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.
- Learn to appreciate the Scala Collections Library and the Resilient Distributed Dataset

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 ever ask for. :)

Why Apache Spark?

- Three reasons

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
- Spurious Correlations

Your customer wants pretty little magical things

Figure 1: pretty little magical thing

WTF is Data Science?

- I am not sure. 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 (testable hypotheses followed by well-designed tests to clarify understanding) to that nebulous thing called "data."
- A couple of facts
  - 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.

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.

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]*

Higher-order function

Function that takes function as argument or returns function

map

- applies a given function to every element of a collection
- returns collection of output 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

```
//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

```
//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
```