
Apache Spark

CS240A

T Yang

Some of them are based on P. Wendell's Spark slides

Parallel Processing using Spark+Hadoop

- **Hadoop: Distributed file system that connects machines.**
- **Mapreduce: parallel programming style built on a Hadoop cluster**
- **Spark: Berkeley design of Mapreduce programming**
- **Given a file treated as a big list**
 - A file may be divided into multiple parts (splits).
- **Each record (line) is processed by a Map function,**
 - produces a set of intermediate key/value pairs.
- **Reduce: combine a set of values for the same key**

Python Examples for List Processing

```
>>> lst = [3, 1, 4, 1, 5]
>>> lst.append(2)
>>> len(lst)
5
>>> lst.sort()
>>> lst.insert(4, "Hello")
>>> [1]+ [2]      → [1,2]
>>> lst[0] ->3
```

Python tuples

```
>>> num=(1, 2, 3, 4)
>>> num + (5) →
(1,2,3,4, 5)
```

```
>>>numset=set([1, 2, 3, 2])
Duplicated entries are deleted
```

```
>>>numset=frozenset([1, 2,3])
Such a set cannot be modified
```

```
for i in [5, 4, 3, 2, 1]:
    print i
```

```
>>>M = [x for x in S if x % 2 == 0]
>>> S = [x**2 for x in range(10)]
[0,1,4,9,16,...,81]
```

```
>>> words = 'hello lazy dog'.split()
→ ['hello', 'lazy', 'dog']
>>> stuff = [(w.upper(), len(w)) for w in words]
→ [ ('HELLO', 5) ('LAZY', 4) , ('DOG', 4)]
```

Python map/reduce

```
a = [1, 2, 3]
```

```
b = [4, 5, 6, 7]
```

```
c = [8, 9, 1, 2, 3]
```

```
f = lambda x: len(x)
```

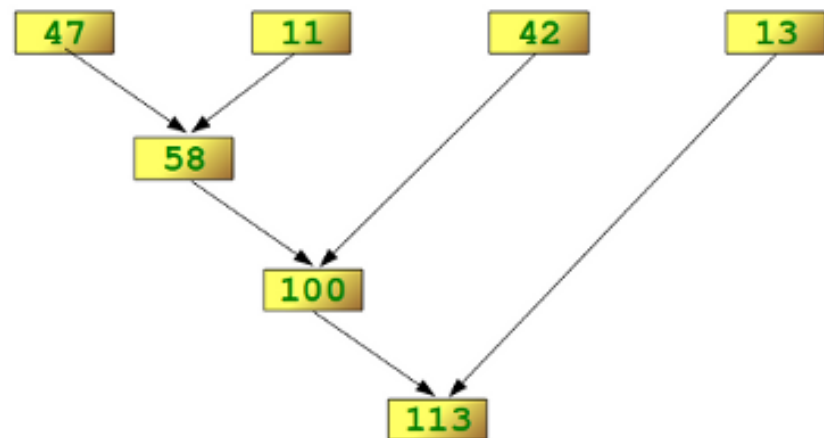
```
L = map(f, [a, b, c])
```

```
[3, 4, 5]
```

```
g = lambda x, y: x + y
```

```
reduce(g, [47, 11, 42, 13])
```

```
113
```

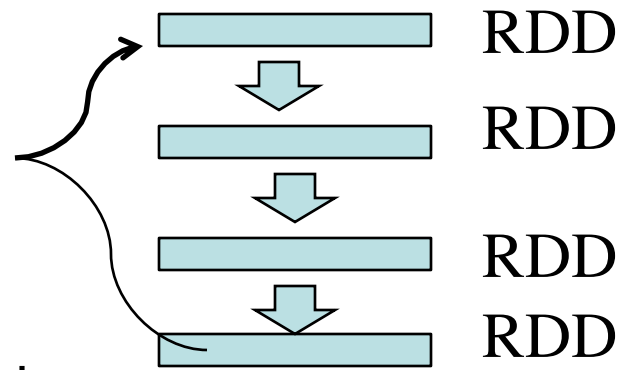


Mapreduce programming with SPAK: key concept

Write programs in terms of **operations** on implicitly distributed **datasets (RDD)**

RDD: Resilient Distributed Datasets

- **Like a big list:**
 - Collections of objects spread across a cluster, stored in RAM or on Disk
- **Built through parallel transformations**
- **Automatically rebuilt on failure**



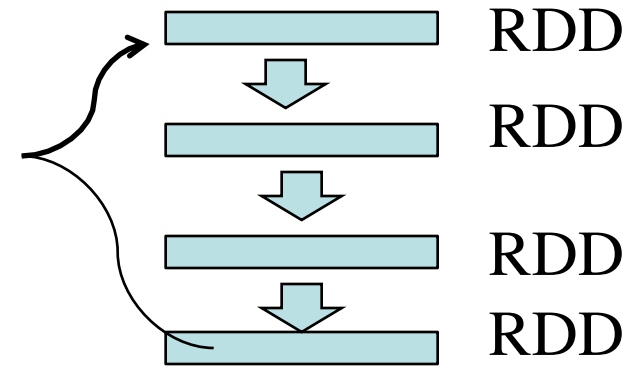
Operations

- **Transformations** (e.g. map, filter, groupBy)
- **Make sure input/output match**

MapReduce vs Spark



Map and reduce tasks operate on key-value pairs



Spark operates on **RDD** with aggressive memory caching

Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

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

Standalone Programs

- Python, Scala, & Java

Interactive Shells

- Python & Scala

Performance

- Java & Scala are faster due to static typing
- ...but Python is often fine

Spark Context and Creating RDDs

#Start with `sc` – `SparkContext` as
Main entry point to Spark functionality

Turn a Python collection into an RDD

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



RDD

Load text file from local FS, HDFS, or S3

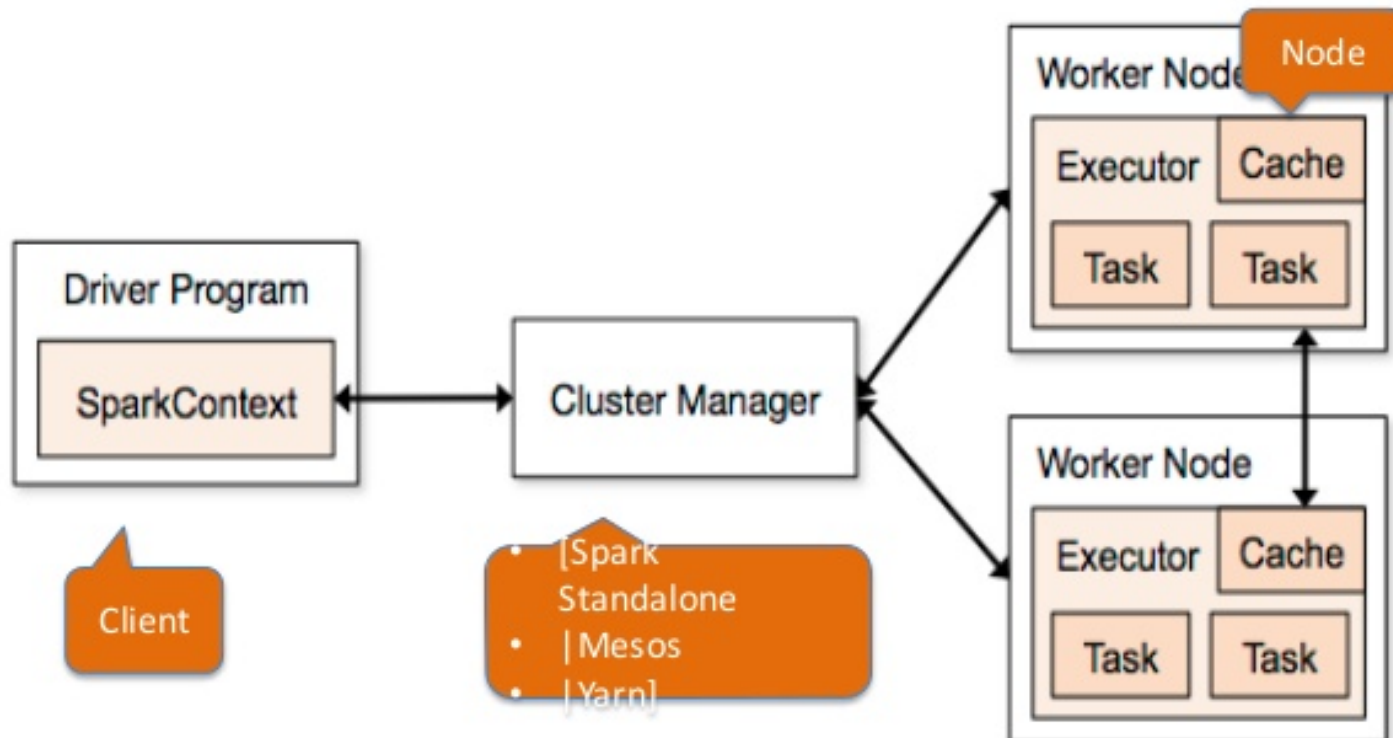
```
> sc.textFile("file.txt")
```

```
> sc.textFile("directory/*.txt")
```

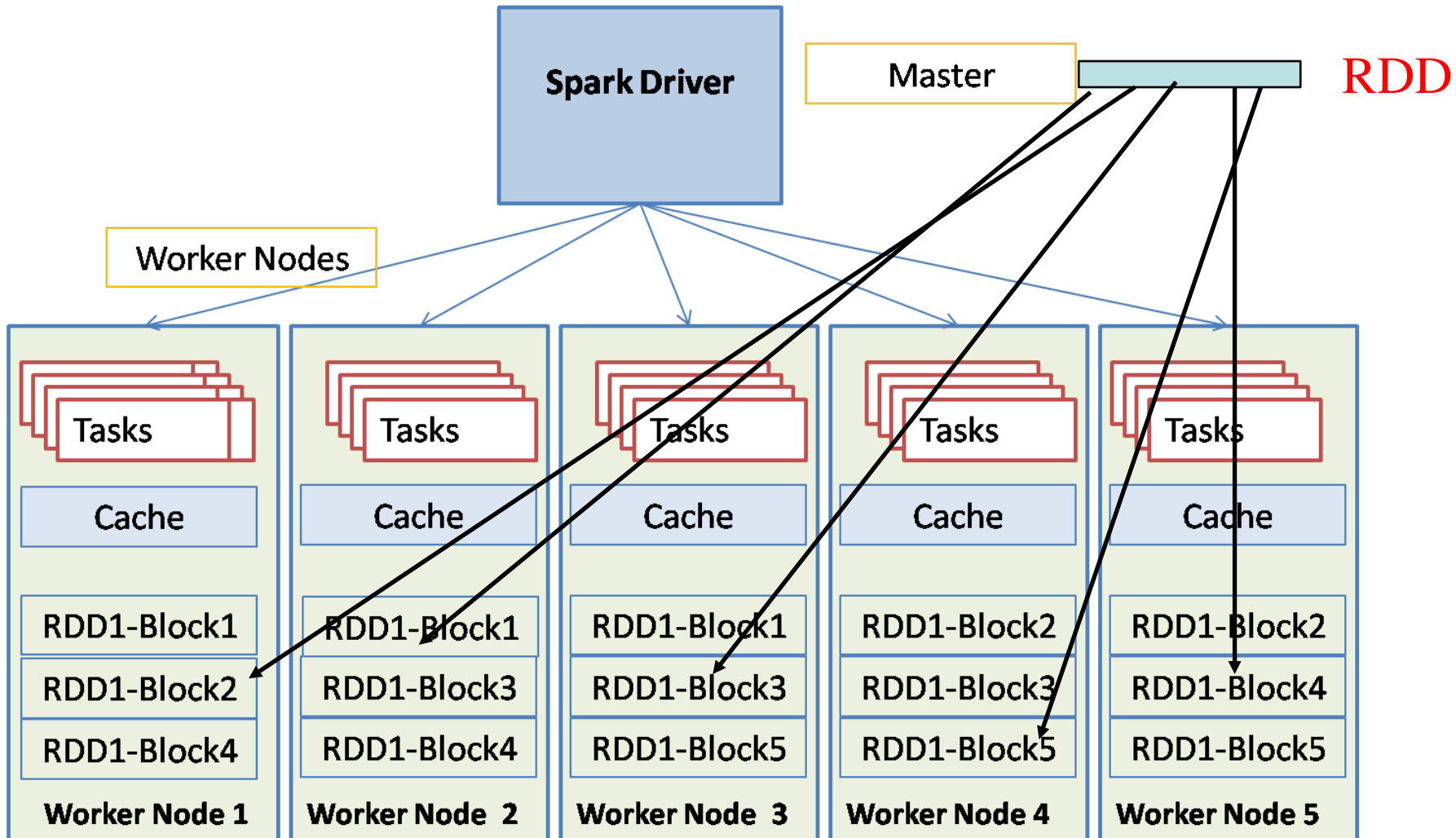
```
> sc.textFile("hdfs://namenode:9000/path/file")
```


Spark Architecture

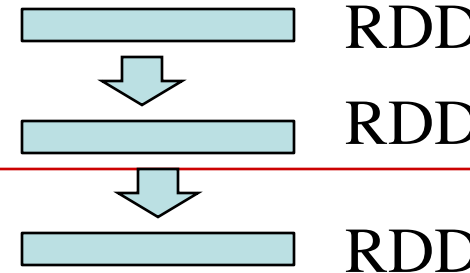
Spark Architecture



Spark Components



Basic Transformations



```
#read a text file and count number of lines  
containing error
```

```
lines = sc.textFile("file.log")  
lines.filter(lambda s: "ERROR" in s).count()
```

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

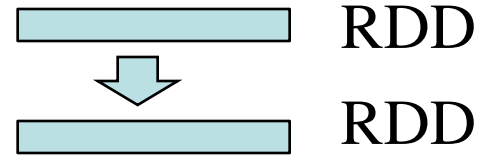
```
# Pass each element through a function
```

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

```
# Keep elements passing a predicate
```

```
> even = squares.filter(lambda x: x % 2 == 0) // {4}
```

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:

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

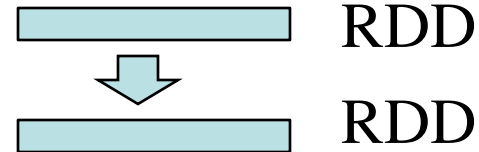
Scala:

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

Java:

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

Some Key-Value Operations

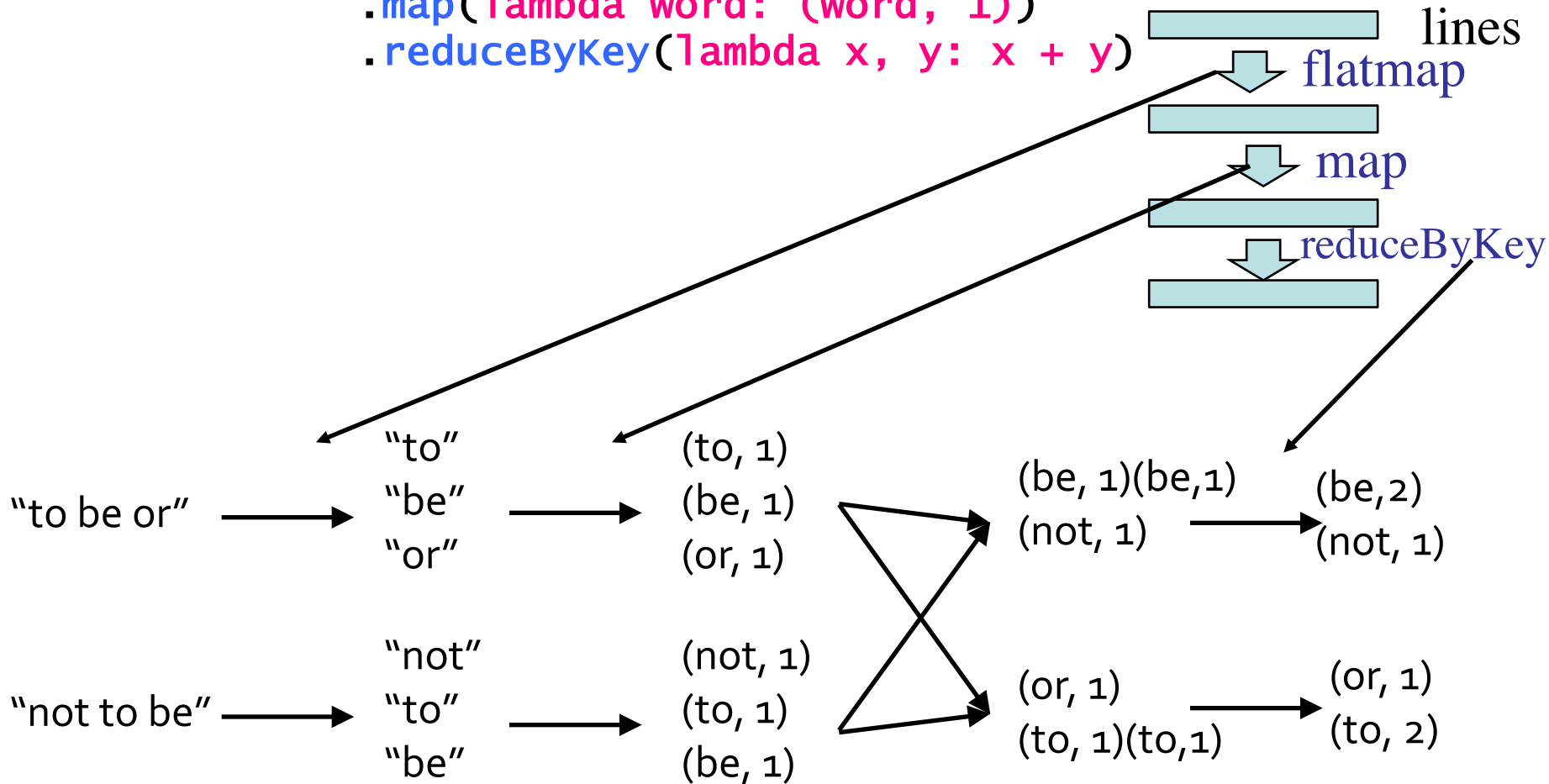


- > `pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])`
- > `pets.reduceByKey(lambda x, y: x + y)`
=> `{(cat, 3), (dog, 1)}`
- > `pets.groupByKey()` # => `{(cat, [1, 2]), (dog, [1])}`
- > `pets.sortByKey()` # => `{(cat, 1), (cat, 2), (dog, 1)}`

`reduceByKey()` also automatically implements combiners on the map side

Example: Word Count

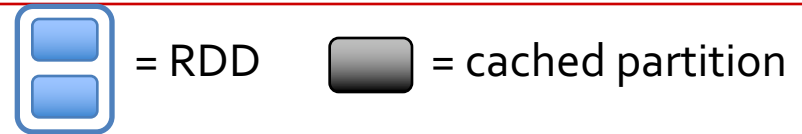
```
> lines = sc.textFile("hamlet.txt")  
> counts = lines.flatMap(lambda line: line.split(" "))  
                  .map(lambda word: (word, 1))  
                  .reduceByKey(lambda x, y: x + y)
```



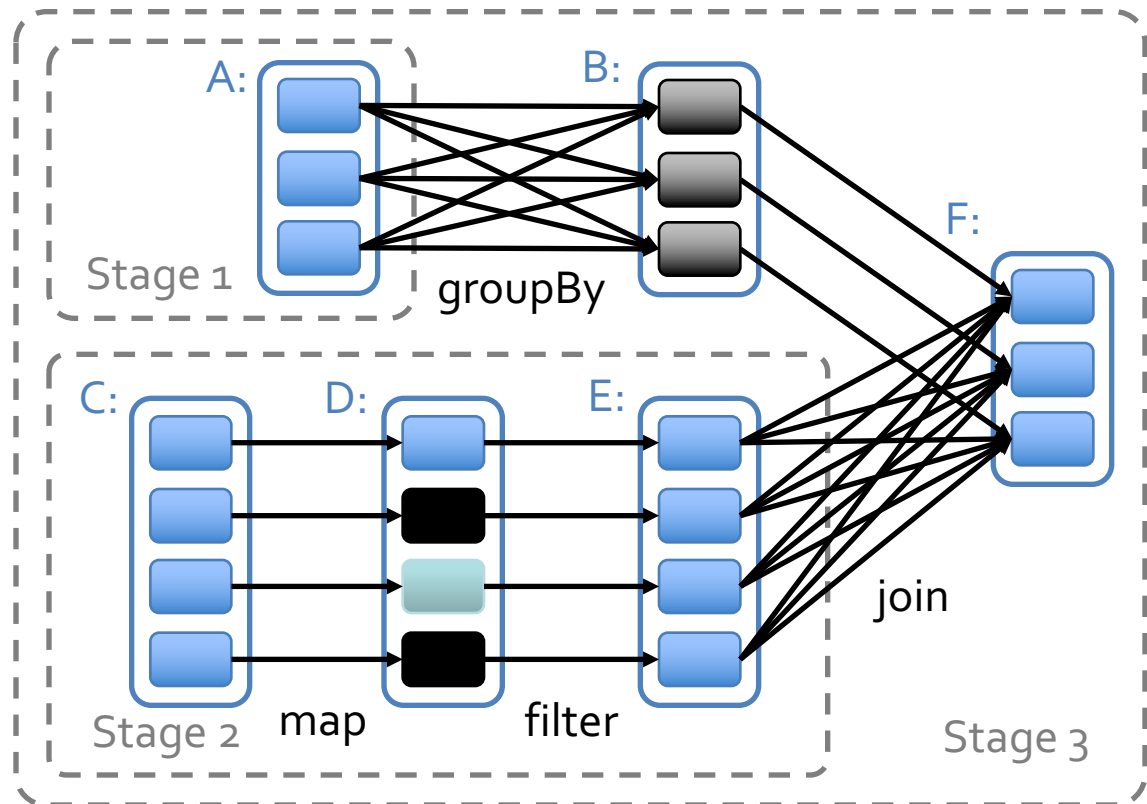
Other Key-Value Operations

- > `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"]))

Under The Hood: DAG Scheduler



- **General task graphs**
- **Automatically pipelines functions**
- **Data locality aware**
- **Partitioning aware to avoid shuffles**



Setting the Level of Parallelism

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

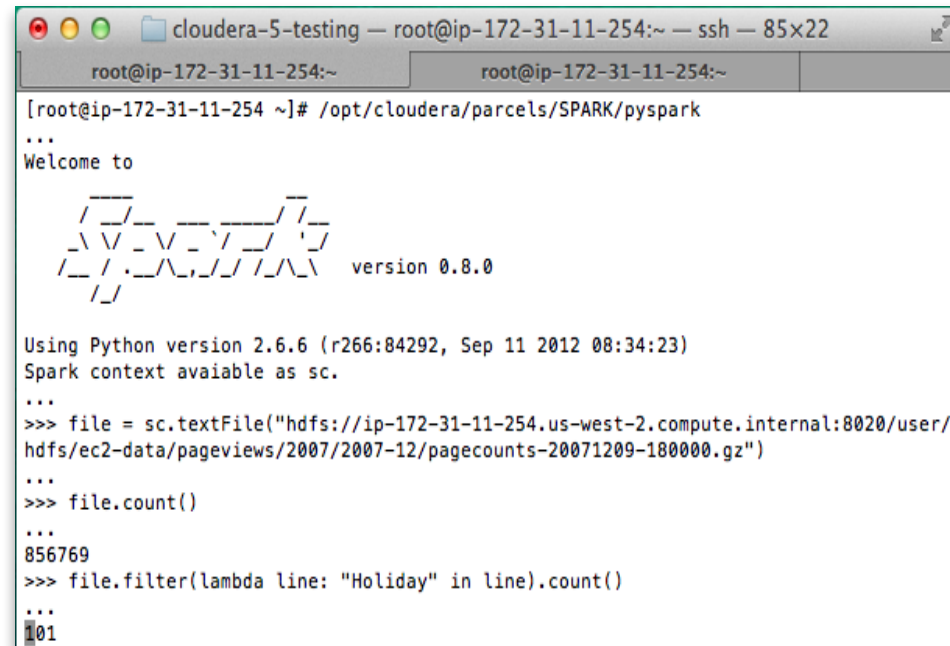
- > words.reduceByKey(lambda x, y: x + y, 5)
- > words.groupByKey(5)
- > visits.join(pageViews, 5)

More RDD Operators

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip
- sample
- take
- first
- partitionBy
- mapWith
- pipe
- save ...

Interactive Shell

- **The Fastest Way to Learn Spark**
- **Available in Python and Scala**
- **Runs as an application on an existing Spark Cluster...**
- **OR Can run locally**



```
[root@ip-172-31-11-254 ~]# /opt/cloudera/parcels/SPARK/pyspark
...
Welcome to

  ____  _
 / ___\/  \ \__  |
 \___ \|  / /  / /
  ___/ | / /_/ /
 /___ \| /___/ /
      /___/

version 0.8.0

Using Python version 2.6.6 (r266:84292, Sep 11 2012 08:34:23)
Spark context available as sc.
...
>>> file = sc.textFile("hdfs://ip-172-31-11-254.us-west-2.compute.internal:8020/user/
hdfs/ec2-data/pageviews/2007/2007-12/pagecounts-20071209-180000.gz")
...
>>> file.count()
...
856769
>>> file.filter(lambda line: "Holiday" in line).count()
...
101
```

... or a Standalone Application

```
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])
```

Create a SparkContext

Scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

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

Java

```
import org.apache. Cluster URL, or local / local[N]
JavaSparkContext sc = new JavaSparkContext App name
    "masterUrl", "name", "sparkHome", new String[] { "app.jar" }); Spark install path on cluster
List of JARs with app code (to ship)
```

Python

```
from pyspark import SparkContext

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

Administrative GUIs

http://<Standalone Master>:8080

(by default)

The image shows two browser windows. The main window displays the Spark Master administrative GUI at localhost:8080. It includes a sidebar with system metrics and a 'Workers' table. A red box highlights the 'app-20131202231712-0000' entry in the 'Running Applications' table. An orange arrow points from this box to the 'Spark Stages' window. The 'Spark Stages' window is at localhost:4040/stages/ and shows details for 'Spark Stages', including 'Total Duration: 3.8 m' and 'Scheduling Mode: FIFO'. It contains two tables: 'Active Stages (0)' and 'Completed Stages (2)'. The 'Completed Stages' table lists two stages: 'count at <console>:13' and 'reduceByKey at <console>:13', both with 2/2 tasks succeeded.

Spark Master at spark://mbp-2.local:7077

URL: spark://mbp-2.local:7077
Workers: 3
Cores: 24 Total, 24 Used
Memory: 45.0 GB Total, 1536.0 MB Used
Applications: Running, 0 Completed

Workers

Id
worker-20131202231645-192.168.1.106-56789
worker-20131202231657-192.168.1.106-56801
worker-20131202231705-192.168.1.106-56806

Running Applications

ID	Name
app-20131202231712-0000	Spark shell

Spark Stages

Total Duration: 3.8 m
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 2
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle
0	count at <console>:13	2013/12/02 21:07:55	83 ms	2/2	754.0 B
1	reduceByKey at <console>:13	2013/12/02 21:07:55	345 ms	2/2	

Failed Stages (0)

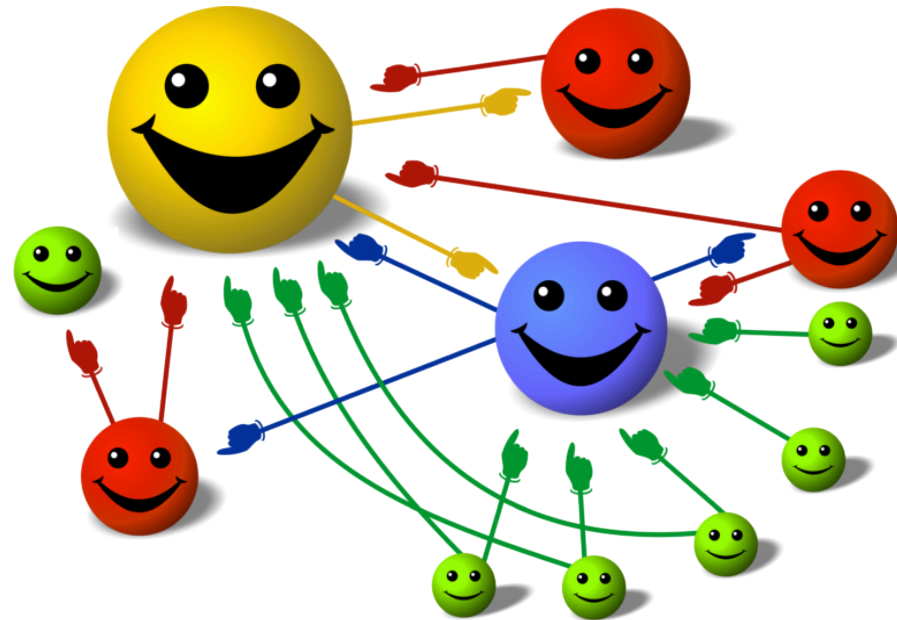
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read
----------	-------------	-----------	----------	------------------------	--------------

EXAMPLE APPLICATION: PAGERANK

Google PageRank

Give pages ranks (scores) based on links to them

- Links from many pages → high rank
- Link from a high-rank page → high rank

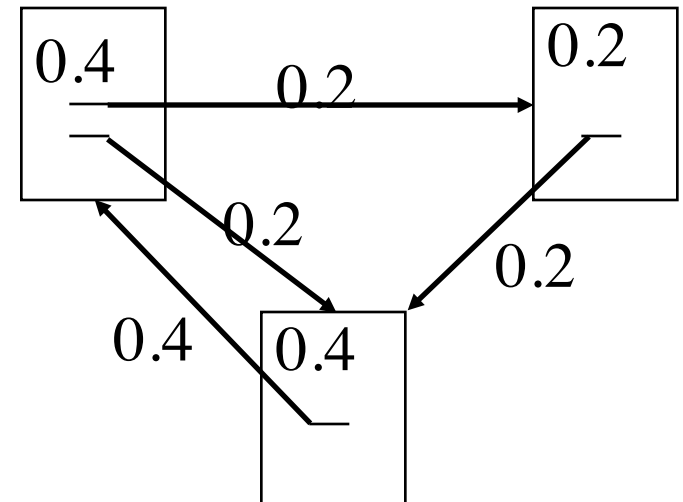


PageRank (one definition)

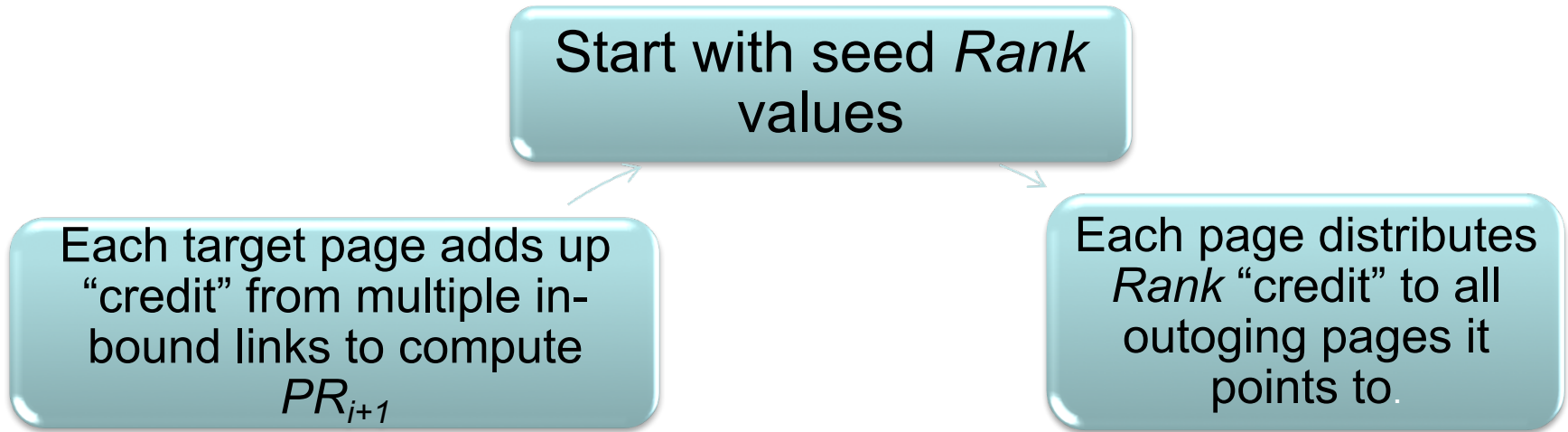
- Model page reputation on the web

$$PR(x) = (1 - d) + d \sum_{i=1}^n \frac{PR(t_i)}{C(t_i)}$$

- $i=1, n$ lists all parents of page x .
- $PR(x)$ is the page rank of each page.
- $C(t)$ is the out-degree of t .
- d is a damping factor .



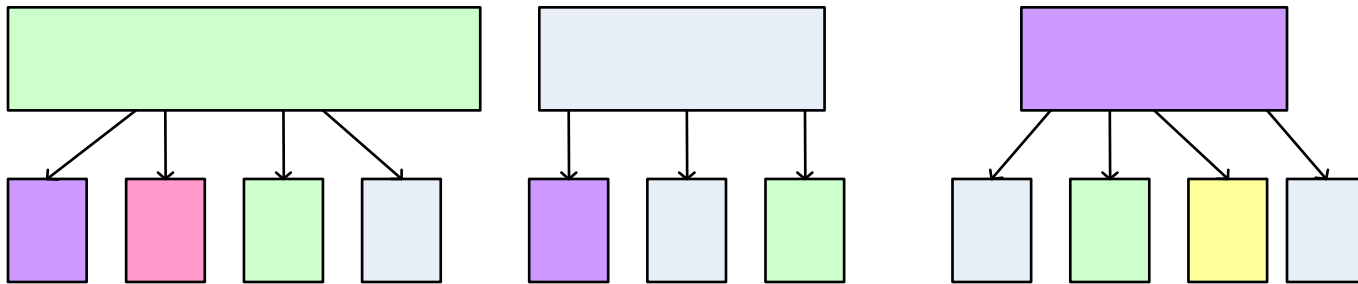
Computing PageRank Iteratively



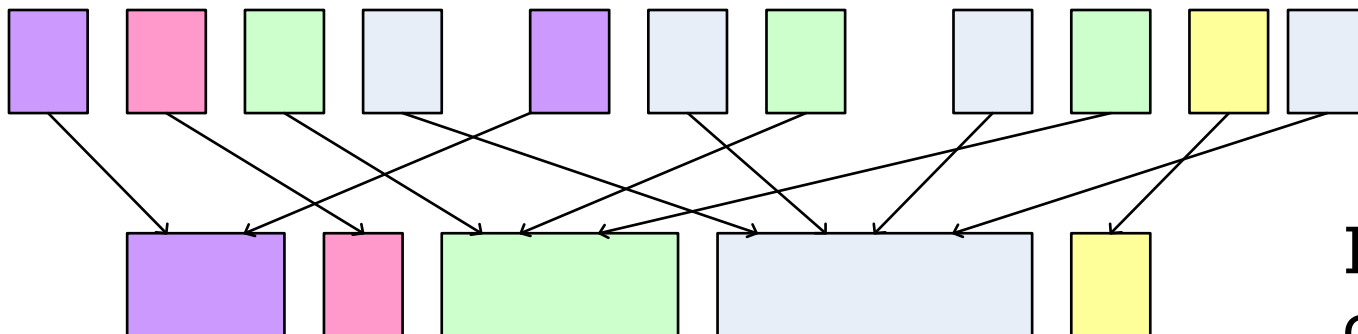
- Effects at each iteration is local. $i+1^{\text{th}}$ iteration depends only on i^{th} iteration
- At iteration i , PageRank for individual nodes can be computed independently

PageRank using MapReduce

Map: distribute PageRank "credit" to link targets



Reduce: gather up PageRank "credit" from multiple sources to compute new PageRank value

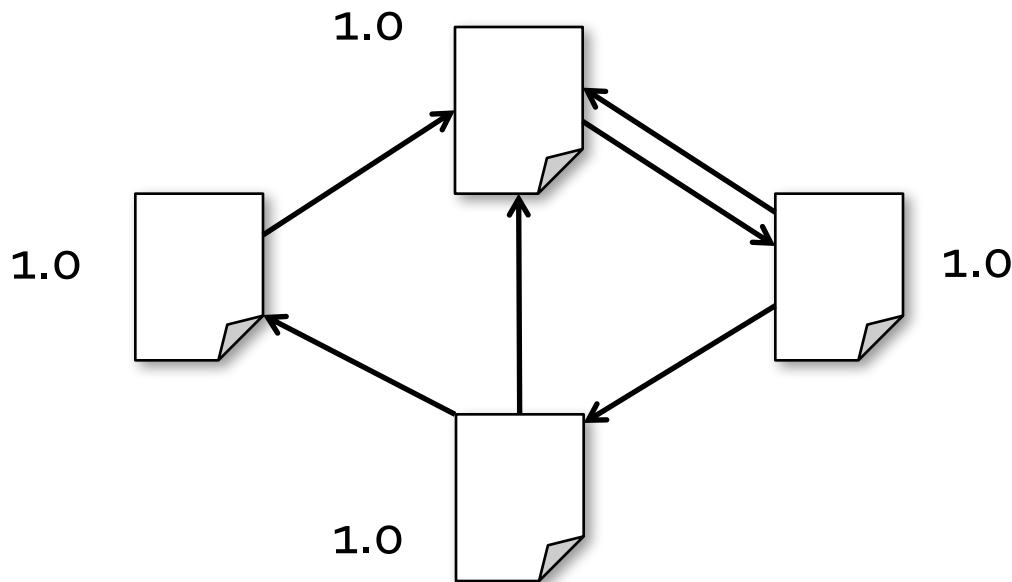


Source of Image: Lin 2008

Iterate until convergence

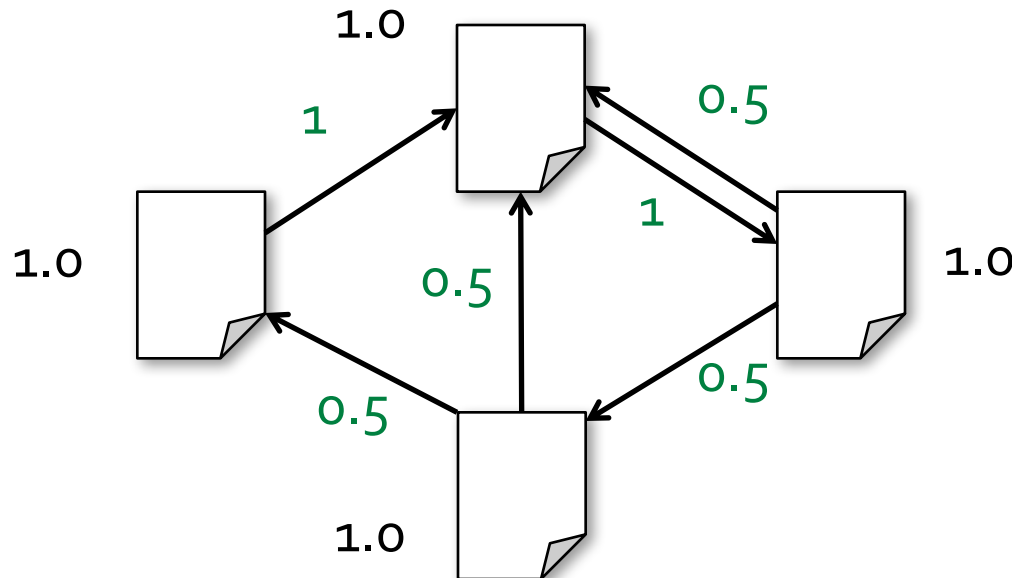
Algorithm demo

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



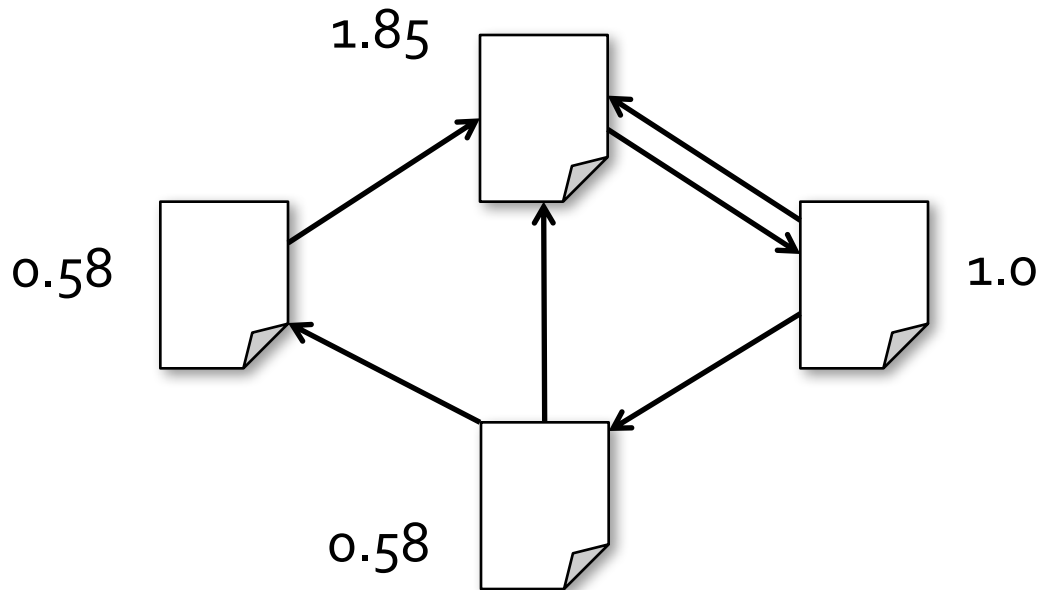
Algorithm

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



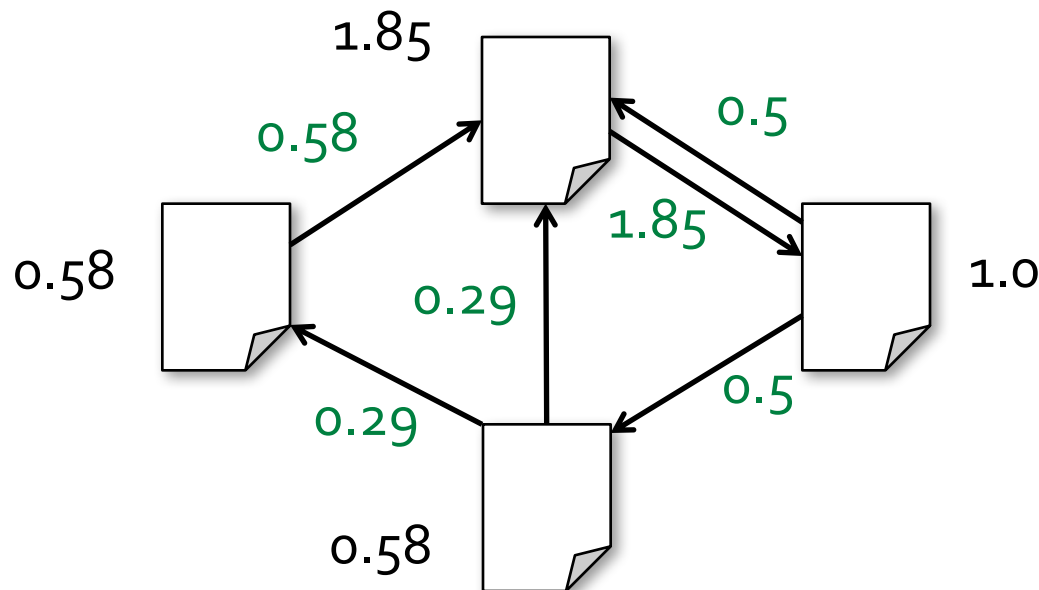
Algorithm

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



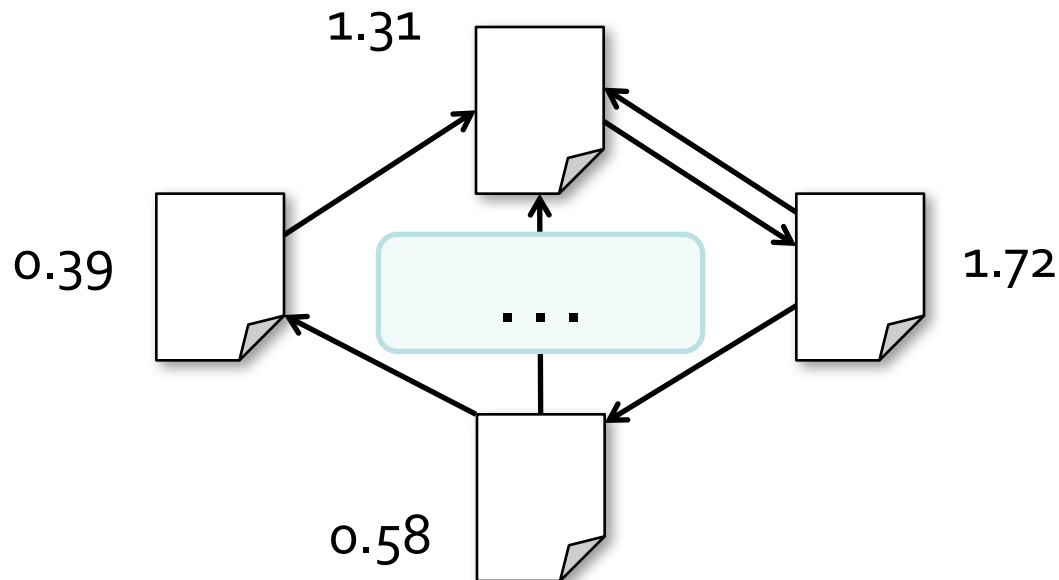
Algorithm

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



Algorithm

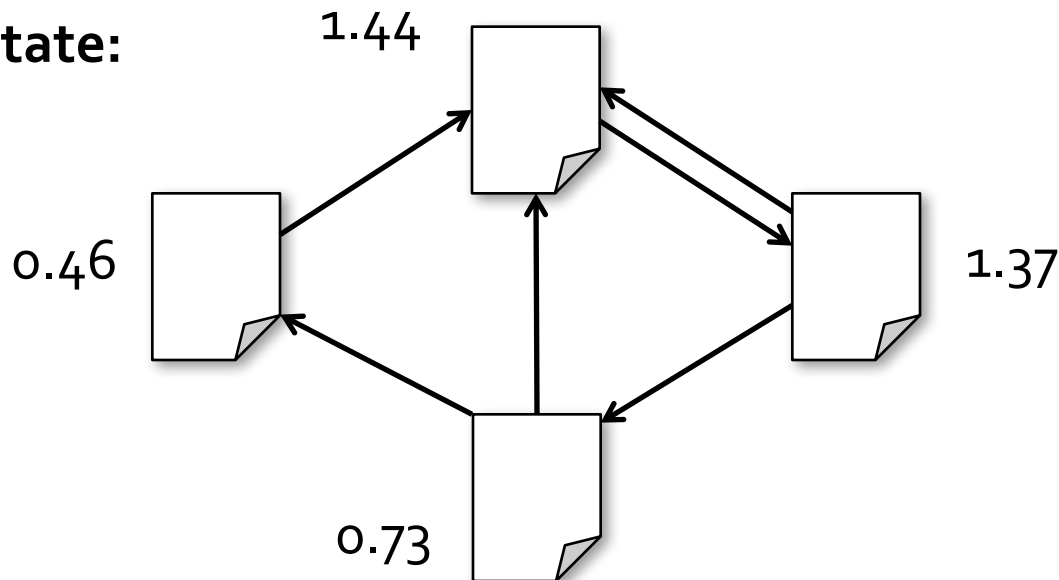
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{outdegree}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

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

Final state:



HW: SimplePageRank

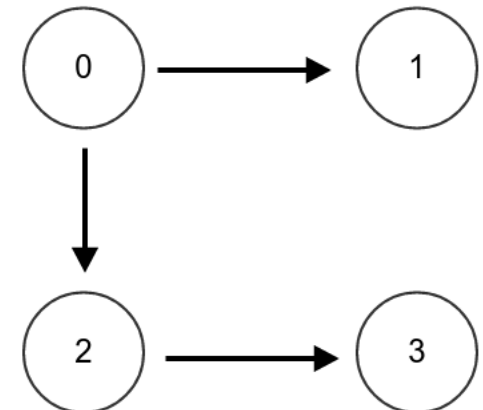
Random surfer model to describe the algorithm

- Stay on the page: $0.05 \cdot \text{weight}$
- Randomly follow a link: $0.85 / \text{out-going-Degree}$ to each child
 - If no children, give that portion to other nodes evenly.
- Randomly go to another page: 0.10
 - Meaning: contribute 10% of its weight to others. Others will evenly get that weight. Repeat for everybody. Since the sum of all weights is num-nodes, $10\% \cdot \text{num-nodes}$ divided by num-nodes is 0.1

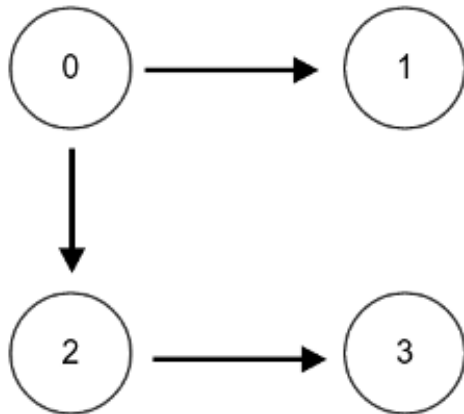
$$R(x) = 0.1 + 0.05 R(x) + \text{incoming-contributions}$$

Initial weight 1 for everybody

To/From	0	1	2	3	Random Factor	New Weight
0	0.05	0.283	0.0	0.283	0.10	0.716
1	0.425	0.05	0.0	0.283	0.10	0.858
2	0.425	0.283	0.05	0.283	0.10	1.141
3	0.00	0.283	0.85	0.05	0.10	1.283



Data structure in SimplePageRank



["# comment line", "0 1", "0 2", "2 3"]

iteration 0
[(0, (1.0, [1, 2])), (1, (1.0, [])),
(2, (1.0, [3])), (3, (1.0, []))]

iteration 1:
[(0, (0.72, [1, 2])), (1, (0.86, [])),
(2, (1.14, [3])), (3, (1.28, []))]

[(3, 1.28), (2, 1.14), (1, 0.86), (0, 0.72)]

