Introduction to Distributed Optimization

Reza Zadeh

Matroid

ICME

Institute for Computational & Mathematical Engineering
at Stanford University

Spark

@Reza_Zadeh | http://reza-zadeh.com
Key Idea

Resilient Distributed Datasets (RDDs)
» Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
» Built via parallel transformations (map, filter, ...)
» The world only lets you make make RDDs such that they can be:

Automatically rebuilt on failure
Life of a Spark Program

1) Create some input RDDs from external data or parallelize a collection in your driver program.

2) Lazily transform them to define new RDDs using transformations like filter() or map().

3) Ask Spark to cache() any intermediate RDDs that will need to be reused.

4) Launch actions such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark.
## Example Transformations

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>map()</td>
<td>intersection()</td>
<td>cartesian()</td>
</tr>
<tr>
<td>flatMap()</td>
<td>distinct()</td>
<td>pipe()</td>
</tr>
<tr>
<td>filter()</td>
<td>groupByKey()</td>
<td>coalesce()</td>
</tr>
<tr>
<td>mapPartitions()</td>
<td>reduceByKey()</td>
<td>repartition()</td>
</tr>
<tr>
<td>mapPartitionsWithIndex()</td>
<td>sortByKey()</td>
<td>partitionBy()</td>
</tr>
<tr>
<td>sample()</td>
<td>join()</td>
<td>...</td>
</tr>
<tr>
<td>union()</td>
<td>cogroup()</td>
<td>...</td>
</tr>
</tbody>
</table>
Example Actions

reduce()
collect()
count()
first()
take()
takeSample()
saveToCassandra()
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByKey()
foreach()
PairRDD

Operations for RDDs of tuples (Scala has nice tuple support)

groupByKey

Employees

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5500</td>
</tr>
<tr>
<td>20</td>
<td>15000</td>
</tr>
<tr>
<td>20</td>
<td>7000</td>
</tr>
<tr>
<td>30</td>
<td>12000</td>
</tr>
<tr>
<td>30</td>
<td>5100</td>
</tr>
<tr>
<td>30</td>
<td>4900</td>
</tr>
<tr>
<td>30</td>
<td>5800</td>
</tr>
<tr>
<td>30</td>
<td>5600</td>
</tr>
<tr>
<td>40</td>
<td>7500</td>
</tr>
<tr>
<td>40</td>
<td>8000</td>
</tr>
<tr>
<td>50</td>
<td>9000</td>
</tr>
<tr>
<td>50</td>
<td>8500</td>
</tr>
<tr>
<td>50</td>
<td>9500</td>
</tr>
<tr>
<td>50</td>
<td>8500</td>
</tr>
<tr>
<td>50</td>
<td>10500</td>
</tr>
<tr>
<td>50</td>
<td>10000</td>
</tr>
<tr>
<td>50</td>
<td>9500</td>
</tr>
</tbody>
</table>

Sum of Salary in Employees table for each department

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5500</td>
</tr>
<tr>
<td>20</td>
<td>22000</td>
</tr>
<tr>
<td>30</td>
<td>33400</td>
</tr>
<tr>
<td>40</td>
<td>15500</td>
</tr>
<tr>
<td>50</td>
<td>65550</td>
</tr>
</tbody>
</table>

Avoid using it – use reduceByKey
Guide for RDD operations

https://spark.apache.org/docs/latest/programming-guide.html

Browse through this.
Communication Costs

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- join with inputs co-partitioned
- groupByKey
- join with inputs not co-partitioned
**MLlib: Available algorithms**

**classification**: logistic regression, linear SVM, naïve Bayes, least squares, classification tree

**regression**: generalized linear models (GLMs), regression tree

**collaborative filtering**: alternating least squares (ALS), non-negative matrix factorization (NMF)

**clustering**: k-means||

**decomposition**: SVD, PCA

**optimization**: stochastic gradient descent, L-BFGS
Optimization

At least two large classes of optimization problems humans can solve:

» Convex

» Spectral
Optimization Example: Gradient Descent
ML Objectives

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i) \]
Scaling

1) Data size

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i) \]

2) Model size

3) Number of models
Logistic Regression

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i) \]

val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1)) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
Separable Updates

Can be generalized for

» Unconstrained optimization

» Smooth or non-smooth

» LBFGS, Conjugate Gradient, Accelerated Gradient methods, …
Logistic Regression Results

Running Time (s)

Number of Iterations

Hadoop: 110 s / iteration
First iteration: 80 s
Further iterations: 1 s

Spark: 100 GB of data on 50 m1.xlarge EC2 machines
Behavior with Less RAM

- Iteration time (s):
  - 0%: 68.8
  - 25%: 58.1
  - 50%: 40.7
  - 75%: 29.7
  - 100%: 11.5

% of working set in memory: 0% to 100%