Numerical Computing with Spark

Hossein Falaki

DATABRICKS

Challenges of numerical computation over big data

When applying any algorithm to big data watch for

1. Correctness
2. Performance
3. Trade-off between accuracy and performance

Three Practical Examples

- Point estimation (Variance)
- Approximate estimation (Cardinality)
- Matrix operations (PageRank)

We use these examples to demonstrate Spark internals, data flow, and challenges of implementing algorithms for Big Data.

1. Big Data Variance

- The plain variance formula requires two passes over data

$$Var(X) = \overbrace{\frac{1}{N}\sum_{i=1}^{N}(x_i - \underbrace{\mu}_{\text{First pass}})^2}^{\text{Second pass}}$$

Fast but inaccurate solution

$$Var(X) = E[X^2] - E[X]^2$$

$$= \frac{\sum x^2}{N} - \left(\frac{\sum x}{N}\right)^2$$

- [x] Can be performed in a single pass, but
- [ ] Subtracts two very close and large numbers!

Accumulator Pattern

An object that incrementally tracks the variance

```
Class RunningVar {
  var variance: Double = 0.0

  // Compute initial variance for numbers
  def this(numbers: Iterator[Double]) {
    numbers.foreach(this.add(_))
  }

  // Update variance for a single value
  def add(value: Double) {
    ...
  }
}
```

Parallelize for performance

- Distribute adding values in map phase
- Merge partial results in reduce phase

```
Class RunningVar {
  ...
  // Merge another RunningVar object
  // and update variance
  def merge(other: RunningVar) = {
    ...
  }
}
```

Computing Variance in Spark

- Use the RunningVar in Spark

```
doubleRDD
   .mapPartitions(v => Iterator(new RunningVar(v)))
   .reduce((a, b) => a.merge(b))
```

- Or simply use the Spark API

```
doubleRDD.variance()
```

2. Approximate Estimations

- Often an approximate estimate is *good enough* especially if it can be computed faster or cheaper

    1. Trade accuracy with memory

    2. Trade accuracy with running time

- We really like the cases where there is a bound on error that can be controlled

Cardinality Problem

Example: Count number of unique words in Shakespeare's work.

- Using a HashSet requires ~10GB of memory
- This can be much worse in many real world applications involving large strings, such as counting web visitors

Linear Probabilistic Counting

1. Allocate a bitmap of size m and initialize to zero.
   A. Hash each value to a position in the bitmap
   B. Set corresponding bit to 1
2. Count number of empty bit entries: v

$$count \approx -m \ln \frac{v}{m}$$

The Spark API

- Use the LogLinearCounter in Spark

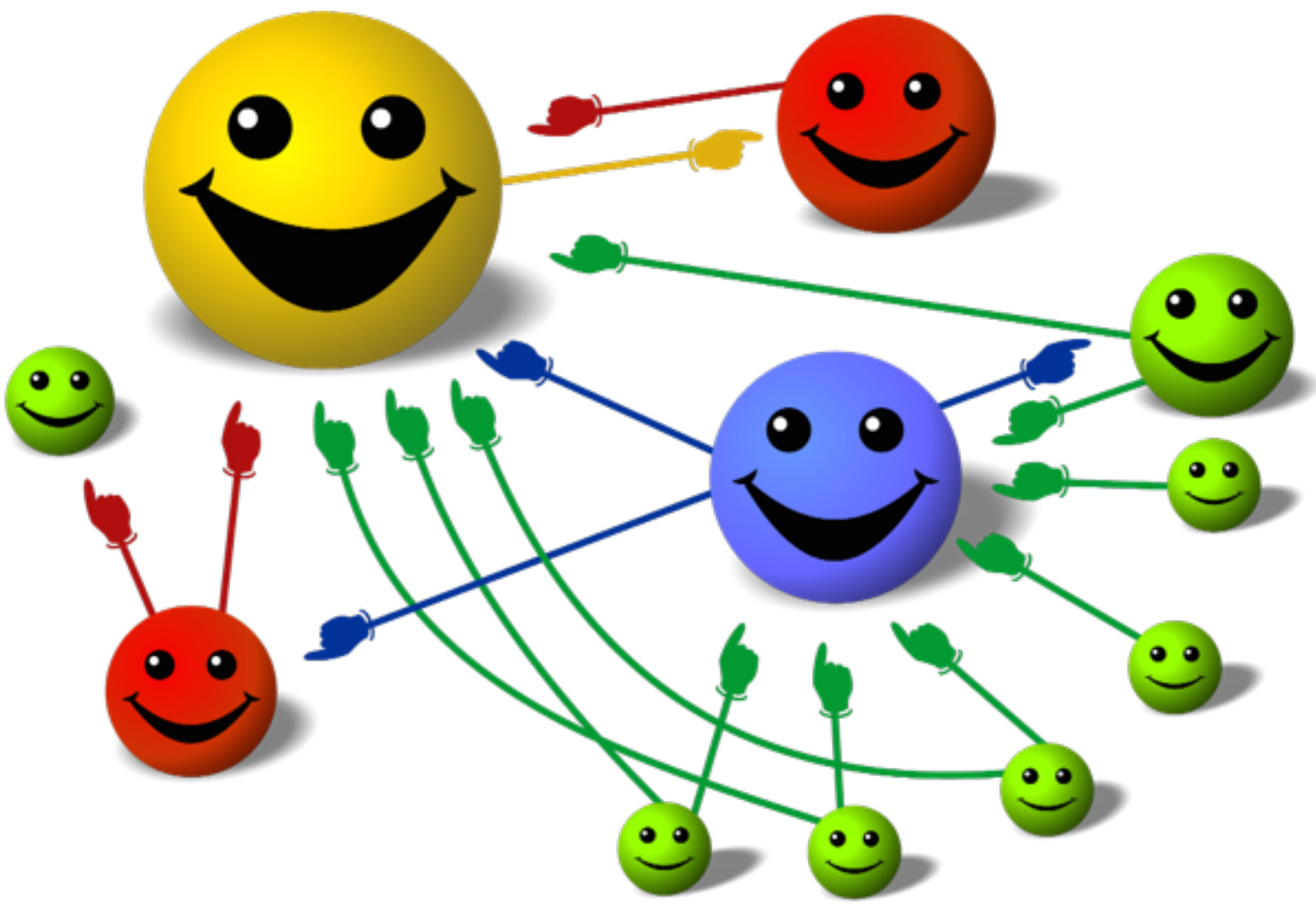
```
rdd
  .mapPartitions(v => Iterator(new LPCounter(v)))
  .reduce((a, b) => a.merge(b)).getCardinality
```

- Or simply use the Spark API

```
myRDD.countApproxDistinct(0.01)
```

3. Google PageRank

Popular algorithm originally introduced by Google



PageRank Algorithm

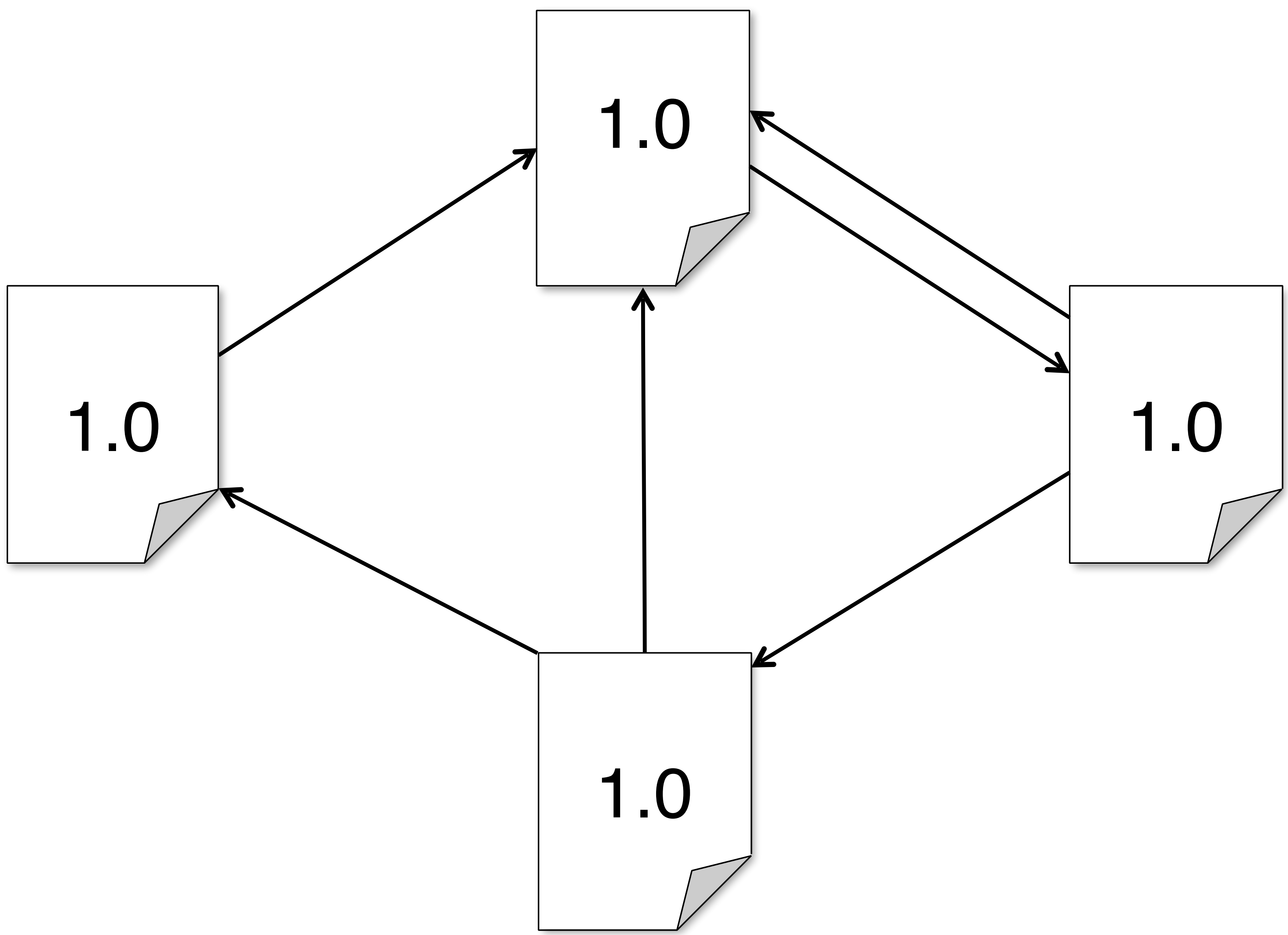
PageRank Algorithm

- Start each page with a rank of 1
- On each iteration:

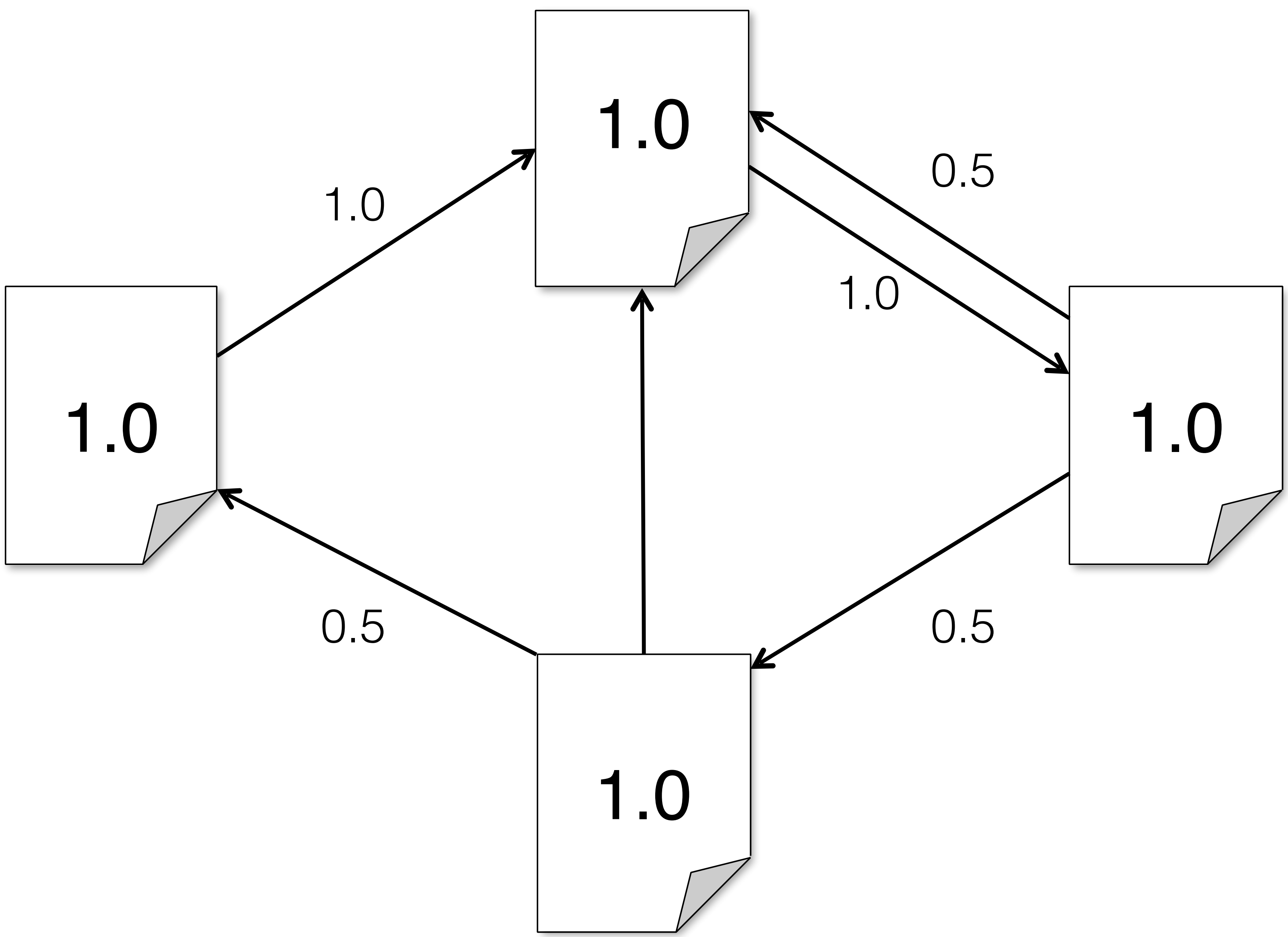
A. $contrib = \frac{curRank}{|neighbors|}$

B. $curRank = 0.15 + 0.85 \sum_{neighbors} contrib_i$

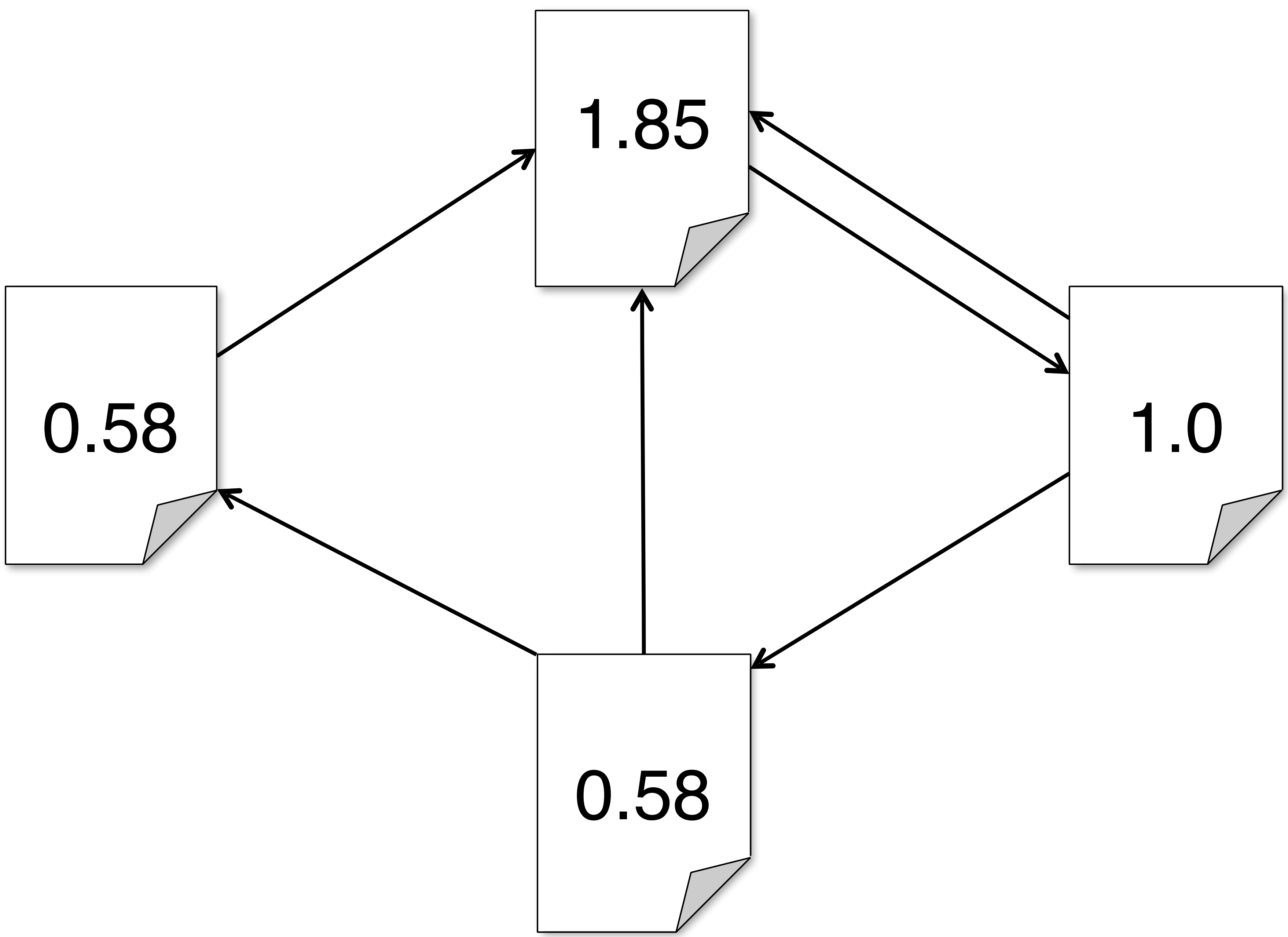
PageRank Example



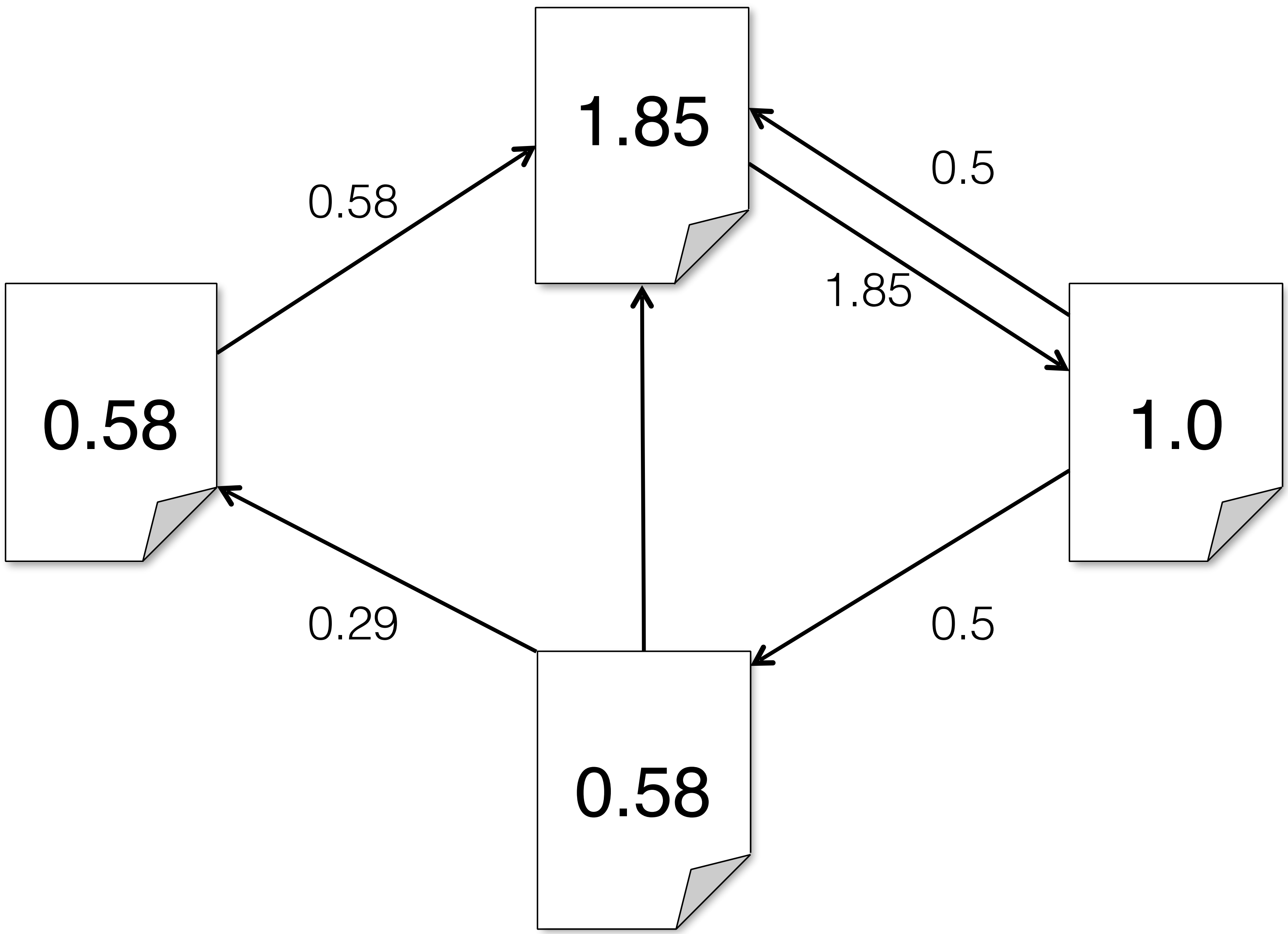
PageRank Example



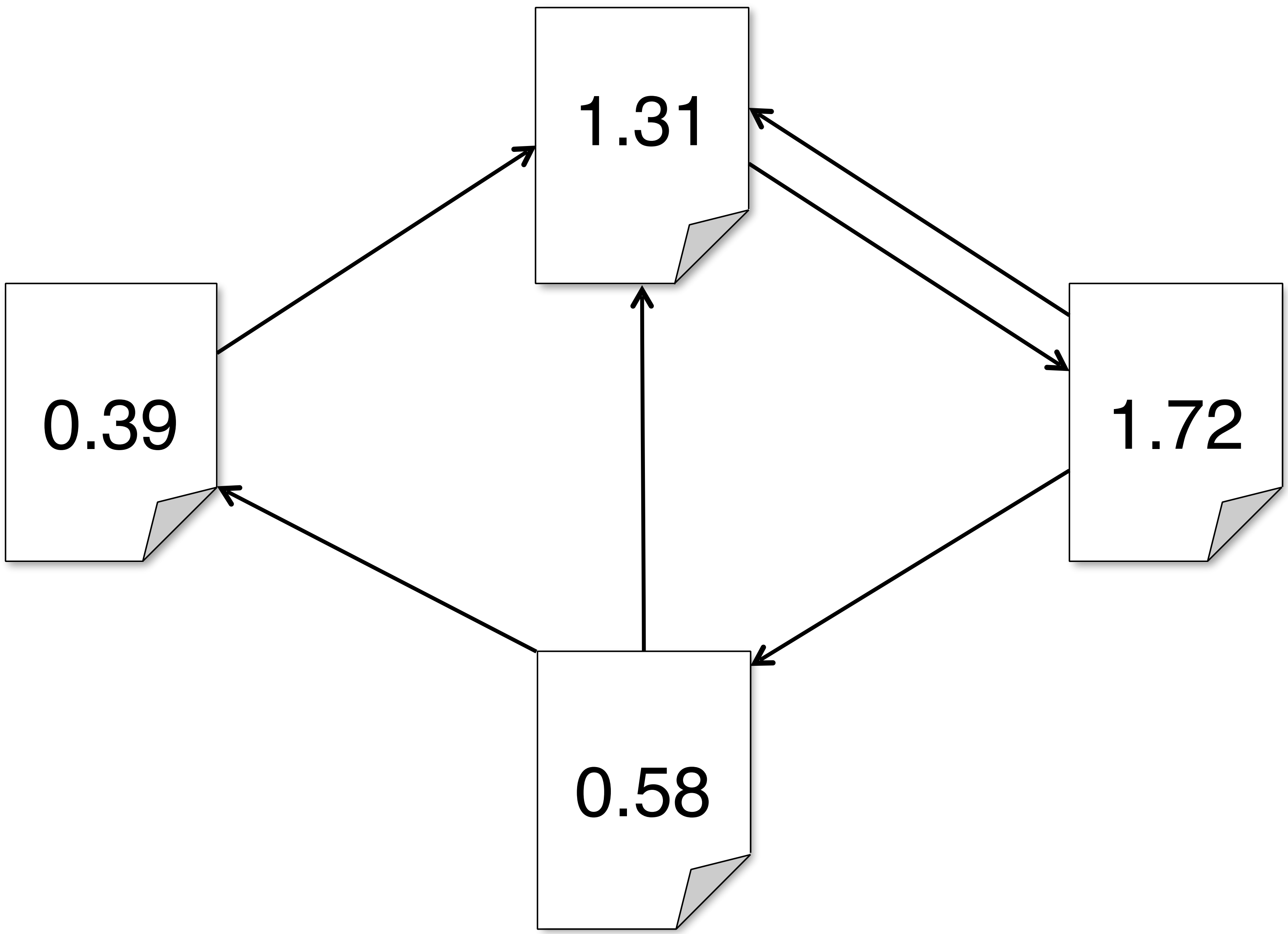
PageRank Example



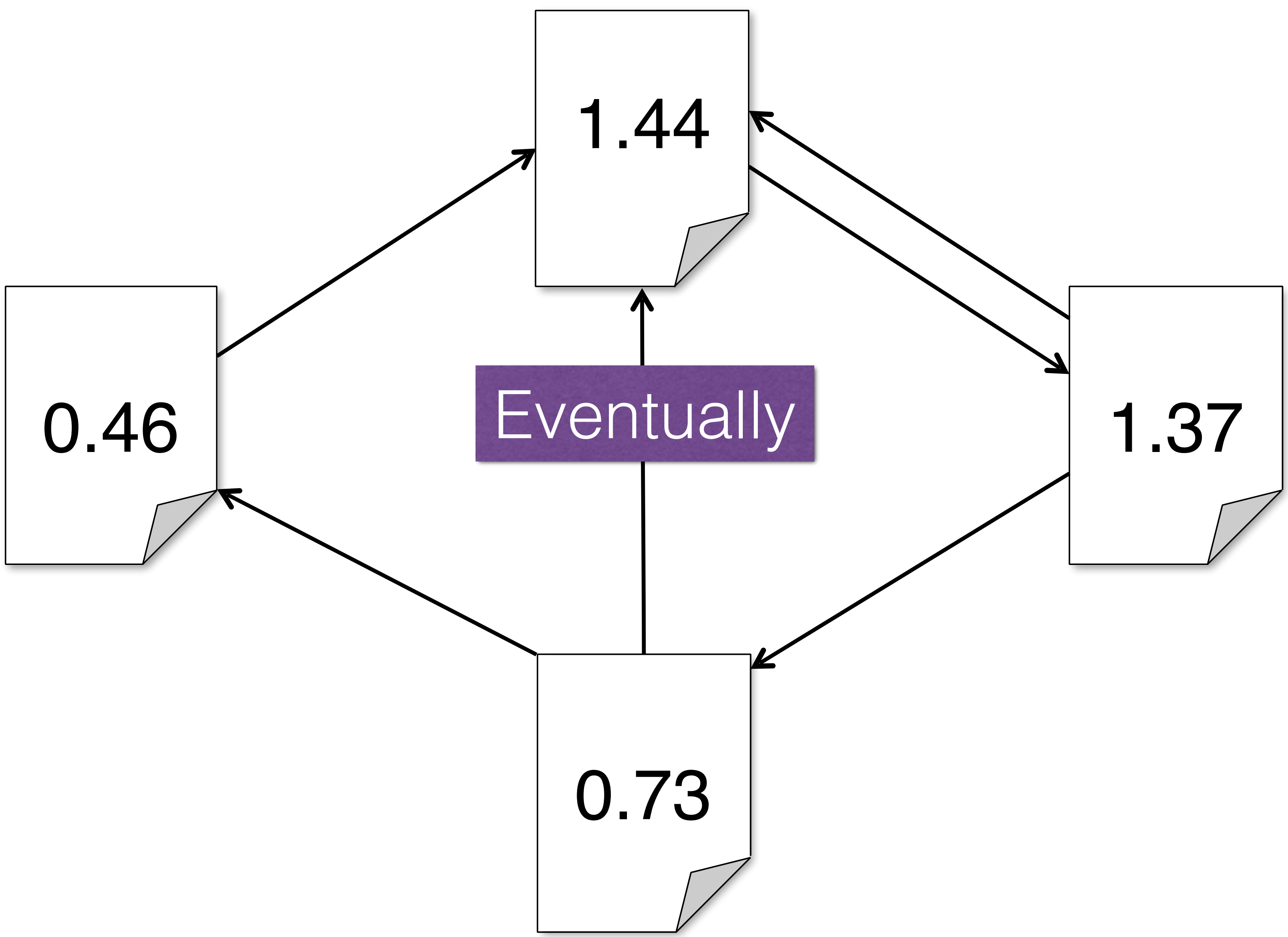
PageRank Example



PageRank Example



PageRank Example



PageRank as Matrix Multiplication

- Rank of each page is the probability of landing on that page for a random surfer on the web

- Probability of visiting all pages after k steps is

$$V_k = A^k \times V^t$$

V: the initial rank vector

A: the link structure (sparse matrix)

Data Representation in Spark

- Each page is identified by its unique URL rather than an index

- Ranks vectors (**V**): RDD[(URL, Double)]

- Links matrix (**A**): RDD[(URL, List(URL))]

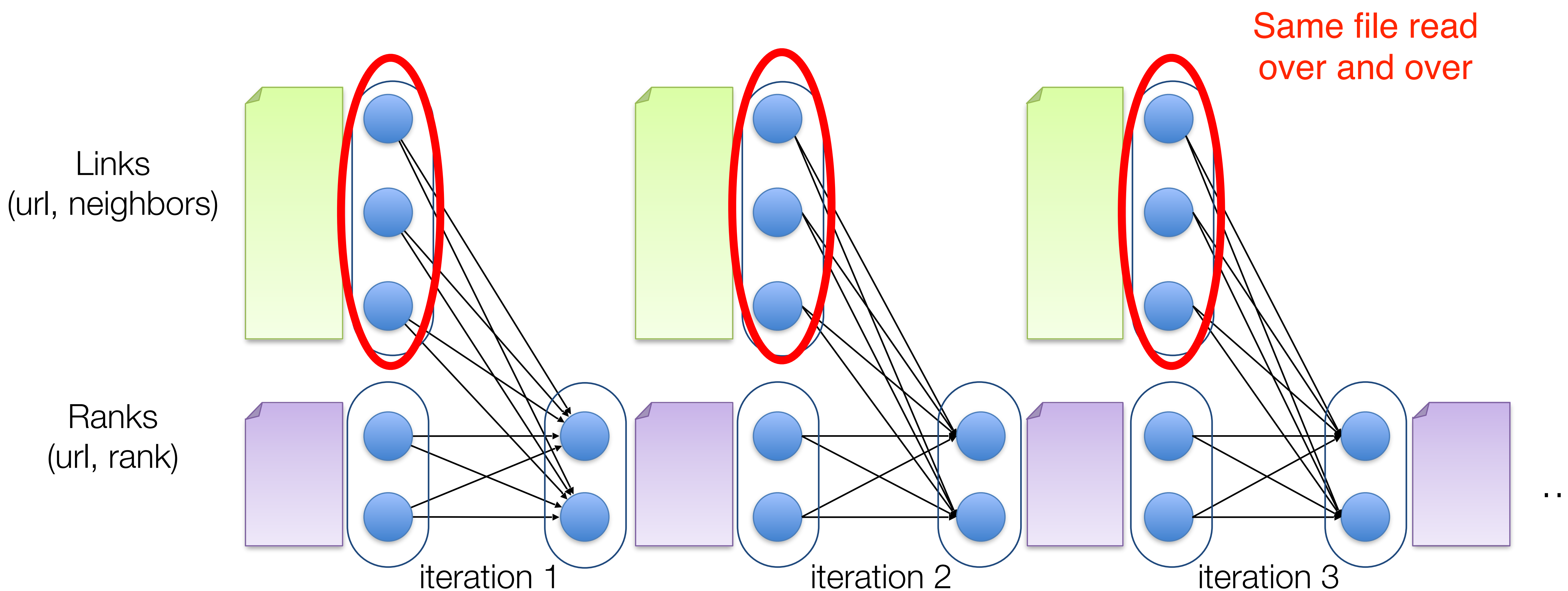
Spark Implementation

```
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

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

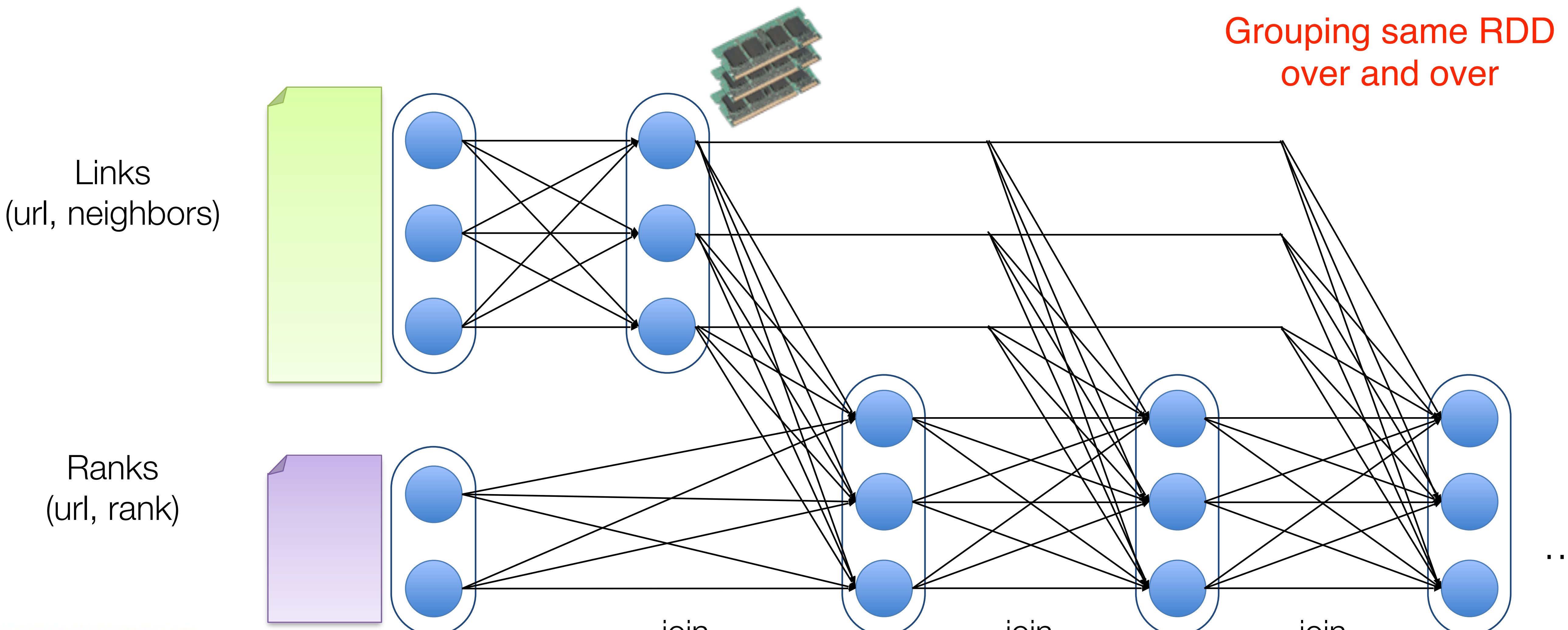
Matrix Multiplication

- Repeatedly multiply sparse matrix and vector



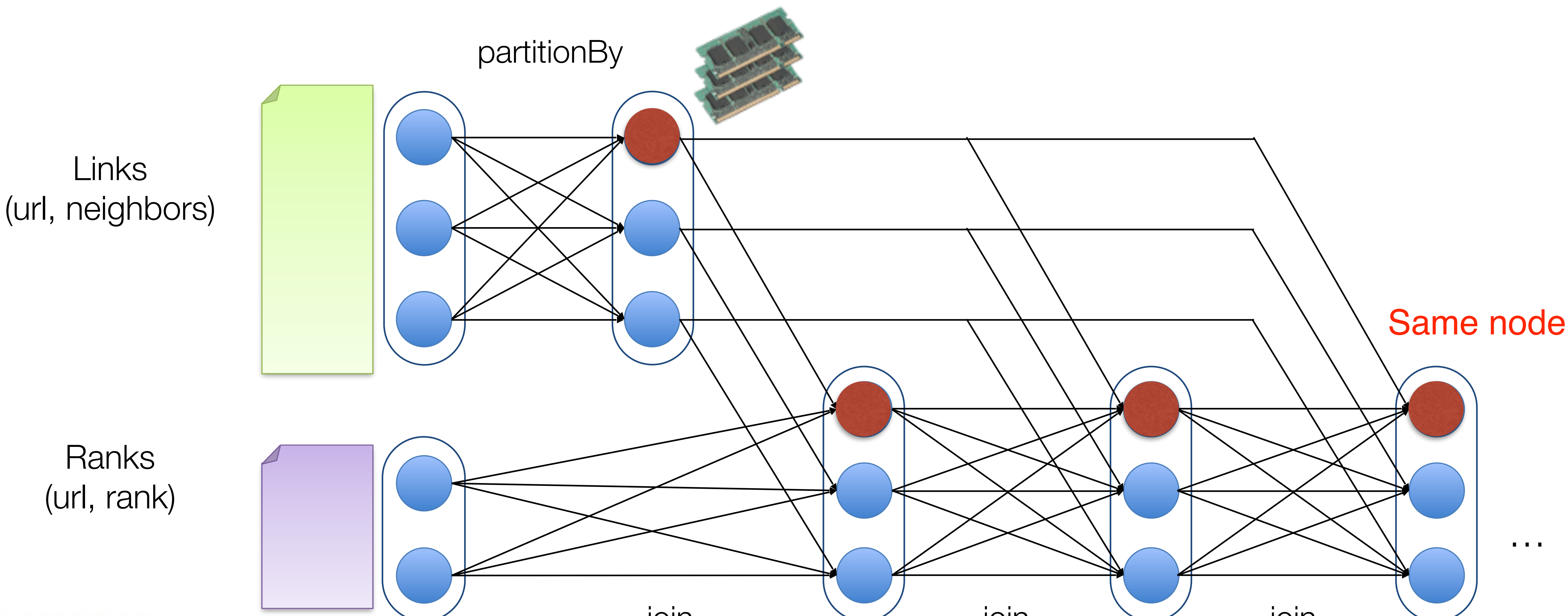
Spark can do much better

- Using cache(), keep neighbors in memory
- Do not write intermediate results on disk



Spark can do much better

- Do not partition neighbors every time



Spark Implementation

```
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

links.partitionBy(hashFunction).cache()

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

Conclusions

When applying any algorithm to big data watch for

1. Correctness

2. Performance

   - Cache RDDs to avoid I/O

   - Avoid unnecessary computation

3. Trade-off between accuracy and performance

Numerical Computing with Spark

DATABRICKS