

Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
 - Multiple steps, including iterations
 - Stores intermediate results in main memory
 - Closer to relational algebra (familiar to you)
- Details:
<http://spark.apache.org/examples.html>

Spark Interface

- Spark supports a Scala interface (and others)
- Scala = extension of Java with functions/closures
- We will illustrate Scala/Spark in the lectures
- Spark also supports a SQL interface, and compiles SQL to its Scala interface

RDD

- RDD = Resilient Distributed Datasets
 - A distributed relation, together with its **lineage**
 - **Lineage**: expression that says how that relation was computed
= a relational algebra plan
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost.
However, the driver (=master node) knows the lineage, and will simply re-compute the lost partition of the RDD

Programming in Spark

- A Spark/Scala program consists of:
 - Transformations (map, reduce, join...). Lazy
 - Actions (count, reduce, save...). Eager
- `RDD[T]` = an RDD collection of type T
 - Partitioned, recoverable (through lineage), not nested
- `Seq[T]` = a Scala sequence
 - Local to a server, may be nested

Example

Given a large log file `hdfs://logfile.log`
retrieve all lines that:

- Start with ERROR
- Contain the string “sqlite”

```
lines = spark.textFile("hdfs://logfile.log");  
  
errors = lines.filter(_.startsWith("ERROR"));  
  
sqlerrors = errors.filter(_.contains("sqlite"));  
  
sqlerrors.collect()
```

Example

Given a large log file `hdfs://logfile.log`
retrieve all lines that:

- Start with ERROR
- Contain the string “sqlite”

```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

Transformation:
Not executed yet...

Action:
triggers execution
of entire program

MapReduce Again...

Steps in Spark resemble MapReduce:

- `col.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection, and returns collection with those `x` where `p(x) = true`
- `col.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection, and returns a new partitioned collection
- Etc

Scala Primer

- Functions with one argument:

`_.contains("sqlite")`

`_ > 6`

- Functions with more arguments

`(x => x.contains("sqlite"))`

`(x => x > 6)`

`((x,y) => x+3*y)`

- Closures (functions with free variables):

`var x = 5; rdd.filter(_ > x)`

`var s = "sqlite"; rdd.filter(x => x.contains(s))`

Persistence

```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

Persistence

RDD:

hdfs://logfile.log

filter(_.startsWith("ERROR"))
filter(_.contains("sqlite"))

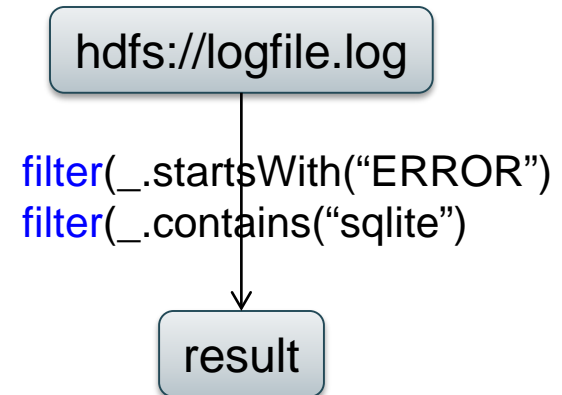
result

```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

Persistence

RDD:



```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

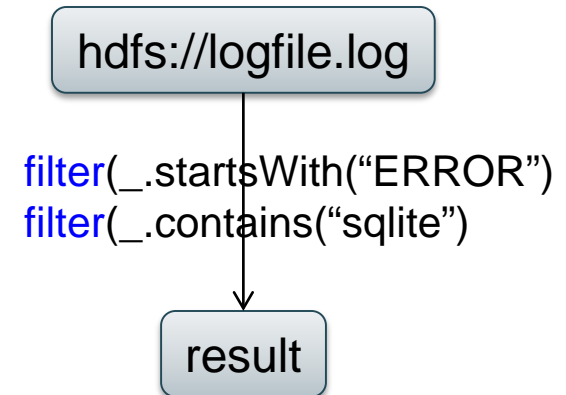
```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
errors.persist()  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

New RDD

Spark can recompute the result from errors

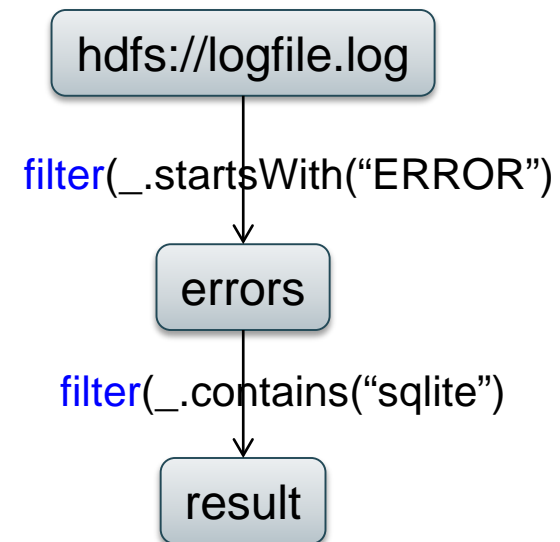
Persistence

RDD:



```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart



```
lines = spark.textFile("hdfs://logfile.log");  
errors = lines.filter(_.startsWith("ERROR"));  
errors.persist() New RDD  
sqlerrors = errors.filter(_.contains("sqlite"));  
sqlerrors.collect()
```

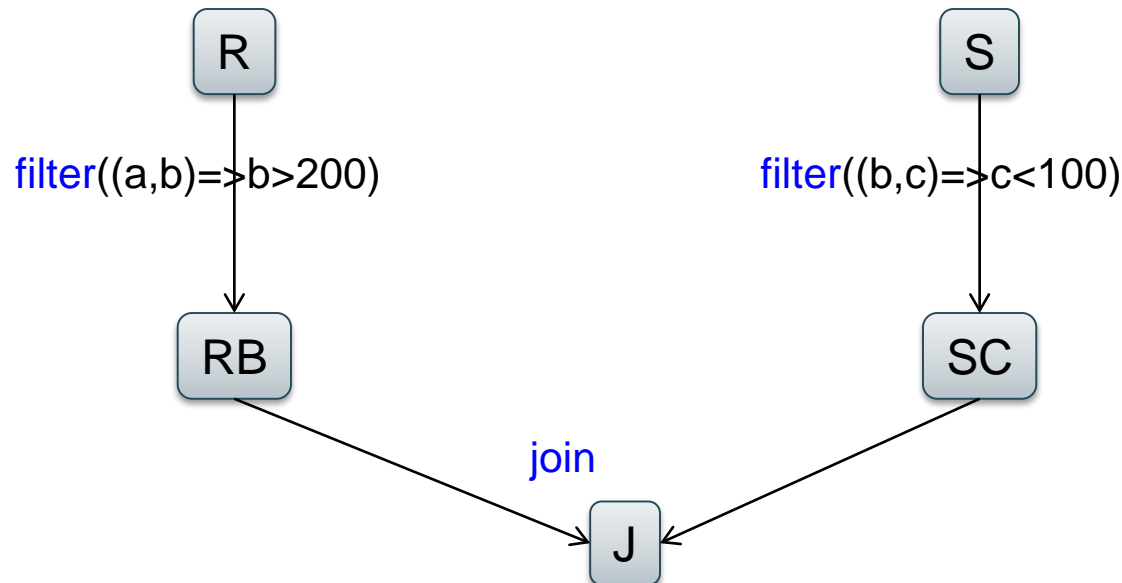
Spark can recompute the result from errors

R(A,B)
S(A,C)

```
SELECT count(*) FROM R, S  
WHERE R.B > 200 and S.C < 100 and R.A = S.A
```

Example

```
R = spark.textFile("R.csv").map(parseRecord).persist()  
S = spark.textFile("S.csv").map(parseRecord).persist()  
RB = R.filter((a,b) => b > 200).persist()  
SC = S.filter((a,c) => c < 100).persist()  
J = RB.join(SC).persist  
J.count();
```



Programming in Spark

- A Spark/Scala program consists of:
 - Transformations (map, reduce, join...). Lazy
 - Actions (count, reduce, save...). Eager
- `RDD[T]` = an RDD collection of type T
 - Partitioned, recoverable (through lineage), not nested
- `Seq[T]` = a Scala sequence
 - Local to a server, may be nested

Transformations:

<code>map(f : T => U):</code>	<code>RDD[T] => RDD[U]</code>
<code>flatMap(f: T => Seq(U)):</code>	<code>RDD[T] => RDD[U]</code>
<code>filter(f:T=>Bool):</code>	<code>RDD[T] => RDD[T]</code>
<code>groupByKey():</code>	<code>RDD[(K,V)] => RDD[(K,Seq[V])]</code>
<code>reduceByKey(F:(V,V) => V):</code>	<code>RDD[(K,V)] => RDD[(K,V)]</code>
<code>union():</code>	<code>(RDD[T],RDD[T]) => RDD[T]</code>
<code>join():</code>	<code>(RDD[(K,V)],RDD[(K,W)]) => RDD[(K,(V,W))]</code>
<code>cogroup():</code>	<code>(RDD[(K,V)],RDD[(K,W)]) => RDD[(K,(Seq[V],Seq[W]))]</code>
<code>crossProduct():</code>	<code>(RDD[T],RDD[U]) => RDD[(T,U)]</code>

Actions:

<code>count():</code>	<code>RDD[T] => Long</code>
<code>collect():</code>	<code>RDD[T] => Seq[T]</code>
<code>reduce(f:(T,T)=>T):</code>	<code>RDD[T] => T</code>
<code>save(path:String):</code>	Outputs RDD to a storage system e.g. HDFS

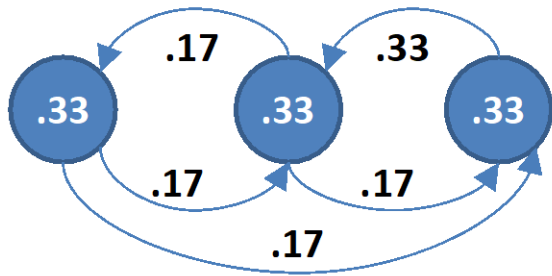
Conclusions

- Parallel databases
 - Predefined relational operators
 - Optimization
 - Transactions
- MapReduce
 - User-defined map and reduce functions
 - Must implement/optimize manually relational ops
 - No updates/transactions
- Spark
 - Predefined relational operators
 - Must optimize manually
 - No updates/transactions

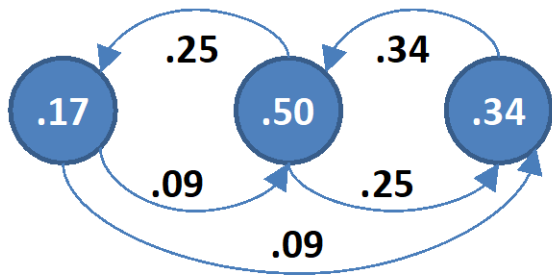
PageRank

- Page Rank is an algorithm that assigns to each page a score such that pages have higher scores if more pages with high scores link to them
- Page Rank was introduced by Google, and, essentially, defined Google

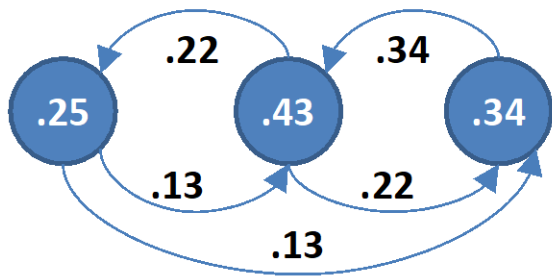
PageRank toy example



Superstep 0

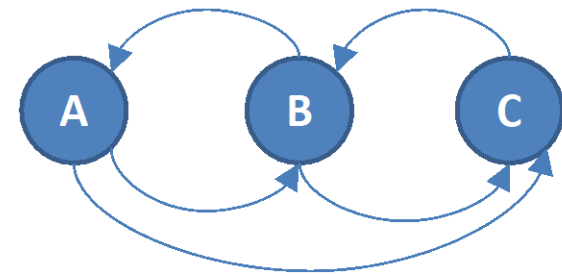


Superstep 1



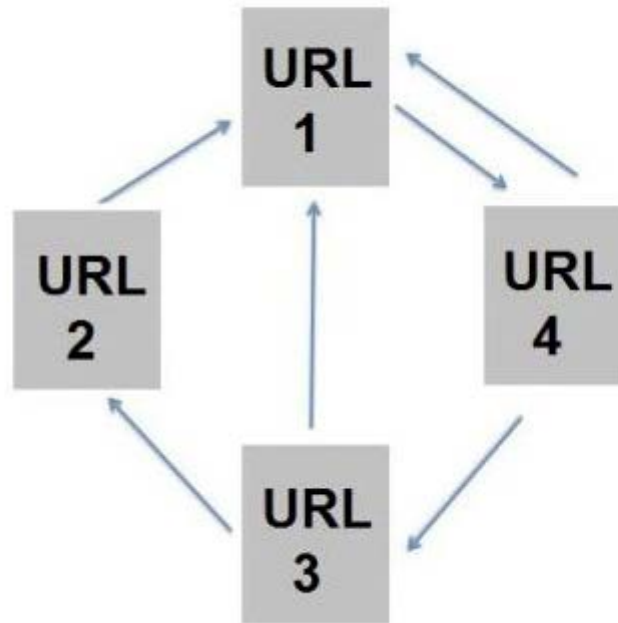
Superstep 2

Input graph



Page Rank: lets write a program

url_1 url_4
url_2 url_1
url_3 url_2
url_3 url_1
url_4 url_3
url_4 url_1



URL	Rank $r[i]$
url_4	0.25
url_3	0.25
url_2	0.25
url_1	0.25

url	Links[i]
url_4	{url_3,url_1}
url_3	{url_2,url_1}
url_2	{url_1}
url_1	{url_4}

for $i = 1$ to n :

$r[i] = 1/n$

repeat

for $j = 1$ to n : $contribs[j] = 0$

for $i = 1$ to n :

$k = \text{links}[i].\text{length}()$

for j in $\text{links}[i]$:

$contribs[j] += r[i] / k$

for $i = 1$ to n : $r[i] = \text{contribs}[i]$

until convergence

/ usually 10-20 iterations */*

URL	Rank $r[i]$
url_4	0.25
url_3	0.25
url_2	0.25
url_1	0.25

URL	contribs[j]
url_4	0
url_3	0
url_2	0
url_1	0

url	Links[i]
url_4	{url_3,url_1}
url_3	{url_2,url_1}
url_2	{url_1}
url_1	{url_4}

URL	Rank $r[i]$
url_4	0.25
url_3	0.25
url_2	0.25
url_1	0.25

URL	contribs[j]
url_4	0.25
url_3	0
url_2	0
url_1	0

url	Links[i]
url_4	{url_3,url_1}
url_3	{url_2,url_1}
url_2	{url_1}
url_1	{url_4}

URL	Rank $r[i]$
url_4	0.25
url_3	0.25
url_2	0.25
url_1	0.25

URL	contribs[j]
url_4	0.25
url_3	0
url_2	0
url_1	0.25

url	Links[i]
url_4	{url_3,url_1}
url_3	{url_2,url_1}
url_2	{url_1}
url_1	{url_4}

URL	Rank $r[i]$
url_4	0.25
url_3	0.25
url_2	0.25
url_1	0.25

URL	contribs[j]
url_4	0.25
url_3	0
url_2	0.125
url_1	0.375

url	Links[i]
url_4	{url_3,url_1}
url_3	{url_2,url_1}
url_2	{url_1}
url_1	{url_4}

URL	Rank $r[i]$
url_4	0.25
url_3	0.25
url_2	0.25
url_1	0.25

URL	contribs[j]
url_4	0.25
url_3	0.125
url_2	0.125
url_1	0.5

url	Links[i]
url_4	{url_3,url_1}
url_3	{url_2,url_1}
url_2	{url_1}
url_1	{url_4}

PageRank: this time in Spark

```
for i = 1 to n:
  r[i] = 1/n

repeat
  for j = 1 to n: contribs[j] = 0
  for i = 1 to n:
    k = links[i].length()
    for j in links[i]:
      contribs[j] += r[i] / k
  for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */
```

```
// SPARK
val links = spark.textFile(..).map(..).persist()
var ranks = // RDD of (URL, 1/n) pairs
for (k <- 1 to ITERATIONS) {
  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links,rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
    .mapValues(sum => a/n + (1-a)*sum)
}
```

