Apache Spark Workshop

J.T. Halbert, Markus Dale, Jason Morris

February 24, 2015

Workshop goals

- Explore the Enron Dataset (`https://www.cs.cmu.edu/~./enron/`) while learning to use Apache Spark. Hopefully we can answer three questions:
  - Which ENRON employees spent too much time organizing their emails?
  - What hour of the week were most emails sent?
  - When did the FBI show up?
- Learn to appreciate the Scala Collections Library and the Resilient Distributed Dataset

Workshop non-goals

- Deploy and monitor Spark Applications
- Graphs
- MLlib

Overview of resources

- This presentation and “answers” to the “exercises” are available at `https://github.com/jt-halbert/spark-workshop/`.
- The scripts we used to create the AWS EMR instances is available at `https://github.com/notjasonmorris/AWS`.
- The ETL code that is making our exploration of the Enron dataset so convenient is available at `https://github.com/medale/spark-mail/`.
- LARGE PORTIONS of this presentation are pulled from Markus’ work. Thanks Markus!!

Who are we?

- I am Tetra's Chief Data Scientist and I help certain people learn certain things about certain parts of their data.
- Tetra Concepts, LLC is the finest collection of development talent anyone could ask for (AND SOME OF THEM ARE WALKING AMONG YOU!!!!11!!1!!!!)

Why Apache Spark?

- Excellent question: four parts of a partial answer.

Data Science is a filthy job

- I am not even sure what Data Science is. They put Science right in the name, so it must be pretty serious right?
- I like to think it is the disciplined application of a scientific mindset to that nebulous thing called "data."
- But really...
  - You spend 90% of your time getting and cleaning that thing.
  - And when you finally get it cleaned and available it very often is unwieldy in some further way (size or speed.)
  - The act of cleaning itself is a data science problem.

The Ecosystem is big and filled with snakes

- Berg Data

You can lie with statistics

- In a big enough database you can find a set of columns to perfectly predict any outcome.
  - `http://www.tylervigen.com/`
- We need better tools so we can spend more time doing the hard work of telling the truth (or some approximation).

Why Apache Spark?

- Spark gives you a way to explore small, medium, large, (very large ?) data in a convenient way.
  - You can actually explore distributed datasets: lazy evaluation and a rich collections api.
  - You can scale your exploratory code up to a full job relatively quickly: REPL driven development.
- It wraps an increasing amount of the Hadoop Ecosystem and plays naturally.

Your customer wants pretty little magical things

Figure 1: Spark wraps a lot of other peoples toys

Let's get started

- The first step is to learn enough Scala to be dangerous.

Combinator functions on Scala collections

- Examples: map, flatMap, filter, reduce, fold, aggregate
- Background - Combinatory logic, higher-order functions...

Combinatory Logic

Moses Schönfinkel and Haskell Curry in the 1920s

> *[C]ombinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments [Combinatory Logic @wikipedia_combinatory_2014]*

A *Higher-Order Function* is a function that takes functions as arguments or returns function.

map

- Applies a given function to every element of a collection
- Returns collection of outputs of that function
- input argument - same type as collection type
- return type - can be any type

map - Scala

```scala
def computeLength(w: String): Int = w.length

val words = List("when", "shall", "we", "three",
  "meet", "again")
val lengths = words.map(computeLength)

> lengths  : List[Int] = List(4, 5, 2, 5, 4, 5)
```

map - Scala syntactic sugar

```scala
//anonymous function (specifying input arg type)
val list2 = words.map((w: String) => w.length)

//let compiler infer arguments type
val list3 = words.map(w => w.length)

//use positionally matched argument
val list4 = words.map(_.length)
```

map - ScalaDoc

See immutable List ScalaDoc

```
List[+A]
...
final def map[B](f: (A) => B): List[B]
```

- Builds a new collection by applying a function to all elements of this list.
- B - the element type of the returned collection.
- f - the function to apply to each element.
- returns - a new list resulting from applying the given function f to each element of this list and collecting the results.

flatMap

- ScalaDoc:

  ```
  List[+A]
  ...
  def flatMap[B](f: (A) =>
          GenTraversableOnce[B]): List[B]
  ```

- GenTraversableOnce - List, Array, Option...
- can be empty collection or None
- flatMap takes each element in the GenTraversableOnce and puts it in order to output List[B]
- removes inner nesting - flattens
- output list can be smaller or empty (if intermediates were empty)

flatMap Example

```
val macbeth = """When shall we three meet again?
|In thunder, lightning, or in rain?""".stripMargin
val macLines = macbeth.split("\n")
// macLines: Array[String] = Array(
//  When shall we three meet again?,
// In thunder, lightning, or in rain?)

//Non-word character split
val macWordsNested: Array[Array[String]] =
      macLines.map{line => line.split("""\W+""")}
//Array(Array(When, shall, we, three, meet, again),
//      Array(In, thunder, lightning, or, in, rain))

val macWords: Array[String] =
     macLines.flatMap{line => line.split("""\W+""")}
//Array(When, shall, we, three, meet, again, In,
//      thunder, lightning, or, in, rain)
```

filter

```
List[+A]
...
def filter(p: (A) => Boolean): List[A]
```

- selects all elements of this list which satisfy a predicate.
- returns - a new list consisting of all elements of this list that satisfy the given predicate p. The order of the elements is preserved.

filter Example

```
val macWordsLower = macWords.map{_.toLowerCase}
//Array(when, shall, we, three, meet, again, in, thunder,
//      lightning, or, in, rain)

val stopWords = List("in","it","let","no","or","the")
val withoutStopWords =
  macWordsLower.filter(word => !stopWords.contains(word))
// Array(when, shall, we, three, meet, again, thunder,
//       lightning, rain)
```

reduce

```
List[+A]
...
def reduce[A1 >: A](op: (A1, A1) => A1): A1
```

- Creates one cumulative value using the specified associative binary operator.
- A1 - A type parameter for the binary operator, a supertype (super or same) of A. (List is covariant $+A$)
- op - A binary operator that must be associative.
- returns - The result of applying op between all the elements if the list is nonempty. Result is same type as (or supertype of) list type.
- UnsupportedOperationException if this list is empty.

reduce Example

```scala
//beware of overflow if using default Int!
val numberOfAttachments: List[Long] =
  List(0, 3, 4, 1, 5)
val totalAttachments =
  numberOfAttachments.reduce((x, y) => x + y)
//Order unspecified/non-deterministic, but one
//execution could be:
//0 + 3 = 3, 3 + 4 = 7,
//7 + 1 = 8, 8 + 5 = 13

val emptyList: List[Long] = Nil
//UnsupportedOperationException
emptyList.reduce((x, y) => x + y)
```

fold

```
List[+A]
...
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

- Very similar to reduce but takes start value z (a neutral value, e.g. 0 for addition, 1 for multiplication, Nil for list concatenation)
- returns start value z for empty list
- Note: See also foldLeft/Right (return completely different type)

  ```
  foldLeft[B](z: B)(f: (B, A)   B): B
  ```

fold Example

```
val numbers = List(1, 4, 5, 7, 8, 11)
val evenCount = numbers.fold(0) { (count, currVal) =>
  println(s"Count: $count, value: $currVal")
  if (currVal % 2 == 0) {
    count + 1
  } else {
    count
  }
}
Count: 0, value: 1
Count: 0, value: 4
Count: 1, value: 5
Count: 1, value: 7
Count: 1, value: 8
Count: 2, value: 11
evenCount: Int = 2
```

aggregate

```
List[+A]
...
def aggregate[B](z: B)(seqop: (B, A) => B,
                        combop: (B, B) => B): B
```

- More general than fold or reduce. Can return different result type.
- Apply seqop function to each partition of data.
- Then apply combop function to combine all the results of seqop.
- On a normal immutable list this is just a foldLeft with seqop (but on a parallelized list both operations are called).

aggregate Example

```
val wordsAll = List("when", "shall", "we", "three",
  "meet", "again", "in", "thunder", "lightning",
  "or", "in", "rain")
//Map(5 letter words ->3, 9->1, 2->4, 7->1, 4->3)
val lengthDistro = wordsAll.aggregate(Map[Int, Int]())(
  seqop = (distMap, currWord) =>
  {
    val length = currWord.length()
    val newCount = distMap.getOrElse(length, 0) + 1
    val newKv = (length, newCount)
    distMap + newKv
  },
  combop = (distMap1, distMap2) => {
    distMap1 ++ distMap2.map {
      case (k, v) =>
      (k, v + distMap1.getOrElse(k, 0))
    }
  })
```

So what does this have to do with Apache Spark?

- Resilient Distributed Dataset (RDD)
- From API docs: “immutable, partitioned collection of elements that can be operated on in parallel”
- map, flatMap, filter, reduce, fold, aggregate...

Spark - RDD API

- RDD API
- Transforms - map, flatMap, filter, reduce, fold, aggregate...
  - Lazy evaluation (not evaluated until action!)
- Actions - count, collect, first, take, saveAsTextFile...

Spark - From RDD to PairRDDFunctions

- If an RDD contains tuples (K,V) - can apply PairRDDFunctions
- Uses implicit conversion of RDD to PairRDDFunctions
- In 1.2 and previous versions available by importing org.apache.spark.SparkContext._

From org.apache.spark.SparkContext:

```scala
implicit def rddToPairRDDFunctions[K, V](
  rdd: RDD[(K, V)])
  (implicit kt: ClassTag[K],
    vt: ClassTag[V],
    ord: Ordering[K] = null) = {
      new PairRDDFunctions(rdd)
    }
```

PairRDDFunctions

- keys, values - return RDD of keys/values
- mapValues - transform each value with a given function
- flatMapValues - flatMap each value (0, 1 or more output per value)
- groupByKey - RDD[(K, Iterable[V])]
  - Note: expensive for aggregation/sum - use reduce/aggregateByKey!
- reduceByKey - return same type as value type
- foldByKey - zero/neutral starting value
- aggregateByKey - can return different type
- join (left/rightOuterJoin), cogroup ...

From RDD to DoubleRDDFunctions

- From API docs: “Extra functions available on RDDs of Doubles through an implicit conversion. Import org.apache.spark.SparkContext._”

```
From org.apache.spark.SparkContext:
implicit def doubleRDDToDoubleRDDFunctions(
  rdd: RDD[Double])
    = new DoubleRDDFunctions(rdd)
```

DoubleRDDFunctions

- mean, stddev, stats (count, mean, stddev, min, max)
- sum
- histogram

Analytic 1 - Mail Folder Statistics In MapReduce

- What are the least/most/average number of folders per user?
- Each MailRecord has user name and folder name

```
lay-k/        <- mailFields(UserName)
   business   <- mailFields(FolderName)
   family
   enron
   inbox
   ...
```

Hadoop Mail Folder Stats - Mapper

- read each mail record
- emits key: userName, value: folderName for each email

Hadoop Mail Folder Stats - Reducer

- reduce method
  - create set from values for a given key (unique folder names per user)
  - set.size == folder count
  - keep adding up all set.size (totalNumberOfFolders)
  - one up counter for each key (totalUsers)
  - keep track of min/max count
- cleanup method
  - compute average for this partition: totalNumberOfFolders/totalUsers
  - write out min, max, totalNumberOfFolders, totalUsers, avgPerPartition

Hadoop Mail Folder Stats - Driver

- Set Input/OutputFormat
- Number of reducers

Hadoop Mail Folder Stats - Results

- if only one reducer - results are overall lowest/highest/avg
- if multiple reducers
  - post-processing overall lowest/highest
  - add totalNumberOfFolders and totalUsers to compute overall average

Hadoop Mapper

```
public void map(AvroKey<MailRecord> key,
NullWritable value, Context context) throws ... {
  MailRecord mailRecord = key.datum();
  Map<CharSequence, CharSequence> mailFields =
      mailRecord.getMailFields();
  CharSequence userName =
      mailFields.get(AvroMailMessageProcessor.USER_NAME);
  CharSequence folderName =
      mailFields.get(AvroMailMessageProcessor.FOLDER_NAME);
  userKey.set(userName.toString());
  folderValue.set(folderName.toString());
  context.write(userKey, folderValue);
}
```

Hadoop Reducer

```
public void reduce(Text userKey,
  Iterable<Text> folderValues,
  Context context) throws ... {
  Set<String> uniqueFolders = new HashSet<String>();
  for (Text folder : folderValues) {
    uniqueFolders.add(folder.toString());
  }
  int count = uniqueFolder.size();
  if (count > maxCount) maxCount = count;
  if (count < minCount) minCount = count;
  totalNumberOfFolder += count
  totalUsers++
}
...
public void cleanup...
//write min, max, totalNumberOfFolders,
//totalUsers, avgPerPartition
```

Let's get to work