Getting to Know Scala for Data Science

@TheTomFlaherty
Bio:

I have been a Chief Architect for 20 years, where I first became enamored by Scala in 2006. I wrote a symbolic math application in Scala at Glaxo in 2008 for molecular dynamics. In 2010 I formed the Front Range Polyglot Panel and participated as its Scala expert. I am currently learning all I can about Spark and applying it to analyzing the flow of information between enterprise architecture practices.
Abstract

- Scala has gained a lot of traction recently,
- Especially in Data Science with:
  - Spark
  - Cassandra with Spark Connector
  - Kafka
Scala's success factors for Data Science

- A Strong Affinity to Data
- State of the art OO for class composition
- Functional Programming with Streaming
- Awesome Concurrency under the Covers
- High performance in the cloud with Akka
- The Spark Ecosystem
- A vibrant Open Source community around Typesafe and Spark
About Scala

- State of the Art Class Hierarchy + Functional Programming
- Fully Leverages the JVM
  - Concurrency from Doug Lea
  - JIT (Just in Time) inlines functional constructs
  - Comparable in speed to Java ±3%
  - Strongly Typed
- Interoperates with Java
  - Can use any Java class (inherit from, etc.)
  - Can be called from Java
• Data Likes To:
  ▪ Declare Itself
  ▪ Assert Its Identity
  ▪ Be a First Class Citizen
  ▪ Remain Intact
  ▪ Be Wrapped
  ▪ Elevate Its Station in Life
  ▪ Reveal Itself
  ▪ Share Its Contents

• Data Scientists Like:
  ▪ A Universal Data Representation
  ▪ Location Aware Data
  ▪ To Simulate Things All at Once
  ▪ To Orchestrate Processing

• Spark
  ▪ Architecture
  ▪ DStreams
  ▪ Illustrated Examples
  ▪ RDD Resilient Distributed Data
  ▪ RDD Location Awareness

• RDD Workflow
  ▪ Processing Steps
  ▪ Spark Configuration and Context
  ▪ Load and Save Methods
  ▪ Transformation Methods
  ▪ Action Methods
  ▪ Word Count

• References
Let's Ask Data What It Likes:

<table>
<thead>
<tr>
<th>Data Likes To</th>
<th>Scala Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare Itself</td>
<td>Class and object</td>
</tr>
<tr>
<td>Assert its Identity</td>
<td>Strong Typing</td>
</tr>
<tr>
<td>Be a First Class Citizen</td>
<td>Primitives As Classes</td>
</tr>
<tr>
<td>Remain Intact</td>
<td>Immutability</td>
</tr>
<tr>
<td>Be Wrapped</td>
<td>Case Classes</td>
</tr>
<tr>
<td>Elevate is Station in Life</td>
<td>Math Expressions</td>
</tr>
<tr>
<td>Reveal Itself</td>
<td>Pattern Matching</td>
</tr>
<tr>
<td>Share its Contents</td>
<td>Pattern Transfer</td>
</tr>
</tbody>
</table>
Class and object Declarations

// [T] is a parameterized type for typing the contents with a class
// You can parameterize a class with many types [T,U,V]
// You can embed parameterized types [Key,List[T]]

trait Trait[T]{} ...

abstract class Abs[T](       i:Int ) extends Trait[T]{} ...

class Concrete[T](  i:Int ) extends Abs[T]( i:Int) {...}

case class Case[T](      i:Int ) extends Abs[T]( i:Int)
    with Trait1[T] with Trait2[T] {...}

// Singleton and Companion objects

object HelloWorld {
  def main (args:Array[String]) {
    println("Hello, world!") }
}

object Add {
  def apply(  u:Exp, v:Exp ) : Add = new Add(u,v)
  def unapply(  u:Exp, v:Exp ) : Option[(Exp,Exp)] = Some(u,v)
}
Assert Identity with Strong Typing
Functional Methods on Seq[T] Collections

```scala
def map[U]( f:(T) => U ) : Seq[U]  // T to U.
def flatMap[U]( f:(T) => Seq[U] ) : Seq[U]  // T to Flattened Seq[U]
def filter( f:(T) => Boolean ) : Seq[T]  // Keep Ts where f true

def exists( f:(T) => Boolean ) : Boolean  // True if one T passes

def forall( f:(T) => Boolean ) : Boolean  // True if all Ts passes

def reduce[U]( f:(T,T) => U ) : U  // Summarize f on T pairs

def groupBy[K]( f:T=>Key): Map[Key,Seq[T]]  // Group Ts into Map

val list = List( 1, 2, 3 )  // Scala nnfer List[Int]
list.map( (n) => n + 2 )  // List(3, 4, 5)
list.flatMap( (n) => List(n,n+1) )  // List(1,2,2,3,3,4)
list.filter( (n) => n % 2 == 1 )  // List(1, 3 )
list.exists( (n) => n % 2 == 1 )  // true list 1, 3 are odd
list.forall( (n) => n % 2 == 1 )  // false 2 ns even
list.reduce( (m,n) => m + n )  // 6
list.map( (n) => List(n,n+1) )  // List(List(1,2),List(2,3),List(3,4))
```
Data is First Class Citizen
with Scala’s Class Hierarchy

Any
AnyVal // Scala’s base class for Java primitives and Unit
        Double Float Long Int Short Char Byte Boolean Unit
scala:Array // compiles to Java arrays [] most of the time
AnyRef // compiles to java.lang.Object
        String // compiles to java.lang.String
        (all other Java Classes ...)
scala.ScalaObject
        (all other Scala Classes ...)
scala.Seq // base Class for all ordered collections
scala.List // Immutable list for pattern matching
scala.Option // Yields to Some(value) or None
scala.Null // Subtype of all AnyRefs. For Java best use Option
scala.Nothing // is a subtype of all Any classes. A true empty value

5.toString() // Valid because the compiler sees 5 as an object
             // then latter makes it a primitive in JVM bytecode
Staying Intact – Immutability Promotes:

- Improves reliability by removing side effects
- Concurrency, because state changes are impossible to synchronize
- Immuatable Object and values can be shared everywhere
- OO got it wrong with encapsulation and the set method
- Almost All OO values in Scala in public
- Data that is owned and encapsulated slowly dies.
- Shared data is living breathing data
// Scala expands the case class Add( u:Exp, v:Exp ) to:
class Add( val u:Exp, val v:Exp ) // Immutable Values
{
    def equals() : Boolean = {..} // Values compared recursively
    def hashCode : Int = {..} // hashCode from Values
    def toString() : String = {..} // Class and value names
}

// Scala creates a companion object with apply and unapply
object Add
{
    def apply( u:Exp, v:Exp ) : Add = new Add(u,v)
    def unapply( u:Exp, v:Exp ) : Option[(Exp,Exp)] = Some(u,v)
}
Abstract Syntax Trees

- All leaf nodes are either Var(v:String) or Num(d:Double)
- Branch nodes are ADTs that can be and infix “+” or the prefix Add
- Extracted contents (u,v) of an ADT i.e. Add(u,v) are the child nodes
- Branch child nodes are processed with a recursive method call
- Operators cannot be used on pattern side, only processing side
- Prefix form does not require Par(), but infix does.
- ADT prefix and infix can be mixed and checked by the compiler

\[
\left(\text{"}a\text{"}+2\right)\left(\text{"}b\text{"}-3\right) = \text{Mul(Add(Var("a"),Num(2)),Sub(Var("b"),Num(3)))}
\]
\[
\left(\text{"}a\text{"}+2\right)\left(\text{"}b\text{"}-3\right) = \text{Mul(Add("a",2),Sub("b",3))} \quad \text{// Implicit}
\]
\[
\left(\text{"}a\text{"}+2\right)\left(\text{"}b\text{"}-3\right) = \text{Mul("a"+2,"b"+3)} \quad \text{// Infix}
\]
Case Classes for Algebraic Expressions

```scala
case class Num(n: Double) extends Exp // wrap Double
case class Var(s: String) extends Exp // wrap String
case class Par(u: Exp) extends Exp // parentheses

case class Neg(u: Exp) extends Exp // -u prefix

case class Pow(u: Exp, v: Exp) extends Exp // u ~^ v infix

case class Mul(u: Exp, v: Exp) extends Exp // u * v infix

case class Div(u: Exp, v: Exp) extends Exp // u / v infix

case class Add(u: Exp, v: Exp) extends Exp // u + v infix

case class Sub(u: Exp, v: Exp) extends Exp // u - v infix

case class Dif(u: Exp) extends Exp // Differentiate
```

Elevating Data’s Station in Life

Exp - Base Math Expression with Math Operators

sealed abstract class Exp extends with Differentiate with Calculate
{
  // Wrap i:Int and d:Double to Num(d) & String to Var(s)
  implicit def int2Exp( i:Int ) : Exp = Num(i.toDouble)
  implicit def dbl2Exp( d:Double ) : Exp = Num(d)
  implicit def str2Exp( s:String ) : Exp = Var(s)

  // Infix operators from high to low using Scala precedence
  def ~^ ( v:Exp ) : Exp = Pow(this,v) // ~^ high precedence
  def / ( v:Exp ) : Exp = Div(this,v)
  def * ( v:Exp ) : Exp = Mul(this,v)
  def - ( v:Exp ) : Exp = Sub(this,v)
  def + ( v:Exp ) : Exp = Add(this,v)

  // Prefix operator for negation
  def unary_~ : Exp = Neg(this)
}
trait Differentiate
{
  this:Exp => // Ties Differentiate to Exp

  def d( e:Exp ) : Exp = e match
  {
    case Num(n) => Num(0)       // diff of constant zero
    case Var(s) => Dif(Var(s))   // x becomes dx
    case Par(u) => Par(d(u))
    case Neg(u) => Neg(d(u))
    case Pow(u,v) => Mul(Mul(v,Pow(u,Sub(v,1))),d(u))
    case Mul(u,v) => Mul(Add(Mul(v,d(u))),u),d(v))
    case Div(u,v) => Div(Sub(Mul(v,d(u)),Mul(u,d(v)) ),Pow(v,2))
    case Add(u,v) => Add(d(u),d(v))
    case Sub(u,v) => Sub(d(u),d(v))
    case Dif(u) => Dif(d(u))     // 2nd dif
  }
}
trait Differentiate
{
    this:Exp => // Ties Differentiate to Exp

    def d( e:Exp ) : Exp = e match
    {
        case Num(n) => 0       // diff of constant zero
        case Var(s)  => Dif(Var(s))  // "x" becomes dx
        case Par(u)  => Par(d(u))
        case Neg(u)  => -d(u)
        case Pow(u,v) => v * u^^(v-1) * d(u)
        case Mul(u,v) => v * d(u) + u * d(v)
        case Div(u,v) => Par( v*d(u) - u*d(v) ) / v^2
        case Add(u,v) => d(u) + d(v)
        case Sub(u,v) => d(u) - d(v)
        case Dif(u)  => Dif(d(u)) // 2nd dif
    }
}
## What Do Data Scientists Like?

<table>
<thead>
<tr>
<th>Data Scientists Like</th>
<th>Spark Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Universal Data Representation</td>
<td>RDD Resilient Distributed Data</td>
</tr>
<tr>
<td>Location Aware Data</td>
<td>Five Main RDD Properties</td>
</tr>
<tr>
<td>To Simulate Things All at Once</td>
<td>Concurrency</td>
</tr>
<tr>
<td>To Orchestrate Processing</td>
<td>Streams</td>
</tr>
</tbody>
</table>
The DStream Programming Model

- Discretized Stream (DStream)
  - Represents a stream of data
  - Implemented as a sequence of RDDs
- DStreams can be either...
  - Created from streaming input sources
  - Created by applying transformations on existing DStreams
Illustrated Example 1 - Initialize an Input DStream

```scala
val scc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(scc, auth)
// tweets are an Input DStream
```

![Diagram](image_url)
Illustrated Example 2 - Get Hash Tags from Twitter

```scala
val scc = new StreamingContext( sparkContext, Seconds(1) )
val tweets = TwitterUtils.createStream( ssc, None )
val hashTags = tweets.flatMap( status => getTags( status )
```

![Diagram showing the process of getting hash tags from Twitter](image-url)
val scc = new StreamingContext( sparkContext, Seconds(1) )
val tweets = TwitterUtils.createStream( ssc, None )
val hashTags = tweets.flatMap( status => getTags( status )
hashTags.saveAsHadoopFiles( "hdfs://..." )
Illustrated Example 4 - Sliding Window

```scala
val tweets = TwitterUtils.createStream( ssc, None )
val hashTags = tweets.flatMap( status => getTags( status )
val tagCounts = hasTags.window( Minutes(1), Seconds(5) ).countByValue()
//                         ^       ^             ^
// (sliding window operation) (window length) (sliding interval)
```
RDD Resilient Distributed Data
Five main properties for RDD Location Awareness

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Hash Partitioner for key-value RDDs
- Optionally, a list of preferred locations to compute each split
RDD Workflow

Load -> RDD -> Transform -> Action -> Value
Processing Steps

- Configure Spark
- Create Spark Context
- Load RDDs
- Transform RDDs
- Produce Results with Actions
- Save RDDs and Results
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object MySparkProgram {

  def main( args: Array[String] ) = {

    sc = new SparkContext( master:String, appName, sparkConf )

    ... RDD Workflow here

  }

}
Spark Context Load Save Methods plus Cassandra

// Load Methods
type S = String
def textFile( path:S ) : RDD[St]
def objectFile[T]( path:S ) : RDD[T]
def sequenceFile[K,V]( path:S ) : RDD[(K,V)] // load Hadoop formats
def wholeTextFiles( path:S ) : RDD[(S,S)] // Directory of HDFS files
def parallelize[T]( seq:Seq[T] ) : RDD[T] // convert a collection
def cassandraTable[Row]( keyspace:S, table:S ) : CassandraRDD[Row]

// Save Methods
def saveAsTextFile( path:S ) Unit
def saveAsObjectFile path:S ) Unit
def saveToCassandra( keyspace:S, table:S ) // Spark Cassandra Connector

// Load an RDD from Cassandra
rdd = sc.cassandraTable( keyspace, table)
  .select("user", "count", "year", "month")
  .where("commits >= ? and year = ??", 1000, 2015)
Transformation Methods on RDD[T]

```scala
def map[U]( f:(T) => U ) : RDD[U]
def flatMap[U]( f:(T) => Seq[U] ) : RDD[U]
def filter( f:(T) => Boolean ) : RDD[T]

def keyBy[K]( f:(T) => K ) : RDD[(K,T)]
def groupBy[K]( f:(T) => K ) : RDD[(K,Seq[T])]
def sortBy[K]( f:(T) => K ) : RDD[T]

def distinct( ) : RDD[T]
def intersection( rdd:RDD[T] ) : RDD[T]
def subtract( rdd:RDD[T] ) : RDD[T]
def union( rdd:RDD[T] ) : RDD[T]
def cartesian[U]( rdd:RDD[U] ) : RDD[(T,U)]
def zip[U]( rdd:RDD[U] ) : RDD[(T,U)]

def sample( r:Boolean, f[Double, s:Long ]: RDD[T]
def pipe(command: String): RDD[String]
```
Transformation on RDD[(K,V)] Key Value Tuples

def groupByKey() : RDD[(K,Seq[V])]
def reduceByKey(f: (V,V) => V) : RDD[(K,V)]
def foldByKey(z:V)( f:(V,V) => V) : RDD[(K,V)]
def aggregateByKey[U](z:U)( s:(U,V)=>U, c:(U,U)=>U) : RDD[(K,U)]

def join[U]( rdd:RDD[(K,U)] ) : RDD[(K,(V,U))]  //groupWith
def cogroup[U]( rdd:RDD[(K,U)] ) : RDD[(K,(Seq[V],Seq[U]))]

def countApproxDistinctByKey(relativeSD: Double) : RDD[(K, Long)]
def flatMapValues[U](f: (V) => TraversableOnce[U]) : RDD[(K, U)]

type Opt[X] = Option[X]
def fullOuterJoin[U]( rdd:RDD[(K,U)] ) : RDD[(K,(Opt[V], Opt[U]))]
def leftOuterJoin[U]( rdd:RDD[(K,U)] ) : RDD[(K,(V, Opt[U]))]
def rightOuterJoin[U]( rdd:RDD[(K,U)] ) : RDD[(K,(Opt[V], U ))]

def keys: RDD[K]
def mapValues[U](f: (V) => U ) : RDD[(K,U)]
def sampleByKey( r:Boolean, f:Map[K,Double], s:Long ) : RDD[(K,V)]
// Trigger execution of DAG.
def reduce(  f:(T,T) => T  ) : T
def fold(z:T)(  f:(T,T) => T  ) : T
def min() : T
def max() : T
def first() : T
def count() : Long
def countByKey() : Map[K,Long]
def collect() : Array[T]
def top( n:Int ) : Array[T]
def take( n:Int ) : Array[T]
def takeOrdered( n:Int ) : Array[T]
def takeSample( r:Boolean, n:Int, s:Long ) : Array[T]
def foreach( f:(T) => Unit ) : Unit // For side effects
val rdd = sc.textFile("README.md")

rdd.flatMap(l => l.split(" "))
  .map(w => (w, 1))
  .reduceByKey(_ + _)
  .saveAsTextFile("WordCount.txt")
```scala
val rddLines : RDD[String] = sc.textFile( "README.md" )
val rddWords : RDD[String] = rddLines.flatMap( (line) => line.split(" "))
val rddWords1 : RDD[(String,Int)] = rddWords.map( (word) => (word,1) )
val rddCount : RDD[(String,Int)] = rddWords1.reduceByKey( (c1,c2) => c1 + c2 )
rddCount.saveAsTextFile( "WordCount.txt" )
```
References

- The Scala Language  http://www.scala-lang.org/
- Apache Spark  https://spark.apache.org/
- Dean Wampler on Spark  http://deanwampler.github.io/
- These slides in PDF  https://speakerdeck.com/axiom6
THE END