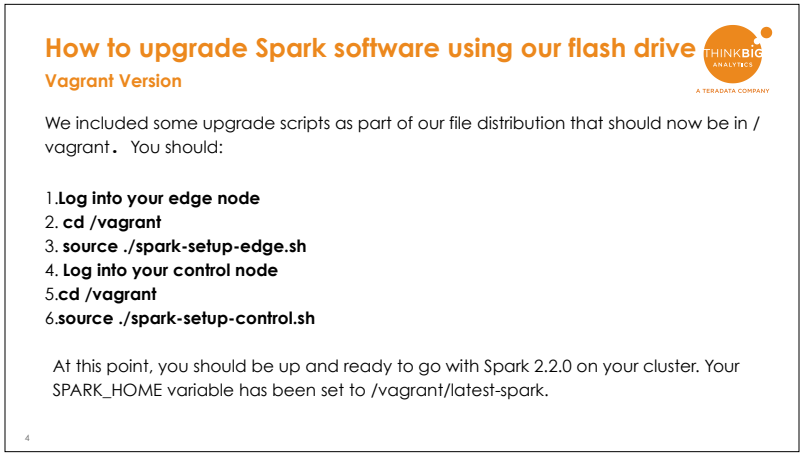
With preliminaries out of the way, let's dive into Spark in Scala. We choose Scala because it's the language Spark was written in. As a result, it poses the cleanest interface to the Spark environment.

What We'll Cover

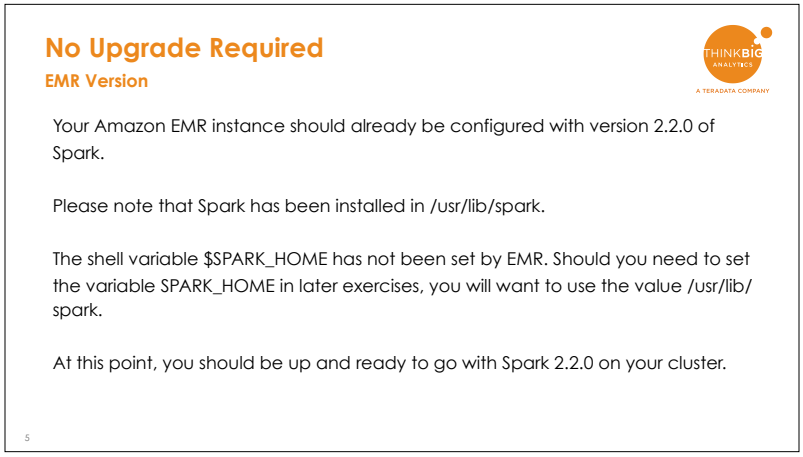
- The Spark execution model and Spark Sessions
- The Spark shell
- Scala immutabiles and mutables
- Resilient Distributed Datasets (RDDs)
- External datasets
- RDD operations: Transformations and Actions
- Function literals (aka closures or anonymous functions)
- Persistence
- Some Spark examples

We'll begin by covering these fundamental topics.

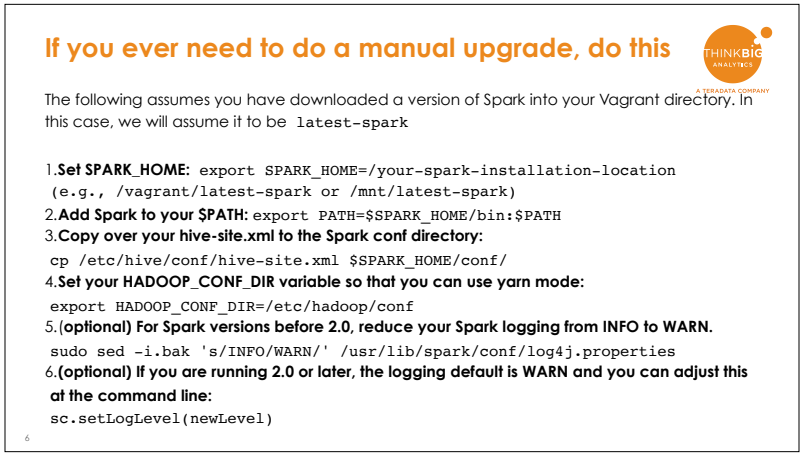
Before we begin, though, we should get you set up.



If you are using a local two-node VirtualBox cluster created using Vagrant, you'll want to execute a couple of scripts to get your environment set up with the latest version of Spark (Spark 2.2.0)



If you are using Amazon Web Services Elastic Map Reduce instances, your instance should be all set up to use Spark 2.2.0.



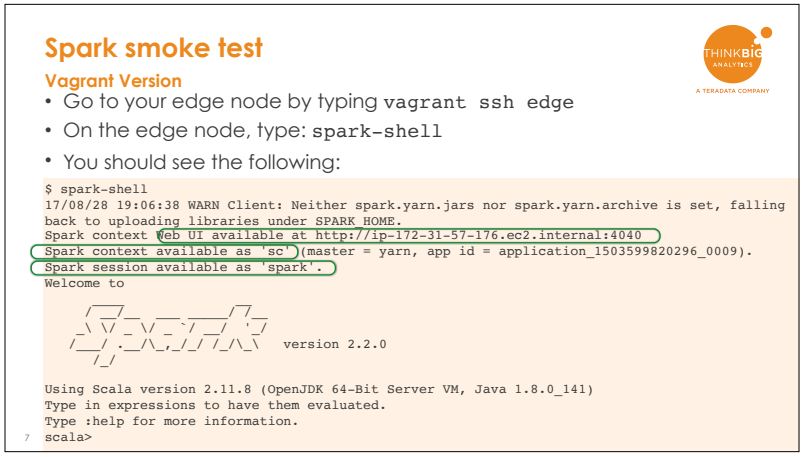
- You can always set up a new spark path and manually

One of the best features of Spark is that it is easy to switch to using a different version than the one installed on your cluster. The process really consists of only 4 essential steps:

1. Setting your SPARK_HOME shell variable to point at the directory where your Spark installation lives.
2. Adding $SPARK_HOME/bin to your execution path.
3. Copying over your hive-site.xml file to your Spark configuration directory
4. Setting the shell variable $HADOOP_CONF_DIR to point at your Hadoop configuration directory.

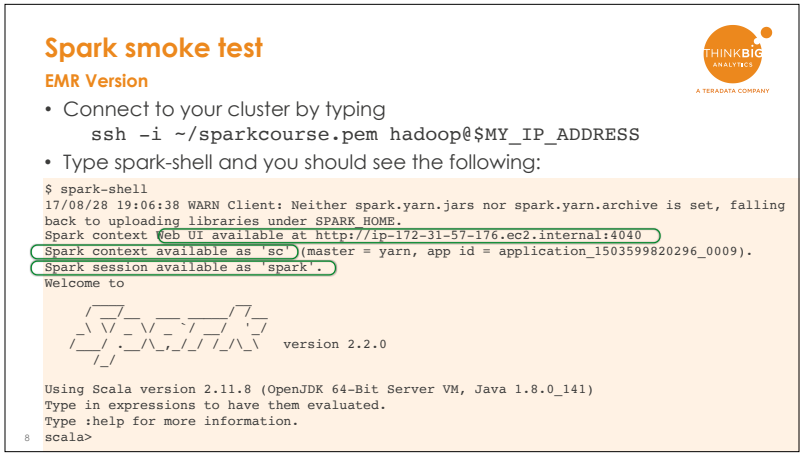
If you are using a Spark version prior to version 2, the default is for Spark to log all INFO messages to the console, which can definitely interfere with seeing what your program is doing. You can avoid this issue by setting your logging level to WARN by editing your log4j.properties file, as shown in step 5.

If you are running Spark 2.0 or later, the default logging level is WARN. You can change it to other levels, say ERROR, using a simple spark-shell command, sc.setLogLevel("ERROR").



If you're using vagrant, Go to your edge node by typing vagrant ssh edge. On the edge node, type: spark-shell. You should see the following.

Note that the spark-shell provides you with the Web address of the Spark user interface. It also has provided you with both a spark context (sc) and a spark session (spark). You'll need those to issue commands to the spark cluster.

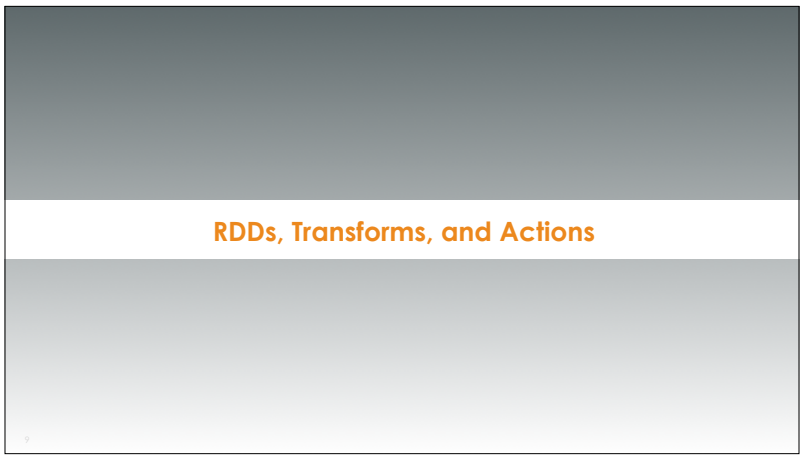


If you're using EMR, Go to your edge node by typing

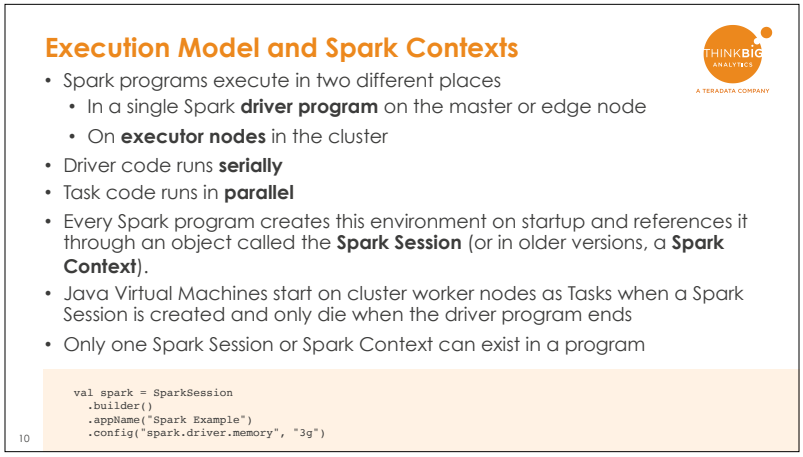
ssh -i ~/sparkcourse.pem hadoop@$MY_IP_ADDRESS

On your cluster node, type spark-shell. You should see the following.

Note that the spark-shell provides you with the Web address of the Spark user interface. It also has provided you with both a spark context (sc) and a spark session (spark). You'll need those to issue commands to the spark cluster.



With setup done, let's jump into the fundamental components of basic Spark: RDDs, transforms, and actions.



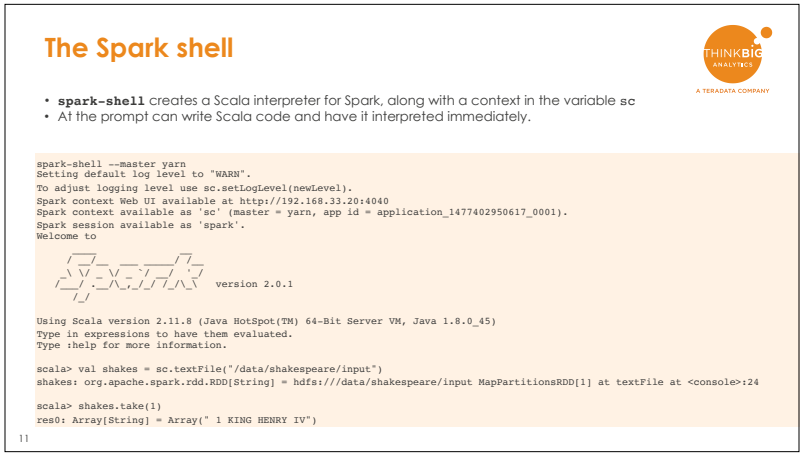
- This has you configure exactly how much memory for your spark shell if you are writing your own spark shell

Spark programs execute in two different places
1. In a single Spark driver program on the master or edge node
2. On executor nodes in the cluster
Driver code runs serially, while Task code runs in parallel. Every Spark program creates this environment on startup and references it through an object called the Spark Session (or in older versions, a Spark Context).
Java Virtual Machines start on cluster worker nodes as Tasks when a Spark Session is created and only die when the driver program ends
Only one Spark Session or Spark Context can exist in a program.

We can create a spark session using the following Scala chain:

```
val spark = SparkSession
    .builder()
    .appName("Spark Example")
    .config("spark.driver.memory", "3g")
```

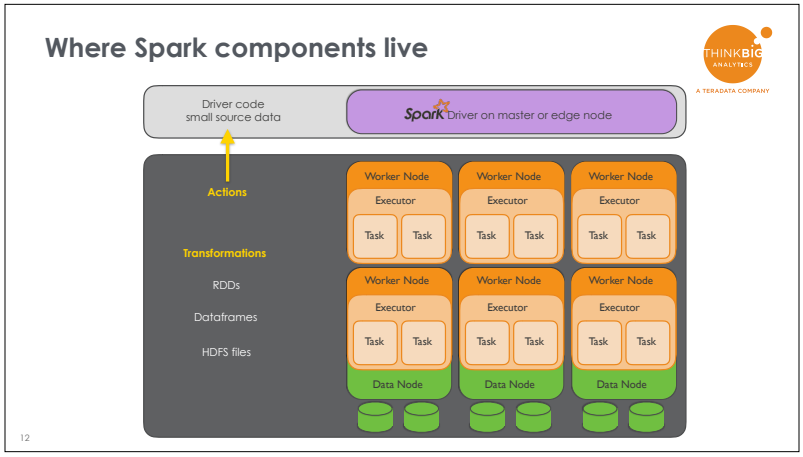
The builder method creates the spark session, appName then names that session, and the config method then sets various spark variables. In this case, we're setting the driver memory size to 3 gigabytes. We could similarly set the executor memory sizes using another config statement.



- take(1) gives you your first line
- http://ec2-54-221-189-104.compute-1.amazonaws.com:8080/
  - Works, but the 4040 does not work
  - 8088 works -- shows all applications
  - 5070 -- hdfs port

The spark shell is a REPL, or Read Evaluate Print Loop program. It reads what you type and then interprets it as Scala code.

The spark-shell is one of the biggest differences between the Spark Scala programming model and a traditional Java one. Scala has a REPL to ease development; at the time of this writing Java has none (although one is promised in Java 9).

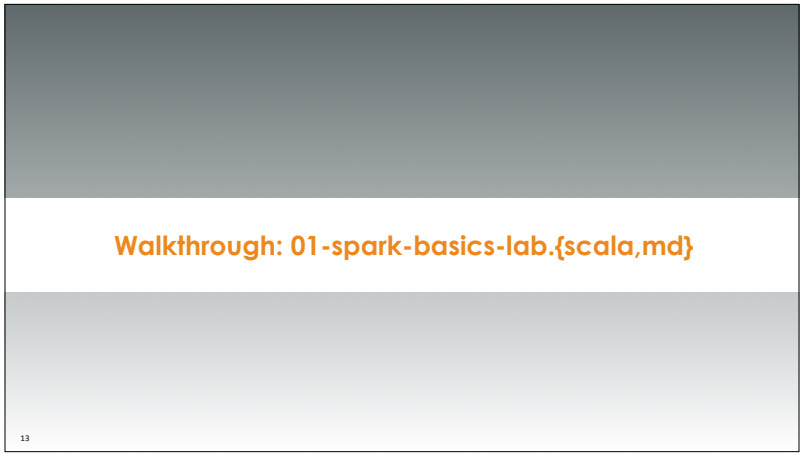


If we look at this visually, the light gray box represents the edge node, depending on your cluster configuration. That's where the Spark driver lives. It runs serially there on one node.

In the dark gray box, we have the rest of the cluster data and worker nodes. That's where HDFS files are distributed over the disks connected to the data nodes. It's also where Spark DataFrames and RDDs exist.

Transformations are Spark methods that execute in parallel on the cluster. They run in parallel.

Actions are Spark methods that take data from the cluster and bring that data back to the driver. Because they must bring the data back to the driver, they force all the nodes involved to synchronize their efforts. They create a momentary serialization.



- Location of these items
- /mnt/sparkclass/exercises/spark-in-scala

Let's go hands-on with Spark with our first Scala lab. Two versions exist: one in Markdown and one in pure Scala code. We recommend you bring up one of those versions in a regular text editor (i.e., Notepad on Windows PCs, TextEdit or your favorite development text editor on a Mac) to view the text while having your terminal window along side it for typing commands.

Hands-On Lab: First Scala Program: Lab 02-spark-first-program.md

We saw in our walkthrough that we count the number of stock quotes for AAPL using code that looks like this:

```
val rdd = sc.textFile("hdfs:///data/stocks-flat/input")
val aapl = rdd.filter( line => line.contains("AAPL") )
aapl.count
```

Try using that pattern to count the number of lines in the file `/data/shakespeare/input` that contain the word "king". You'll want to use the `toLowerCase` function before the contains function to ensure all the text is lower case.

A solution is on the next slide or in the lab script

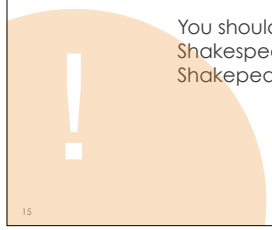
14

- Parallelize RDD for Hadoop
- rdd.first is like the take(1) function
-

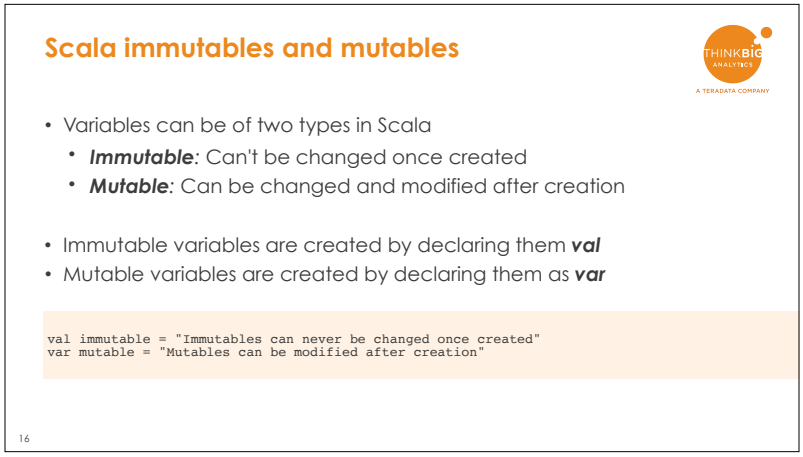
Hands-On Lab: First Scala Program

One solution looks like this

```
val rdd = sc.textFile("hdfs:///data/shakespeare/input")
val kings = rdd.filter( line => line.toLowerCase().contains("king"))
kings.count
```



You should have gotten 4773 lines that reference a king in all of Shakespeare. That's almost 3% of the 175,376 lines in all of Shakepeare's plays and poems.



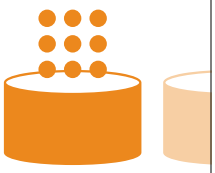
Variables can be of two types in Scala
Immutable: Can't be changed once created
Mutable: Can be changed and modified after creation

This is probably the most profound addition to cluster programming in the 21st century. By explicitly allowing the programmer to tell Scala which variables are immutable, Scala knows it can copy those variables throughout the cluster safely without affecting the validity of the computation. That means frequently used variables are likely to be local to a node and avoids excessive network traffic fetching variables. This is one of the fundamental insights within Spark: Immutables allow cluster programs to run faster because of greater data locality.

Immutable variables are created by declaring them **val**
Mutable variables are created by declaring them as **var**

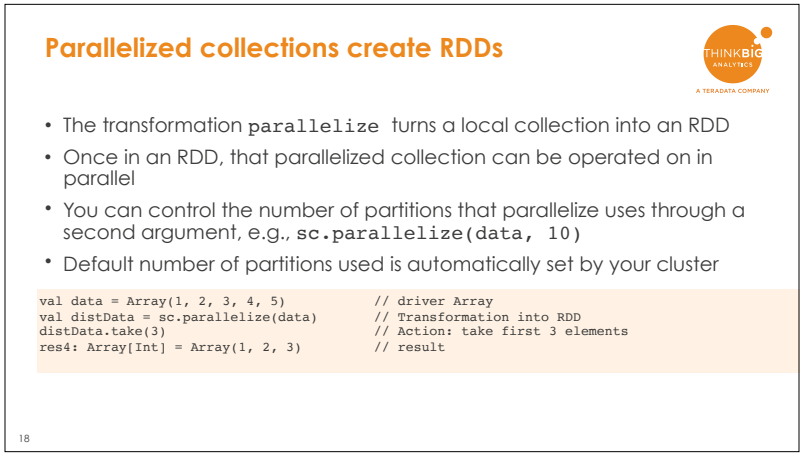
Resilient Distributed Datasets (RDDs) are the primary parallel data structure

- Data is represented as **Resilient Distributed Datasets (*"RDDs"*)**
  - Can be created on the fly in the driver (*e.g.,* `parallelize()`)
  - Could be a file on disk, HDFS path, result set, etc.
  - RDDs are immutable and cannot be modified once created
- New RDDs can be created from existing ones via ***Transformations*** which are executed ***lazily***
- ***Actions*** compute a result based on an RDD. The result can be returned to the driver environment or saved to disk or elsewhere

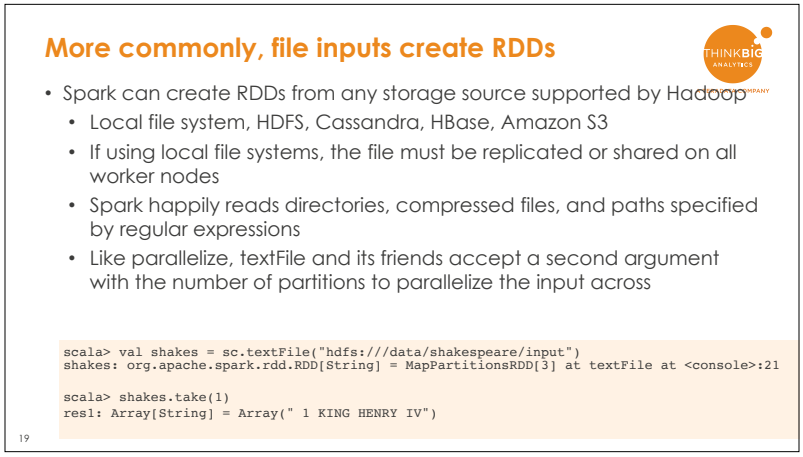


17

Data is represented as Resilient Distributed Datasets (“RDDs”)
Can be created on the fly in the driver (e.g., parallelize())
Could be a file on disk, HDFS path, result set, etc.
RDDs are immutable and cannot be modified once created
New RDDs can be created from existing ones via Transformations which are executed lazily
Actions compute a result based on an RDD. The result can be returned to the driver environment or saved to disk or elsewhere



The transformation parallelize turns a local collection into an RDD
Once in an RDD, that parallelized collection can be operated on in parallel
You can control the number of partitions that parallelize uses through a second argument, e.g., sc.parallelize(data, 10)
Default number of partitions used is automatically set by your cluster



Spark can create RDDs from any storage source supported by Hadoop
Local file system, HDFS, Cassandra, HBase, Amazon S3
If using local file systems, the file must be replicated or shared on all worker nodes
Spark happily reads directories, compressed files, and paths specified by regular expressions
Like parallelize, textFile and its friends accept a second argument with the number of partitions to parallelize the input across

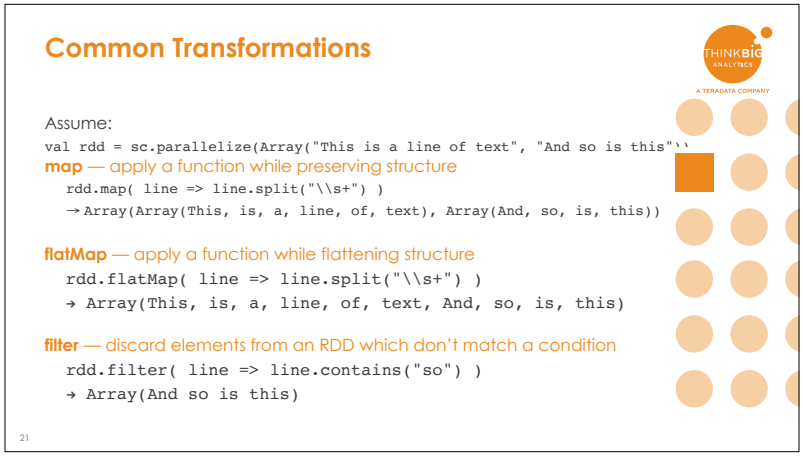
RDD Operations: Transformations and Actions

- **Transformations** convert an RDD into another RDD
- **Actions** often turn an RDD into something else
- Only **actions** cause evaluation

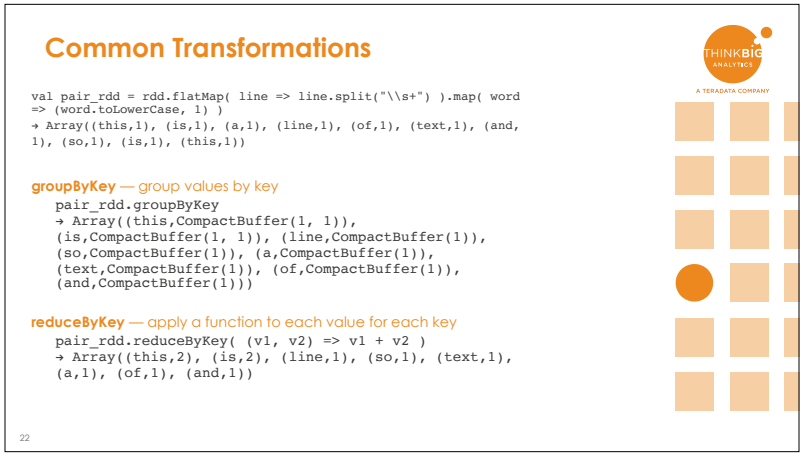
```
val lines = sc.textFile("hdfs:///data/shakespeare/input")      // Transformation
val lineLengths = lines.map(s => s.length)                     // Transformation
val totalLength = lineLengths.reduce((a, b) => a + b)          // Action
```

20

Transformations convert an RDD into another RDD
Actions often turn an RDD into something else
Only actions cause evaluation



- Filter - acts like the where in SQL
- map/flatMap



- Key value pairs -
-

More Transformations

For RDDs

map
filter
flatMap
mapPartitions
mapPartitionsWithIndex
sample
union
intersection
distinct
cartesian
pipe
coalesce

For PairRDDs:

groupByKey
reduceByKey
aggregateByKey
sortByKey
join
cogroup
keys
values
repartition
repartitionAndSortWithinPartitions

see http://spark.apache.org/docs/2.2.0/programming-guide.html#transformations

Common Spark Actions

- **collect** — gather results from nodes and return
- **first** — return the first element of the RDD
- **take(*N*)** — return the first N elements of the RDD
- **saveAsTextFile** — write the RDD as a text file
- **saveAsSequenceFile** — write the RDD as a SequenceFile
- **count** — count elements in the RDD
- **countByKey** — count elements in the RDD by key
- **foreach** — process each element of an RDD
  - (e.g., rdd.collect.foreach(println))

See http://spark.apache.org/docs/2.2.0/programming-guide.html#actions

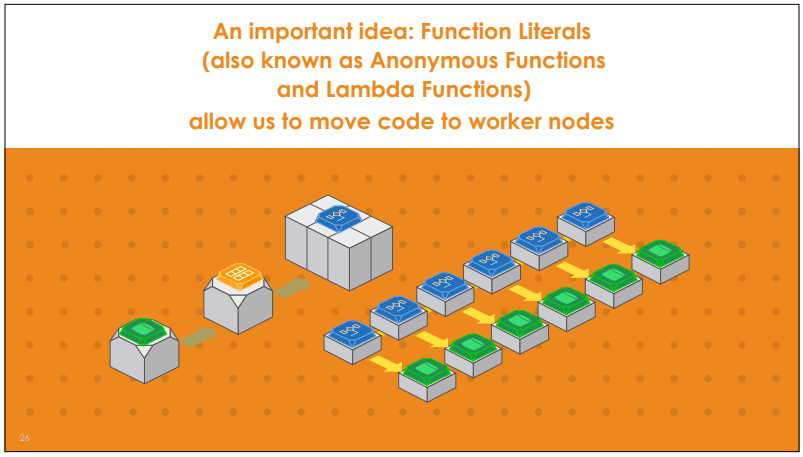
Spark Actions

reduce
collect
count
first
take
takeSample
takeOrdered

saveAsTextFile
saveAsSequenceFile
saveAsObjectFile
countByKey
foreach

See http://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#actions

•



- Anonymous functions vs. functional literals

Function definitions are pretty common in most languages

- A function definition lets us name a stored operation that takes typed arguments and returns a typed result
- Scala can infer result types and even argument types

```
def add(x:Int, y:Int):Int = {      // Takes two Int arguments and returns an Int
  return x + y
}
println(add(42,13))

// Implicit typing and return
def add(x:Int, y:Int) = {          //result type is inferred
  x + y                            //"return" keyword is optional
}

//Curly braces are optional on single line blocks
def add(x:Int, y:Int) = x + y
```

At least two ways exist to define functions and methods

- **=>** is a definition or *transformation* operator; it takes its left hand arguments and transforms them using the right hand expression
- Types can be explicit or implied

```
/ Many ways exist to define functions and methods. Here are several

val even = (i: Int) => { i % 2 == 0 } // explicit long form
val even: (Int) => Boolean = i => { i % 2 == 0 }
val even: Int => Boolean = i => { i % 2 == 0 }
val even: Int => Boolean = i => i % 2 == 0
val even: Int => Boolean = _ % 2 == 0 // _ means first argument

// implicit result approach
val add = (x: Int, y: Int) => { x + y }
val add = (x: Int, y: Int) => x + y

// explicit result approach
val add: (Int, Int) => Int = (x,y) => { x + y }
val add: (Int, Int) => Int = (x,y) => x + y
```

Anonymous functions allow us to do without the name

- ***Function literals*** (a term from Jason Schwartz who wrote the O'Reilly Learning Scala book) are functions defined and passed in-line without giving them a name
- Function literals also go by the names of ***anonymous functions*** *or* ***lambda functions***

```
// a function with a name greeting

val greeting = (x: String) => "Hello " + x
val names = List("Joe", "Mary", "Barbara")
names.map(greeting)

// should get
//     res2: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)

// Now let's get rid of the name greeting

names.map((x: String) => "Hello " + x)
// should get the same answer:
//    res2: List[String] = List(Hello Joe, Hello Mary, Hello Barbara)
```

29

- Mapping a greeting to each name

Scala provides shorthand representations

- => allows us to define a function quickly
- _ provides a quick argument reference by appearance
  - First _ means first argument, second _ means second argument

```
// Define maximize to compute the maximum of two integers using a function literal
// instead of def. Function literals are also referred to as
// anonymous or lambda functions

val maximize = (a: Int, b: Int) => if (a > b) a else b // Two arguments function literal
maximize(5, 3)

// Define doubler as an immutable variable whose value is a function
// (note we aren't using def)
val doubler = (x: Int) => x * 2     // This assigns a function literal to
                                    // immutable doubler
doubler(4)

// OK, now use the _ placeholder for the doubler function argument
// Read this as "doubler is defined as a function literal that takes an integer and
// returns and integer; the function definition value multiplies its argument times 2.

val doubler: (Int) => Int = _ * 2
doubler(4)
```

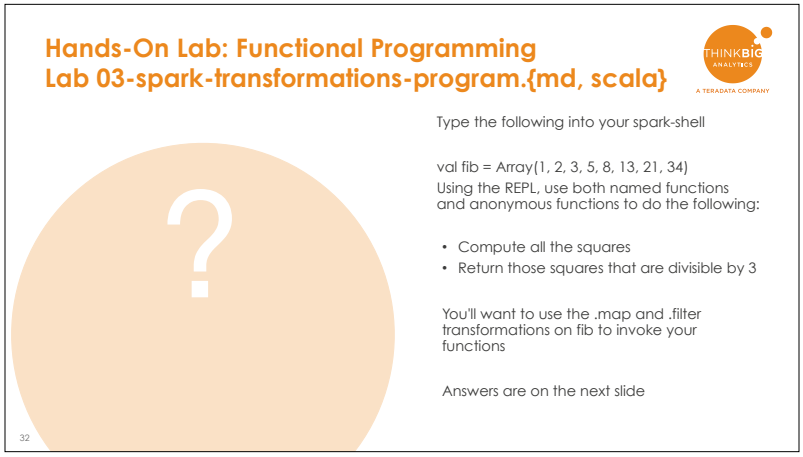

- `_ defines the first argument`

Underscores imply anonymous function arguments

- Assuming the compiler can infer the argument types, you can use an underscore to both define an anonymous function and stand for its first argument
- If there is more than one argument, the arguments are assigned sequentially
- Use with caution; _ means many things in Scala and is very sensitive to context

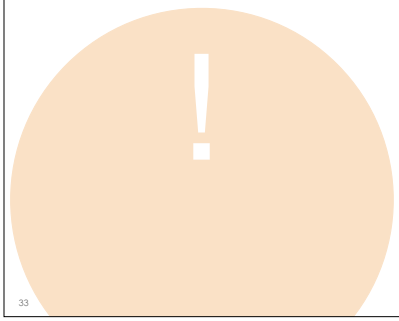
```
/// Because we can infer the types, we can use the underscore shortcut
// for this function definition

names.map("Hello " + _)

// We can also use this for our even number tester
val nums = Array(1, 2, 3, 4, 5)
nums.map(_ % 2 == 0)

// Should produce
// Array[Boolean] = Array(false, true, false, true, false)
// This may be harder to understand; it's a reducing function
// similar to that used in MapReduce

nums.reduceLeft(_ + _)  // first plus second argument
Int = 15
```


- Reduce is the action. The _ is for two different args

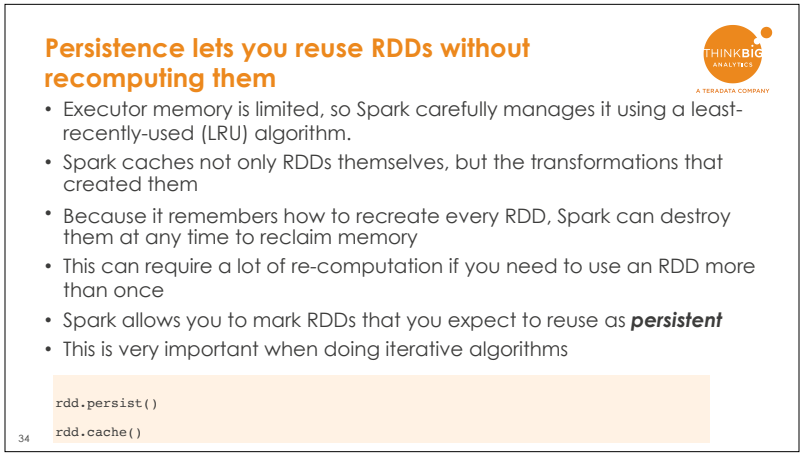


- val sets the values
- The def <something> defines the functions
- Anonymous functions like this can be sent to where the data lives

Hands-On Lab: Functional Programming



```
// Compute all the squares and sum all the
values provided

val fib = Array(1, 2, 3, 5, 8, 13, 21, 34)

def square(x: Int) = x * x
def divisible(x: Int) = (x % 3 == 0)

// First using named functions

fib.map(square).filter(divisible)

// Now use functional literals

fib.map(x => x*x).filter(_ % 3 == 0)

// Result is Array[Int] = Array(9, 441)
```

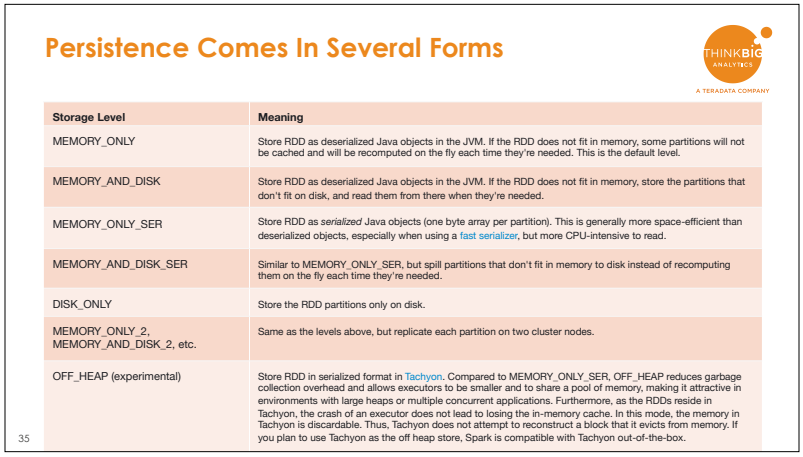


Caching is one of the secrets of creating high-performing Spark applications. The cache or persist function (both work) allows you to specify which of your datasets should be retained in memory.

This technique is most useful in iterative programs that reference the same data structure many many times.

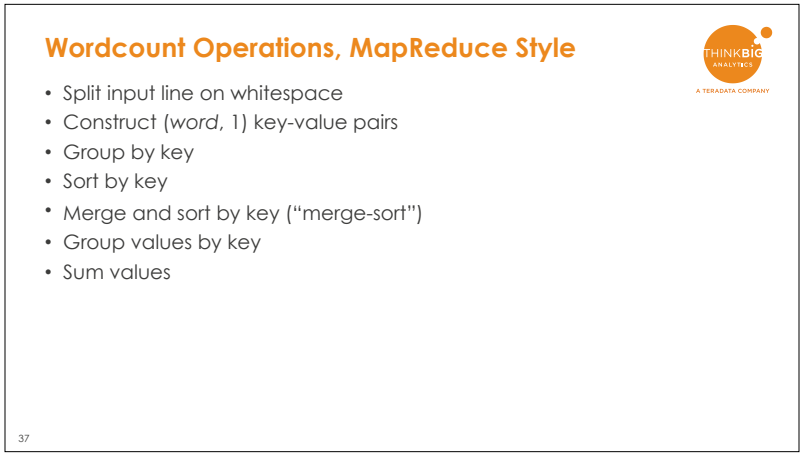
However, you should always make sure you time your program with and without explicit caching. Often Spark will do a better job of managing its memory than you will because it can optimize all aspects of the DAG, not just the piece you are looking at.

Also, you should be aware that because of lazy evaluation, you will gain no performance benefit the first reference after your persist command; the first persist is the one that puts the RDD into memory and saves it. However, the second and subsequent references should run faster.



You have many options in how to persist your RDDs. MEMORY_ONLY is the default and is the most common one. In those cases where your dataset is quite large, however, you may wish to specify MEMORY_ONLY_SER. Most of the other options are for special cases or where you have long-running computations that you can't afford to re-run in case of a node failure.

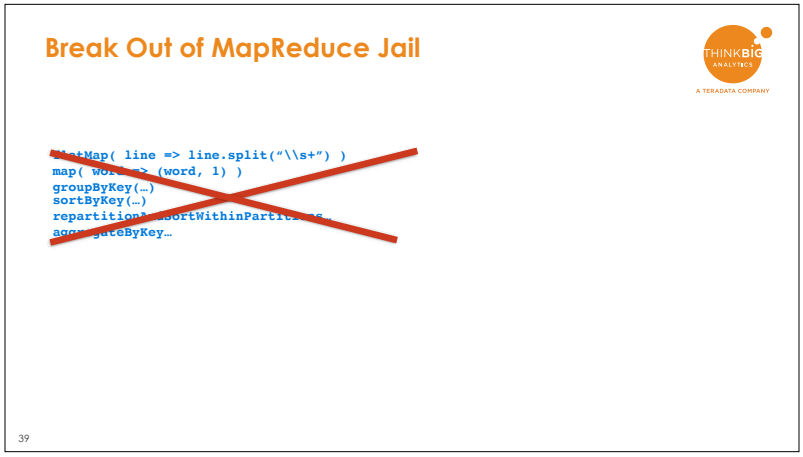
Spark Lab: 04-spark-functions-lab.{scala,md}



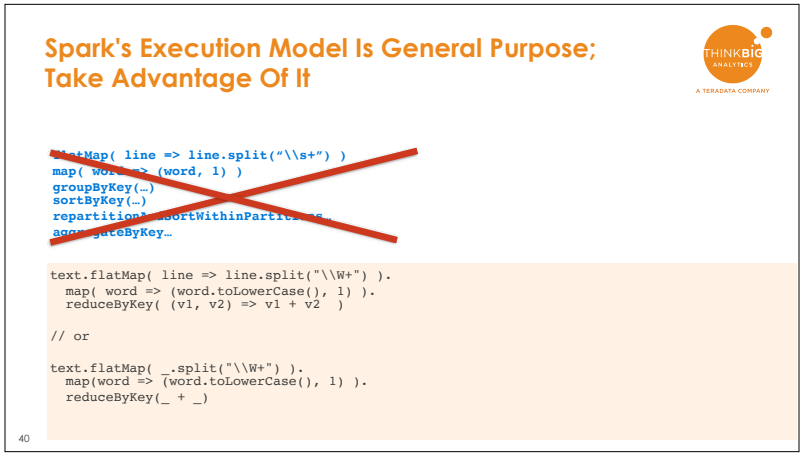
Finally, A common mistake some programmers make is simply copying the functions they used in a MapReduce program over into their Spark code. For example, if we were simply emulating MapReduce wordcount, we'd be tempted to write our code as shown on the next slide.


Wordcount Operations, MapReduce Style
THINKBIG ANALYTICS
A TERADATA COMPANY
If we translated that directly to Spark we'd get....
flatMap(line => line.split("\\s+"))
map(word => (word, 1))
groupByKey(…)
sortByKey(…)
repartitionAndSortWithinPartitions…
aggregateByKey…
38

Don't do this.


Break Out of MapReduce Jail
THINKBIG ANALYTICS
A TERADATA COMPANY
flatMap(line => line.split("\\s+"))
map(word => (word, 1))
groupByKey(…)
sortByKey(…)
repartitionAndSortWithinPartitions…
aggregateByKey…
39

Don't do this.



Don't be afraid to take advantage of Spark's flexibility and rewrite your algorithm with a more flexible DAG.

This lab is optional -- we touched on Wordcount already in lab 3, but if you want to step through it in detail, this is where to do it.

This has been a quick introduction to Spark at the RDD level. It's a new way of thinking about cluster computing and it has huge uptake in the Hadoop community. However, the best is yet to come, as we'll see when we begin working with Spark SQL.