Big Data Frameworks: Scala Tutorial +Developing Spark Algorithms

14.03.2017

Eemil Lagerspetz, Mohammad Hoque, Ella Peltonen, Professor Sasu Tarkoma

These slides: https://is.gd/bigdataalgo2017

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Outline

- Functional Programming and Scala Tutorial
- Spark Algorithms: Why?
- Designing Spark algorithms
- Important numbers
- The Hard Part

Part I: Functional Programming and Scala Tutorial

You can ask anything at any time. We can concentrate on aspects of Scala that you find useful to explore.

To start with Scala, either download Spark pre-built for Hadoop 2.7 or later at https://spark.apache.org/downloads.html

Spark works currently with Scala 2.11 (currently comes with 2.11.8)

Or start using Scala immediately at http://www.tryscala.com/ or http://www.scalakata.com/

You can also download Scala without Spark at http://www.scala-lang.org/

Or the Scala IDE which is based on eclipse: http://scala-ide.org/

Functional Programming

Functional operations create new data structures, they do not modify existing ones

-Immutability helps with debugging threaded access problems, immutable collections are inherently thread-safe

After an operation, the original data still exists in unmodified form

The program design implicitly captures data flows

The order of the operations is not significant

Word Count in Scala

```
val lines = scala.io.Source.fromFile("textfile.txt").getLines
val words = lines.flatMap(line => line.split(" "))
val counts = words.groupBy(identity).map(words =>
  words._1 -> words._2.size)
val top10 = counts.toSeq.sortBy(_._2).reverse.take(10)
println(top10.mkString("\n"))
```

Scala can be used to concisely express pipelines of operations

Map, flatMap, filter, groupBy, … operate on *entire collections* with one element in the function's scope at a time

This allows **implicit parallelism** in Spark

About Scala

Scala is a statically typed language

Support for generics: `case class MyClass(a: Int) implements Ordered[MyClass]`

All the variables and functions have types that are defined at compile time

The compiler will find many unintended programming errors

The compiler will try to infer the type, say “val=2” is implicitly of integer type

→ Use an IDE for complex types: http://scala-ide.org or IDEA with the Scala plugin

Everything is an object

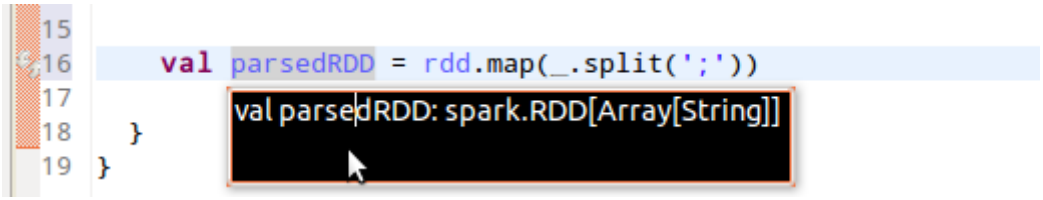
Functions defined using the def keyword

Laziness, avoiding the creation of objects except when absolutely necessary

Online Scala coding: http://www.tryscala.com/

A Scala Tutorial for Java Programmers

http://www.scala-lang.org/docu/files/ScalaTutorial.pdf



Functions are objects

```
def noCommonWords(w: (String, Int)) = { // Without the =, this would be a void (Unit) function
  val (word, count) = w
  word != "the" && word != "and" && word.length > 2
    }

val better = top10.filter(noCommonWords)
println(better.mkString("\n"))
```

Functions can be passed as arguments and returned from other functions

Functions as filters

They can be stored in variables

This allows flexible program flow control structures

Functions can be applied for all members of a collection, this leads to very compact coding

Notice above: the return value of the function is always **the value of the last statement**

Scala Notation

‘_’ is the default value or wild card
‘=>’ Is used to separate match expression from block to be evaluated
The anonymous function ‘(x,y) => x+y’ can be replaced by ‘_+_’
The ‘v=>v.Method’ can be replaced by ‘_.Method’

"->" is the tuple delimiter
Iteration with for:

```
val list =  Seq(1, 2, 4, 6, 7, 8, 10, 12, 3, 2)
for (i <- 0 until 10) // with 0 to 10, 10 is included
  println(s"Item $i: ${list(i)}") // String interpolation: s”$i” or f”$i%2.2f”
// See also: http://docs.scala-lang.org/overviews/core/string-interpolation.html
```

More examples:

```
words.filter(v => v.length>2) // is the same as
words.filter(_.length>2)
// (2, 3) is equal to 2 -> 3
2 -> (3 -> 4) == (2,(3,4))
2 -> 3 -> 4 == ((2,3),4)
```

Scala Examples

map: val out = input.map(f) // or for example val out = lsts.map(x => x * 4)

Instantiates a new list by applying f to each element of the input list.

flatMap: uses the given function to create a new list, then places the resulting list elements at the top level of the collection. For example: `val words = lines.flatMap(line => line.split(" "))`

lsts.sort(_<_): sorting ascending order

fold and **reduce** functions combine adjacent list elements using a function. Processes the list starting from left or right:

lst.foldLeft(0)(_+_) starts from 0 and adds the list values to it iteratively starting from left

tuples: a set of values enclosed in parenthesis (2, ‘z’, 3), access with the underscore: (2, ‘z’, 3)._2 == ‘z’

Single-statement functions do not need curly braces { }: `def sum(list: List[Int]) = list.foldLeft(0)(_ + _)`

- Arrays are indexed with ( ), not [ ]. [ ] is used for type bounds (like Java's < >)

REMEMBER: these do not modify the collection, but create a new one, so you need to assign it: val sorted = lsts.sort(_ < _)

Implicit parallelism

The map function has implicit parallelism as we saw before

This is because the order of the application of the function to the elements in a list is commutative

We can parallelize or reorder the execution in Scala with

lsts.par.map(…)

MapReduce and Spark build on this parallelism and make it work across many computers

Map and Fold is the Basis

Map takes a function and applies to every element in a list

Fold iterates over a list and applies a function to aggregate the results

The map operation can be parallelized: each application of function happens in an independent manner

The fold operation has restrictions on data locality

Elements of the list must be together before the function can be applied; however, the elements can be aggregated in groups in parallel

Scala 2.11 vs 2.10

Smaller, faster, xml functions moved to separate library

Scala shell improved: http://docs.scala-lang.org/scala/2.11/

Experimental support for Java 8 (I think this works just fine)

Many compiler improvements

Part II: Spark Algorithms

You can ask anything at any time. We can concentrate on e.g. flatMap or another issue that you find useful to explore.

Installing Spark

I use Spark 2.1.0 or newer.

For local installation:

Download http://spark.apache.org/downloads.html , go for pre-built for Hadoop 2.7 or later from FUNET mirror

Extract it to a folder of your choice and run bin/spark-shell in a terminal

(or double click bin/spark-shell.cmd on Windows)

For Eclipse / another IDE, take the assembly jar from spark-2.1.0/assembly/target/scala-2.11 or spark-2.1.0/lib

Spark can be used in larger projects with Maven or SBT, or Gradle.

For this course, you can compile code with an IDE with the jar in the classpath, and run your classes or jar with spark-submit. More sophisticated build tools are not necessary, but you are free to use them.

You need to have

- Java 7+
- For pySpark: Python 2.6+

For Cluster installations

Each machine will need Spark in the same folder, and key-based SSH access from the master for the user running Spark. You can have a password in the key, in that case, use ssh-add /path/to/keyfile to add that in your ssh keyring before starting the Spark Master

Slave machines will need to be listed in the slaves file

See spark/conf/

For better performance: Spark running in the YARN scheduler

http://spark.apache.org/docs/latest/running-on-yarn.html

Running Spark on Amazon AWS **EC2**: http://spark.apache.org/docs/latest/ec2-scripts.html

Further reading: Running Spark on Mesos

http://spark.apache.org/docs/latest/running-on-mesos.html

First examples

```
# Running the shell with your own classes, given amount of memory, and
# the local computer with two threads as slaves
./bin/spark-shell --driver-memory 1G \
  --jars your-project-jar-here.jar \
  --master "local[2]"

// And then creating some data
val data = 1 to 5000
data: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, …
// Creating an RDD for the data:
val dData = sc.parallelize(data)

// Then selecting values less than 10
dData.filter(_ < 10).collect()
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

SparkContext and SparkSession

A Spark program creates a SparkContext object, denoted by the “sc” variable in Scala and Python shell. In Spark 2.0+, you also have access to a SparkSession, usually called “spark”

Outside of the Spark shell, a builder is used to instantiate a SparkSession:

```
val spark = SparkSession
      .builder()
      .appName("my app").getOrCreate()
val sc = spark.sparkContext
```

SparkContext and SparkSession are used to interact with the Spark cluster

SparkContext and SparkSession: Which one to use?

SparkSession gives access to SQL-like operations and many data formats, e.g.

```
val jsDataFrame = spark.read.json(“filename.json”)
val csvDataFrame = spark.read.csv(“filename.csv”)
```

SparkContext gives access to RDDs, allowing more precise programming.

You can always convert RDDs to DataFrames and back to switch between the two (jsDataFrame.rdd).

The following slides will use SparkContext, and in the end, we will take a look at loading various data types with SparkSession.

SparkContext master parameter

Can be given to spark-shell, specified in code, or given to spark-submit

Code takes precedence, so don't hardcode this

Determines which cluster to utilize

local
    with one worker thread
local[K]
    local with K worker threads
local[*]
    local with as many threads as your computer has logical cores
spark://host:port
    Connect to a Spark cluster, default port 7077
mesos://host:port
    Connect to a Mesos cluster, default por 5050

Example: Log Analysis

```
/* Java String functions (and all other functions too) also work in Scala */
val lines = sc.textFile("hdfs://..." )
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(_(1))
messages.persist()
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

```
File access methods with sc.textFile, sc.objectFile, spark.read.[filetype]:
   hdfs://server:port/directory/…/  | Access files and dirs on Hadoop's HDFS
   file:///home/user/Desktop/file.csv  | Access local files and dirs
   /somefile.csv  | Without a scheme, this defaults to HDFS, so remember file:///
```

WordCounting

```
/* When giving Spark file paths, those files need to be accessible with the same
   path from all slaves */
val file = sc.textFile("file:///home/user/Desktop/README.md")
val wc = file.flatMap(l => l.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

wc.saveAsTextFile("file:///home/user/Desktop/wc_out.txt")
wc.collect.foreach(println)
```

Join

```
val f1 = sc.textFile("file:///home/user/Desktop/README.md")
val sparks = f1.filter(_.startsWith("Spark"))
val wc1 = sparks.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ +
    _)

val f2 = sc.textFile("file:///home/user/Desktop/CHANGES.txt")
val sparks2 = f2.filter(_.startsWith("Spark"))
val wc2 = sparks2.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ +
    _)

wc1.join(wc2).collect.foreach(println)
```

Developing Spark algorithms: Why?

- Why not Hadoop or <Insert another framework here>?
  - Richer programming model
  - Not just Map and Reduce (and Combine)
  - Speed and In-Memory computation
- With Hadoop/Map-Reduce, it is difficult to represent complex algorithms

Spark Algorithms: Use Cases

- Analytics and Statistics, Data Mining, Machine Learning, …
  - Pattern recognition, anomaly detection (spam, malware, fraud)
  - Identification of key or popular topics
  - Content classification and clustering, recommender systems
- Large-scale, Scalable Systems
- More Efficient Parallel Algorithms
  - You don't need to implement the parallelism every time
- Cost Optimization, Flexibility – Cloud instead of Grid

Data mining techniques

- Classification, Clustering
- Collaborative filtering
- Dimensionality reduction
- Frequent pattern mining
- Regression, Anomaly detection
- Supervised learning, Feature learning, Online learning
- Topic models
- Unsupervised learning
- All of this can probably be done with Spark, but **may require case-by-case algorithm redesign**
  - Why? **Ability to process very large datasets**

Developing Spark Algorithms

- Do not duplicate work
- Check if it exists in MLlib
- Is there a Spark Package for it? http://spark-packages.org/
- Has someone made it for Hadoop?
- Can you use Apache math3 or some other lib in a parallel way?
- If no, then…
  - Find a pseudocode or the math
  - Think it through in a parallel way
    - **The hard part**

The Hard Part: Global State 1/2

- Check for immutable global data structures
- and replace them with Broadcasts in Spark

```
int[] supportData = {0, 1, 2, 3, 4}  →

val supportArray = Array(0, 1, 2, 3, 4)

val supportData = sc.broadcast(supportArray)
```

The Hard Part: Global State 2/2

- Check for mutable global data structures
  - Change them to Broadcasts, and only change their content after a transformation/action is complete

```
int[] mutableData = {0, 1, 2, 3, 4}  →

var mut = sc.broadcast(Array(0, 1, 2, 3, 4))

val updated:Array[(Int, Int)] = dataRdd.map{x =>

  x%5 -> mut.value(x%5)*x
  /* which index was updated, and what is the new value */

}.reduceByKey(_ + _).collect.sortBy(_._1)

mut = sc.broadcast(updated.map(_._2))
```

- And then loop again ...

Dealing with sliding windows

```
int[] data = input;
int[] result = new int[data.length];
for (int i = 1; i < data.length; i++){
  result[i-1] = data[i] * data[i-1];
  }
```

- The above takes pairs starting from 0, 1
- In Spark we can use zip to do this

Dealing with sliding windows

```
val data0 = 12
val rdd = sc.textFile("input")
val paired:RDD[(Int, Int)] =
    rdd.dropRight(1).zip(rdd.drop(1))
paired.map(p =>
    p._1*p._2).saveAsTextFile("result")
```

- If we also need ordering, we need to
  - Have line numbers in the original file, or
  - Use `rdd.zipPartitions(rdd2).mapPartitionsWithIndex(...)`

Dealing with ordering

```
val ord = rdd.mapPartitionsWithIndex{part =>
  val (idx:Int, items:Iterable[(Int, Int)]) = part
  items.toArray.zipWithIndex.map{p =>
    val ((p1, p2), index) = p
    (idx, index) -> p._1*p._2 }}
```

Now items are prefixed with file part and their position in that part, e.g. $(0,\ 0),\ (0,\ 1),\ (0,\ 2),\ \cdots\ (1,\ 0),\ (1,\ 1),\ (1,\ 2),\ \cdots$

We can sort by this and save the results to preserve order

```
ord.sortBy(_._1).map(_._2)
  .saveAsTextFile("results")
```

The Hard Part: Interdependent loops

- Check for state dependencies in loops

```
int[] data = new int[n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i; j++)
    data[i] += i * data[j];
  }
```

- This depends on all the previously calculated values
  - It will be very hard to convert this kind of algorithm to Spark, perhaps find another approach
  - Or figure it out case-by-case...

The Hard Part: Figuring it out

```
int[] data = new int[n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i; j++)
    data[i] += i * data[j];
  }
```

- The above just does `i * sum(0 until i)` (i is not included)
- We can do that in Spark easily:

```
val rdd = sc.parallelize(0 until n)
val finalData = rdd.map(i => i * (0 until i).sum)
```

- So, even interdependent loops can be converted,
  if you figure them out

Spark API

https://spark.apache.org/docs/latest/api/scala/index.html

For Python

https://spark.apache.org/docs/latest/api/python/

Spark Programming Guide:

https://spark.apache.org/docs/latest/programming-guide.html

Check which version's documentation (stackoverflow, blogs, etc) you are looking at, the API had big changes after version 1.0.0, and since version 1.6.0, you no longer need to set storageFraction. Also, choice of master now happens via spark-submit, and some memory-related properties have been renamed.

Intro to Apache Spark: http://databricks.com

More information

These slides:

https://is.gd/bigdataalgo2017

https://is.gd/bigdataspark2017

Contact:

These slides:

Eemil Lagerspetz, Eemil.lagerspetz@cs.helsinki.fi

Big Data Frameworks course:

Mohammad Hoque, mohammad.hoque@cs.helsinki.fi

Course page:

https://www.cs.helsinki.fi/en/courses/582740/2017/k/k/1