

# Spark Programming Basics (2)

Lecture 4

November 1<sup>st</sup>, 2017

Jae W. Lee ([jaewlee@snu.ac.kr](mailto:jaewlee@snu.ac.kr))

Computer Science and Engineering

Seoul National University

***Slide credits:*** Paco Nathan (DataBricks), Prof. Anthony Joseph (UCBerkeleyX), Holden Karau et al. (Learning Spark)

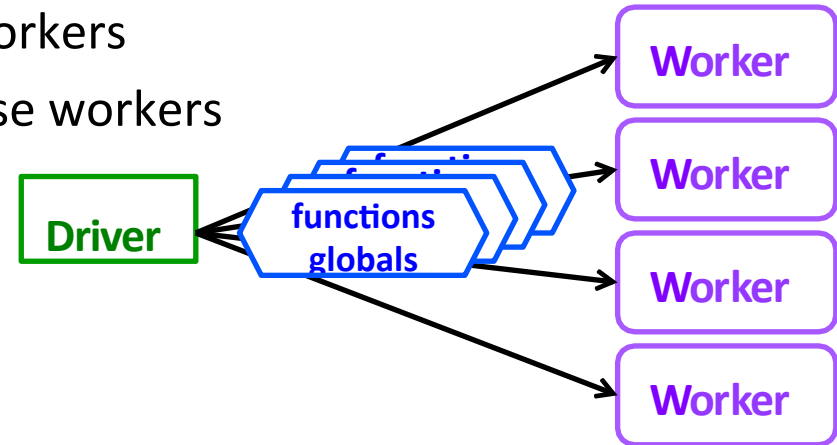
# Outline

- **Understanding Closures and Shared Variables**
- RDD Persistence
- Example: PageRank
- More Transformations and Actions
- Creating Spark Applications

# pySpark Closures

## ■ Spark automatically creates closures for:

- Functions that run on RDDs at workers
- Any global variables used by those workers



## ■ One closure per worker

- Sent for every task
- No communication between workers
- Changes to global variables at workers are not sent to driver

# Consider These Use Cases



- **Iterative or single jobs with large global variables**
  - Sending large read-only lookup table to workers
  - Sending large feature vector in a ML algorithm to workers
- **Counting events that occur during job execution**
  - How many input lines were blank?
  - How many input records were corrupt?

# Consider These Use Cases

- **Iterative or single jobs with large global variables**
  - Sending large read-only lookup table to workers
  - Sending large feature vector in a ML algorithm to workers
- **Counting events that occur during job execution**
  - How many input lines were blank?
  - How many input records were corrupt?

## Problems:

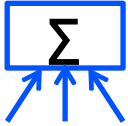
- Closures are (re-)sent with *every* job
- Inefficient to send large data to each worker
- Closures are one way: driver → worker

# pySpark Shared Variables



## Broadcast Variables

- Efficiently send large, *read-only* value to all workers
- Saved at workers for use in one or more Spark operations
- Like sending a large, read-only lookup table to all the nodes




+ + +

## Accumulators

- Aggregate values from workers back to driver
- Only driver can access value of accumulator
- For tasks, accumulators are write-only
- Use to count errors seen in RDD across workers

# Broadcast Variables



- Keep *read-only* variable cached on workers
  - Ship to each worker only once instead of with each task
- Example: efficiently give every worker a large dataset
- Usually distributed using efficient broadcast algorithms 

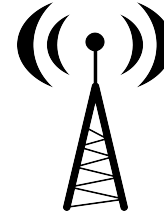
At the driver:

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
```

At a worker (in code passed via a closure)

```
>>> broadcastVar.value  
[1, 2, 3]
```

# Broadcast Variables Example



## ■ Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the  
# RDD contactCounts. We load a list of call sign  
# prefixes to country code to support this lookup  
signPrefixes = loadCallSignTable()
```

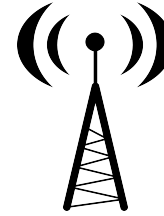
```
def processSignCount(sign_count, signPrefixes):  
    country = lookupCountry(sign_count[0], signPrefixes)  
    count = sign_count[1]  
    return (country, count)  
  
countryContactCounts = (contactCounts  
                        .map(processSignCount)  
                        .reduceByKey((lambda x, y: x+ y)))
```

Expensive to send large table  
(Re-)sent for every processed file

From: <http://shop.oreilly.com/product/0636920028512.do>



# Broadcast Variables Example



## ■ Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the  
# RDD contactCounts. We load a list of call sign  
# prefixes to country code to support this lookup  
signPrefixes = sc.broadcast(loadCallSignTable())
```

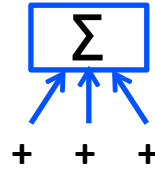
Efficiently sent once to workers

```
def processSignCount(sign_count, signPrefixes):  
    country = lookupCountry(sign_count[0], signPrefixes.value)  
    count = sign_count[1]  
    return (country, count)
```

```
countryContactCounts = (contactCounts  
    .map(processSignCount)  
    .reduceByKey((lambda x, y: x+ y)))
```

From: <http://shop.oreilly.com/product/0636920028512.do>

# Accumulators



- Variables that can only be “added” to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can read an accumulator’s value, not tasks

```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>     global accum
>>>     accum += x
```



```
>>> rdd.foreach(f)
>>> accum.value
Value: 10
```

# Accumulators Example


## ■ Counting empty lines

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```

# Accumulators

- **Tasks at workers cannot access accumulator's values**
- **Tasks see accumulators as write-only variables** 
- **Accumulators can be used in actions or transformations:**
  - Actions: each task's update to accumulator is **applied only once**
  - Transformations: **no guarantees** (use only for debugging)
- **Types: integers, double, long, float**

# Outline

- Understanding Closures and Shared Variables
- **RDD Persistence**
- Example: PageRank
- More Transformations and Actions
- Creating Spark Applications

# RDD Persistence

- Spark can *persist()* (or *cache()*) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than **10x** faster
- The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

# RDD Persistence

- **Storage level: you can do the following by controlling it**
  - allows you to persist the dataset on disks
  - allows you to persist it in memory but as serialized Java objects (to save space)
  - allows you to replicate it across nodes
- **Specifying storage level**
  - `persist()`: takes a `StorageLevel` object as parameter
  - `cache()`: uses the default storage level (`StorageLevel.MEMORY_ONLY`) to store de-serialized objects in memory

# RDD Persistence

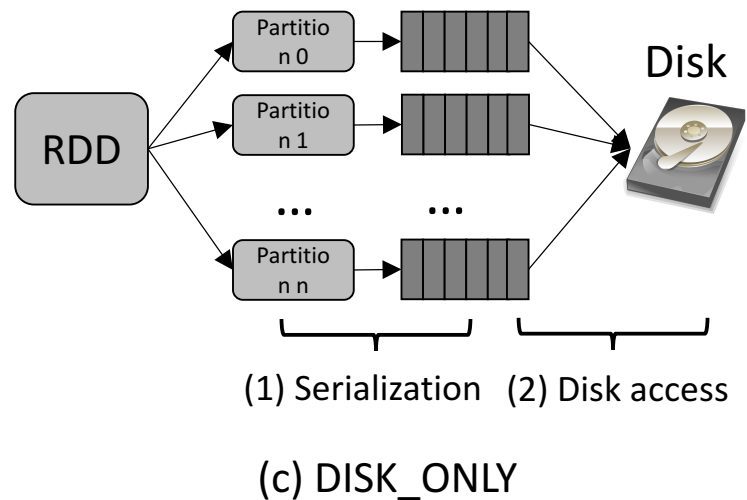
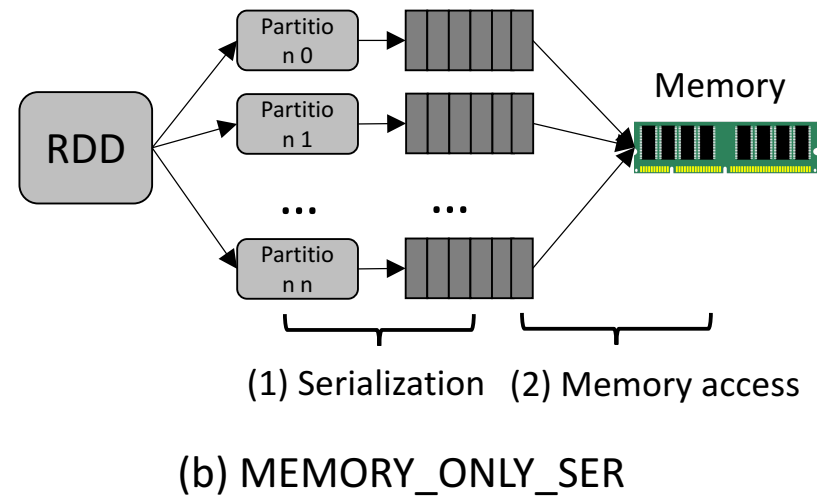
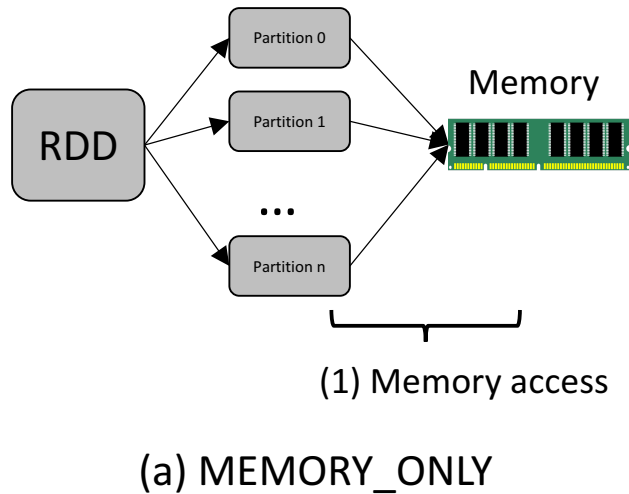
## ■ Example in Python

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' '))
    .map(lambda x: (x, 1))
    .cache()
w.reduceByKey(add).collect()
```




# RDD Persistence

## ■ Storage levels



# RDD Persistence

## ■ Storage levels

<i>Storage Level</i>	<i>Meaning</i>
<b>MEMORY_ONLY</b>	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
<b>MEMORY_AND_DISK</b> 	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
<b>MEMORY_ONLY_SER</b> (Not for Python)	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
<b>MEMORY_AND_DISK_SER</b> (Not for Python)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
<b>DISK_ONLY</b>	Store the RDD partitions only on disk.
<b>MEMORY_ONLY_2,</b> <b>MEMORY_AND_DISK_2, etc.</b>	Same as the levels above, but replicate each partition on two cluster nodes.

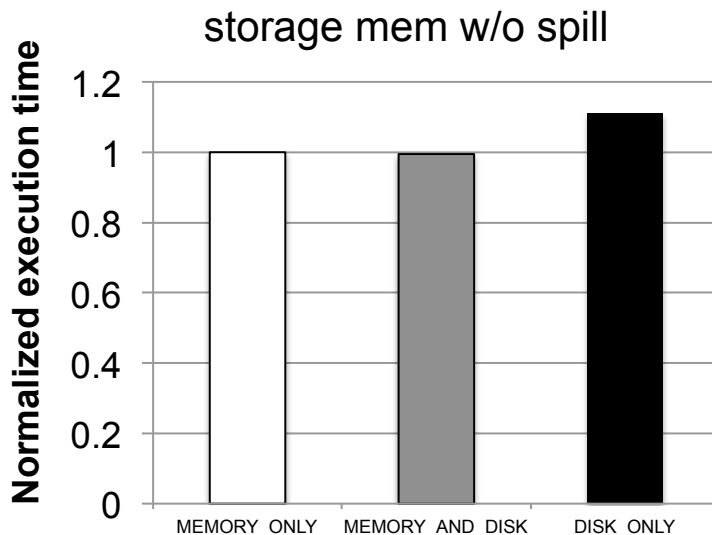
# RDD Persistence



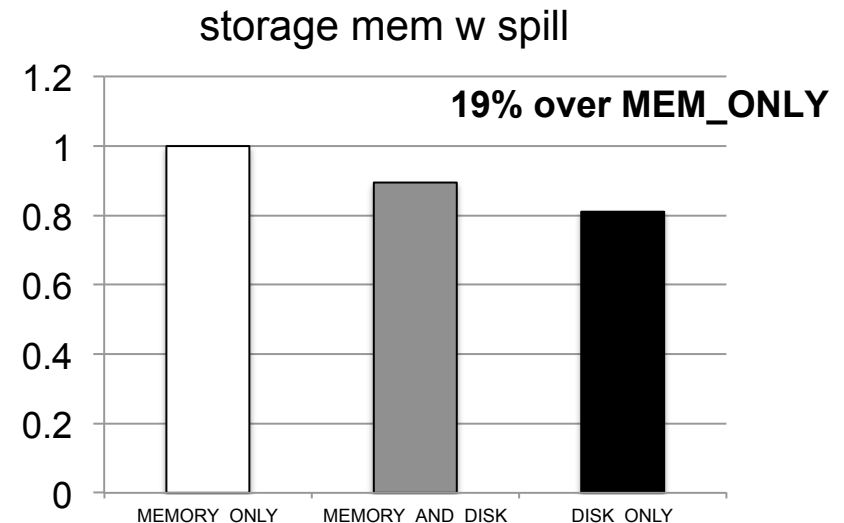
## ■ Performance issue of caching

- In-memory RDD caching is not always effective.
- For iterative applications, when the cached RDD overflows the (storage) memory, MEMORY\_ONLY caching causes significant overhead.

PageRank: 5,500,000 pages



PageRank: 15,000,000 pages



# Outline

- Understanding Closures and Shared Variables
- RDD Persistence
- **Example: PageRank**
- More Transformations and Actions
- Creating Spark Applications

# Example: PageRank

- **The main idea behind Google search engine (1998)**
  - Written by its two co-founders: Larry Page and Sergey Brin
- **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

## The PageRank Citation Ranking: Bringing Order to the Web

January 29, 1998

### Abstract

The importance of a Web page is an inherently subjective matter, which depends on the readers interests, knowledge and attitudes. But there is still much that can be said objectively about the relative importance of Web pages. This paper describes PageRank, a method for rating Web pages objectively and mechanically, effectively measuring the human interest and attention devoted to them.

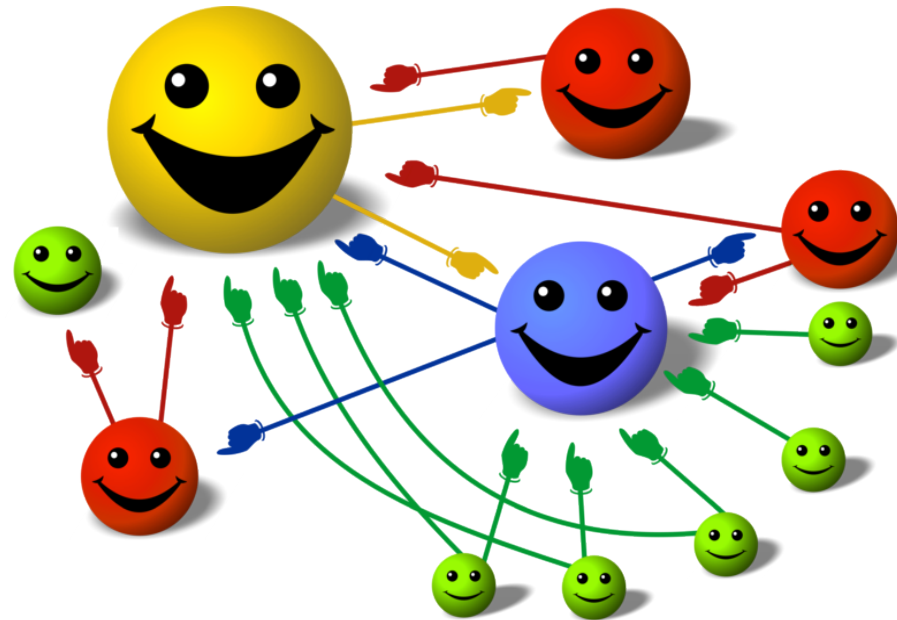
We compare PageRank to an idealized random Web surfer. We show how to efficiently compute PageRank for large numbers of pages. And, we show how to apply PageRank to search and to user navigation.

### 1 Introduction and Motivation

The World Wide Web creates many new challenges for information retrieval. It is very large and heterogeneous. Current estimates are that there are over 150 million web pages with a doubling life of less than one year. More importantly, the web pages are extremely diverse, ranging from "What is Joe having for lunch today?" to journals about information retrieval. In addition to these major challenges, search engines on the Web must also contend with inexperienced users and pages engineered to manipulate search engine ranking functions.

# Example: PageRank

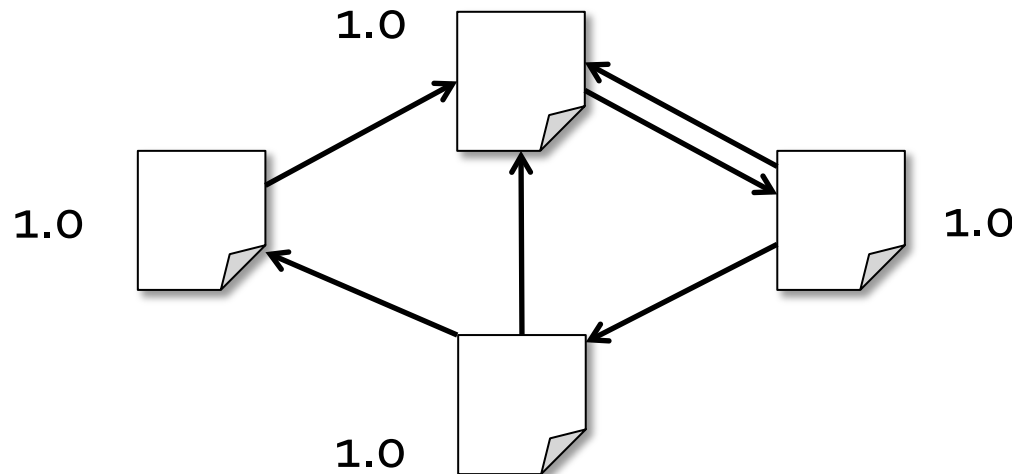
- **Basic idea: Give pages ranks (scores) based on links to them**
  - Links from many pages → high rank
  - Link from a high-rank page → high rank



# Example: PageRank

## ■ Algorithm

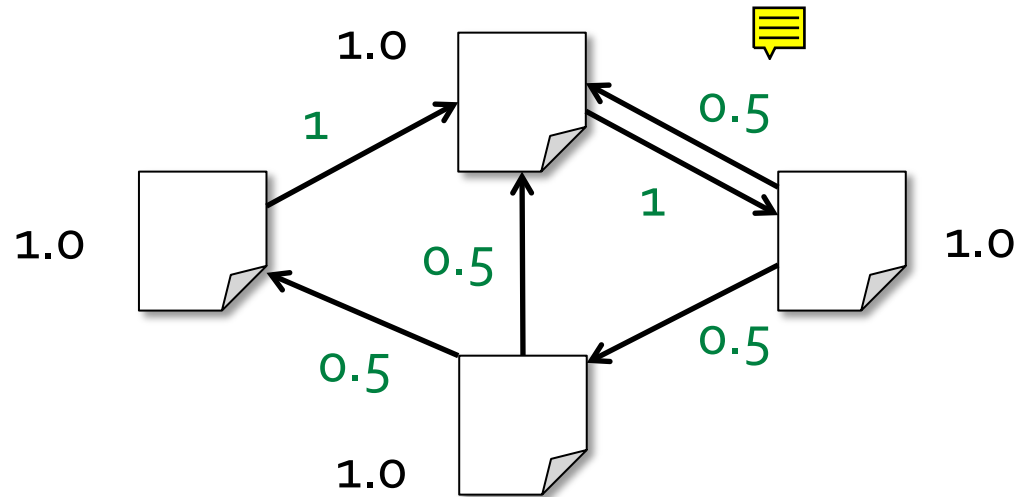
1. Start each page at a rank of 1
2. On each iteration, have page **p** contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



# Example: PageRank

## ■ 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  $0.15 + 0.85 \times \text{contribs}$

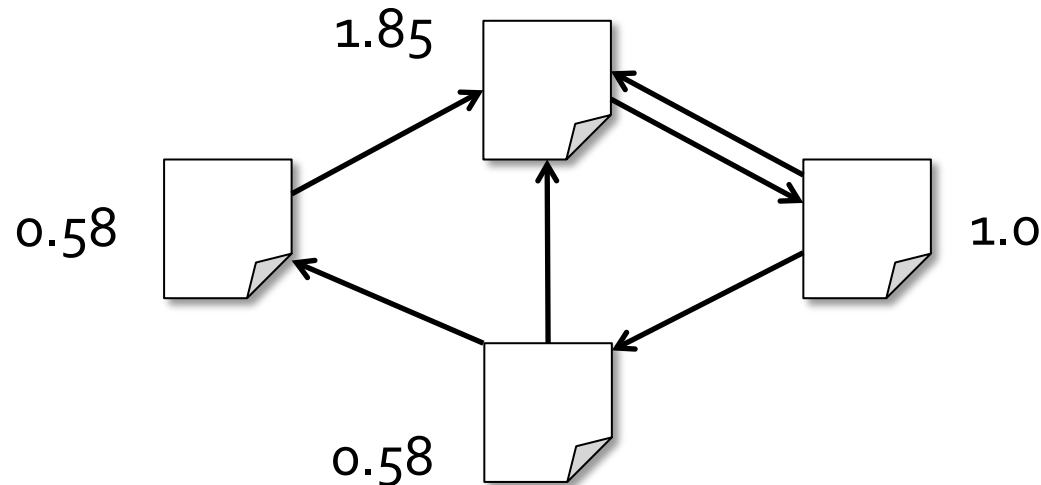




# Example: PageRank

## ■ Algorithm

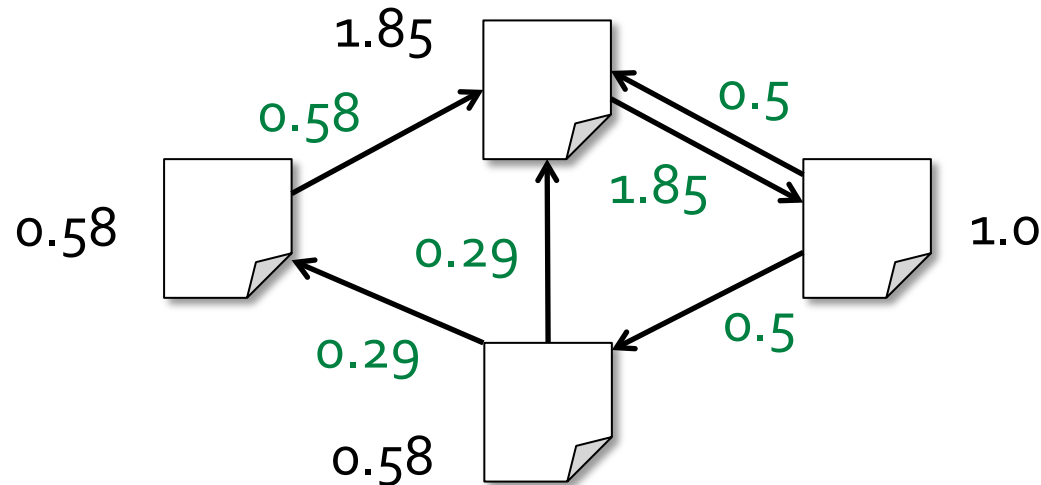
1. Start each page at a rank of 1
2. On each iteration, have page **p** contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



# Example: PageRank

## ■ Algorithm

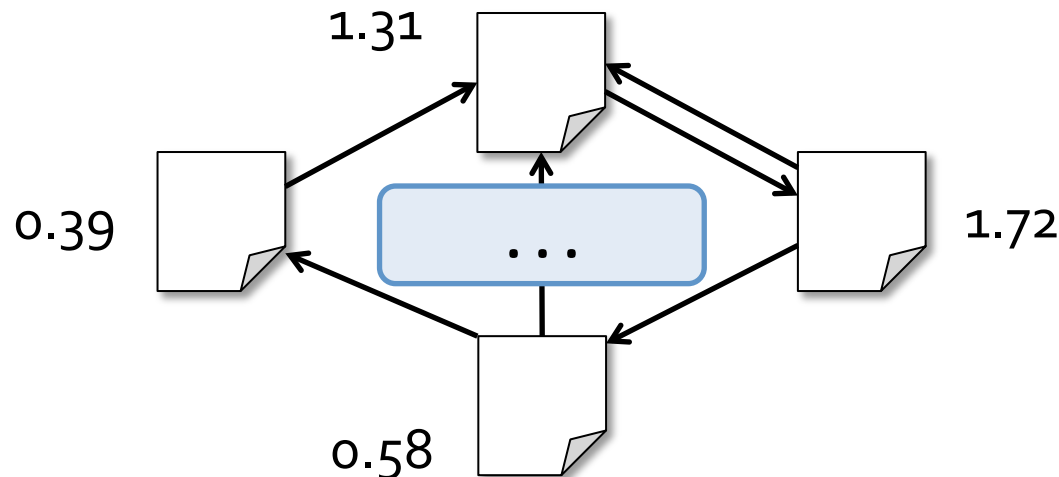
1. Start each page at a rank of 1
2. On each iteration, have page **p** contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



# Example: PageRank

## ■ 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  $0.15 + 0.85 \times \text{contribs}$

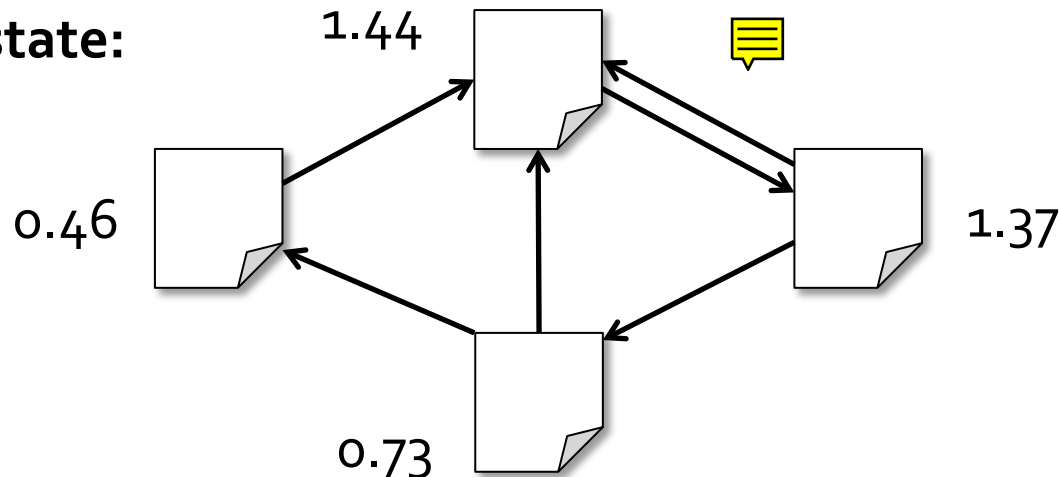


# Example: PageRank

## ■ 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:



# Example: PageRank

## ■ A Python implementation

- You will see more details at the lab tomorrow.

```
# Loads in input file. It should be in format of:
#   URL           neighbor URL
#   URL           neighbor URL
#   URL           neighbor URL
#   ...
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])

# Loads all URLs from input file and initialize their neighbors.
links = lines.map(lambda urls: parseNeighbors(urls)).distinct().groupByKey().cache()

# Loads all URLs with other URL(s) link to from input file and initialize ranks of them to one.
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))

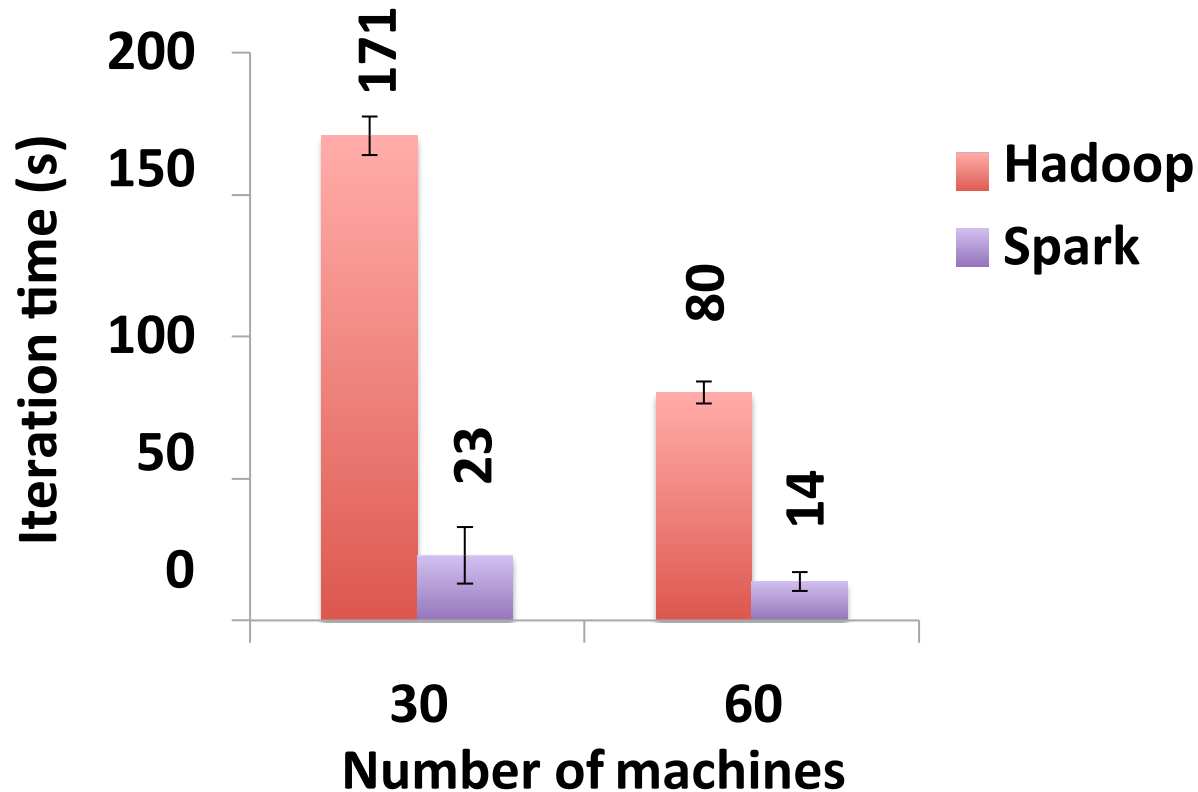
# Calculates and updates URL ranks continuously using PageRank algorithm.
for iteration in range(int(sys.argv[2])):
    # Calculates URL contributions to the rank of other URLs.
    contribs = links.join(ranks).flatMap(
        lambda url_urls_rank: computeContribs(url_urls_rank[1][0], url_urls_rank[1][1]))

    # Re-calculates URL ranks based on neighbor contributions.
    ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15)

# Collects all URL ranks and dump them to console.
for (link, rank) in ranks.collect():
    print("%s has rank: %s." % (link, rank))
```

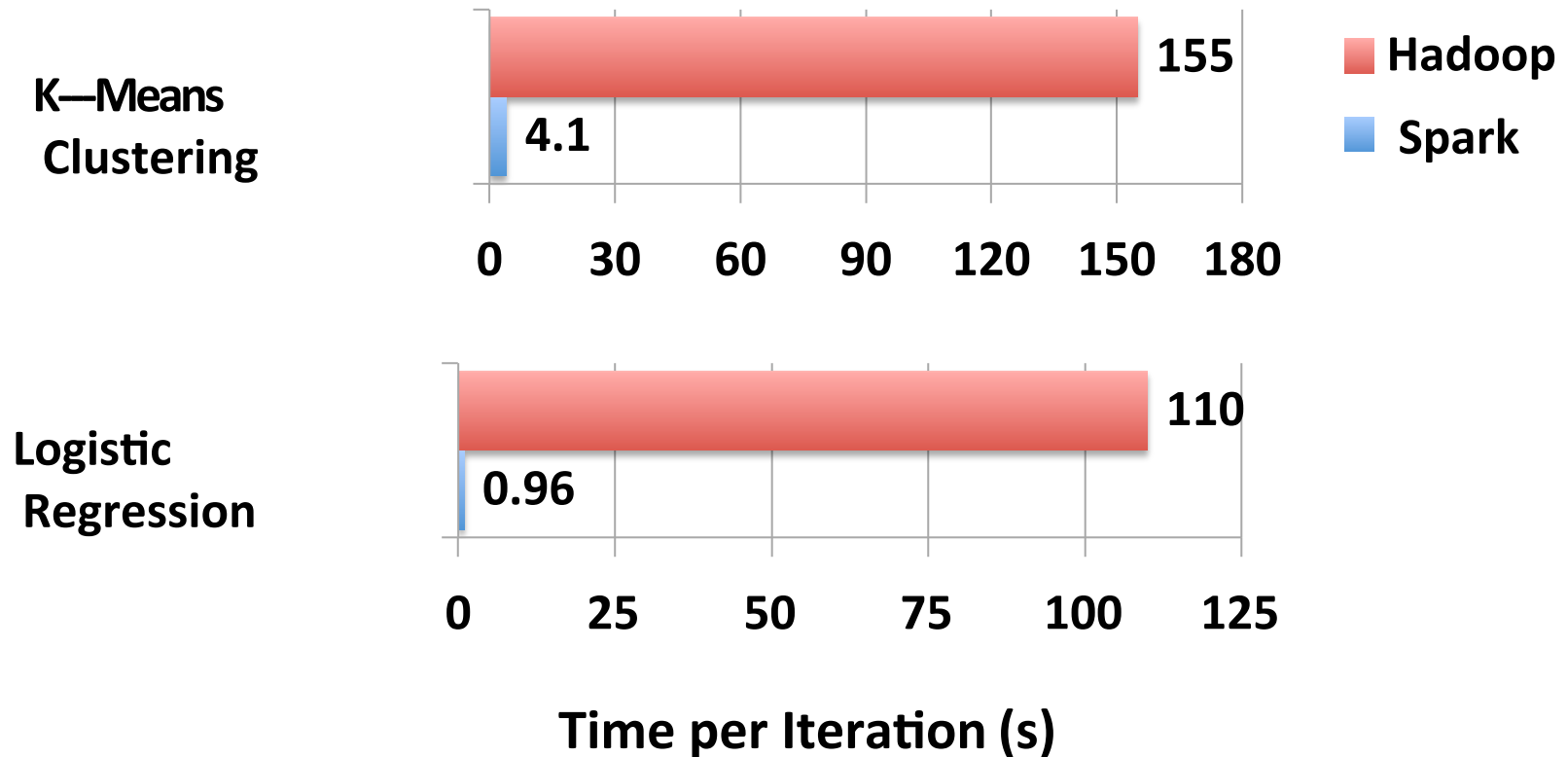
# Example: PageRank

## ■ Performance



# Example: PageRank

## ■ Other iterative algorithms



# Outline

- Understanding Closures and Shared Variables
- RDD Persistence
- Example: PageRank
- **More Transformations and Actions**
- Creating Spark Applications



# More Transformations (1)

- `mapPartitions(func, preservesPartitioning=False)`
  - Return a new RDD by applying a function to each partition of this RDD.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

# More Transformations (2)

- `mapPartitionsWithIndex(func, preservesPartitioning=False)`
  - Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(splitIndex, iterator): yield splitIndex
>>> rdd.mapPartitionsWithIndex(f).sum()
6
```

# More Transformations (3)

## ■ `sample(withReplacement, fraction, seed=None)`

- Return a sampled subset of this RDD
- Parameters
  - *withReplacement*: can elements be sampled multiple times (replaced when sampled out)
  - *fraction*: expected size of the sample as a fraction of this RDD's size without replacement
  - *seed*: seed for the random number generator

```
>>> rdd = sc.parallelize(range(100), 4)
>>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14
True
```

# More Transformations (4)

## ■ `aggregate(zeroValue, seqOp, combOp)`

- Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral “zero value.”
- The first function (seqOp) can return a different result type, U, than the type of this RDD.
  - Thus, we need one operation for merging a T into an U and one operation for merging two Us.

```
>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))  
>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))  
>>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)  
(10, 4)  
>>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)  
(0, 0)
```



# More Transformations (5)

## ■ `cartesian(otherDataset)`

- Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements  $(a, b)$  where **a** is in **self** and **b** is in **other**.

```
>>> rdd = sc.parallelize([1, 2])
>>> sorted(rdd.cartesian(rdd).collect())
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

# More Transformations (6)

## ■ `pipe(command, [envVars])`

- Return an RDD created by piping elements to a forked external process

```
>>> sc.parallelize([1, 2, 3, 4]).pipe('cat').collect()
[u'1', u'2', u'3', u'4']
>>> sc.parallelize([1, 2, 3, 4]).pipe('ls').collect()
[u'CONTRIBUTING.md', u'LICENSE', u'NOTICE', u'R',
u'README.md', u'appveyor.yml', u'assembly', u'bin',
u'build', u'common', u'conf', u'core', u'data', u'dev',
u'docs', u'examples', u'external', u'graphx',
u'hello.txt', u'launcher', u'licenses', u'logs',
u'mesos', u'mllib', u'mllib-local', u'pom.xml',
u'project', u'python', u'repl', u'sbin', u'scalastyle-
config.xml', u'slaves', u'spark-default.conf', u'spark-
warehouse', u'sql', u'streaming', u'target', u'tools',
u'work', u'world.txt', u'yarn']
```

# More Transformations (7)

## ■ `coalesce(numPartitions)`

- Return a new RDD that is reduced into *numPartitions* partitions.

```
>>> sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
[[1], [2, 3], [4, 5]]
>>> sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()
[[1, 2, 3, 4, 5]]
```

# More Transformations (8)

## ■ `repartition(numPartitions)`

- Return a new RDD that has exactly *numPartitions* partitions.
- Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data. If you are decreasing the number of partitions in this RDD, consider using *coalesce()*, which can avoid performing a shuffle.

```
>>> rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
>>> sorted(rdd.glom().collect())
[[1], [2, 3], [4, 5], [6, 7]]
>>> len(rdd.repartition(2).glom().collect())
2
>>> len(rdd.repartition(10).glom().collect())
10
```



# More Transformations (9)

## ■ `zip(other)`

- Zips this RDD with another one, returning key-value pairs with the first element in each RDD, second element in each RDD etc.
- Assumes that the two RDDs have the same number of partitions and the same number of elements in each partition (e.g. one was made through a map on the other)

```
>>> x = sc.parallelize(range(0,5))
>>> y = sc.parallelize(range(1000, 1005))
>>> x.zip(y).collect()
[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

# More Actions (1)

- **takeSample(withReplacement, num, [seed])**
  - Return a fixed-size sampled subset of this RDD.
  - **Note:** This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> rdd = sc.parallelize(range(0, 10))
>>> len(rdd.takeSample(True, 20, 1))
20
>>> len(rdd.takeSample(False, 5, 2))
5
>>> len(rdd.takeSample(False, 15, 3))
10
```

# More Actions (2)


## ■ `foreach(f)`

- Applies a function  $f$  to all elements of this RDD.

```
>>> def f(x): print(x)
>>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)
```

```
1
3
2
4
5
```

# Outline

- Understanding Closures and Shared Variables
- RDD Persistence
- Example: PageRank
- More Transformations and Actions
- **Creating Spark Applications** 


# Launching a Program

## ■ **spark-submit** script

- No matter which cluster manager you use, you can use `spark-submit` to submit your program to it.
- Through various options, you can connect to different cluster managers and control how many resources your application gets.
  - For some cluster managers, `spark-submit` can run the driver within the cluster (e.g., on a YARN worker node), while for others, it can run it only on your local machine.

# Launching a Program

## ■ What happens if you run your application?

1. The user submits an application using `spark-submit`. 
2. `spark-submit` launches the driver program and invokes the `main()` method specified by the user.
3. The driver program contacts the cluster manager to ask for resources to launch executors.
4. The cluster manager launches executors on behalf of the driver program.
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks.
6. Tasks are run on executor processes to compute and save results.
7. If the driver's `main()` method exits or it calls `SparkContext.stop()`, it will terminate the executors and release resources from the cluster manager.

# Launching a Program



## ■ General format for spark-submit

```
$ bin/spark-submit [options] <app jar | python file> [app options]
```

## ■ Common flags

<i>Flag</i>	<i>Description</i>
<b>--master</b>	Indicates the cluster manager to connect to. (e.g., local[N], yarn, spark://host:port)
<b>--name</b>	A human-readable name for your application. This will be displayed in Spark's web UI.
<b>--py-files</b>	A list of files to be added to the PYTHONPATH of your application. This can contain .py, .egg, or .zip files.
<b>--files</b>	A list of files to be placed in the working directory of your application. This can be used for data files that you want to distribute to each node.
<b>--executor-memory</b>	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes).
<b>--driver-memory</b>	The amount of memory to use for the driver process, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes).

# Creating a pySpark Application

## ■ Creating a Spark context

```
from pyspark import SparkContext
```

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

- masterUrl: path to master node (local, cluster manager, etc.)
- appName: application name
- sparkHome: path where Spark is installed on worker nodes
- pyFiles: Python files to include



# Creating a pySpark Application

## ■ WordCount: A complete app

```
import sys
from pyspark import SparkContext

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

    counts = lines.flatMap(lambda s: s.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda x, y: x + y)

    counts.saveAsTextFile(sys.argv[2])
```

# Creating a pySpark Application

## ■ Run it using spark-submit

```
$ bin/spark-submit wordcount.py LICENSE output
17/11/01 02:25:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
```

```
$ cat output/part-00000
(u'', 1430)
(u'limited', 4)
(u'all', 3)
(u'code', 2)
(u'managed', 1)
(u'incurred', 1)
(u'(Two-clause', 1)
(u'Fortran', 1)
(u'(org.antlr:ST4:4.0.4', 1)
(u'customary', 1)
(u'(all', 1)
(u'Works,', 2)
(u'APPENDIX:', 1)
(u'"Legal', 1)
(u'granting', 1)
(u'description', 1)
(u'Object', 4)
(u'acting', 1)
(u'special,', 1)
(u'to', 42)...
```