Lesson 7

Spark and SparkSQL:

Commanding All the Laws of Nature from the Source

Wholeness of the Lesson

The declarative style of functional programming makes it possible to write methods (and programs) just by declaring *what* is needed, without specifying the details of *how* to achieve the goal. Including support for functional programming in Scala makes it possible to write parts of Scala programs more concisely, in a more readable way, in a more threadsafe way, in a more parallelizable way, and in a more maintainable way, than ever before.

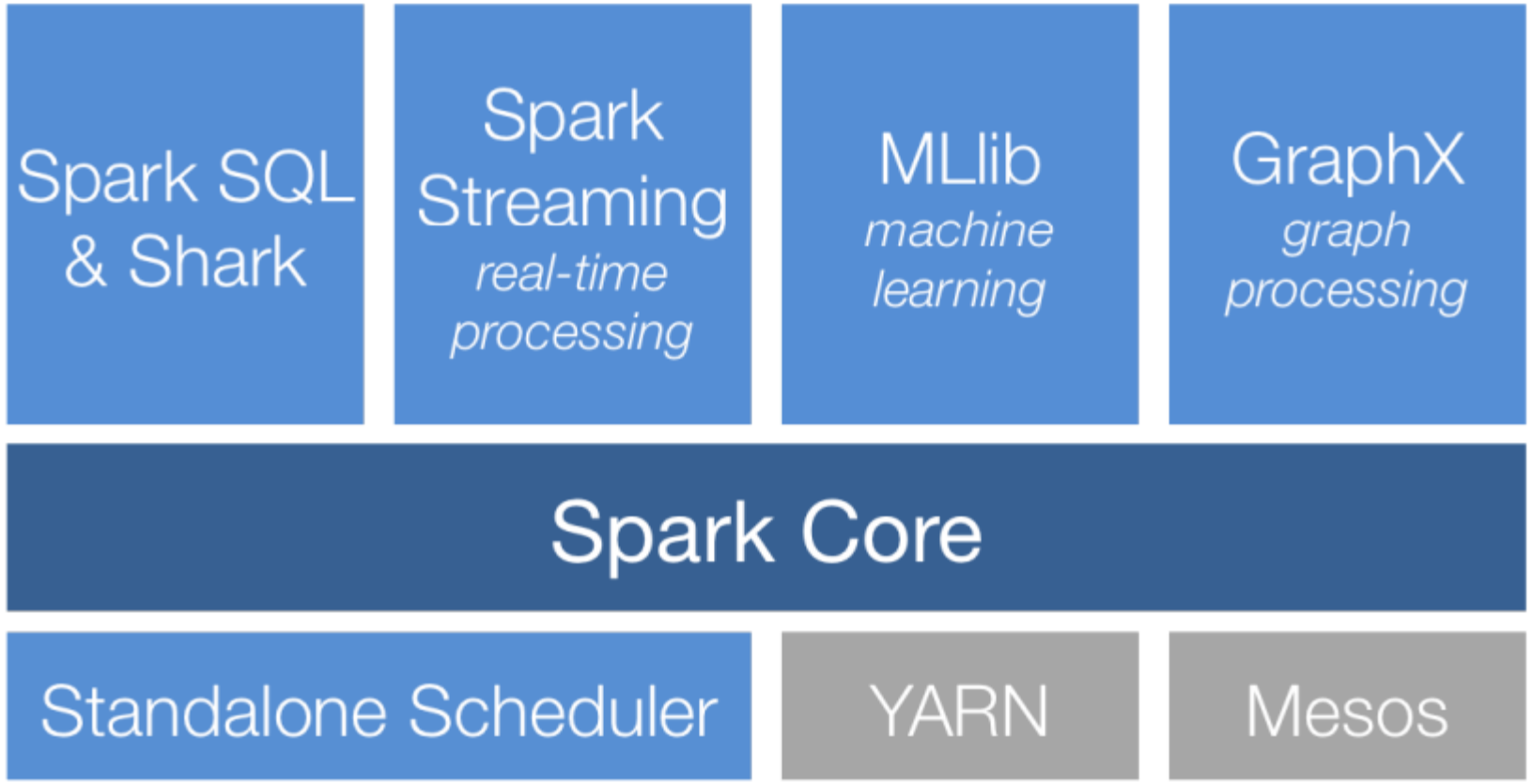
Just as a king can simply declare what he wants – a banquet, a conference, a meeting of all ministers – without having to specify the details about how to organize such events, so likewise can one who is awake to the home of all the laws of nature, the "king" among laws of nature, command those laws and thereby fulfill any intention. The royal road to success in life is to bring awareness to the home of all the laws of nature, through the process of transcending, and live life established in this field.

Spark Introduction

- Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.
- You can also run Spark interactively through a modified version of the Scala shell. This is a great way to learn the framework.

```
/usr/bin/spark-shell
```

Spark Introduction



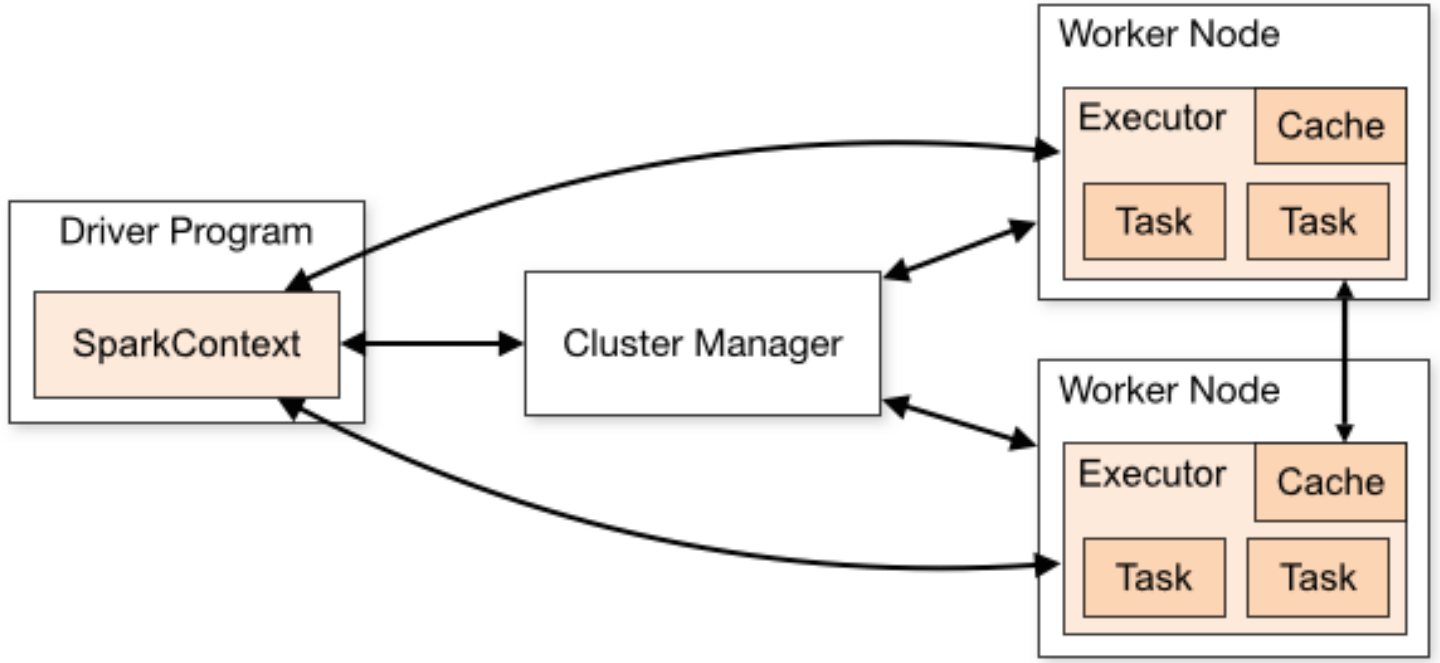
Spark Cluster Mode

- Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called thedriver program).
- **Driver program**: the process running the main() function of the application and creating the SparkContext
- To run on a cluster, the SparkContext can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications.

Spark Cluster Mode

- Spark connects to a *cluster manager* which allocate resources across applications
- Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application.

- Next, it sends your application code to the executors.
- Finally, SparkContext sends *tasks* to the executors to run.



RDD

- Resilient Distributed Datasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel
  - Resilient – fault tolerant and Immutable.
  - Distributed – Is spilt into multiple partitions (which may be computed on different nodes of the cluster)
  - Dataset – giant set of data, row after row of information. Can contain any type of Python, Java or Scala objects, including user-defined classes.

RDD

There are currently two types of RDD.

- parallelized collections – take an existing Scala collection and run functions on it in parallel. (User can create an RDD by distributing a collection of objects such as a list or a set.)
- Hadoop datasets – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop. (User can create an RDD by loading an external dataset.)

RDD

- Making RDD from parallelized collections:

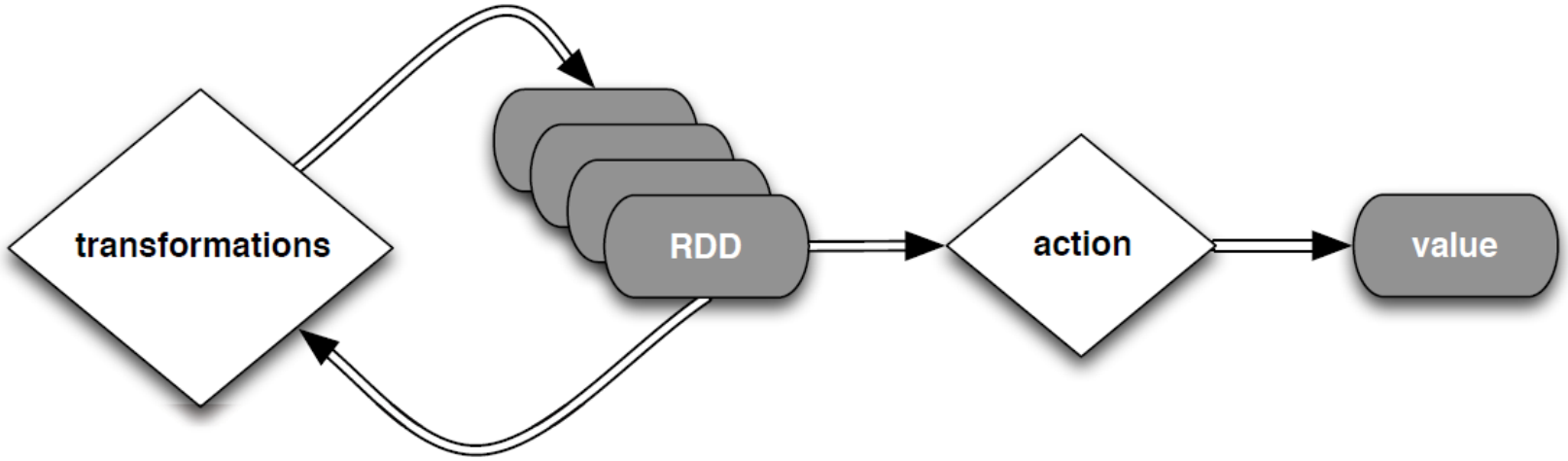
```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)
scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

- Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, Elasticsearch, JSON, CSV, Cassandra, JDBC and so on.

```
scala> val lines = sc.textFile("README.md")!
lines: spark.RDD[String] = spark.HadoopRDD@1d4cee08
```

RDD

- Two types of operations on RDDs: *transformations* and *actions*
- Transformations are lazy (not computed immediately)
- The transformed RDD gets recomputed when an action is run on it (default)
- An RDD can be *persisted* into storage in memory or disk



RDD - Transformation

- Transformations create a new dataset from an existing one

```
scala> val lines = sc.textFile("README.md")!
distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
scala>val  scalaLines = lines.filter(w => w.contains(“Scala”))
scalaLines: spark.RDD[String] = FilteredRDD[…]
```

- All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset for:
  - optimize the required calculations
  - recover from lost data partitions

RDD - Actions

- Compute a result based on an RDD
- Return it to either the driver program or save it to an external storage system (e.g., HDFS)

Word count

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1)).reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Is reduceByKey a transformation or an action?

RDD - Actions

Word count

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

reduceByKey an transformation. Thus counts is an RDD.

RDD – Common Transformations and Actions

Element-wise transformations

map

```
val input = sc.parallelize(List(7, 5, 2, 9))
val result = input.map(x => x * x)
println(result.collect().mkString(", "))   // 49, 25, 4, 81
```

Note: result.foreach(println) will print each number in a separate line and result.foreach(print) will print 4925481

RDD – Common Transformations and Actions

Element-wise transformations

flatMap. The function must return a list;

```
val lines = sc.parallelize(List("Once upon a time “ , “Spark came “))
val words = lines.flatMap(w => w.split(“, “))
words.foreach(println)
var numbers = sc.parallelize(List(7, 2, 9))
var numberList = numbers.flatMap(x => List(x * x, x * x * x))
println(numberList.collect().mkString(", "))   // 49, 343, 4, 8, 81, 729
var numberList1 = numbers.flatMap(x => List(x * x))      // Correct
var numberList 2= numbers.flatMap(x => x * x)                    // Error
```

flatMap “flattens”. Thus instead of an RDD of lists, we get an RDD of all elements in all those lists.

RDD – Common Transformations and Actions

Element-wise transformations

filter

```
scala> val one = sc.parallelize(List(1, 2, 3, 4))
scala> val two = one.filter(x => x > 2)
println(two.collect().mkString(", "))   // 3, 4
```

sample

```
scala> val three = one.sample(false, 0.5)
println(three.collect().mkString(", "))  // 2, 3
```

Does not always return the correct number o items
Alternative solution
val three: Array[Int] =
one.takeSample(withReplacement, number)

RDD – Common Transformations and Actions

Pseudo set operations

union, intersection, subtract and distinct. distinct is costly. use it only when necessary.

```
scala> val one = sc.parallelize(List(1, 2, 3, 4))

scala> val two = sc.parallelize(List(6, 5, 4, 3))

scala> val one_union_two = one union two

println(one_union_two.distinct().collect().mkString(", "))  // 4, 1, 5, 6, 2, 3

OR println(one.union(two).distinct().collect().mkString(", "))  // 4, 1, 5, 6, 2, 3

println(one.intersection(two).collect().mkString(", "))  // 4, 3

println(one.subtract(two).collect().mkString(", "))  // 2, 1

println(one.cartesian(two).collect().mkString(", "))    // see result below.

(1,6), (1,5), (2,6), (2,5), (1,4), (1,3), (2,4), (2,3), (3,6), (3,5), (4,6), (4,5), (3,4), (3,3), (4,4), (4,3)
```

RDD - Transformation

Transformation	description
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
mapValues(func)	Used with a paired RDD. Similar to map. The func is applied to the values only. The keys are left as it is.
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement***,** *fraction***,** *seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

RDD - Transformation

Transformation	description
groupByKey([*numTasks***])**	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func***, [***numTasks***])**	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending***], [***numTasks***])**	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset***, [***numTasks***])**	when called on 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(*otherDataset***, [***numTasks***])**	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset***)**	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

map Vs mapValues

```
val rdd1 = sc.parallelize(List((“Mech", 30), (“Mech", 40), (“Elec", 50)))
val valmapped = rdd1.mapValues(mark => (mark, 1));

val reduced
        = valmapped.reduceByKey((x, y) => (x._1 + y._1,  x._2 + y._2))
reduced.collect()
```

Array[(String, (Int, Int))] = Array((Elec, (50,1)), (Mech,(70,2)))

Array[(String, (Int, Int))] = Array((Elec, (50,1)), (Mech,(70,2)))

```
val average = reduced.map { x =>
                    val temp = x._2
                    val total = temp._1
                    val count = temp._2
                    (x._1, total / count)
                    }
average.collect()
```

Array[(String, Int)] = Array((Elec, 50), (Mech, 35))

Main Point 1

In Spark, RDD is immutable. A transformation will not change the existing RDD. Rather it creates a new RDD through lazy evaluation. *Self is immutable and no transformation can destroy it.*

Scala Closures

A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function but in the enclosing scope. Consider the following piece of code with anonymous function:

```
val multiplier = (i:Int) => i * 10
```

Here the only variable used in the function body, i * 10, is i, which is defined as a parameter to the function. Now let us take another piece of code:

```
val multiplier = (i:Int) => i * factor
```

There are two variables in multiplier: **i** and **factor**. One of them, i, is a formal parameter to the function. Hence, it is bound to a new value each time multiplier is called. However, **factor** is not a formal parameter, then what is this? It is a free variable.

Scala Closures

Let us add one more line of code:

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

Now, **factor** has a reference to a variable outside the function but in the enclosing scope. Let us try the following example:

```
object Test {
        def main(args: Array[String]) {
                println( "muliplier(1) value = " + multiplier(1) )
                println( "muliplier(2) value = " + multiplier(2) )
        }
        var factor = 3
        val multiplier = (i:Int) => i * factor
}
```

Scala Closures

When the code in the previous slide is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
muliplier(1) value = 3
muliplier(2) value = 6

C:/>
```

Above function references **factor** and reads its current value each time. If a function has no external references, then it is trivially closed over itself. No external context is required.

Action

reduce

```
scala> val numbers = sc.parallelize(List(7, 2, 9))
scala> val sum = numbers.reduce((x, y) => x + y)
scala> val squareSum = numbers.map(x => x*x).reduce((x, y) => x + y)
```

Problem : Given a list determine whether there are more evens or odds

Solution

```
scala> val numbers = sc.parallelize(List(7, 8, 9, 10, 12)).map(x => if (x %2 == 0) 1 else -1).reduce(_+_)
```

OR

```
scala> val numbers = sc.parallelize(List(7, 8, 9, 10, 12)).map(x => if (x %2 == 0) 1 else -1).reduce((x, y) => x + y)
```

RDD - Action

Transformation	description
reduce(*func***)**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n***)**	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement*, *fraction*, *seed***)**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

RDD - Action

Transformation	description
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
countByKey()	only available on RDDs of type (K, V). Returns a `Map` of (K, Int) pairs with the count of each key
countByValue()	only available on RDDs of type (K, V). Returns a `Map` of (V, Int) pairs with the count of each value
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD - Action

```
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect().foreach(println)
```

What is collect()? Is a transformation or an action?

Can we “chain” actions?

RDD - Action

```
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split("\\W+")).map(word => (word, 1))
words.reduceByKey(_ + _).collect().foreach(println)
```

collect() is an action.

We cannot “chain” actions.

In the above example foreach is not a Spark action. It is part of Scala.

Passing Functions in Scala

Most of Spark's transformations, and some of its actions, depend on passing in functions that are used by Spark to compute data.

In scala, we can pass in

- functions defined inline,
- references to methods,
- static functions

Passing Functions in Scala

Function passed by reference:

```
// A function that splits a line returns (firstname, age) tuples
 def parseLine(line: String) = {
    val fields = line.split(",")
    val age = fields(2).toInt
    val name = fields(1).toString
    (name,age)
 }
  val lines = sc.textFile("../people.csv") //CSV format: ID, Firstname, Age, #Friends
  val rdd = lines.map(parseLine)
```

(Review slides 20 and 21 if you cannot follow next two lines of code)

```
  val total = rdd.mapValues(x=>(x, 1)).reduceByKey(
                                  (x,y)=>(x._1 + y._1, x._2 + y._2)).mapValues(x=>x._1/x._2)
  val result = total.collect().sorted.foreach(println)
```

Passing Functions in Scala

Function defined inline:

```
val total = rdd.mapValues(x=>(x, 1)).reduceByKey(
                (x,y)=>(x._1 + y._1, x._2 + y._2)).mapValues(x=>x._1/x._2)
```

Static function passed:

```
val result = total.collect().sorted.foreach(println)
```

RDD - Persistence

```
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

For more details about the Scala/Java API:

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package

RDD - Persistence

- Spark can *persist* (or cache) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

RDD - Persistence

Transformation	description
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, **MEMORY_AND_DISK_2**, etc	Same as the levels above, but replicate each partition on two cluster nodes.

SparkSQL in Spark 2.0

- DataSet: a set structure data //RDD can be converted to DataSet with .toDS()

The trend is to use DataSets more, RDDs less. //DataSet's performance is better

When using SparkSQL/DataSet, SparkSession object is used instead of SparkContext:

- SparkContext can be created from this session
- Stop the session when you're done
- DataFrame: //extending RDD, dataSet of row objects
  - ❑Contain row objects
  - ❑Can run SQL queries
  - ❑Has a schema
  - ❑read/write to JSON, Hive
  - ❑Communicates with JDBC/ODBC

Spark, SparkSQL, DataFrame, DataSet

Practice programs

- SparkCore.scala
- SparkDataFrame.scala
- SparkDataSets.scala

Using DataSet instead of RDD

```
// Use new SparkSession interface in Spark 2.0
val spark = SparkSession
    .builder.appName("PopularItems").master("local[*]").getOrCreate()
val lines = spark.sparkContext.textFile("../file.txt").map(x => (x.split("\t")))
// Convert to a DataSet
import spark.implicits._
val myDS = lines.toDS()
val topIDs =myDS.groupBy("myID").count().orderBy(desc("count")).cache()
topIDs.show()                    // the result format: |myID|count|
val top10 = topIDs.take(10) //show the top 10
spark.stop()                     //always stop the session when you are done
```

Using DataSet instead of RDD

Make the result human-readable:

```
//1. Create a Map of Ints to Strings, and populate it from u.item.
var myMap:Map[Int, String] = Map()
 val lines = Source.fromFile(“file1.txt").getLines()
 for (line <- lines) {
   var fields = line.split('|')
    myMap += (fields(0).toInt -> fields(1))
 }
//2. Result is just a Row; we need to cast it back. Say each row has ID & count.
for (result <- top10) {
  println (myMap(result(0).asInstanceOf[Int]) + ": " + result(1))
 }
```

Main Point 2

In functional programming, functions are first-class citizens – they are passed as arguments and occur as return values. *In ordinary human life, it is sometimes hard to recognize that one's mind does not function as well as it could; that one's emotions are rougher than they need to be; that one's priorities in life may not be as clear as they could be. Ordinary life is in this way an approximation to a truly full life. It is possible to purify one's inner life so that functioning in the world is smooth and successful, just as it is possible to tune a car engine or purify muddy water. An effective way to do this is to allow the mind to expand to its infinite nature and allow the body, at the same time, to rest deeply.*

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

1. The Spark employs two basic operations: Transformation and action.

2. An RDD is exclusively used by Spark to execute basic operations.

---

3. ***Transcendental consciousness*** *can be experienced in the stillness of one's awareness through transcending, is where the laws of nature begin to operate – it is the home of all the laws of nature.*

4. ***Impulses within the Transcendental Field:*** *As TC becomes more familiar, more and more, intentions and desires reach fulfillment effortlessly, because of the hidden support of the laws of nature.*

5. ***Wholeness moving within itself:*** *In Unity Consciousness, one finally recognizes the universe in oneself – that all of life is simply the impulse of one's own consciousness. In that state, one effortlessly commands the laws of nature for all good in the universe.*

References

- http://spark.apache.org/docs/latest/
- http://spark.apache.org/docs/latest/cluster-overview.html
- http://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf
- https://spark.apache.org/docs/2.1.0/sql-programming-guide.html