Communication Patterns

Reza Zadeh

ICME
INSTITUTE for COMPUTATIONAL & MATHEMATICAL ENGINEERING
at STANFORD UNIVERSITY

Matroid

Spark

@Reza_Zadeh | http://reza-zadeh.com
Outline

Shipping code to the cluster
Shuffling
Broadcasting
Other programming languages
Outline

Shipping code to the cluster
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

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce()</td>
<td></td>
</tr>
<tr>
<td>collect()</td>
<td></td>
</tr>
<tr>
<td>count()</td>
<td></td>
</tr>
<tr>
<td>first()</td>
<td></td>
</tr>
<tr>
<td>take()</td>
<td></td>
</tr>
<tr>
<td>takeSample()</td>
<td></td>
</tr>
<tr>
<td>saveToCassandra()</td>
<td></td>
</tr>
<tr>
<td>takeOrdered()</td>
<td></td>
</tr>
<tr>
<td>saveAsTextFile()</td>
<td></td>
</tr>
<tr>
<td>saveAsSequenceFile()</td>
<td></td>
</tr>
<tr>
<td>saveAsObjectFile()</td>
<td></td>
</tr>
<tr>
<td>countByKey()</td>
<td></td>
</tr>
<tr>
<td>count()</td>
<td></td>
</tr>
<tr>
<td>foreach()</td>
<td></td>
</tr>
<tr>
<td>saveToCassandra()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Sending your code to the cluster
RDD \rightarrow \text{Stages} \rightarrow \text{Tasks}

- **RDD Objects**: `rdd1.join(rdd2).groupBy(...) .filter(...)`
  - Build operator DAG

- **DAG Scheduler**: Split graph into stages of tasks
  - Submit each stage as ready

- **Task Scheduler**: Launch tasks via cluster manager
  - Retry failed or straggling tasks

- **Worker**: Execute tasks
  - Store and serve blocks

The diagram illustrates the workflow of processing with RDDs, starting with operations on RDDs, followed by the construction of a Directed Acyclic Graph (DAG) of tasks, submission of these tasks, and finally the execution of these tasks on a cluster.
Communication Patterns

Narrow Dependencies:
- map, filter
- union
- join with inputs co-partitioned

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned
Example Stages

- **Stage 1**: RDD
- **Stage 2**: map, filter
- **Stage 3**: join

Legend:
- = RDD
- cached partition
- lost partition
Talking to Cluster Manager

Manager can be:

- YARN
- Mesos
- Spark Standalone
Shuffling
Shuffle

\[ \text{groupByKey} = \text{sortByKey} \]

\[ \text{reduceByKey} \]

Sort: use advances in sorting single-machine memory-disk operations for all-to-all communication
Sorting

Distribute Timsort, which is already well-adapted to respecting disk vs memory

Sample points to find good boundaries

Each machines sorts locally and builds an index
# Sorting (shuffle)

<table>
<thead>
<tr>
<th></th>
<th>Hadoop World Record</th>
<th>Spark 100 TB</th>
<th>Spark 1 PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Size</td>
<td>102.5 TB</td>
<td>100 TB</td>
<td>1000 TB</td>
</tr>
<tr>
<td>Elapsed Time</td>
<td>72 mins</td>
<td>23 mins</td>
<td>234 mins</td>
</tr>
<tr>
<td># Nodes</td>
<td>2100</td>
<td>206</td>
<td>190</td>
</tr>
<tr>
<td># Cores</td>
<td>50400</td>
<td>6592</td>
<td>6080</td>
</tr>
<tr>
<td># Reducers</td>
<td>10,000</td>
<td>29,000</td>
<td>250,000</td>
</tr>
<tr>
<td>Rate</td>
<td>1.42 TB/min</td>
<td>4.27 TB/min</td>
<td>4.27 TB/min</td>
</tr>
<tr>
<td>Rate/node</td>
<td>0.67 GB/min</td>
<td>20.7 GB/min</td>
<td>22.5 GB/min</td>
</tr>
<tr>
<td>Sort Benchmark</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Daytona Rules</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Environment</td>
<td>dedicated data center</td>
<td>EC2 (i2.8xlarge)</td>
<td>EC2 (i2.8xlarge)</td>
</tr>
</tbody>
</table>

Distributed TimSort
Example Join

```scala
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.txt").map(...)

val joined = visits.join(pageNames)
```

Shuffles both pageNames and visits over network
Broadcasting
Broadcasting

Often needed to propagate current guess for optimization variables to all machines

The exact wrong way to do it is with “one machines feeds all” – use bit-torrent instead

Needs log(n) rounds of communication
Bit-torrent Broadcast

[Diagram of a networked system with multiple downloaders and one uploader]
Broadcast Rules

Create with SparkContext.broadcast(initialVal)

Access with .value inside tasks (first task on each node to use it fetches the value)

Cannot be modified after creation
Replicated Join

```scala
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap)

val visits = sc.textFile("visits.txt").map(...)

val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))
```

- Type is Broadcast[Map[...]]
- Call .value to access value
- Only sends pageMap to each node once
Model Broadcast

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

```scala
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
}
```
Model Broadcast

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

Call `sc.broadcast`

```scala
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
}
```

Use via `.value`
Spark for Python (PySpark)
PySpark and Pipes

Spark core is written in Scala

PySpark calls existing scheduler, cache and networking layer (2K-line wrapper)

No changes to Python