

CS60021: Scalable Data Mining

Spark

Sourangshu Bhattacharya

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:

- **in Java 1.5:**

```
Pair<Integer, String> p =  
    new Pair<Integer, String>(1, "Scala");
```

- **in Scala:**

```
val p = new MyPair(1, "scala");
```

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!`

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)`

“Statements”

- Scala’s “statements” should really be called “expressions,” because *every statement has a value*
- The value of many statements, for example the while loop, is `()`
 - `()` is a value of type `Unit`
 - `()` is the *only* value of type `Unit`
 - `()` basically means “*Nothing to see here. Move along.*”
- The value of a `if` or `match` statement is the last value computed
- The value of a block, `{...}`, is the last value computed in the block
- A statement is ended by the end of the line (not with a semicolon) unless it is obviously incomplete, or if the next line cannot begin a valid statement
 - For example, `x = 3 * (2 * y +` is obviously incomplete
 - Because Scala lets you leave out a lot of unnecessary punctuation, sometimes a line that you *think* is complete really isn’t complete (or vice versa)
- You *can* end statements with semicolons, but that’s not good Scala practice

Familiar statement types

- These are the same as in Java, but have a value of ():
 - *variable* = *expression* // also +=, *=, etc.
 - while (*condition*) { *statements* }
 - do { *statements* } while (*condition*)
- These are the same as in Java, but may have a useful value:
 - { *statements* }
 - The value of the block is the last value computed in it
 - if (*condition*) { *statements* } else { *statements* }
 - The value is the value of whichever block is chosen
 - If the value is to be used, both blocks should have the same type, otherwise the type of the result is the “least upper bound” of the two types
 - if (*condition*) { *statements* }
 - The value is the value of the last statement executed, but its type is **Any** – if you want a value, you really should use an **else**
- As in Java, braces around a single statement may be omitted

Arrays

- Arrays in Scala are **parameterized types**
 - `Array[String]` is an Array of Strings, where `String` is a **type parameter**
 - In Java we would call this a “generic type”
- When no initial values are given, `new` is required, along with an explicit type:
 - `val ary = new Array[Int](5)`
- When initial values are given, `new` is not allowed:
 - `val ary2 = Array(3, 1, 4, 1, 6)`
- Array syntax in Scala is just object syntax
- 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

Simple `List` operations

- 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?)
 - `value :: list` returns a list with `value` appended to the front
 - `list1 ::: list2` appends (“concatenates”) the two lists
 - `list.contains(value)` (or `list contains value`) tests whether `value` is in `list`
- Many operations on Lists also work on other kinds of Sequences
- An operation on a Sequence *may* return a Sequence of a different type
 - ```
scala> "abc" :: List(1, 2, 3)
res22: List[Any] = List(abc, 1, 2, 3)
```
  - This happens because `Any` is the only type that can hold both integers and strings
- There are over 150 built-in operations on Lists—use the API!

# Tuples

- Scala has tuples, up to size 22 (why 22? I have no idea.)
  - `scala> val t = Tuple3(3, "abc", 5.5)`  
`t: (Int, java.lang.String, Double) = (3,abc,5.5)`
  - `scala> val tt = (3, "abc", 5.5)`  
`tt: (Int, java.lang.String, Double) = (3,abc,5.5)`
- Tuples are referenced *starting from 1*, using `_1`, `_2`, ...
  - `scala> val t = ('a', "one", 1)`  
`t: (Char, String, Int) = (a,one,1)`
  - `scala> t._2`  
`res3: String = one`
- Tuples, like Lists, are immutable
- Tuples are a great way to return more than one value from a method

# Simple method definitions

- ```
def isEven(n: Int) = {  
    val m = n % 2  
    m == 0  
}
```

 - The result is the last value (in this case, a Boolean)
 - This is really kind of poor style; the extra variable isn't needed
- ```
def isEven(n: Int) = n % 2 == 0
```

  - This is much better style
  - The result is just a single expression, so no braces are needed
- ```
def countTo(n: Int) {  
    for (i <- 1 to 10) { println(i) }  
}
```

 - It's good style to omit the = when the result is ()
 - If you omit the =, the result *will* be ()
 - You need either braces or an =; you can't leave out both

Methods and functions

- A **method** is a function that belongs to an object
 - Methods usually use, and manipulate, the fields of an object
- A function does not belong to any object
 - The inputs to a function are, or should be, just its parameters
 - The result of calling a function is, or should be, just its return value
- Scala can sometimes accept a method where a function is expected
 - A method can sometimes be “converted” to a function by following with an underscore (for example, `isEven _`)

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 define a function, you must specify the types of each of its parameters
- Therefore, 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
```



# 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 ll = List(2, 3, 5, 7, 11)`  
`ll: List[Int] = List(2, 3, 5, 7, 11)`
  - `scala> ll map double`  
`res5: List[Int] = List(4, 6, 10, 14, 22)`
  - `scala> ll map (n => 3 * n)`  
`res6: List[Int] = List(6, 9, 15, 21, 33)`
  - `scala> ll map (n => n > 5)`  
`res8: List[Boolean] = List(false, false, false, true, true)`
- **filter** applies a one-parameter test to every element of a List, returning a List of those elements that pass the test
  - `scala> ll filter(n => n < 5)`  
`res10: List[Int] = List(2, 3)`
  - `scala> ll 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.

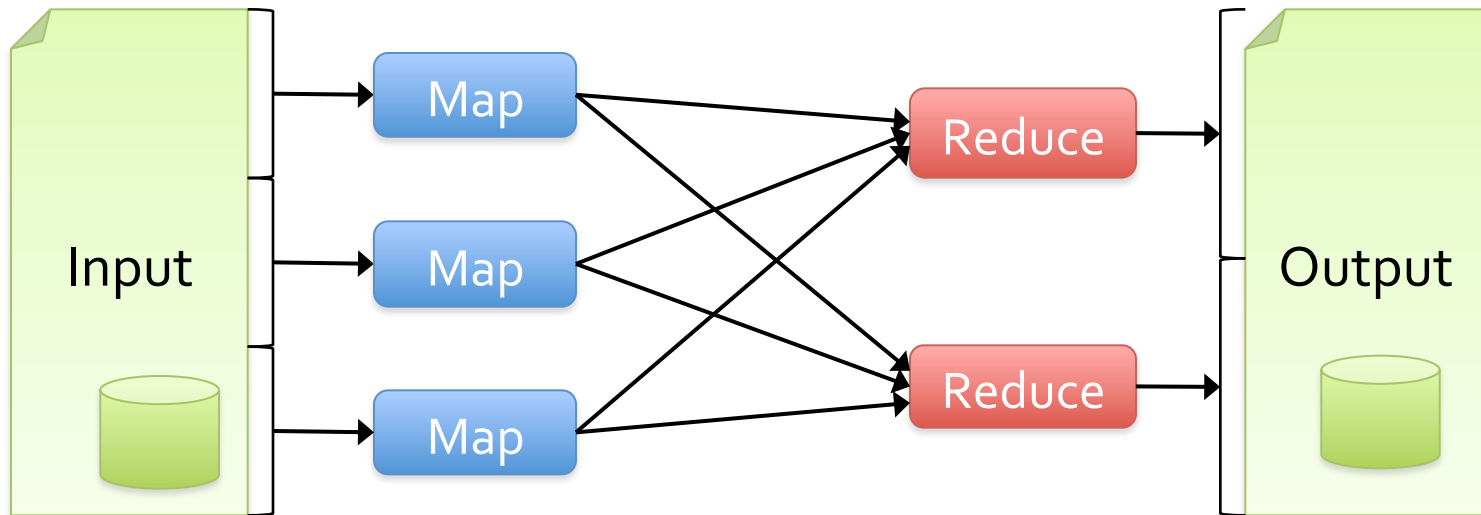
# Background

- 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

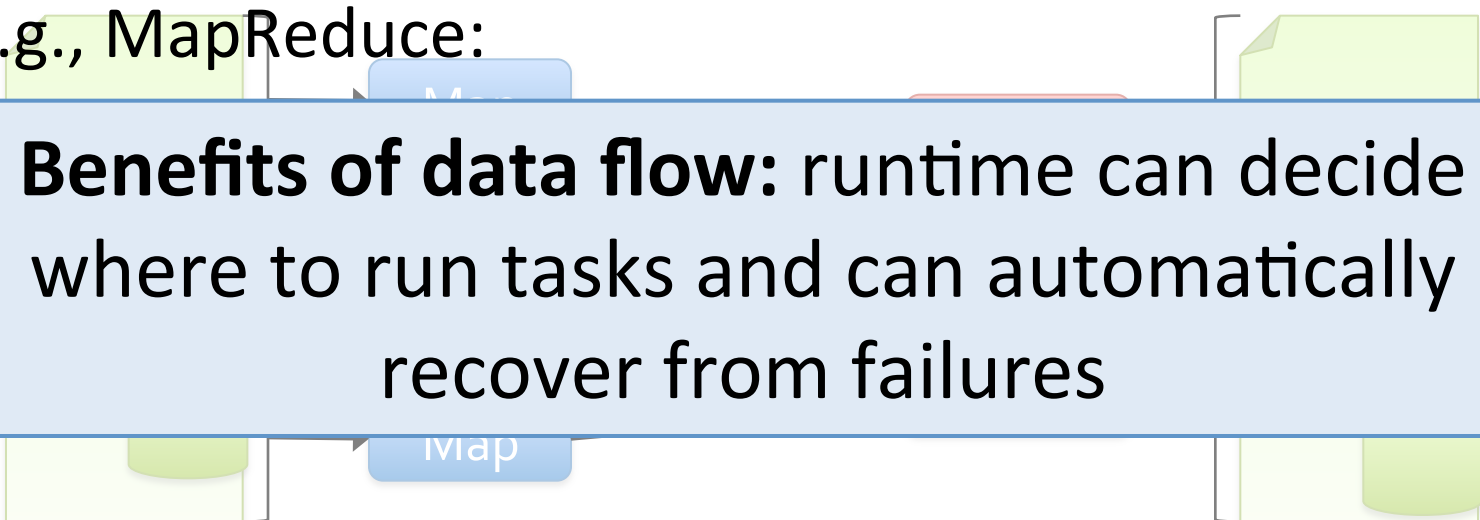
Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



# Motivation

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



**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)

map  
filter  
sample  
union  
groupByKey  
reduceByKey  
join  
cache  
...

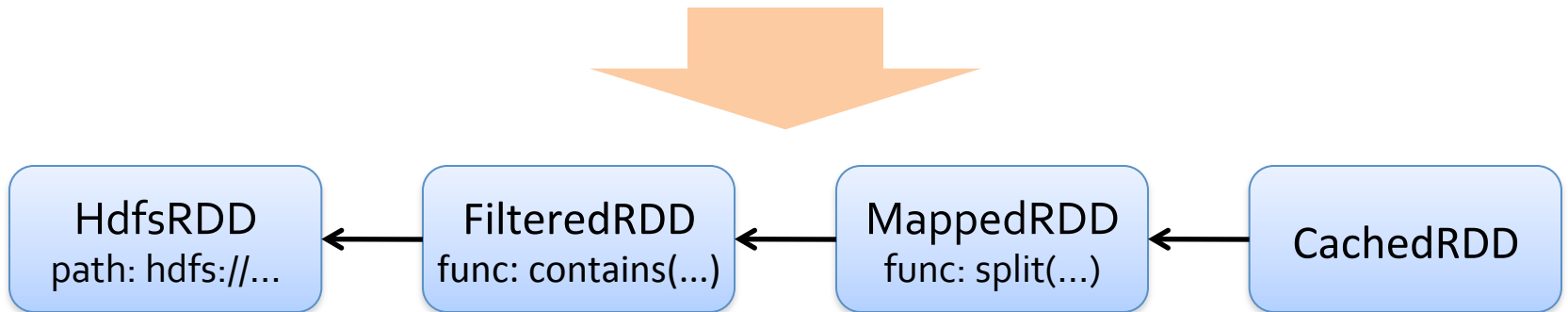
## Actions (return a result to driver)

reduce  
collect  
count  
save  
lookupKey  
...

# RDD Fault Tolerance

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

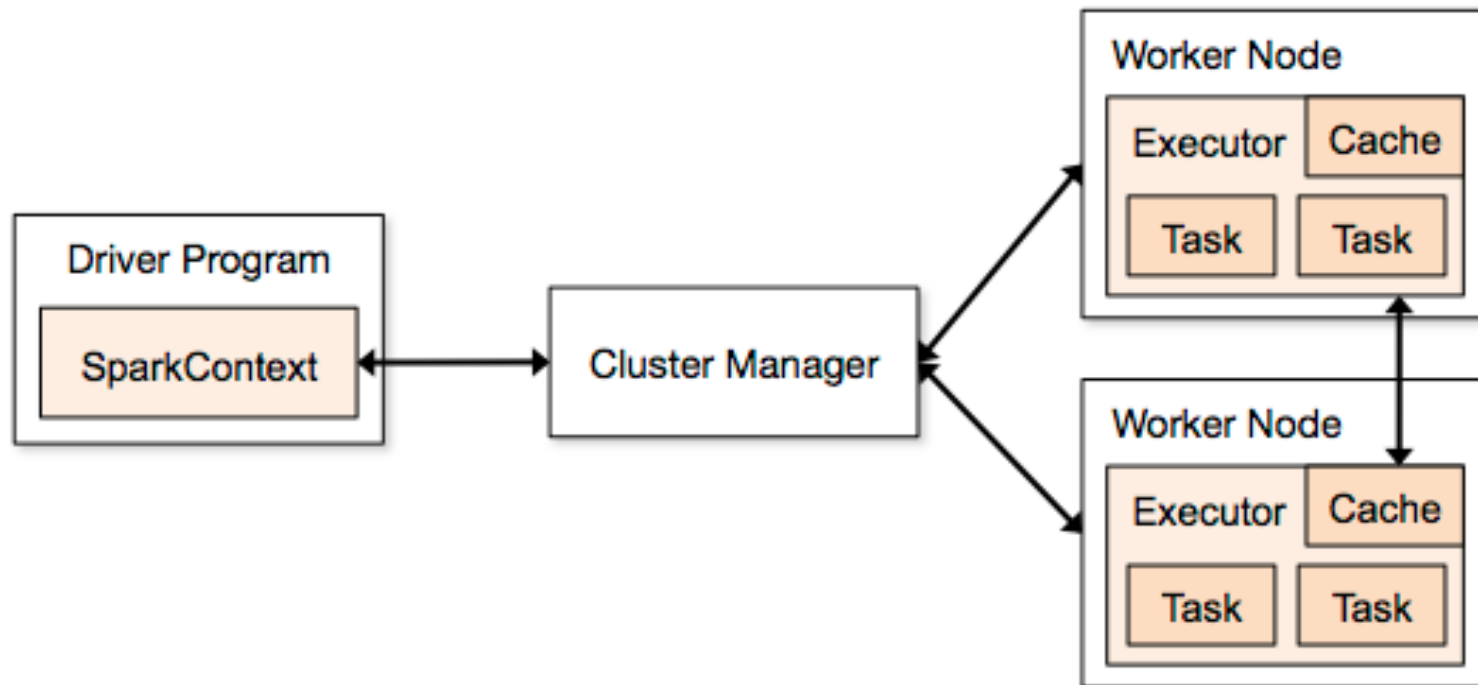
```
cachedMsgs = textFile(...).filter(_.contains("error"))
 .map(_.split('\t')(2))
 .cache()
```



# 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





# 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))
```

# Word Count in Spark

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

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

counts.save("hdfs://...")
```

# 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.  $y \in \{+1, -1\}$
- 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:

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

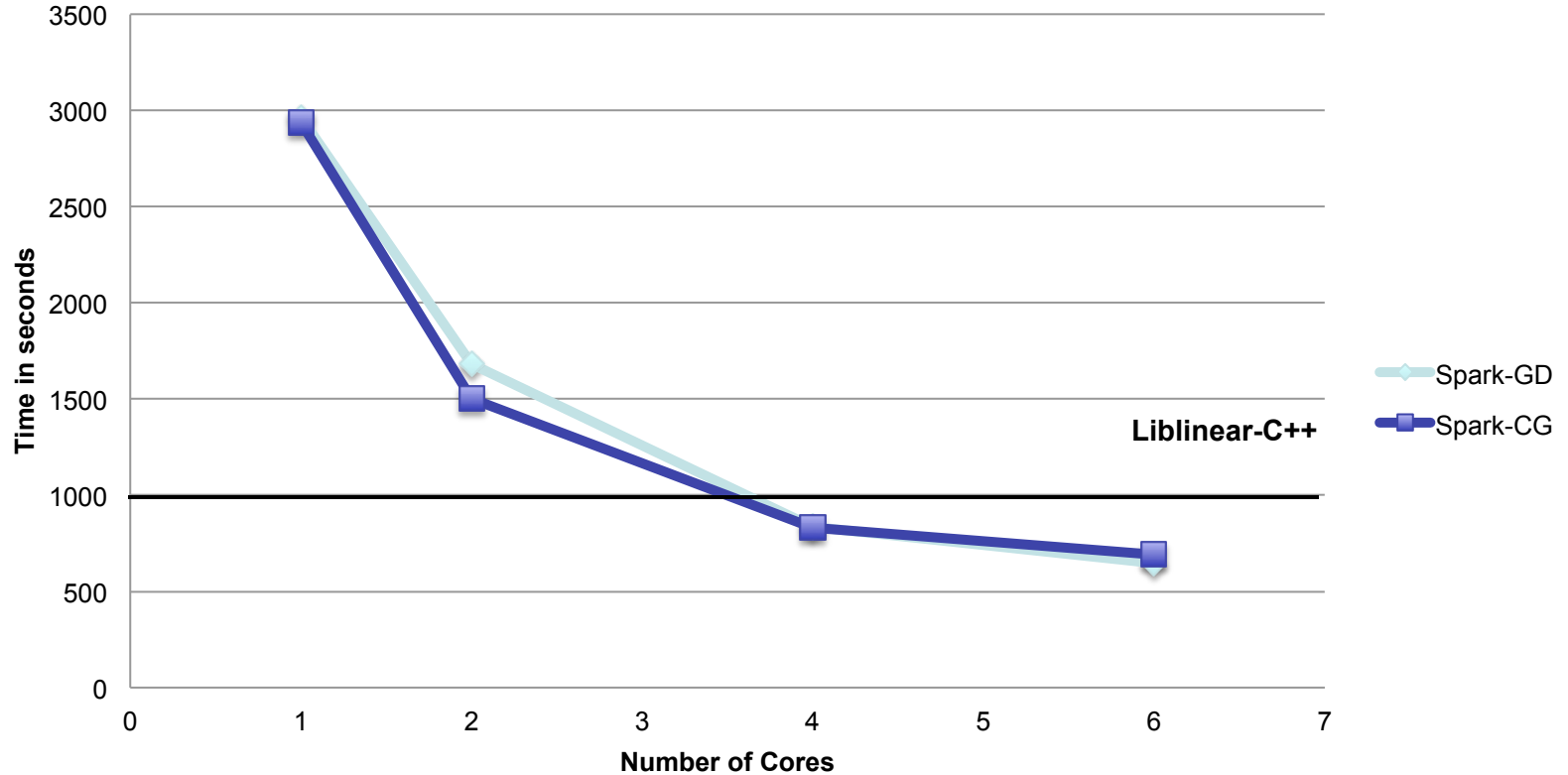
# Spark Implementation

```
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)
```



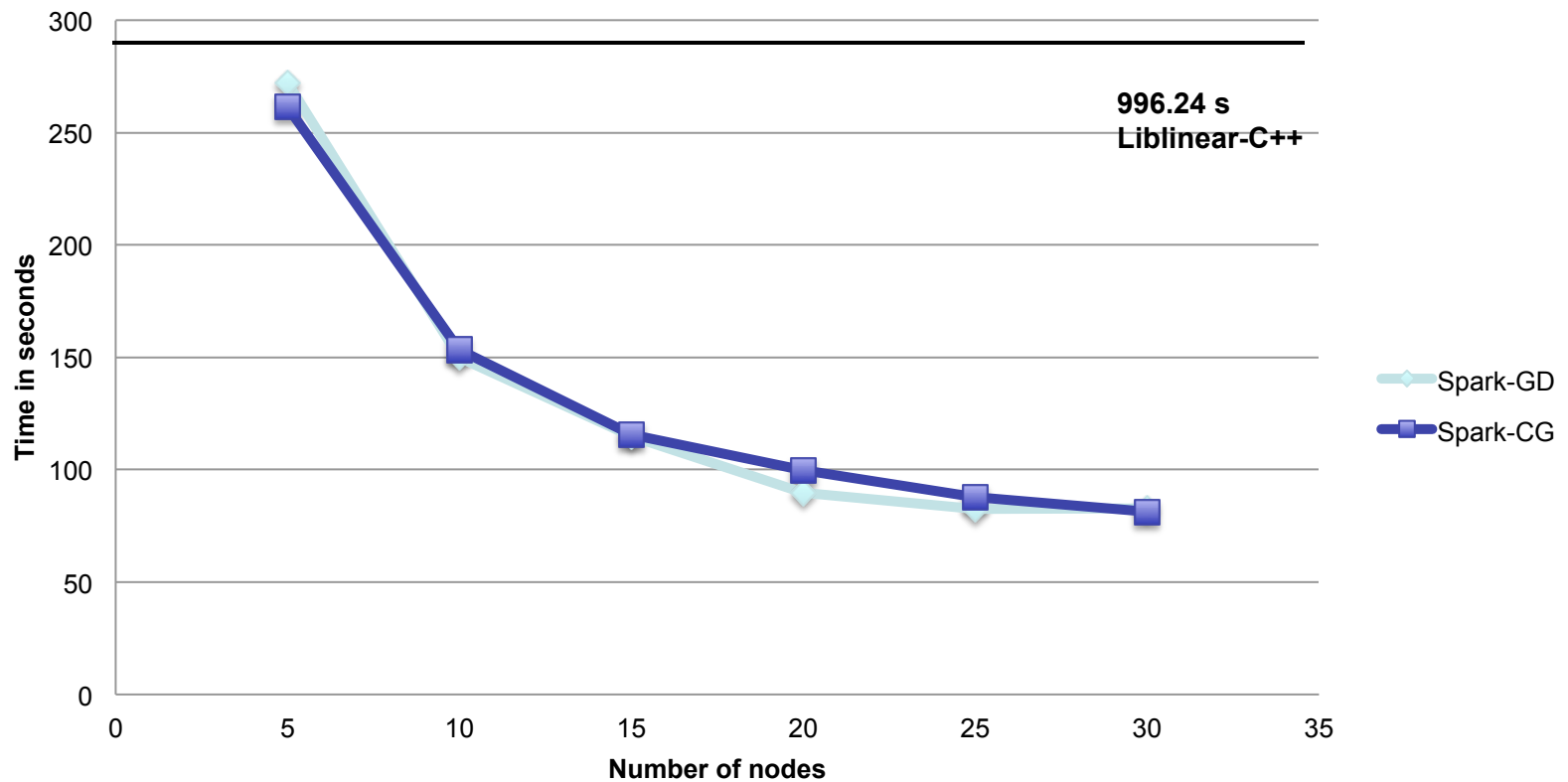
# Scaleup with Cores

## Epsilon (Pascal Challenge)



# Scaleup with Nodes

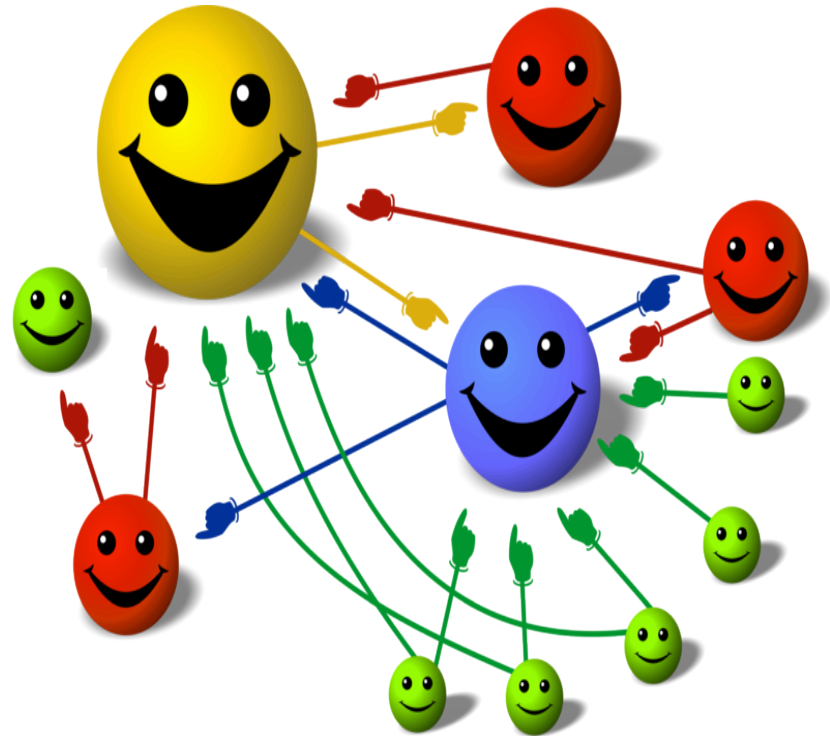
## Epsilon (Pascal Challenge)



# Example: PageRank

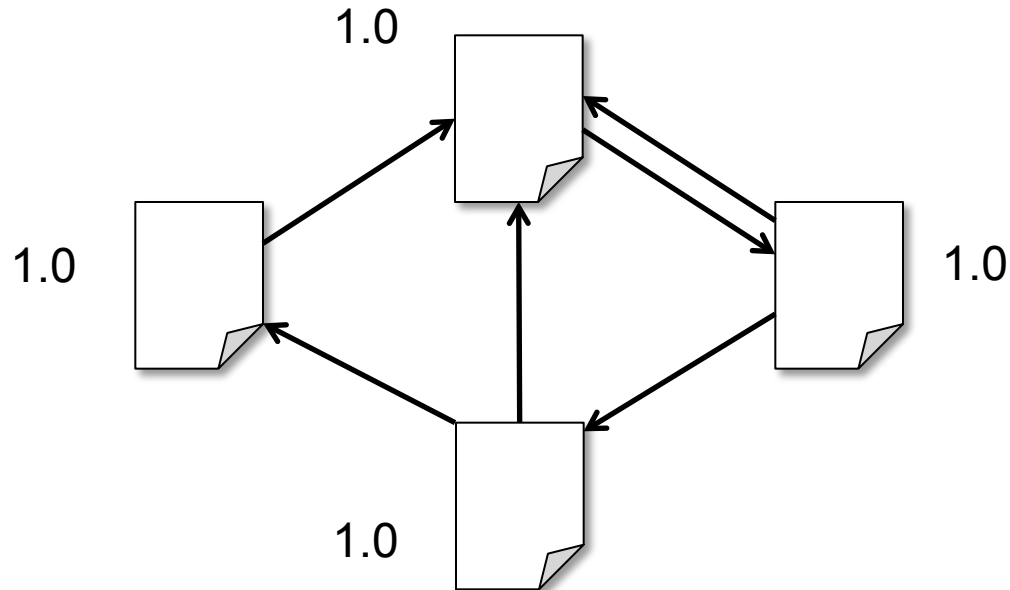
# Basic Idea

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



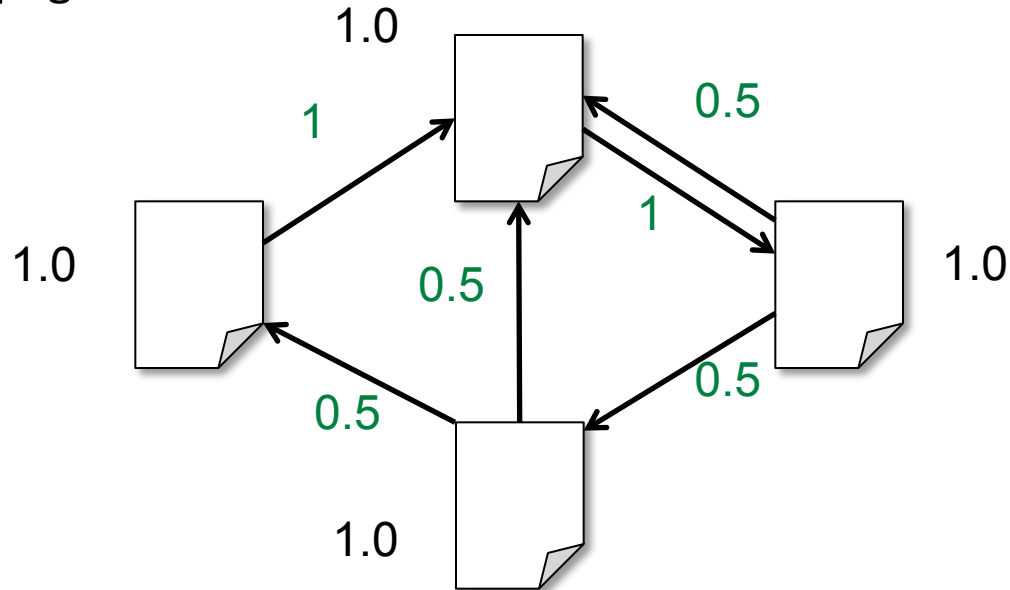
# Algorithm

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



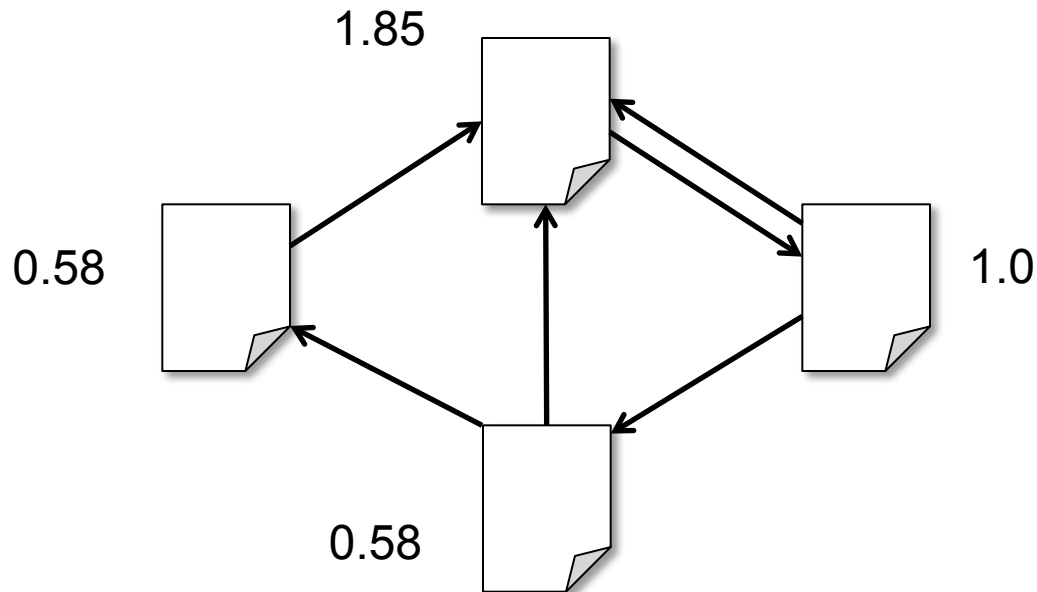
# Algorithm

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



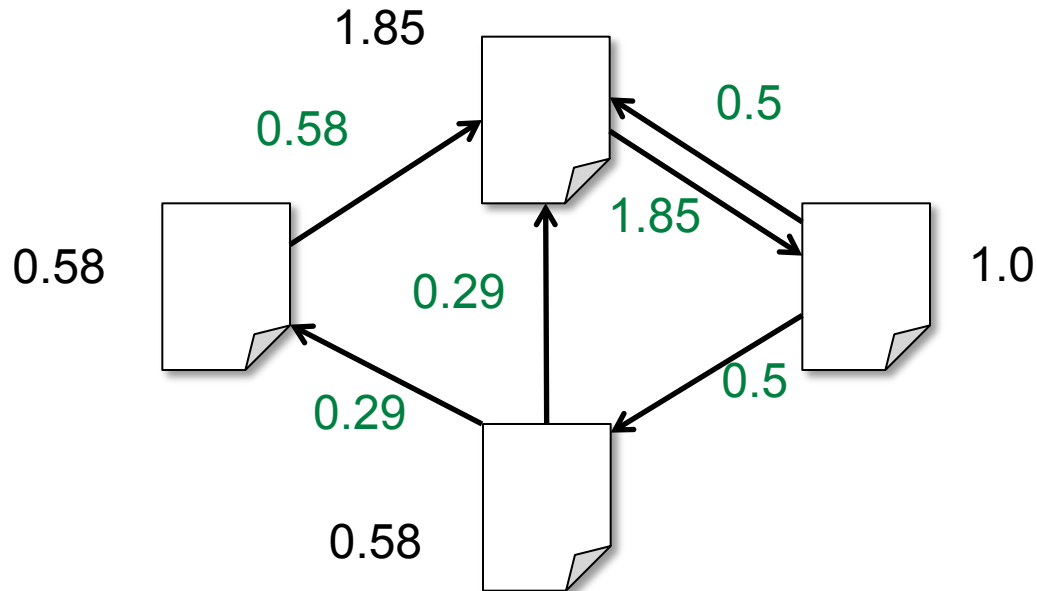
# Algorithm

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

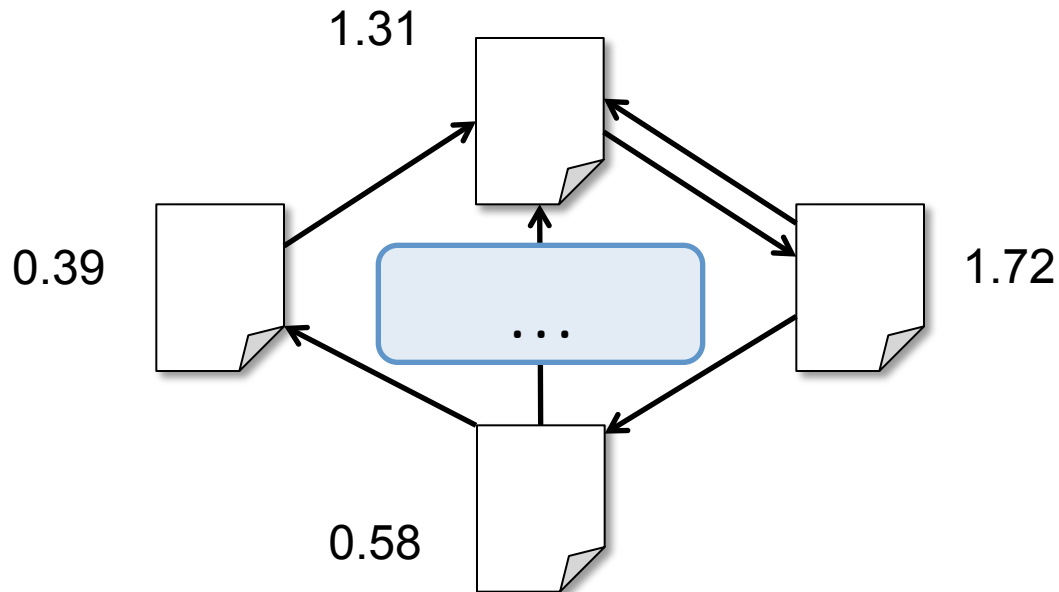
1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$





# Algorithm

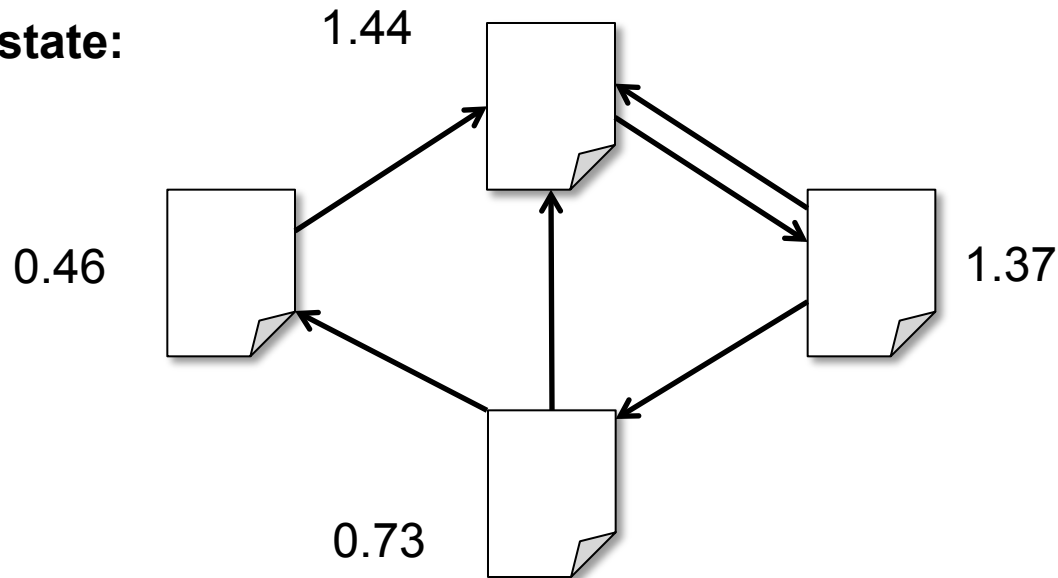
1. Start each page at a rank of 1
2. On each iteration, have page  $p$  contribute  $\text{rank}_p / |\text{neighbors}_p|$  to its neighbors
3. Set each page's rank to  $0.15 + 0.85 \times \text{contribs}$



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

**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(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

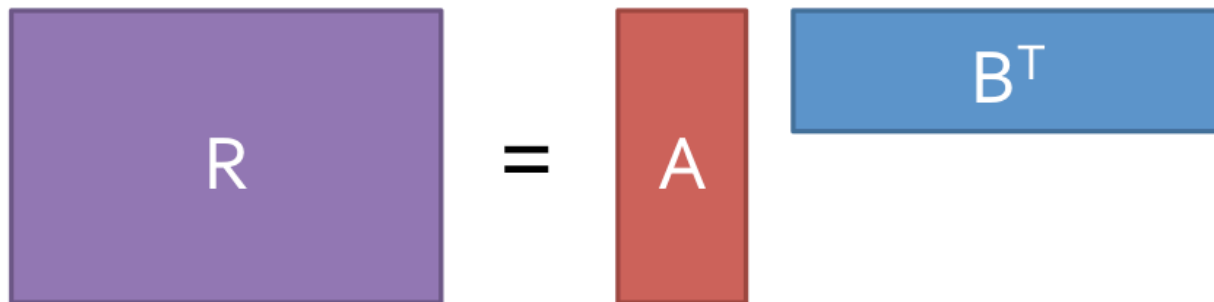
$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

← Movies →

↑ Users  
↓

# Matrix Factorization

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



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”

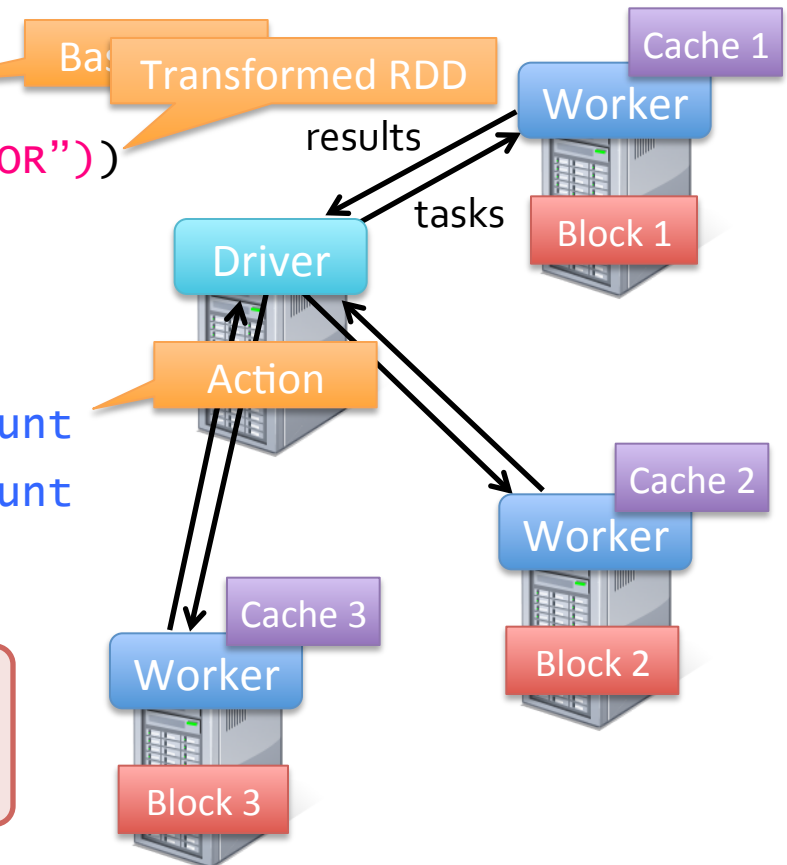
# Example: Log Mining

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()

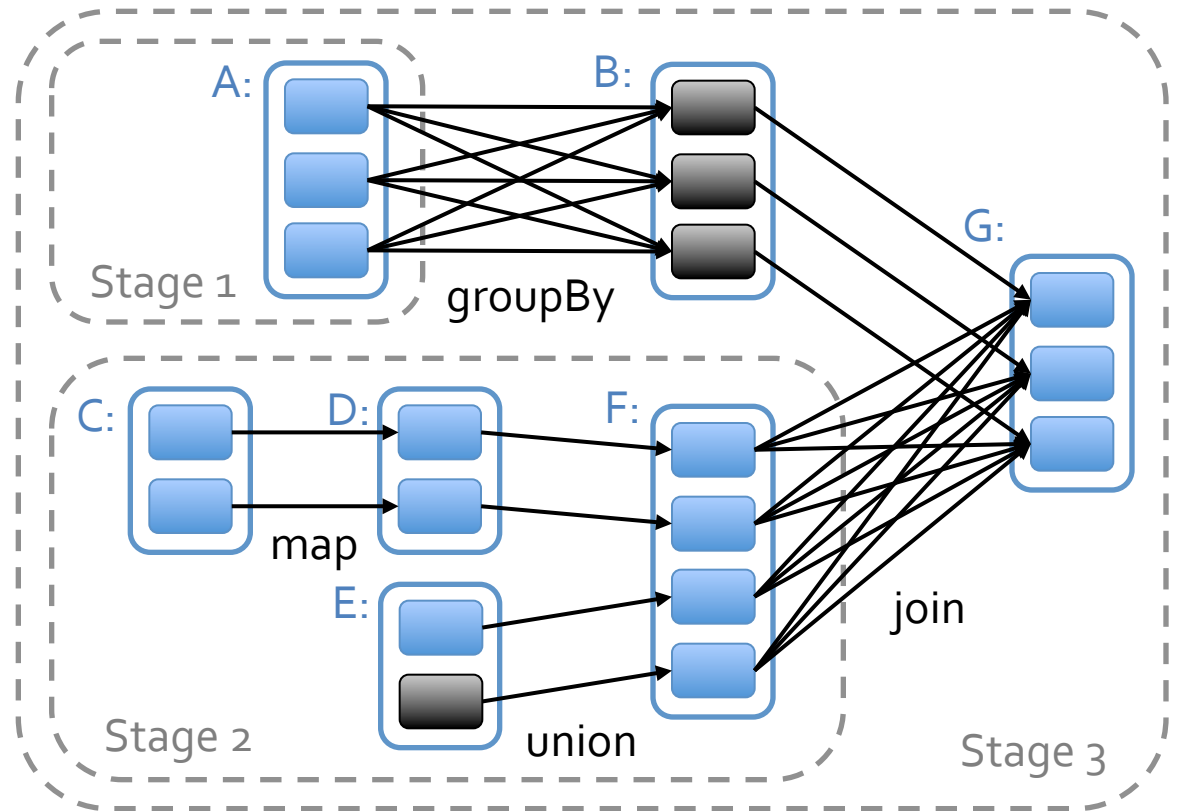
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```

**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



# User Log Mining

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

def processNewLogs(logFileName: String) {

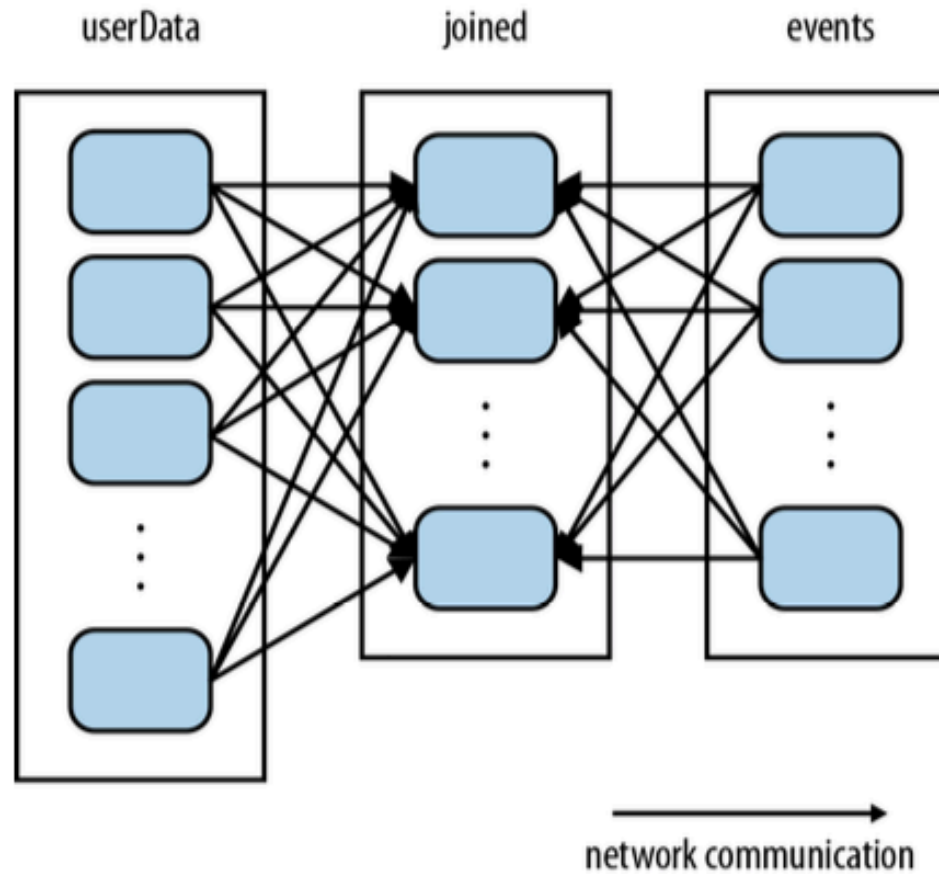
val events = sc.sequenceFile[UserID, LinkInfo](logFileName)

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)
```

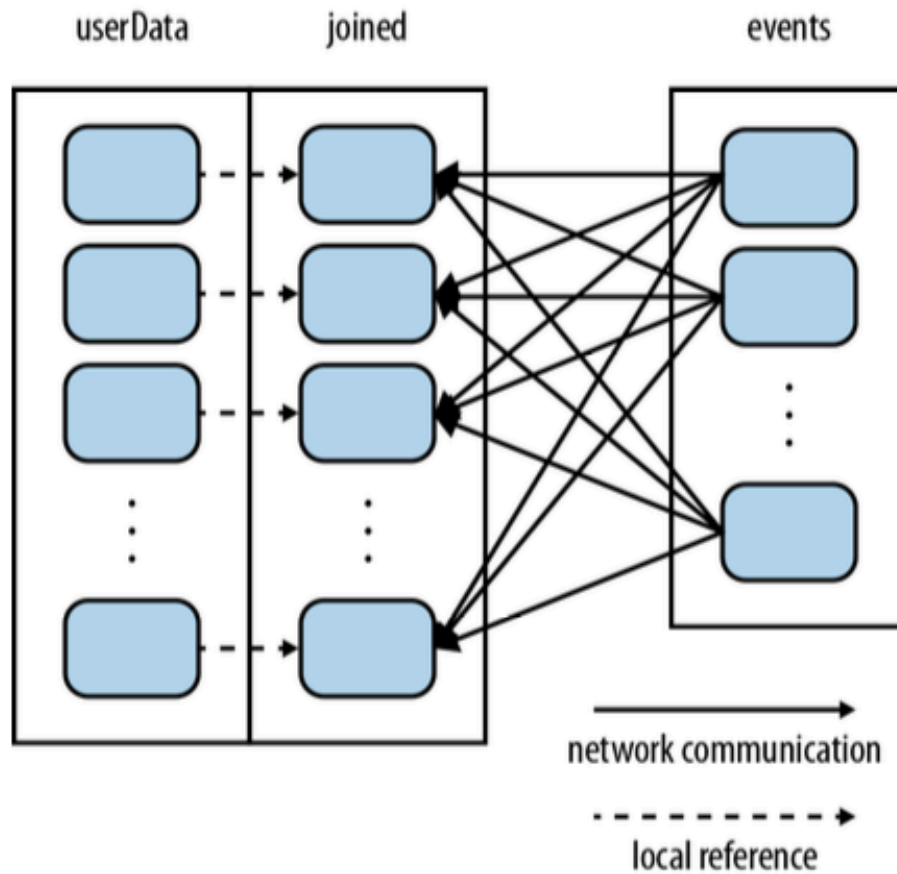
```
val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
```

```
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)
```

# 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 must be computed. This requires computing its parent RDDs as well. Spark's scheduler submits a job to compute all needed RDDs. That job will have one or more stages, which are parallel waves of computation composed of tasks. 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()).