

Updated: 2021-03-31

6.8

Spark Performance Tuning

© 2015-21 Robin Hillyard



Northeastern
University

Spark Performance Tips

- Know what your Scala code is doing!

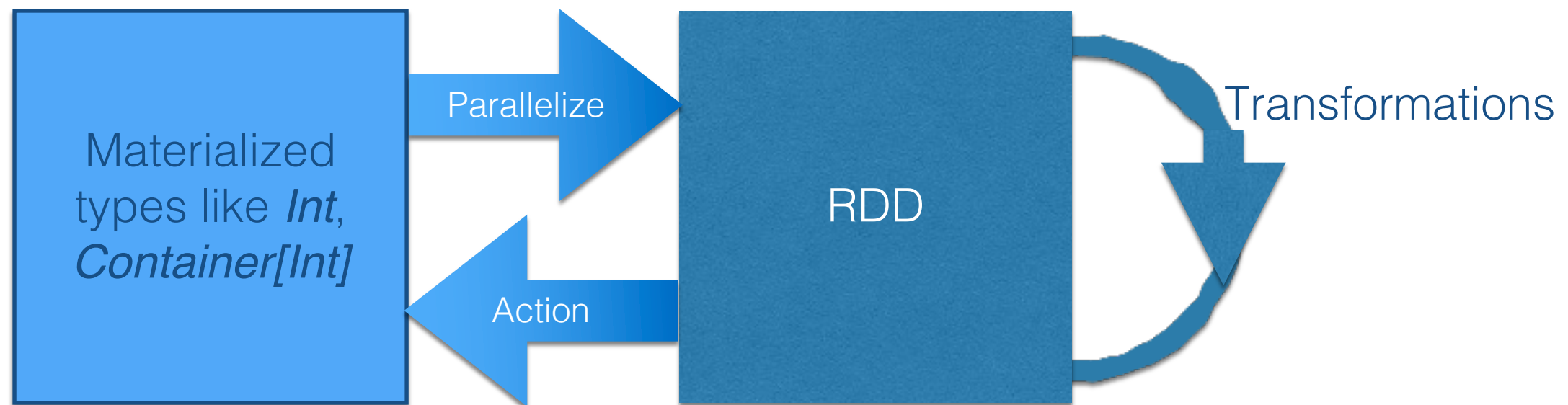
```
def main(args: Array[String]) {  
  if (args.size < 1) {  
    println("Please provide train.csv path as input argument");  
    return;  
  }  
  val data = LoadClaimData.loadData(args(0));  
  val pca = new PCA(60).fit(data.map(_.features))  
  val projected = data.map(p => p.copy(features = pca.transform(p.features)))  
  // Split data into training (60%) and test (40%).  
  val splits = projected.randomSplit(Array(0.6, 0.4), seed = 11L)  
  val training = splits(0).cache()  
  val test = splits(1)  
  // Run training algorithm to build the model  
  val model = new LogisticRegressionWithLBFGS()  
    .setNumClasses(2)  
    .run(training)  
  // Scala class evaluation Get evaluation metrics.  
  val predictionAndLabels2 = test.map {  
    case LabeledPoint(label, features) =>  
      val prediction = model.setThreshold(0.6).predict(features)  
      (prediction, label)  
  }  
  // Precision and Recall using BinaryClassificationMetrics  
  val metrics = new BinaryClassificationMetrics(predictionAndLabels2)  
  println(s"Recall = ${metrics.pr().collect().tail.head._1}")  
  println(s"Precision = ${metrics.pr().collect().tail.head._2}")  
  // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.  
  val predictionAndLabels = test.map {  
    case LabeledPoint(label, features) =>  
      val prediction = model.clearThreshold().predict(features)  
      (prediction, label)  
  }  
  // Calculating logloss error using the real values in the range 0 to 1.  
  val loss = for {  
    pair <- predictionAndLabels  
    error = calculateErrorFactor(pair._1, pair._2)  
  } yield error  
  val logloss = loss.collect().toSeq.reduce(_ + _)  
  println(s"logloss= ${logloss / loss.count * (-1)}")  
}
```

What uses up all the time in Spark?

- Actions!
 - Remember: actions speak louder than transformations—and they use up far more resources
 - Extractions:
 - *collect* (think of this if you like as the opposite of *sc.parallelize*)
 - *foreach*, *saveAsTextFile*, *saveAsObjectFile*.
 - Aggregations—measure an *RDD*:
 - *count*, *aggregate*, *max*, *min*, *reduce*, *treeReduce*, *fold*.
- Transformations (don't take any time: they are lazy!)
 - single *RDD*s:
 - *map*, *flatMap*, *filter*, *distinct*, *sample*, *take*, *drop*, *collect(f)*,
 - single *RDD*s of an appropriate type:
 - *top*, *takeOrdered*, *takeSample*, *countByValue*, *groupBy*, *sortBy*
 - Combinations—two *RDD*s (of same underlying type):
 - *union*, *intersection*, *subtract*, *cartesian*, *zip*

Working with RDDs (2)

- Because RDD[T] is lazy, it's essentially *opaque*



- Once we have parallelized a container as an RDD, we can apply transformations to RDDs, creating new RDDs. Since these of course are lazy, the RDDs are just *decorated*. In order to materialize something from an RDD, we need to apply an “action”.

logging, timing, etc.

- Added a logger, a timer, and type annotations so we can see what we're dealing with:

```
def doClaimsPrediction(filename: String): Unit = {
  val log = Logger.getLogger(getClass.getName)
  val timer1 = Timer.apply
  val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
  val pca: PCAModel = new PCA(60).fit(data.map(_.features))
  val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
  // Split data into training (60%) and test (40%).
  val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
  val training: RDD[LabeledPoint] = splits(0).cache()
  val test: RDD[LabeledPoint] = splits(1)
  // Run training algorithm to build the model
  val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
    .setNumClasses(2)
    .run(training)
  // Scala class evaluation Get evaluation metrics.
  val predictionAndLabels2: RDD[(Double, Double)] = test.map {
    case LabeledPoint(label, features) =>
      val prediction = model.setThreshold(0.6).predict(features)
      (prediction, label)
  }
  val timer2 = timer1.lap("after modeling", log.info(_))
  // Precision and Recall using BinaryClassificationMetrics
  val metrics: BinaryClassificationMetrics = new BinaryClassificationMetrics(predictionAndLabels2)
  log.info(s"Recall = ${metrics.pr().collect().tail.head._1}")
  log.info(s"Precision = ${metrics.pr().collect().tail.head._2}")
  val timer3 = timer2.lap("after metrics", log.info(_))
  // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
  val predictionAndLabels: RDD[(Double, Double)] = test.map {
    case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
  }
  // Calculating logloss error using the real values in the range 0 to 1.
  val loss: RDD[Double] = for (pr <- predictionAndLabels) yield calculateErrorFactor(pr._1, pr._2)
  val logloss = loss.collect().toSeq.reduce(_ + _)
  log.info(s"logloss= ${logloss / loss.count * (-1)}")
  timer3.lap("end", log.info(_))
}
```

Timings:

- lap1: 19.441 (modeling: logistic regression)
- lap2: 2.817 (precision/recall)
- lap3: 5.176 (logloss)

Improving performance

- Do you see any bad code?

```
def doClaimsPrediction(filename: String): Unit = {
  val log = Logger.getLogger(getClass.getName)
  val timer1 = Timer.apply
  val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
  val pca: PCAModel = new PCA(60).fit(data.map(_.features))
  val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
  // Split data into training (60%) and test (40%).
  val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
  val training: RDD[LabeledPoint] = splits(0).cache()
  val test: RDD[LabeledPoint] = splits(1)
  // Run training algorithm to build the model
  val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
    .setNumClasses(2)
    .run(training)
  // Scala class evaluation Get evaluation metrics.
  val predictionAndLabels2: RDD[(Double, Double)] = test.map {
    case LabeledPoint(label, features) =>
      val prediction = model.setThreshold(0.6).predict(features)
      (prediction, label)
  }
  val timer2 = timer1.lap("after modeling", log.info(_))
  // Precision and Recall using BinaryClassificationMetrics
  val metrics: BinaryClassificationMetrics = new
BinaryClassificationMetrics(predictionAndLabels2)
  log.info(s"Recall = ${metrics.pr().collect().tail.head._1}")
  log.info(s"Precision = ${metrics.pr().collect().tail.head._2}")
  val timer3 = timer2.lap("after metrics", log.info(_))
  // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
  val predictionAndLabels: RDD[(Double, Double)] = test.map {
    case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
  }
  // Calculating logloss error using the real values in the range 0 to 1.
  val loss: RDD[Double] = for (pr <- predictionAndLabels) yield
calculateErrorFactor(pr._1, pr._2)
  val logloss = loss.collect().toSeq.reduce(_ + _)
  log.info(s"logloss= ${logloss / loss.count * (-1)}")
  timer3.lap("end", log.info(_))
}
```

Improving performance

- Do you see any bad code?

```
def doClaimsPrediction(filename: String): Unit = {  
  val log = Logger.getLogger(getClass.getName)  
  val timer1 = Timer.apply  
  val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)  
  val pca: PCAModel = new PCA(60).fit(data.map(_.features))  
  val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))  
  // Split data into training (60%) and test (40%).  
  val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)  
  val training: RDD[LabeledPoint] = splits(0).cache()  
  val test: RDD[LabeledPoint] = splits(1)  
  // Run training algorithm to build the model  
  val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()  
    .setNumClasses(2)  
    .run(training)  
  // Scala class evaluation Get evaluation metrics.  
  val predictionAndLabels2: RDD[(Double, Double)] = test.map {  
    case LabeledPoint(label, features) =>  
      val prediction = model.setThreshold(0.6).predict(features)  
      (prediction, label)  
  }  
  val timer2 = timer1.lap("after modeling", log.info(_))  
  // Precision and Recall using BinaryClassificationMetrics  
  val metrics: BinaryClassificationMetrics = new  
BinaryClassificationMetrics(predictionAndLabels2)  
  log.info(s"Recall = ${metrics.pr().collect().tail.head._1}")  
  log.info(s"Precision = ${metrics.pr().collect().tail.head._2}")  
  val timer3 = timer2.lap("after metrics", log.info(_))  
  // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.  
  val predictionAndLabels: RDD[(Double, Double)] = test.map {  
    case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)  
  }  
  // Calculating logloss error using the real values in the range 0 to 1.  
  val loss: RDD[Double] = for (pr <- predictionAndLabels) yield  
calculateErrorFactor(pr._1, pr._2)  
  val logloss = loss.collect().toSeq.reduce(_ + _)  
  log.info(s"logloss= ${logloss / loss.count * (-1)}")  
  timer3.lap("end", log.info(_))  
}
```

There are two *collect* operations on the same variable

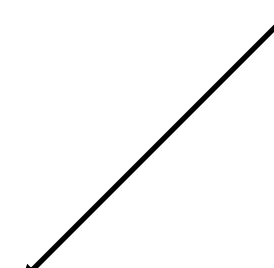
We are performing the reduce after collect so it can't be done in parallel

Improving performance

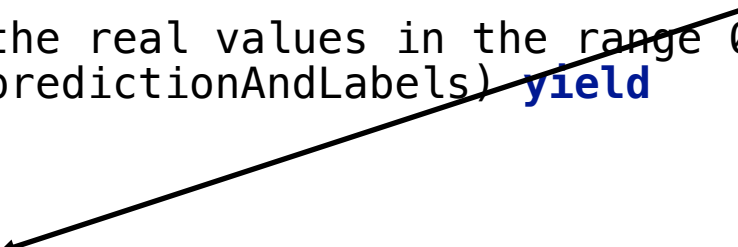
- After a little cleanup...

```
def doClaimsPrediction(filename: String): Unit = {
  val log = Logger.getLogger(getClass.getName)
  val timer1 = Timer.apply
  val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
  val pca: PCAModel = new PCA(60).fit(data.map(_.features))
  val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
  // Split data into training (60%) and test (40%).
  val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
  val training: RDD[LabeledPoint] = splits(0).cache()
  val test: RDD[LabeledPoint] = splits(1)
  // Run training algorithm to build the model
  val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
    .setNumClasses(2)
    .run(training)
  // Scala class evaluation Get evaluation metrics.
  val predictionAndLabels2: RDD[(Double, Double)] = test.map {
    case LabeledPoint(label, features) =>
      val prediction = model.setThreshold(0.6).predict(features)
      (prediction, label)
  }
  val timer2 = timer1.lap("after modeling", log.info(_))
  // Precision and Recall using BinaryClassificationMetrics
  val metrics: BinaryClassificationMetrics = new
BinaryClassificationMetrics(predictionAndLabels2)
  val metricsTuple: (Double, Double) = metrics.pr().collect().tail.head
  log.info(s"Recall = ${metricsTuple._1}")
  log.info(s"Precision = ${metricsTuple._2}")
  val timer3 = timer2.lap("after metrics", log.info(_))
  // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
  val predictionAndLabels: RDD[(Double, Double)] = test.map {
    case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
  }
  // Calculating logloss error using the real values in the range 0 to 1. 3300 msecs
  val loss: RDD[Double] = for (pr <- predictionAndLabels) yield
calculateErrorFactor(pr._1, pr._2)
  val logloss: Double = loss.mean()
  log.info(s"logloss= ${logloss / loss.count * (-1)}")
  timer3.lap("end", log.info(_))
}
```

Doing only one *collect*
saved 667 msecs



using *mean*, i.e. instead of
collect then *reduce* saved



3300 msecs

Saving/caching/persisting

- Spark provides several ways that you can save previous work:
 - `cache`:
 - `persist`:
 - `MLLib: save (model)`

Testing

- It doesn't matter how fast your application runs if it gets incorrect results—
 - Actually, the faster an incorrect application runs, the more work you might have to do to mitigate the damage!
 - The students who wrote the example that I've used here actually claimed a much “better” *logloss* value—until I noticed that there was no unit test to check *logloss*. It turned out that they had the wrong formula!
- All software must be unit-tested and functional-tested—
 - Like a verbal contract that “isn't worth the paper it's printed on”, a chunk of software that doesn't have good unit test coverage isn't worth the space it takes up on the disk!

Unit testing *without* Spark

- Where possible, create and run **unit tests** that don't invoke Spark (they're just going to run a lot faster):

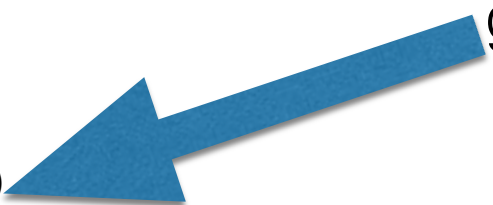
```
package learn
import org.scalatest.{ FlatSpec, Matchers, BeforeAndAfterAll }
class ClaimPredictionSpec extends FlatSpec with BeforeAndAfterAll with Matchers {
  import org.scalactic._
  implicit val efEq = new Equality[Double] {
    def areEqual(a: Double, b: Any): Boolean = math.abs(a-b.asInstanceOf[Double])<5E-3
  }
  val preal = 0.6
  "calculateErrorFactor(preal, 1)" should "match case Math.log(preal)" in {
    ClaimsPrediction.calculateErrorFactor(preal, 1) should === (Math.log(preal))
  }
  it should "match case 0.0" in {
    ClaimsPrediction.calculateErrorFactor(0.0, 1) should === (0.0)
  }
  it should "match case 1.0" in {
    ClaimsPrediction.calculateErrorFactor(1.0, 1) should === (0.0)
  }
  "calculateErrorFactor(preal, 0)" should "match case Math.log(1 - preal)" in {
    ClaimsPrediction.calculateErrorFactor(preal, 0) should === (Math.log(1 - preal))
  }
  it should "match case 0.0" in {
    ClaimsPrediction.calculateErrorFactor(0.0, 0) should === (0.0)
  }
  it should "match case 1.0" in {
    ClaimsPrediction.calculateErrorFactor(1.0, 0) should === (0.0)
  }
}
```

Unit testing *with* Spark

- But don't be afraid to use Spark in your unit tests when necessary [Spark is happy to run on a single machine]

```
package learn
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SQLContext
import org.scalatest.{BeforeAndAfterAll, FlatSpec, Matchers}
class LoglossSpec extends FlatSpec with BeforeAndAfterAll with Matchers {
  var sc: SparkContext = _
  var sqlContext: SQLContext = _
  override protected def beforeAll(): Unit = {
    super.beforeAll()
    val conf = new SparkConf().setMaster("local[2]")
    .setAppName("Insurance Claim")
    .set("spark.driver.allowMultipleContexts", "true")
    sc = new SparkContext(conf)
    sqlContext = new SQLContext(sc)
  }
  override protected def afterAll(): Unit = {
    try {
      sc.stop()
    } finally {
      super.afterAll()
    }
  }
  "logloss((0.5, 0.5))" should "equal log one half" in {
    val rdd = sc.parallelize(Array((0.5, 0.5)))
    val loss = for (pr <- rdd) yield ClaimsPrediction.calculateErrorFactor(pr._1, pr._2)
    loss.sum should === (-math.log(2.0) +- 0.005)
  }
}
```

You must set this if you're going to run multiple unit tests



Performance tuning

- There are some good (and obvious) rules that apply to performance tuning—ignore them at your peril:
 - Don't infer results for a full-scale application from a sample—
 - there will typically be some helpful indications but the behavior of a full-scale app might surprise you.
 - Don't change more than one parameter at a time—
 - you will not be able to deduce which change actually resulted in the increase (or decrease) of performance.
 - Don't guess! Instrument your app and run it—
 - performance tuning can be very *counter-intuitive*;
 - *document* your trials in a notebook (maybe even on paper).
 - Don't waste time on untested code—
 - Ensure that your code is fully functional (unit tests are green!) before performance tuning—and if you have to change the code, go back and run your earlier performance tests to ensure no degradation.

Tweaking:

Command Line parameters

- Common params (don't touch these without really understanding what you're doing—Spark will generally use sensible values for these):
 - —*num.executors*
 - —*executor.cores*
 - For example, using HDFS, e.g., it may make sense to have more executors and fewer cores per executor because of the HDFS blocks
 - —*driver-memory*
- Partitions (of an RDD)
 - Ideal number is the total number of cores that you have available.
 - If that number varies, it's best to set the number of partitions to the maximum number of cores that will be in play (the overhead of having too many partitions is small compared to the penalty of having insufficient partitions)

Tweaking:

Configuration parameters

- Some of the common configuration parameters...
 - *spark.executor.memory*: memory available to each executor (shared by all cores)
 - *spark.executor.extraJavaOptions*: for tuning each executor's JVM
 - *spark.executor.parallelis*: default number of data partitions
 - *spark.storage.memoryFraction* (0.6 by default) is for persisted RDDs
 - *spark.shuffle.memoryFraction* (0.2 by default) is reserved for the shuffle process

System factors which impact performance

- What sort of files do you work with?
 - Compressed files? Are they splittable into convenient chunks?
 - Text files?
 - Free-form?
 - Structured (CSV, JSON, XML, etc.)
 - In this case, you should read them into a DataFrame (e.g. use the Databricks CSV reader)
 - avoid using XML because such files typically straddle many lines and are therefore not splittable.
 - Sequence files and Avro files
 - Parquet files: excellent for using SparkSQL.

Understanding how it works

- Perhaps the most important thing for writing performant Spark applications:
 - is to understand exactly (or almost) what Spark is doing.
 - that's why it's so important to understand Scala, since Spark is written in Scala.
 - And the most important things about Scala, for understanding Spark, are the functional programming style and the concept of lazy evaluation.
 - Any time you are confused by something, or just want a better understanding, dive into the Spark source code!
 - example: *map*:

```
def map[U: ClassTag](f: T => U): RDD[U] = withScope {  
  val cleanF = sc.clean(f)  
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))  
}
```

a function

a reference to this

Serialization

- Serialization performance is important for any network-distributed application.
- By default, Spark uses Java serialization—
 - but you can configure it to use Kryo (typically much faster):
`conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
 - it's a little more complicated if you have your own custom classes, but if you are just using standard Spark classes, you should be all set.
- Functions:
 - Remember that functions, e.g. those you apply in transformations, must be serializable.
 - Most of the time, that means that they should be defined anonymously, or within “objects”. If you pass a class method (i.e. an instance method) as a function, then the class itself must be serializable too.

Compression

- Just like serialization, compression can also be important for the same reasons.
- By default, Spark uses snappy compression—but you can configure other codecs if you really know what you're doing...

Spark UI

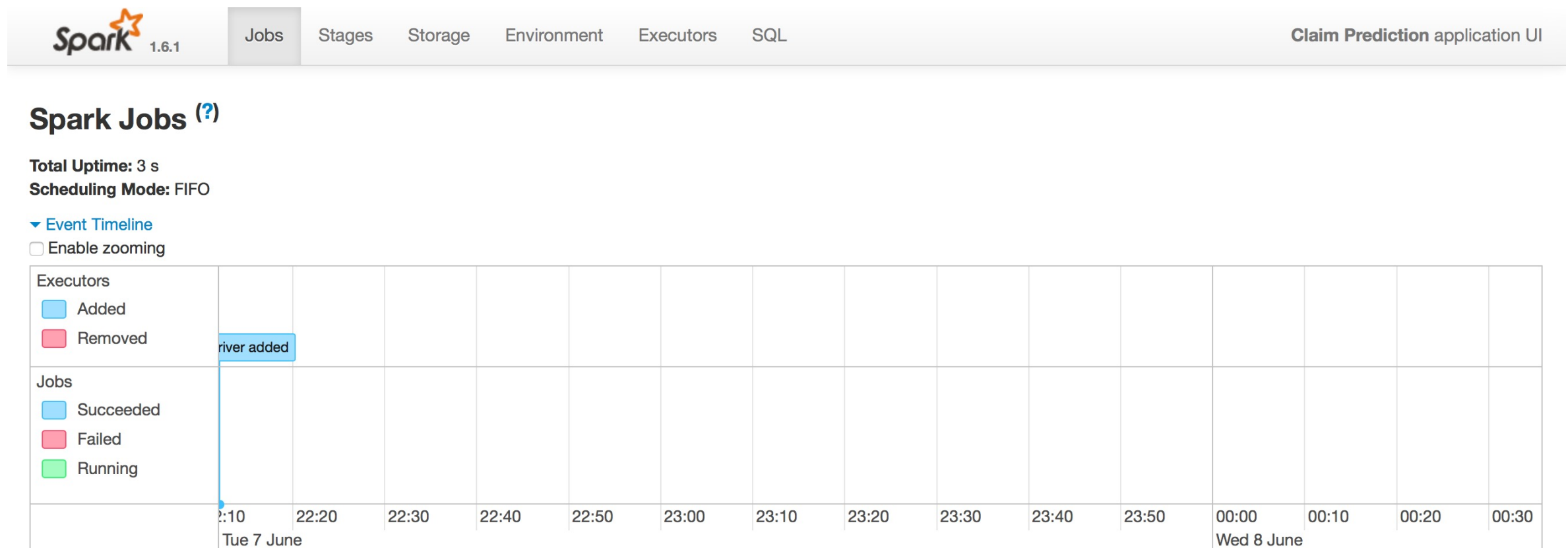
- Don't forget to use the Spark UI (port 4040) to show you what your DAGs look like.
- You can also use accumulators (which show up in the UI) so that you can see if things are actually happening—and how fast.

Spark resources

- Overview: <http://spark.apache.org/docs/latest/index.html>
- Running on AWS EC2: <http://spark.apache.org/docs/latest/ec2-scripts.html>
- Configuration: <http://spark.apache.org/docs/latest/configuration.html>
- Monitoring: <http://spark.apache.org/docs/latest/monitoring.html>
- Tuning: <http://spark.apache.org/docs/latest/tuning.html>
- Spark Streaming: <http://spark.apache.org/docs/latest/streaming-programming-guide.html>
- Data frames/sets, SparkSQL, etc.:
<http://spark.apache.org/docs/latest/sql-programming-guide.html>
- MLlib: <http://spark.apache.org/docs/latest/mllib-guide.html>
- GraphX: <http://spark.apache.org/docs/latest/graphx-programming-guide.html>

Monitoring

- Spark comes with a built-in monitoring UI:
 - <http://driver:4040>



AWS/Spark alternatives

- spark-ec2: kind of a hack that has just stuck around
- Databricks: comprehensive eco-system:
<https://www.gitbook.com/book/databricks/databricks-spark-reference-applications/details>
- Amazon EMR:
<https://aws.amazon.com/elasticmapreduce/details/spark/>
- Flintrock: <https://spark-summit.org/east-2016/events/flintrock-a-faster-better-spark-ec2/>
- Apache BigTop, Ubuntu Juju, Docker, etc.

Miscellaneous topics

- Scala/Java interaction
 - Scala and Java both inhabit the JVM (like Groovy, Clojure, etc.) so they are natural fellows
 - There are some differences in the languages that require you being a little bit careful:
 - The Scala compiler will warn you about these problems (generally speaking, the Scala compiler is very smart about all of these issues);
 - Scala doesn't have primitives (stack objects like *int*, *double*): all objects in Scala live on the heap: *Int*, *Double*, etc. Scala will do the proper thing for you.
 - Collections are similar but not the same!! However, there are implicit conversions which you can use. But bear in mind that these take time.
 - Generics are different (Scala has higher “kinds”): but both languages have type erasure so at run-time, the types aren't known either way.