

CS60021: Scalable Data Mining

Sourangshu Bhattacharya

In this Lecture:

- Outline:
 - Scala
 - Spark
 - RDDs
 - Spark programming
 - Spark system

SCALA

Scala

- Scala is both functional and object-oriented
 - every value is an object
 - every function is a value--including methods
- Scala is interoperable with java.
- Scala is statically typed
 - includes a local type inference system:

Var and Val

❑ Use `var` to declare variables:

❑ `var x = 3;`

❑ `x += 4;`

❑ Use `val` to declare values (final vars)

❑ `val y = 3;`

❑ `y += 4; // error`

❑ Notice no types, but it is statically typed

❑ `var x = 3;`

❑ `x = "hello world"; // error`

❑ Type annotations:

❑ `var x : Int = 3;`

Class definition

```
class Point(val xc: Int, val yc: Int) {  
    var x: Int = xc  
    var y: Int = yc  
  
    def move(dx: Int, dy: Int) {  
        x = x + dx  
        y = y + dy  
        println ("Point x location : " + x);  
        println ("Point y location : " + y);  
    }  
}
```

Scala

❑ Class instances

- ❑ `val c = new IntCounter[String];`

❑ Accessing members

- ❑ `println(c.size); // same as c.size()`

❑ Defining functions:

- ❑ `def foo(x : Int) { println(x == 42); }`

- ❑ `def bar(y : Int): Int = y + 42; // no braces needed!`

- ❑ `def return42 = 42; // No parameters either!`

Functions are first-class objects

- Functions are **values** (like integers, etc.) and can be assigned to variables, passed to and returned from functions, and so on
- Wherever you see the `=>` symbol, it's a literal function
- Example (assigning a literal function to the variable `foo`):

```
– scala> val foo =  
    (x: Int) => if (x % 2 == 0) x / 2 else 3 * x + 1  
foo: (Int) => Int = <function1>
```

```
scala> foo(7)  
res28: Int = 22
```

- The basic syntax of a function literal is
parameter_list => function_body
- In this example, `foreach` is a function that takes a function as a parameter:
 - `myList.foreach(i => println(2 * i))`

Functions as parameters

- To have a function parameter, you must know how to write its type:

- *(type1, type2, ..., typeN) => return_type*
- *type => return_type // if only one parameter*
- *() => return_type // if no parameters*

- Example:

```
scala> def doTwice(f: Int => Int, n: Int) = f(f(n))  
doTwice: (f: (Int) => Int,n: Int)Int
```

```
scala> def collatz(n: Int) = if (n % 2 == 0) n / 2 else 3 * n + 1  
collatz: (n: Int)Int
```

```
scala> doTwice(collatz, 7)  
res2: Int = 11
```

```
scala> doTwice(a => 101 * a, 3)  
res4: Int = 30603
```

Lists

- Scala's **Lists** are more useful, and used more often, than Arrays
 - `val list1 = List(3, 1, 4, 1, 6)`
 - `val list2 = List[Int]()` // An empty list must have an explicit type
- By default, **Lists**, like **Strings**, are **immutable**
 - Operations on an immutable List return a new List
- Basic operations:
 - `list.head` (or `list head`) returns the first element in the list
 - `list.tail` (or `list tail`) returns a list with the first element removed
 - `list(i)` returns the i^{th} element (starting from 0) of the list
 - `list(i) = value` is **illegal** (immutable, remember?)
- There are over 150 built-in operations on Lists—use the API!

Scala

❑ Defining lambdas – nameless functions (types sometimes needed)

- ❑ `val f = {x : Int => x + 42;}`

❑ Closures (context sensitive functions)

- ❑ `var y = 3;`

- ❑ `val g = {x : Int => y += 1; x+y; }`

❑ Maps (and a cool way to do some functions)

- ❑ `List(1,2,3).map(_+10).foreach(println)`

❑ Filtering (and ranges!)

- ❑ `1 to 100 filter (_ % 7 == 3) foreach (println)`

Higher-order methods on Lists

- `map` applies a one-parameter function to every element of a List, returning a new List
 - `scala> def double(n: Int) = 2 * n`
`double: (n: Int)Int`
 - `scala> val l1 = List(2, 3, 5, 7, 11)`
`l1: List[Int] = List(2, 3, 5, 7, 11)`
 - `scala> l1 map double`
`res5: List[Int] = List(4, 6, 10, 14, 22)`
 - `scala> l1 map (n => 3 * n)`
`res6: List[Int] = List(6, 9, 15, 21, 33)`
- `filter` applies a one-parameter test to every element of a List, returning a List of those elements that pass the test
 - `scala> l1 filter(n => n < 5)`
`res10: List[Int] = List(2, 3)`
 - `scala> l1 filter (_ < 5) // abbreviated function where parameter is used once`
`res11: List[Int] = List(2, 3)`

More higher-order methods

- `def filterNot(p: (A) => Boolean): List[A]`
 - Selects all elements of this list which do not satisfy a predicate
- `def count(p: (A) => Boolean): Int`
 - Counts the number of elements in the list which satisfy a predicate
- `def forall(p: (A) => Boolean): Boolean`
 - Tests whether a predicate holds for every element of this list
- `def exists(p: (A) => Boolean): Boolean`
 - Tests whether a predicate holds for at least one of the elements of this list
- `def find(p: (A) => Boolean): Option[A]`
 - Finds the first element of the list satisfying a predicate, if any
- `def sortWith(lt: (A, A) => Boolean): List[A]`
 - Sorts this list according to a comparison function

SPARK

Spark

Spark is an In-Memory Cluster Computing platform for Iterative and Interactive Applications.

<http://spark.apache.org>

Spark

- ❑ Started in AMPLab at UC Berkeley.
- ❑ Resilient Distributed Datasets.
- ❑ Data and/or Computation Intensive.
- ❑ Scalable – fault tolerant.
- ❑ Integrated with SCALA.
- ❑ Straggler handling.
- ❑ Data locality.
- ❑ Easy to use.

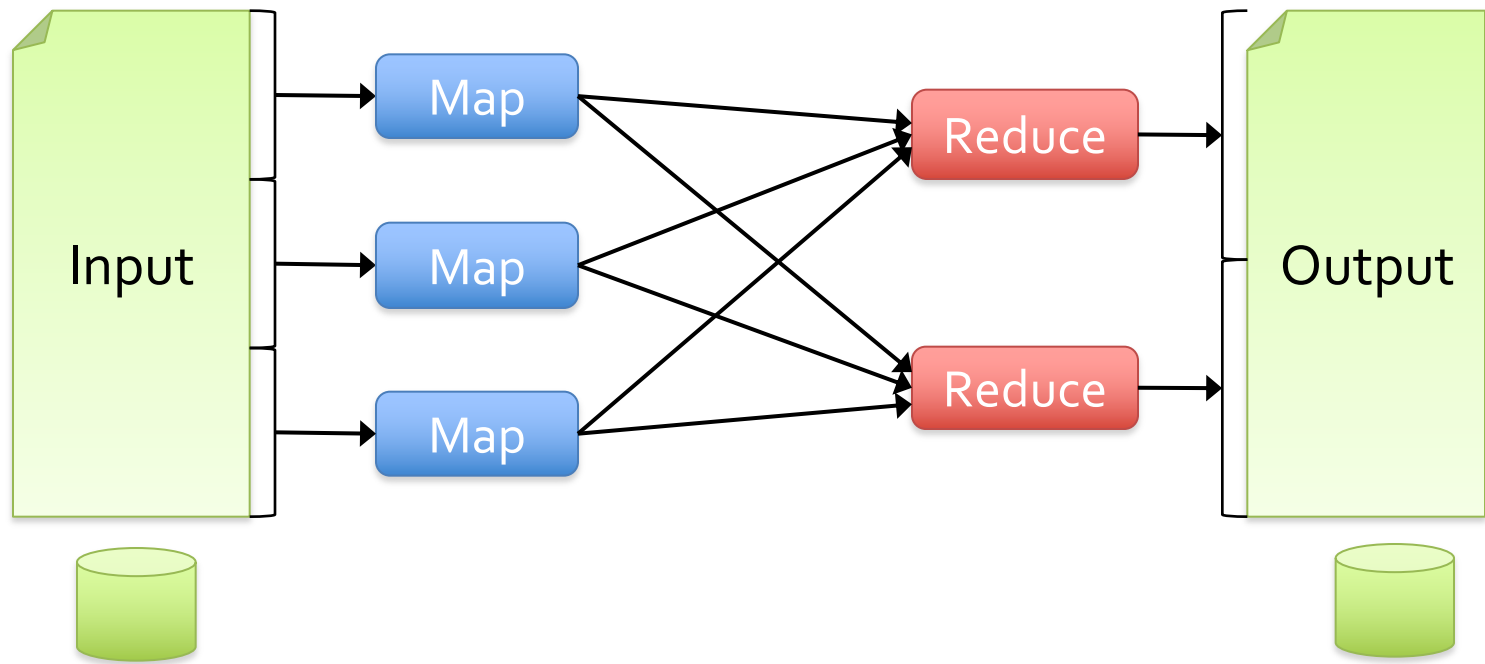
Background

- Commodity clusters have become an important computing platform for a variety of applications
 - **In industry:** search, machine translation, ad targeting, ...
 - **In research:** bioinformatics, NLP, climate simulation, ...
- High-level cluster programming models like MapReduce power many of these apps
- *Theme of this work: provide similarly powerful abstractions for a broader class of applications*

Motivation

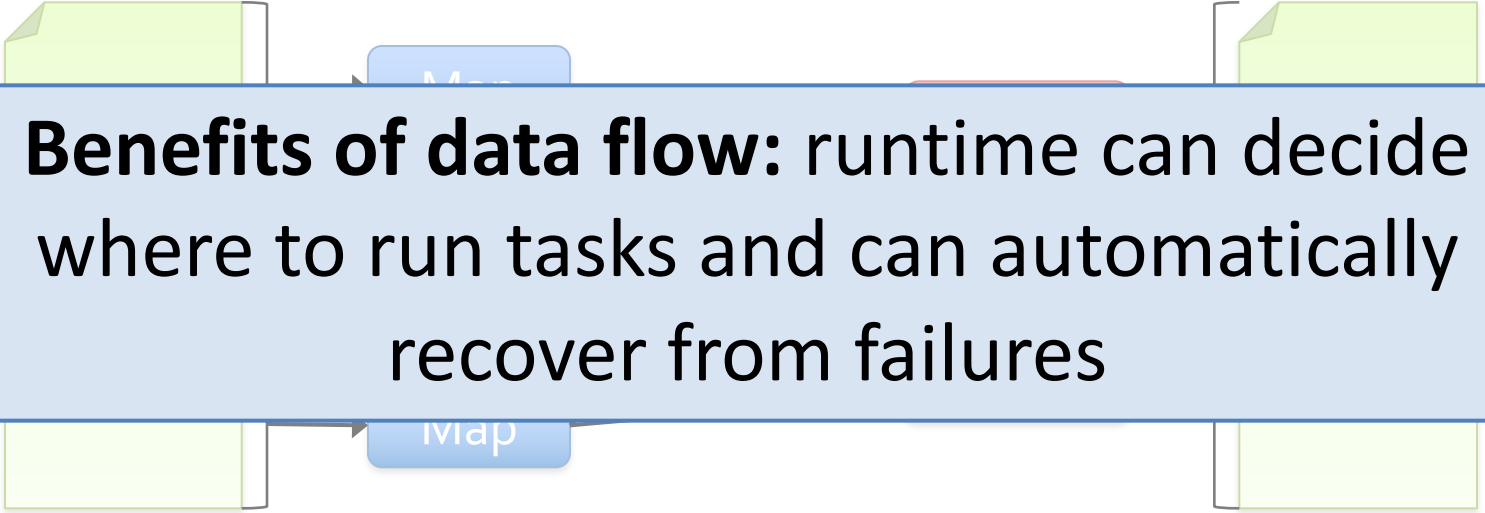
Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



Motivation

- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- E.g., MapReduce:



The diagram illustrates a data flow process. It features two light green rectangular blocks representing stable storage, one on the left and one on the right. A blue rounded rectangular box labeled 'Map' is positioned between them. A red arrow points from the left storage block to the 'Map' box, and another red arrow points from the 'Map' box to the right storage block. A large, light blue rounded rectangular box with a dark blue border is overlaid on the diagram, containing the text 'Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures'.

Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
 - **Iterative** algorithms (many in machine learning)
 - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

Spark Goal

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
 - Fault tolerance (for crashes & stragglers)
 - Data locality
 - Scalability

Solution: augment data flow model with “resilient distributed datasets” (RDDs)

Resilient Distributed Datasets

- Immutable distributed SCALA collections.
 - Array, List, Map, Set, etc.
- Transformations on RDDs create new RDDs.
 - Map, ReduceByKey, Filter, Join, etc.
- Actions on RDD return values.
 - Reduce, collect, count, take, etc.
- Seamlessly integrated into a SCALA program.
- RDDs are materialized when needed.
- RDDs are cached to disk – graceful degradation.
- Spark framework re-computes lost splits of RDDs.

RDDs in More Detail

- ❑ An RDD is an immutable, partitioned, logical collection of records
 - ❑ Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- ❑ Partitioning can be based on a key in each record (using hash or range partitioning)
- ❑ Built using bulk transformations on other RDDs
- ❑ Can be cached for future reuse

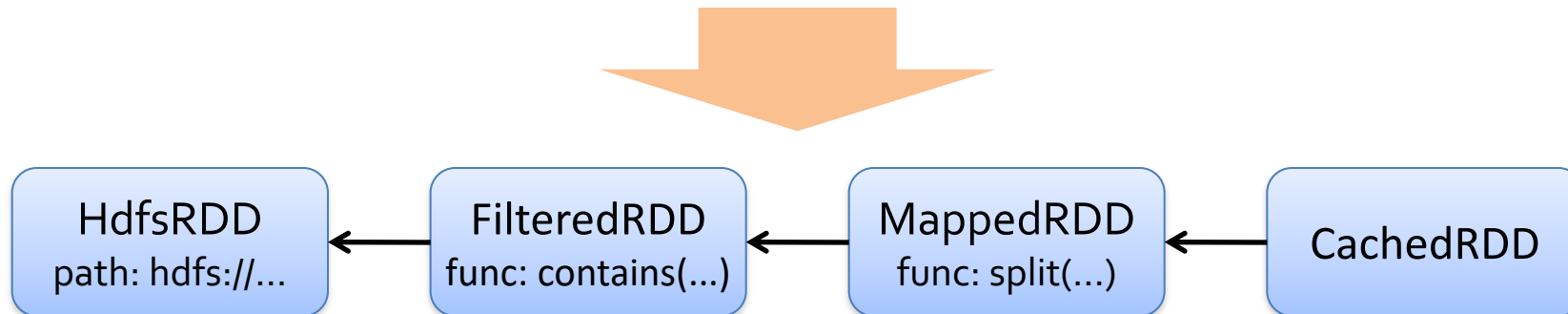
RDD Operations

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

RDD Fault Tolerance

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

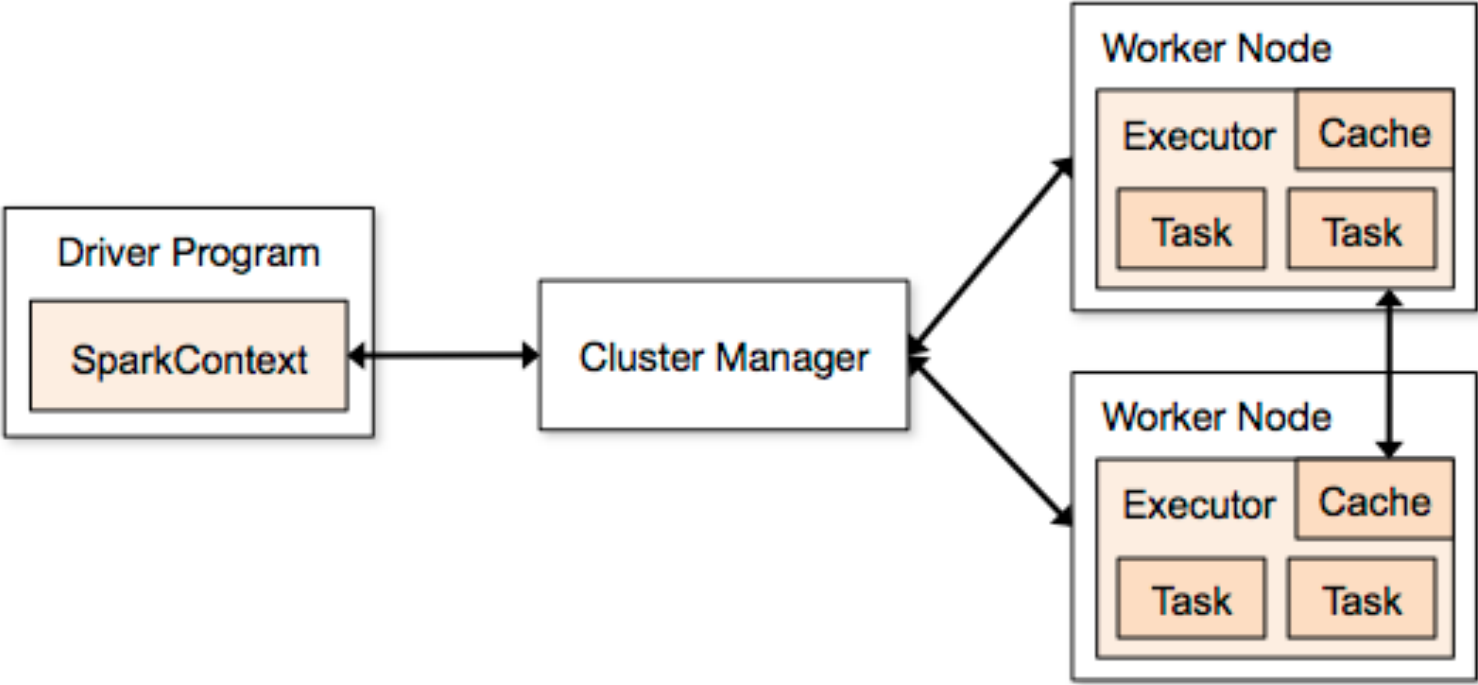
```
cachedMsgs = textFile(...).filter(_.contains("error"))  
                          .map(_.split('\t')(2))  
                          .cache()
```



Benefits of RDD Model

- ❑ Consistency is easy due to immutability
- ❑ Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- ❑ Locality-aware scheduling of tasks on partitions
- ❑ Despite being restricted, model seems applicable to a broad variety of applications

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)}")
```

Example: Logistic Regression

Logistic Regression

- Binary Classification. $y \in \{+1, -1\}$
- Probability of classes given by linear model:

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

- 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:

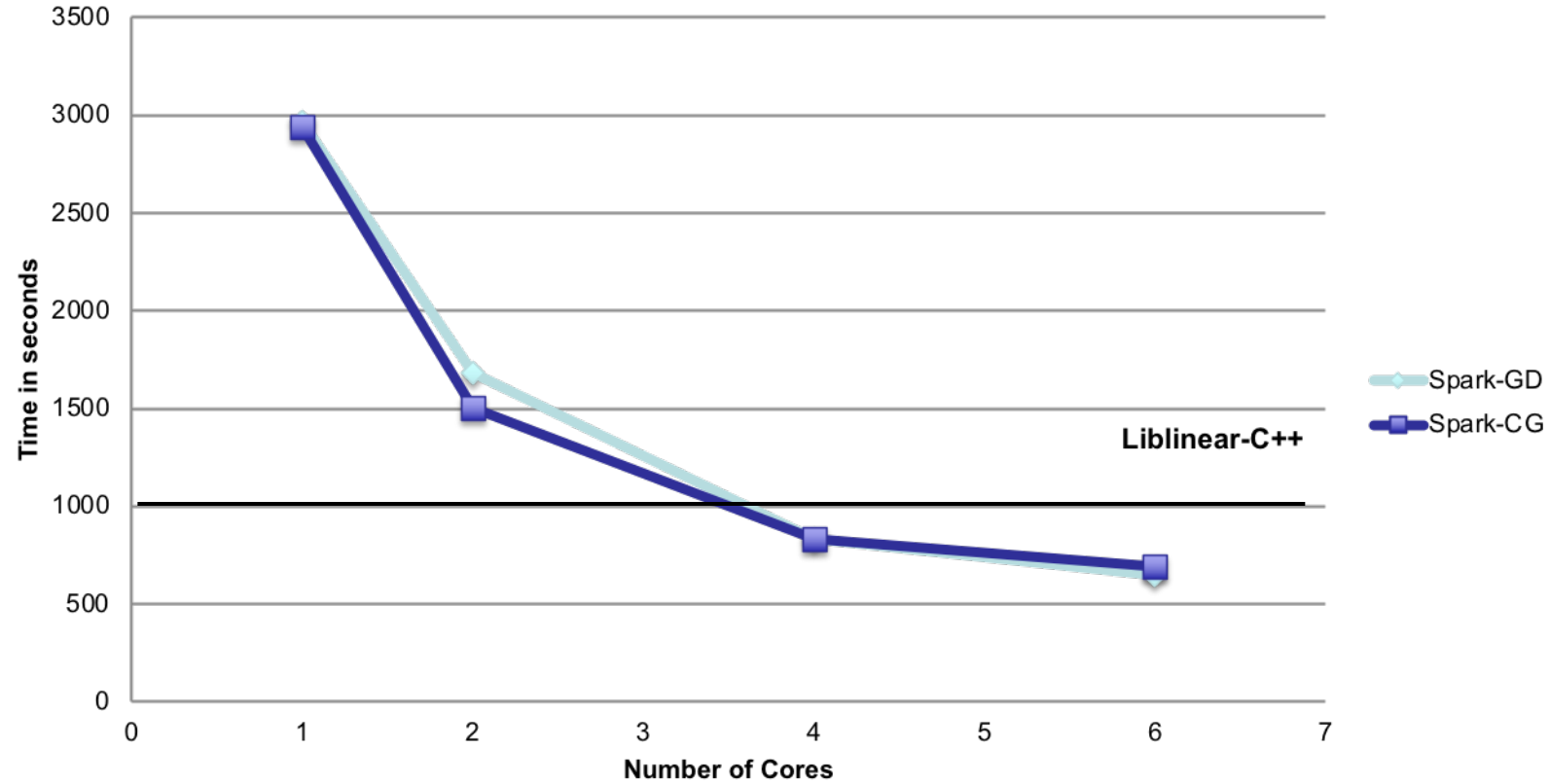
$$w^{t+1} = w^t - s \nabla l(w^t)$$

Spark Implementation

```
val x = loadData(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)
```

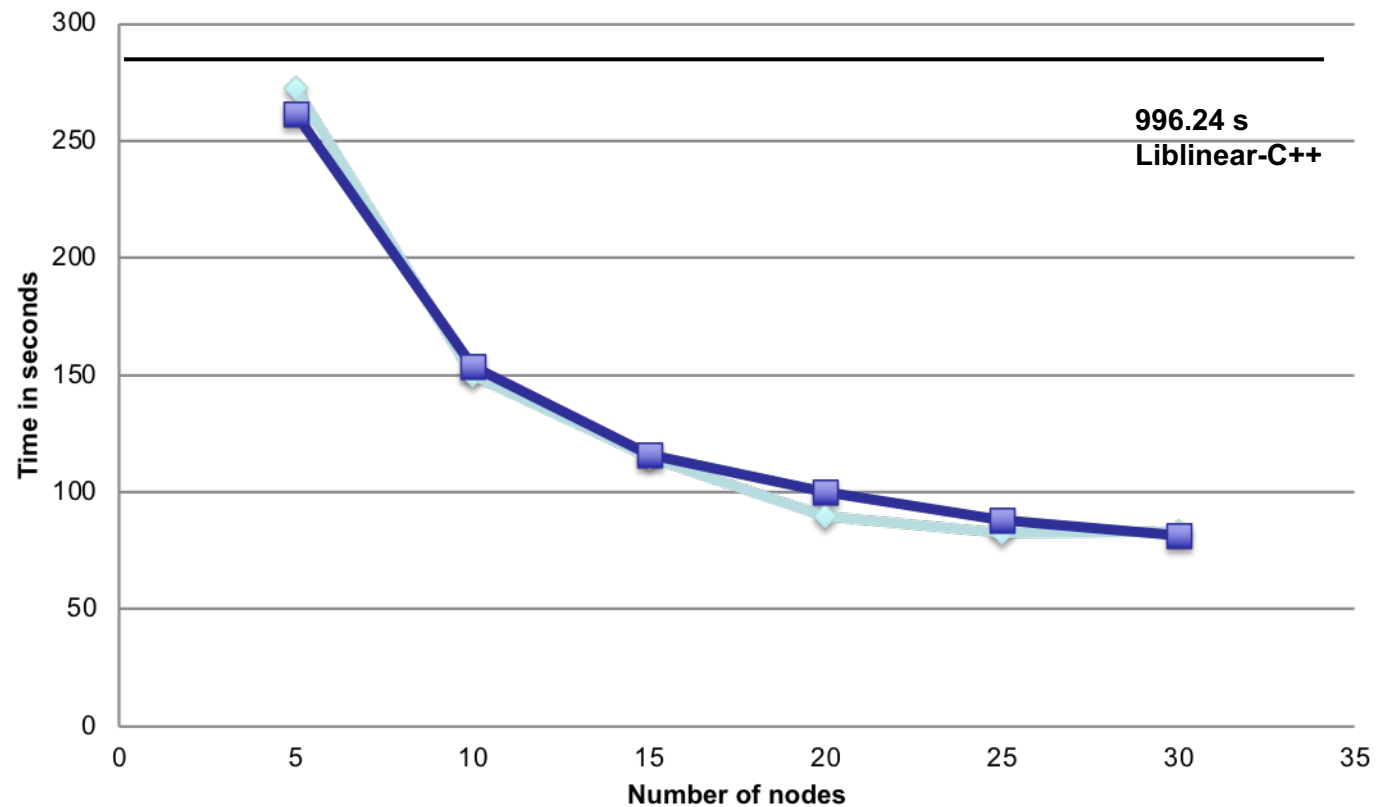
Scaleup with Cores

Epsilon (Pascal Challenge)



Scaleup with Nodes

Epsilon (Pascal Challenge)



Example: Matrix Multiplication

Matrix Multiplication

◆ Representation of Matrix:

- ◆ List <matrix id, Row index, Col index, Value>

- ◆ Size of matrices: First matrix (A): $m*k$, Second matrix (B): $k*n$

◆ Scheme:

- ◆ For each record: if matrix 1 : emit n records <(row ind, i), (col ind, value)>

- ◆ Else: emit m records <(i , col ind), (row ind, value)>

◆ GroupByKey: so that there are $m*n$ groups, each with $2*k$ records:

- ◆ (col ind, value) for first matrix or (row ind,value) for second matrix

◆ Foreach group and for each record (i, value):

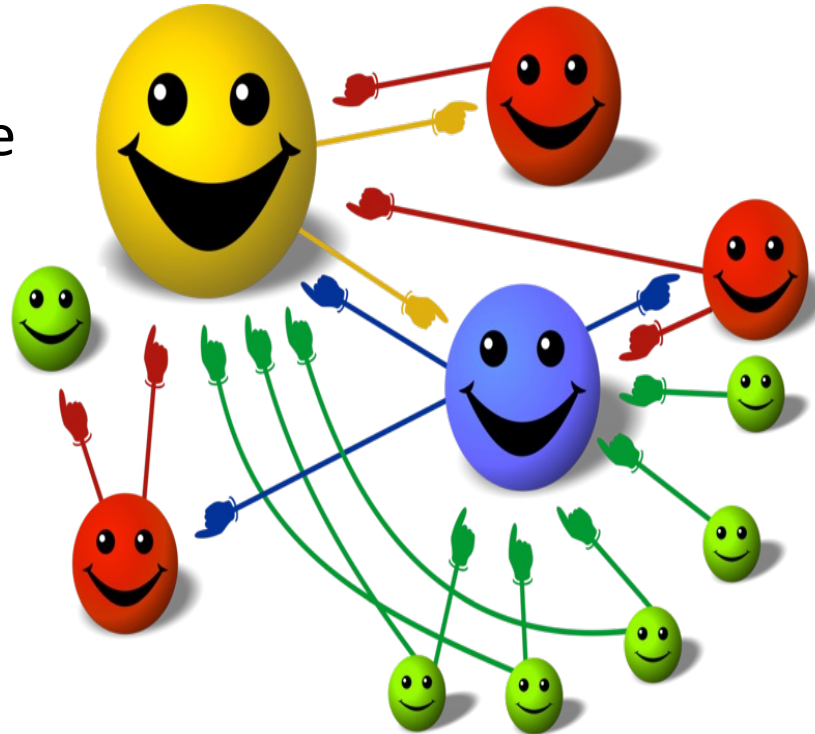
- ◆ Find another record (j, value) such that $i=j$

- ◆ Multiply the corresponding values and add to sum

Example: PageRank

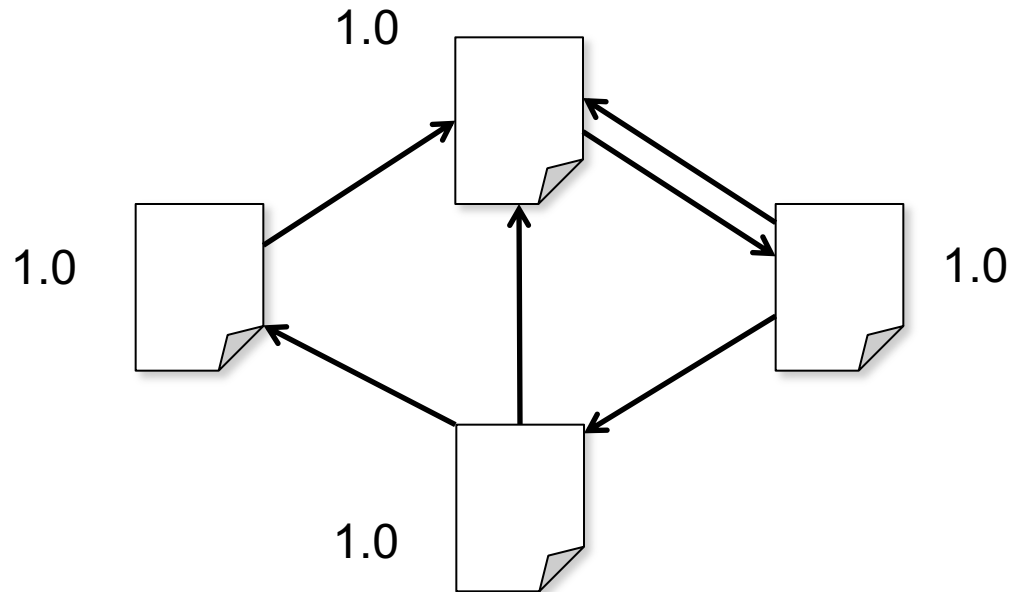
Basic Idea

- Give pages ranks (scores) based on links to them
 - Links from many pages
 - ➔ high rank
 - Link from a high-rank page
 - ➔ high rank



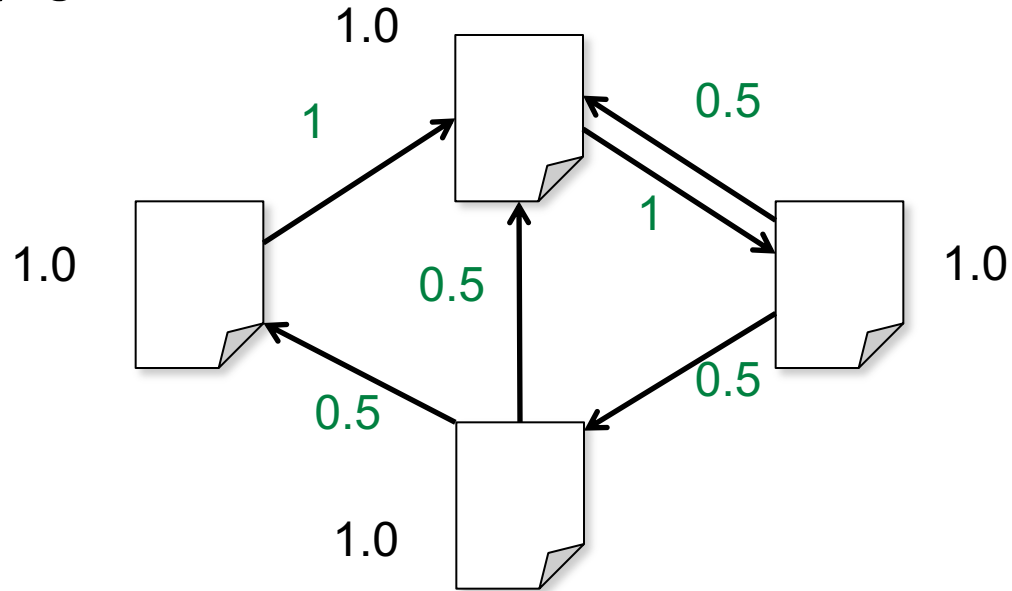
Algorithm

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 $0.15 + 0.85 \times \text{contribs}$



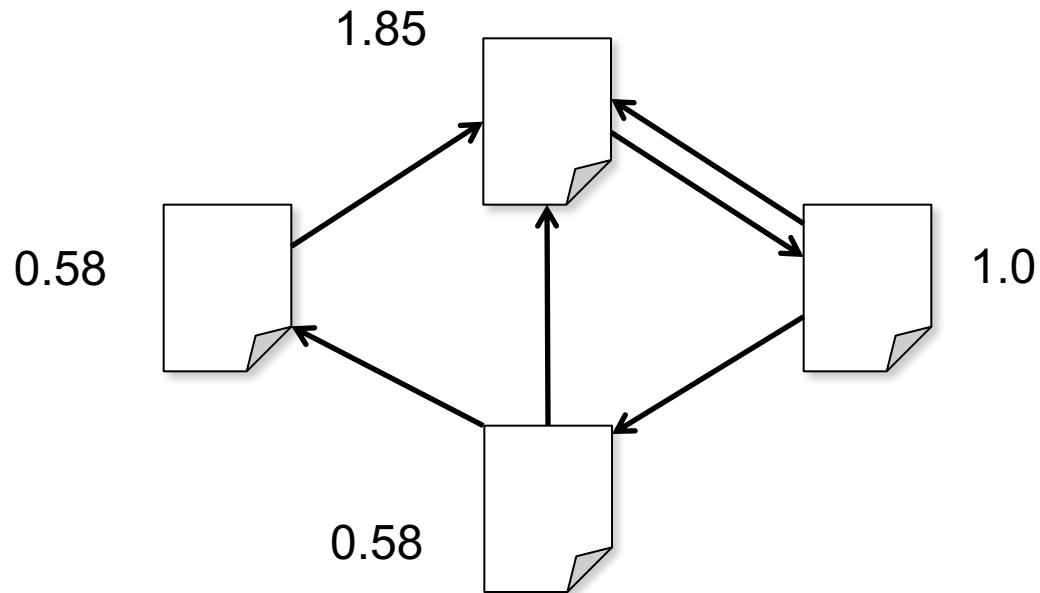
Algorithm

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



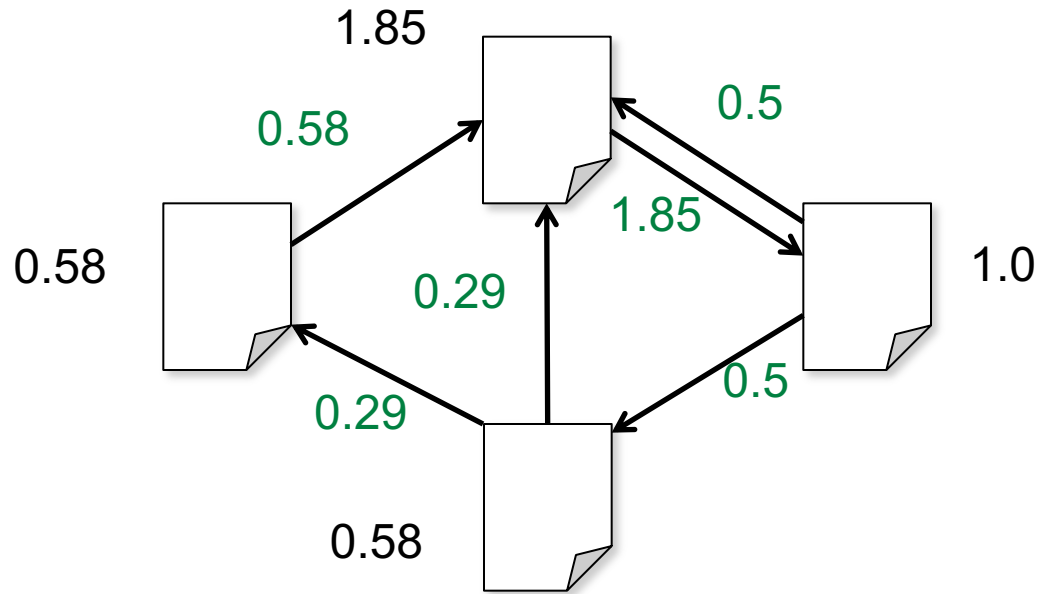
Algorithm

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



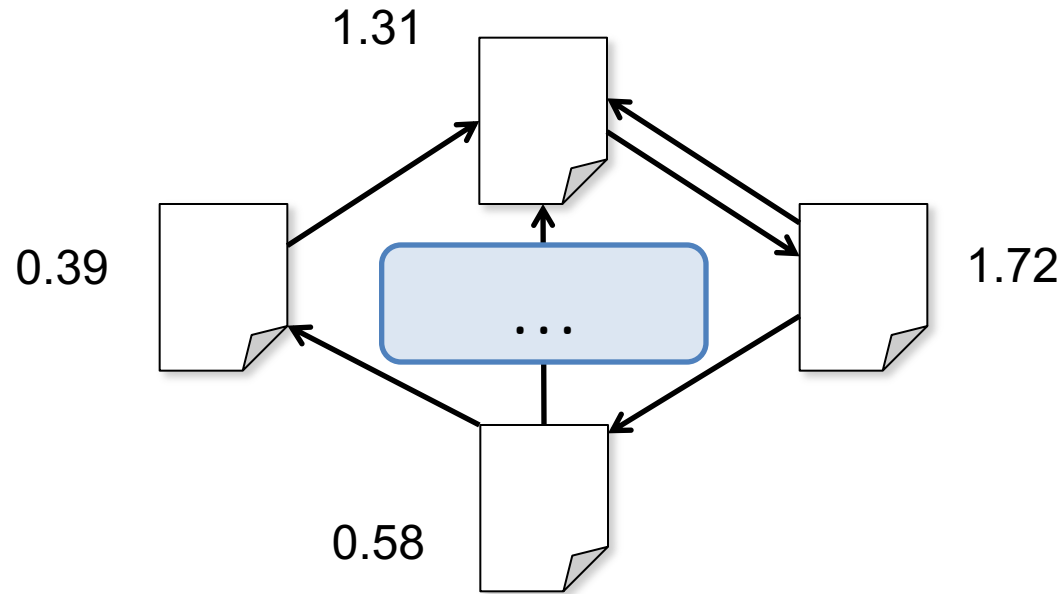
Algorithm

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



Algorithm

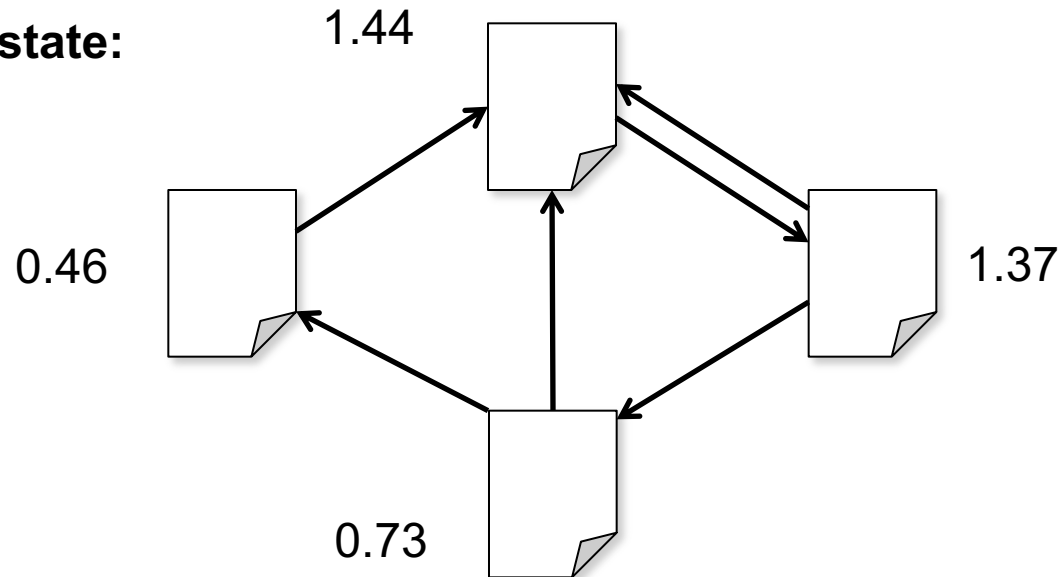
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

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

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.flatMap(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

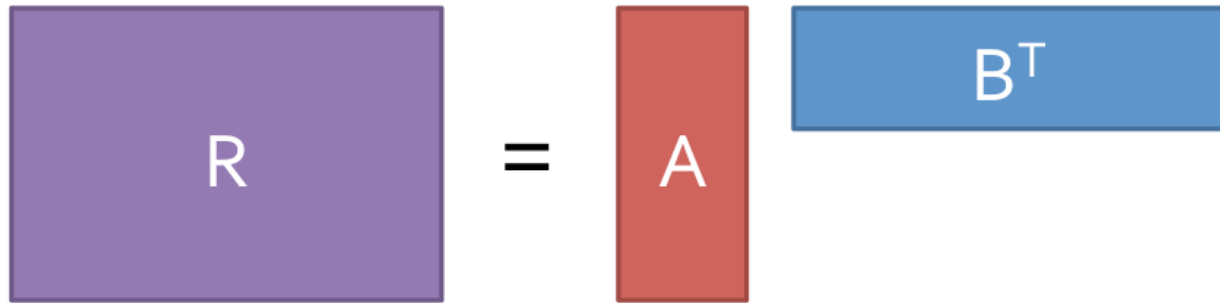
$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

← Movies →

↑ Users
↓

Matrix Factorization

Model R as product of user and movie matrices A and B of dimensions $U \times K$ and $M \times K$


$$R = A B^T$$

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”

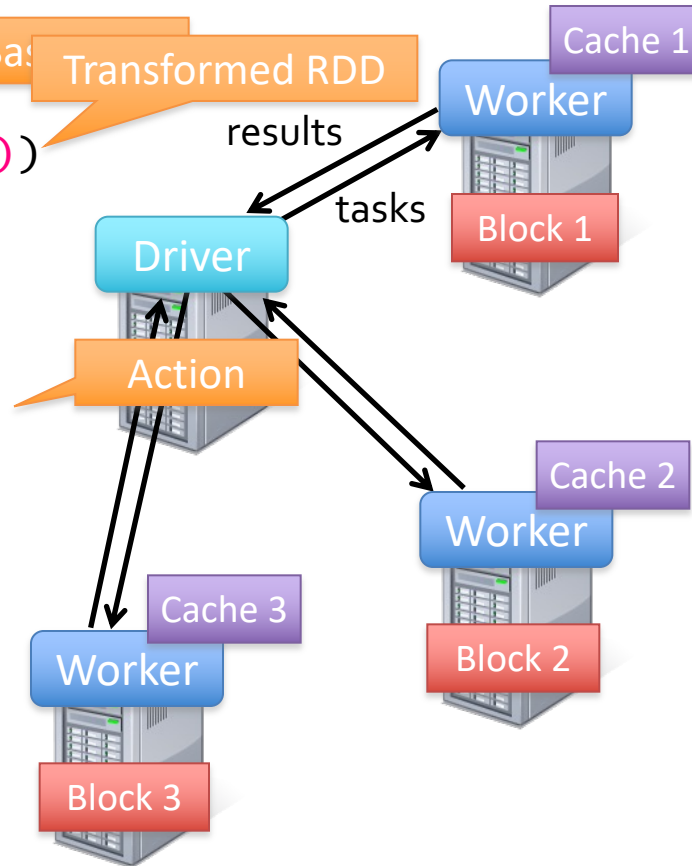
Caching of RDDs

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)



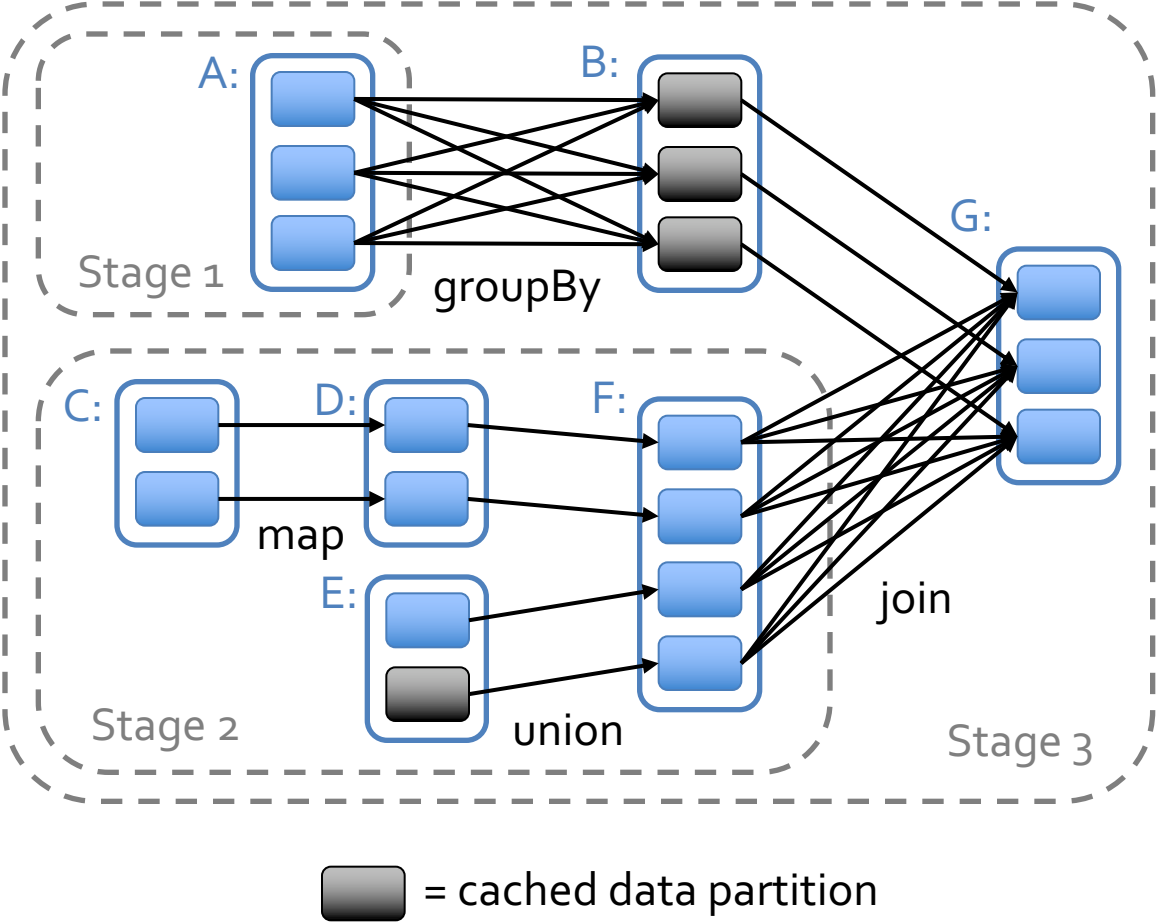
Spark Scheduler

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

Calculate the number of off-topic visits for a user.

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

def processNewLogs(logFileName: String) {

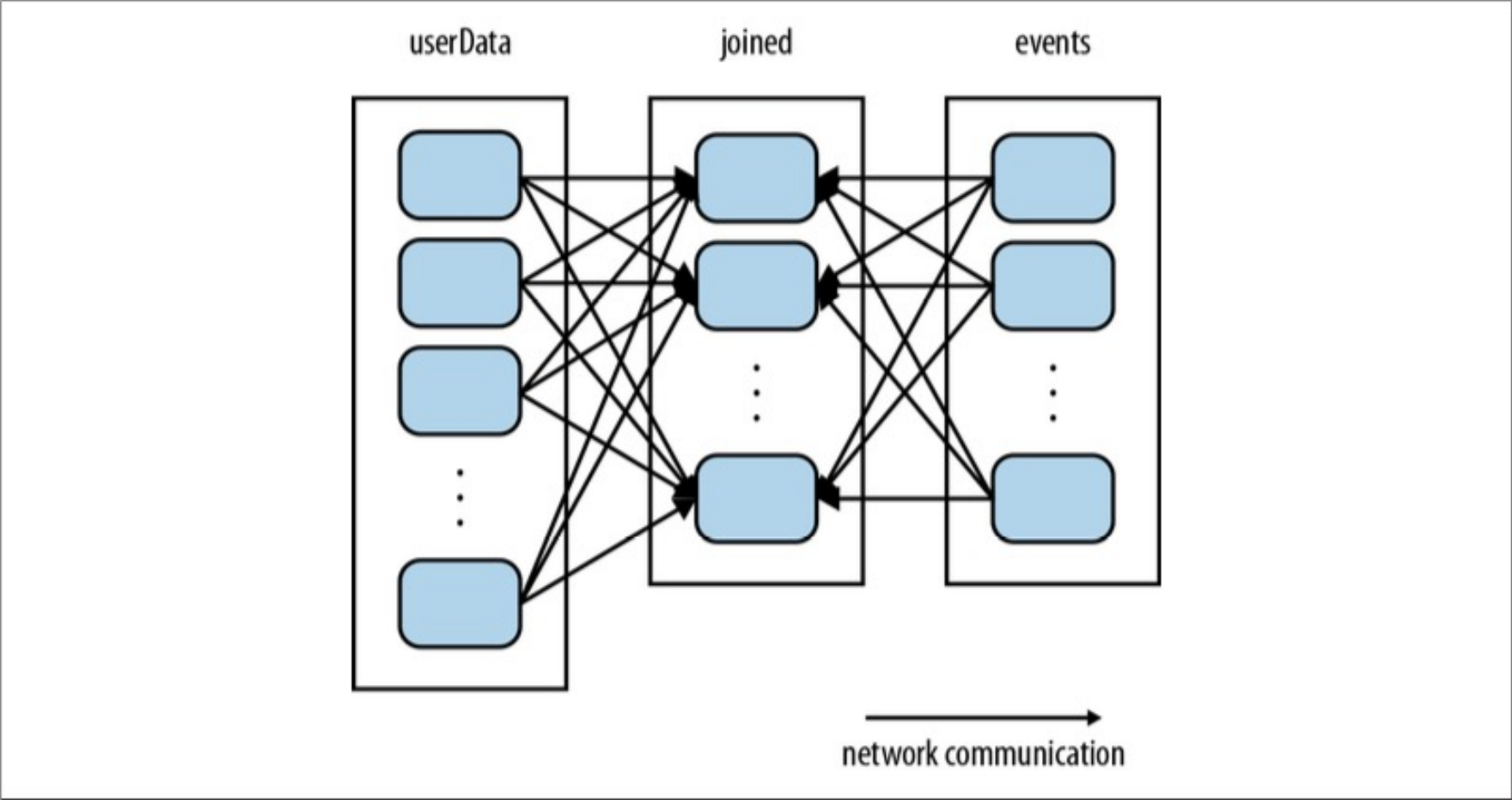
val events = sc.sequenceFile[UserID, LinkInfo](logFileName)

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()

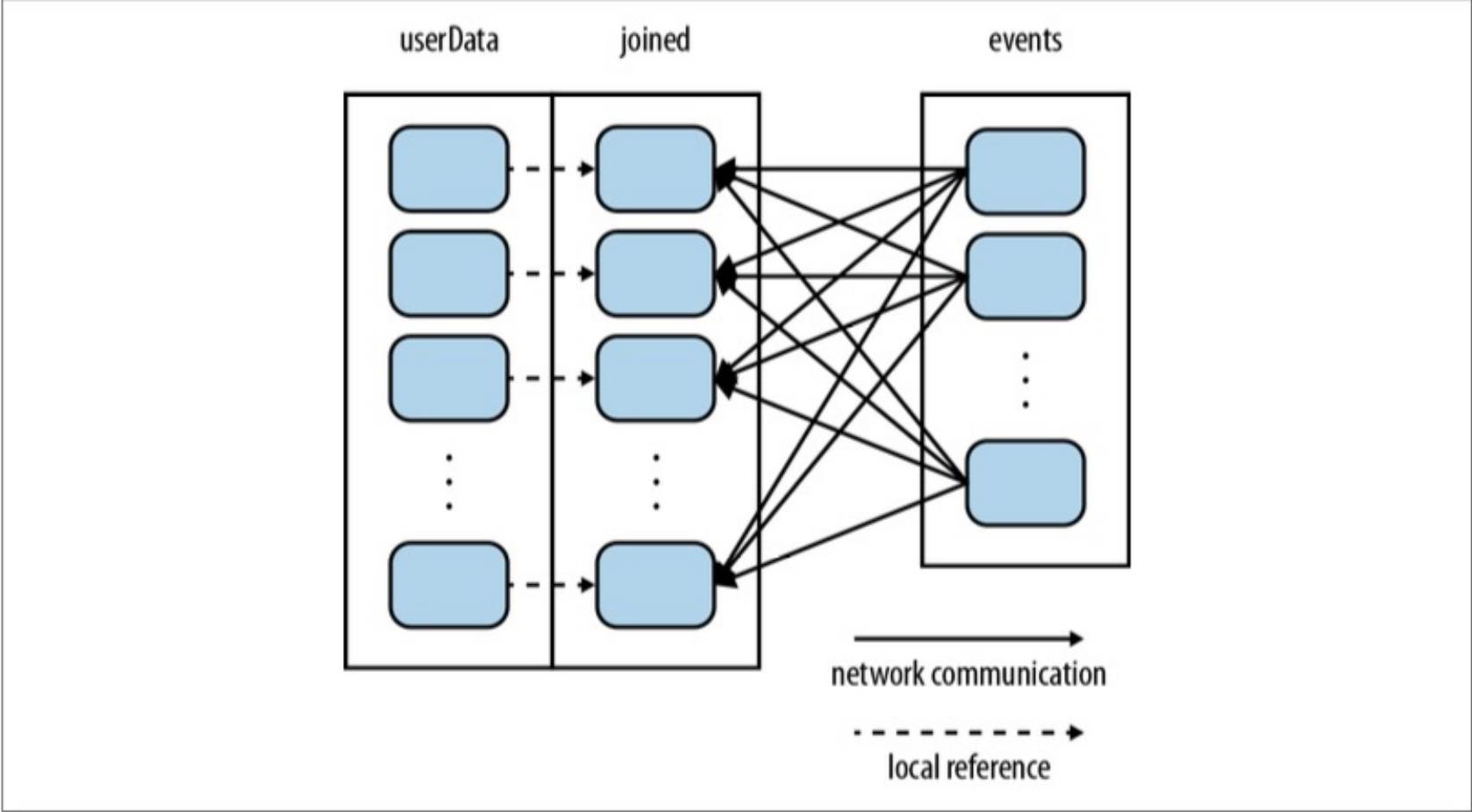
def processNewLogs(logFileName: String) {

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.