CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2016
Doc 24 Spark Intro
Nov 22, 2016

Some Spark Resources

Spark Programming Guide

http://spark.apache.org/docs/latest/programming-guide.html#shuffle-operations

https://goo.gl/hQgxkL

Hadoop: The Definitive Guide, Chapter 19

Scala

General-purpose programming language by Martin Odersky

Inspired by Java's shortcomings

Runs on JVM, Interoperates with Java

Functional & object-oriented programming

Strong static type system with type inference

Functional features

- Currying, immutability, lazy evaluation

Features not in Java

- Operator overloading, optional parameters
- Names parameters, raw strings
- No checked exceptions
- REPL

Time Line

1991 - Java project started

1995 - Java 1.0 released, Design Patterns book published

2000 - Java 3

2001 - Scala project started

2002 - Nutch started

2004 - Google MapReduce paper

Scala version 1 released

2005 - F# released

2006 - Hadoop split from Nutch

Scala version 2 released

2007 - Clojure released

2009 - Spark project started

2012 - Hadoop 1.0

2014 - Spark 1.0

Some Scala

```
val constant: Int = 21 *2
constant = 10  // error

val inferred = 42

var canChange = 42
canChange = 11


val scalaObject: Array[Int] = Array(1,2,3)
scalaObject(1) = 20


 val javaObject = new java.util.ArrayList[String]()
```

Scala Functions Methods

```
def abs(x: Double): Double = if (x >= 0) x else -x
def abs(x: Double) = if (x >= 0) x else -x


def fac(n : Int) = {
  var r = 1
  for (i <- 1 to n) r = r * i
  r
}
```

```
def hello(name:String) {println(s"Hello $name!")}
def hello(name:String):Unit =  {println("Hello " + name + "!")}


val functionRef = hello _
val functionRef2 = hello(_)
```

Anonymous Functions & Function Arguments

```
val twice = (x: Double) => 2 *x
twice(3)
```

```
val data = Array(1, 2, 3)
data.map( (x:Int) => 2*x)        //Array(2, 4, 6)
data.map( (x) => 2*x)
data.map(x => 2*x)
data.map( 2*_)

data.map{ (x:Int) => 2*x}
data.map { (x:Int) => 2*x}
data.map { x => 2*x}
data.map { 2*_}
data map { 2*_}
```

Note: You don't implement map
You pass a function to map

Classes

```
class Counter {

  var value = 0 // You must initialize the field

  def increment() { value += 1 } // Methods are public by default

  def current():Int = value
}

val myCounter = new Counter // Or new Counter()

myCounter.increment()
myCounter.increment
```

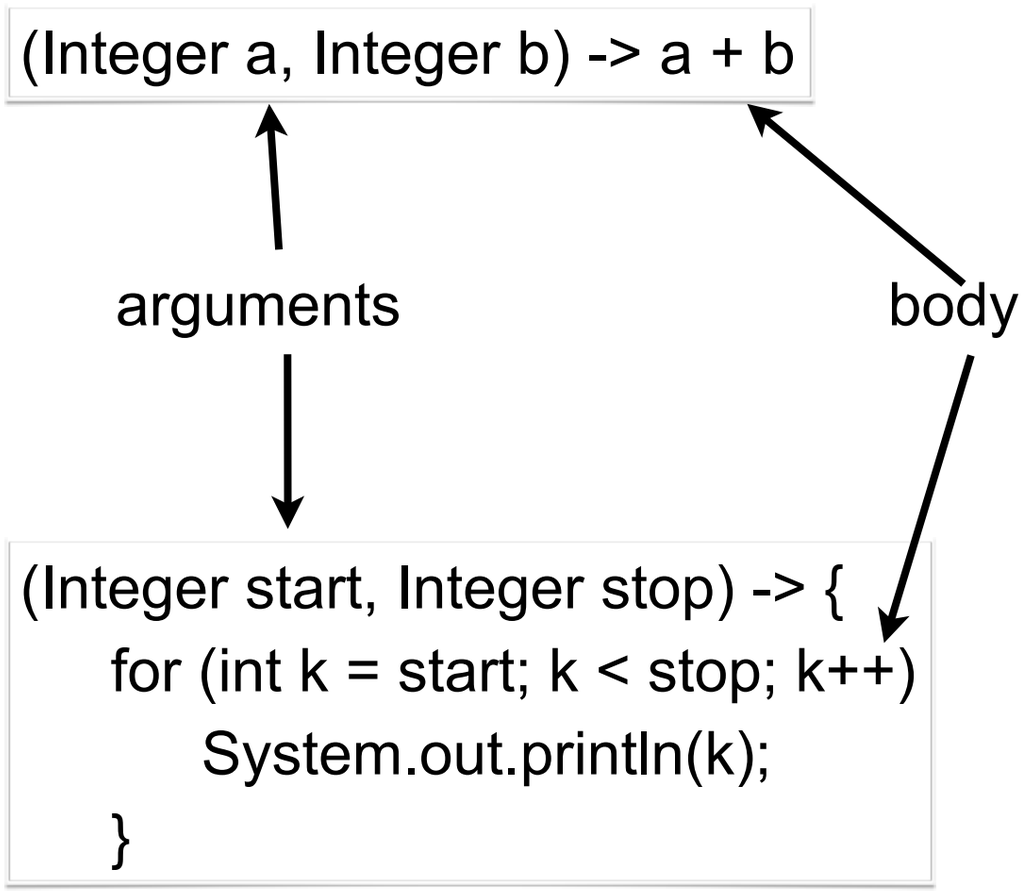
Objects

```
object Counter {
  def create(start:Int) = {
    val aCounter = new Counter()
    aCounter.value = start
    aCounter
  }

  def main(args: Array[String]) = {
    val test = Counter.create(5)
    test.increment()
    println("The value is" + test.current())
  }
}
```

Java Lambda Expression - Java 8

Anonymous Function



Short Version of Java Lambda Syntax

(String text) -> text.length();

text -> text.length();

(Integer a, Integer b) -> a + b

(a, b) -> a + b

Using Lambdas

Function<String,Integer> length = **text -> text.length();**

int nameLength = **length.apply**("Roger Whitney");

BiFunction<Integer,Integer,Integer> adder = **(a, b) -> a + b;**

int sum = **adder.apply**(1, 2);

Other Types of Lambdas

```
Predicate<Integer> isLarge =  value -> value > 100;
if (isLarge.test(59))
      System.out.println("large");
```

```
Consumer<String> print = text -> System.out.println(text);
print.accept("hello World");
```

```
int size = xxx;
      Supplier<List> listType = size > 100 ? (()-> new ArrayList()): (() -> new Vector());
      List elements = listType.get();
      System.out.println(elements.getClass().getName());
```

Lambda Types

New - See java.util.function Interfaces

Predicate<T> -- a boolean-valued property of an object
Consumer<T> -- an action to be performed on an object
Function<T,R> -- a function transforming a T to a R
Supplier<T> -- provide an instance of a T (such as a factory)
UnaryOperator<T> -- a function from T to T
BinaryOperator<T> -- a function from (T, T) to T

Pre-existing

java.lang.Runnable
java.util.concurrent.Callable
java.security.PrivilegedAction
java.util.Comparator
java.io.FileFilter
java.beans.PropertyChangeListener
etc.

Functional Interfaces

Interface with one method

Can be used to hold a lambda

java.lang.Runnable

```
void run()
```

Runnable Example

```
Runnable test = () -> System.out.println("hello from thread");
Thread example = new Thread(test);
example.start();
```

OnClickListener Example

```
button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View source) {
                makeToast();
        }
});
```

```
button.setOnClickListener( event -> makeToast());
```

Lazy Evaluation

```
String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);

wordList.stream()
                    .map( s -> s + "ed")
                    .map( s ->  "de" +  s)
                    .map( s -> s.toUpperCase());
```

Spark

Created at UC Berkeley's AMPLab

- 2009 Project started
- 2014 May Version 1.0
- 2016 July Version 2.0.2

Programming interface for

- Java, Python, Scala, R

Interactive shell for

- Python, Scala, R (experimental)

Runs on

- Linux, Mac, Windows

Cluster manager

- Native Spark cluster
- Hadoop YARN
- Apache Mesos

File System

- HDFS
- MapR File System
- Cassandra
- OpenStack Swift
- S3

Pseudo-Distributed Mode

- Single machine
- Uses local file system

Word Count - Scala Version

```
val data = sc.textFile("/Users/whitney/Downloads/MarkTwain.txt")

val tokens = data.flatMap(_.split(" "))

val wordFreq = tokens.map((_,1)).reduceByKey(_ + _)

val sortedWords = wordFreq.sortBy(s => -s._2)

val result = sortedWords.collect()

val out = new java.io.PrintWriter("TwainWords.txt")

result.foreach(out.println(_))
```

Type inference is nice until you need to know what methods you can use

Word Count - Java Version

```
public final class JavaWordCount {
   private static final Pattern SPACE = Pattern.compile(" ");
   public static void main(String[] args) throws Exception {
      if (args.length < 2) {
         System.err.println("Usage: JavaWordCount <input> <output>");
         System.exit(1);
      }
      JavaSparkContext sc = new JavaSparkContext();

      JavaRDD<String> lines = jsc.textFile(args[0]);
      JavaRDD<String> words = lines.flatMap(line -> Arrays.asList(SPACE.split(line)).iterator());
      JavaPairRDD<String, Integer> ones = words.mapToPair(word -> new Tuple2<>(word, 1));
      JavaPairRDD<String, Integer> counts = ones.reduceByKey( (count1, count2) ->
                                                                  count1 + count2);
      counts.saveAsTextFile(args[1]);
      sc.stop();
   }
}
```

Major Parts of Spark

Spark Core

- Resilient Distributed Dataset (RDD)

Spark SQL

- SQL, csv, json
- Dataframe

Spark Streaming

- Near real-time response

MLib Machine Learning Library

- Statistics, regression, clustering, dimension reduction, feature extraction
- Optimization

GraphX

Major Data Structures

Resilient Distributed Datasets (RDDs)

- Fault-tolerant collection of elements that can be operated on in parallel

Dataframes & Dataset

- Fault-tolerant collection of elements that can be operated on in parallel
- Rows & Columns
- JSON, csv, SQL tables
- Part of SparkSQL

Spark Context

Connection to Spark cluster
Runs on master node
Used to create RDDs, accumulators, broadcast variables
Only one SparkContext per JVM
stop() the current SparkContext before starting another

SparkContext org.apache.spark.SparkContext
Scala version

JavaSparkContext org.apache.spark.api.java.JavaSparkContext
Java version

SparkSession org.apache.spark.sql.SparkSession
Contains a SparkContext
Entry point to use Dataset & DataFrame

Accessing SparkContext

```
import org.apache.spark.api.java.JavaSparkContext;

JavaSparkContext jsc = new JavaSparkContext();
```

Versions of constructor with arguments

```
import org.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
  .builder()
  .appName("JavaWordCount")
  .getOrCreate();
```

Resilient Distributed Datasets (RDDs)

RDDs are read only
Distributed across cluster
Transformations on RDDs are done in parallel

Transformations
- Performed on workers in parallel
- Lazy
- Create new RDDs

Actions
- Triggers all previous transformations
- Returns results to master
- Saves data

Common RDD Transformations

map(func)

filter(func)
flatMap(func)
mapPartitions(func)
mapPartitionsWithIndex(func)
sample(withReplacement, fraction, seed)
union(otherDataset)
intersection(otherDataset)
distinct([numTasks]))
groupByKey([numTasks])
reduceByKey(func, [numTasks])
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])
sortByKey([ascending], [numTasks])
join(otherDataset, [numTasks])
cogroup(otherDataset, [numTasks])
cartesian(otherDataset)
pipe(command, [envVars])
coalesce(numPartitions)
repartitionAndSortWithinPartitions(partitioner)

Common RDD Actions

reduce(func)

collect()
count()
first()
take(n)
takeSample(withReplacement, num, [seed])
takeOrdered(n, [ordering])
saveAsTextFile(path)
saveAsSequenceFile(path)
saveAsObjectFile(path)
countByKey()
foreach(func)

Creating RDDs

Parallelize an existing collection

```
JavaSparkContext sc = new JavaSparkContext();

List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

parallelize(java.util.List<T> list)

parallelizePairs(java.util.List<scala.Tuple2<K,V>> list)

parallelizeDoubles(java.util.List<Double> list)

scala.Tuple2

Read-only pair of values

Like Hadoop Spark deals with key-value pairs represented as a object

Java does not have a Tuple class so use Scala's Tuple2 class

```
import scala.Tuple2;
Tuple2<String, Integer> sample = new Tuple2<>("key", 12);
String key = sample._1();
key = sample._1;
int value = sample._2();

Tuple2<Integer, String> swapped = sample.swap();
```

Creating RDDs

Read from a data source

```
JavaRDD<String> lines = sc.textFile("s2://rw-wc-input-data/foo.txt");
```

JavaRDD<String> textFile(String path)

Reads contents of file as lines

JavaPairRDD<String,String> wholeTextFiles(String path)

path - to a directory

Reads all files in directory as key-value pair

key - path of each file

value - content of each file

JavaPairRDD<String,PortableDataStream> binaryFiles(String path)

Reads contents of all files in directory as byte array

JavaPairRDD<K,V>

org.apache.spark.api.java.JavaPairRDD<K,V>

RDD of scale.Tuple2<K,V>

Simple Program - Total line length

```
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;

public final class JavaWordCount {

    public static void main(String[] args)  {
        JavaSparkContext sc = new JavaSparkContext();

        JavaRDD<String> lines = sc.textFile(args[0]);                    no action

        JavaRDD<Integer> lineLengths = lines.map(s -> s.length());       no action

        int totalLength = lineLengths.reduce((a, b) -> a + b);           Now do everything

        System.out.println("Total length = " + totalLength);
        sc.stop();
    }
}
```

Total Line length + Max line Length

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<String> lines = sc.textFile(args[0]);
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());

int totalLength = lineLengths.reduce((a, b) -> a + b);

int maxLength = lineLengths.reduce((a, b) -> Math.max(a, b));

sc.stop();
```

Problem - File is read twice, lineLengths is “created” twice

Total Line length + Max line Length

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<String> lines = sc.textFile(args[0]);
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());

lineLengths.persist(StorageLevel.MEMORY_ONLY());

int totalLength = lineLengths.reduce((a, b) -> a + b);

int maxLength = lineLengths.reduce((a, b) -> Math.max(a, b));

sc.stop();
```

Now lineLengths is only computed once
But consumes memory

persist & unpersist

persist(StorageLevel newLevel)

MEMORY_AND_DISK_SER()
MEMORY_AND_DISK()
MEMORY_ONLY_SER()
MEMORY_ONLY()
MEMORY_ONLY_2, MEMORY_AND_DISK_2 - replicate on two nodes

unpersist(boolean blocking)

blocking - Whether to block until all blocks are deleted

Closures & Multiple Machines

```
JavaSparkContext sc = new JavaSparkContext();
JavaRDD<String> lines = sc.textFile(args[0]);
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());

int totalLength = 0;

lineLengths.foreach(x -> totalLength += x);

System.out.println(totalLength);                    // prints 0
sc.stop();
```

Main runs on master

RDD computations are run on slaves

When slave local state is copied to slave

Word Count - Java Version

```
public final class JavaWordCount {
    private static final Pattern SPACE = Pattern.compile(" ");
    public static void main(String[] args) throws Exception {
        JavaSparkContext sc = new JavaSparkContext();
        JavaRDD<String> lines = sc.textFile(args[0]);

        JavaRDD<String> words =
                            lines.flatMap(line -> Arrays.asList(SPACE.split(line)).iterator());

        JavaPairRDD<String, Integer> ones =
                            words.mapToPair(word -> new Tuple2<>(word, 1));

        JavaPairRDD<String, Integer> counts =
                    ones.reduceByKey( (count1, count2) -> count1 + count2);

        JavaPairRDD<Integer, String> sorted = counts.sortByKey();
        sorted.saveAsTextFile(args[1]);
        sc.stop();
    }
}
```

flatMap

```
JavaRDD<U> flatMap(FlatMapFunction<T,U> f)
```

```
public interface FlatMapFunction<T,R> {
        java.util.Iterator<R>   call(T t) throws Exception
}
```

Lambda verses Anonymous Class

```
JavaRDD<String> words =
                              lines.flatMap(line -> Arrays.asList(SPACE.split(line)).iterator());
```

```
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  @Override
  public Iterator<String> call(String s) {
    return Arrays.asList(SPACE.split(s)).iterator();
  }
});
```

Shuffle

```
JavaPairRDD<Integer, String> sorted = counts.sortByKey();
```

Data is sorted globally

So data needs to be shuffled between machines

Operations that cause a shuffle

- repartition, coalesce
- groupByKey, reduceByKey, other ‘ByKey
- cojoin, join
- sortBy