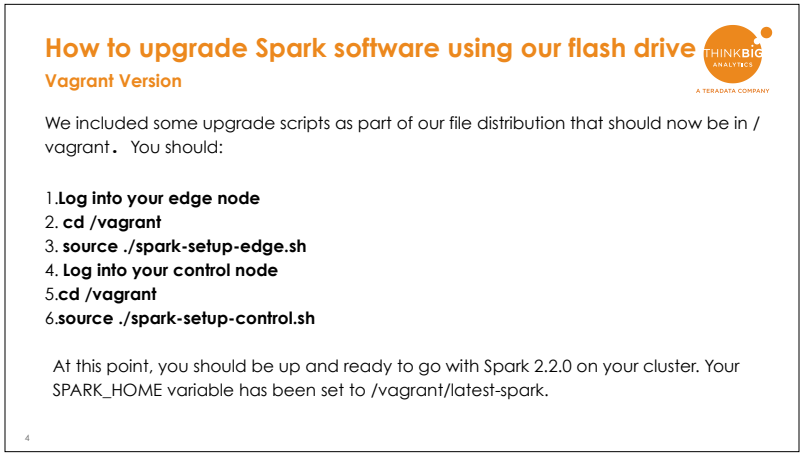
Now that we know how to program Spark in Scala, let's look at the same material in Python. Most people find this easier than Scala because they already know Python.

What We'll Cover

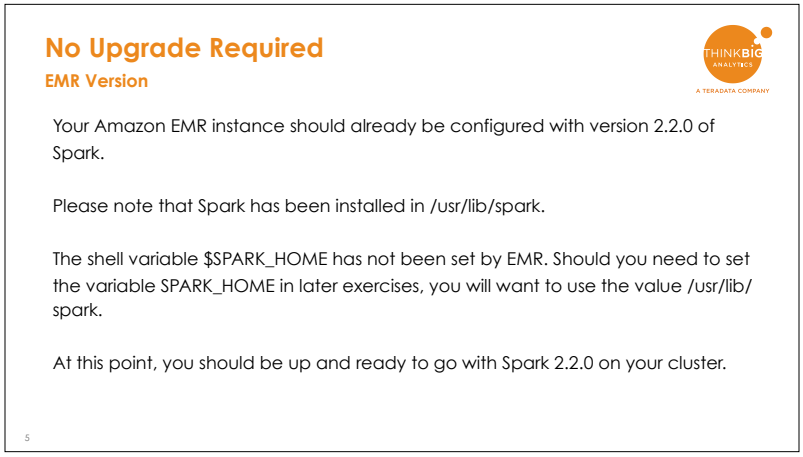
- The Spark execution model and Spark Contexts
- The PySpark shell
- Resilient Distributed Datasets (RDDs)
- External datasets
- RDD operations: Transformations and Actions
- Lambda functions
- Persistence
- Some Spark examples

Just as with Scala, we'll cover these fundamental topics.

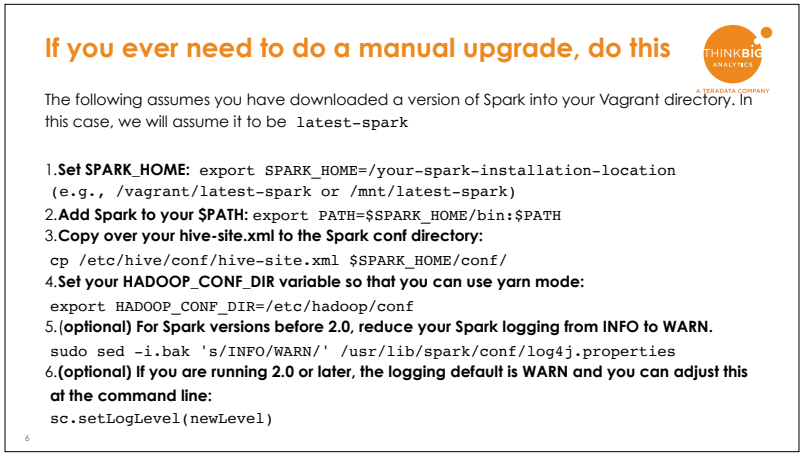
As with Scala, we'll get set up first.



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.

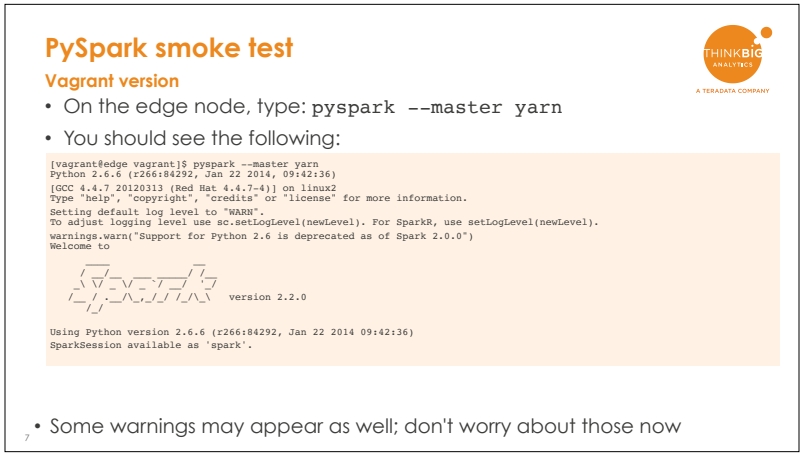


One of the best features of Spark is that it is easy to switch to using a different version than the one installed on your cluster. The process really consists of only 4 essential steps:

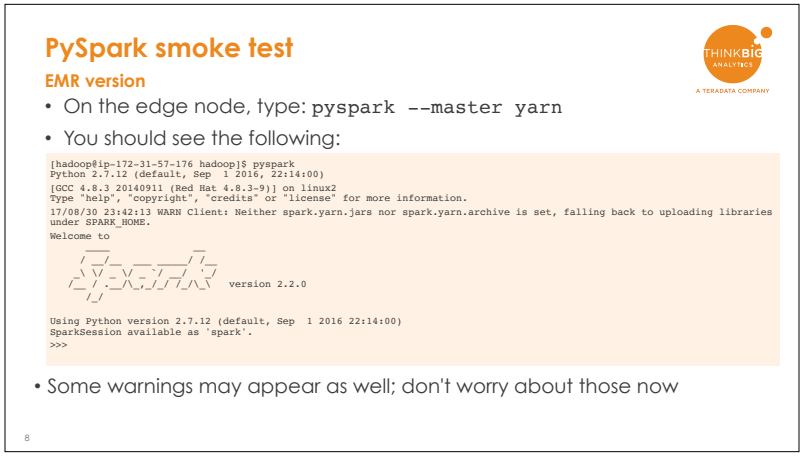
1. Setting your SPARK_HOME shell variable to point at the directory where your Spark installation lives.
2. Adding $SPARK_HOME/bin to your execution path.
3. Copying over your hive-site.xml file to your Spark configuration directory
4. Setting the shell variable $HADOOP_CONF_DIR to point at your Hadoop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

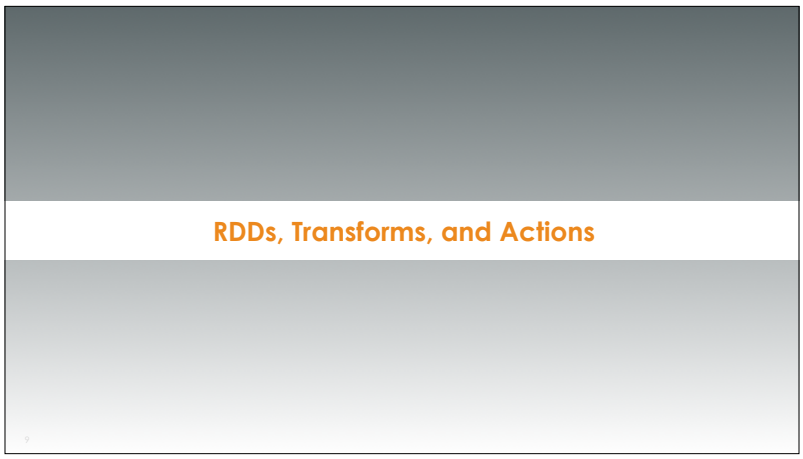
If you are running Spark 2.0 or later, the default logging level is WARN. You can change it to other levels, say ERROR, using a simple spark-shell command, sc.setLogLevel("ERROR").



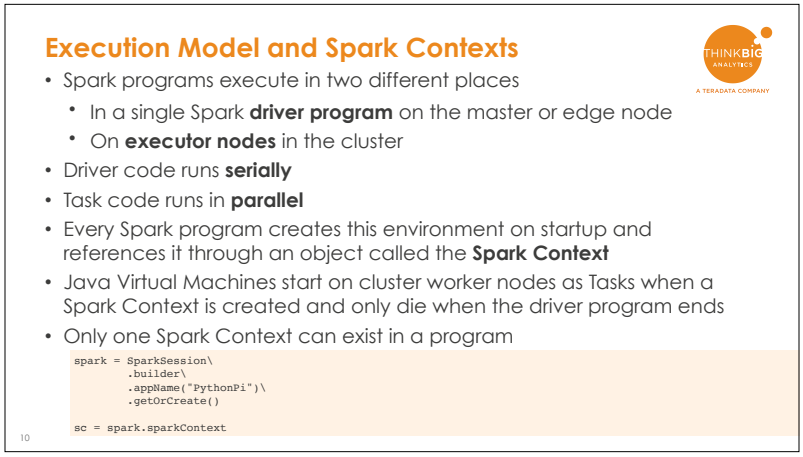
If you're using vagrant, Go to your edge node by typing vagrant ssh edge. On the edge node, type: pyspark. You should see the following.



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



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



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.

The PySpark shell

- **pyspark** creates a Python interpreter for Spark, along with a context in the variable **sc**
- At the prompt can write Python code and have it interpreted immediately.

```
pyspark
Python 2.7.12 (default, Sep  1 2016, 22:14:00)
[GCC 4.8.3 20140911 (Red Hat 4.8.3-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
17/08/30 23:42:13 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to
uploading libraries under SPARK_HOME.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 2.7.12 (default, Sep  1 2016 22:14:00)
SparkSession available as 'spark'.
>>> shakes = sc.textFile("hdfs:///data/shakespeare/input/")
>>> shakes.take(1)
[u'\t1 KING HENRY IV']
>>>
```

Options for running Python with Spark

To run `bin/pyspark` locally on exactly four cores, use:

```
$ ./bin/pyspark --master local[4]
```

Or, to also add `code.py` to the search path (in order to later be able to `import code`), use:

```
$ ./bin/pyspark --master local[4] --py-files code.py
```

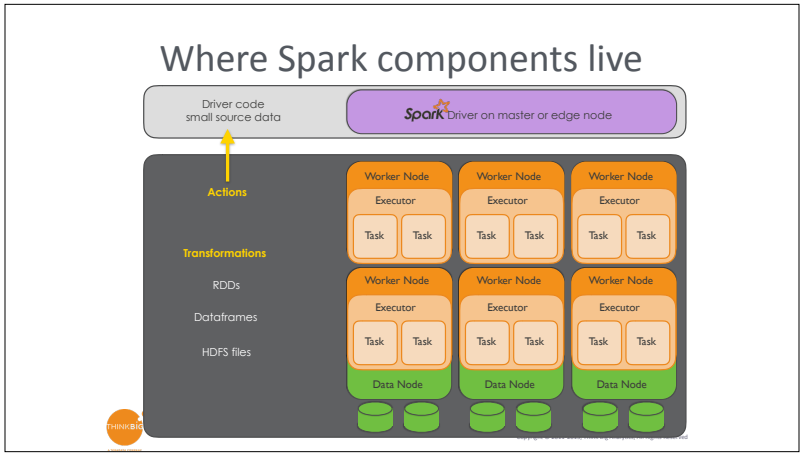
For a complete list of options, run `pyspark --help`. Behind the scenes, `pyspark` invokes the more general `spark-submit` script.

It is also possible to launch the PySpark shell in IPython, the enhanced Python interpreter. PySpark works with IPython 1.0.0 and later. To use IPython, set the `PYSPARK_DRIVER_PYTHON` variable to `ipython` when running `bin/pyspark`:

```
$ PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark
```

You can customize the `ipython` command by setting `PYSPARK_DRIVER_PYTHON_OPTS`. For example, to launch the IPython Notebook with PyLab plot support.

```
$ PYSPARK_DRIVER_PYTHON=ipython PYSPARK_DRIVER_PYTHON_OPTS="notebook" ./bin/pyspark
```

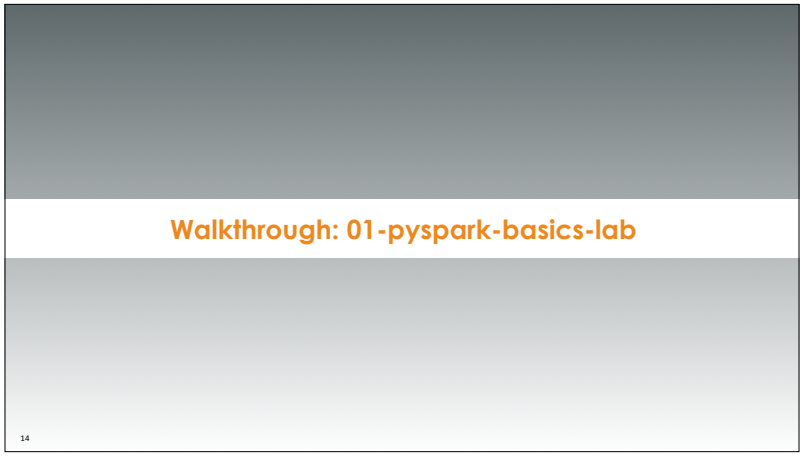


If we look at this visually, the light gray box represents the edge node, depending on your cluster configuration. That's where the Spark driver lives. It runs serially there on one node.

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.



Let's go hands-on with Spark with our first Python lab. Two versions exist: one in Markdown and one in pure Python 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-python-first-program.md

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

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

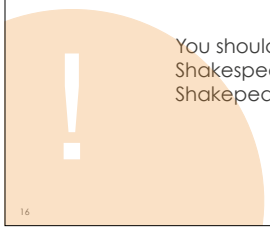
Try using that pattern to count the number of lines in the file `/data/shakespeare/input` that contain the word "king". You'll want to use the `toLowerCase` function before the contains function to ensure all the text is lower case.

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

Hands-On Lab: First Scala Program

One solution looks like this

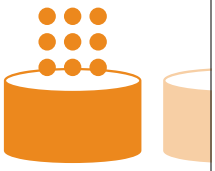
```
rdd = sc.textFile("hdfs:///data/shakespeare/input")
kings = rdd.filter(lambda line: "king" in line.lower())
kings.count()
```



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.

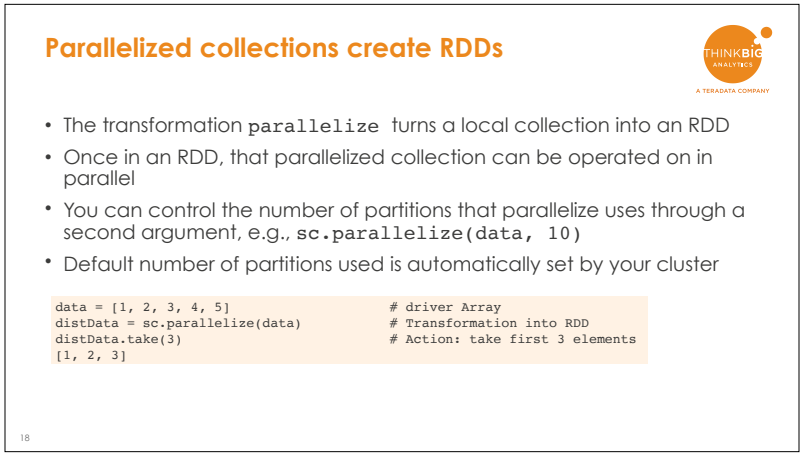
Resilient Distributed Datasets (RDDs) are the primary parallel data structure

- 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

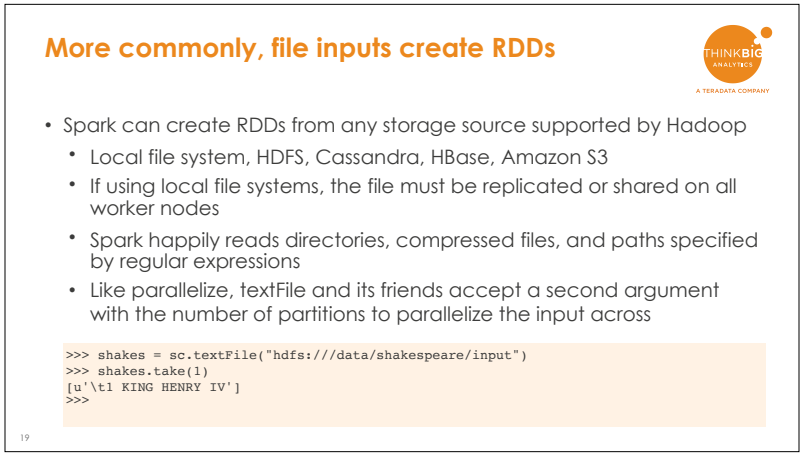


17

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

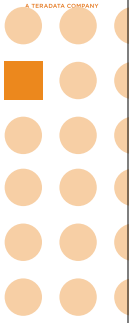
- **Transformations** convert an RDD into another RDD
- **Actions** often turn an RDD into something else
- Only **actions** cause evaluation

```
lines = sc.textFile("hdfs:///data/shakespeare/input")      # Transformation
lineLengths = lines.map(lambda s: len(s))                  # Transformation
totalLength = lineLengths.reduce(lambda a, b: a + b)       # Action
totalLength    # print result
# should be 5167385
```

20

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

Common Transformations



```
Assume: rdd = sc.parallelize(["This is a line of text", "And so is this"])
```

map — apply a function while preserving structure

```
rdd.map( lambda line: line.split() ).collect()
→ [['This', 'is', 'a', 'line', 'of', 'text'], ['And', 'so', 'is', 'this']]
```

flatMap — apply a function while flattening structure

```
rdd.flatMap( lambda line: line.split()).collect()
→ ['This', 'is', 'a', 'line', 'of', 'text', 'And', 'so', 'is', 'this']
```

filter — discard elements from an RDD which don't match a condition

```
rdd.filter( lambda line: "so" in line ) .collect()
→ ['And so is this']
```

Common Transformations


```
pair_rdd = rdd.flatMap(lambda line: line.split() ).map( lambda word: (word.lower(), 1)).collect(
→ [('this', 1), ('is', 1), ('a', 1), ('line', 1), ('of', 1), ('text', 1), ('and', 1), ('so', 1), ('is', 1),
('this', 1)]
```

groupByKey — group values by key

```
pair_rdd.groupByKey().collect()
→ [('a', <pyspark.resultiterable.ResultIterable object at 0x20f7f50>), ('and',
<pyspark.resultiterable.ResultIterable object at 0x1fa05d0>), ('this',
<pyspark.resultiterable.ResultIterable object at 0x20f7b10>), ('is',
<pyspark.resultiterable.ResultIterable object at 0x1fd20d0>), ('so',
<pyspark.resultiterable.ResultIterable object at 0x1fd2190>), ('line',
<pyspark.resultiterable.ResultIterable object at 0x20d6e90>), ('text',
<pyspark.resultiterable.ResultIterable object at 0x20d6e10>), ('of',
<pyspark.resultiterable.ResultIterable object at 0x20459d0>)]
```

reduceByKey — apply a function to each value for each key

```
pair_rdd.reduceByKey( lambda v1, v2: v1 + v2 ).collect()
→ [('a', 1), ('and', 1), ('this', 2), ('is', 2), ('so', 1), ('line', 1), ('text', 1), ('of',
```

Spark Transformations

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

For PairRDD:

- groupByKey
- reduceByKey
- aggregateByKey
- sortByKey
- join
- cogroup
- keys
- values
- repartition
- repartitionAndSortWithinPartitions

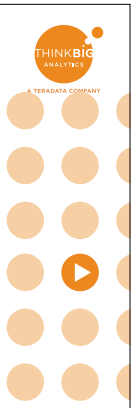
see http://spark.apache.org/docs/2.2.0/programming-guide.html#transformations

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

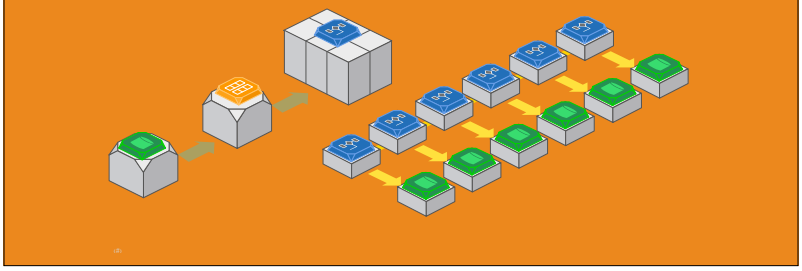


- reduce
- collect
- count
- first
- take
- takeSample
- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- countByKey
- foreach

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

An important idea: Function Literals
AKA Anonymous Functions
AKA Lambda Functions
allow us to move code to worker nodes



Function definitions are pretty common in most languages

- A function definition lets us name a stored operation that takes arguments and returns a result
- Python function result types are determined by the types of the arguments and the function operations

```
def add(x, y):
  return x + y

add(42,13)

# Shorter version all on one line
def add(x, y): return x + y
```

Anonymous functions allow us to do without the name

- ***Function literals*** 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 named function greeting
def greeting(x): return "Hello " + x

greeting("Joe")

# an anonymous function whose definition is assigned to variable greeting
greeting = lambda x: "Hello " + x
greeting("Joe")
```

Anonymous functions allow us to do without the name

- Either type of function can be used for Spark functions

```
names = ["Joe", "Mary", "Barbara"]
# comprehension for a local list; local lists don't have the map method defined.
[greeting(x) for x in names]

# Now do this in Spark
names = sc.parallelize(["Joe", "Mary", "Barbara"])
names.map(greeting).collect()

# should get
#    ['Hello Joe', 'Hello Mary', 'Hello Barbara']

# Now let's get rid of the name greeting

names.map(lambda x: "Hello " + x).collect()
# should get the same answer:
#    ['Hello Joe', 'Hello Mary', 'Hello Barbara']
```

Hands-On Lab: Functional Programming
Lab 03-python-transformations-program.{md, scala}

Type the following into pyspark

fib = sc.parallelize([1, 2, 3, 5, 8, 13, 21, 34])

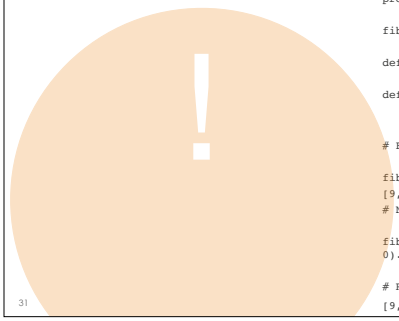
Using the REPL, use both named functions and anonymous functions to do the following:

- Compute all the squares
- Return those squares that are divisible by 3

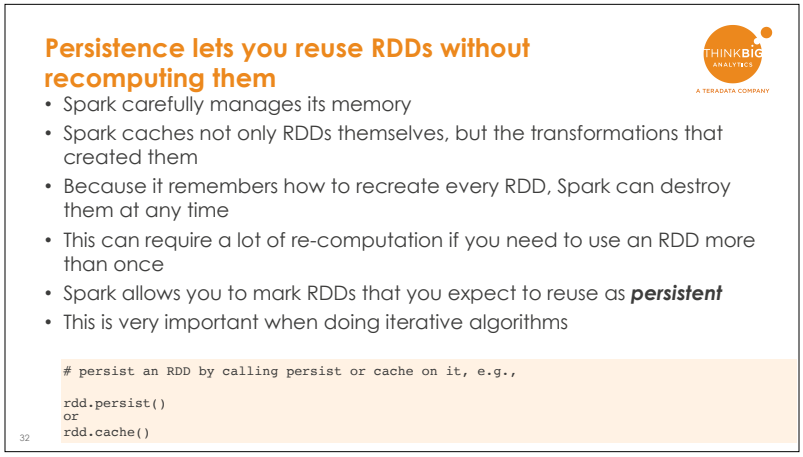
You'll want to use the .map and .filter transformations on fib to invoke your functions

Answers are on the next slide

Hands-On Lab: Functional Programming



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

fib = sc.parallelize([1, 2, 3, 5, 8, 13, 21, 34])

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

# First using named functions

fib.map(square).filter(divisible).collect()
[9, 441]
# Now use functional literals

fib.map(lambda x: x*x).filter(lambda x: x % 3 ==
0).collect()

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

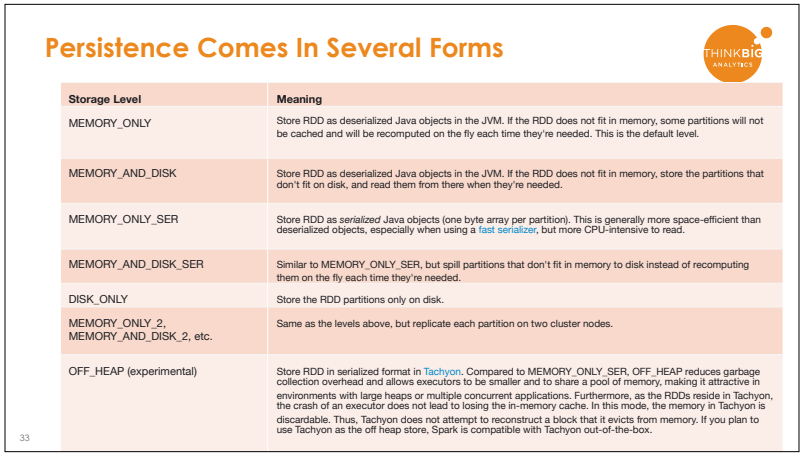


Caching is one of the secrets of creating high-performing Spark applications. The cache or persist function (both work) allows you to specify which of your datasets should be retained in memory.

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.

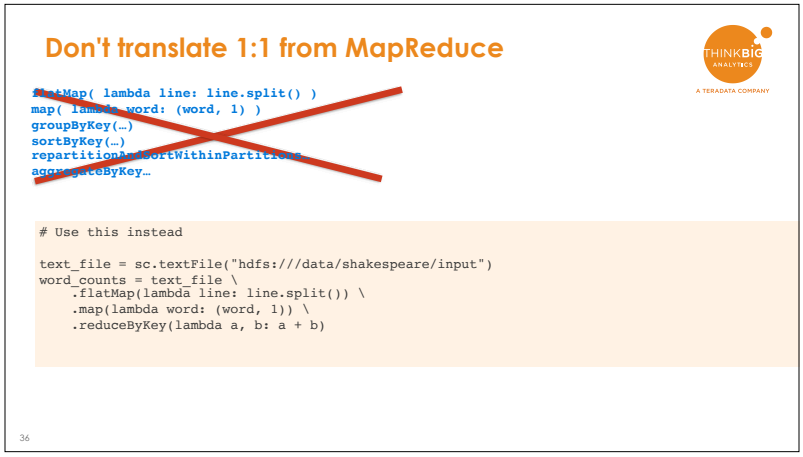


You have many options in how to persist your RDDs. MEMORY_ONLY is the default and is the most common one. In those cases where your dataset is quite large, however, you may wish to specify MEMORY_ONLY_SER. Most of the other options are for special cases or where you have long-running computations that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

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

Wordcount operations

- Split input line on whitespace
- Construct (*word*, 1) key-value pairs
- Group by key
- Sort by key
- Merge and sort by key ("merge-sort")
- Group values by key
- Sum values



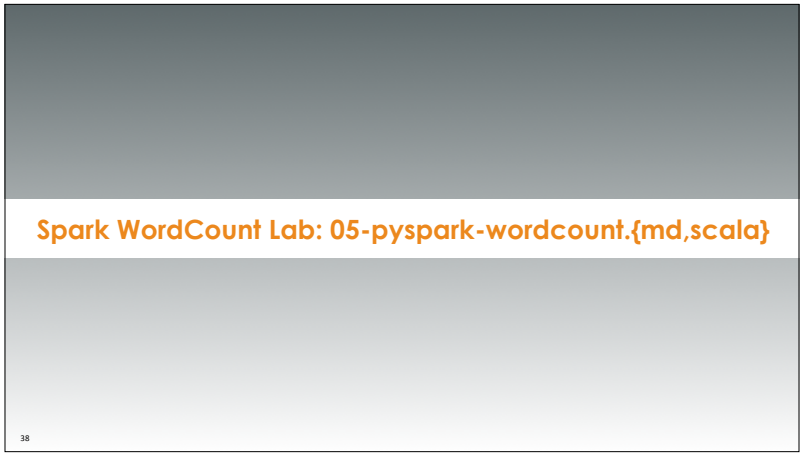
Don't be afraid to take advantage of Spark's flexibility and rewrite your algorithm with a more flexible DAG.

Summary

- Spark provides a new full-stack cluster development environment
- Resilient distributed datasets (RDDs) provide in-memory storage of data across the cluster instead of spilling to disk
- Transformation operations don't execute until an action is called
- Functional programming allows us to push code to the cluster nodes dynamically

However……

SparkSQL and Dataframes are how you will usually want to use Spark



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.

Summary

- Pyspark and Python on Spark provide a similar experience to Scala, but without the type anxiety
- Most of what you've learned in Scala is directly transferrable, provided you remember to use lambdas instead of Scala's shortcuts
- Be sure to check the Python Spark API documentation for differences with Scala