

Introduction to Data Management CSE 344

Lecture 26: Spark

CSE 344 - Winter 2016

1

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
- 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
- For HW8: you only need the SQL interface!

CSE 344 - Winter 2016

3

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 recompute the lost partition of the RDD

CSE 344 - Winter 2016

4

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

8

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))`

9

Persistence

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

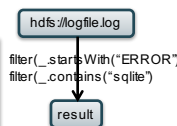
If any server fails before the end, then Spark must restart

Persistence

```
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

RDD:



Persistence

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

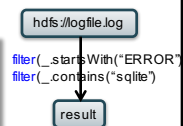
If any server fails before the end, then Spark must restart

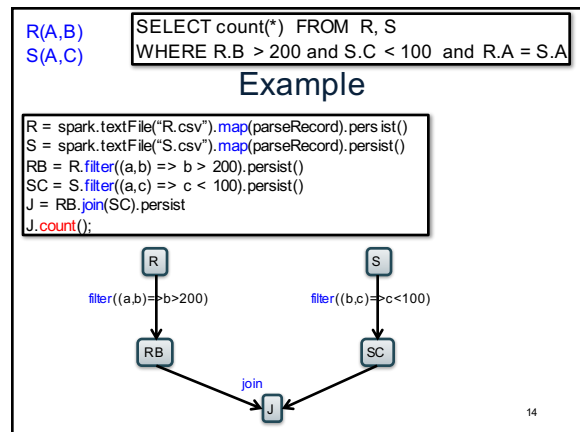
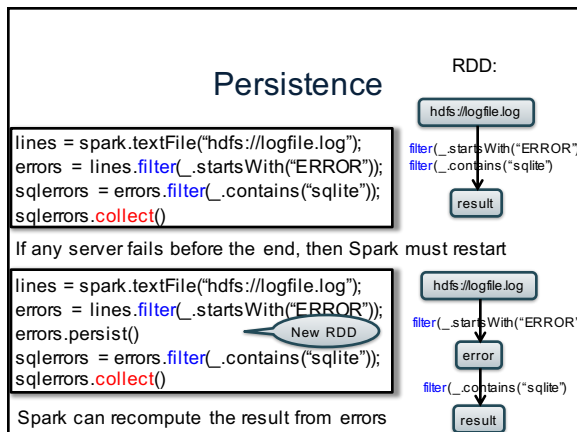
```
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_startsWith("ERROR"));
errors.persist()
sqlerrors = errors.filter(_contains("sqlite"));
sqlerrors.collect()
```

Spark can recompute the result from errors

New RDD

RDD:





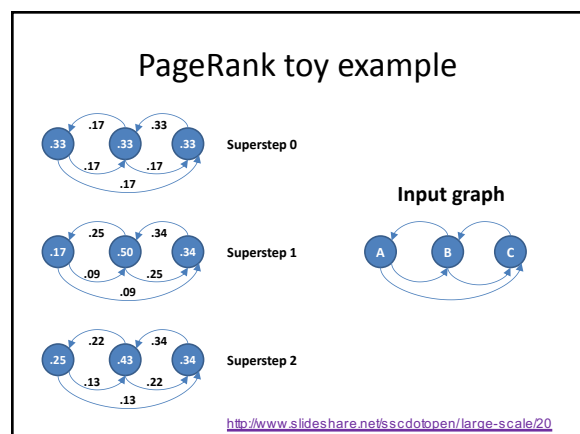
Programming in Spark

- 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

Transformations:	
<code>map(f : T => U):</code>	$RDD[T] \Rightarrow RDD[U]$
<code>flatMap(f: T => Seq(U)):</code>	$RDD[T] \Rightarrow RDD[U]$
<code>filter(f:T=>Bool):</code>	$RDD[T] \Rightarrow RDD[T]$
<code>groupByKey():</code>	$RDD[(K,V)] \Rightarrow RDD[(K,Seq[V])]$
<code>reduceByKey(f:(V => V)):</code>	$RDD[(K,V)] \Rightarrow RDD[(K,V)]$
<code>union():</code>	$(RDD[T], RDD[T]) \Rightarrow RDD[T]$
<code>join():</code>	$(RDD[(K,V)], RDD[(K,W)]) \Rightarrow RDD[(K,(V,W))]$
<code>cogroup():</code>	$(RDD[(K,V)], RDD[(K,W)]) \Rightarrow RDD[(K,(Seq[V], Seq[W]))]$
<code>crossProduct():</code>	$(RDD[T], RDD[U]) \Rightarrow RDD[(T,U)]$
Actions:	
<code>count():</code>	$RDD[T] \Rightarrow \text{Long}$
<code>collect():</code>	$RDD[T] \Rightarrow Seq[T]$
<code>reduce(f:(T,T) => T):</code>	$RDD[T] \Rightarrow T$
<code>save(path:String):</code>	Outputs RDD to a storage system e.g. HDFS

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

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

Random walk interpretation:

Start at a random node i
At each step, randomly choose
an outgoing link and follow it.

Repeat for a very long time

$r[i]$ = prob. that we are at node i

```
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] = contribs[i]
until convergence
/* usually 10-20 iterations */
```

19

PageRank

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

Random walk interpretation:

Start at a random node i
At each step, randomly choose
an outgoing link and follow it.

Improvement: with small prob. a
restart at a random node.

```
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] = contribs[i]
until convergence
/* usually 10-20 iterations */
```

$r[i] = a/n + (1-a)*contribs[i]$

where $a \in (0,1)$
is the restart
probability

20

PageRank

```
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)
}
```

21

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

22