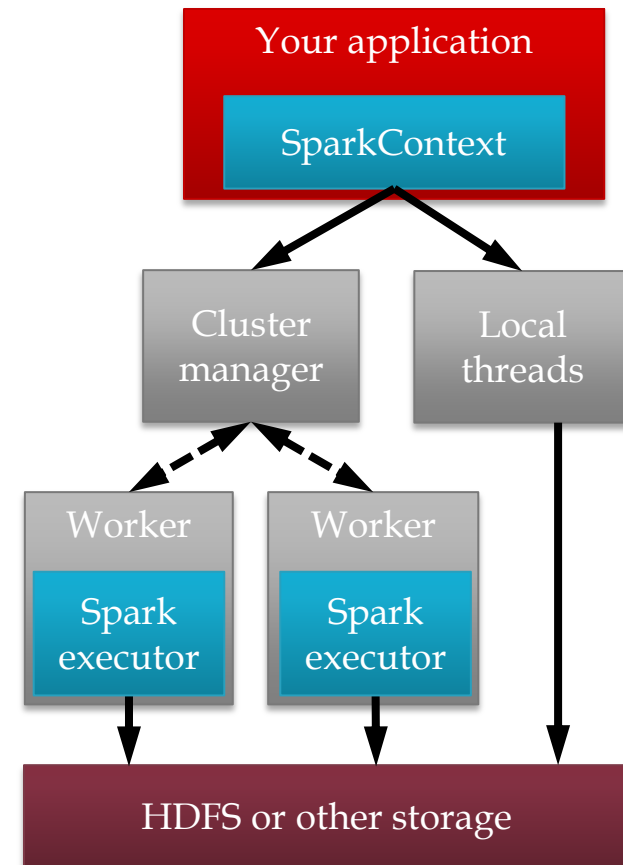
More Details

- ❑ Spark supports lots of other operations!
- ❑ Full programming guide: spark-project.org/documentation

Job Execution

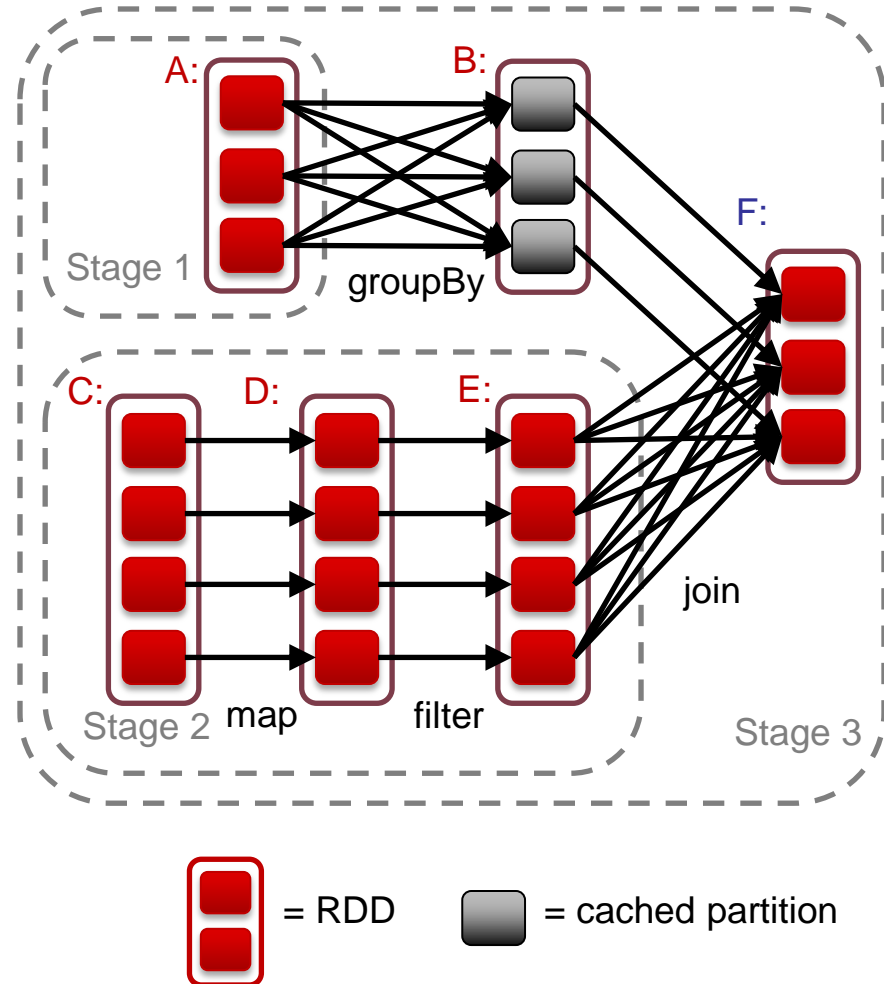
Software Components

- ❑ Spark runs as a library in your program (one instance per app)
- ❑ Runs tasks **locally** or on a **cluster**
 - Standalone deploy cluster, Mesos or YARN
- ❑ Accesses storage via Hadoop InputFormat API
 - Can use HBase, HDFS, S3, ...



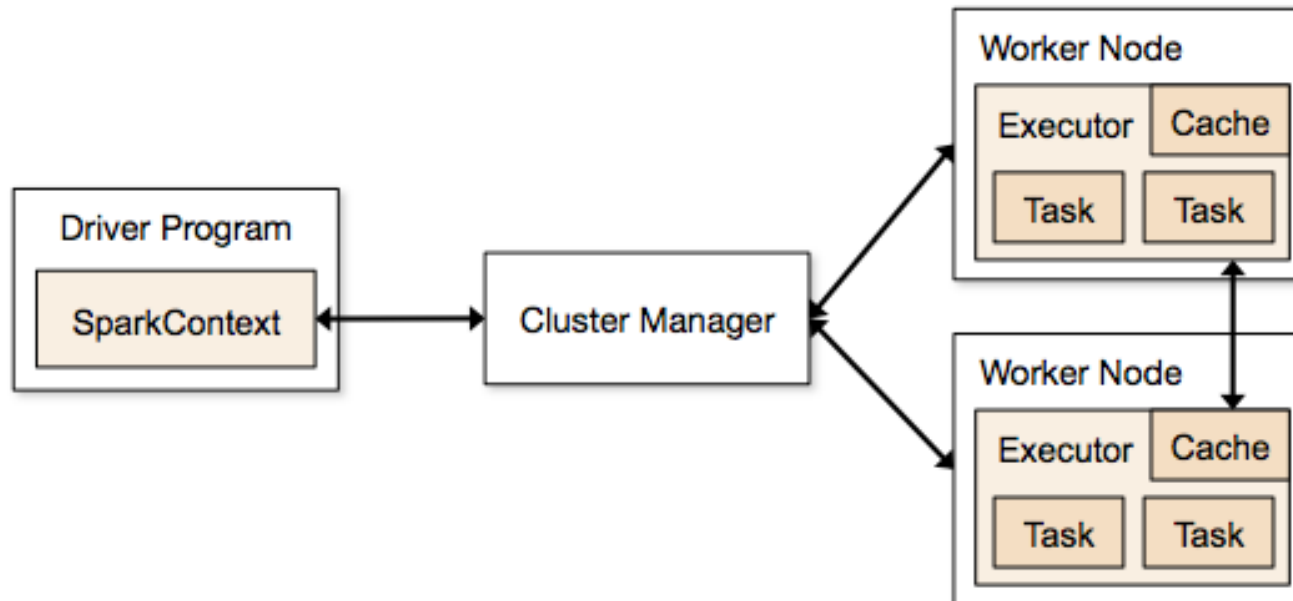
Task Scheduler

- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles



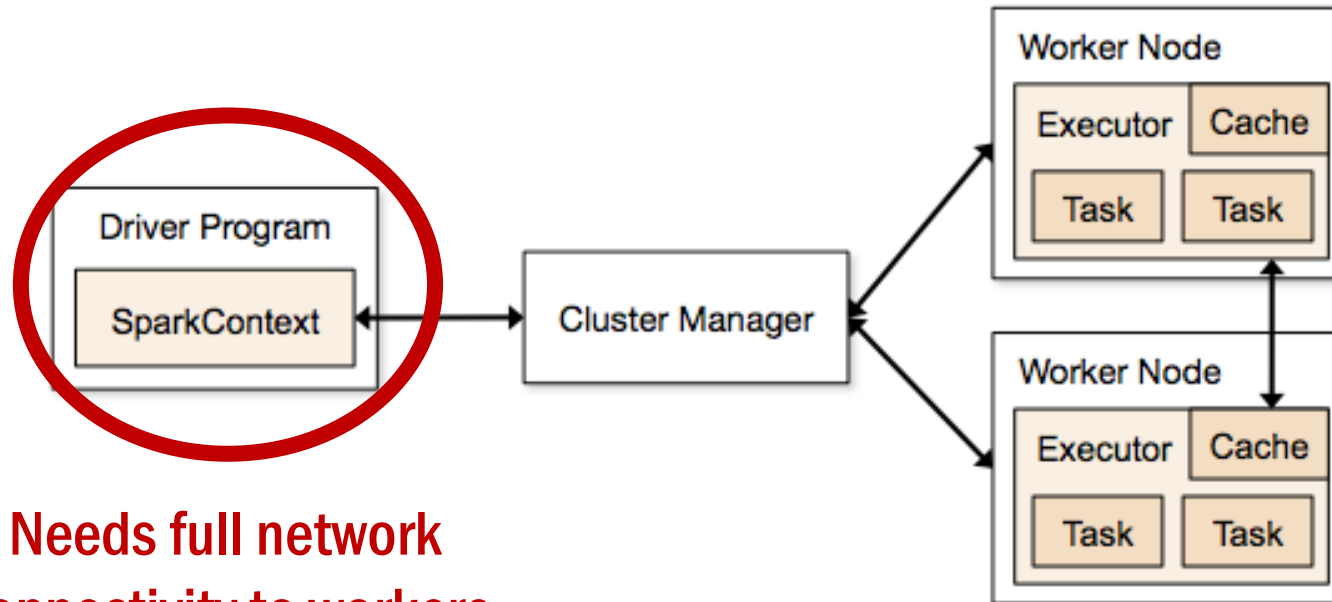
Cluster manager

- Cluster manager grants executors to a Spark application



Driver program

- Driver program decides when to launch tasks on which executor



**Needs full network
connectivity to workers**

Hadoop Compatibility

- Spark can read/write to any storage system / format that has a plugin for Hadoop!
 - Examples: HDFS, S3, HBase, Cassandra, Avro, SequenceFile
 - Reuses Hadoop's InputFormat and OutputFormat APIs

- APIs like `SparkContext.textFile` support filesystems, while `SparkContext.hadoopRDD` allows passing any Hadoop JobConf to configure an input source

Standalone Programs

Build Spark

- Requires Java 6+, Scala 2.9.2

```
git clone git://github.com/mesos/spark
cd spark
sbt/sbt package
```

```
# Optional: publish to local Maven cache
sbt/sbt publish-local
```

Add Spark to Your Project

- ❑ Scala and Java: add a Maven dependency on

```
groupId:    org.spark-project
artifactId: spark-core_2.9.1
version:    0.7.0-SNAPSHOT
```

- ❑ Python: run program with our pyspark script

Create a SparkContext

Scala

```
import spark.SparkContext
import spark.SparkContext._

val sc = new SparkContext("masterUrl", "name", "sparkHome", Seq("app.jar"))
```

Cluster URL, or
local / local[N]

App
name

Spark install
path on
cluster

List of JARs
with app code
(to ship)

Java

```
import spark.api.java

JavaSparkContext sc = new JavaSparkContext(
    "masterUrl", "name", "sparkHome", new String[] {"app.jar"});
```

Python

```
from pyspark import SparkContext

sc = SparkContext("masterUrl", "name", "sparkHome", ["library.py"])
```

Complete App: Scala

```
import spark.SparkContext
import spark.SparkContext._

object WordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "WordCount", args(0),
Seq(args(1)))
    val lines = sc.textFile(args(2))
    lines.flatMap(_.split(" "))
      .map(word => (word, 1))
      .reduceByKey(_ + _)
      .saveAsTextFile(args(3))
  }
}
```

Complete App: Python

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
    sc = SparkContext("local", "WordCount", sys.argv[0], None)
    lines = sc.textFile(sys.argv[1])

    lines.flatMap(lambda s: s.split(" ")) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda x, y: x + y) \
        .saveAsTextFile(sys.argv[2])
```

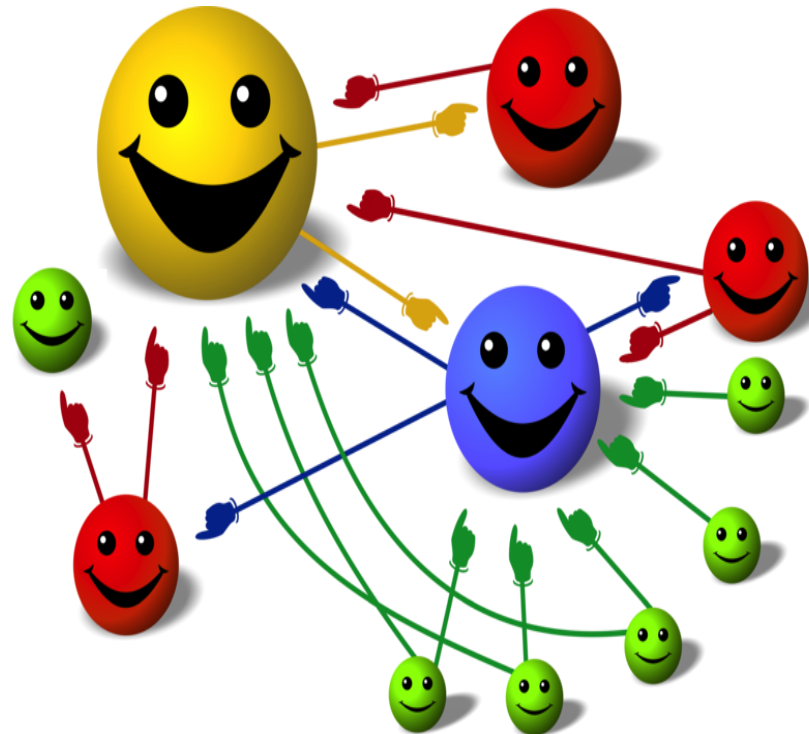
Example: PageRank

Why PageRank?

- 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

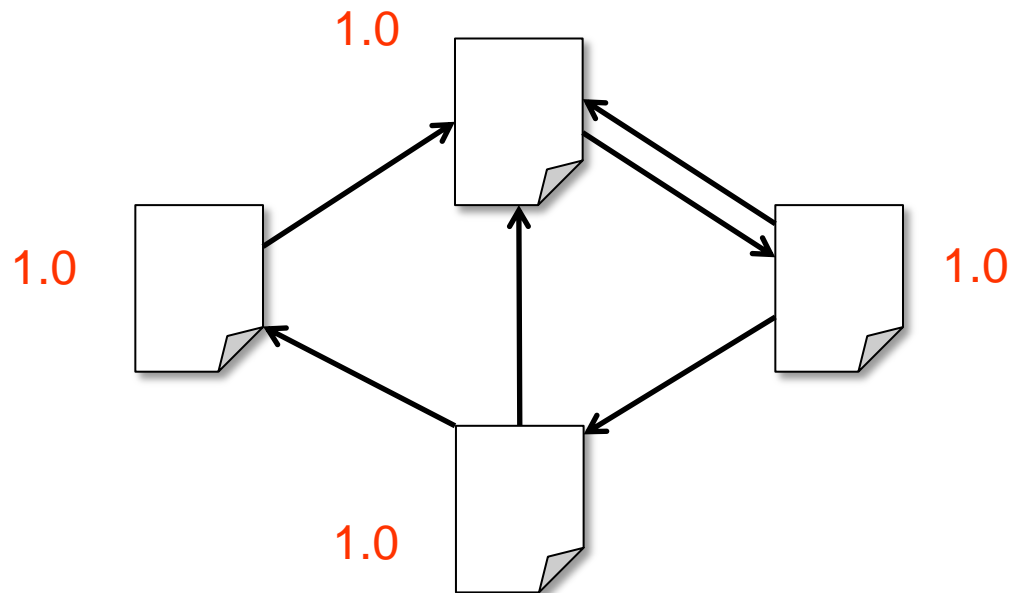
Basic Idea

- Give pages ranks (scores) based on links to them
 - Links from many pages → high rank
 - Link from a high-rank page → high rank



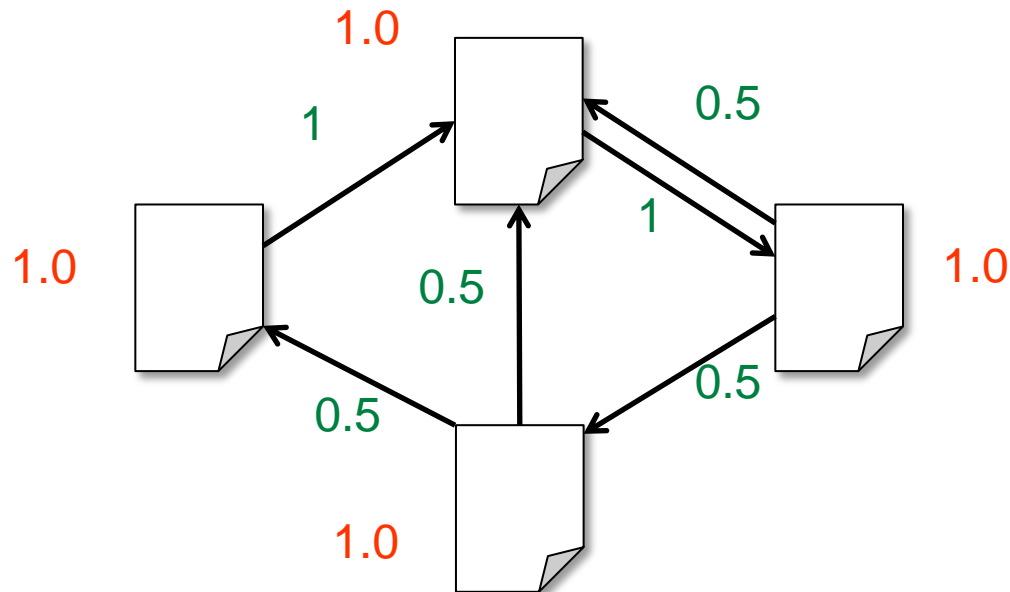
Algorithm

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



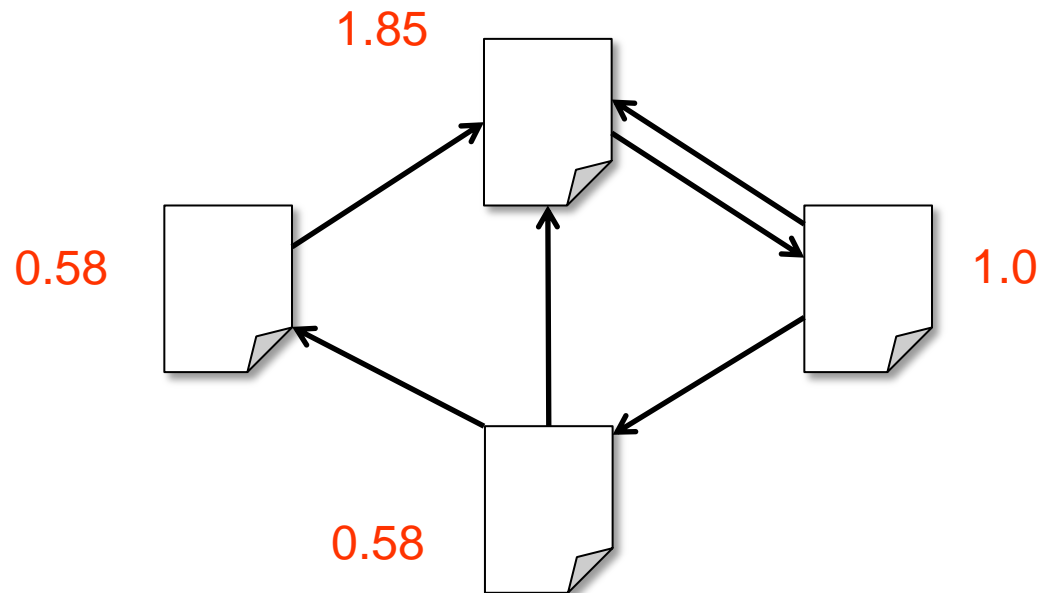
Algorithm

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



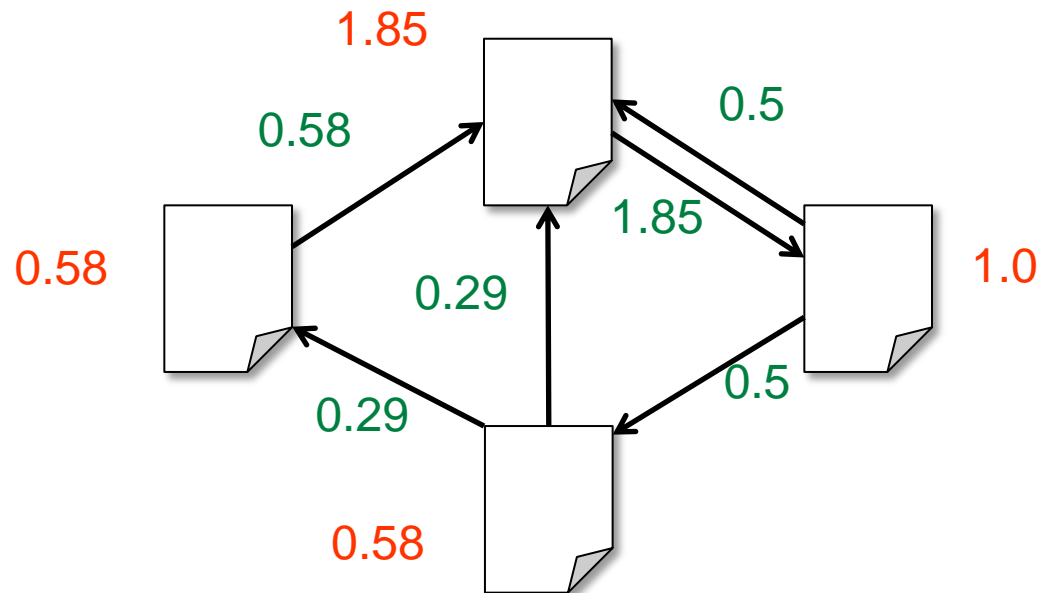
Algorithm

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



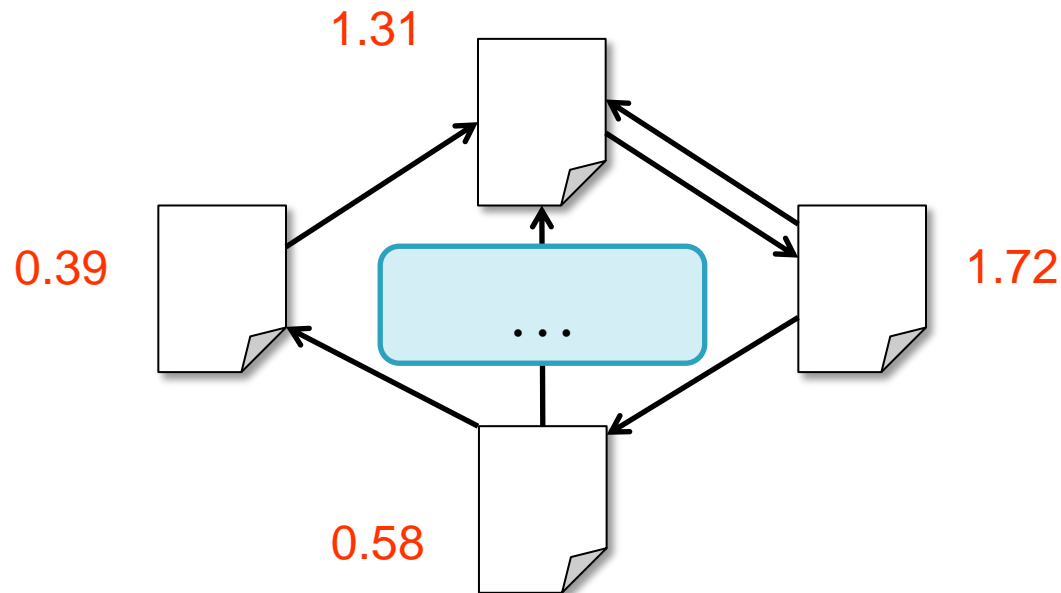
Algorithm

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



Algorithm

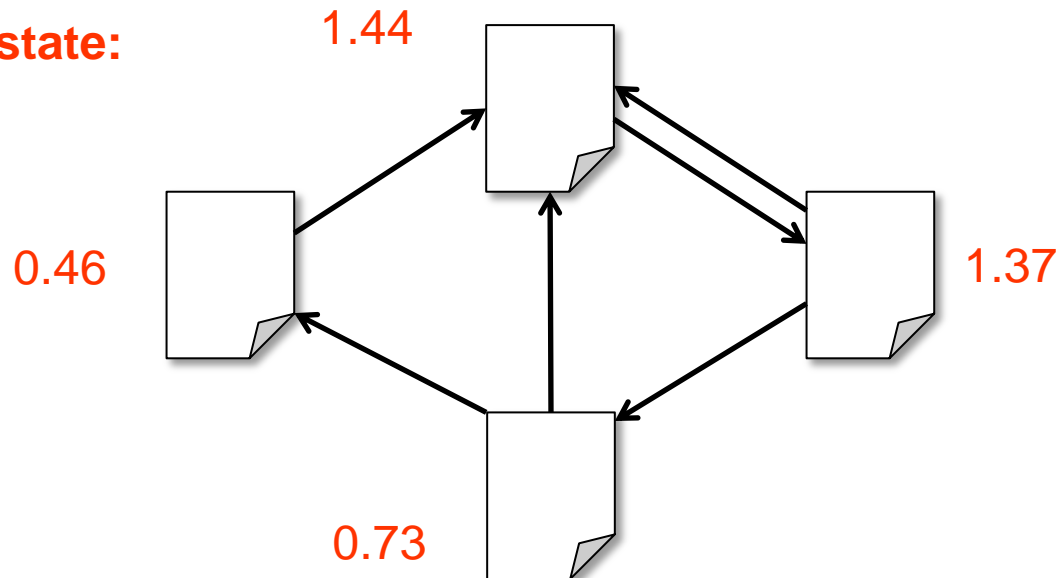
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

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

Final state:



Scala Implementation

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

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

ranks.saveAsTextFile(...)
```

Python Implementation

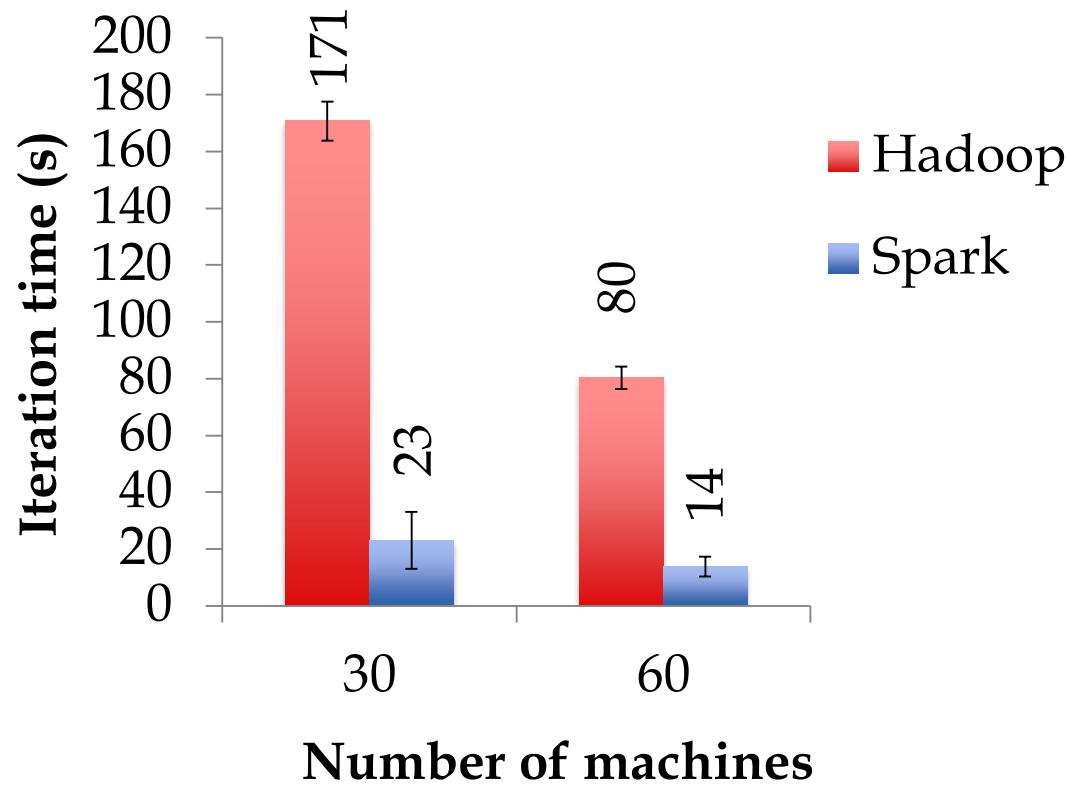
```
links = # RDD of (url, neighbors) pairs
ranks = # RDD of (url, rank) pairs

for i in range(NUM_ITERATIONS):
    def compute_contribs(pair):
        [url, [links, rank]] = pair # split key-value pair
        return [(dest, rank/len(links)) for dest in links]

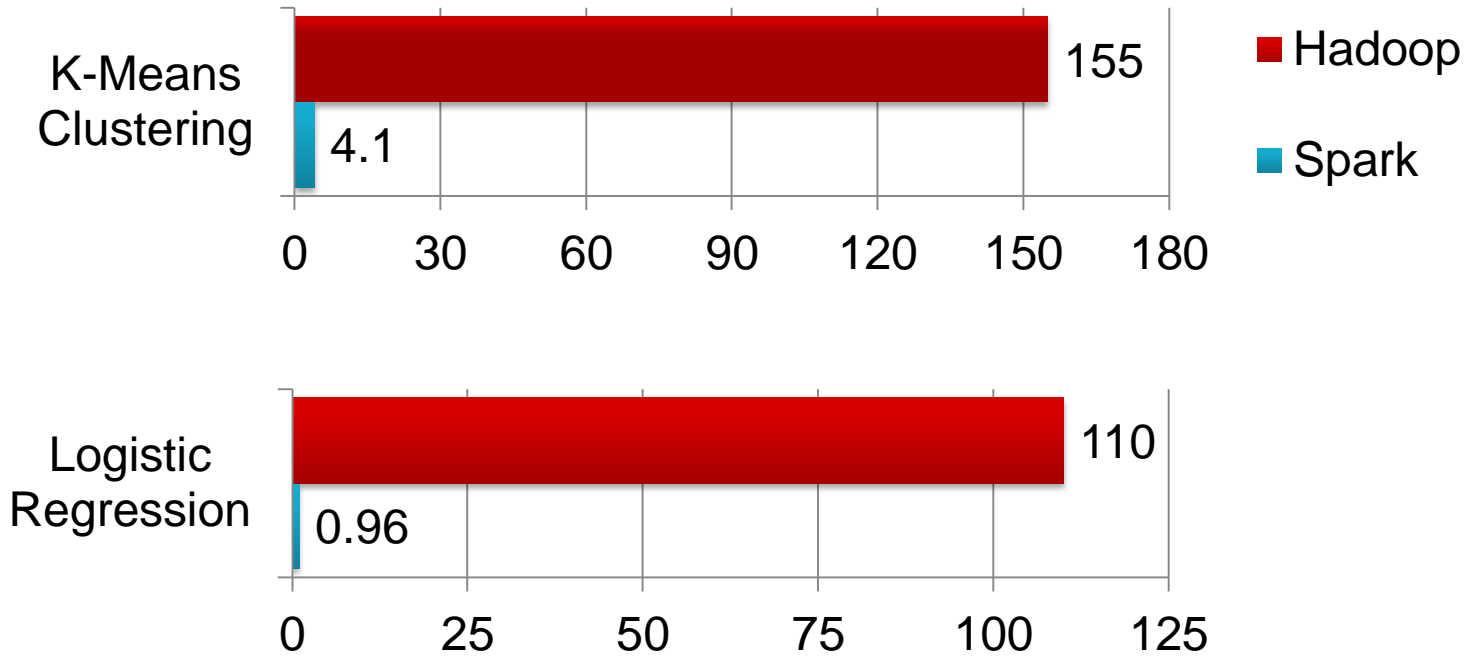
    contribs = links.join(ranks).flatMap(compute_contribs)
    ranks = contribs.reduceByKey(lambda x, y: x + y) \
        .mapValues(lambda x: 0.15 + 0.85 * x)

ranks.saveAsTextFile(...)
```

PageRank Performance



Other Iterative Algorithms



Time per Iteration (s)

Deployment Options

Local Mode

- ❑ Just pass `local` or `local[k]` as master URL
- ❑ Still serializes tasks to catch marshaling errors
- ❑ Debug using local debuggers
 - For Java and Scala, just run your main program in a debugger
 - For Python, use an attachable debugger (e.g. PyDev, winpdb)
- ❑ Great for unit testing

Private Cluster

- Can run with one of:
 - Standalone deploy mode (similar to Hadoop cluster scripts)
 - Apache Mesos: spark-project.org/docs/latest/running-on-mesos.html
 - Hadoop YARN: spark-project.org/docs/0.6.0/running-on-yarn.html

- Basically requires configuring a list of workers, running launch scripts, and passing a special cluster URL to SparkContext

Amazon EC2

- Easiest way to launch a Spark cluster

```
git clone git://github.com/mesos/spark.git
```

```
cd spark/ec2
```

```
./spark-ec2 -k keypair -i id_rsa.pem -s slaves \  
[launch|stop|start|destroy] clusterName
```

- Details: spark-project.org/docs/latest/ec2-scripts.html

Viewing Logs

- ❑ Click through the web UI at master:8080
- ❑ Or, look at stdout and stderr files in the Spark or Mesos “work” directory for your app:
`work/<ApplicationID>/<ExecutorID>/stdout`
- ❑ Application ID (Framework ID in Mesos) is printed when Spark connects

Conclusion

- ❑ Spark offers a rich API to make data analytics fast
 - both fast to write and fast to run
- ❑ Achieves 100x speedups in real applications
- ❑ Growing community with 14 companies contributing
- ❑ Details, tutorials, videos: www.spark-project.org

