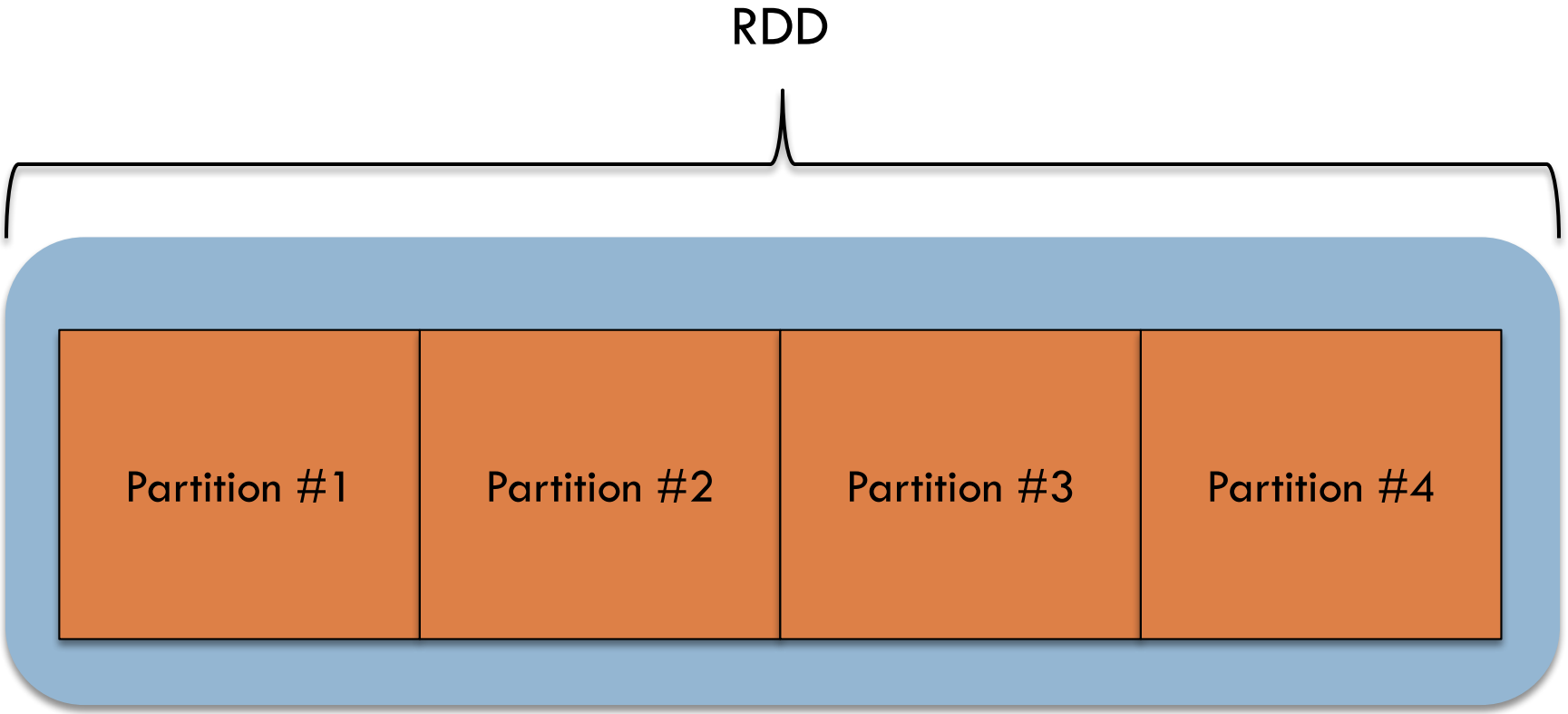
INTRODUCTION TO SPARK WITH SCALA

Apache Spark Programing Model

Agenda

- RDD Dependency
- RDD Data Partitioning
- RDD Persistence
- RDD Loading & Saving
- RDD Shared Variables
- PageRank in Spark

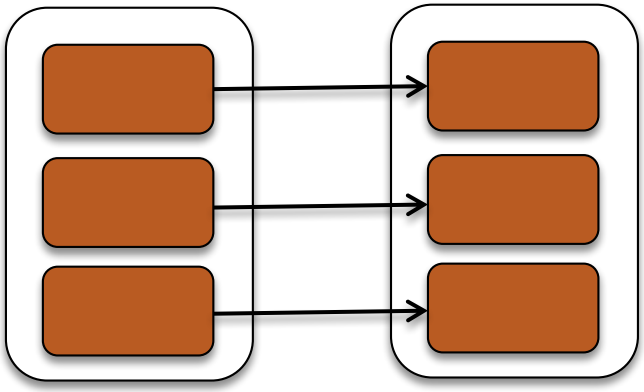
RDD Dependency



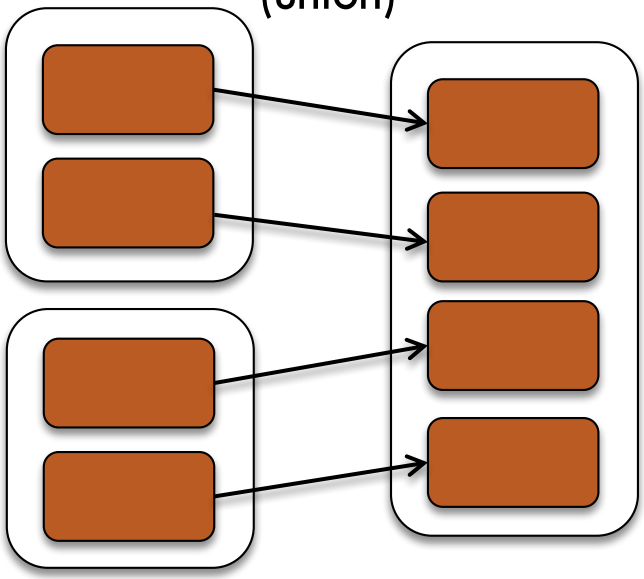
Immutable partitioned data set distributed across multiple nodes

RDD Dependency

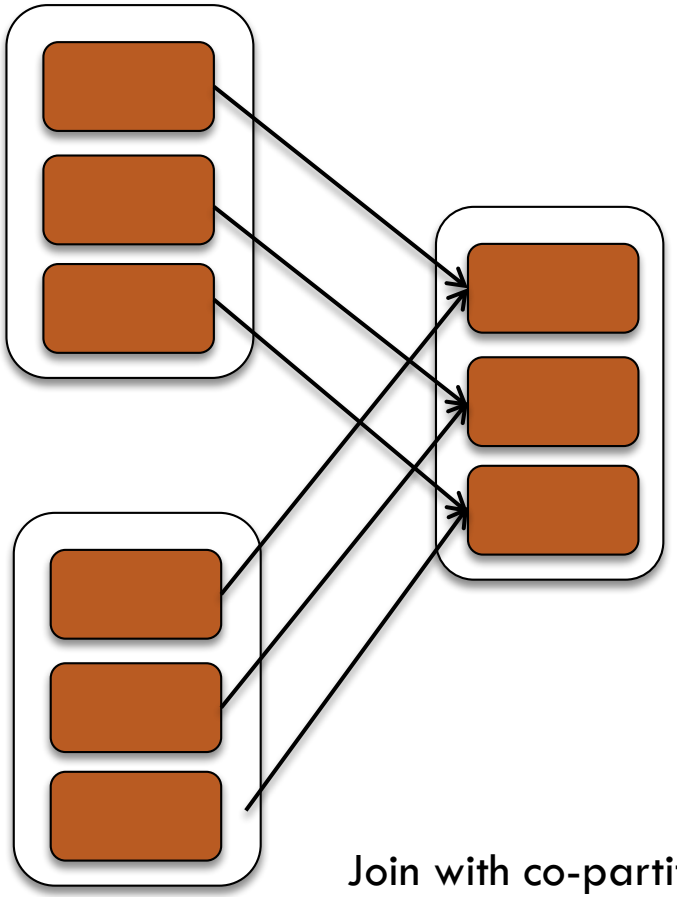
(map, filter)



(union)



Narrow Dependency

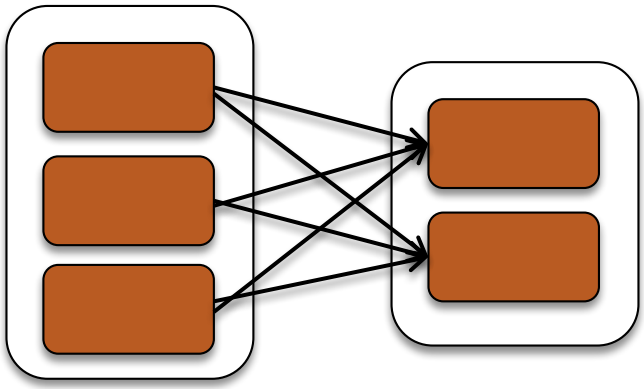


Join with co-partitioned

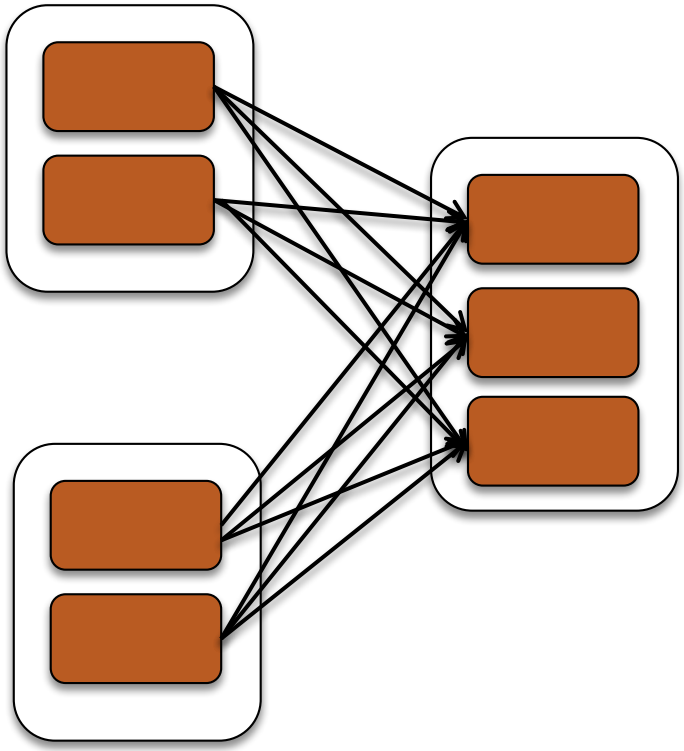
RDD Dependency

Wide Dependency

(groupByKey)



Multiple child RDD partitions may depend on a single parent RDD partition



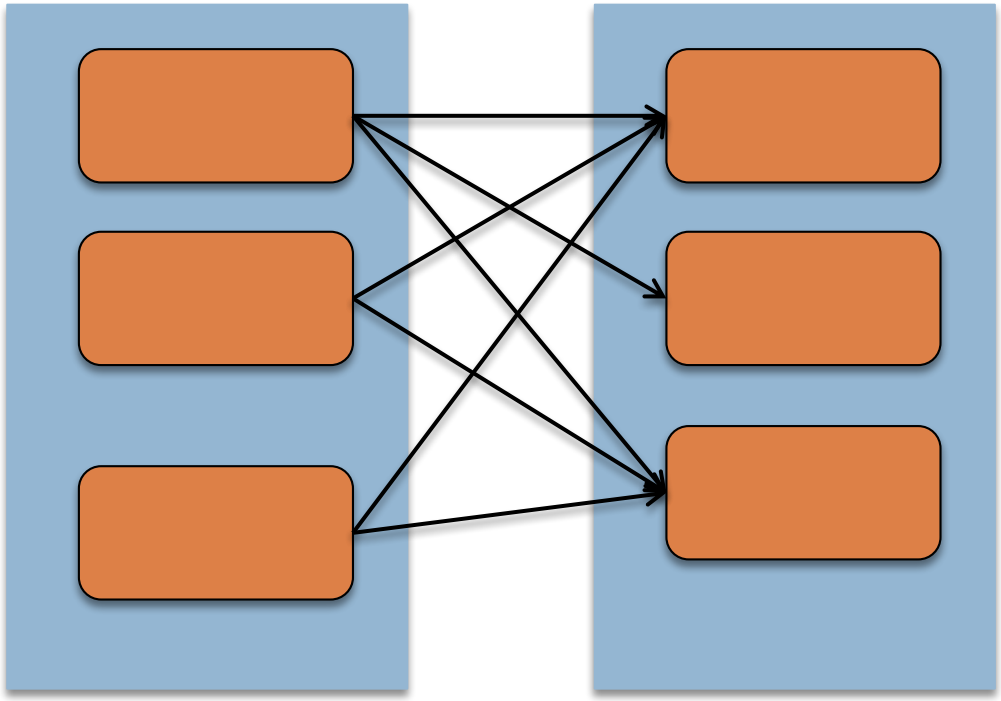
Join without co-partitioned

RDD Data Partitioning

- Partitioning
  - Control the degree of parallelism
  - Group elements based on a function of each key
  - Ensure a set of keys appear together on some node
- Minimize data movement for performance
- Applicable when RDD is reused multiple times

RDD Data Partitioning

Shuffling while reduceByKey



Expensive Operation

- Disk I/O
- Data serialization
- Network I/O

Physical movement of data between partitions

RDD Data Partitioning

- Use cases
  - In key/value pair operations to ensure a set of keys reside on some node
  - join, reduceByKey, groupByKey, cogroup, sortByKey
- Partition options
  - Hash partitioning
    - Hash(key) & numPartitions
  - Range partitioning
    - Partition data elements into roughly equal range

RDD Data Partitioning

Data Partition APIs

Transformation	Description
partitionBy(partitioner)	Return new RDD with specified number of partitions partition by given partitioner
coalesce(numPartitions)	Decrease number of partitions
repartition(numPartitions)	Reshuffle data randomly to create either more or fewer partitions and balance across them

RDD Data Partitioning

partitionBy(partitioner) –

- New RDD with specified # of partitions and partition data elements based on given partitioner

```
val rdd1  = sc.texFile("<path>")
Val keyValueRdd = rdd1 .. Convert to key value pair RDD

val partitionedRDD = keyValueRdd.partitionBy(new
HashPartitioner(20))

// to find out # of partitions

partitionedRDD.partitions.length  // 20
partitionedRDD.partitioner        // HashPartitioner
```

RDD Data Partitioning

coalesce(numPartitions) –

- Reduce number of partitions down to given partitions w/o shuffling
- Useful after filtering down a large dataset

```
import scala.util.Random
val randNums = Seq.fill(100000)(Random.nextInt(80000))
val numberCounts = sc.parallelize(randNums, 10)

val oddNumbers = numberCounts.filter(n => n % 2 == 1)

val rdd2 = oddNumbers.coalesce(3)
rdd2.partitions.length
```

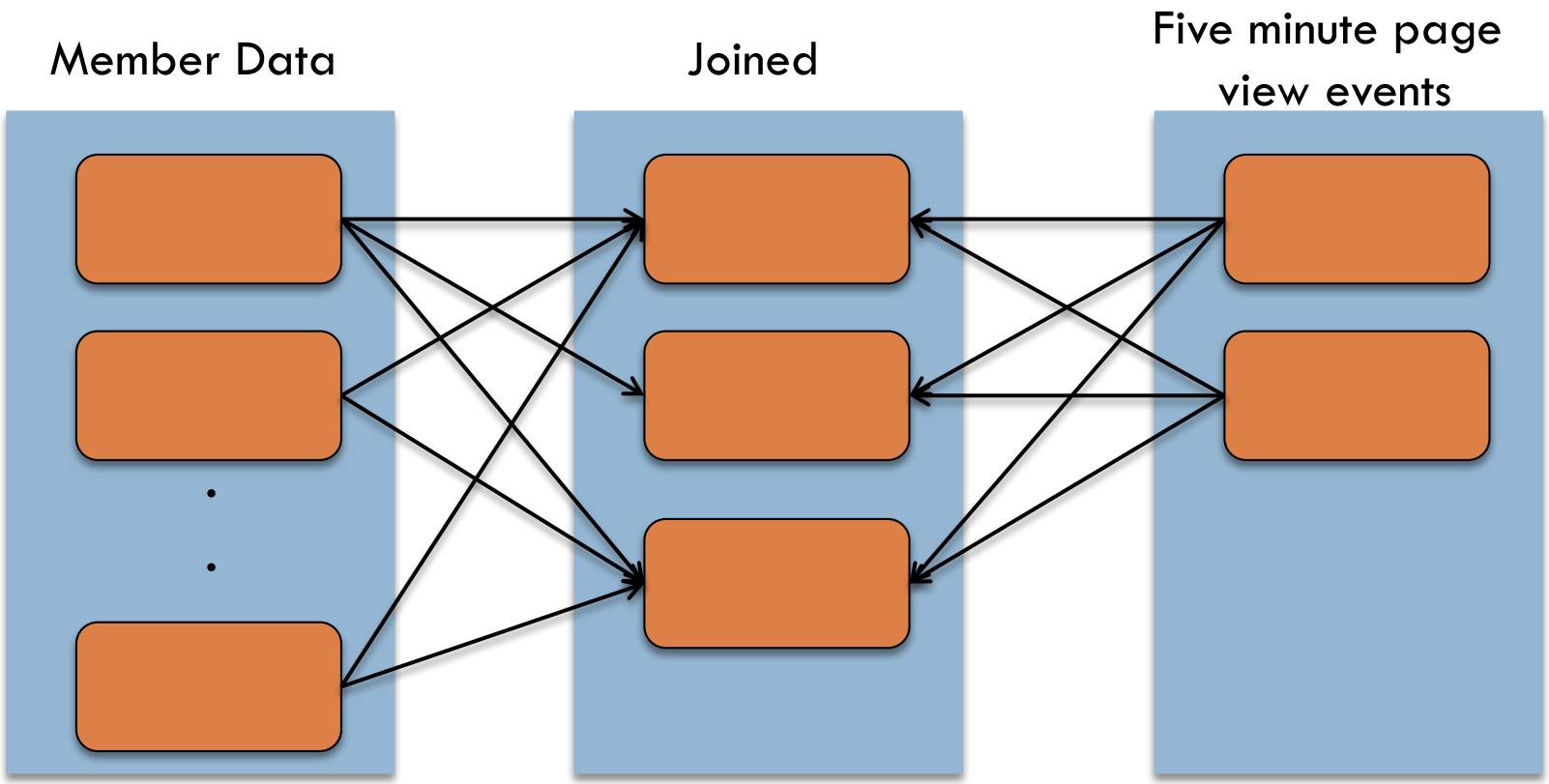
RDD Data Partitioning

repartition(numPartitions) –

- Reshuffle data randomly to create more or fewer partitions

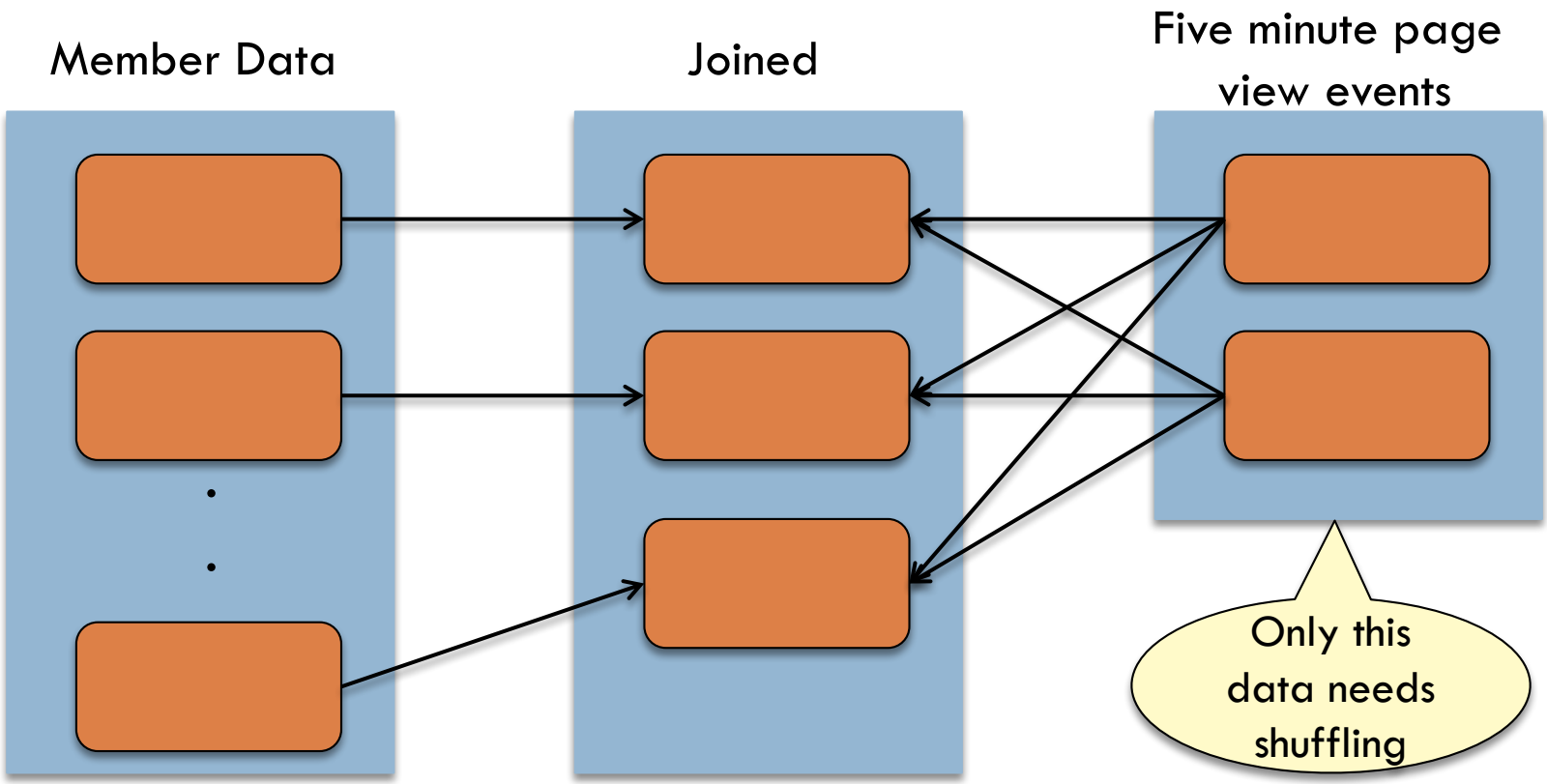
```
val rdd1  = sc.parallelize(1 to 15 by 3, 3)
rdd1.glom().collect

// Array(Array(1), Array(4, 7), Array(10, 13))

val rdd2 = rdd1.repartition(2)
rdd2.glom().collect

//Array(Array(1, 4, 10), Array(7, 13))
```

RDD Data Partitioning



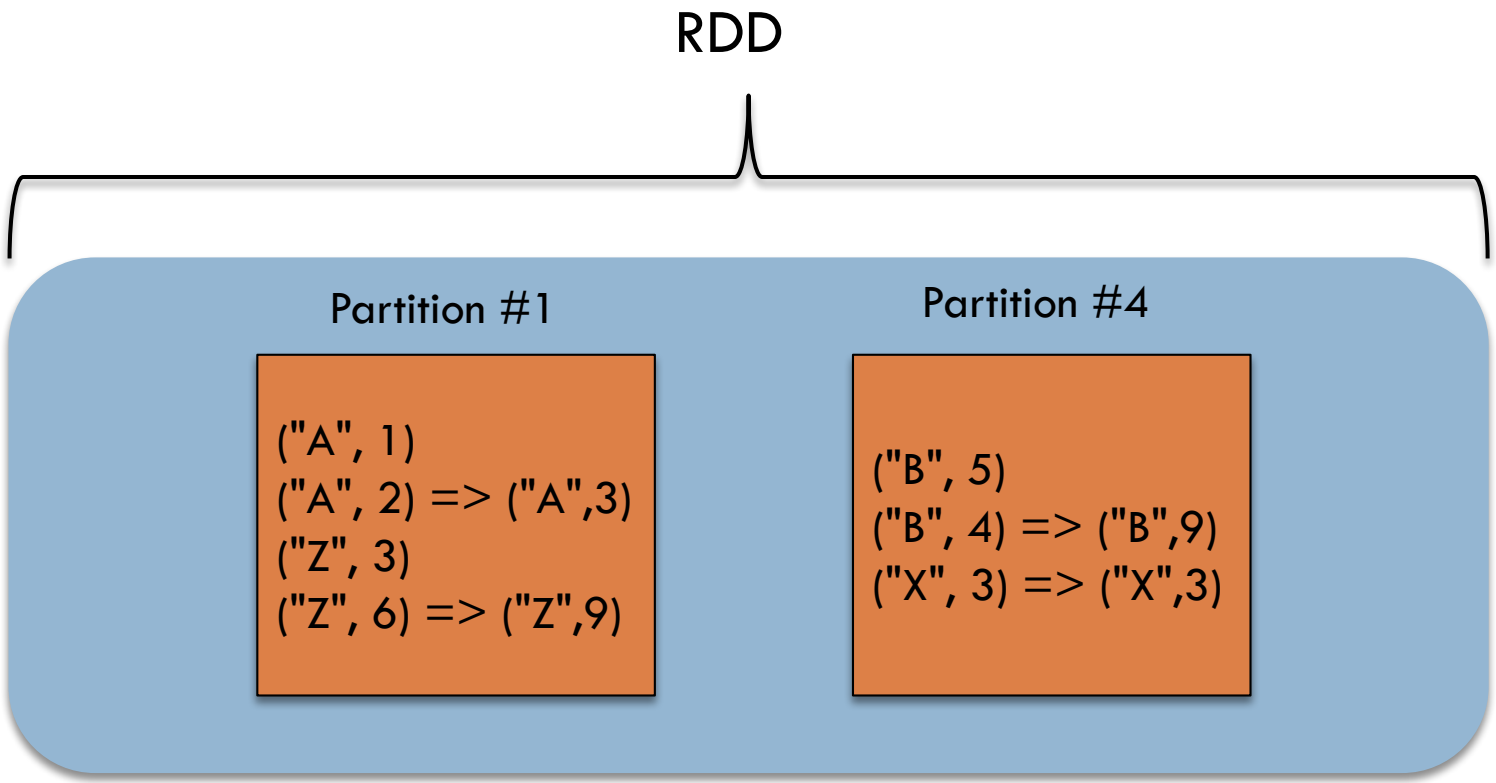

```
val memberData = sc.textFile("<path>")
val pageViewData = sc.textFile("<path>")

memberData.join(pageViewData)
```

RDD Data Partitioning




```
val memberData =
sc.textFile("<path>").partitionBy(..).persist()
val pageViewData = sc.textFile("<path>")
memberData.join(pageViewData)
```

RDD Data Partitioning




```
rdd.reduceByKey(_ + _ ).collect()

// ("A",3),("Z",9),("B",9),("X",3)
```

RDD Persistence

- RDD Persistence/caching
  - Important and unique feature in Spark
  - Persisting dataset in memory across operations
  - Intermediate result of a computation – reuse
  - Great for interactive use case or iterative algorithms
  - Fault-tolerant caching
  - Caching storage options
    - Tradeoff between memory usage and CPU usage
    - Memory, disk
    - Serialized vs deserialized objects
  - LRU eviction policy

RDD Persistence

Caching Option	Description
MEMORY_ONLY (default)	Store as Java deserialized Java objects. Recompute partitions that don't fit in memory on the fly
MEMORY_AND_DISK	Store as Java deserialized Java objects. Spill onto disk for don't fit partitions
MEMORY_ONLY_SER	Store as serialized Java objects. Space efficient but more CPU-intensive
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill out to disk for don't fit partitions
DISK_ONLY	Store only on disk
MEMORY_ONLY2 MEMORY_AND_DISK	Same ask above, but replicate each partition on two nodes
OFF_HEAP	Store in serialized format on Tachyon

RDD Persistence

Caching Option Tradeoffs

Caching Option	Space Used	CPU Time	In Memory	On Disk
MEMORY_ONLY (default)	High	Low	Y	N
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK_SER	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

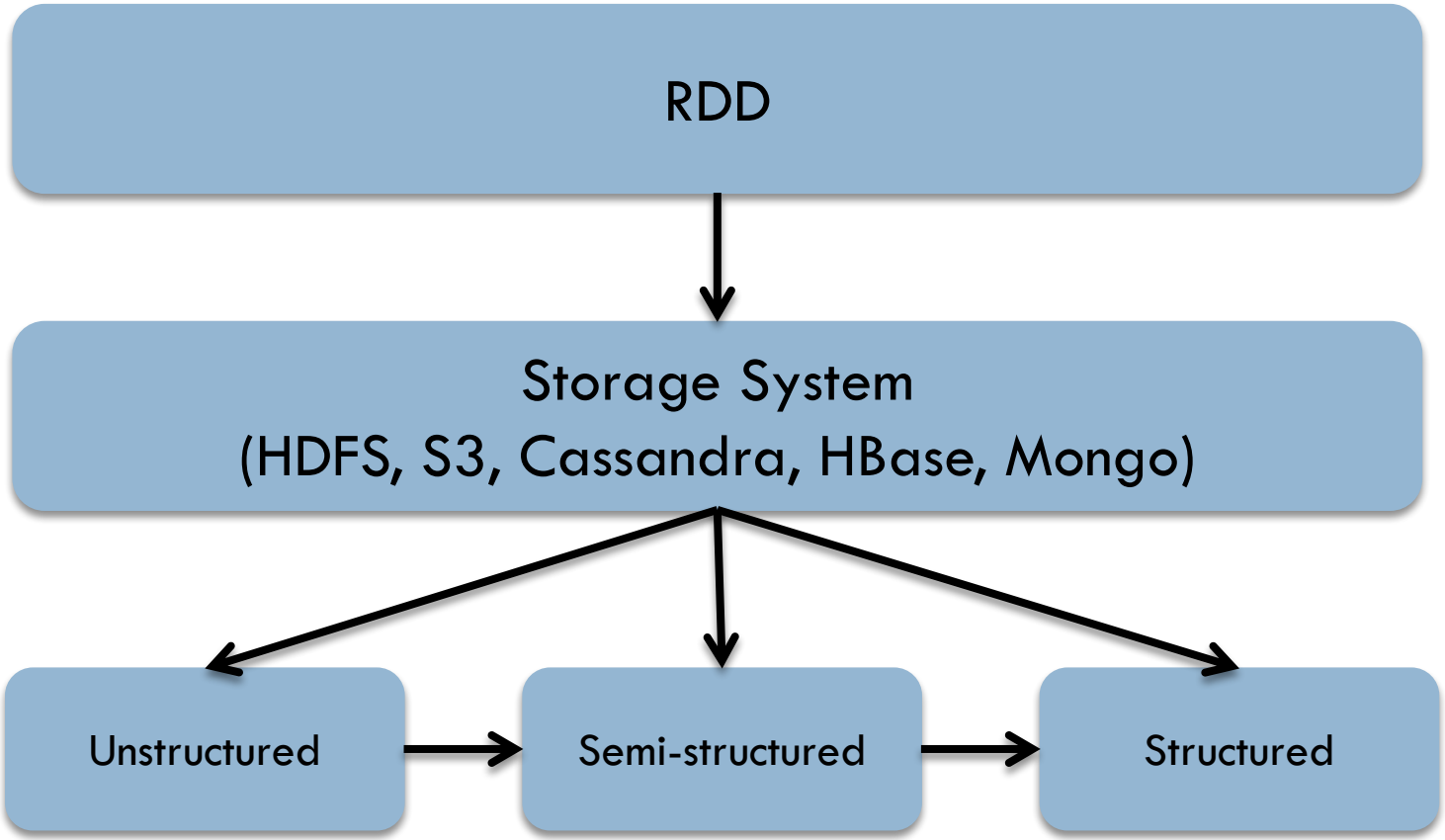
RDD Persistence

persist(storageLevel) – persist in cache

unpersist() – remove from cache

```
// loading data file
val logs = sc.textFile("hdfs://service-log/")

val errors = logs.filter(_.contains("ERROR"))
errors.persist()

// figure out what kind of errors
errors.filter(_.contains("SQLException")).count

// should be faster at this point
errors.filter(_.contains("RemoteException")).count

// to evict RDD from cache
// errors.unpersist()
```

RDD Loading & Saving



RDD Loading & Saving

File Formats

File Format	Structured	Description
Text	No	Records are assumed to be one per line
JSON	Semi	Common text-based format
CSV	Yes	Popular text-based format
Protocol Buffers	Yes	A fast space efficient cross language format
Avro	Yes	A fast, space efficient, schema based format
Parquet	Yes	A columnar storage format
ORC	Yes	A columnar storage format

RDD Loading & Saving

textFile(uri, numPartitions) –

- Lazily loading data set

```
// load a specific file
val rdd1  = sc.textFile("/data/part1.txt")
val rdd1  = sc.textFile("file:///data/part1.txt")

// load all files in /data
val rdd2  = sc.textFile("file:///data")

// load all files end with ".txt" in /data on HDFS
val rdd3  = sc.textFile("hdfs://data/*.txt")


// load all compressed files end with ".tgz" in /data
val rdd4  = sc.textFile("/data/*.gz")
```

RDD Loading & Saving

wholeTextFiles(uri, numPartitions) –

- Read files in a directory
- Return tuples with (file name, content)

```
val rdd1  = sc.wholeTextFiles("/data")

val fileNames = rdd1.map({
   case(fn, content) => (fn, content.split(" ").length)
})

// file name with # of words in each file
fileNames.collect()
```

RDD Loading & Saving

saveAsTextFile(path) –

- Save RDD as text file(s) to given path
- Path shouldn't exist
- One part file per partition

```
// load a specific file
val rdd1  = sc.texFile("file:///data/part1.txt")

val rdd2 = rdd1.filter(line => line.contains("ERROR"))

// rdd2.saveAsTextFile("/data-with-error)")

rdd2.saveAsTextFile("file:///data-with-error)")
```

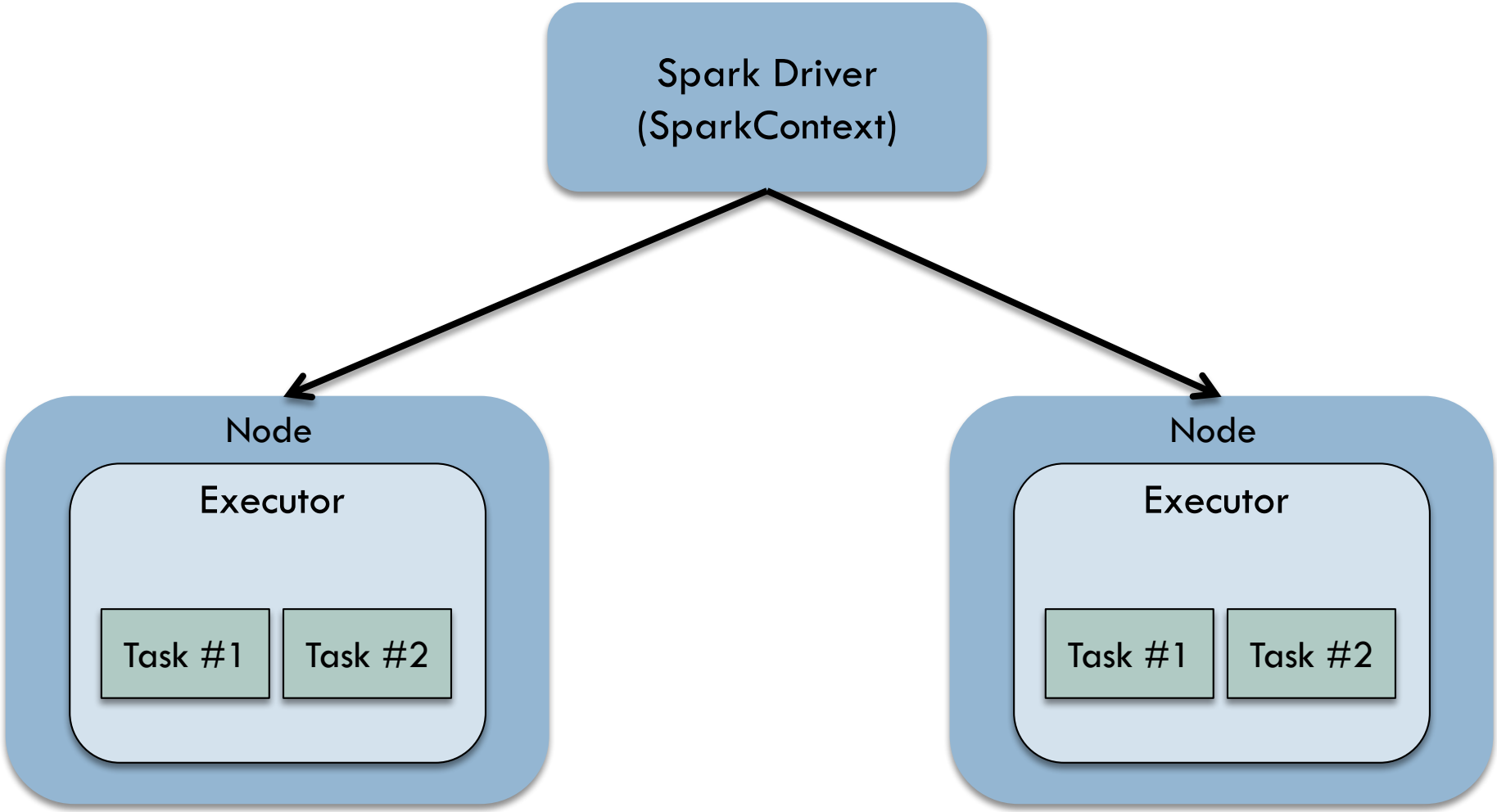
RDD Loading & Saving

- JSON Support
  - Better in SparkSQL
- JDBC
  - Through JdbcRDD
- Cassandra
  - Datastax Cassandra connector
- HBase
  - SparkContext.newAPIHadoopRDD()
  - TableInputFormat
- ElasticSearch
  - ElastciSearch InputFormat

RDD Shared Variables

- Accumulators – distributed counters
  - Aggregate values from workers
  - Can only “add” to using an associative operation
  - Only driver can read the values
  - Can be used to implement counters like in MapReduce
  - Generally used for debugging purposes only
  - Out of the box support
    - Int, Double, Long, Float
  - Custom accumulator is supported

RDD Shared Variables



RDD Shared Variables

Regular Variable Example

```
// load a specific file
val rdd1  = sc.texFile("file:///data/part1.txt")

// create a variable
var badLines = 0;

// logic below is executed on each of the workers
val rdd2 = rdd1.flatMap(line => {
   if (line == "") {
     badLines += 1
   }
   line.split(" ")
})
rdd2.saveAsText("file:///output")
println("Bad lines: " + badLines)
```

RDD Shared Variables

Accumulator Example

```
// load a specific file
val rdd1  = sc.texFile("file:///data/part1.txt")

// create an accumulator
val badLines = sc.accumulator(0)

// logic below is executed on each of the workers
val rdd2 = rdd1.flatMap(line => {
   if (line == "") {
     badLines += 1
   }
   line.split(" ")
})

rdd2.saveAsText("file:///output")  // needed?
println("Bad lines: " + badLines.value)
```

RDD Shared Variables

- Broadcast Variables
  - Efficiently share large, read-only value to worker nodes
  - Efficient broadcast algorithm to reduce network costs
  - When multiple tasks across stages need same data
  - Read-only lookup table
    - Large feature vector
  - Efficiently join when one of dataset is small

RDD Shared Variables

Broadcast Variables Example

```
// load a specific file
val rdd1  = sc.texFile("file:///data/part1.txt")
val countryMap = loadContryMap(…)

val countryMapBC = sc.broadcast(countryMap)

// countryMapBC will be broadcasted to each nodes
val rdd2 = rdd1.map ({line =>
  val row = line.split(",")
  (row(0), countryMapBC.value.get(row(1)))
)

rdd2.saveAsText("file:///output")
```

PageRank in Spark

- Showing off the power of Spark
- Iterative application
  - Join and shuffling
- Partitioning to reduce shuffling

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

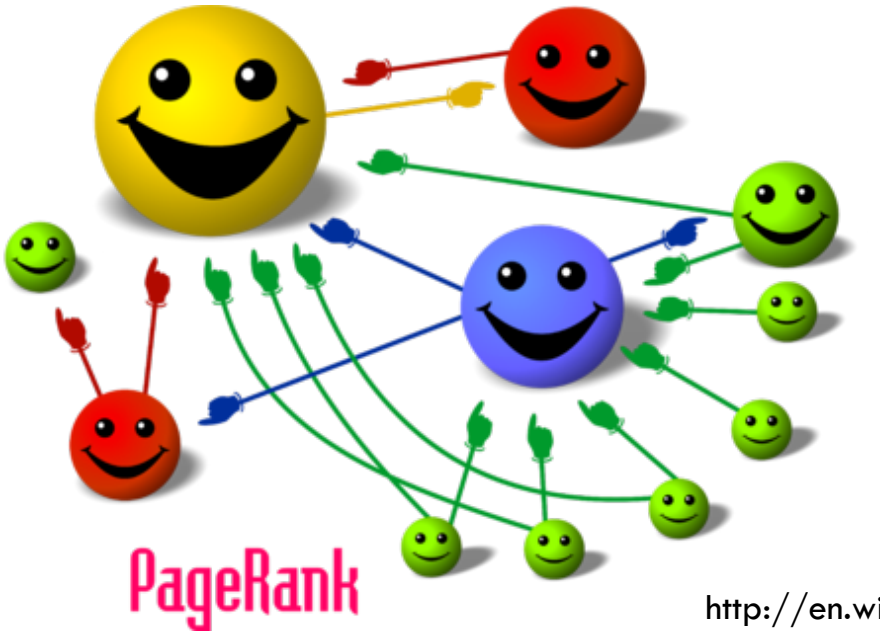
PageRank Algorithm

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

- Wikipedia (https://en.wikipedia.org/wiki/PageRank)

PageRank in Spark

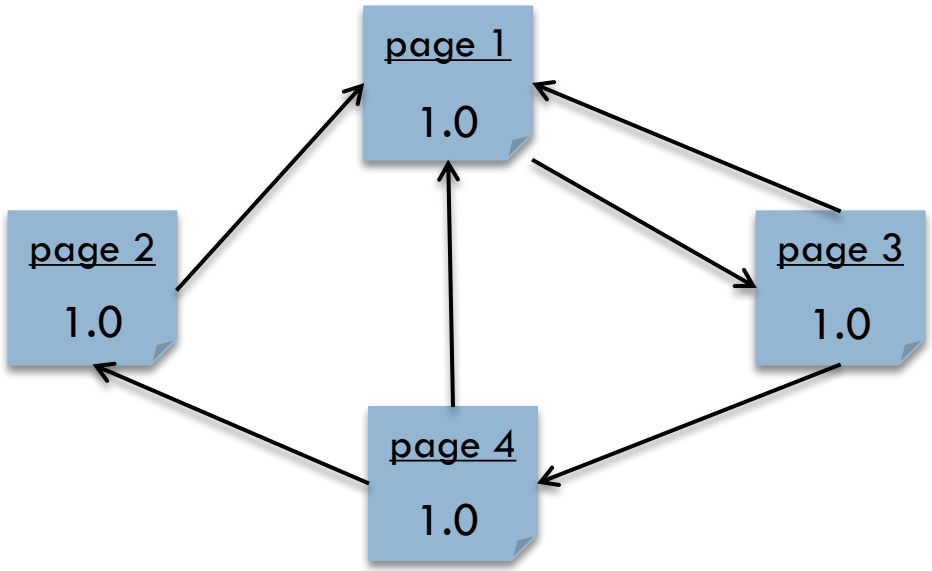
- PageRank Algorithm
  - Give page ranking scores based on # links to them
  - Links from many pages scores higher rank
  - Link from a high-rank page boosts rank higher



http://en.wikipedia.org/wiki/Page_rank

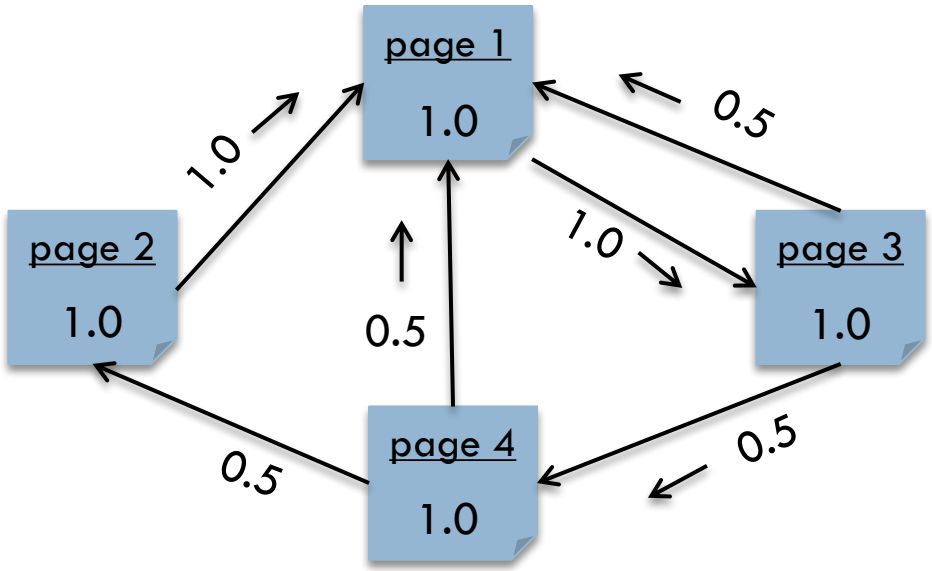
PageRank in Spark

- Algorithms
  - Start each page at a rank of 1
  - On each iteration, have page p contribute
    - $rank_p$ / $neighbors_p$
  - Set each page's rank to 0.15 + 0.85 * contribs



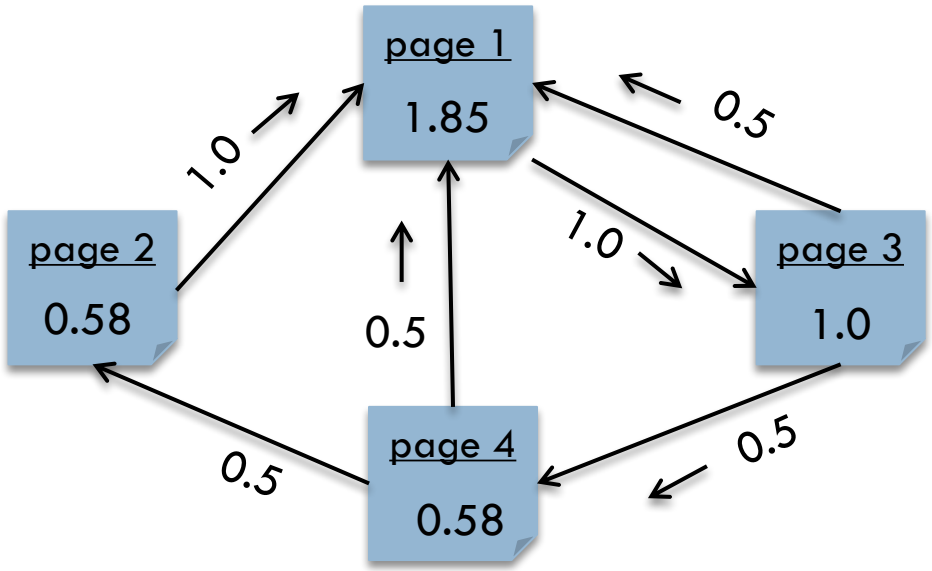
PageRank in Spark

- Algorithms
  - On each iteration, have page p contribute
    - $contrib_p = rank_p / neighbors_p$
  - Set each page's rank to 0.15 + 0.85 * contribs



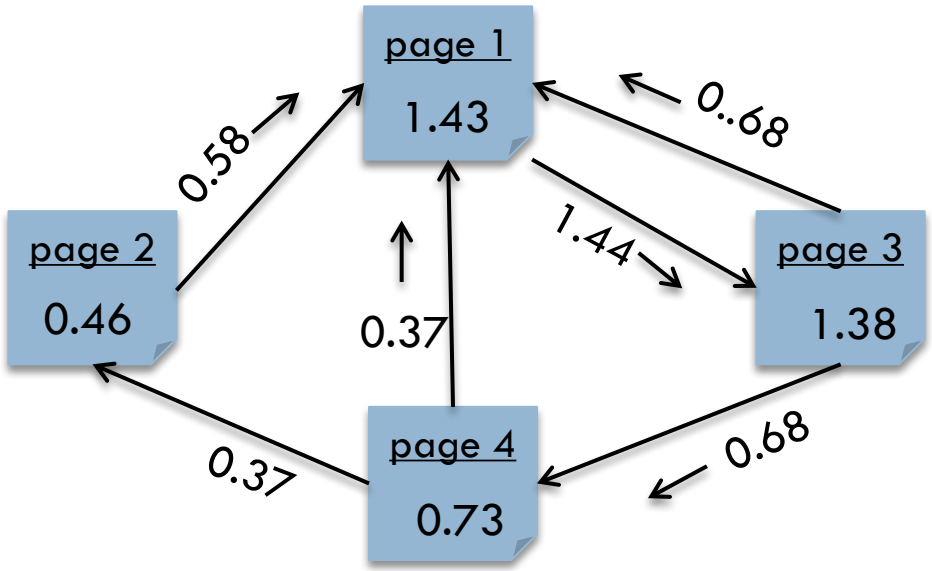
PageRank in Spark

- Algorithms
  - On each iteration, have page p contribute
    - $contrib_p = rank_p / neighbors_p$
  - Set each page's rank to 0.15 + 0.85 * contribs



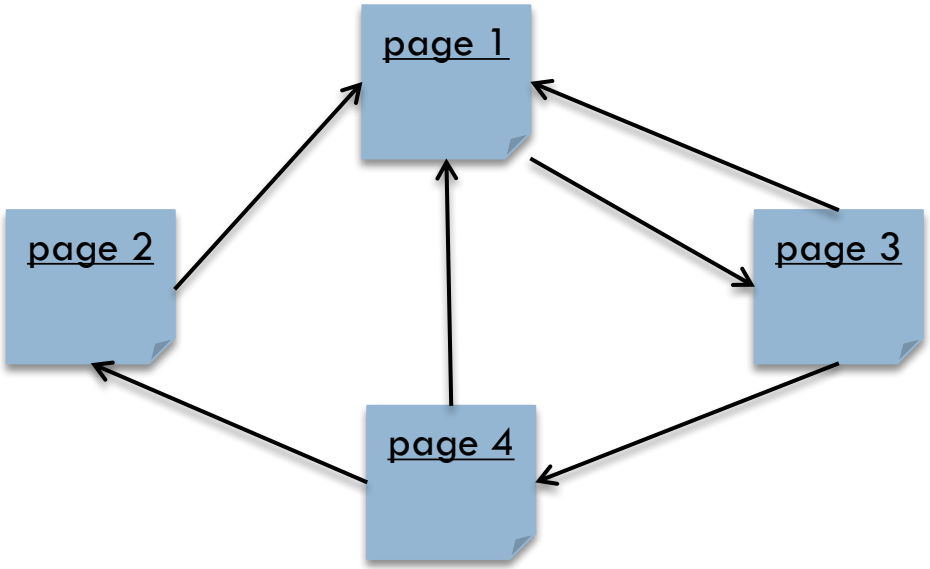
PageRank in Spark

- Algorithms
  - On each iteration, have page p contribute
    - $contrib_p = rank_p / neighbors_p$
  - Set each page's rank to 0.15 + 0.85 * contribs



PageRank in Spark

- Data format
  - { source, destination }



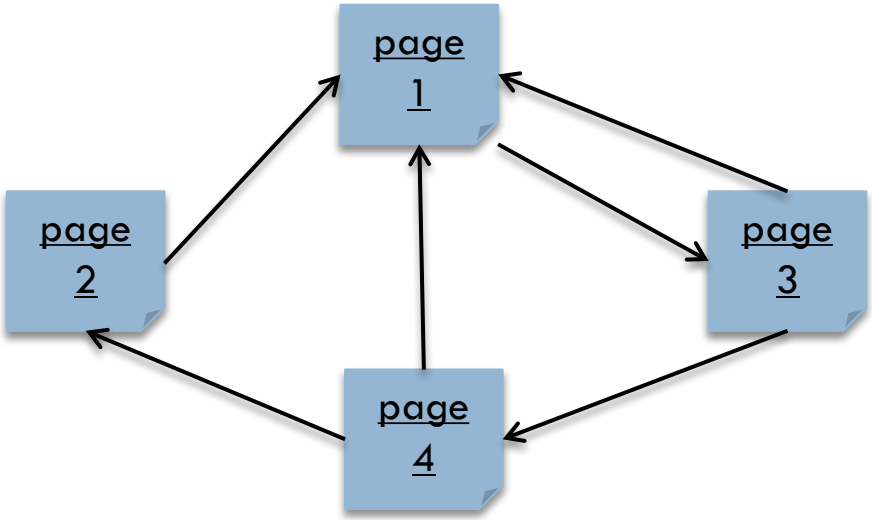
page1, page3
page2, page1
page3, page1
page3, page4
page4, page1
page4, page2

PageRank in Spark

Iterative Algorithm Example

```
val lines = sc.textFile("<path>")
val links = lines.map{ s =>
      val parts = s.split(",")
      (parts(0), parts(1))
    }.distinct().groupByKey().cache()
```

page1, page3
page2, page1
page3, page1
page3, page4
page4, page1
page4, page2



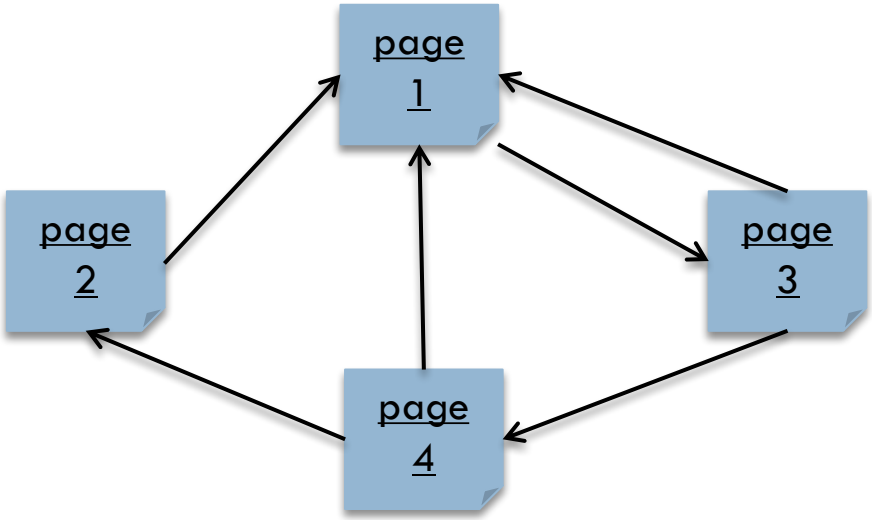
(page1, [page3])
(page2, [page1])
(page3, [page1, page4])
(page4, [page2, page1])

PageRank in Spark

Iterative Algorithm Example

```
val lines = sc.textFile("<path>")
val links = lines.map( s => {
      val parts = s.split(",")
      (parts(0), parts(1))
    }).distinct().groupByKey().cache()
var ranks = links.mapValues(v => 1.0)
```

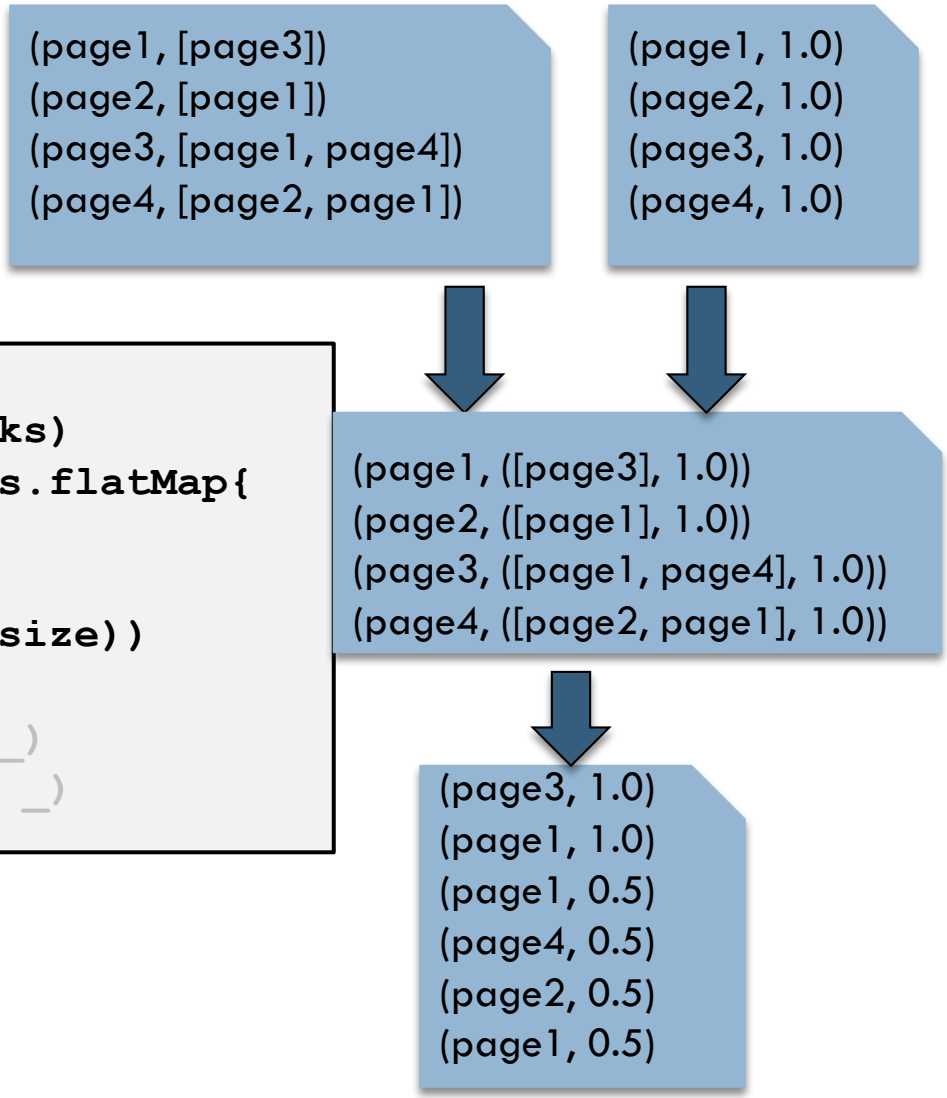
(page1, [page3])
(page2, [page1])
(page3, [page1, page4])
(page4, [page2, page1])



(page1, 1.0)
(page2, 1.0)
(page3, 1.0)
(page4, 1.0)

PageRank in Spark


```
for (i <- 1 to 10) {
  val linkWithRank = links.join(ranks)
  val contribs = linkWithRank.values.flatMap{
    case (urls, rank) =>
      val size = urls.size
      urls.map(url => (url, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _)
           .mapValues(0.15 + 0.85 * _)
}
```



PageRank in Spark

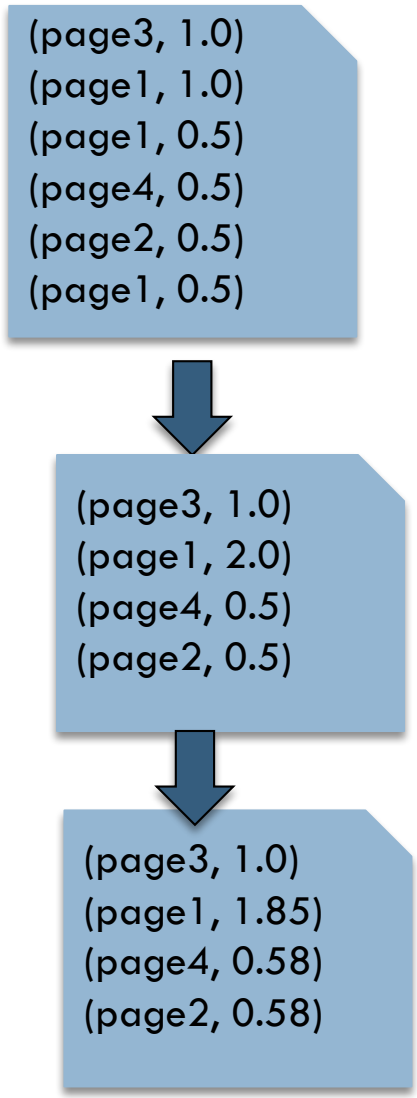

```
for (i <- 1 to 10) {
  val linkWithRank = links.join(ranks)
  val contribs = linkWithRank.values.flatMap{
    case (urls, rank) =>
      val size = urls.size
      urls.map(url => (url, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _)
           .mapValues(0.15 + 0.85 * _)
}
```

(page3, 1.0)
(page1, 1.0)
(page1, 0.5)
(page4, 0.5)
(page2, 0.5)
(page1, 0.5)

(page3, 1.0)
(page1, 2.0)
(page4, 0.5)
(page2, 0.5)

PageRank in Spark


```
for (i <- 1 to 10) {
  val linkWithRank = links.join(ranks)
  val contribs = linkWithRank.values.flatMap{
    case (urls, rank) =>
      val size = urls.size
      urls.map(url => (url, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _)
           .mapValues(0.15 + 0.85 * _)
}
```



PageRank in Spark

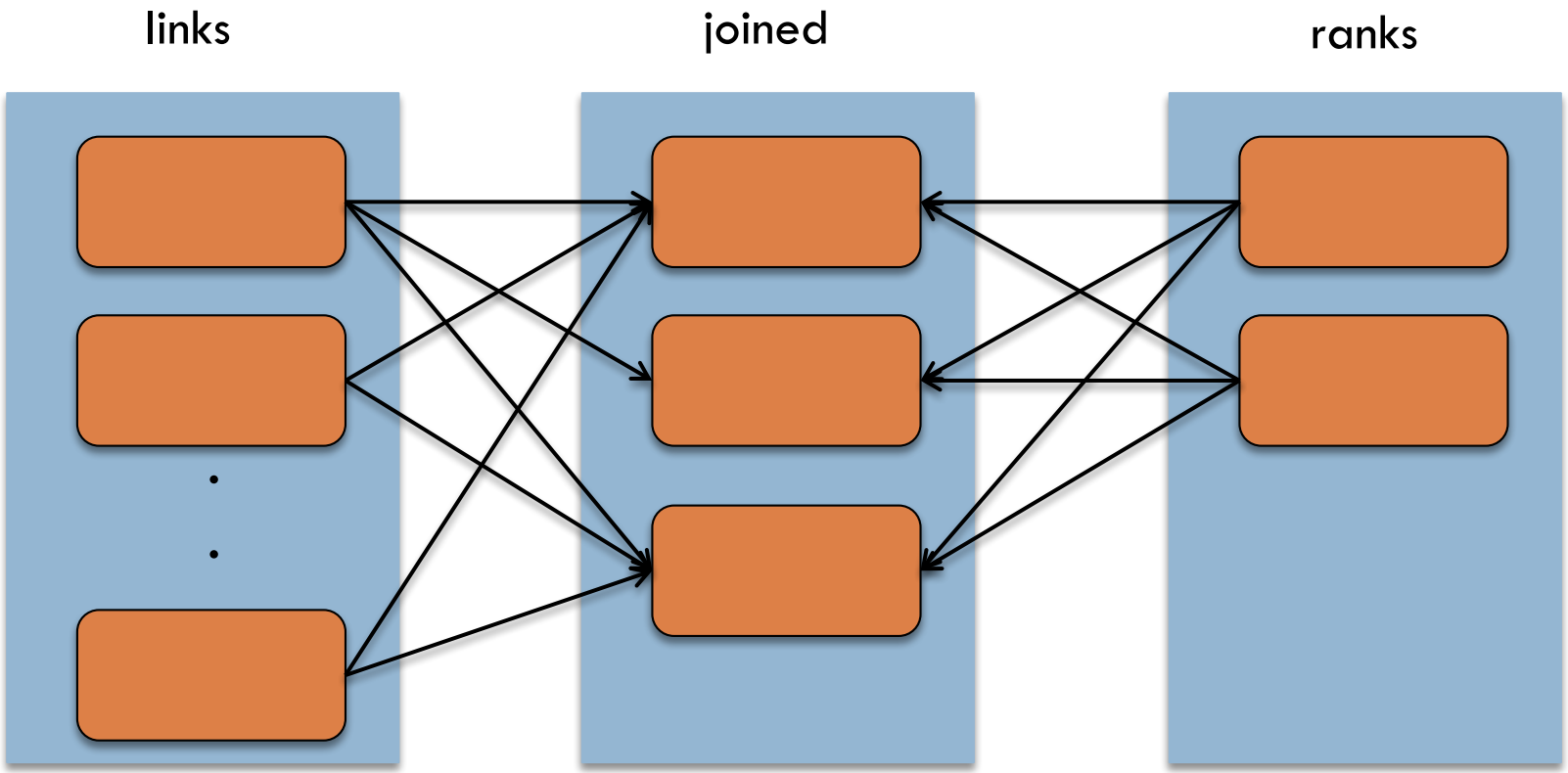
Iterative Algorithm Example

```
val lines = sc.textFile("<path>")
val links = lines.map{ s =>
      val parts = s.split("\\s+")
      (parts(0), parts(1))
    }.distinct().groupByKey().cache()
var ranks = links.mapValues(v => 1.0)   // (url, rank)
for (i <- 1 to 10) {
  val linkWithRank = links.join(ranks)
  val contribs = linkWithRank.values.flatMap{
    case (urls, rank) =>
      val size = urls.size
      urls.map(url => (url, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
val output = ranks.collect()
output.foreach(rank => println(ranks._1 + " has rank " + ranks._2))
```

https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/SparkPageRank.scala

PageRank in Spark

- Each iteration requires a join - links and ranks
- Each join requires shuffling data over the network



PageRank in Spark


```
val lines = sc.textFile("<path>")
val links = lines.map(...).partitionBy(new HashPartitioner(8))
```