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

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

```scala
val googleWeblinks=sc.textFile(googlePath).filter(!_.contains("#")).map(_.split("\t")).map(x => (x(0),x(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 until the ranks converge

Load the dataset

PageRank

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

```
ks=sc.textFile(googlePath).filter(!_.contains("#")).map(_.split("\t"))
```

Filter out comments and the header row

PageRank

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

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

```
oglePath).filter(!_.contains("#")).map(_.split("\t")).map(x => (x(0),
```

Step 3: Initialize a ranks RDD with all ranks

Step 4: Join the links and ranks RDDS

Split the row into an Array

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 until the ranks converge

PageRank

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

```
.contains("#")).map(_.split("\t")).map(x => (x(0),x(1)))
```

Represent each row as a tuple

(From Node Id, To Node Id)

PageRank

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

```
val 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

```
val 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

```
val 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

```
val 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

```
val 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

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

```
val 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

```
val 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

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

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

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

```
var ranks = links.mapValues(v => 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

Step 8: Repeat Steps 4-7 until the ranks converge

PageRank

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

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

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

```
var ranks = links.mapValues(v => 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
..	..

PageRank

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
..	..

PageRank

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)
```

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)
```

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

That is the rank transferred to the neighbors of the node

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

PageRank

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

0	11342, 824020,867923,891835	1

List of URLS

PageRank

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

```scala
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

0	11342, 824020,867923,891835	1

Rank

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

The number of outgoing links

PageRank

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

For each outgoing link,
a tuple is generated
(url, contributing rank)

PageRank

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

(url, contributing rank)

Note that the rank is equally distributed

PageRank

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

This function returns an array of tuples

PageRank

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

0	11342, 824020,867923,891835

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

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

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

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
```

At the end of this we have all the transferred ranks

PageRank

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

```
ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
```

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(_ + _).mapValues(0.15 + 0.85 * _)
```

Apply the damping factor on every node

PageRank

Step 8: Repeat Steps 4-7

```
for (i <- 1 to iters) {
```

```
val contribs = links.join(ranks).values.flatMap{case (urls, rank) =>
  val size = urls.size
  urls.map(url => (url, rank / size))
}
ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
```

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....
..	..

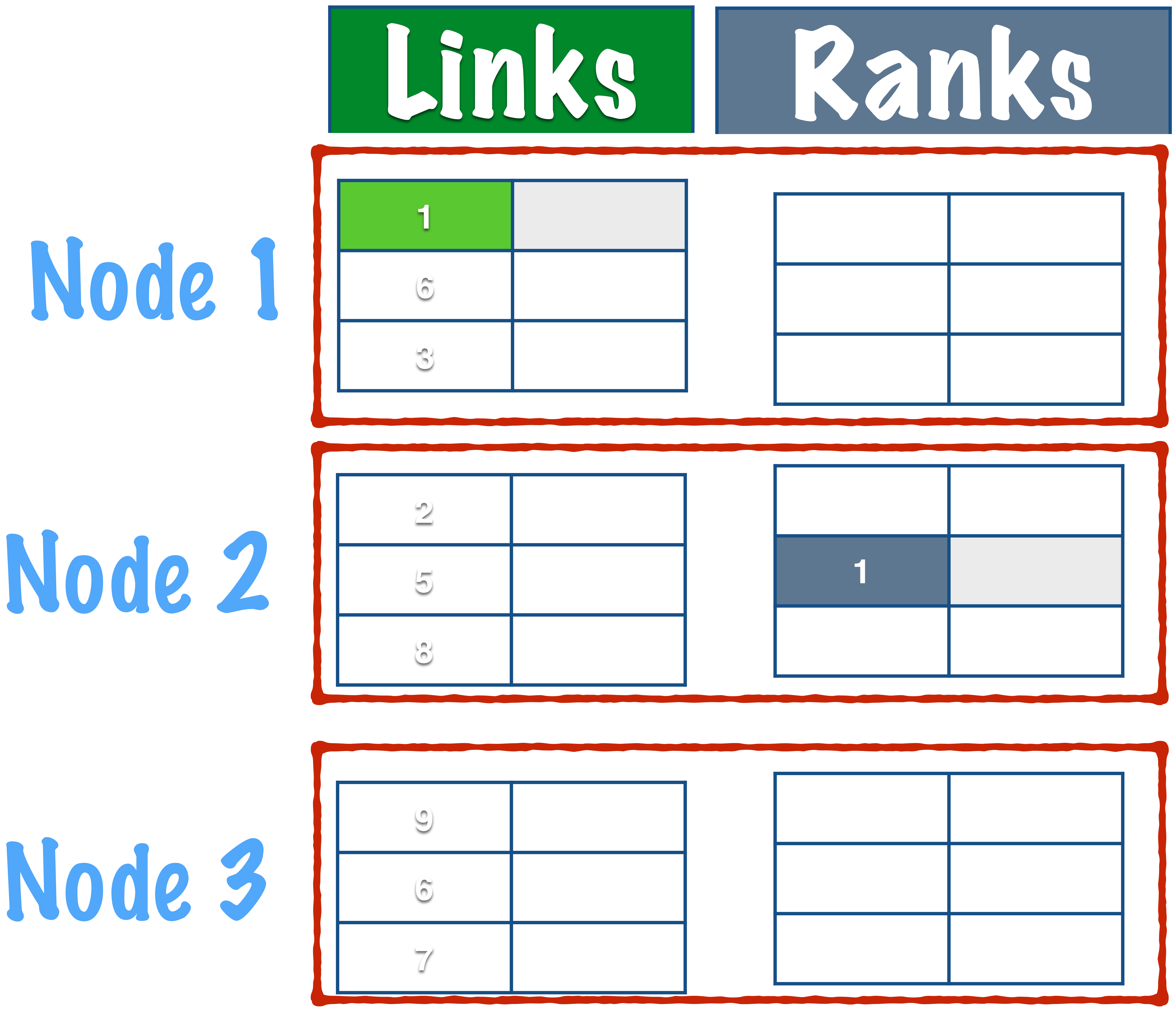
Ranks	
NodeI	Rank
0	1
11342	1
..	..

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

Join 2 Pair RDDs

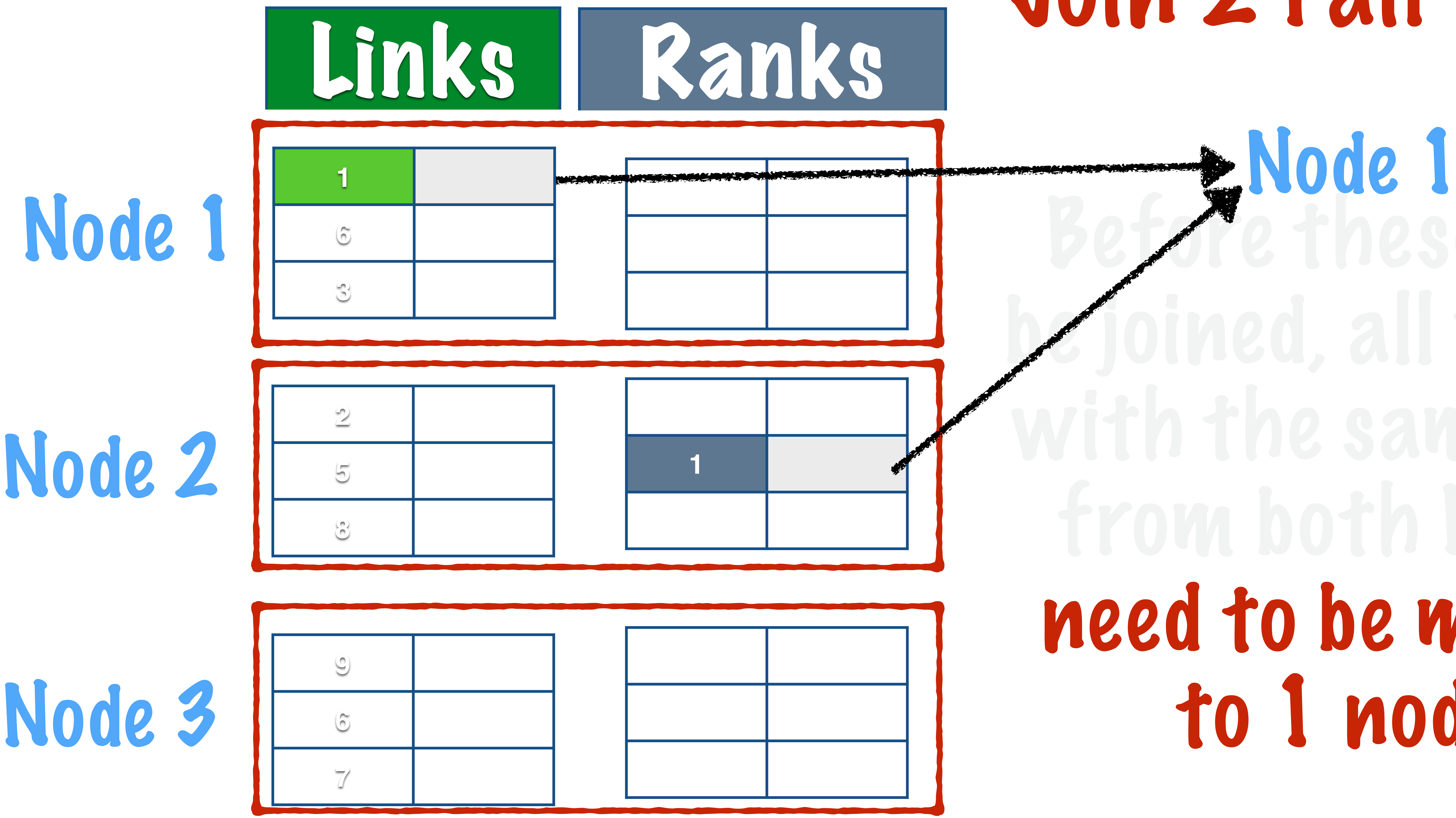
Before these can be joined, all values with the same key from both RDDS need to be moved to 1 node

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

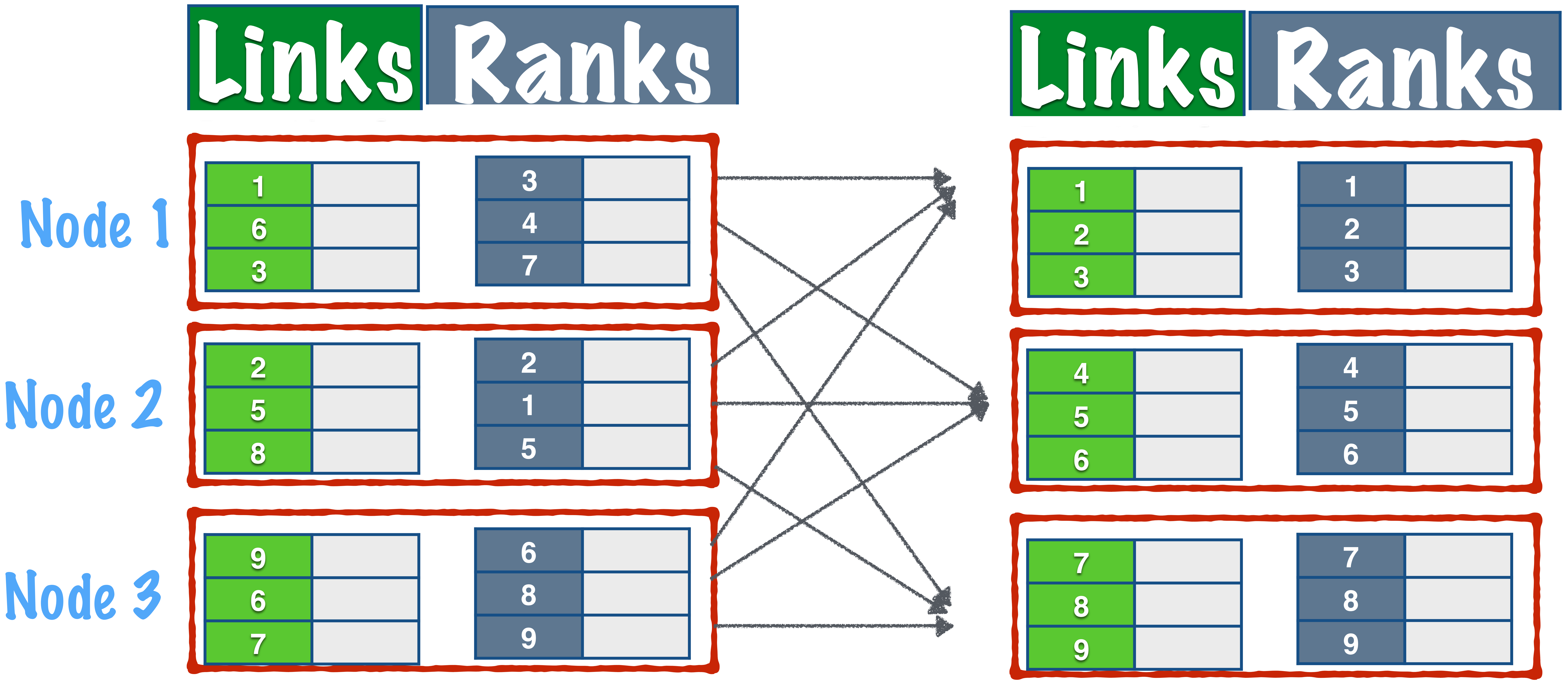
Join 2 Pair RDDs



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

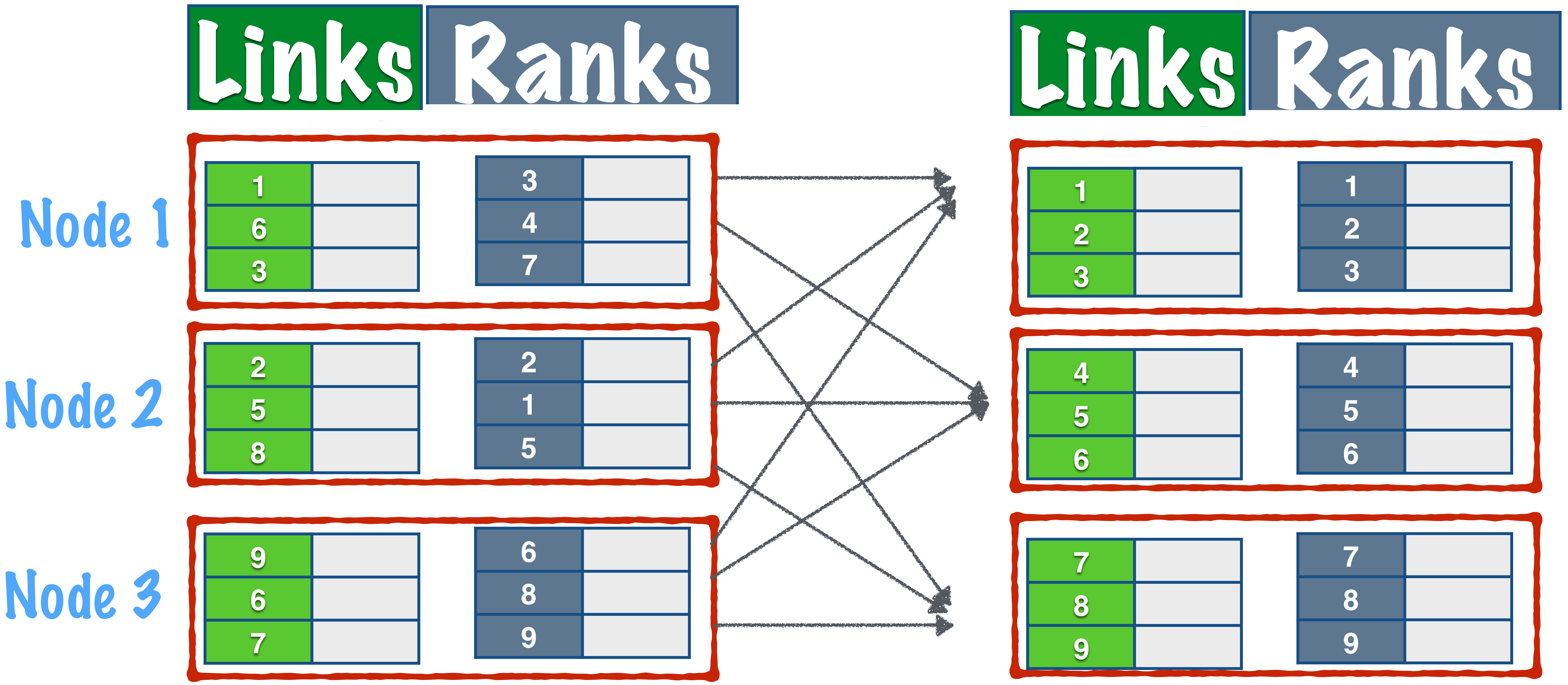


The records are shuffled across nodes

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(new HashPartitioner(100)).
```

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

Custom Partitioning

```
partitionBy(new HashPartitioner(100)).
```

hash index for the keys

The hash id for a key is computed using this number

Custom Partitioning

```
partitionBy(new HashPartitioner(100)).
```

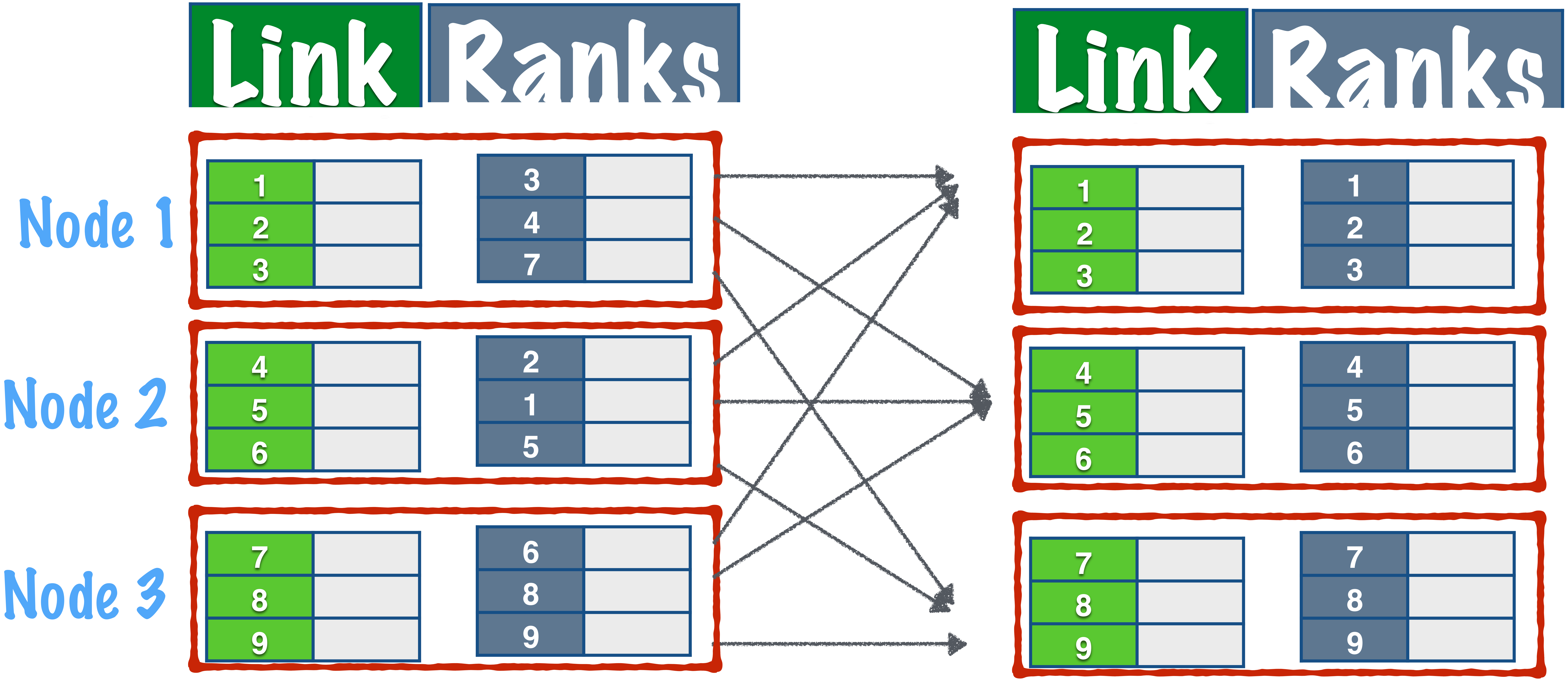
hash index for the keys

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

Custom Partitioning

```
partitionBy(new HashPartitioner(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

Recap

PageRank

The links RDD is reused many times

Step 4: Join the links and ranks RDDS

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

var ranks = links.mapValues(v => 1.0)
val iters = 2

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

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

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

var ranks = links.mapValues(v => 1.0)
val iters = 2

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

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

PageRank

Step 4: Join the links and ranks RDDS

```
val links = googleWeblinks.partitionBy(new HashPartitioner(100)).groupByKey.cache()
var ranks = links.mapValues(v => 1.0)
val iters = 2

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

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

PageRank

Step 4: Join the links and ranks RDDS

```
val links = googleWeblinks.partitionBy(new HashPartitioner(100)).groupByKey.cache()
```

Partitioning the links RDD will lead to a significant optimization in the execution of the PageRank algorithm

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