Spark for Big Data

Nicolas Courty
Master 1 info

Outline

- Apache Spark and Spark Resilient Distributed Datasets (RDD)
- Spark context, Transformation and Actions
- RDD Partitions
- Page rank (again)

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

Apache Spark and PySpark

- Apache Spark is written in Scala programming language that compiles the program code into byte code for the JVM for spark big data processing.

- The open source community has developed a wonderful utility for spark python big data processing known as PySpark.

- 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

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)

Transformation and Actions in Spark

- RDDs have ***actions***, which return values, and ***transformations***, which return pointers to new RDDs.

- RDDs' value is only updated once that RDD is computed as part of an action

- Lazy operations to build RDDs from other RDDs

- Lazy Evaluation: the ability to lazily evaluate code, postponing running a calculation until absolutely necessary.

Example: Mining Console Logs

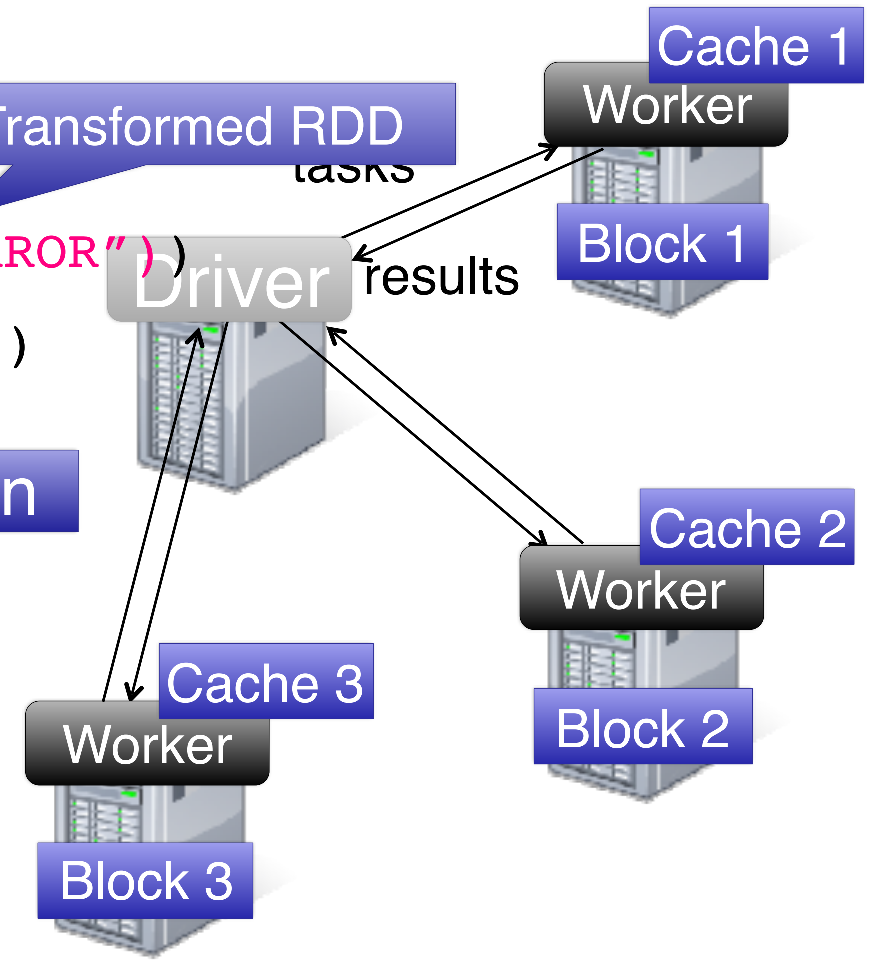
- Load error messages from a log into memory, then interactively search for patterns

```
lines = spark.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()
. . .
```

Base RDD

Transformed RDD

Action



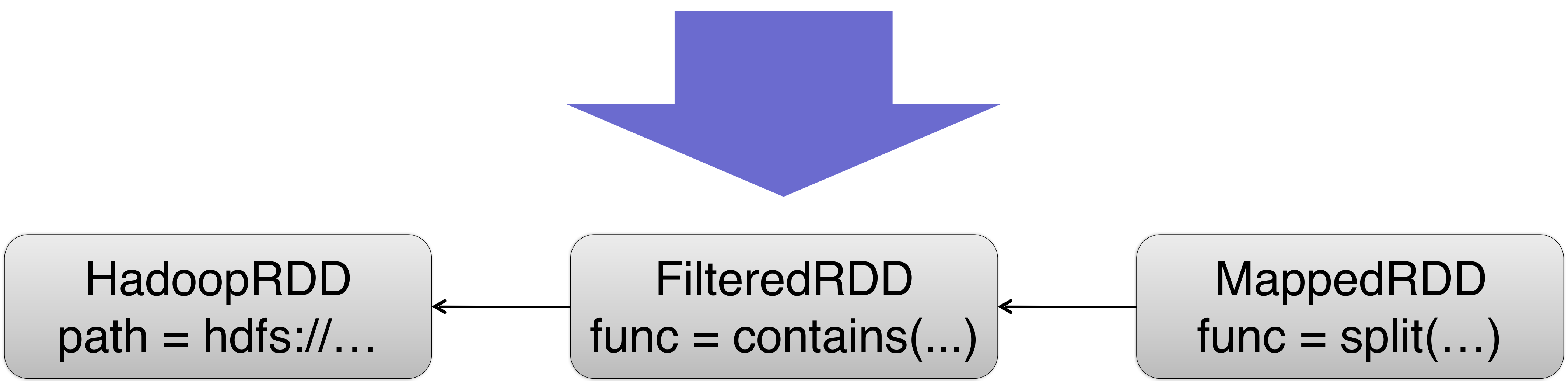
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

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])
```



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:

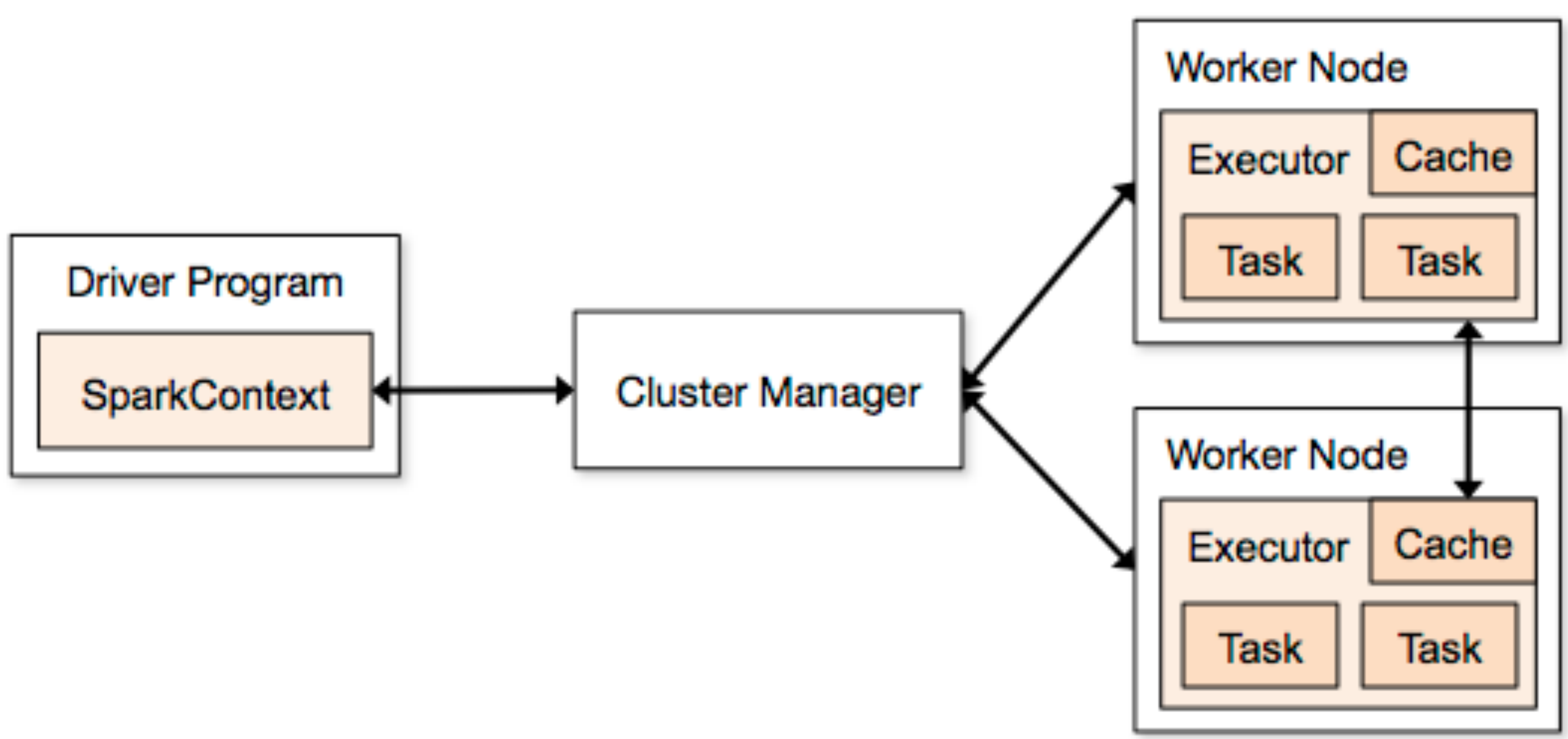
```
MASTER=local      ./spark-shell            # local, 1 thread
MASTER=local[2]   ./spark-shell            # local, 2 threads
MASTER=spark://host:port ./spark-shell     # Spark standalone cluster
```

Outline

- Apache Spark and Spark Resilient Distributed Datasets (RDD)
- Spark context, Transformation and Actions
- RDD Partitions
- Page rank (again)

SparkContext

- SparkContext is the object that:
  - manages the connection to the clusters in Spark
  - coordinates running processes on the clusters
  - connects to cluster managers, which manage the actual executors that run the specific computations



Create a SparkContext

Scala

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

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

- List of JARs with app code (to ship)
- Cluster URL, or local / local[N]
- App name
- Spark install path on cluster

Java

```java
import spark.api.java.JavaSparkContext;

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

Python

```python
from pyspark import SparkContext

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

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])
```

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)
```

Transformation and Actions

Spark Transformations

`map()`

`flatMap()`

`filter()`

`mapPartitions()`

Spark Actions

`reduceByKey()`

`collect()`

`count()`

`take()`

`takeOrdered()`

map() and **flatMap()**

- `map()`

map() transformation applies changes on each line of the RDD and returns the transformed RDD as iterable of iterables i.e. each line is equivalent to a iterable and the entire RDD is itself a list

- `flatMap()`

This transformation apply changes to each line same as map but the return is not a iterable of iterables but it is only an iterable holding entire RDD contents.

(disgression) Iterables, generator in Python

- A generator is an on the fly list

```
  mygenerator = (x*x for x in range(3))
   for i in mygenerator:
    print(i)
0
1
4
```

- Cannot perform `mygenerator = (x*x for x in range(3))` for a second time !
- **`Yield`** is the key word for defining function as generator

```
>>> def createGenerator():
...    mylist = range(3)
...    for i in mylist:
...        yield i*i
...
>>> mygenerator = createGenerator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
1
4
```

(disgression) Iterables, generator in Python

- An iterable is an object where you can read its items one by one

```
mylist = [1, 2, 3]
for i in mylist:
    print(i)
1
2
3
```

- Same for a a list comprehension

```
mylist = [x*x for x in range(3)]
for i in mylist:
    print(i)
0
1
4
```

`map()` and **`flatMap()`** examples

- `lines.take(2)`

['#good d#ay #','#good #weather']

- `words=lines.map(lambda lines: lines.split(' '))`

[['#good', 'd#ay', '#'],
['#good', '#weather']]

- `words=lines.flatMap(lambda lines: lines.split(' '))`

['#good', 'd#ay', '#', '#good', '#weather']

Instead of using an anonymous function (with the lambda keyword in Python), we can also use named function

anonymous function is easier for simple use

Filter()

- **Filter()** transformation is used to reduce the old RDD based on some condition.

- How to filter out hashtags from words

```
hashtags = words.filter(lambda word: "#" in word)
```

['#good', 'd#ay', '#', '#good', '#weather']

which is wrong.

```
hashtags = words.filter(lambda word: 
word.startswith("#")).filter(lambda word: word != "#")
```

['#good', '#good', '#weather']

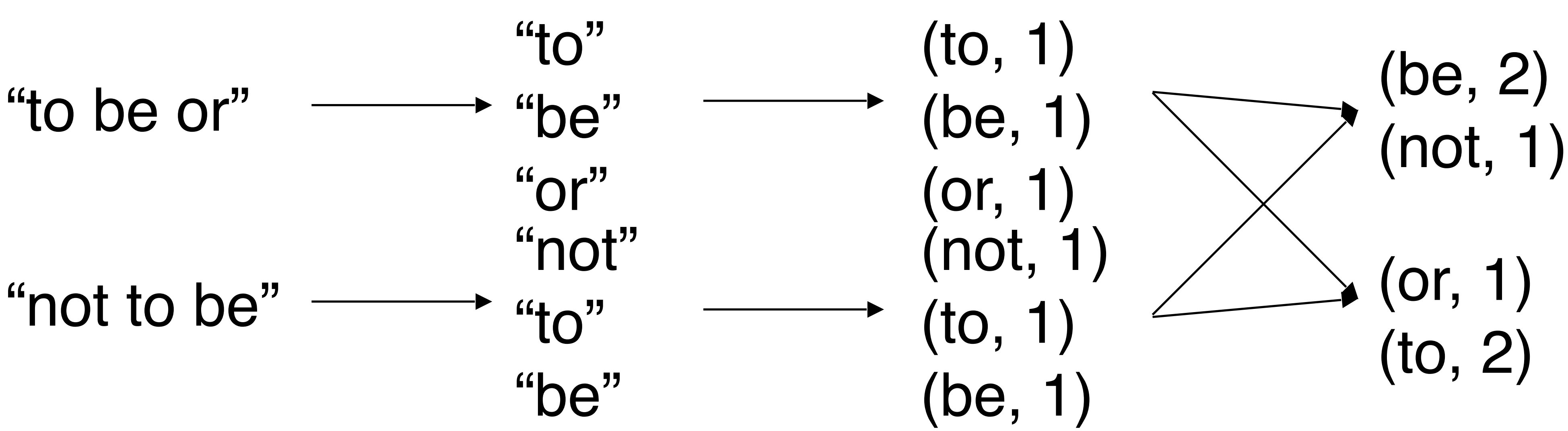
reduceByKey()

- reduceByKey(f) combines tuples with the same key using the function we specify f.

```
hashtagsNum = hashtags.map(lambda word: (word, 1))
[(‘#good’,1), (‘#good’, 1), (‘#weather’, 1)]

hashtagsCount = hashtagsNum.reduceByKey(lambda a,b: a+b)
or
hashtagsCount = hashtagsNum.reduceByKey(add)
[(‘#good’,2), (‘#weather’, 1)]
```

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!)

Outline

- Apache Spark and Spark Resilient Distributed Datasets (RDD)
- Spark context, Transformation and Actions
- RDD Partitions
- Page rank (again)

RDD Partitions

- Map and Reduce operations can be effectively applied in parallel in apache spark by dividing the data into multiple partitions.

- A copy of each partition within an RDD is distributed across several workers running on different nodes of a cluster so that in case of failure of a single worker the RDD still remains available.

`mapPartitions()`

- **`mapPartitions(func)`** transformation is similar to **`map()`**, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

Example-1: Sum Each Partition

```
def f(iterator):
    for x in iterator:
        print(x)
        print "==="
def adder(iterator):
    yield sum(iterator)
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
rdd = sc.parallelize(numbers, 3)
rdd.foreachPartition(f)
1
2
3
===
7
8
9
10
===
4
5
6
===
rdd.mapPartitions(adder).collect()
[6, 15, 34]
```

Example-2: Find Minimum and Maximum

```
def minmax(iterator):
    firsttime = 0
    min = 0;
    max = 0;
    for x in iterator:
        if (firsttime == 0):
            min = x;
            max = x;
            firsttime = 1
        else:
            if x > max:
                max = x
            if x < min:
                min = x
    return (min, max)

data = [10, 20, 3, 4, 5, 2, 2, 20, 20, 10]
print minmax(data)
[2, 20]
```

```
def f(iterator):
    for x in iterator:
        print(x)
    print "==="
data = [10, 20, 3, 4, 5, 2, 2, 20, 20, 10]
rdd = sc.parallelize(data, 3)
rdd.foreachPartition(f)
10
20
3
===
4
5
2
===
2
20
20
10
===
minmaxlist = rdd.mapPartitions(minmax).collect()
minmaxlist
[3, 20, 2, 5, 2, 20]
min(minmaxlist)
2
max(minmaxlist)
20
```

Outline

- Apache Spark and Spark Resilient Distributed Datasets (RDD)
- Spark context, Transformation and Actions
- RDD Partitions
- Page rank (again)

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

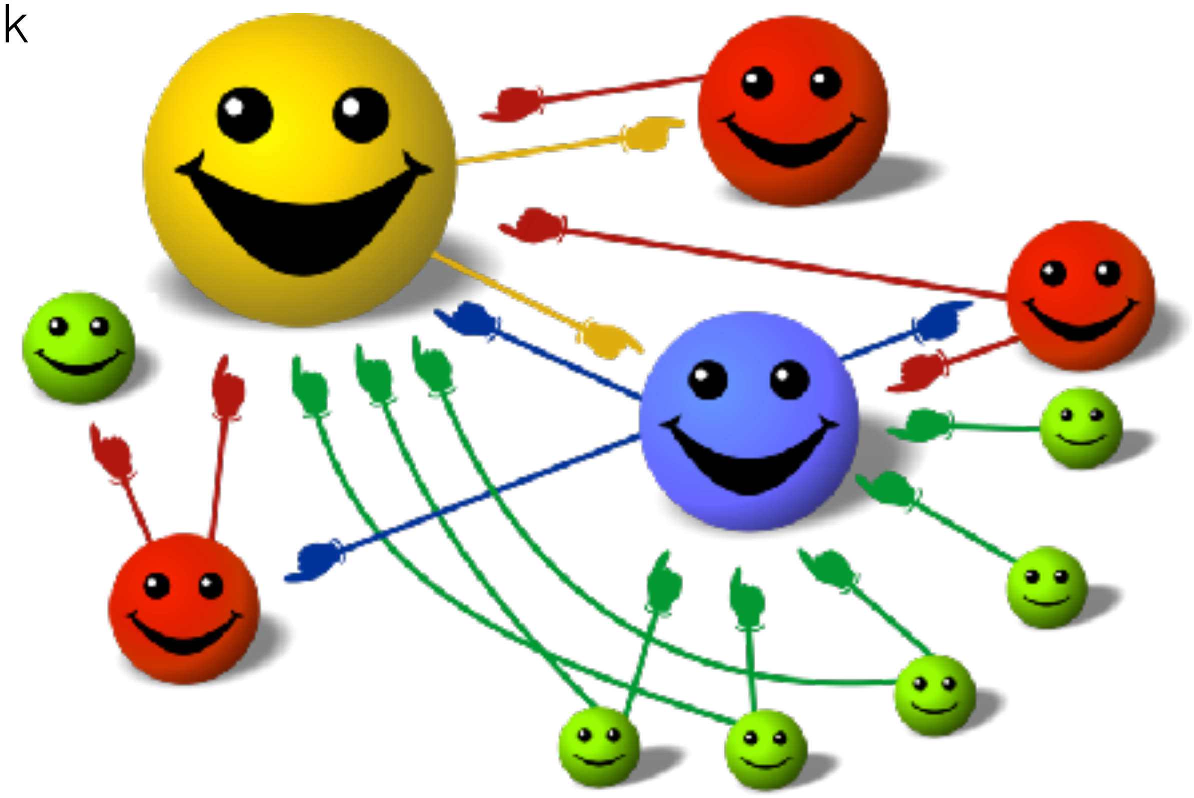
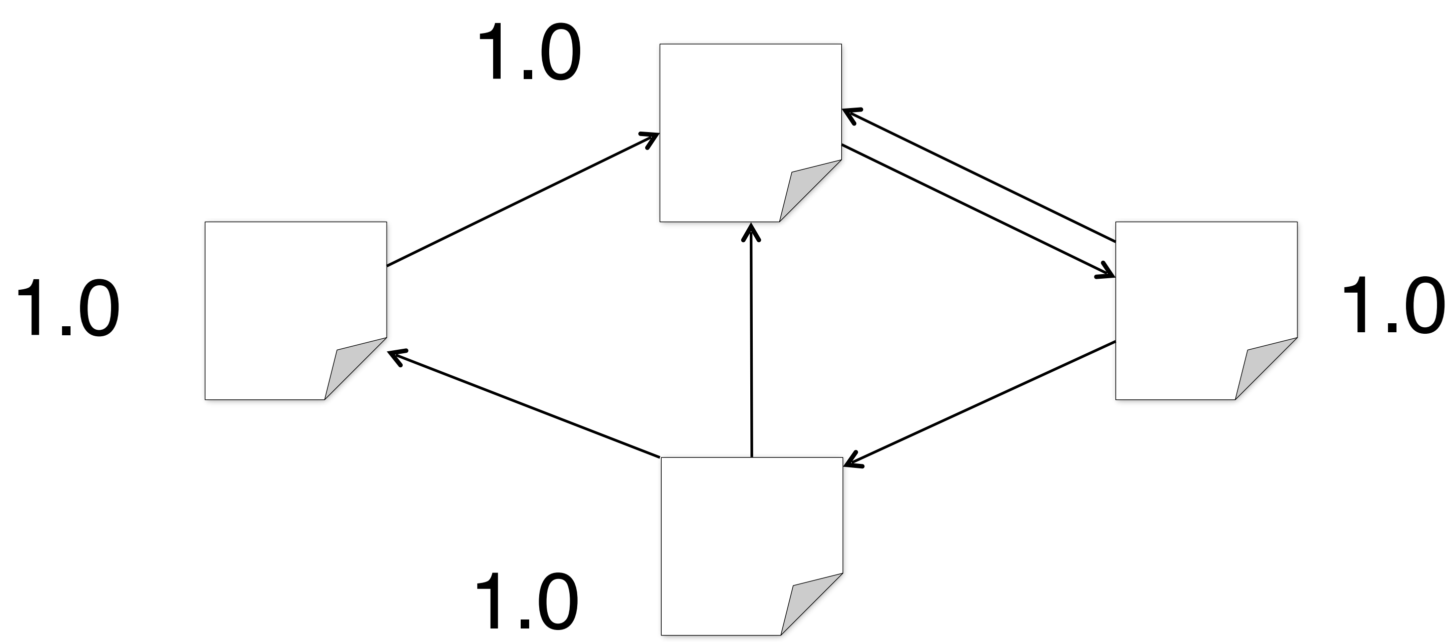


Image: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

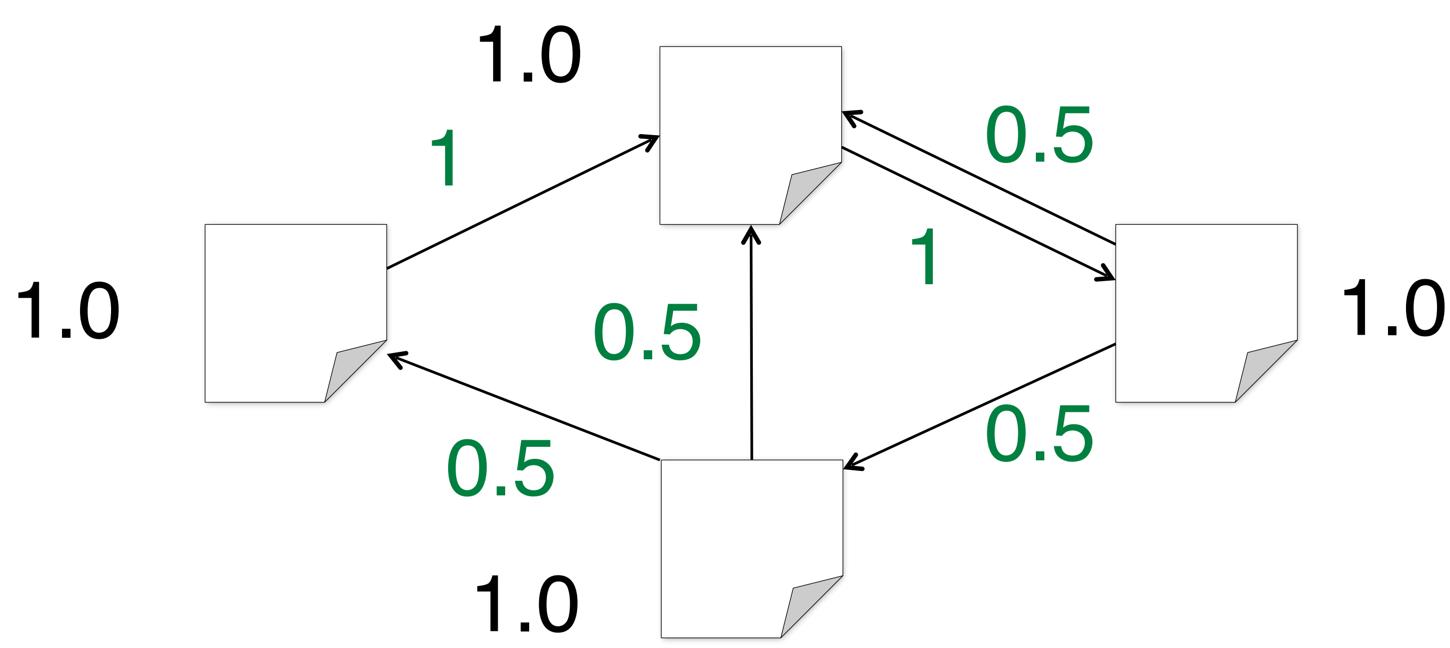
Algorithm

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



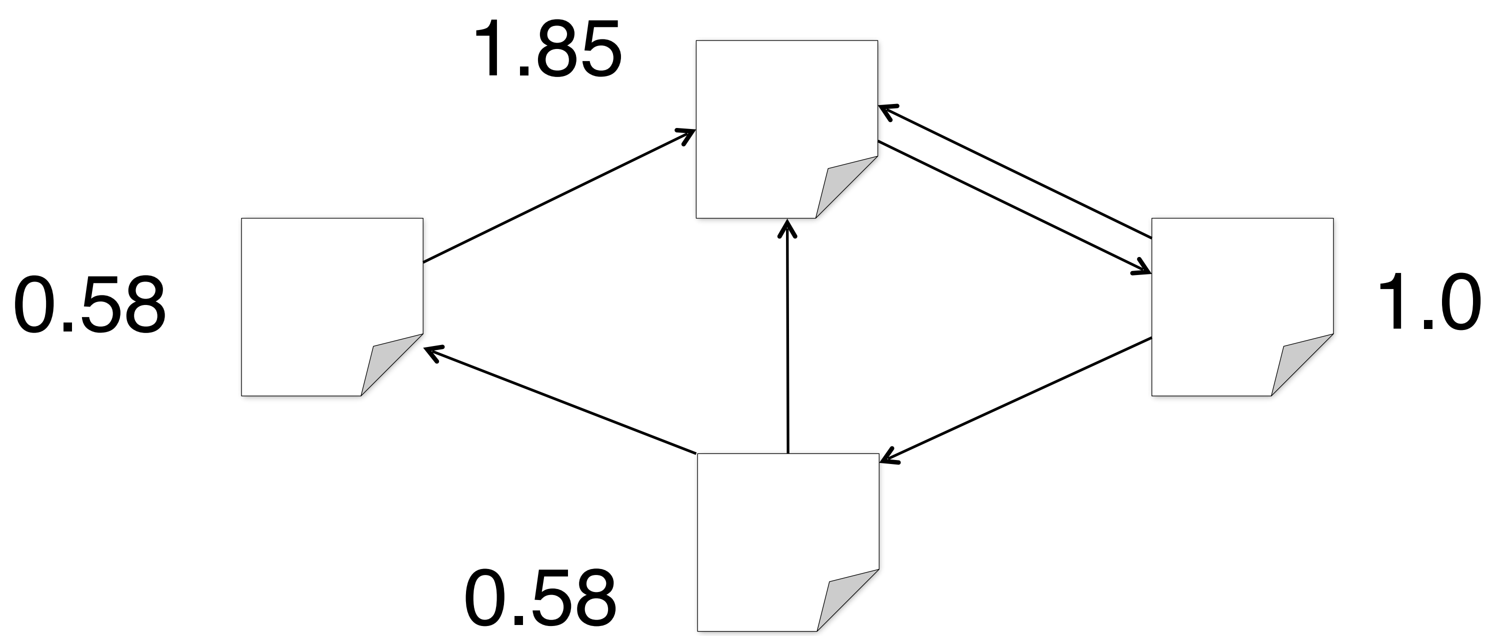
Algorithm

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 $0.15 + 0.85 \times contribs$



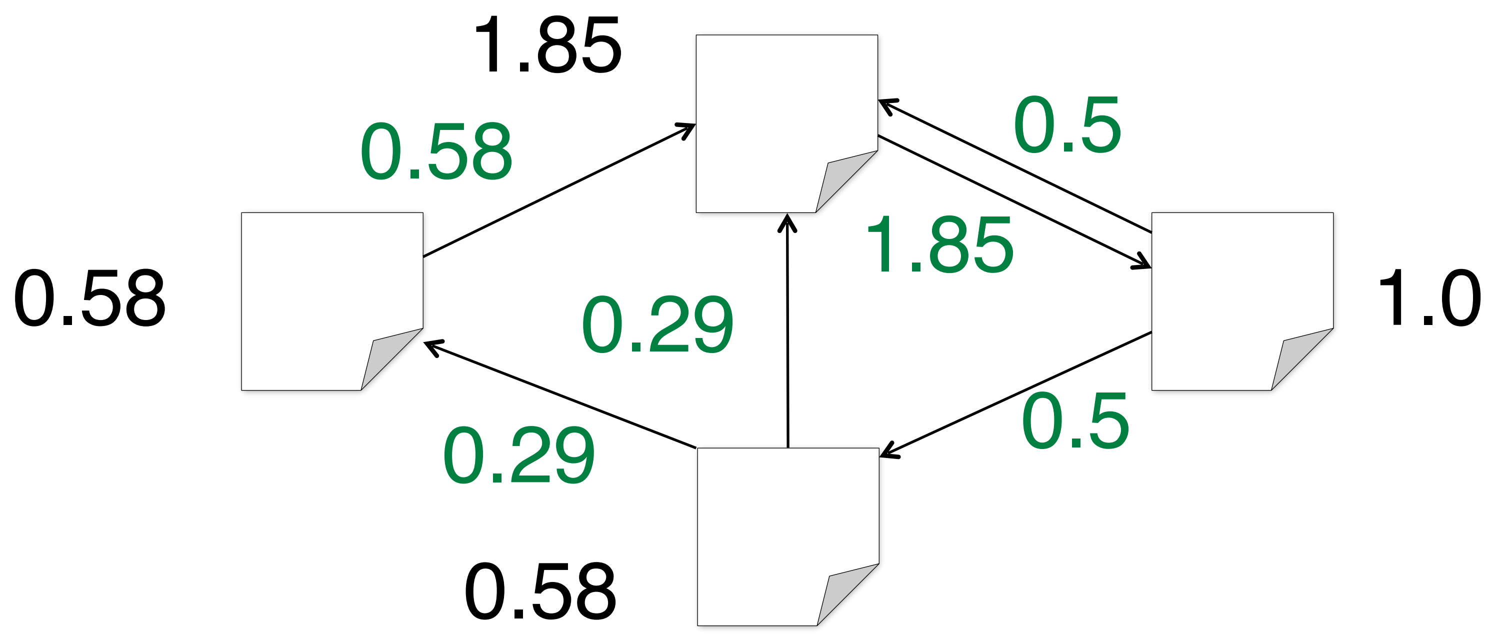
Algorithm

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 $0.15 + 0.85 \times contribs$



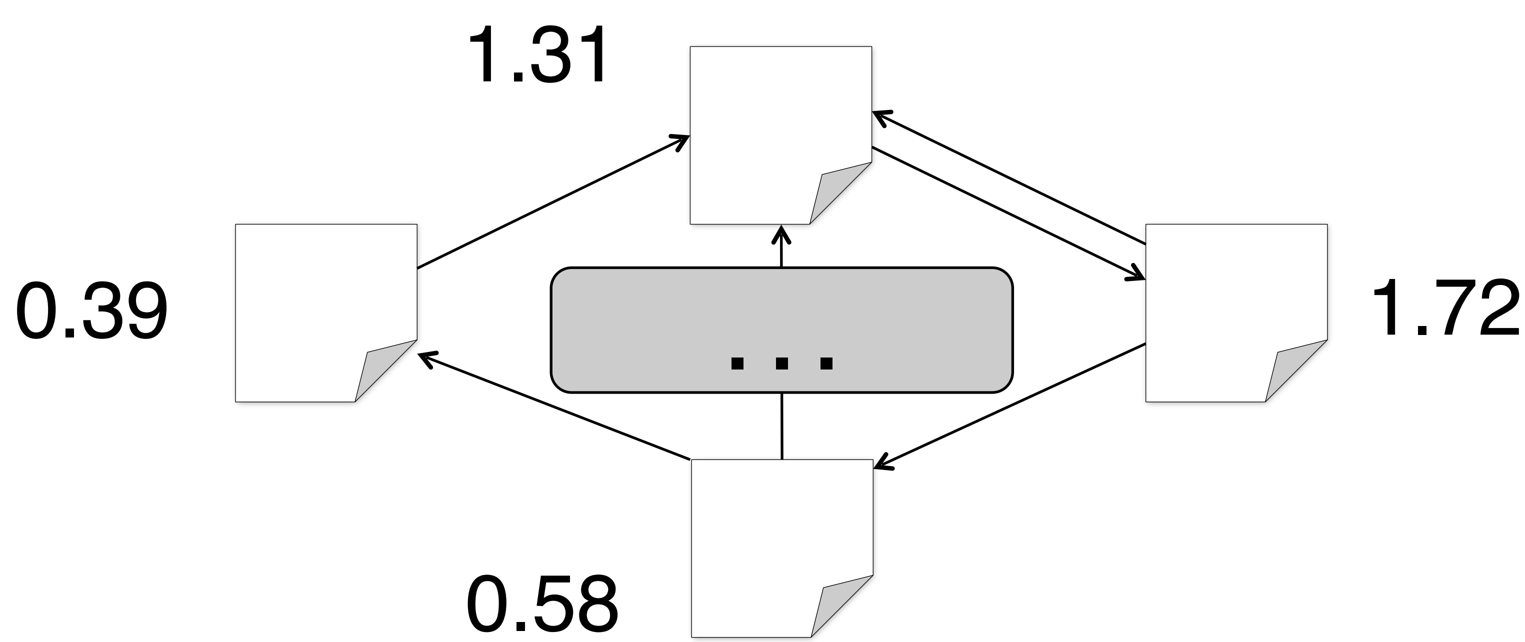
Algorithm

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



Algorithm

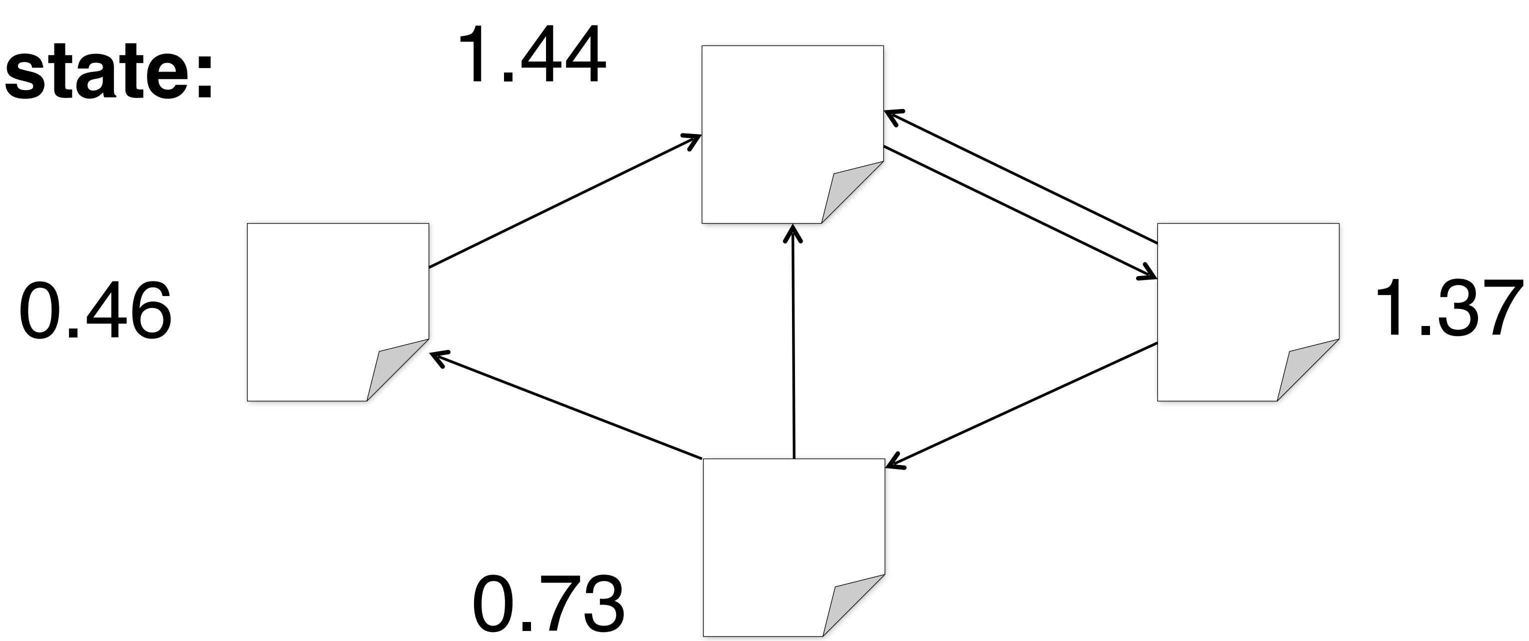
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



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

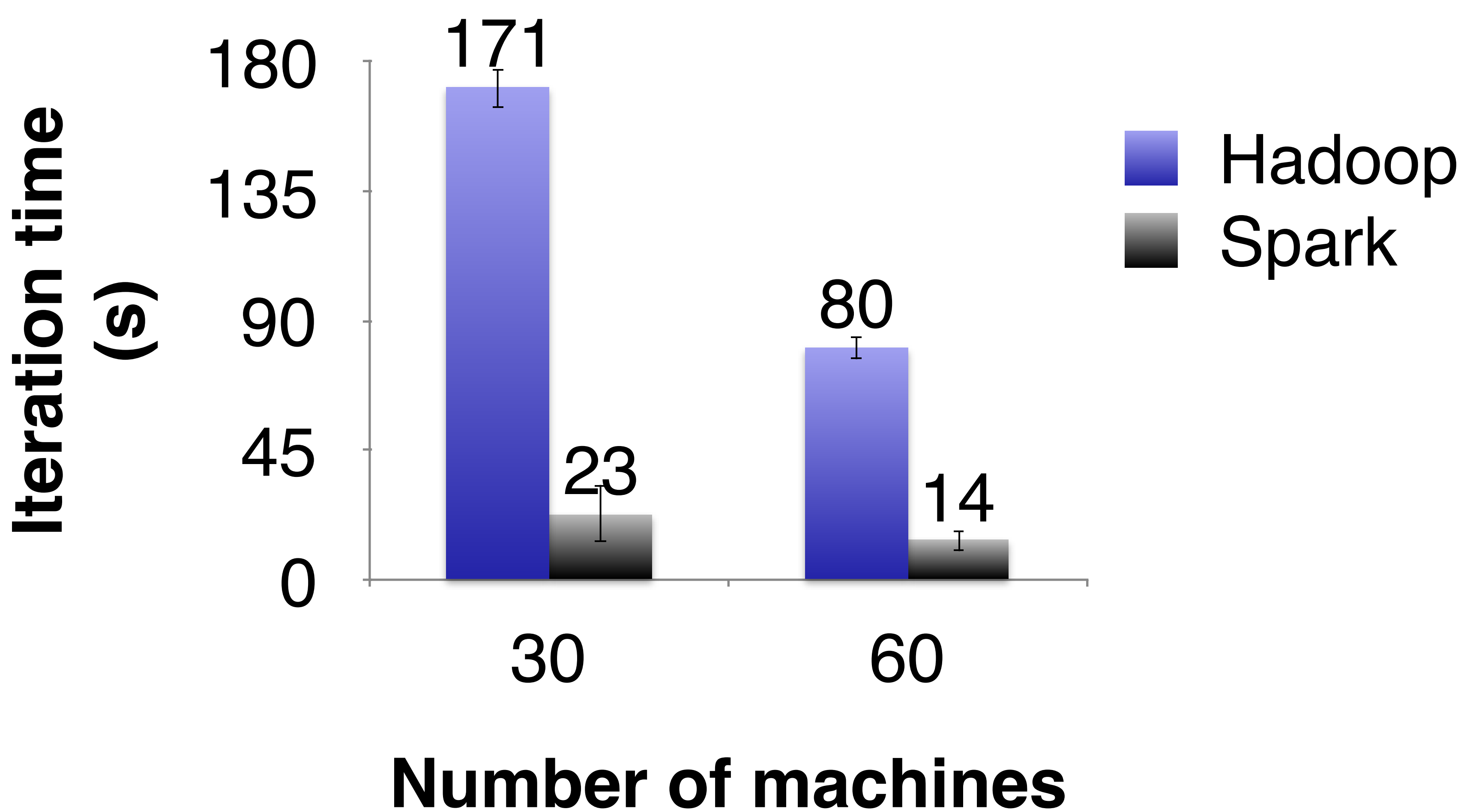
Final state:



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