Introduction to Data Management
CSE 344

Lecture 26: More Spark
Announcements

• HW8 due Wednesday
  – Make sure you monitor your AWS usage!

• Final next Monday
  – 2:30 – 4:20pm, JHN 102
  – 2 sheets of notes
  – Review session this Saturday afternoon
  – Previous exams are posted on course website
Announcements

• No sections this week

• Wednesday will be our last lecture 😞

• Today: Spark
Issues with MapReduce

• Difficult to write more complex queries

• Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk
Implementing Relational Operators in MapReduce

Given relations $R(A,B)$ and $S(B,C)$ compute:

- **Selection**: $\sigma_{A=123}(R)
- **Group-by**: $\gamma_{A,\text{sum}(B)}(R)$
- **Join**: $R \bowtie S$
Selection $\sigma_{A=123}(R)$

map(String key, Tuple t):
    if t.A == 123:
        EmitIntermediate(key, t);

reduce(String k, Iterator values):
    for each v in values:
        Emit(v);
Selection $\sigma_{A=123}(R)$

map(String key, Tuple t):
  if t.A = 123:
    EmitIntermediate(key, t);

reduce(String k, Iterator values):
  for each v in values:
    Emit(v);

No need for reduce.
But need system hacking
to remove reduce from MapReduce
Spark Interface

- Spark supports a Scala interface
- Scala = extension of Java with 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!
RDD

- RDD = Resilient Distributed Datasets
  - A distributed collection of data items, together with its lineage
  - Collection = list of ints, list of KV pairs, etc
  - Lineage = expression that says how that relation was computed = a relational algebra plan

- Spark stores intermediate results as RDD
- Spark operators take in (and generate) RDDs
RDD

• If a server crashes, its RDD in memory is lost

• But the master node knows the lineage, and will simply recompute the RDDs
  – Improve over MapReduce: we can recompute even within a map / reduce task

• How is this done?
  – Store intermediate RDDs to disk
  – Separate operators into lazy and eager
  – Construct a graph of operators
Programming in Spark

• A Spark/Scala program consists of:
  – Transformations (map, reduceByKey, 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
Scala Primer

• Functions with one argument:
  
  ```scala
  _.contains("sqlite")
  _ > 6
  ```

• Functions with more arguments
  
  ```scala
  (x => x.contains("sqlite"))
  (x => x > 6)
  ((x,y) => x+3*y)
  ```

• Closures (functions with free variables):
  
  ```scala
  var x = 5; rdd.filter(_ > x)
  var s = "sqlite"; rdd.filter(x => x.contains(s))
  ```
Example

Given a large log file hdfs://logfile.log
retrieve all lines that:
  • Start with ERROR
  • Contain the string “sqlite”

```scala
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()
```
Example

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

```
lines = spark.textFile("hdfs://logfile.log")
  .filter(_.startsWith("ERROR"))
  .filter(_.contains("sqlite"))
  .collect();
```
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
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
If any server fails before the end, then Spark must restart
Persistence

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

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

Spark can recompute the result from errors
Persistence

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

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

Spark can recompute the result from errors

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

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

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();
result = J.count();
```

result: 
- filter((a,b)=>b>200)
- filter((b,c)=>c<100)

```
R
  ↓
RB

S
  ↓
SC

J
```

```
join
```
# Transforms and Actions in Spark

## Transformations:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Input Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T =&gt; U)</code></td>
<td><code>RDD[T]</code></td>
<td><code>RDD[U]</code></td>
</tr>
<tr>
<td><code>flatMap(f : T =&gt; Seq(U))</code></td>
<td><code>RDD[T]</code></td>
<td><code>RDD[U]</code></td>
</tr>
<tr>
<td><code>filter(f:T=&gt;Bool)</code></td>
<td><code>RDD[T]</code></td>
<td><code>RDD[T]</code></td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD[(K,V)]</code></td>
<td><code>RDD[(K,Seq[V])]</code></td>
</tr>
<tr>
<td><code>reduceByKey(F:(V,V) =&gt; V)</code></td>
<td><code>RDD[(K,V)]</code></td>
<td><code>RDD[(K,V)]</code></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD[T],RDD[T])</code></td>
<td><code>RDD[T]</code></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD[(K,V)],RDD[(K,W)])</code></td>
<td><code>RDD[(K,(V,W))]</code></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD[(K,V)],RDD[(K,W)])</code></td>
<td><code>RDD[(K,(Seq[V],Seq[W]))]</code></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD[T],RDD[U])</code></td>
<td><code>RDD[(T,U)]</code></td>
</tr>
</tbody>
</table>

**Outputs 1 object per input**

**Output multiple objects per input**

## Actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Input Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>RDD[T]</code></td>
<td><code>Long</code></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD[T]</code></td>
<td><code>Seq[T]</code></td>
</tr>
<tr>
<td><code>reduce(f:(T,T)=&gt;T)</code></td>
<td><code>RDD[T]</code></td>
<td><code>T</code></td>
</tr>
<tr>
<td><code>save(path:String)</code></td>
<td></td>
<td>Outputs RDD to a storage system e.g. HDFS</td>
</tr>
</tbody>
</table>

**Outputs 1 object per input**
An Example: 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

http://www.slideshare.net/sscdotopen/large-scale/20