Big Data Frameworks: Scala and Spark Tutorial

13.03.2015

Eemil Lagerspetz, Ella Peltonen

Professor Sasu Tarkoma

These slides: http://is.gd/bigdatascala

Functional Programming

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

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(" ")).toIterable
val counts = words.groupBy(identity).map(words =>
    words._1 -> words._2.size)
val top10 = counts.toArray.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.simplyscala.com

A Scala Tutorial for Java Programmers

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

```
    val parsedRDD = rdd.map(_.split(';'))
                    val parsedRDD: spark.RDD[Array[String]]
  }
}
```

part-00000

```
Matti;45;2010
Matti;70;2011
Matti;124;2012
Matti;181;2013
Eija;20;2010
Eija;62;2012
Eija;101;2011
Eija;140;2013
Kaarlo;55;2010
Kaarlo;111;2011
Kaarlo;189;2012
Kaarlo;202;2013
Tuomas;56;2010
Tuomas;112;2011
Tuomas;140;2012
Tuomas;201;2013
Saija;60;2010
```

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:

```
for (i <- 0 until 10) { // with 0 to 10, 10 is included
  println(s"Item: $i")
}
```

Examples:
import scala.collection.immutable._

lsts.filter(v=>v.length>2) is the same as lsts.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: lsts.map(x => x * 4)

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

flatMap: lsts.flatMap(_.toList) uses the given function to create a new list, then places the resulting list elements at the top level of the collection

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,’<‘)._2

Notice above: single-statement functions do not need curly braces { }

- 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 (you need to assign the return value)
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

MapReduce and Spark build on this parallelism

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

Apache Spark

Spark is a general-purpose computing framework for iterative tasks

API is provided for Java, Scala and Python

The model is based on MapReduce enhanced with new operations and an engine that supports execution graphs

Tools include Spark SQL, MLLlib for machine learning, GraphX for graph processing and Spark Streaming

Obtaining Spark

Spark can be obtained from the spark.apache.org site

Spark packages are available for many different HDFS versions

Spark runs on Windows and UNIX-like systems such as Linux and MacOS

The easiest setup is local, but the real power of the system comes from distributed operation

Spark runs on Java6+, Python 2.6+, Scala 2.1+

Newest version works best with Java7+, Scala 2.10.4

Installing Spark

We use Spark 1.2.1 or newer on this course

For local installation:

Download http://is.gd/spark121

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 the IDE, take the assembly jar from spark-1.2.1/assembly/target/scala-2.10 OR spark-1.2.1/lib

You need to have

- Java 6+
- For pySpark: Python 2.6+

For Cluster installations

Each machine will need Spark in the same folder, and key-based passwordless SSH access from the master for the user running Spark

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 sc

A Spark program creates a SparkContext object, denoted by the sc variable in Scala and Python shell

Outside shell, a constructor is used to instantiate a SparkContext

```
val conf = new SparkConf().setAppName("Hello").setMaster("local[2]")
val sc = new SparkContext(conf)
```

SparkContext is used to interact with the Spark cluster

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

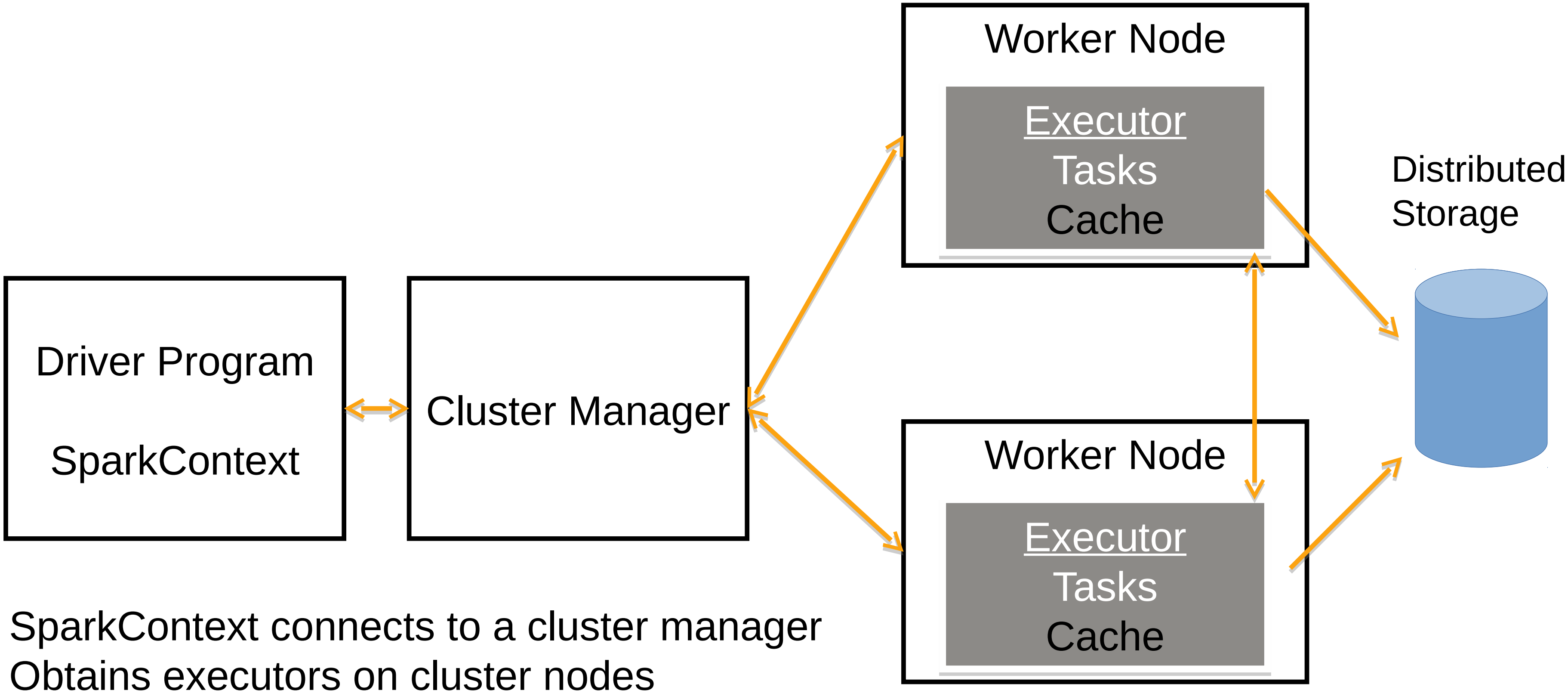
local[K]
- local with K worker threads

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

Spark overview



SparkContext connects to a cluster manager
Obtains executors on cluster nodes
Sends app code to them
Sends task to the executors

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()
```

WordCounting

```
/* When giving Spark file paths, those files need to be accessible
   with the same path from all slaves */

val file = sc.textFile("README.md")

val wc = file.flatMap(l => l.split(" "))

  .map(word => (word, 1))

  .reduceByKey(_ + _)


wc.saveAsTextFile("wc_out.txt")

wc.collect.foreach(println)
```

Join

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

val f2 = sc.textFile("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)
```

Transformations

Create a new dataset from an existing dataset

All transformations are lazy and computed when the results are needed

Transformation history is retained in RDDs

- calculations can be optimized
- data can be recovered

Some operations can be given the **number of tasks**. This can be very important for performance. Spark and Hadoop prefer larger files and smaller number of tasks if the data is small. However, the number of tasks should always be **at least the number of CPU cores** in the computer / cluster running Spark.

Spark Transformations I/IV

Transformation	**Description**
map(**func**)	Returns a new RDD based on applying function **func** to the each element of the source
filter(**func**)	Returns a new RDD based on selecting elements of the source for which **func** is true
flatMap(**func**)	Returns a new RDD based on applying function **func** to each element of the source while **func** can return a sequence of items for each input element
mapPartitions(**func**)	Implements similar functionality to map, but is executed separately on each partition of the RDD. The function **func** must be of the type (Iterator <T>) => Iterator<U> when dealing with RDD type of T.
mapPartitionsWithIndx(**func**)	Similar to the above transformation, but includes an integer index of the partition with **func**. The function **func** must be of the type (Int, Iterator <T>) => Iterator<U> when dealing with RDD type of T.

Transformations II/IV

Transformation	Description
sample(**withReplac, frac**, **seed**)	Samples a fraction (**frac**) of the source data with or without replacement (**withReplac**) based on the given random **seed**
union(**other**)	Returns an union of the source dataset and the given dataset
intersection(**other**)	Returns elements common to both RDDs
distinct([**nTasks**])	Returns a new RDD that contains the distinct elements of the source dataset.

Spark Transformations III/IV

Transformation	Description
groupByKey([**numTask**])	Returns an RDD of (K, Seq[V]) pairs for a source dataset with (K,V) pairs.
reduceByKey(**func**, [**numTasks**])	Returns an RDD of (K,V) pairs for an (K,V) input dataset, in which the values for each key are combined using the given reduce function **func.**
aggregateByKey(**zeroVal**)(**seqOp**, **comboOp**, [**numTask**])	Given an RDD of (K,V) pairs, this transformation returns an RDD RDD of (K,U) pairs for which the values for each key are combined using the given combine functions and a neutral zero value.
sortByKey([**ascending**], [**numTasks**])	Returns an RDD of (K,V) pairs for an (K,V) input dataset where K implements *Ordered*, in which the keys are sorted in ascending or descending order (**ascending** boolean input variable).
join(**inputdataset**, [**numTask**])	Given datasets of type (K,V) and (K, W) returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(**inputdataset**, [**numTask**])	Given datasets of type (K,V) and (K, W) returns a dataset of (K, Seq[V], Seq[W]) tuples.
cartesian(**inputdataset**)	Given datasets of types T and U, returns a combined dataset of (T, U) pairs that includes all pairs of elements.

Spark Transformations IV

Transformation	Description
pipe(**command**, [**envVars**])	Pipes each partition of the given RDD through a shell command (for example bash script). Elements of the RDD are written to the stdin of the process and lines output to the stdout are returned as an RDD of strings.
coalesce(**numPartitions**)	Reduces the number of partitions in the RDD to **numPartitions.**
repartition(**numPartitions**)	Facilitates the increasing or reducing the number of partitions in an RDD. Implements this by reshuffling data in a random manner for balancing.
repartitionAndSortWithinPartitions(**partitioner**)	Repartitions given RDD with the given partitioner sorts the elements by their keys. This transformation is more efficient than first repartitioning and then sorting.

Spark Actions I/II

Transformation	Description
reduce(**func**)	Combine the elements of the input RDD with the given function **func** that takes two arguments and returns one. The function should be commutative and associative for correct parallel execution.
collect()	Returns all the elements of the source RDD as an array for the driver program.
count()	Returns the number of elements in the source RDD.
first()	Returns the first element of the RDD. (Same as take(1))
take(**n**)	Returns an array with the first n elements of the RDD. Currently executed by the driver program (not parallel).
takeSample(**withReplac, frac, seed)**	Returns an array with a random sample of **frac** elements of the RDD. The sampling is done with or without replacement (**withReplac**) using the given random **seed**.
takeOrdered(**n**, [**ordering**])	Returns first n elements of the RDD using natural/custom ordering.

Spark Actions II

Transformation	**Description**
saveAsTextFile(**path**)	Saves the elements of the RDD as a text file to a given local/HDFS/Hadoop directory. The system uses toString on each element to save the RDD.
saveAsSequenceFile(**path**)	Saves the elements of an RDD as a Hadoop SequenceFile to a given local/HDFS/Hadoop directory. Only elements that conform to the Hadoop *Writable* interface are supported.
saveAsObjectFile(**path**)	Saves the elements of the RDD using Java serialization. The file can be loaded with SparkContext.objectFile().
countByKey()	Returns (K, Int) pairs with the count of each key
foreach(**func**)	Applies the given function **func** for each element of the RDD.

Spark API

https://spark.apache.org/docs/1.2.1/api/scala/index.html

For Python

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

Spark Programming Guide:

https://spark.apache.org/docs/1.2.1/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.

More information

These slides: http://is.gd/bigdatascala

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

Project that can be used to start (If using Maven):

https://github.com/Kauhsa/spark-code-camp-example-project

This is for Spark 1.0.2, so change the version in pom.xml.