CS60021: Scalable Data Mining

Sourangshu Bhattacharya

SPARK

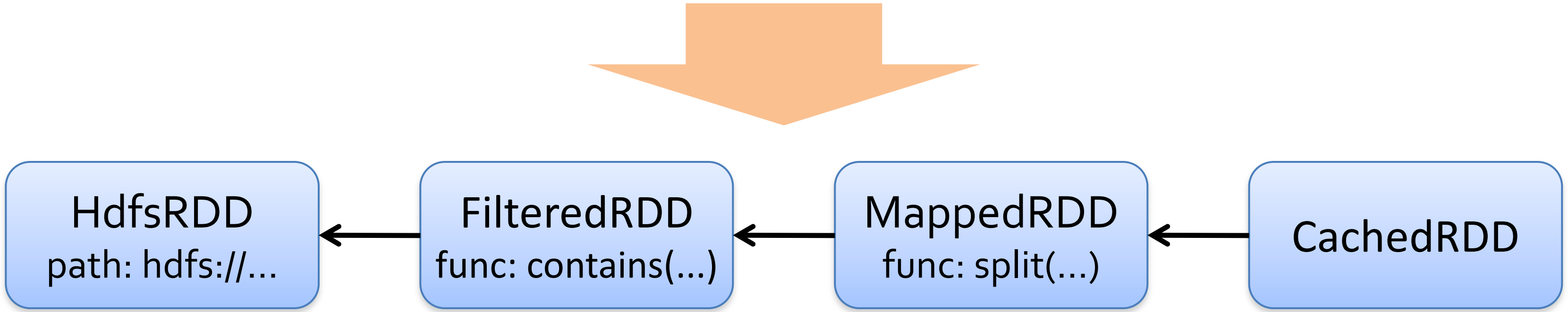
RDD Operations

Transformations (define a new RDD)	Actions (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

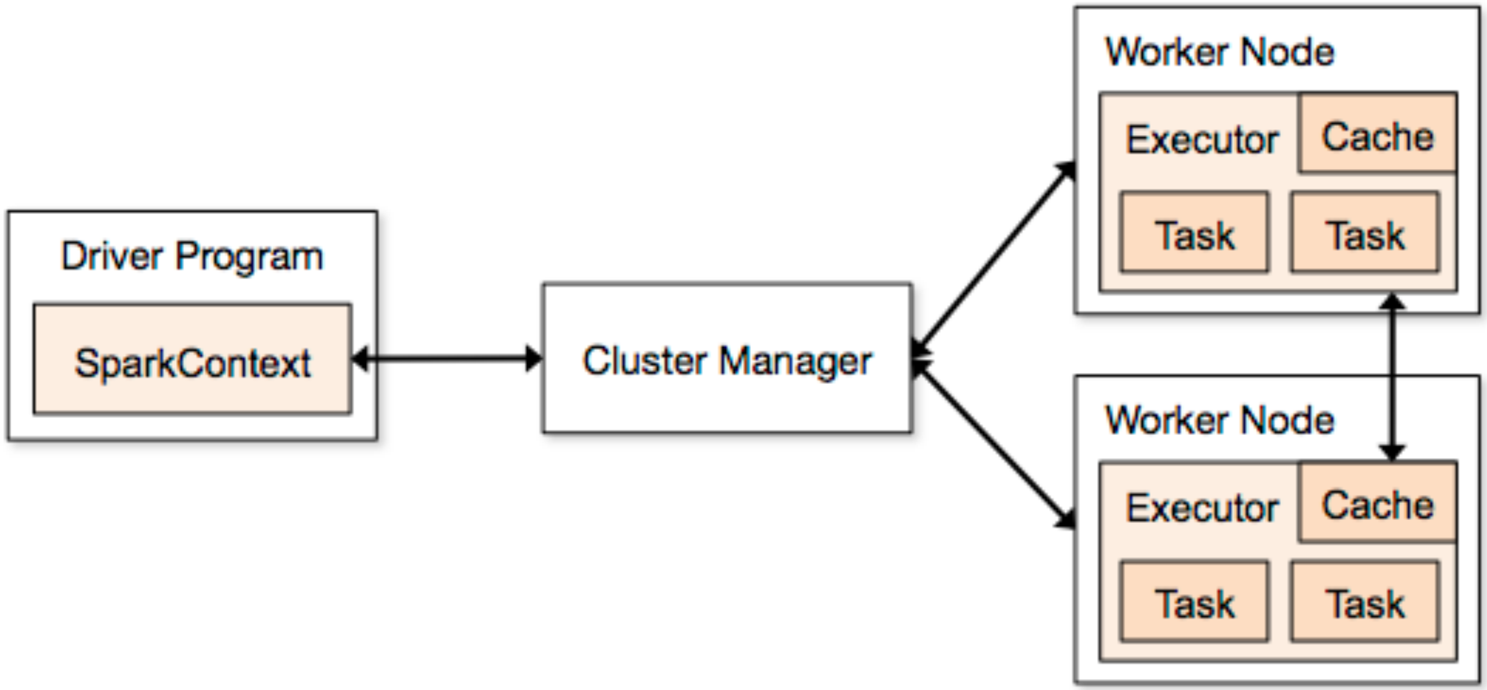
RDD Fault Tolerance

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions
- Ex:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



Spark Architecture



Word Count in Spark

```
val lines = spark.textFile(“hdfs://...”)

val counts = lines.flatMap(_.split(“\\s”))
                  .reduceByKey(_ + _)

counts.save(“hdfs://...”)
```

Example: MapReduce

- MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

Spark Pi

```
val slices = if (args.length > 0) args(0).toInt else 2

val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid
overflow

val count = spark.sparkContext.parallelize(1 until n,
slices).map { i =>
      val x = random * 2 - 1
      val y = random * 2 - 1
      if (x*x + y*y <= 1) 1 else 0
    }.reduce(_ + _)

println(s"Pi is roughly ${4.0 * count / (n - 1)}")
```

Example: Matrix Multiplication

Matrix Multiplication

- Representation of Matrix:
  - List <Row index, Col index, Value>
  - Size of matrices: First matrix (A): m*k, Second matrix (B): k*n
- Scheme:
  - For each input record: If input record
- Mapper key: <row_index_matrix_1, Column_index_matrix_2>
- Mapper value: < column_index_1 / row_index_2, value>
- GroupByKey: List(Mapper Values)
- Collect all (two) records with the same first field multiply them and add to the sum.

Example: Logistic Regression

Logistic Regression

- Binary Classification. *y ε {+1, -1}*
- Probability of classes given by linear model:

$$p(y \mid x, w) = \frac{1}{1 + e^{(-y w^T x)}}$$

- Regularized ML estimate of w given dataset (x_i, y_i) is obtained by minimizing:

$$l(w) = \sum_i \log(1 + \exp(-y_i w^T x_i)) + \frac{\lambda}{2} w^T w$$

Logistic Regression

- Gradient of the objective is given by:

$$\nabla l(w) = \sum_i (1 - \sigma(y_i w^T x_i)) y_i x_i - \lambda w$$

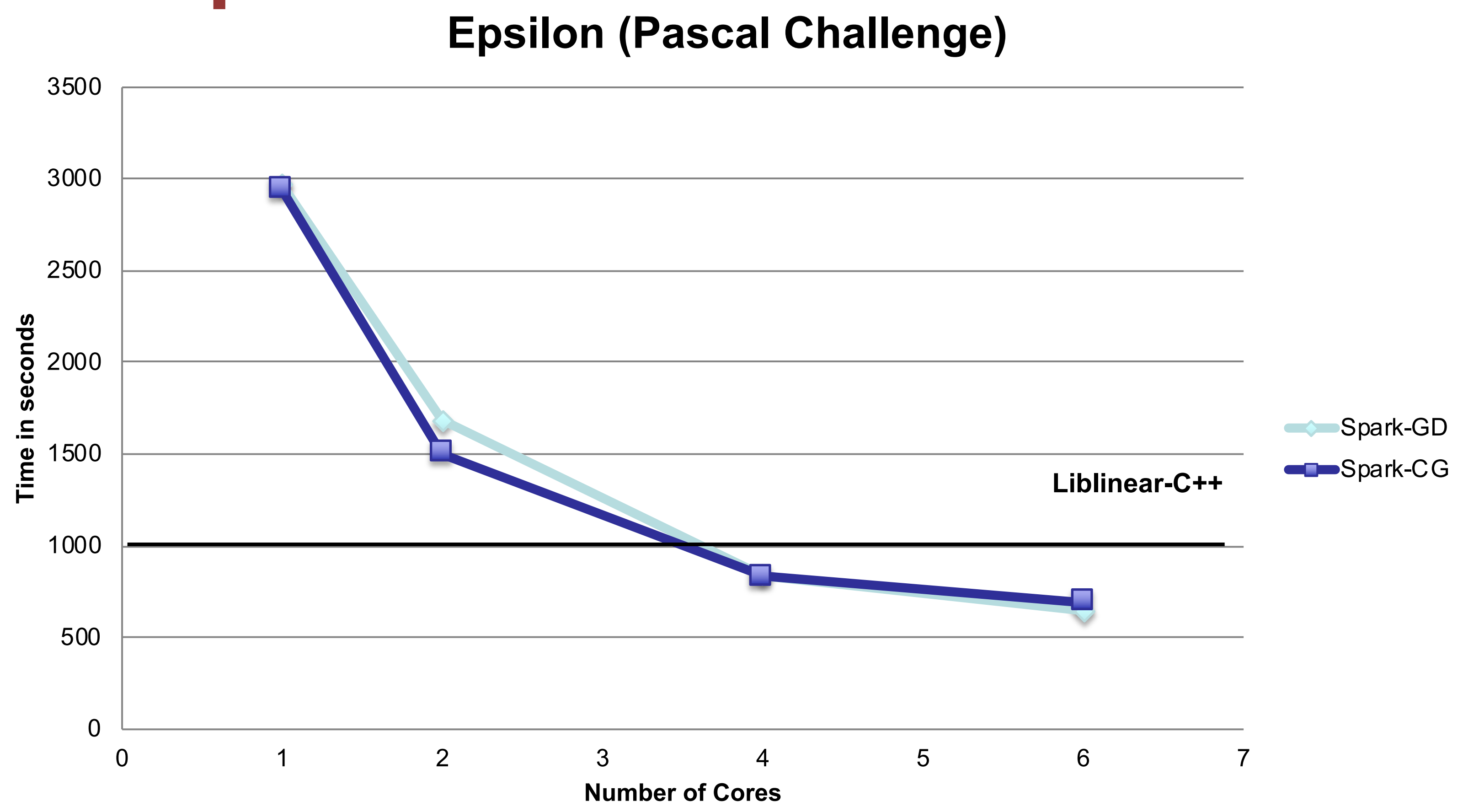
- Gradient Descent updates are:

$$w^{t+1} = w^t - s\nabla l(w^t)$$

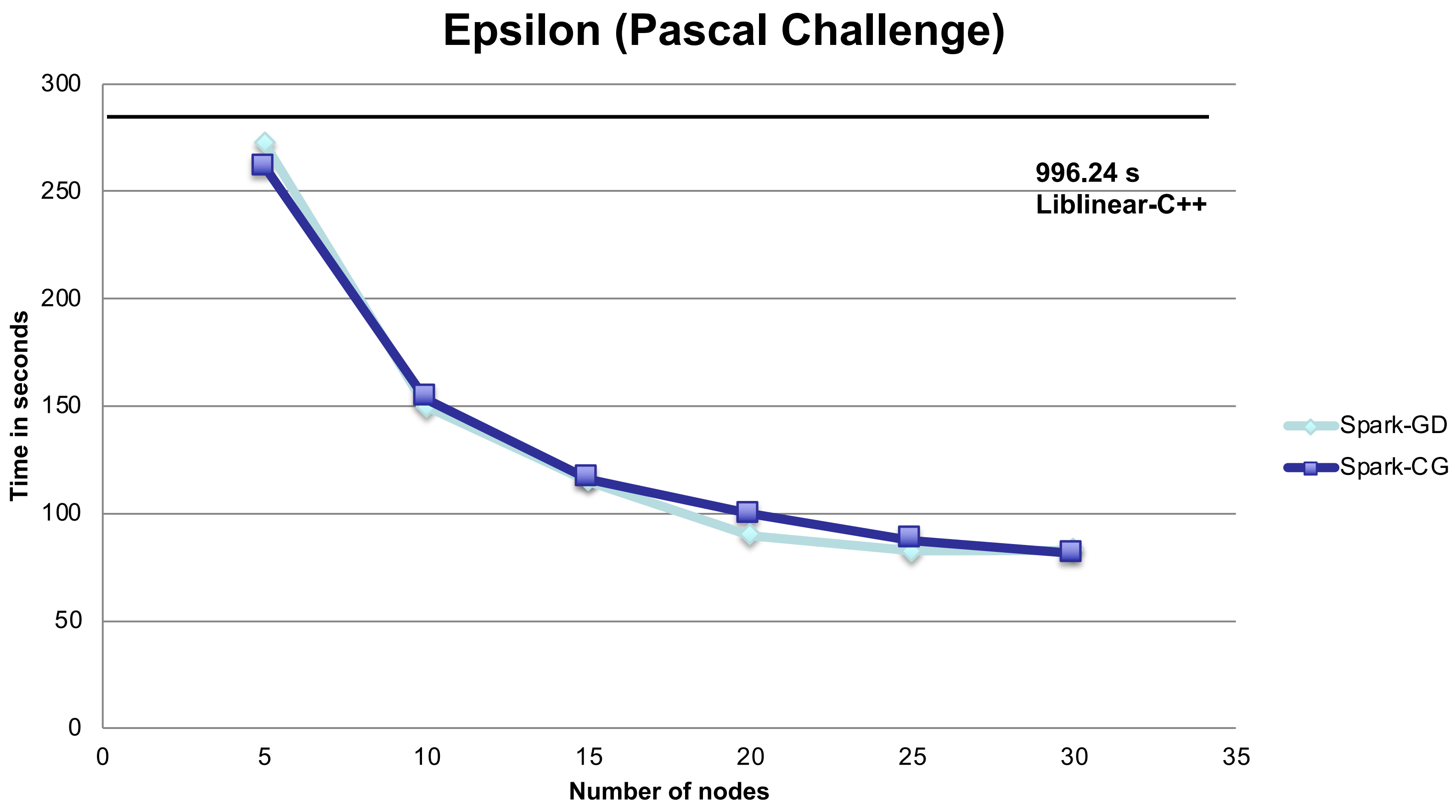
Spark Implementation

```
val x = sc.textFile(file) //creates RDD
var w = 0
do {
//creates RDD
val g = x.map(a => grad(w,a)).reduce(_+_)
s = linesearch(x,w,g)
w = w - s * g
}while(norm(g) > e)
```

Scaleup with Cores



Scaleup with Nodes



Example: PageRank

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

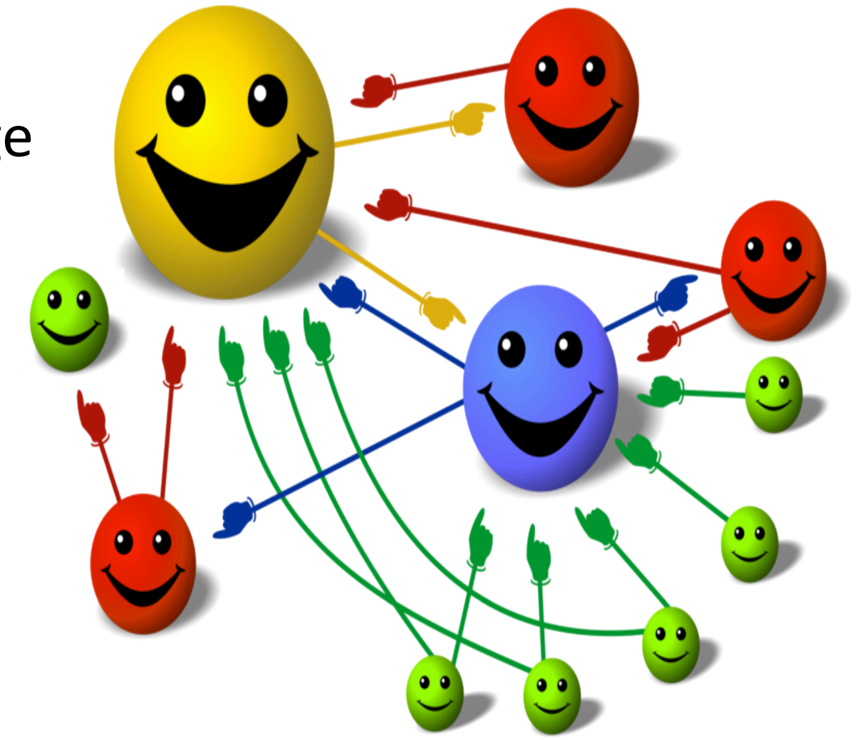
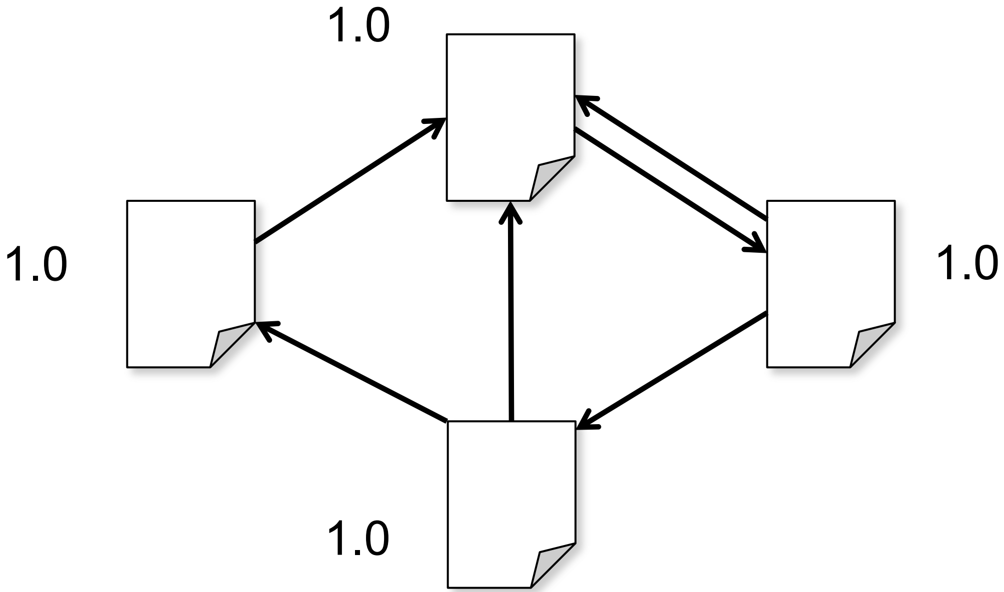


Image: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

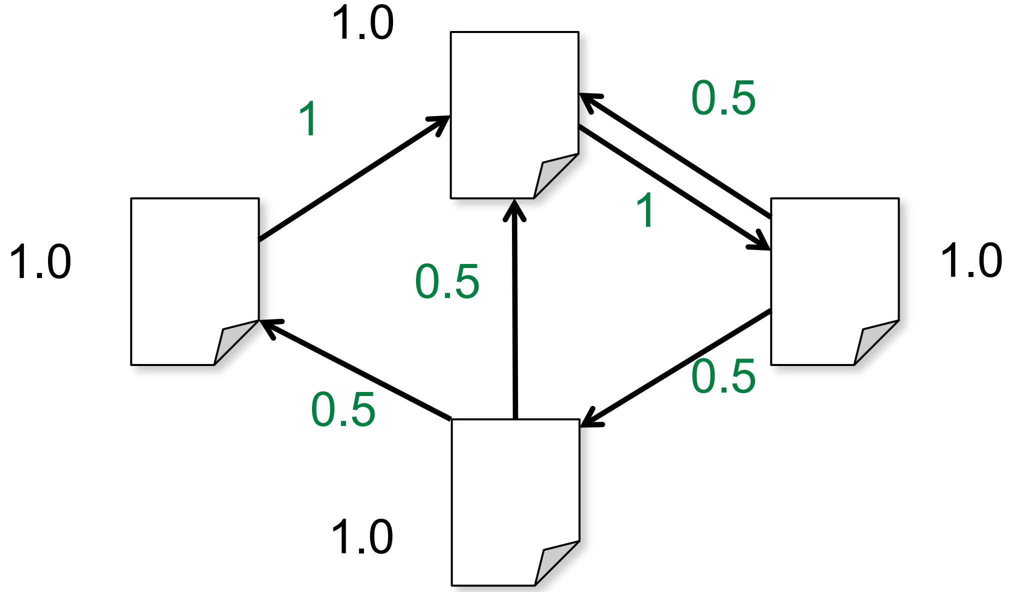
Algorithm

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



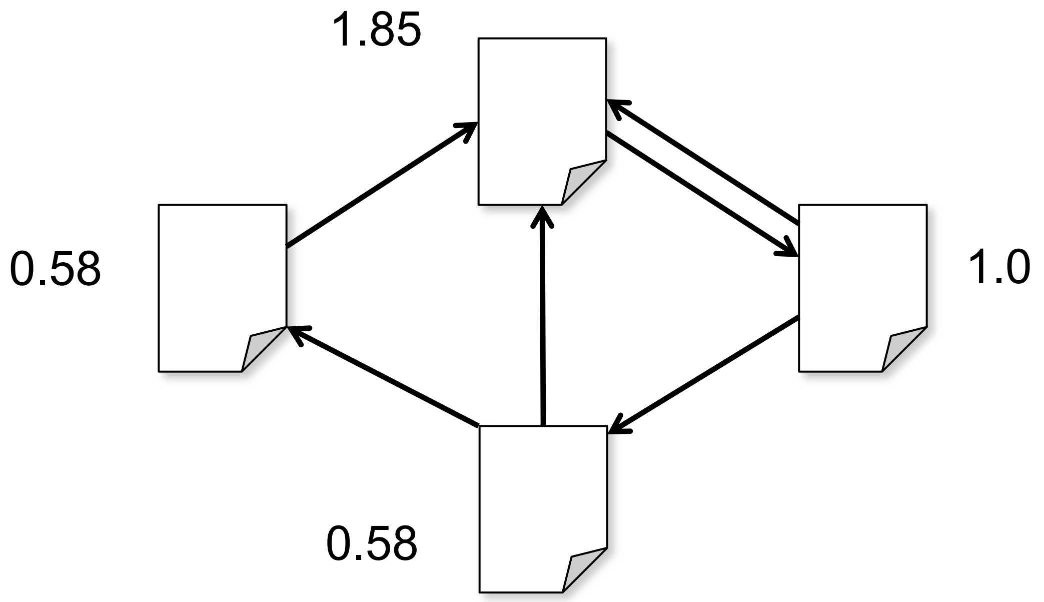
Algorithm

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 × contribs



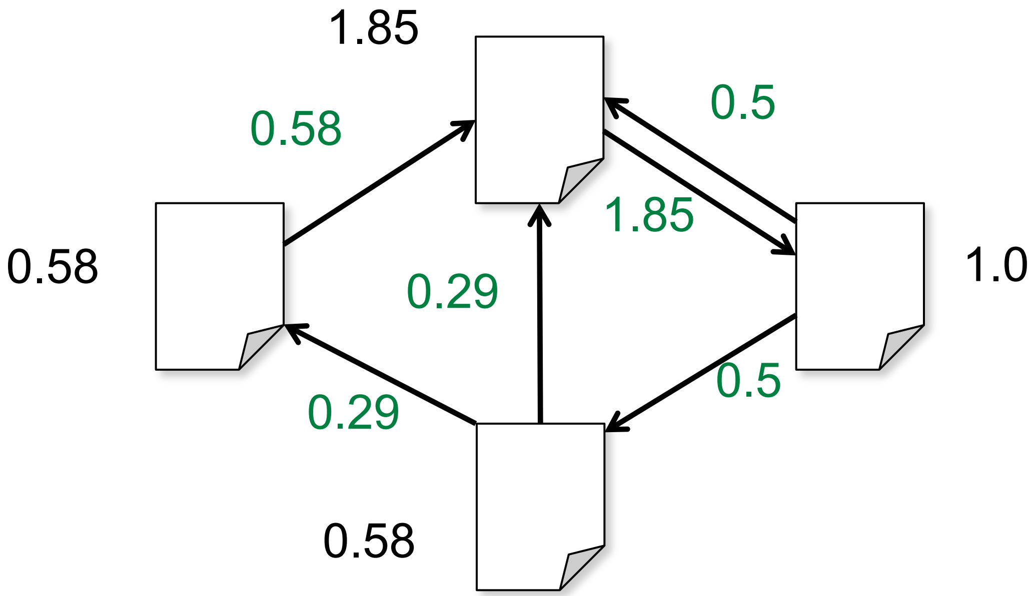
Algorithm

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



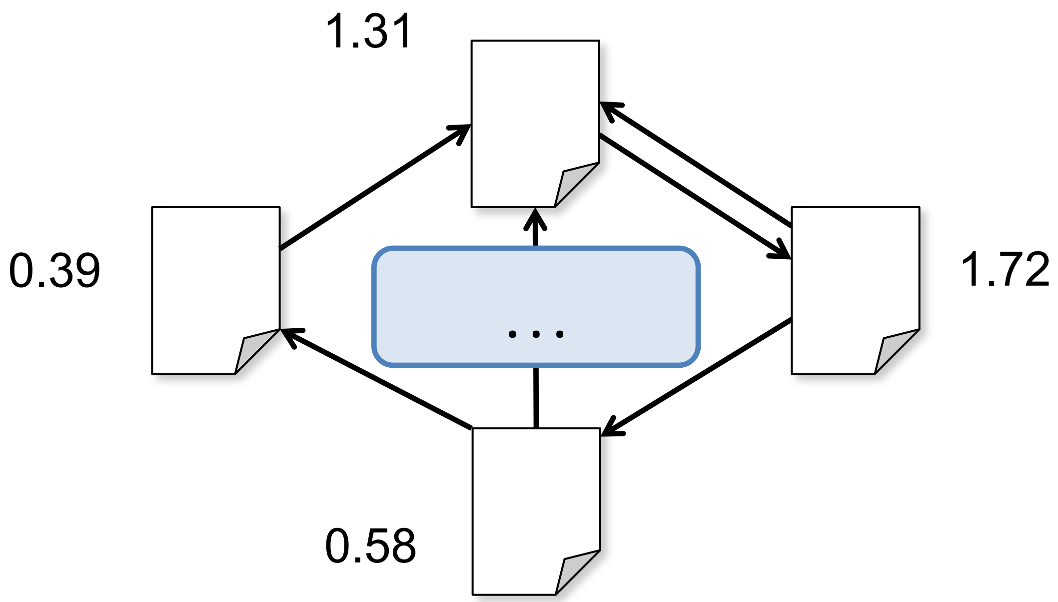
Algorithm

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



Algorithm

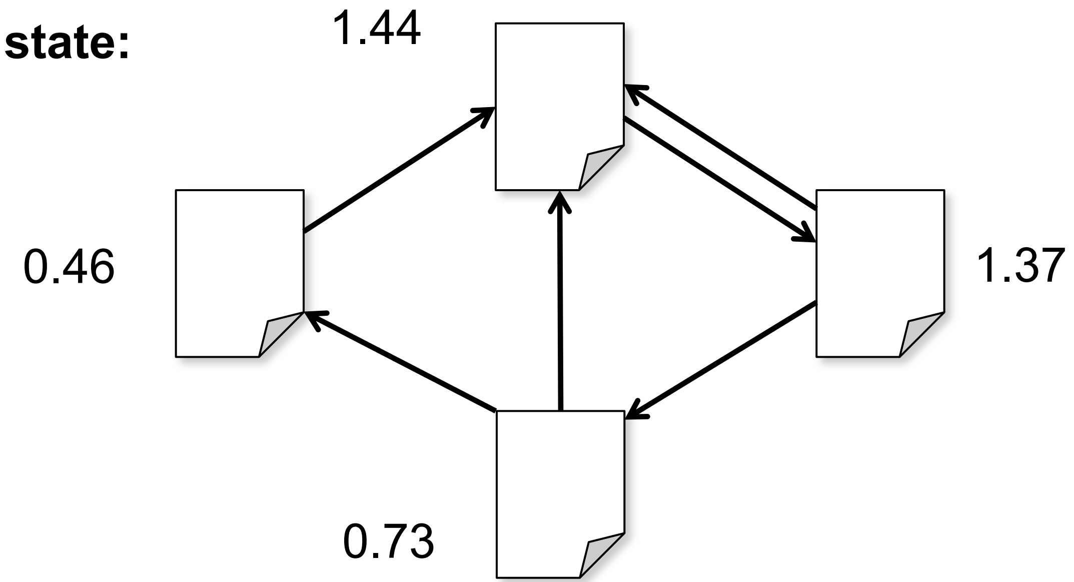
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs



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:



Spark Implementation

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

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

ranks.saveAsTextFile(...)
```

Example: Alternating Least squares

Collaborative filtering

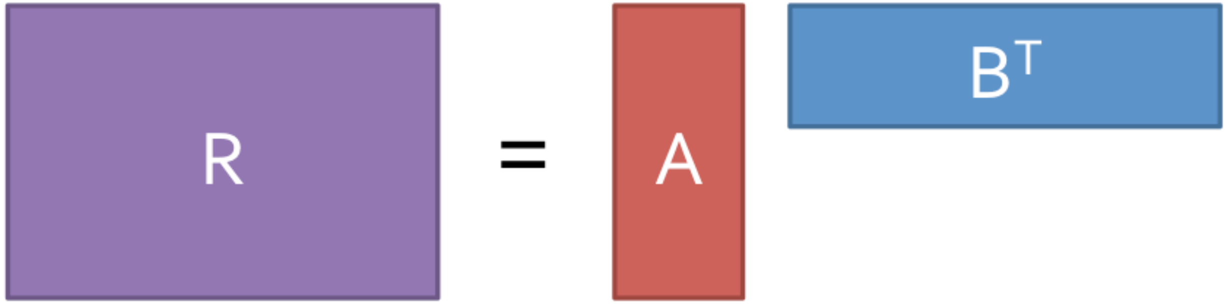
Predict movie ratings for a set of users based on their past ratings of other movies

$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix} \quad \updownarrow \text{Users}$$

$$\longleftrightarrow \text{Movies}$$

Matrix Factorization

Model R as product of user and movie matrices A and B of dimensions U×K and M×K



Problem: given subset of R, optimize A and B

Alternating Least Squares

Start with random A and B

Repeat:

1. Fixing B, optimize A to minimize error on scores in R
2. Fixing A, optimize B to minimize error on scores in R

Naïve Spark ALS

```
val R = readRatingsMatrix(...)

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updateUser(i, B, R))
           .toArray()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R))
           .toArray()
}
```

Efficient Spark ALS

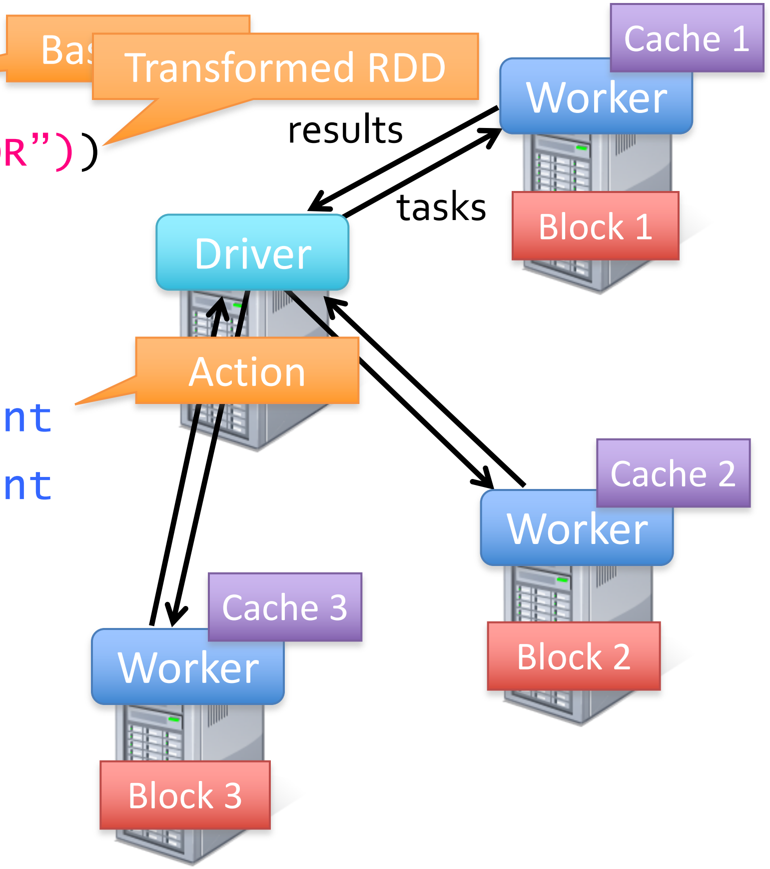
```
val R = spark.broadcast(readRatingsMatrix(...))

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updateUser(i, B, R.value))
           .toArray()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R.value))
           .toArray()
}
```

Solution:
mark R as "broadcast variable"

Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



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

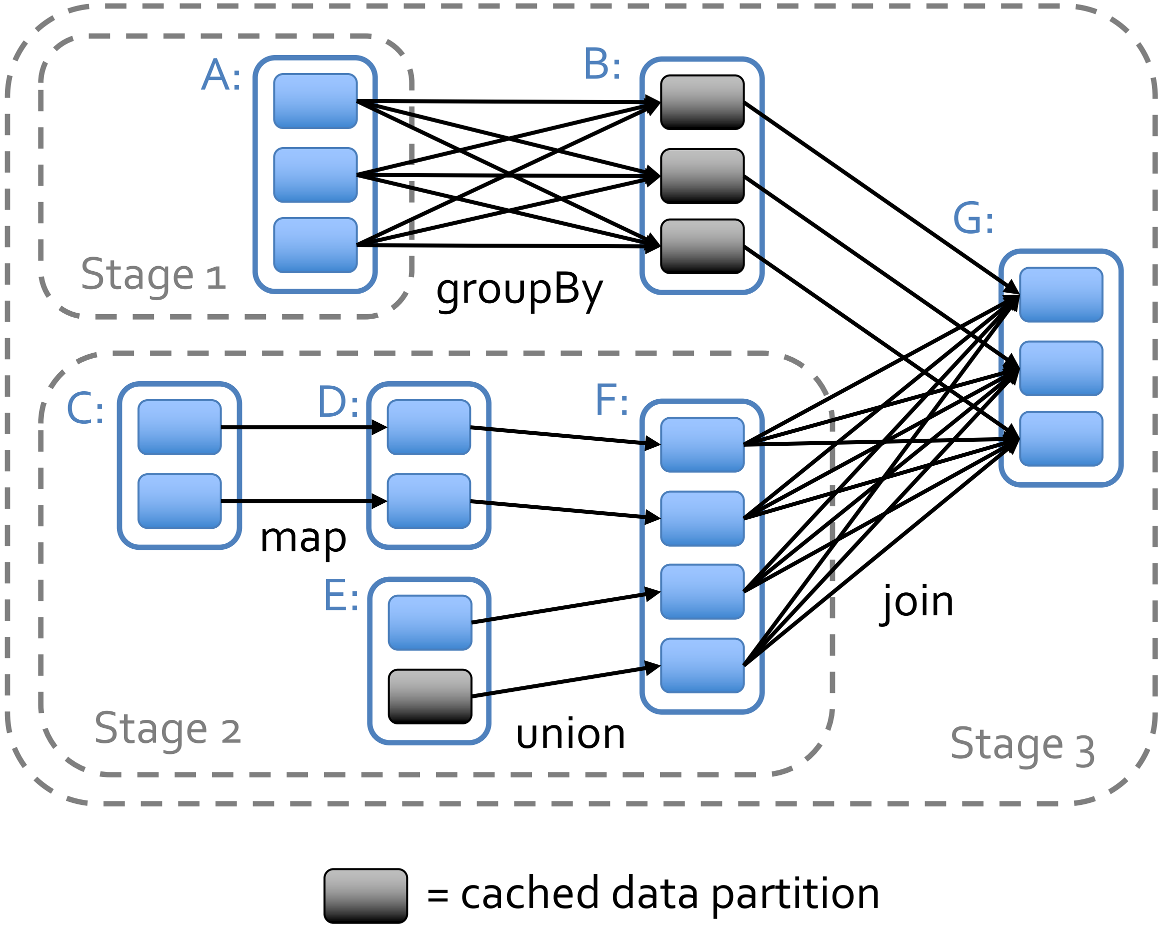
Spark Scheduler

Dryad-like DAGs

Pipelines functions within a stage

Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles



Physical Execution Plan

- User code defines a DAG (directed acyclic graph) of RDDs
  - Operations on RDDs create new RDDs that refer back to their parents, thereby creating a graph.
- Actions force translation of the DAG to an execution plan
  - When you call an action on an RDD, it’s parents must be computed. That job will have one or more stages, with tasks for each partition. Each stage will correspond to one or more RDDs in the DAG. A single stage can correspond to multiple RDDs due to pipelining.
- Tasks are scheduled and executed on a cluster
  - Stages are processed in order, with individual tasks launching to compute segments of the RDD. Once the final stage is finished in a job, the action is complete.

Tasks

- Each task internally performs the following steps:

  - Fetching its input, either from data storage (if the RDD is an input RDD), an existing RDD (if the stage is based on already cached data), or shuffle outputs.

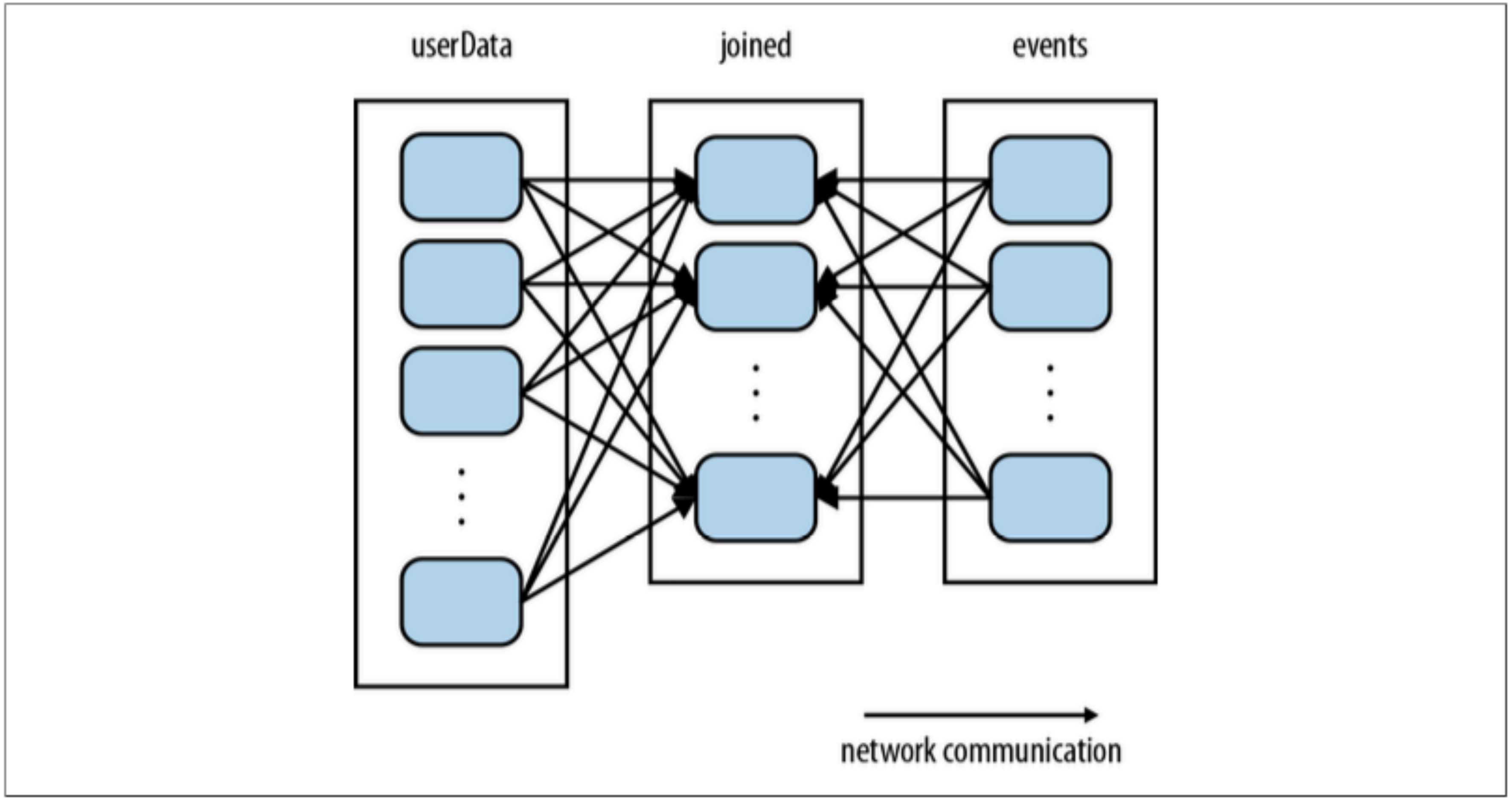
  - Performing the operation necessary to compute RDD(s) that it represents. For instance, executing filter() or map() functions on the input data, or performing grouping or reduction.

  - Writing output to a shuffle, to external storage, or back to the driver (if it is the final RDD of an action such as count()).

User Log Mining

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

def processNewLogs(logFileName: String) {

val events = sc.sequenceFile[UserID, LinkInfo](logFileName)

val joined = userData.join(events) // RDD of (UserID, (UserInfo,
LinkInfo)) pairs

val offTopicVisits = joined.filter {
case (userId, (userInfo, linkInfo)) => // Expand the tuple into its
components      userInfo.topics.contains(linkInfo.topic)
}.count()

println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

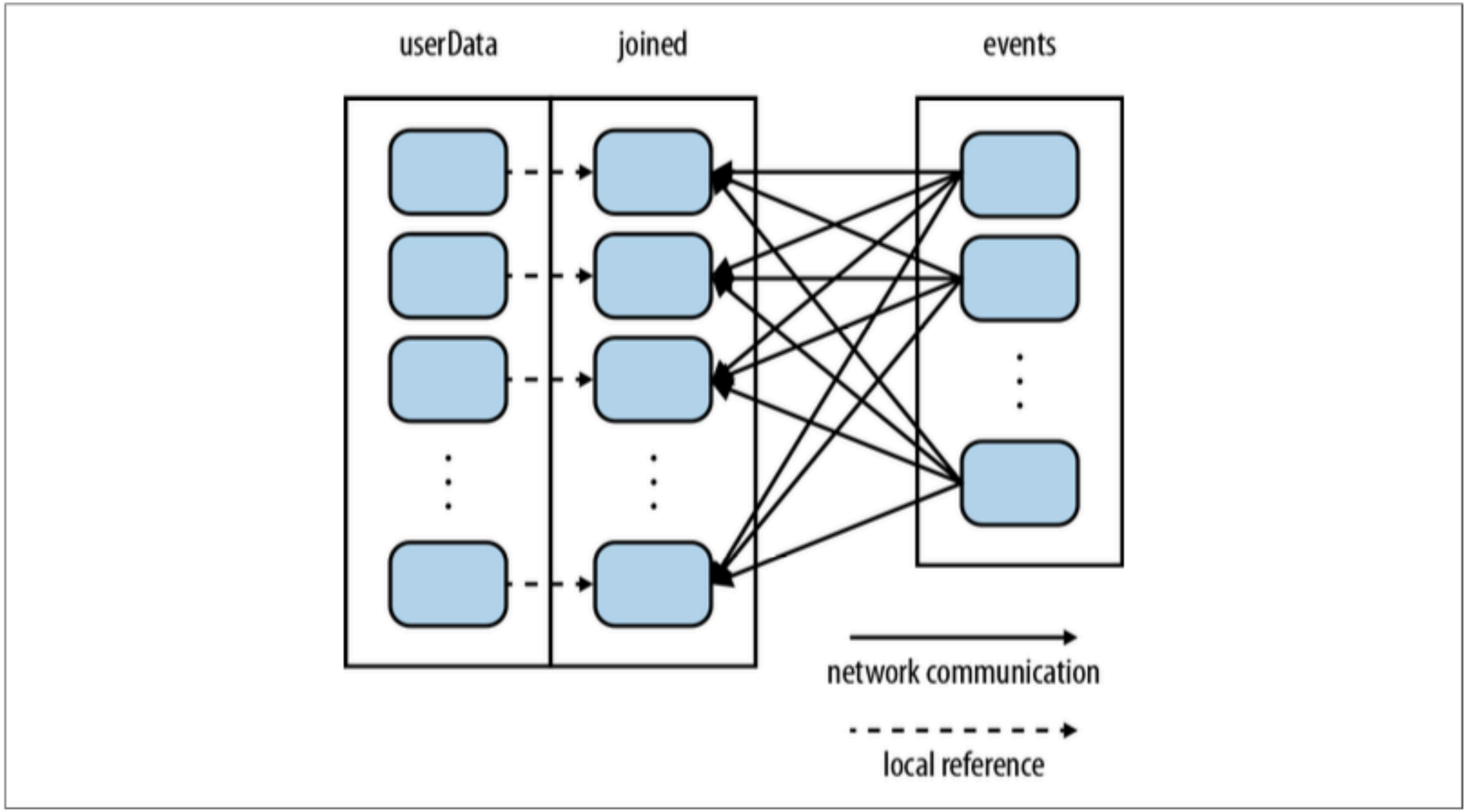
User Log Mining



User Log Mining

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
.partitionBy(new HashPartitioner(100)) // Create 100 partitions
.persist()

def processNewLogs(logFileName: String) {

val events = sc.sequenceFile[UserID, LinkInfo](logFileName)

val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs

val offTopicVisits = joined.filter {
case (userId, (userInfo, linkInfo)) =>
 // Expand the tuple into its components
userInfo.topics.contains(linkInfo.topic)
}.count()
println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

User Log Mining



Partitioning

- Operations **benefitting** from partitioning:

  cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter Join(), groupByKey(), reduceByKey(), combineByKey(), and lookup().
- Operations **affecting** partitioning:

  cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter Join(), groupByKey(), reduceByKey(), combineByKey(), partitionBy(), sort()

  mapValues() (if the parent RDD has a partitioner),
  flatMapValues() (if parent has a partitioner)
  filter() (if parent has a partitioner).

Page Rank (Revisited)

```
val links = sc.objectFile[(String, Seq[String])]("links") .
partitionBy(new HashPartitioner(100)).persist()

var ranks = links.mapValues(v => 1.0)

for(i<-0 until 10) {
val contributions = links.join(ranks).flatMap {
case (pageId, (nbh, rank)) => nbh.map(dest => (dest, rank / nbh.size))
}
ranks = contributions.reduceByKey((x, y) => x + y).
mapValues(v => 0.15 + 0.85*v)
 }
ranks.saveAsTextFile("ranks")
```

Accumulators

```
val sc = new SparkContext(...) val file = sc.textFile("file.txt")

val blankLines = sc.accumulator(0)
// Create an Accumulator[Int] initialized to 0
val callSigns = file.flatMap(
line => {    if (line == "") {
blankLines += 1                    // Add to the accumulator
}
line.split(" ") })

callSigns.saveAsTextFile("output.txt”)

println("Blank lines: " + blankLines.value)
```

Conclusion:

- We have seen:
  - Motivation
  - RDD
  - Actions and transformations
  - Examples:
    - Matrix multiplication
    - Logistic regression
    - Pagerank
    - Alternating least squares
    - User log mining
  - Partitioning
  - Accumulators
  - Scheduling of tasks

References:

- Learning Spark: Lightning-Fast Big Data Analysis. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. O Reilly Press 2015.

- Any book on scala and spark.

- Spark RDD programming guide: https://spark.apache.org/docs/latest/rdd-programming-guide.html