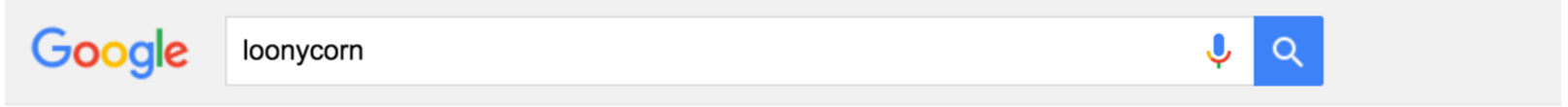
PAGERANK

Have you ever wondered how Google Search works?



All Videos News Images Maps More Search tools

About 2,790 results (0.27 seconds)

Loony Corn | A 4-person team;ex-Google; Stanford, IIM Ahme...
https://www.udemy.com/user/janani-ravi-2/
Loonycorn is us, Janani Ravi, Vitthal Srinivasan, Swetha Kolalapudi and Navdeep Singh. Between the four of us, we have studied at Stanford, IIM Ahmedabad, ...

From 0 to 1: Raspberry Pi and the Internet of Things | Udemy
https://www.udemy.com/from-0-to-1-raspberry-pi/
★★★★½ Rating: 4.3 - 44 votes
May 5, 2016 - **Loonycorn** is us, Janani Ravi, Vitthal Srinivasan, Swetha Kolalapudi and Navdeep Singh. Between the four of us, we have studied at Stanford, ...

Loonycorn
www.**loonycorn**.com/
From 0 to 1: Machine Learning, NLP & Python-Cut to the Chase. From 0 to 1: Design Patterns - 24 That Matter - In Java. From 0 to 1: Raspberry Pi and the ...

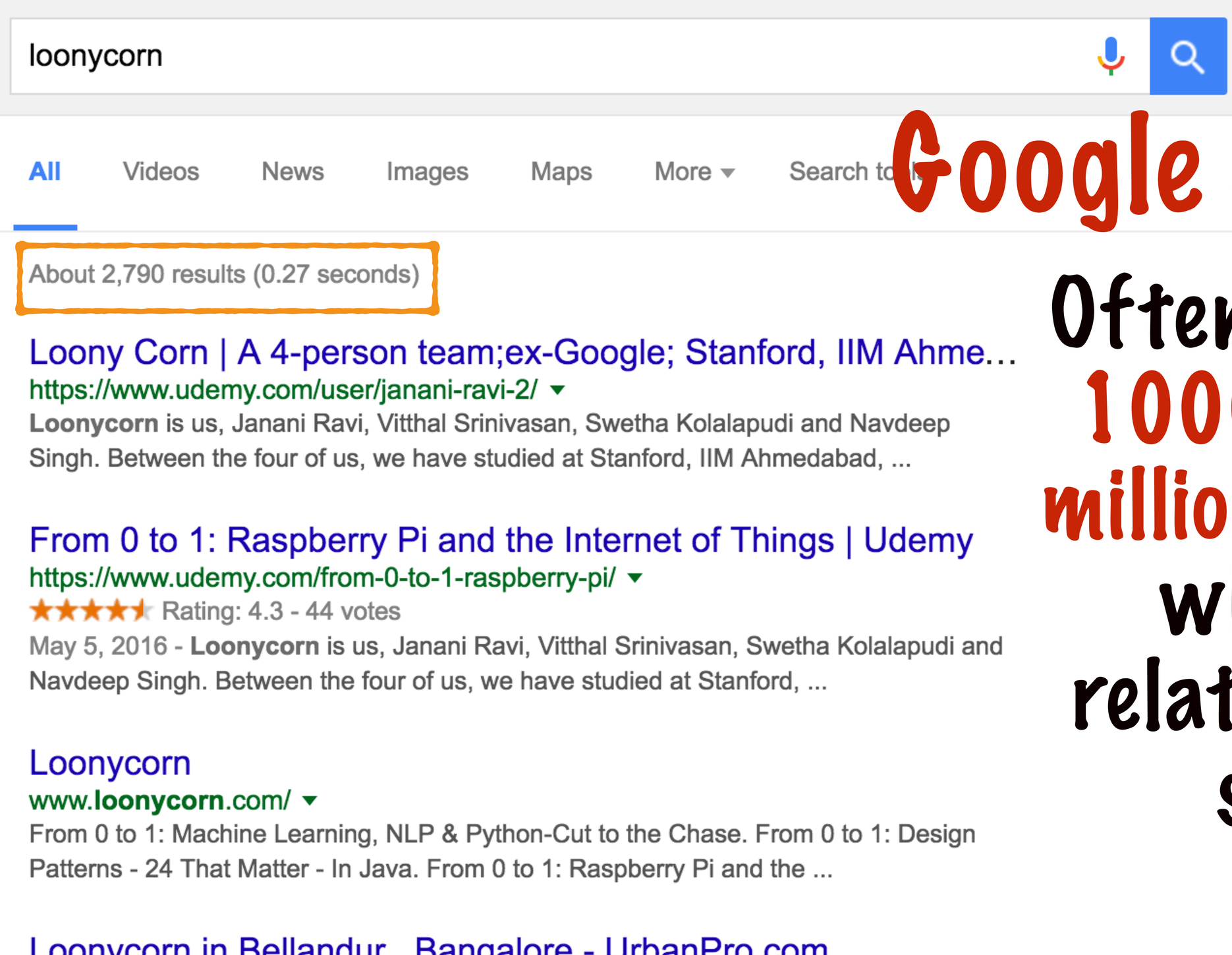
Loonycorn in Bellandur , Bangalore - UrbanPro.com
https://www.urbanpro.com/bangalore/**loonycorn**-bellandur/4069590
★★★★★ Rating: 5 - 2 reviews
Find contact number, address, user reviews, courses, classes details and trainers of **Loonycorn** in Bellandur , Bangalore.

Janani Ravi | LinkedIn
https://in.linkedin.com/in/jananiravi
Bengaluru Area, India - Co-founder at Loonycorn - Loonycorn
Main content starts below. Janani Ravi. Co-founder at **Loonycorn**. Location: Bengaluru Area, India; Industry: Computer Software. Current. **Loonycorn**. Previous.

Loonycorn - StackSkills
stackskills.com/courses/author/24669
Loonycorn. A 4-person team;ex-Google; Stanford, IIM-A, IIT. Learn By Example: Hadoop, MapReduce for Big Data problems. A hands-on workout in Hadoop, ...

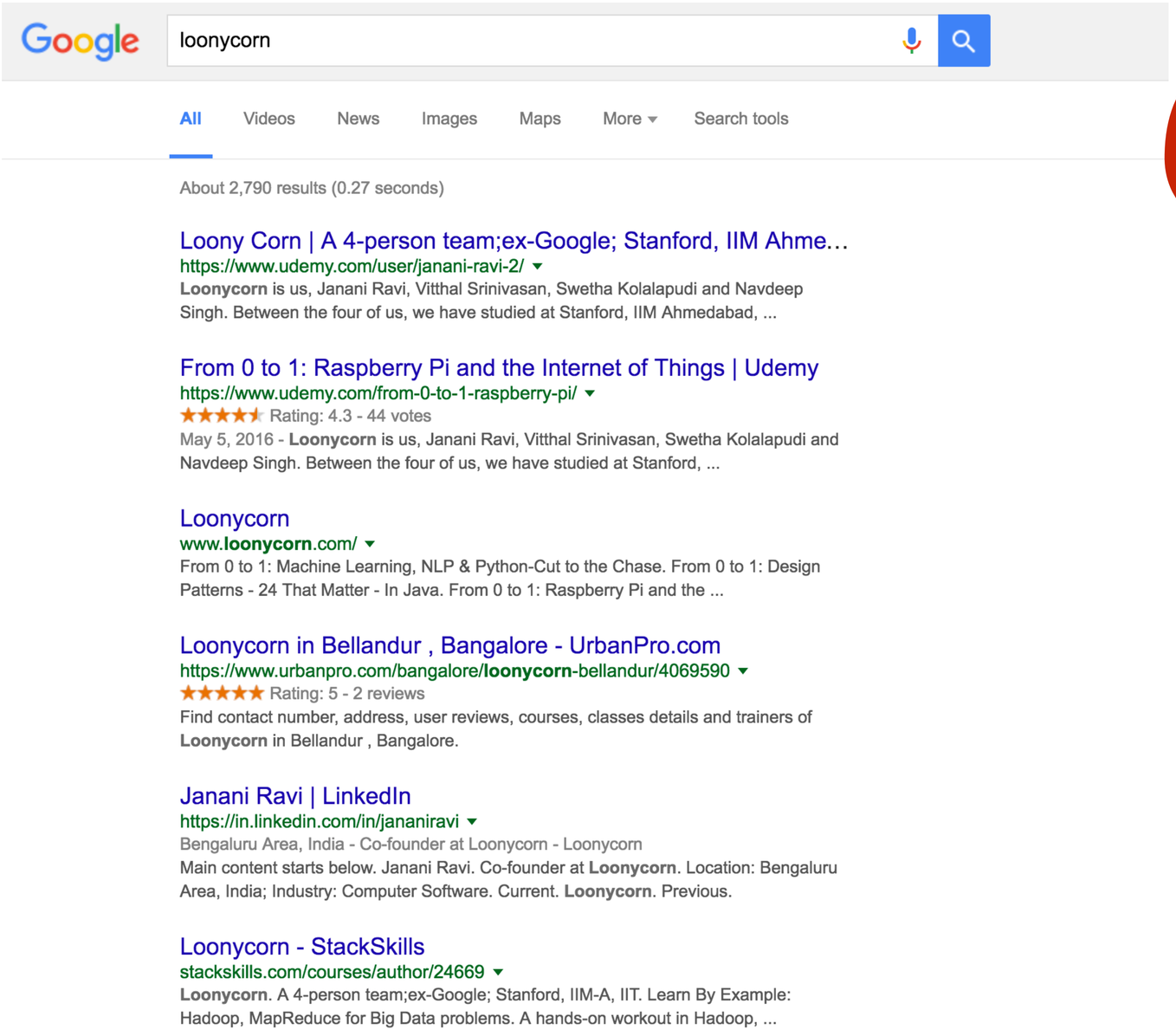
Google Search

When you search for something on Google, it returns a bunch of results



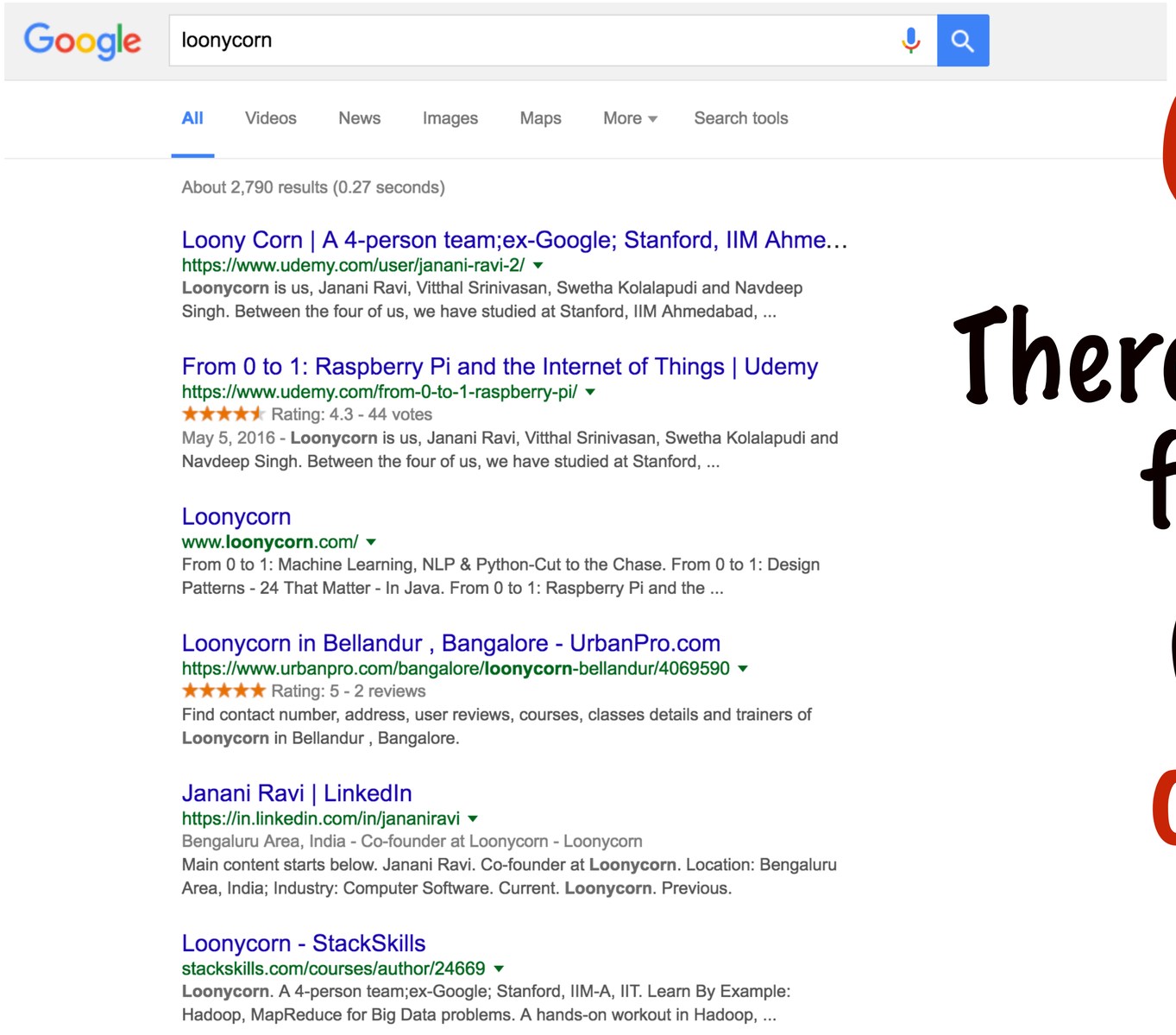
Google Search

Often, there are 1000s or even millions of pages which are related to your search



Google Search

How does Google decide which results should be shown first?



Google Search

There are many factors

One of them is the **overall importance** of a webpage

Google Search

How do we decide what the overall importance of a webpage is?

We let other webpages tell us!

Google Search

We let other webpages tell us!

The more links there are to a page

.. the more important it is

Google Search

We let other webpages tell us!

If the pages that link to it are themselves important

The page's importance increases further

Google Search

If the pages that link to it are themselves important

A page can be assumed to be important if the New York Times references it

Google Search

Google has an algorithm to capture exactly this idea

PageRank

PageRank

(Named after Larry Page)

PageRank plays a significant role in deciding the ranking of search results

PageRank

(Named after Larry Page)

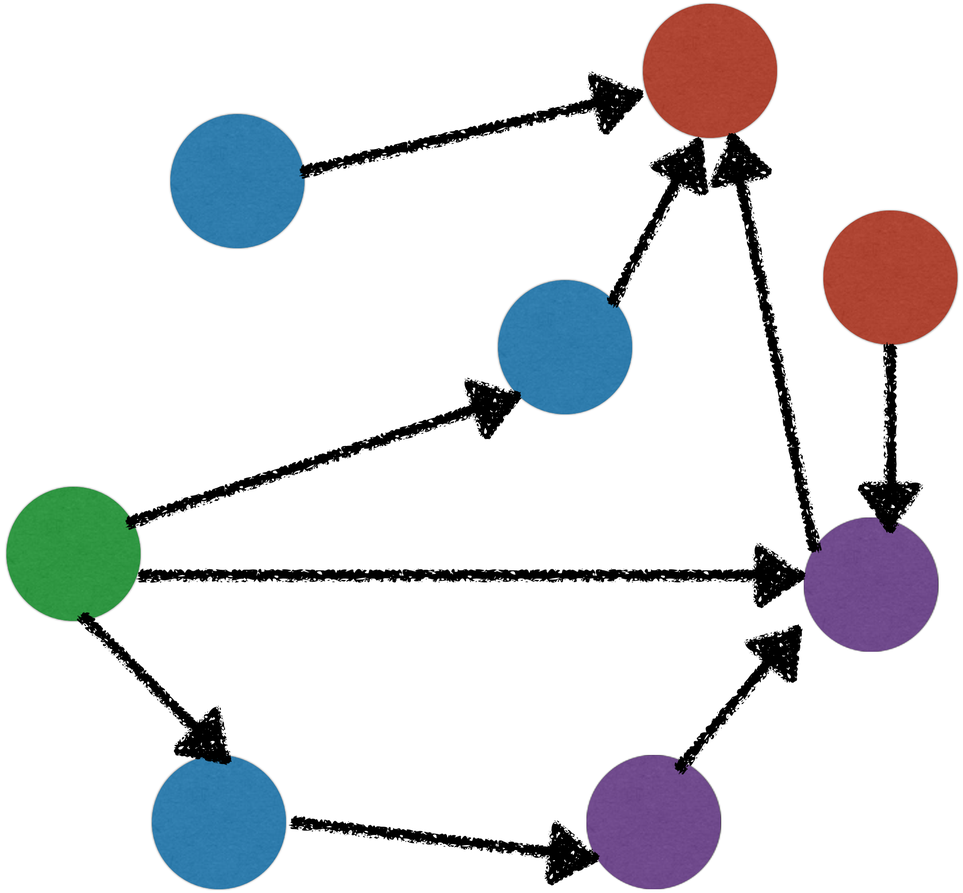
Higher the PageRank higher the page in the search results

PageRank

PageRank is best
understood using graphs

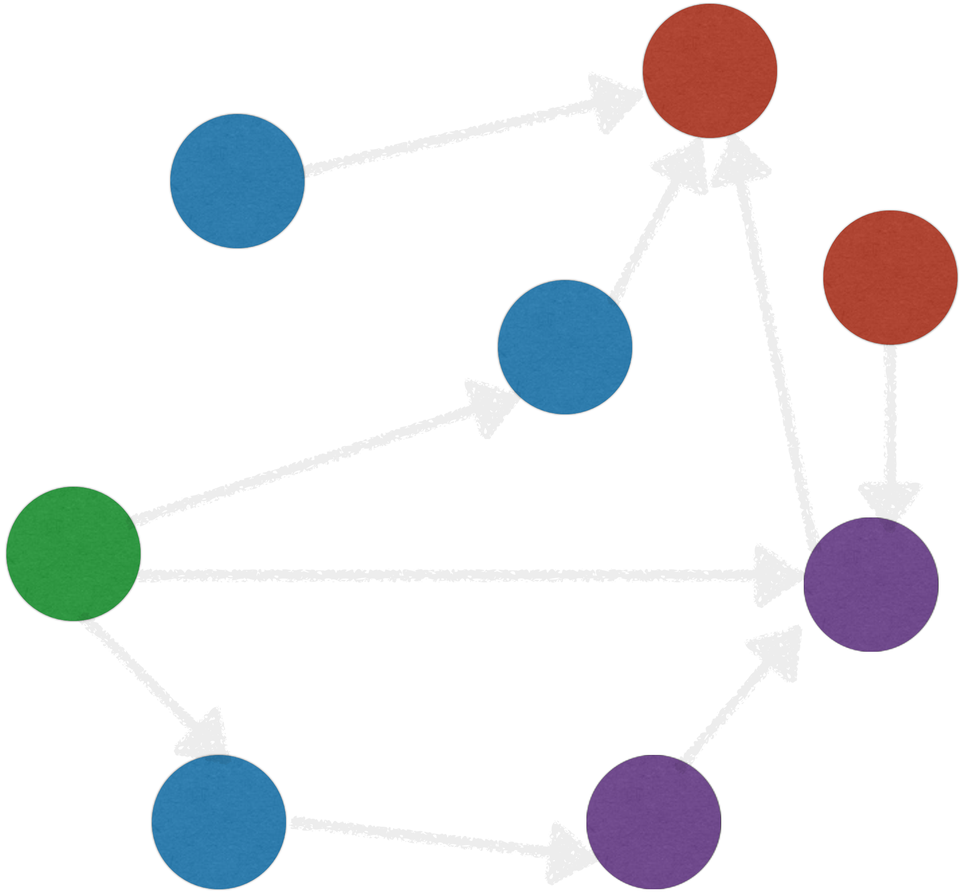
PageRank

Here is a graph representing links between webpages



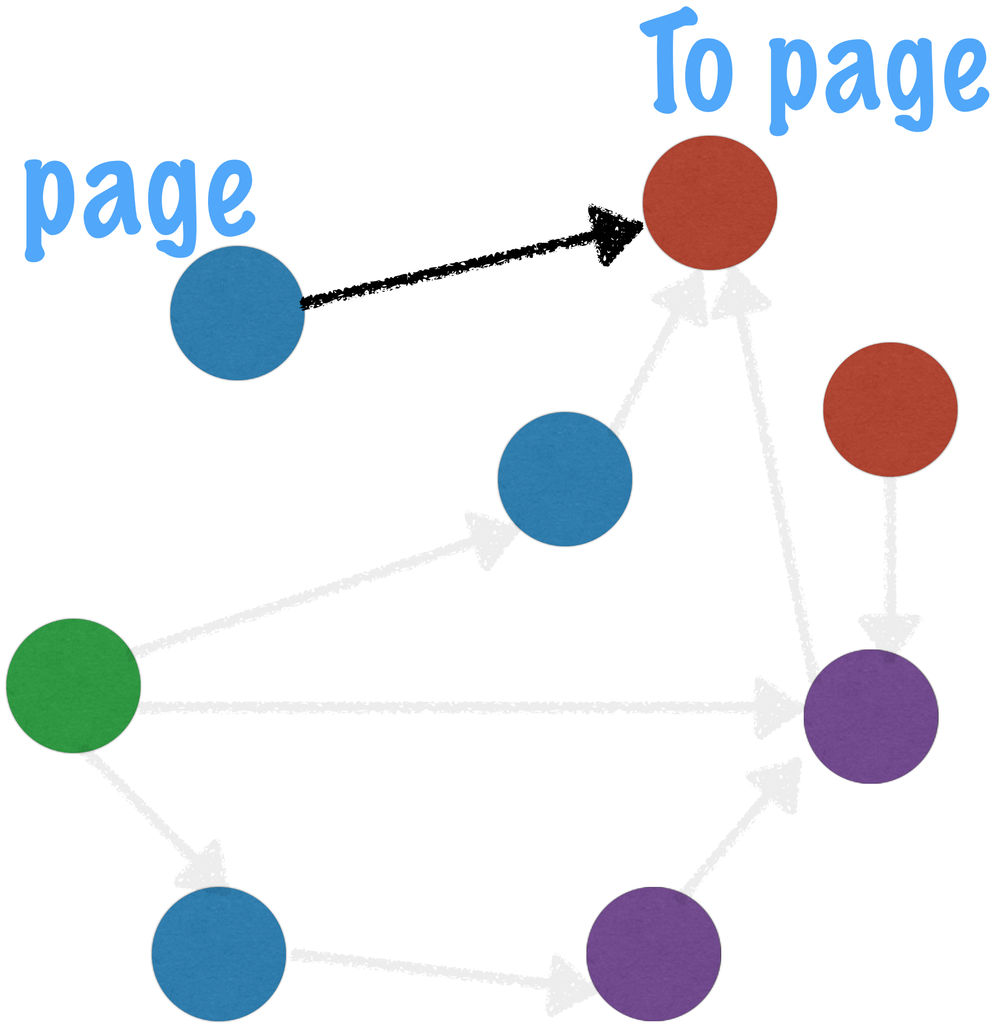
PageRank

Nodes represent
webpages



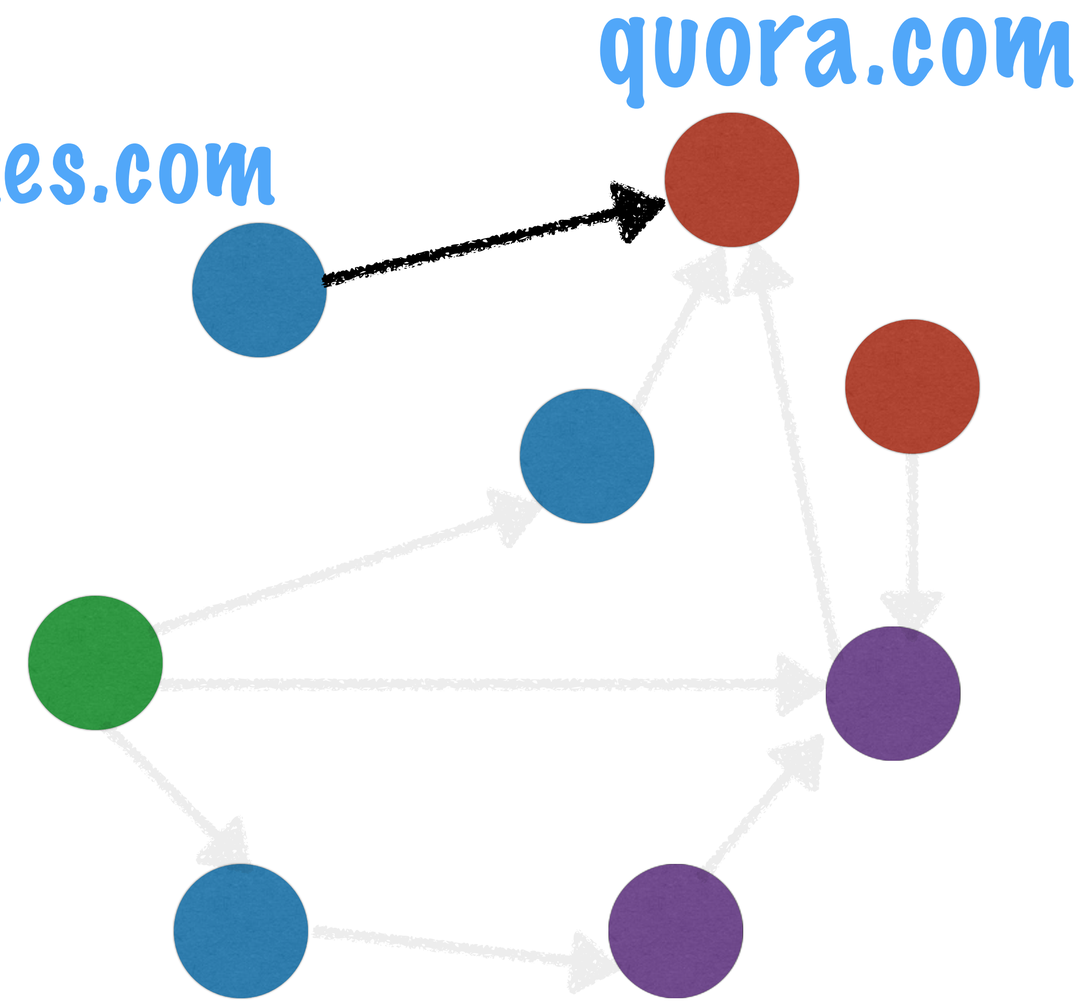
PageRank

An edge represents a link from one webpage to another



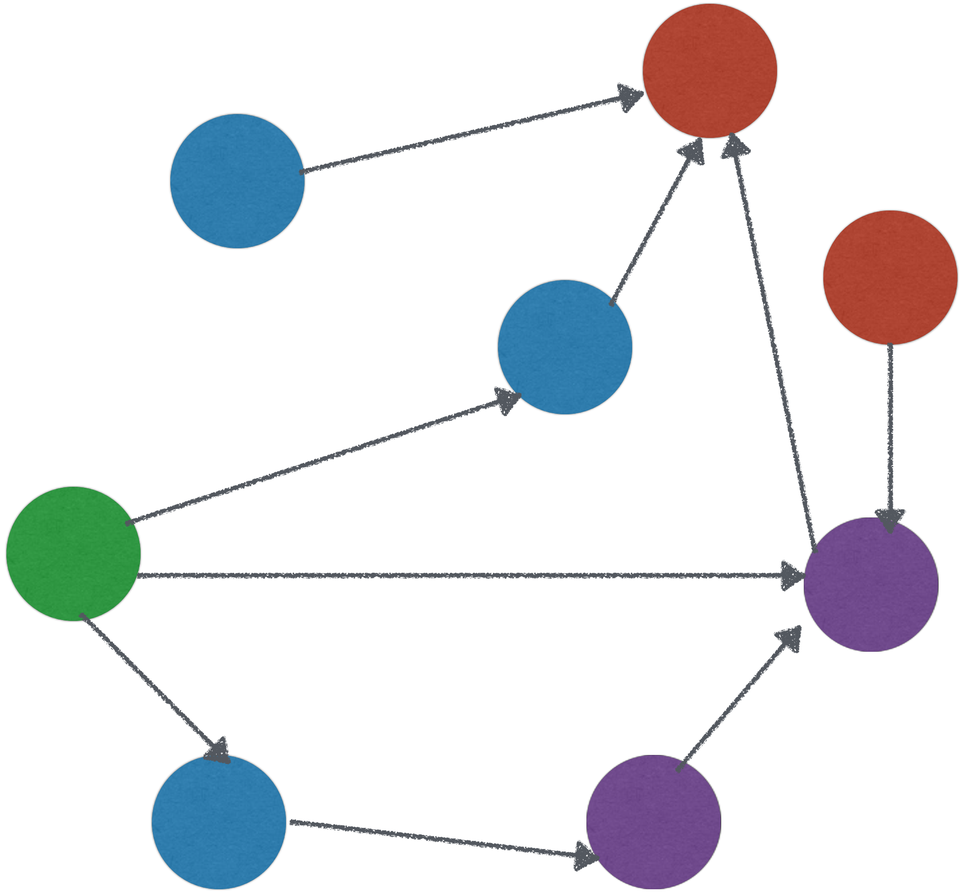
PageRank

An article in the NYT about Quora will have a reference of this type



PageRank

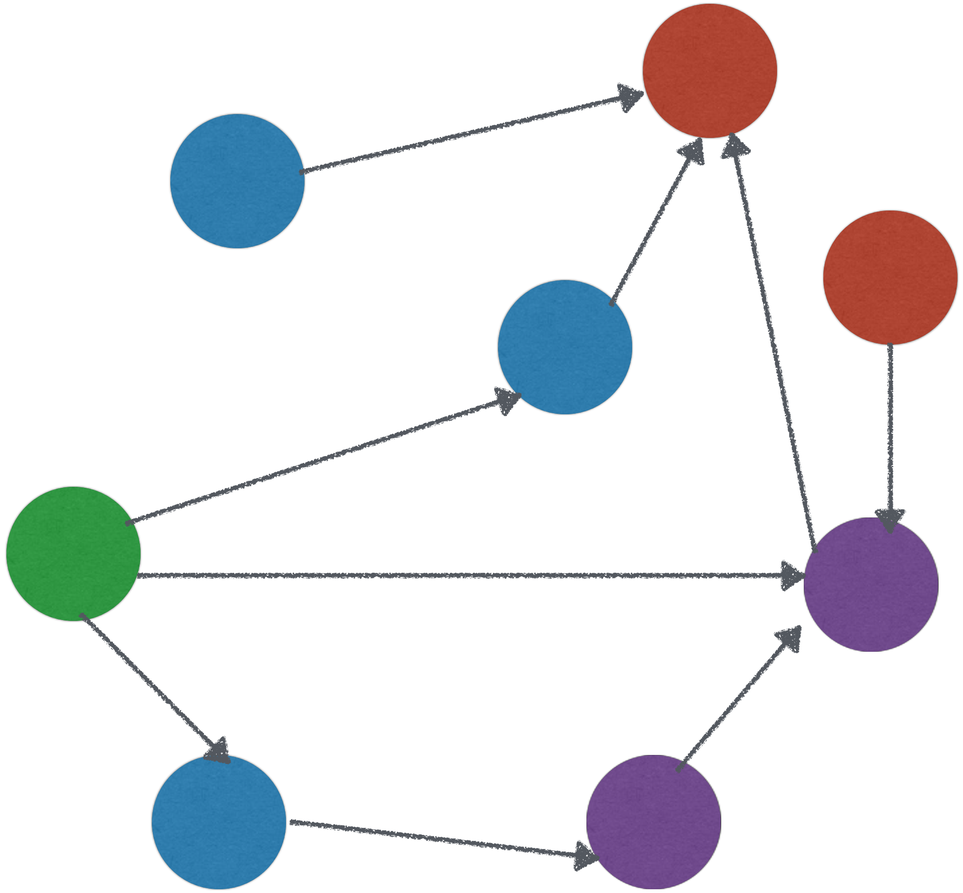
Using PageRank we can assign a rank to each of these webpages



PageRank

The rank increases with the number of links to the page

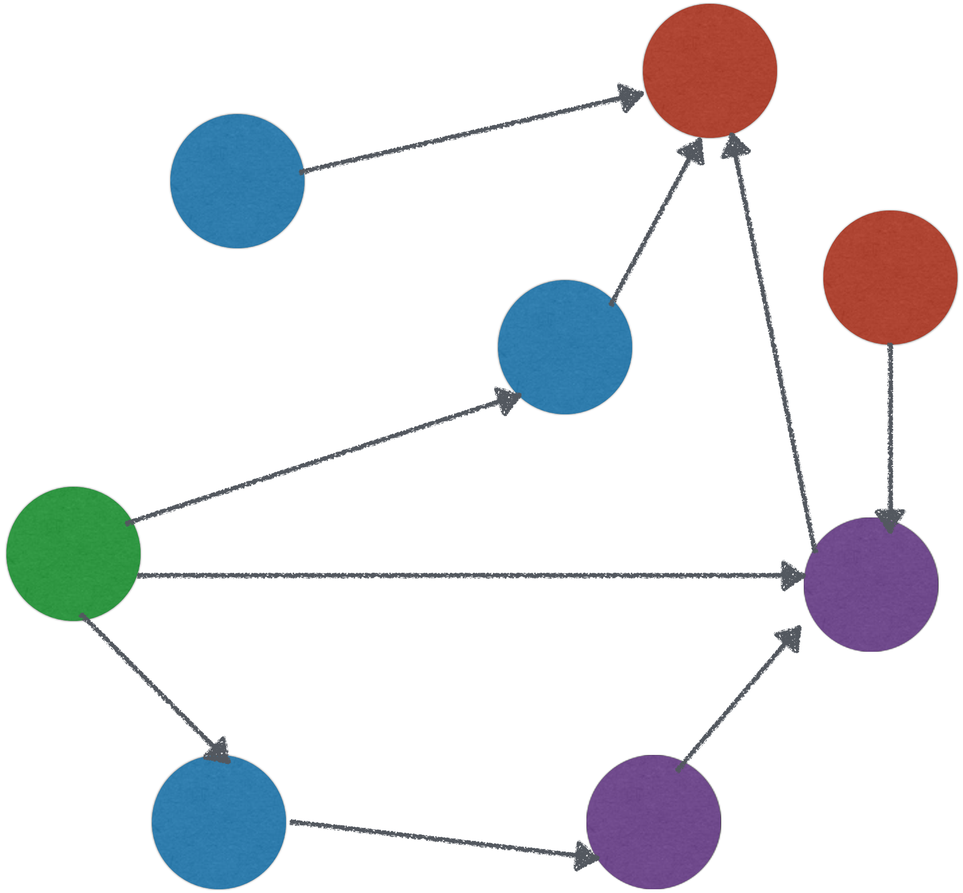
The higher the value of rank, more important the webpage



PageRank

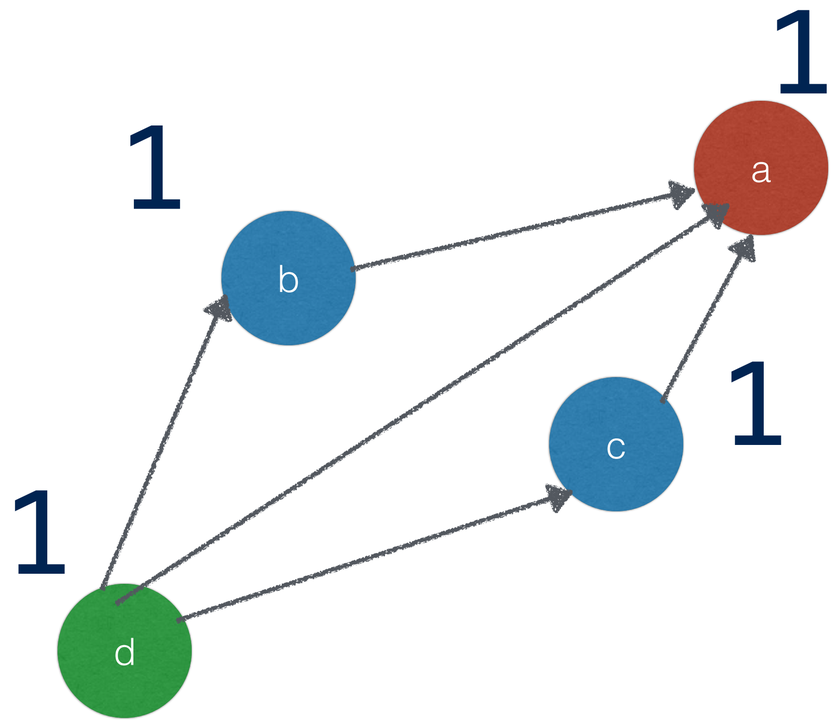
Let's take a small graph with just 4 webpages

And go through the PageRank algorithm



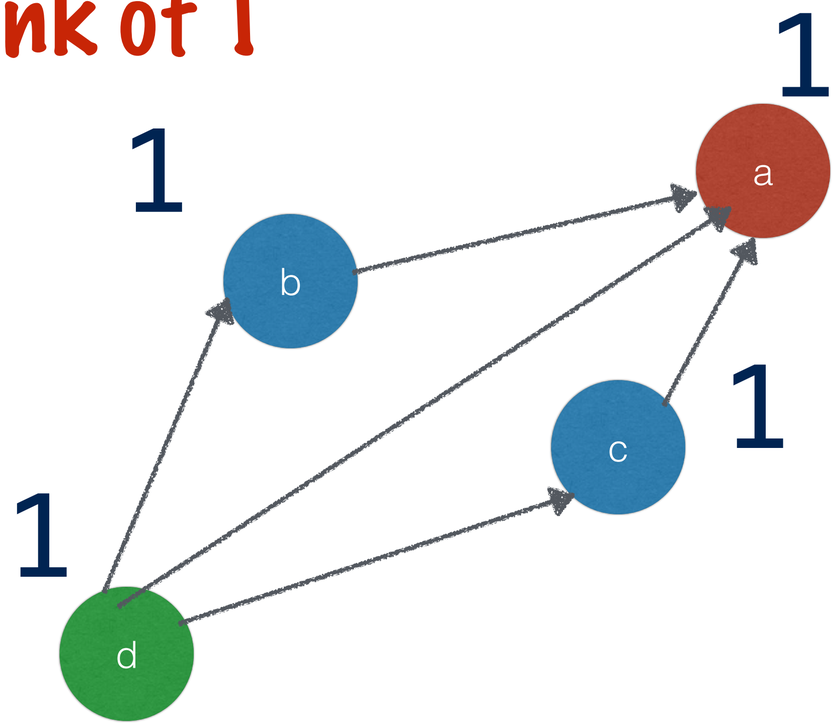
PageRank

Initially, all the webpages are given a rank of 1



PageRank

Initially, all the webpages are given a rank of 1



A rank defines the probability of landing on a page

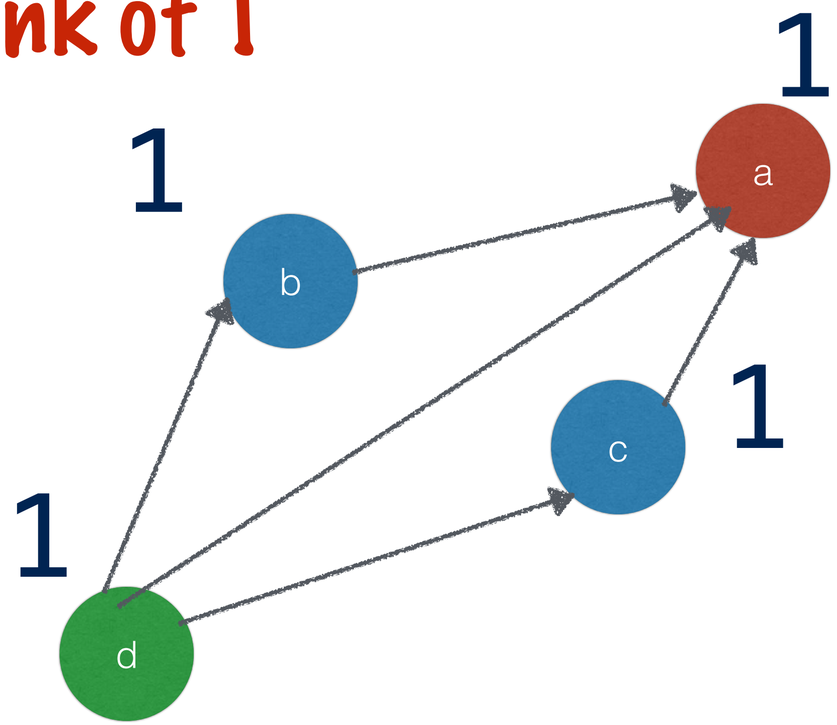
With N pages each start off with the same probability

PageRank

Initially, all the webpages are given a rank of 1

A rank defines the probability of landing on a page

With N pages each start off with the same probability

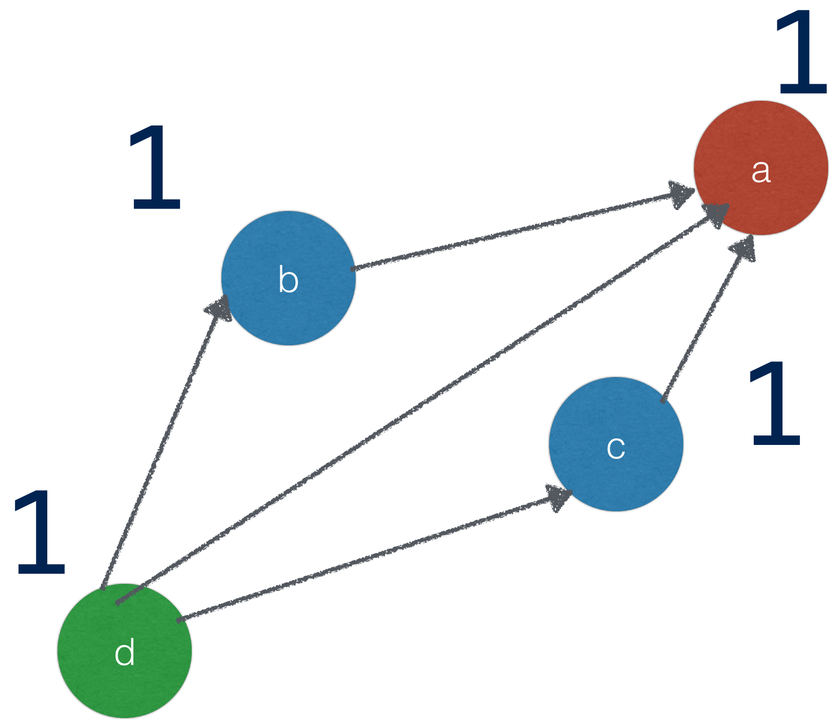


Each node has a 1/N probability of being visited

PageRank

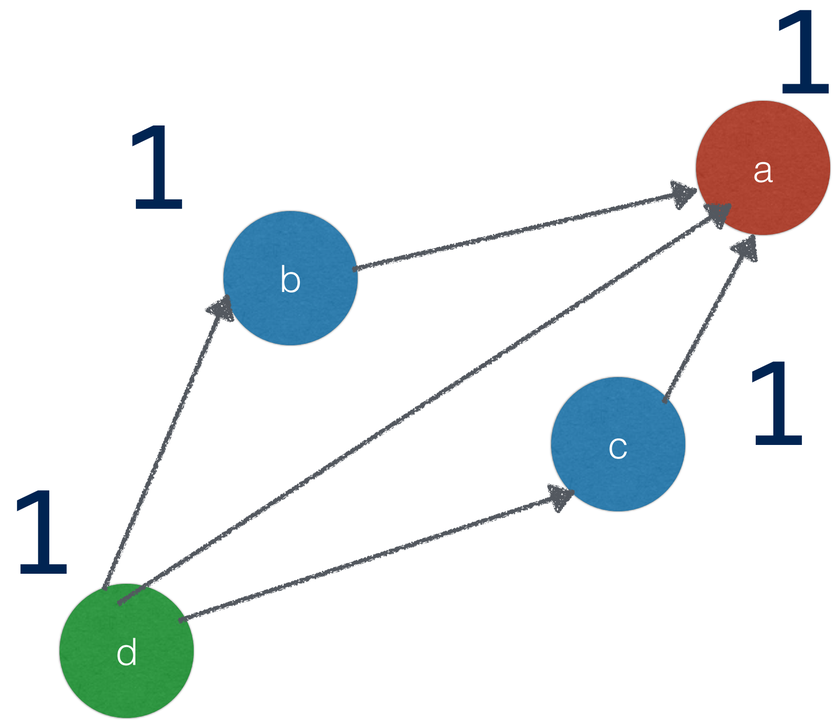
The ranks are updated iteratively

The total across all nodes should always remain constant = N



PageRank

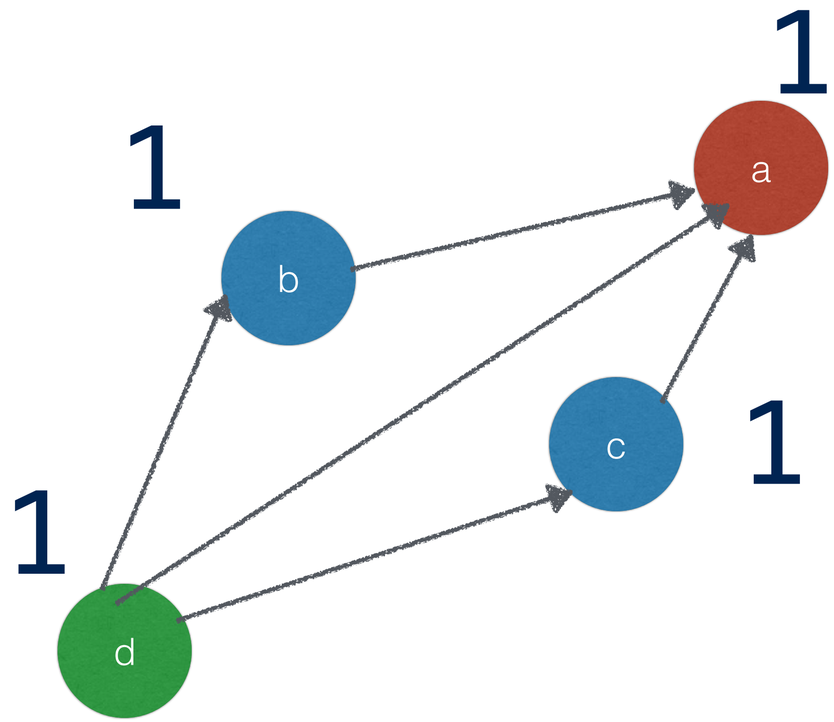
The total across all nodes should always remain constant = N



If a web surfer is more likely to visit page X, he will be less likely to visit page Y

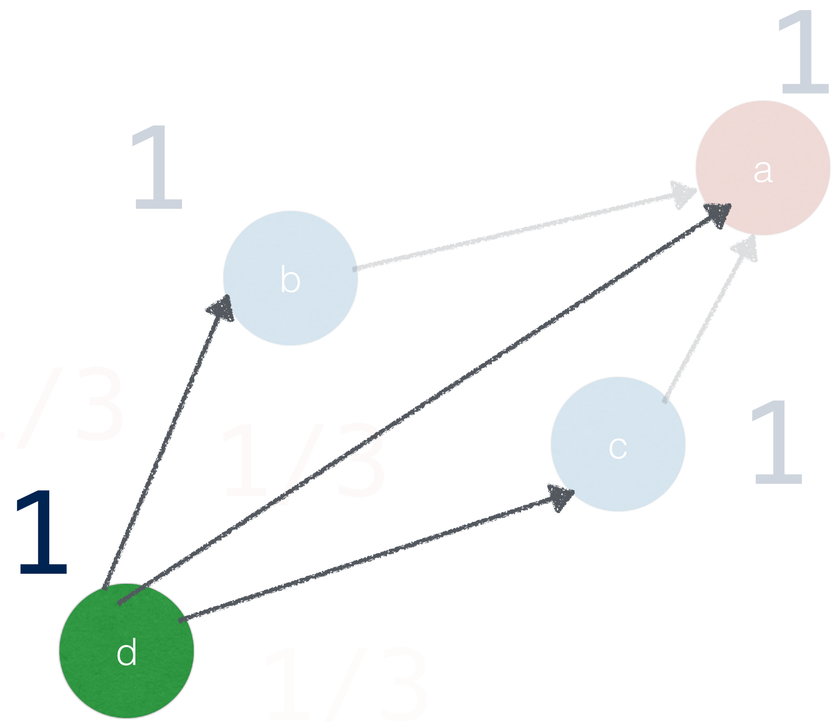
PageRank

In each iteration, pages transfer their rank equally to each of their neighbors



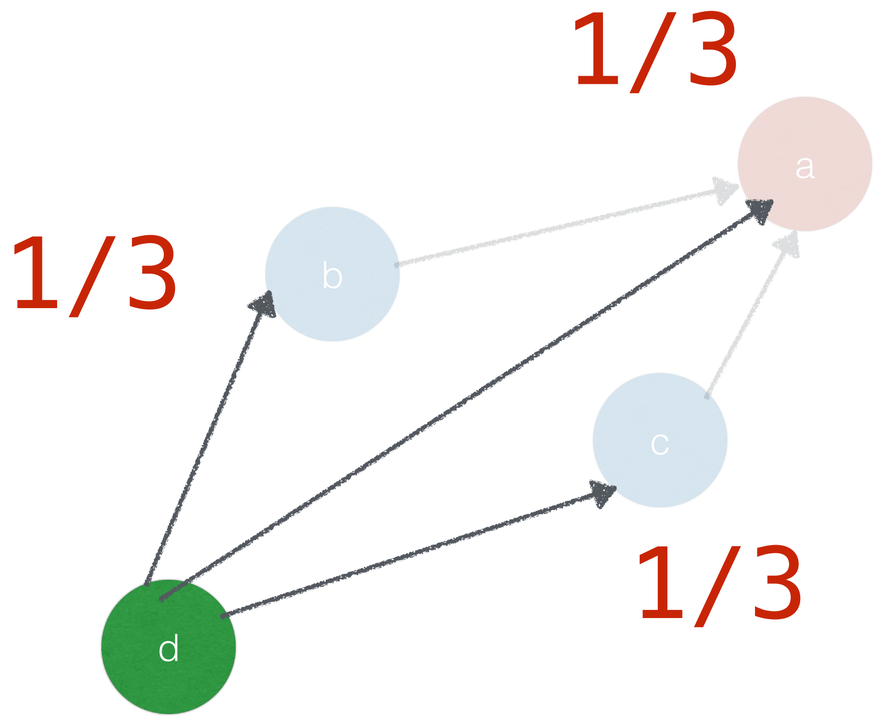
PageRank

d will transfer 1/3
to each of a,b,c



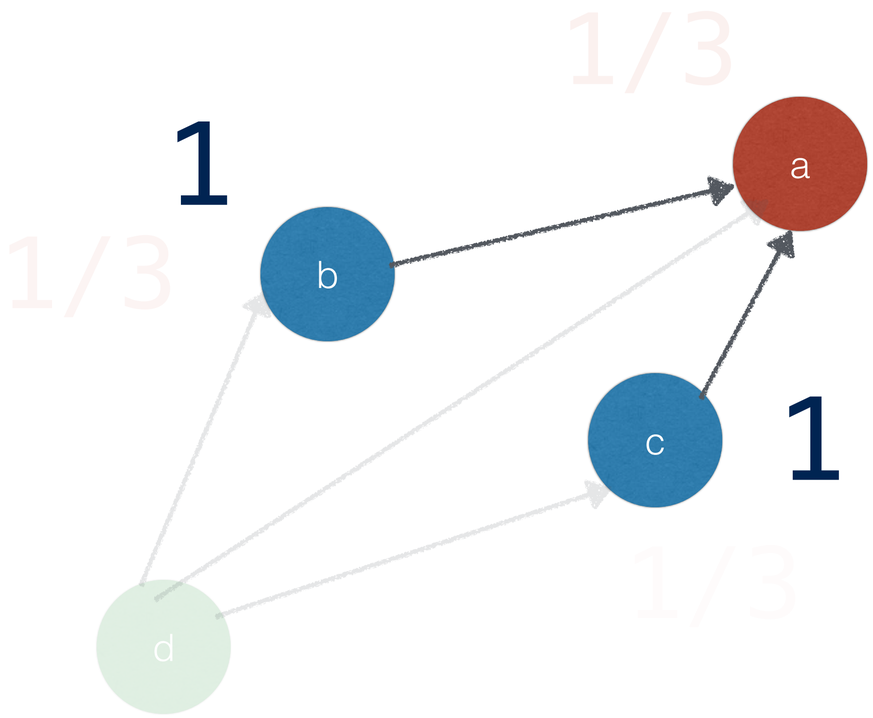
PageRank

d will transfer 1/3 to each of a,b,c



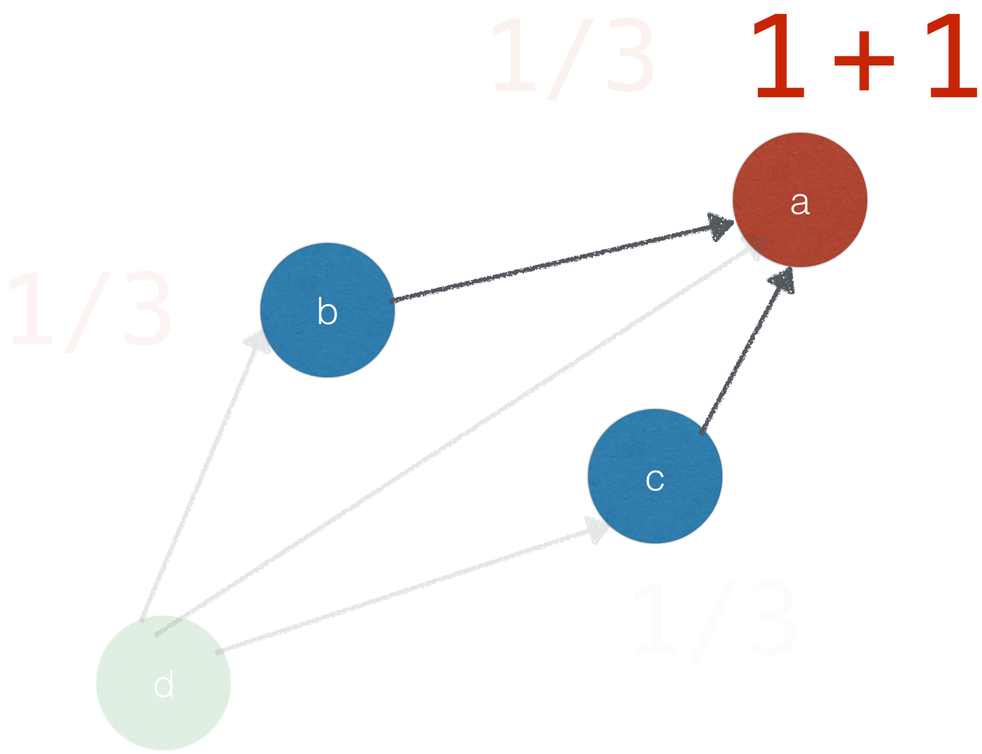
PageRank

b and c will each transfer the value 1 to a



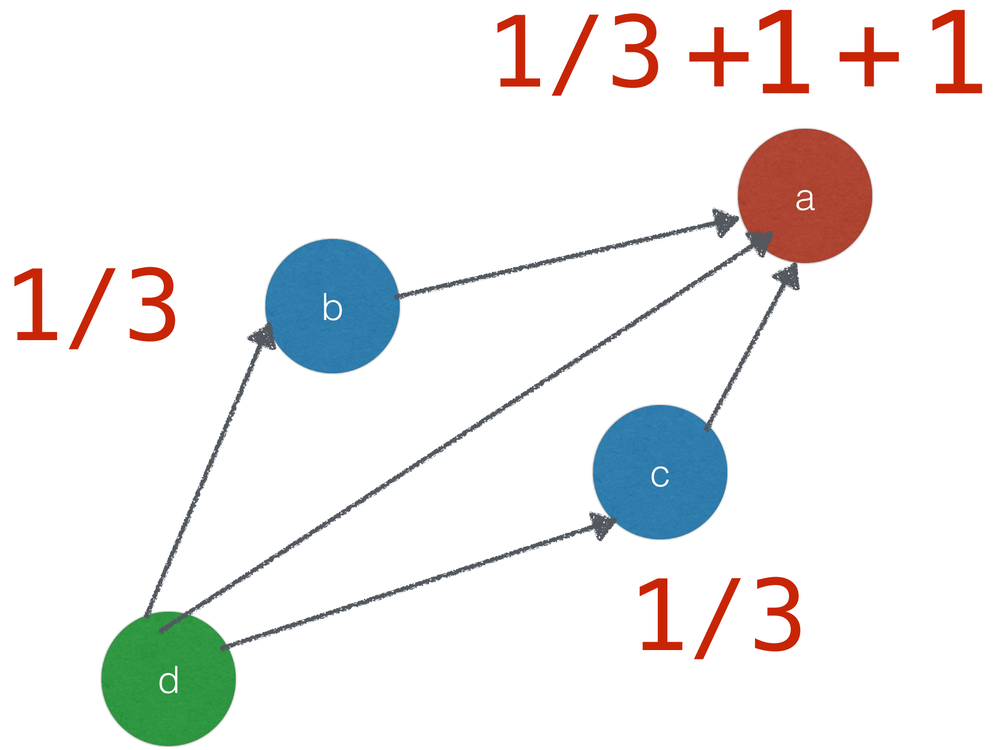
PageRank

b and c will each transfer the value 1 to a



PageRank

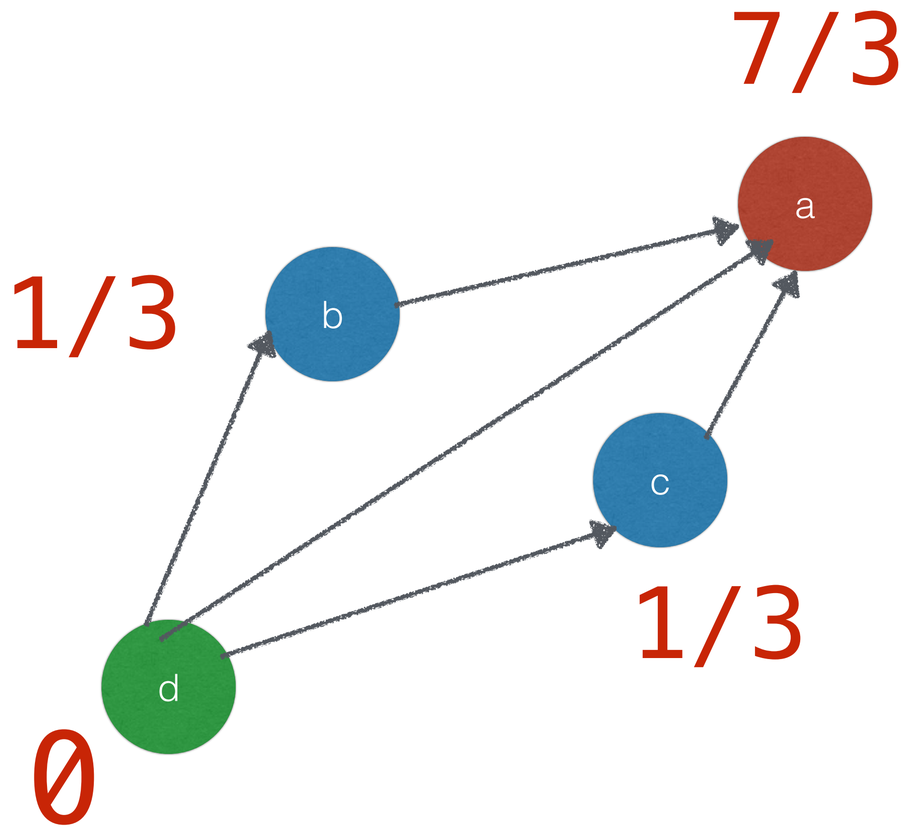
The new ranks are calculated using the sum of transferred values



PageRank

This process is repeated until the ranks converge

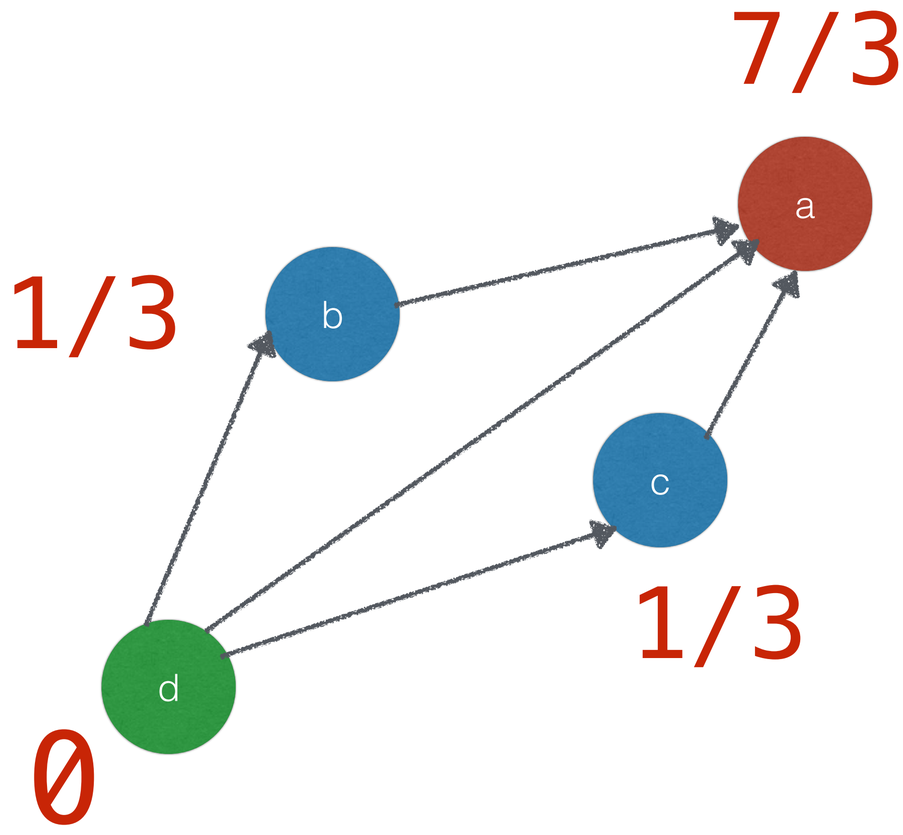
i.e. until the values don't change any further



PageRank

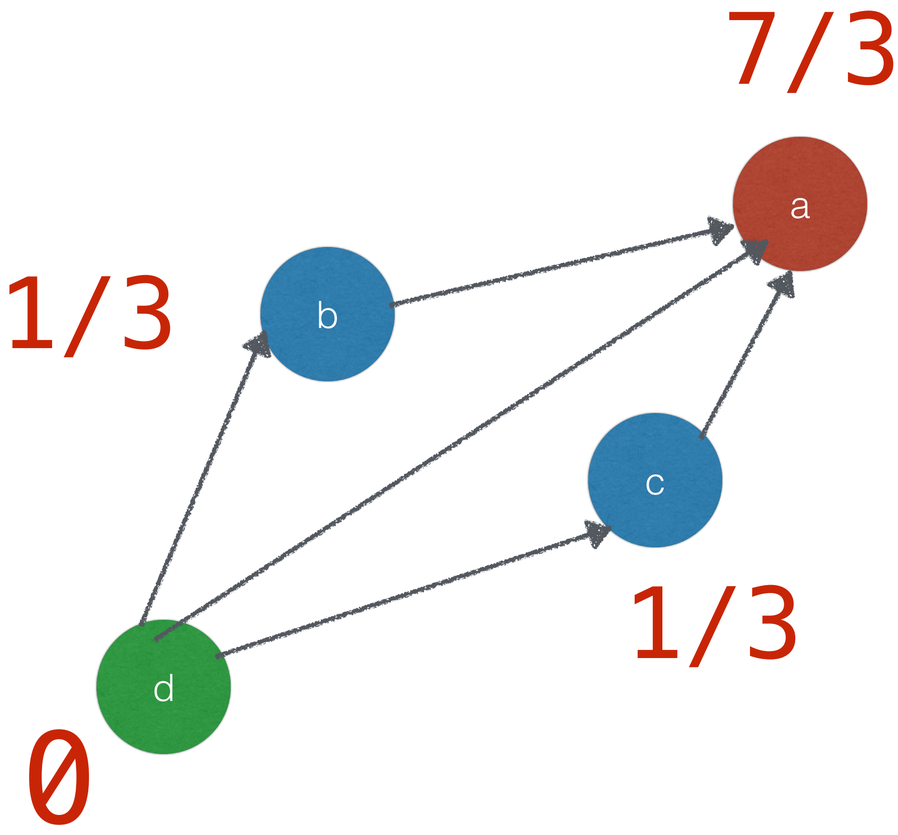
Damping factor

A damping factor is applied to the updated ranks after each iteration



PageRank

Damping factor



This is because in each iteration, the transferred value has travelled farther across the graph

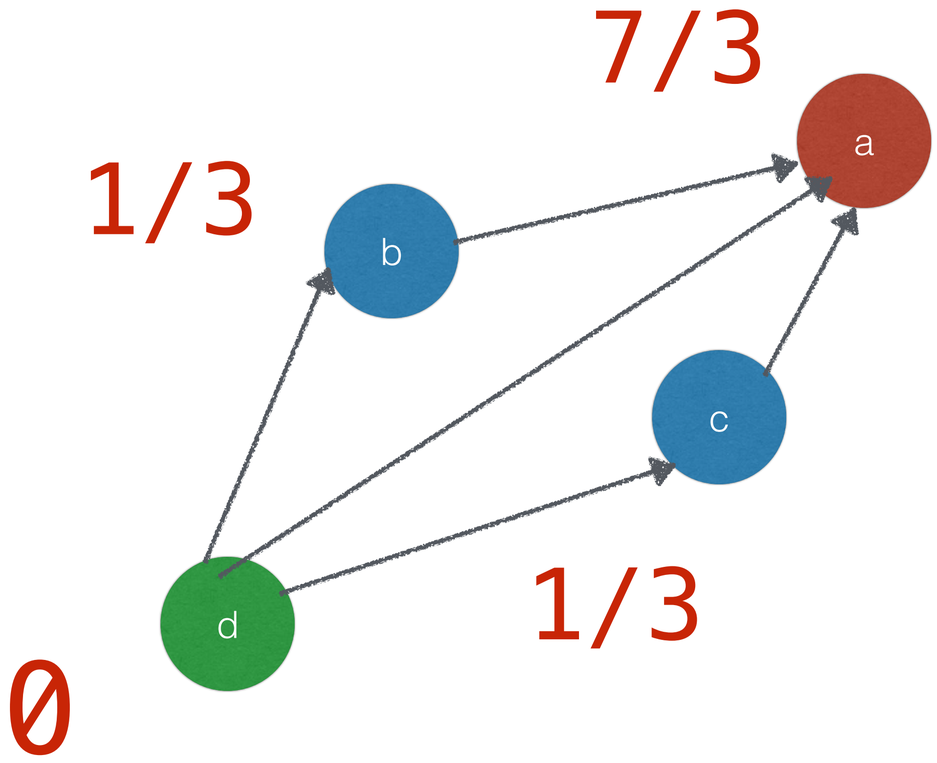
It's influence will have fallen

PageRank

Damping factor

Typically damping is done as follows:

rank * 0.85 + 0.15

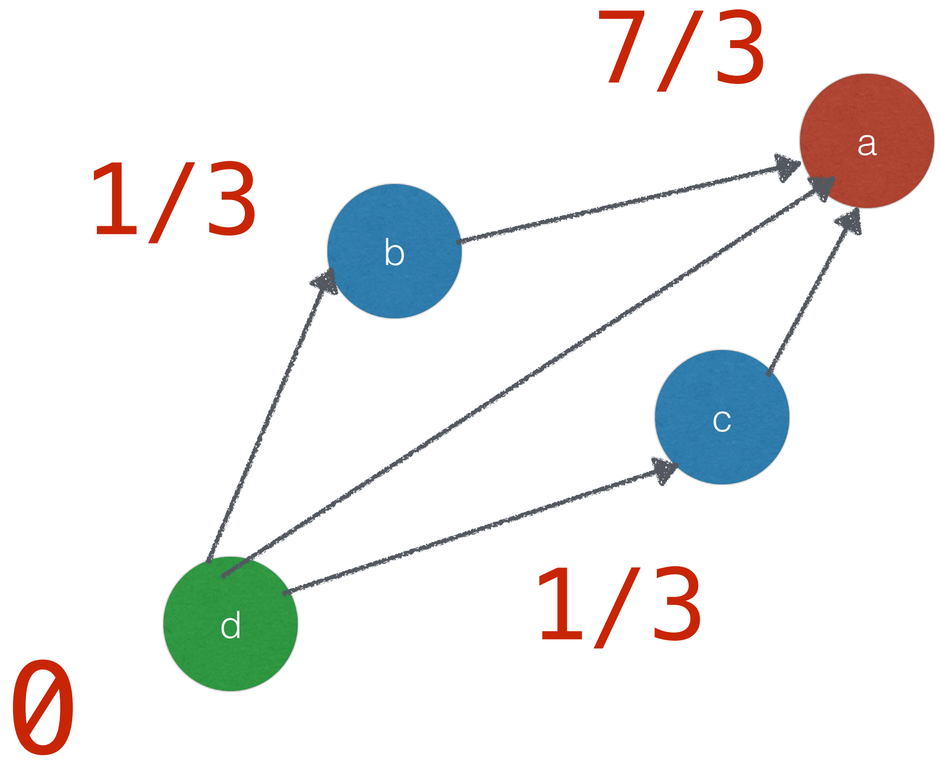


PageRank

Damping factor

rank * 0.85 + 0.15

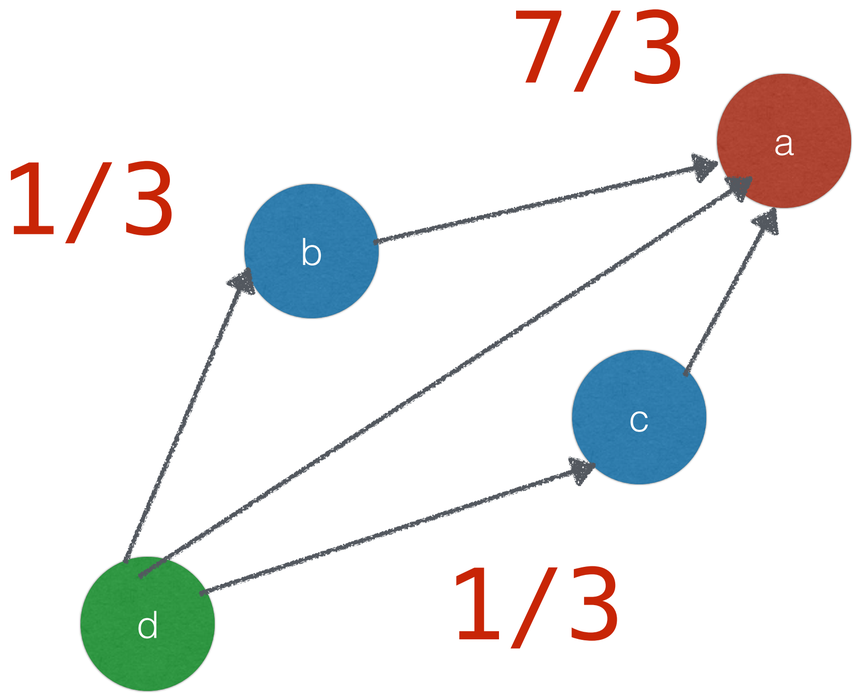
Consider only 85% of the current rank of every node



PageRank

Damping factor

rank * 0.85 + 0.15



Add this constant back to every node, so that the sum of ranks remains constant at N

0

PageRank

Damping factor

The damping factor accounts for this

7/3*0.85+0.15

1/3*0.85+0.15

1/3*0.85+0.15

0*0.85+0.15

PageRank

Let's see how to implement this in Spark

PageRank

We'll use a dataset released by Google in 2002 for a programming contest

PageRank

Google web graph

Dataset information

Nodes represent web pages and directed edges represent hyperlinks between them. The data was released in 2002 by Google as a part of Google Programming Contest.

Dataset statistics	
Nodes	875713
Edges	5105039
Nodes in largest WCC	855802 (0.977)
Edges in largest WCC	5066842 (0.993)
Nodes in largest SCC	434818 (0.497)
Edges in largest SCC	3419124 (0.670)
Average clustering coefficient	0.5143
Number of triangles	13391903
Fraction of closed triangles	0.01911
Diameter (longest shortest path)	21
90-percentile effective diameter	8.1

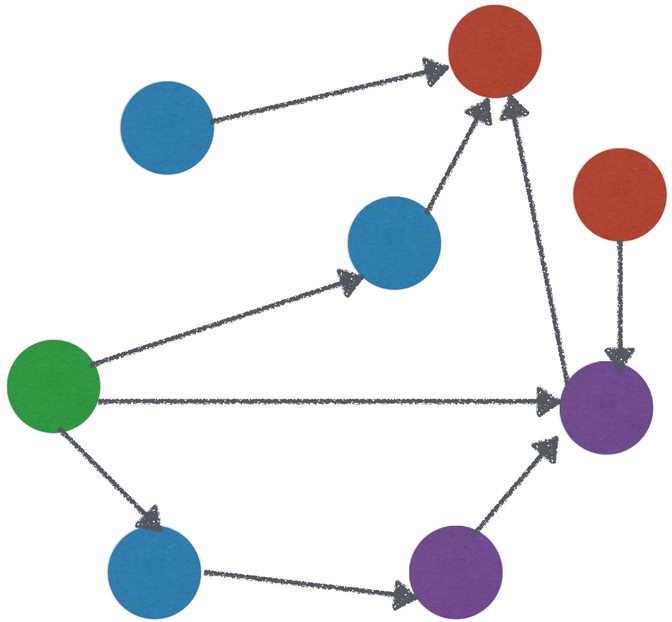
This dataset is publicly available as part of the Stanford Network Analysis Project

PageRank

FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835

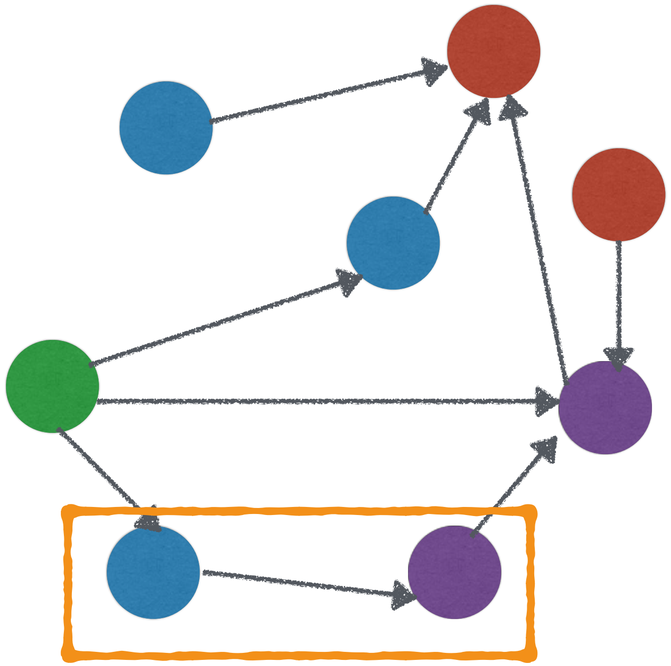
The dataset consists of a set of edges that form a web graph

=



PageRank

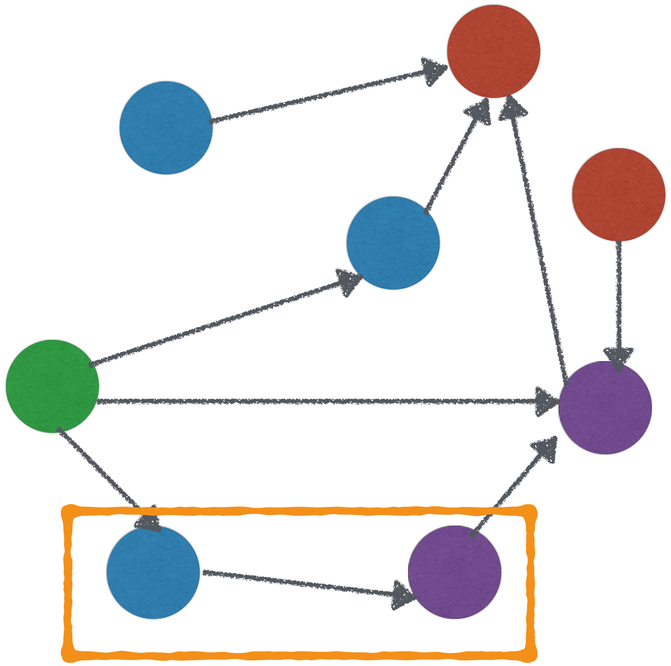
FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835



Each row represents an edge in the graph

PageRank

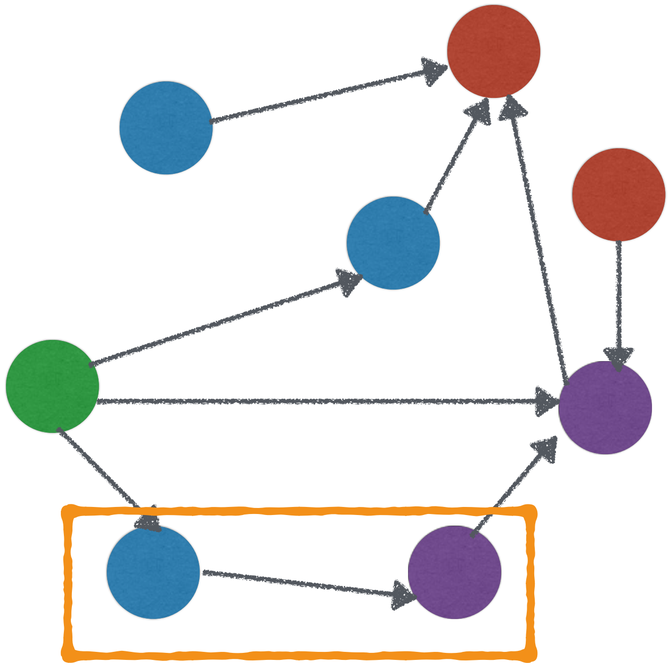
FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835



FromNodeId is a webpage which references the ToNodeId page

PageRank

FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835



Our objective: Calculate the PageRank for every webpage

weblinks

FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835

Step 1: We'll load this dataset into an RDD

PageRank

weblinks

FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing links from a page

PageRank

Step 1: We'll load this dataset into an RDD

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Step 2: Create a links RDD with all outgoing links from a page

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Ranks

NodeI	Rank
0	1
11342	1
..	..

Step 3: Initialize a ranks RDD with all ranks=1

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing

Step 3: Initialize a ranks RDD with all ranks=1

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Ranks

NodeI	Rank
0	1
11342	1
..	..

Step 4: Join the links and ranks RDDS

PageRank

Links		Ranks
FromNodeId	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178....	1
..	..	..

Step 1: We'll load this dataset into an RDD

eate a links RDD with all outgoing links from a pag

ep 3: Initialize a ranks RDD with all ranks=1

Step 4: Join the links and ranks RDDS

Step 5: Each node transfers its rank equally to its neighbors

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing links from a pa

Step 3: Initialize a ranks RDD with all ranks=1

Step 4: Join the links and ranks RDDS

Step 5: Each node transfers its rank equally to its neighbors

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25
11342	0.1
27469	0.2
638706	0.25
891835	0.15

Step 6: Apply a reduce operation on this RDD, to sum up values for the same node

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing links from a pa

Step 3: Initialize a ranks RDD with all ranks=1

Step 4: Join the links and ranks RDDS

Step 5: Each node transfers its rank equally to its neighbors

Step 6: Apply a reduce operation on this RDD, to sum up values for the same node

Ranks

NodeId	NewRank
11342	0.44
824020	0.36
867923	0.36
891835	0.36

Step 7: Apply the damping factor and use these as the updated ranks RDD

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing links from a pa

Step 3: Initialize a ranks RDD with all ranks=1

Step 4: Join the links and ranks RDDS

Step 5: Each node transfers its rank equally to its neighbors

Step 6: Apply a reduce operation on this RDD, to sum up values for the same node

Step 7: Apply the damping factor and use these as the updated ranks RDD

Ranks

NodeId	NewRank
11342	0.44
824020	0.36
867923	0.36
891835	0.36

Step 8: Repeat Steps 4-7 for a number of iterations

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing links from a page

Step 3: Initialize a ranks RDD with all ranks=1

Step 4: Join the links and ranks RDDS

Step 5: Each node transfers its rank equally to its neighbors

Step 6: Apply a reduce operation on this RDD, to sum up values for the same node

Step 7: Apply the damping factor and use these as the updated ranks RDD

Step 8: Repeat Steps 4-7 for a number of iterations

PageRank

Step 1: We'll load this dataset into an RDD

```
googleWeblinks=sc.textFile(googlePath).filter(lambda x:"#" not in x).map(lambda x:x.split("\t"))
```

Load the dataset

PageRank

Step 1: We'll load this dataset into an RDD

```
googleWeblinks=sc.textFile(googlePath).filter(lambda x:"#" not in x).map(lambda x:x.split("\t"))
```

Filter out comments and the header row

PageRank

Step 1: We'll load this dataset into an RDD

```
googleWeblinks=sc.textFile(googlePath).filter(lambda x:"#" not in x).map(lambda x:x.split("\t"))
```

Split the row into
(From Node Id, To Node Id)

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835

All values with the same key are grouped into a list

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

FromNodeId	ToNodeId
0	11342
0	824020
0	867923
0	891835
11342	0
11342	27469
11342	38716
11342	309564
11342	322178
11342	387543
11342	427436
11342	538214
11342	638706
11342	645018
11342	835220
11342	856657
11342	867923
11342	891835

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

This works similar to the persist() method

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

This RDD will be reused multiple times, so we persist it in-memory

PageRank

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

This is the advantage of using Spark for this kind of iterative processing

PageRank

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

In a system like MapReduce, this data would have been written to disk

And read from disk again in each iteration

PageRank

Step 2: Create a links RDD with all outgoing links from a page

```
links = googleWeblinks.groupByKey().cache()
```

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

With Spark, the data is just kept in-memory and passed on to the next iteration

PageRank

```
links = googleWeblinks.groupByKey().cache()
```

Step 3: Initialize a ranks RDD with all ranks=1

```
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))
```

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

All ranks are initially set to 1

PageRank

```
links = googleWeblinks.groupByKey().cache()
```

Step 3: Initialize a ranks RDD with all ranks=1

```
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))
```

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Ranks

NodeID	Rank
0	1
11342	1
..	..

Step 4: Join the links and ranks RDDS

```
links.join(ranks)
```

Links

FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Ranks

NodeI	Rank
0	1
11342	1
..	..

Step 4: Join the links and ranks RDDS

```
links.join(ranks)
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
links.join(ranks).flatMap(lambda url_urls_rank:
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

Divide the rank by the number of outgoing links from this node

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
links.join(ranks).flatMap(lambda url_urls_rank:
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

Divide the rank by the number of outgoing links from this node

That is the rank transferred to the neighbors of the node

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
links.join(ranks).flatMap(lambda url_urls_rank: computeContribs(url_urls_rank[1][0], url_urls_rank[1][1]))
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

Divide the rank by the number of outgoing links from this node

That is the rank transferred to the neighbors of the node

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
links.join(ranks).flatMap(lambda url_urls_rank: computeContribs(url_urls_rank[1][0], url_urls_rank[1][1]))
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
links.join(ranks).flatMap(lambda url_urls_rank: computeContribs url_urls_rank[1][0], url_urls_rank[1][1]))
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

PageRank

Links

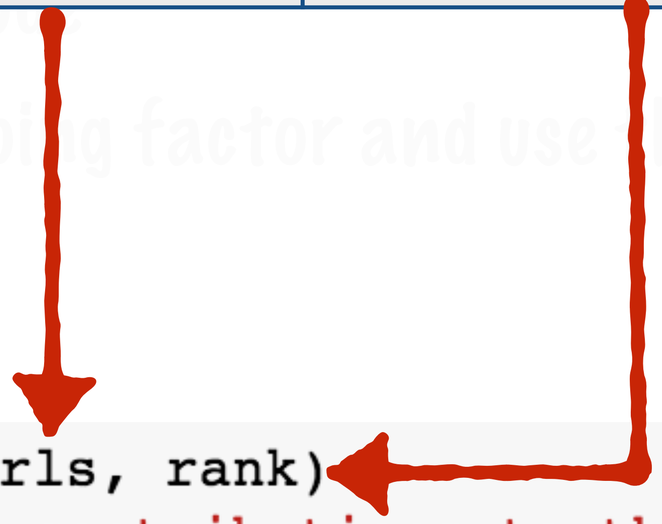
FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

```
def computeContribs(urls, rank):
    """Calculates URL contributions to the rank of other URLs."""
    contribs=[]
    num_urls = len(urls)
    for url in urls:
        contribs.append((url, rank / num_urls))
```

PageRank

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1



```
def computeContribs(urls, rank)
    """Calculates URL contributions to the rank of other URLs."""
    contribs=[]
    num_urls = len(urls)
    for url in urls:
        contribs.append((url, rank / num_urls))
```

Note that the rank is equally distributed

PageRank

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1

(url, contributing rank)

```
def computeContribs(urls, rank):
    """Calculates URL contributions to the rank of other URLs."""
    contribs=[]
    num_urls = len(urls)
    for url in urls:
        contribs.append((url, rank / num_urls))
```

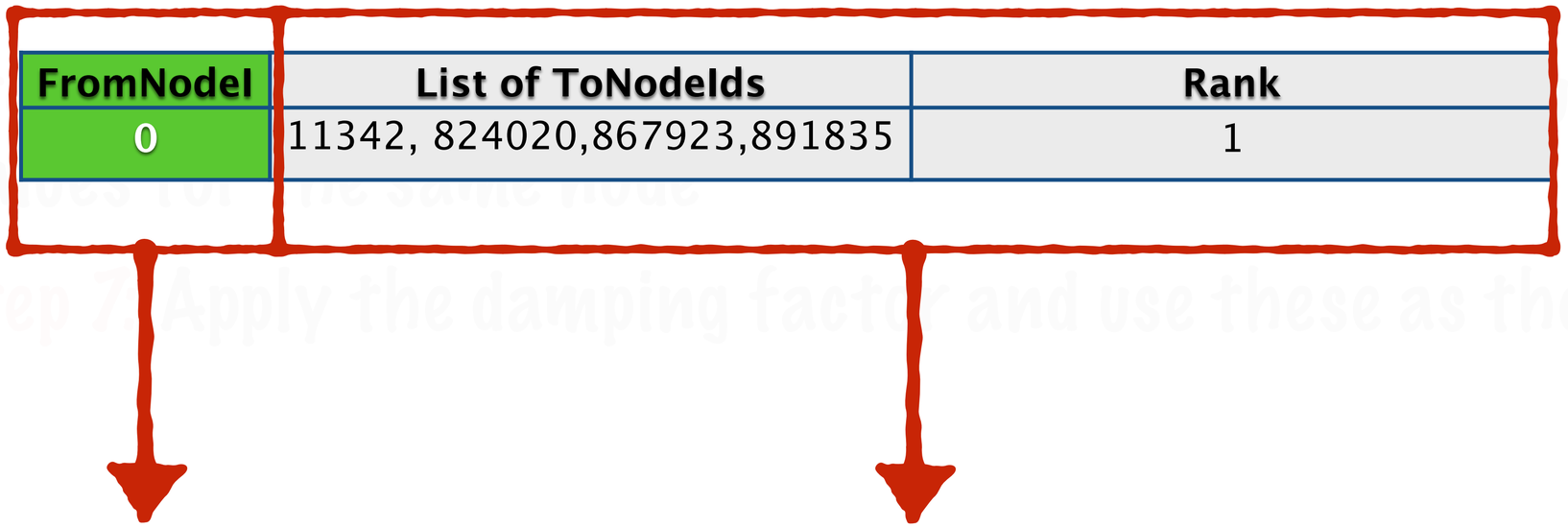
PageRank

```
lambda url_urls_rank: computeContribs(
      url_urls_rank[1][0], url_urls_rank[1][1]
)
```

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1

PageRank

```
lambda url_urls_rank: computeContribs(
        url_urls_rank[1][0], url_urls_rank[1][1]
)
```

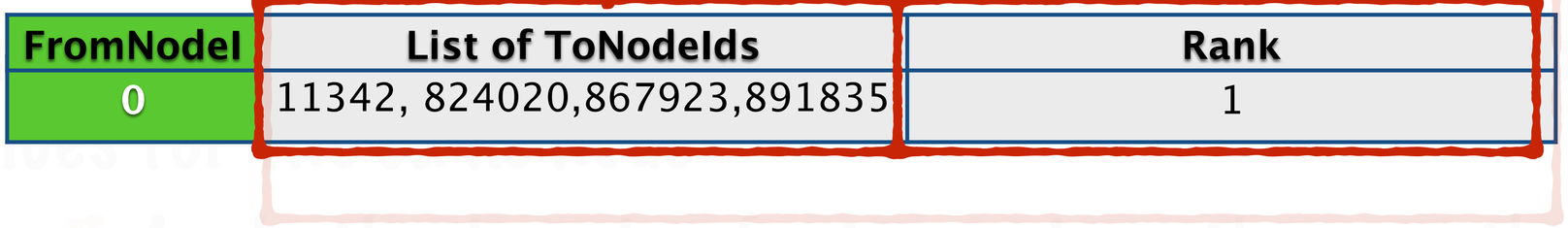


FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1

`url_urls_rank[0]` `url_urls_rank[1]`

PageRank

```
lambda url_urls_rank: computeContribs(
      url_urls_rank[1][0], url_urls_rank[1][1]
)
```



FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1

url_urls_rank[1]

The input arguments to computeContribs is the list of urls and rank

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
links.join(ranks).flatMap(lambda url_urls_rank: computeContribs url_urls_rank[1][0], url_urls_rank[1][1]))
```

Links

FromNodeI	List of ToNodeIds	Rank
0	11342, 824020,867923,891835	1
11342	0,27469,38716,309564,322178	1
..	..	..

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
contribs = links.join(ranks).flatMap(lambda url_urls_rank: computeContribs(url_urls_rank[1][0], url_urls_rank
```

0	11342, 824020,867923,891835

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

flatMap flattens any list/collection in the values portion of the RDD

PageRank

Step 5: Each node transfers its rank equally to its neighbors

```
contribs = links.join(ranks).flatMap(lambda url_urls_rank: computeContribs(url_urls_rank[1][0], url_urls_rank
```

At the end of this we have all the transferred ranks

NodeId	TransferredRank
11342	0.25
824020	0.25
867923	0.25
891835	0.25

PageRank

Step 6: Apply a reduce operation on this RDD, to sum up values for the same node

```
ranks = contribs.reduceByKey(lambda x,y:x+y).ma
```

We get the sum of contributions on a per node basis

Step 7: Apply the damping factor and use these as the updated ranks RDD

```
ranks = contribs.reduceByKey(lambda x,y:x+y).mapValues(lambda rank: rank * 0.85 + 0.15)
```

Apply the damping factor on every node

Step 8: Repeat Steps 4-7

```
for iteration in range(10):
  contribs = links.join(ranks).flatMap(lambda url_urls_rank: computeContribs(url_urls_rank
  ranks = contribs.reduceByKey(lambda x,y:x+y).mapValues(lambda rank: rank * 0.85 + 0.15)
```

We can set up a stopping condition, or just run for a large number of iterations

CUSTOM PARTITIONING

What happens when we join 2 Pair RDDs?

Join 2 Pair RDDs

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

Ranks	
NodeI	Rank
0	1
11342	1
..	..

Both of these RDDs are distributed across some nodes in the cluster

Join 2 Pair RDDs

Links	
FromNodeId	List of ToNodeIds
0	11342, 824020,867923,891835
11342	0,27469,38716,309564,322178....
..	..

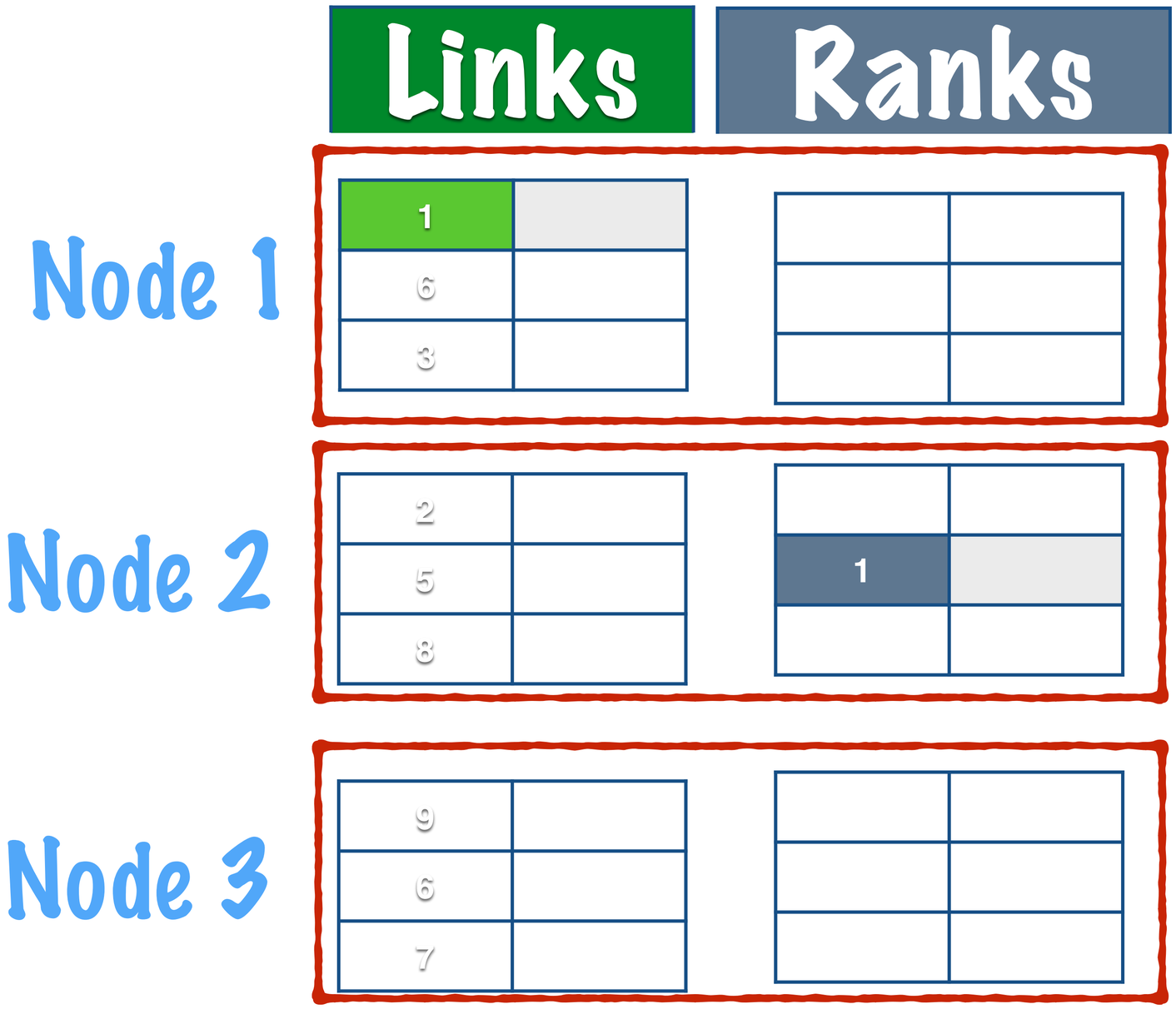
Ranks	
NodeI	Rank
0	1
11342	1
..	..

Both of these RDDs are distributed across some nodes in the cluster

Join 2 Pair RDDs

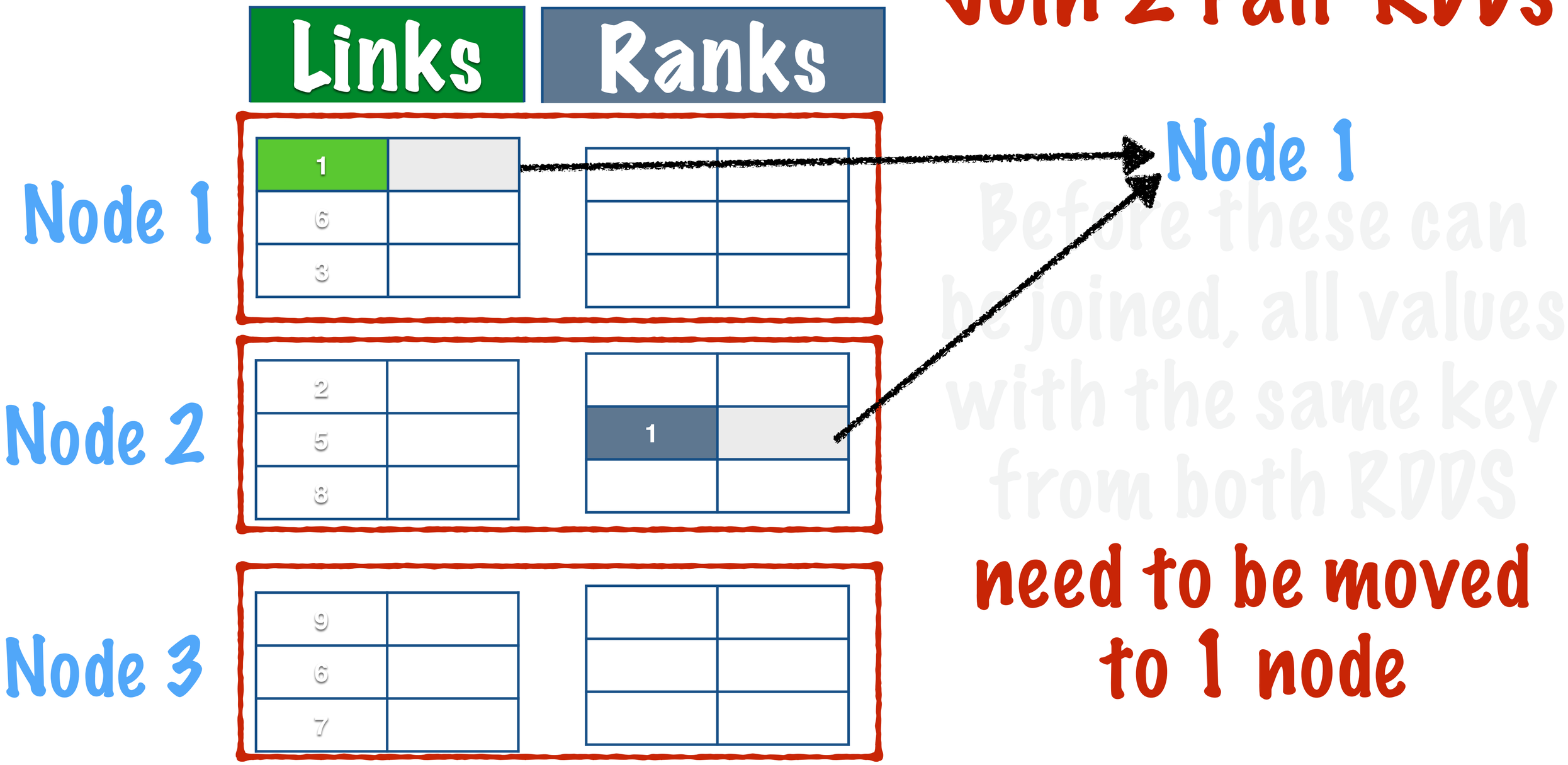
Before these can be joined, all values with the same key from both RDDS need to be moved to 1 node

Join 2 Pair RDDs

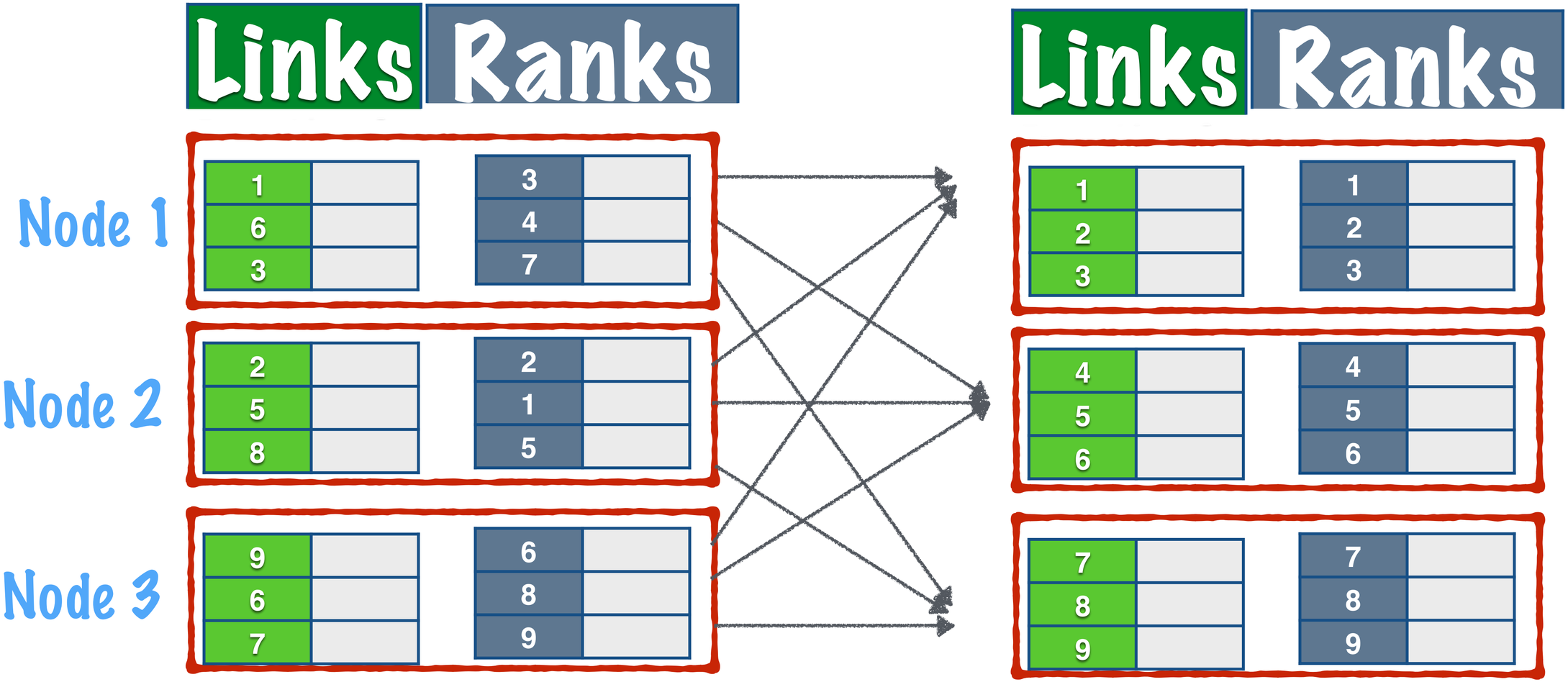


Before these can be joined, **all values with the same key from both RDDS** need to be moved to 1 node

Join 2 Pair RDDs



Join 2 Pair RDDs

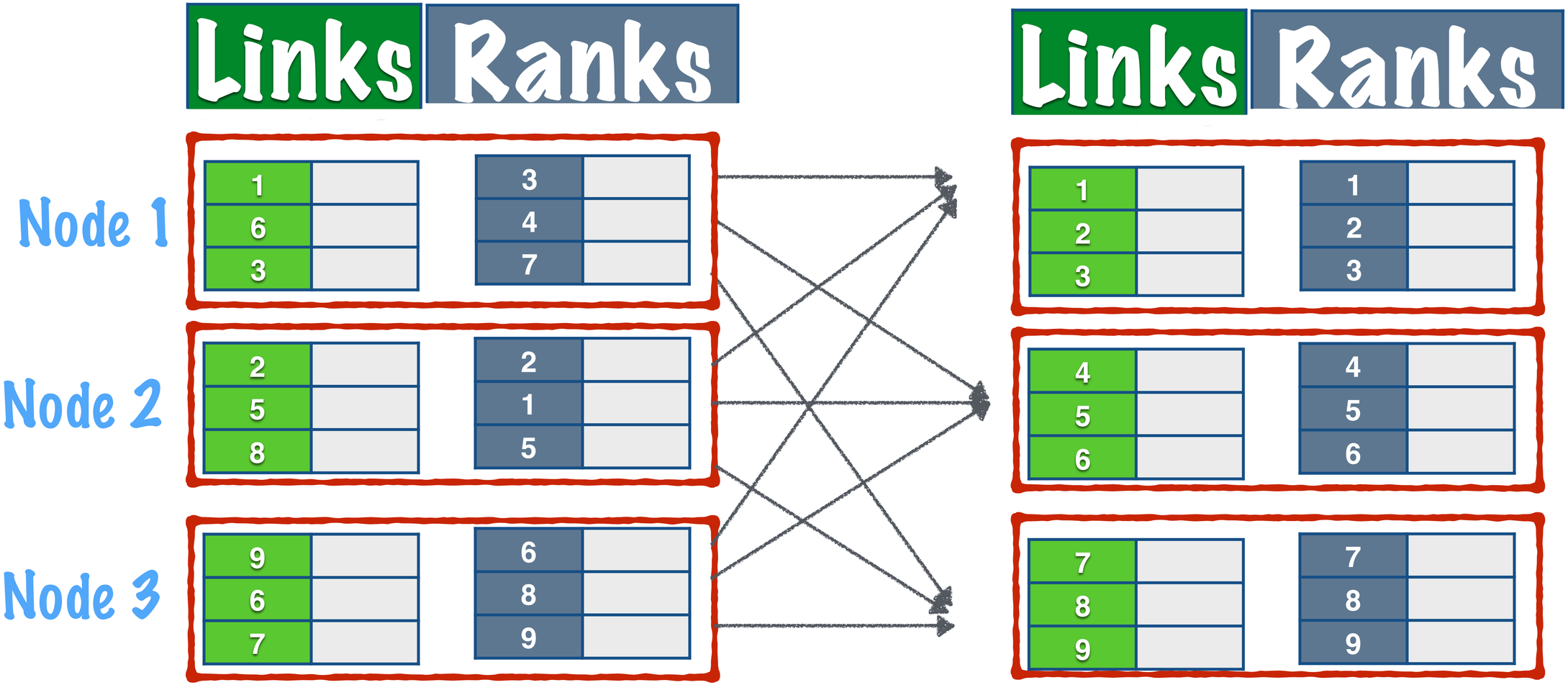


The records are shuffled across nodes

The records are shuffled across nodes

Shuffle operations are very expensive

Join 2 Pair RDDs



By default both RDDs are shuffled

Spark has a feature to help optimize such operations

Custom Partitioning

Custom Partitioning

Say you have a Pair RDD that you know will be reused often

In particular, the RDD will be used for multiple join operations

Custom Partitioning

You can explicitly set a partitioning option for this RDD

```
.partitionBy(100)
```

This will create a hash index for the keys of the RDD

Custom Partitioning

```
.partitionBy(100)
```

hash index for the keys

The hash id for a key is computed using this number

Custom Partitioning

```
.partitionBy(100)
```

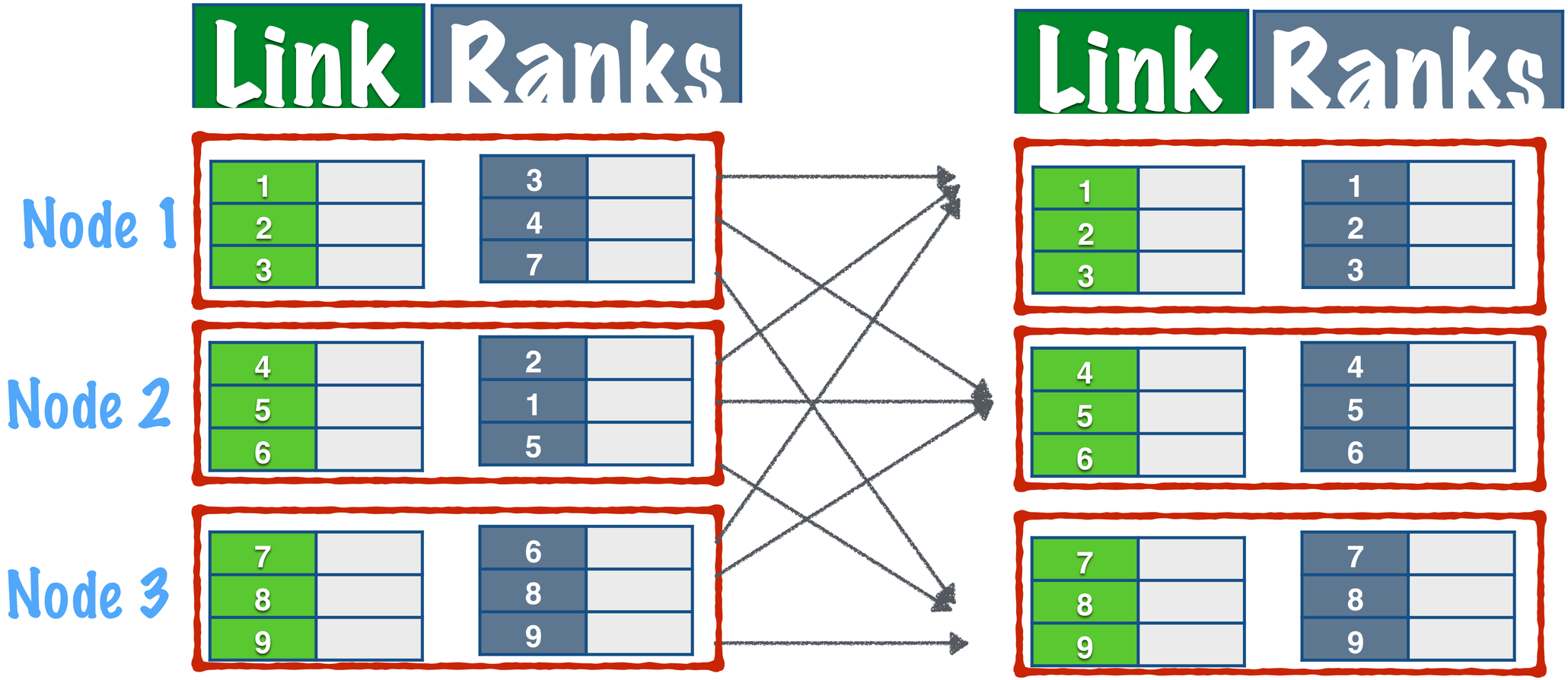
hash index for the keys

All records with the same hash id are distributed to the same node

Custom Partitioning

```
.partitionBy(100)
```

Spark will not re-shuffle the Pair RDDS which have been explicitly partitioned



If you partition the Links RDD, only the Ranks RDD is reshuffled

Custom Partitioning

Note: Custom Partitioning is only available for PairRDDs

CUSTOM PARTITIONING IN PAGERANK

Recap

PageRank

Step 1: We'll load this dataset into an RDD

Step 2: Create a links RDD with all outgoing links from a page

Step 3: Initialize a ranks RDD with all ranks=1

Step 4: Join the links and ranks RDDS

Step 5: Each node transfers its rank equally to its neighbors

Step 6: Apply a reduce operation on this RDD, to sum up values for the same node

Step 7: Apply the damping factor and use these as the updated ranks RDD

Step 8: Repeat Steps 4-7 for a number of iterations

The links RDD is reused many times

Step 4: Join the links and ranks RDDS

```
links = googleWeblinks.groupByKey().cache()
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))

for iteration in range(10):
    contribs = links.join(ranks).flatMap(lambda url_urls_rank: computeContribs(url_urls_rank[1][0], url_urls_rank[1][1]))
    ranks = contribs.reduceByKey(lambda x,y:x+y).mapValues(lambda rank: rank * 0.85 + 0.15)
```

It does not change once set up

Step 8: Repeat Steps 4-7 for a number of iterations

PageRank

We can set the partitioning for the links RDD

Step 4: Join the links and ranks RDDS

```
links = googleWeblinks.groupByKey().cache()
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))

for iteration in range(10):
    contribs = links.join(ranks).flatMap(lambda url_urls_rank: comput
    ranks = contribs.reduceByKey(lambda x,y:x+y).mapValues(lambda ran
```

Step 8: Repeat Steps 4-7 for a number of iterations

PageRank

Step 4: Join the links and ranks RDDS

```
links = googleWeblinks.groupByKey().partitionBy(100).cache()
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))

for iteration in range(10):
    contribs = links.join(ranks).flatMap(lambda url_urls_rank: comput
    ranks = contribs.reduceByKey(lambda x,y:x+y).mapValues(lambda ran
```

Step 8: Repeat Steps 4-7 for a number of iterations