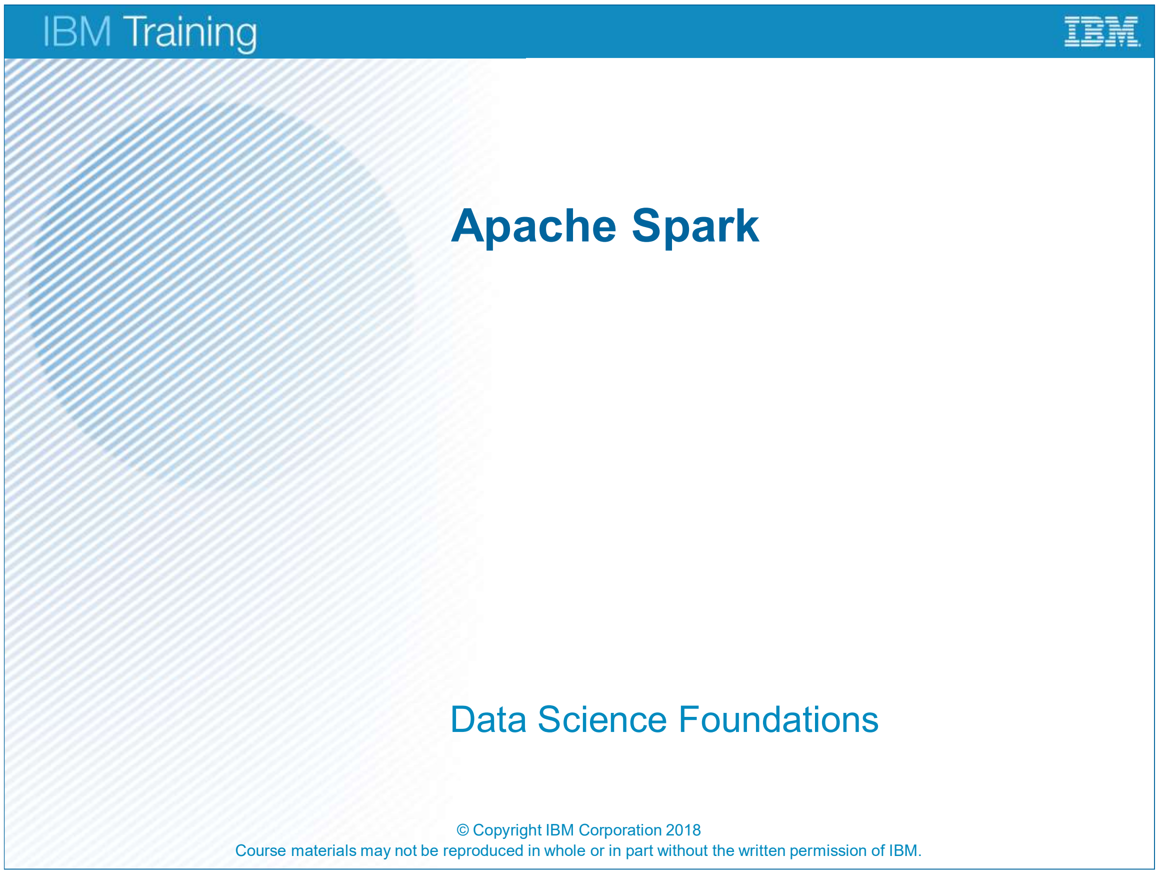
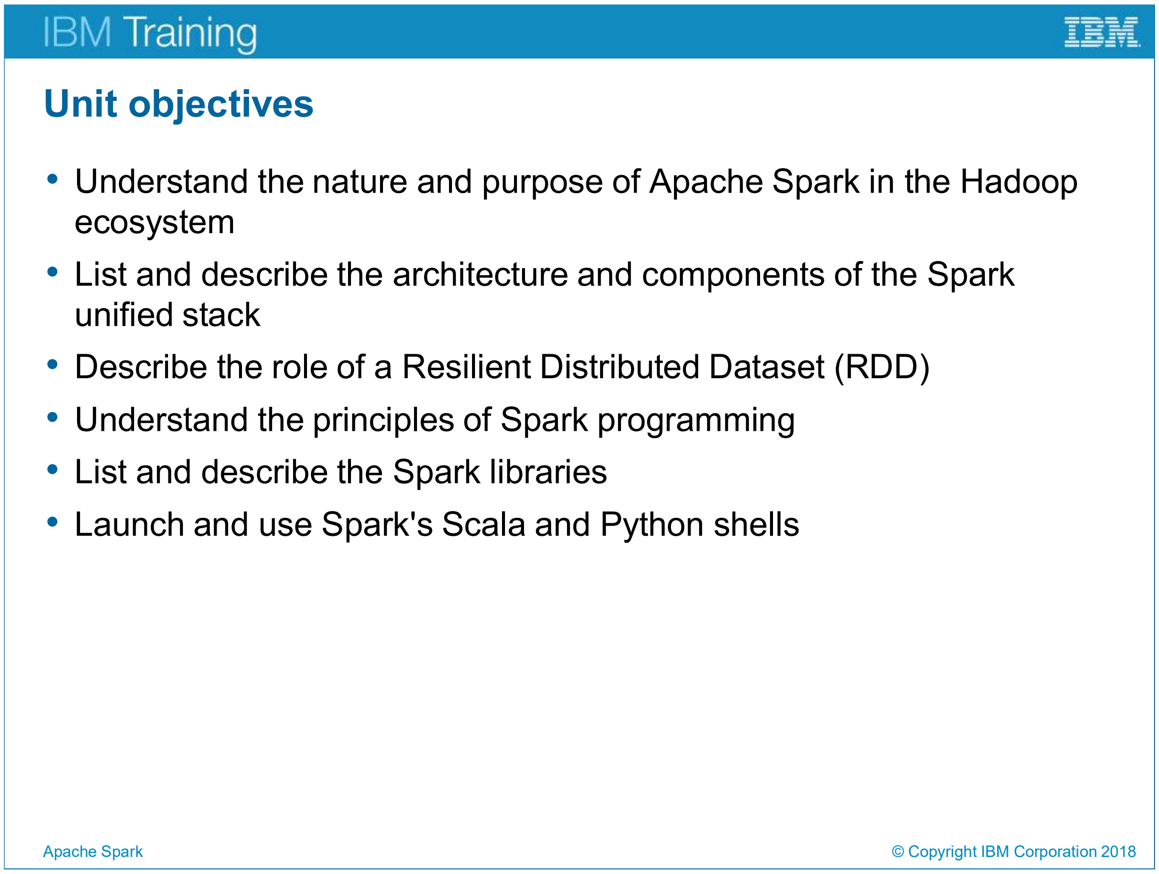
Unit 6 Apache Spark



IBM Training

Apache Spark

Data Science Foundations



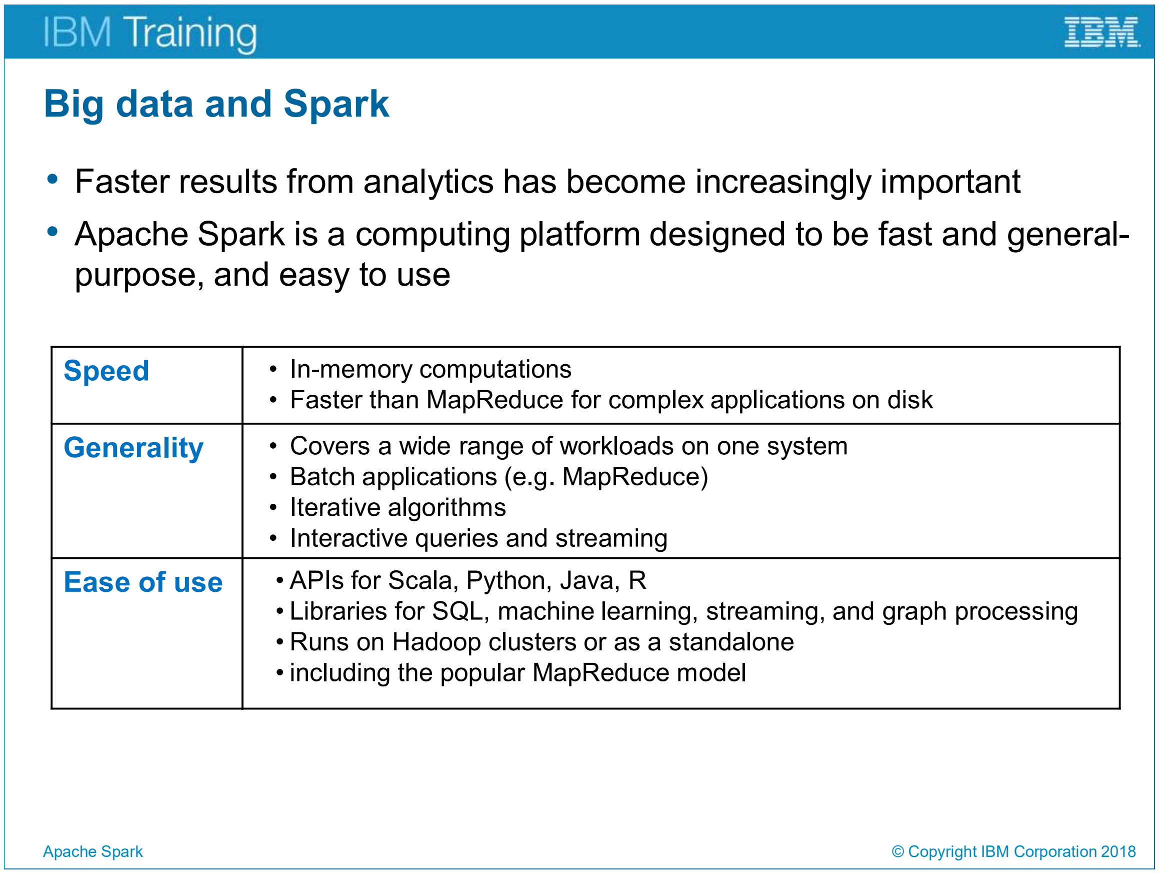
IBM Training

Unit objectives

- Understand the nature and purpose of Apache Spark in the Hadoop ecosystem
- List and describe the architecture and components of the Spark unified stack
- Describe the role of a Resilient Distributed Dataset (RDD)
- Understand the principles of Spark programming
- List and describe the Spark libraries
- Launch and use Spark's Scala and Python shells

Apache Spark

Unit objectives



IBM Training

Big data and Spark

- Faster results from analytics has become increasingly important
- Apache Spark is a computing platform designed to be fast and general-purpose, and easy to use

Speed	• In-memory computations • Faster than MapReduce for complex applications on disk
Generality	• Covers a wide range of workloads on one system • Batch applications (e.g. MapReduce) • Iterative algorithms • Interactive queries and streaming
Ease of use	• APIs for Scala, Python, Java, R • Libraries for SQL, machine learning, streaming, and graph processing • Runs on Hadoop clusters or as a standalone • including the popular MapReduce model

Apache Spark

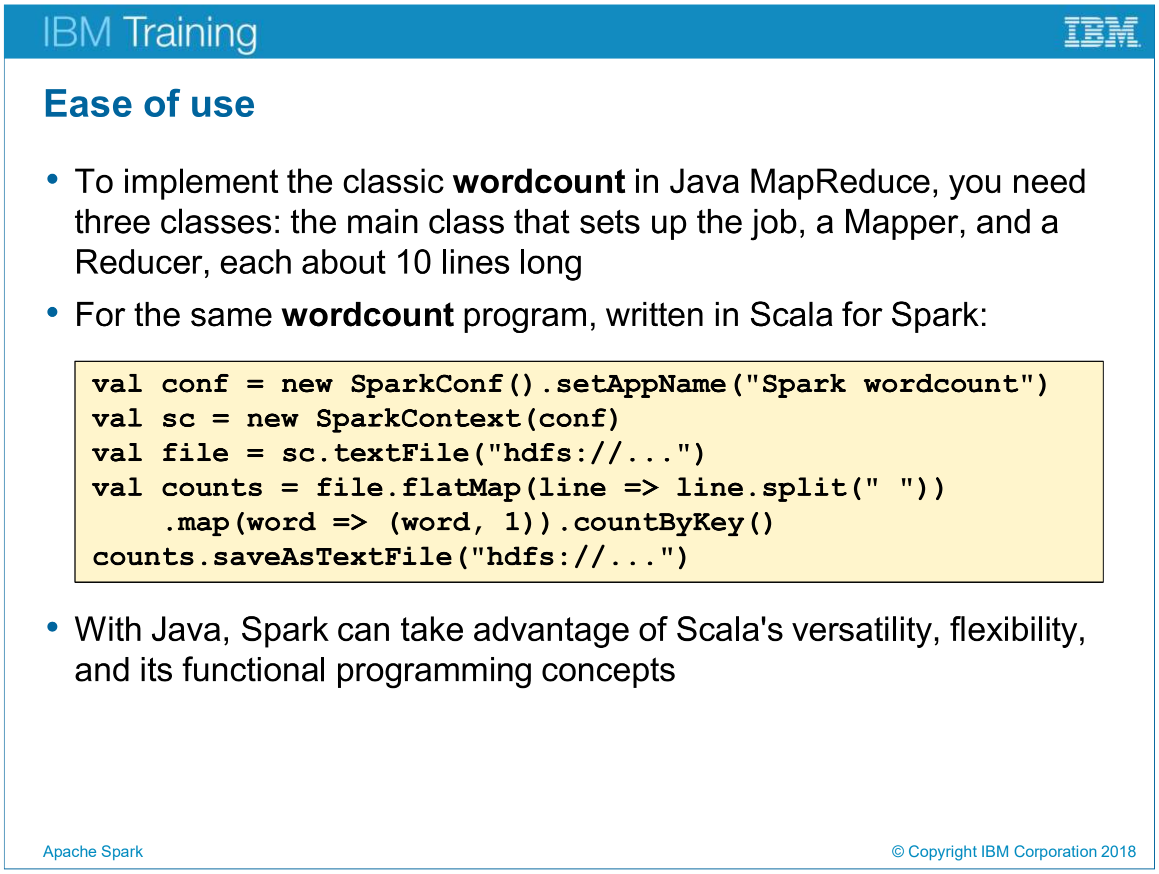
Big data and Spark

There is an explosion of data, and no matter where you look, data is everywhere. You get data from social media such as Twitter feeds, Facebook posts, SMS, and a variety of others. The need to be able to process those data as quickly as possible becomes more important than ever. How can you find out what your customers want and be able to offer it to them right away? You do not want to wait hours for a batch job to complete; you need to have it in minutes or less.

MapReduce has been useful, but the amount of time it takes for the jobs to run is no longer acceptable in many situations. The learning curve to writing a MapReduce job is also difficult as it takes specific programming knowledge and expertise. Also, MapReduce jobs only work for a specific set of use cases. You need something that works for a wider set of use cases.

Apache Spark was designed as a computing platform to be fast, general-purpose, and easy to use. It extends the MapReduce model and takes it to a whole other level.

- The speed comes from the in-memory computations. Applications running in memory allows for a much faster processing and response. Spark is even faster than MapReduce for complex applications on disk.
- This generality covers a wide range of workloads under one system. You can run batch application such as MapReduce type jobs or iterative algorithms that builds upon each other. You can also run interactive queries and process streaming data with your application. In a later slide, you'll see that there are a number of libraries which you can easily use to expand beyond the basic Spark capabilities.
- The ease of use with Spark enables you to quickly pick it up using simple APIs for Scala, Python, Java, and R. As mentioned, there are additional libraries which you can use for SQL, machine learning, streaming, and graph processing. Spark runs on Hadoop clusters such as Hadoop YARN or Apache Mesos, or even as a standalone with its own scheduler.



IBM Training

Ease of use

- To implement the classic **wordcount** in Java MapReduce, you need three classes: the main class that sets up the job, a Mapper, and a Reducer, each about 10 lines long
- For the same **wordcount** program, written in Scala for Spark:

```
val conf = new SparkConf().setAppName("Spark wordcount")
val sc = new SparkContext(conf)
val file = sc.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
    .map(word => (word, 1)).countByKey()
counts.saveAsTextFile("hdfs://...")
```

- With Java, Spark can take advantage of Scala's versatility, flexibility, and its functional programming concepts

Apache Spark © Copyright IBM Corporation 2018

Ease of use

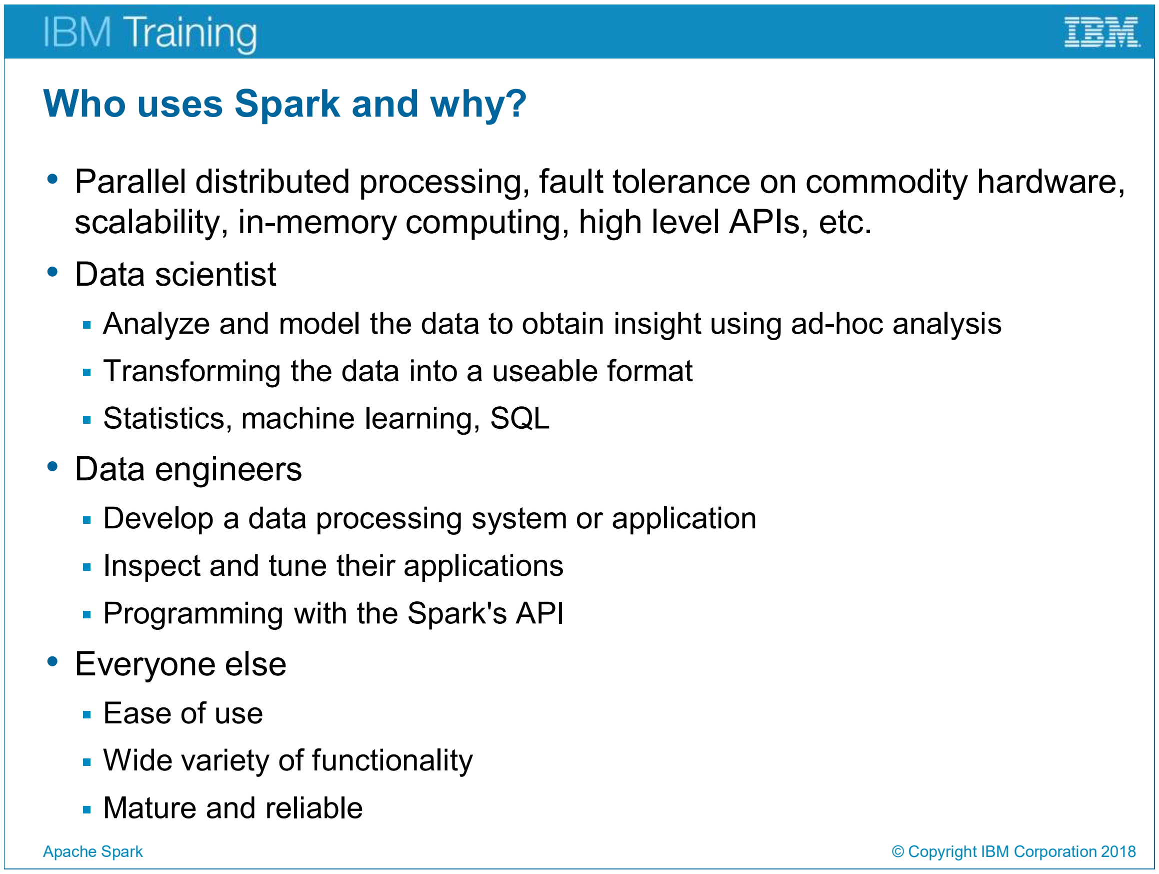
Spark supports Scala, Python, Java, and R programming languages, all now in widespread use among data scientists. The slide shows programming in Scala. Python is widespread among data scientists and in the scientific community, bringing those users on par with Java and Scala developers.

An important aspect of Spark is the ways that it can combine the functionalities of many tools in available in the Hadoop ecosystem to provide a single unifying platform. In addition, the Spark execution model is general enough that a single framework can be used for:

- batch processing operations(similar to that provided by MapReduce),
- stream data processing,
- machine learning,
- SQL-like operations, and
- graph operations.

The result is that many different ways of working with data are available on the on same platform, bridging the gap between the work of the classic big data programmer, data engineers and data scientists.

All the same, Spark has its own limitations. There are no universal tools. Thus Spark is not suitable for transaction processing and other ACID types of operations.


IBM Training

Who uses Spark and why?

- Parallel distributed processing, fault tolerance on commodity hardware, scalability, in-memory computing, high level APIs, etc.
- Data scientist
  - Analyze and model the data to obtain insight using ad-hoc analysis
  - Transforming the data into a useable format
  - Statistics, machine learning, SQL
- Data engineers
  - Develop a data processing system or application
  - Inspect and tune their applications
  - Programming with the Spark's API
- Everyone else
  - Ease of use
  - Wide variety of functionality
  - Mature and reliable

Apache Spark © Copyright IBM Corporation 2018

Who uses Spark and why?

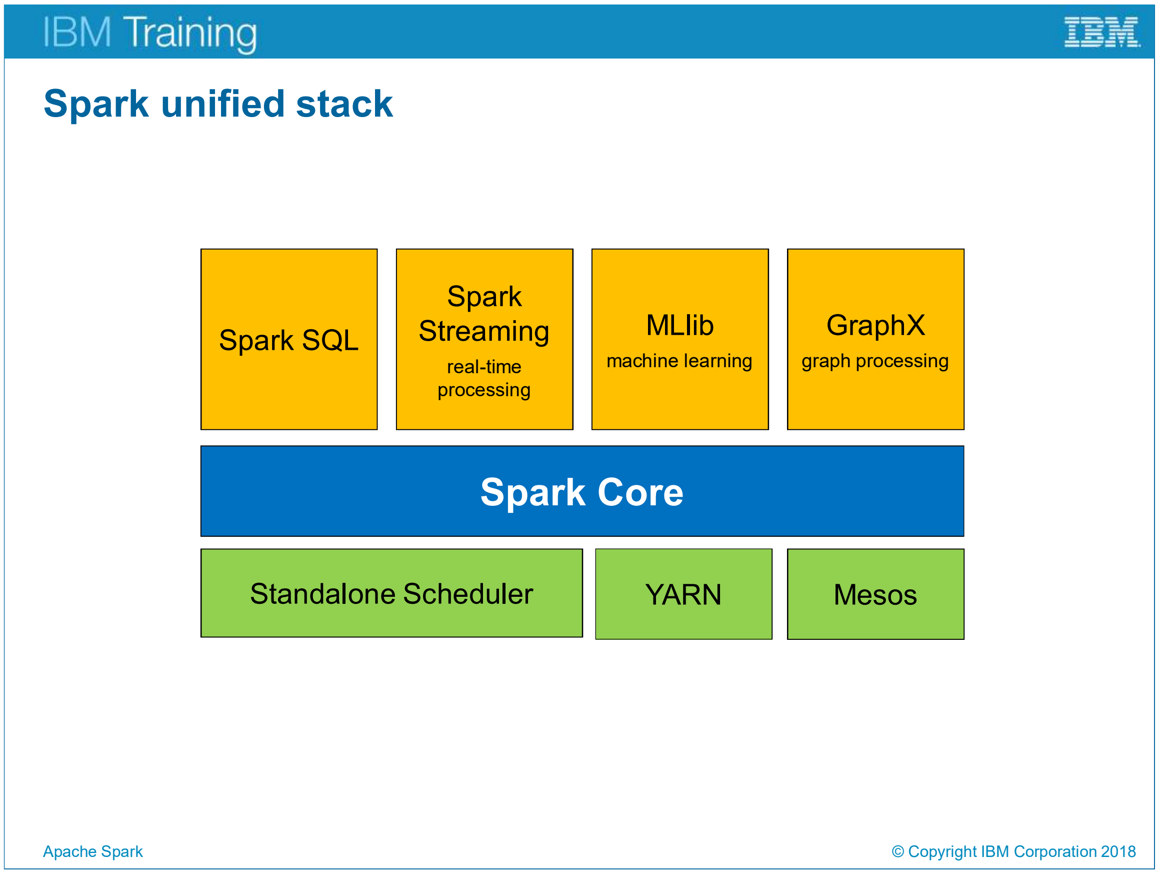
You may be asking why you would want to use Spark and what you would use it for.

Spark is related to MapReduce in a sense that it expands on Hadoop's capabilities. Like MapReduce, Spark provides parallel distributed processing, fault tolerance on commodity hardware, scalability, etc. Spark adds to the concept with aggressively cached in-memory distributed computing, low latency, high level APIs and stack of high level tools described on the next slide. This saves time and money.

There are two groups that we can consider here who would want to use Spark: Data Scientists and Engineers - and they have overlapping skill sets.

- Data scientists need to analyze and model the data to obtain insight. They need to transform the data into something they can use for data analysis. They will use Spark for its ad-hoc analysis to run interactive queries that will give them results immediately. Data scientists may have experience using SQL, statistics, machine learning and some programming, usually in Python, MatLab or R. Once the data scientists have obtained insights on the data and determined that there is a need develop a production data processing application, a web application, or some system to act upon the insight, the work is handed over to data engineers.
- Data engineers use Spark's programming API to develop a system that implement business use cases. Spark parallelize these applications across the clusters while hiding the complexities of distributed systems programming and fault tolerance. Data engineers can employ Spark to monitor, inspect, and tune applications.

For everyone else, Spark is easy to use with a wide range of functionality. The product is mature and reliable.


IBM Training

Spark unified stack

Apache Spark

© Copyright IBM Corporation 2018

Spark unified stack

The Spark core is at the center of the Spark Unified Stack. The Spark core is a general-purpose system providing scheduling, distributing, and monitoring of the applications across a cluster.

The Spark core is designed to scale up from one to thousands of nodes. It can run over a variety of cluster managers including Hadoop YARN and Apache Mesos, or more simply, it can run standalone with its own built-in scheduler.

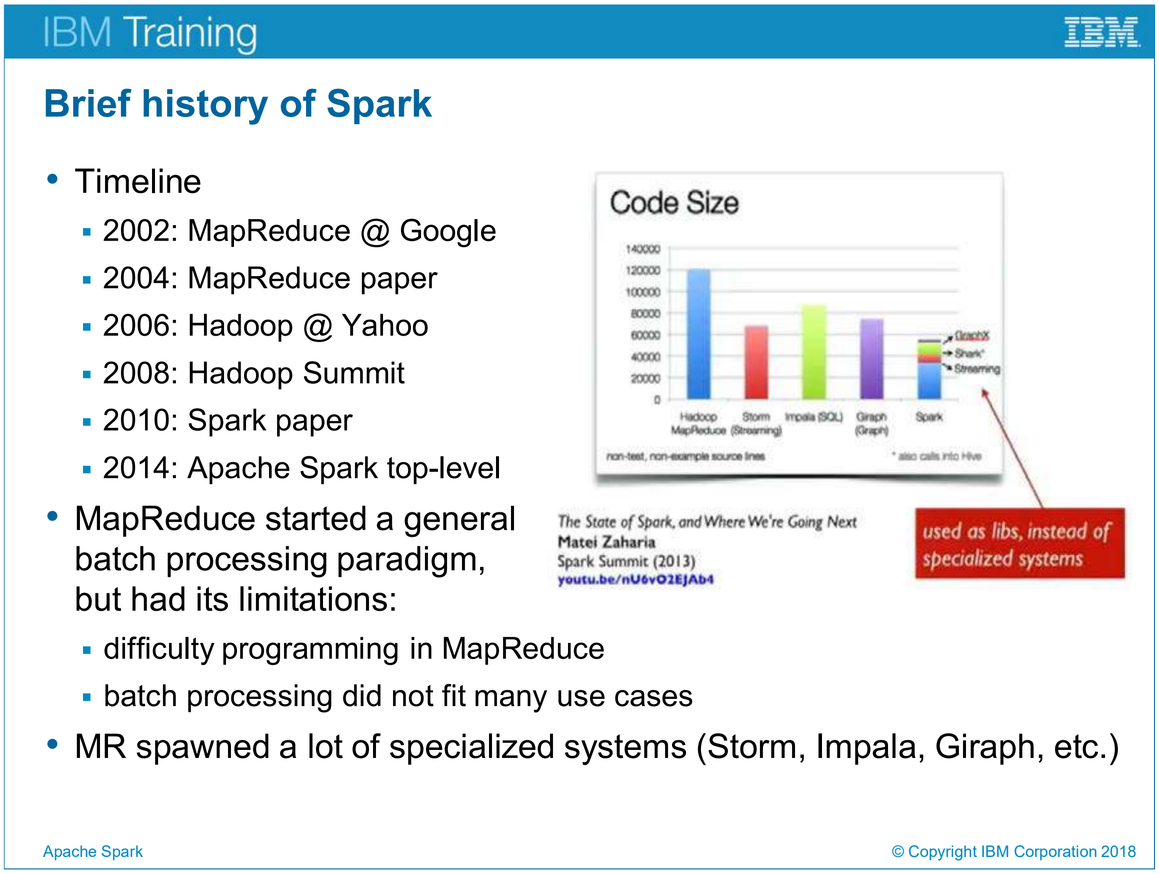
Spark Core contains basic Spark functionalities required for running jobs and needed by other components. The most important of these is the RDD concept, or resilient distributed dataset, the main element of Spark API. RDD is an abstraction of a distributed collection of items with operations and transformations applicable to the dataset. It is resilient because it is capable of rebuilding datasets in case of node failures.

Various add-in components can run on top of the core; these are designed to interoperate closely, letting the users combine them, just like they would any libraries in a software project. The benefit of the Spark Unified Stack is that all the higher layer components will inherit the improvements made at the lower layers. Example: Optimization to the Spark Core will speed up the SQL, the streaming, the machine learning and graph processing libraries as well.

Spark simplifies the picture by providing many of Hadoop ecosystem functions through several purpose-built components. These are Spark Core, Spark SQL, Spark Streaming, Spark MLib, and Spark GraphX:

- **Spark SQL** is designed to work with the Spark via SQL and HiveQL (a Hive variant of SQL). Spark SQL allows developers to intermix SQL with Spark's programming language supported by Python, Scala, Java, and R.
- **Spark Streaming** provides processing of live streams of data. The Spark Streaming API closely matches that of the Sparks Core's API, making it easy for developers to move between applications that processes data stored in memory vs arriving in real-time. It also provides the same degree of fault tolerance, throughput, and scalability that the Spark Core provides.
- **MLlib** is the machine learning library that provides multiple types of machine learning algorithms. These algorithms are designed to scale out across the cluster as well. Supported algorithms include logistic regression, naive Bayes classification, SVM, decision trees, random forests, linear regression, k-means clustering, and others.
- **GraphX** is a graph processing library with APIs to manipulate graphs and performing graph-parallel computations. Graphs are data structures comprised of vertices and edges connecting them. GraphX provides functions for building graphs and implementations of the most important algorithms of the graph theory, like page rank, connected components, shortest paths, and others.

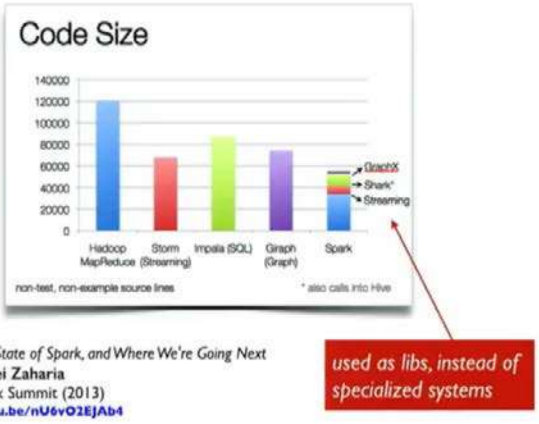
"If you compare the functionalities of Spark components with the tools in the Hadoop ecosystem, you can see that some of the tools are suddenly superfluous. For example, Apache Storm can be replaced by Spark Streaming, Apache Giraph can be replaced by Spark GraphX and Spark MLlib can be used instead of Apache Mahout. Apache Pig, and Apache Sqoop are not really needed anymore, as the same functionalities are covered by Spark Core and Spark SQL. But even if you have legacy Pig workflows and need to run Pig, the Spork project enables you to run Pig on Spark." - Bonaći, M., & Zečević, P. (2015). *Spark in action*. Greenwich, CT: Manning Publications. ("Spork" is Apache Pig on Spark, see https://github.com/sigmoidanalytics/spork).



IBM Training

Brief history of Spark

- Timeline
  - 2002: MapReduce @ Google
  - 2004: MapReduce paper
  - 2006: Hadoop @ Yahoo
  - 2008: Hadoop Summit
  - 2010: Spark paper
  - 2014: Apache Spark top-level
- MapReduce started a general batch processing paradigm, but had its limitations:
  - difficulty programming in MapReduce
  - batch processing did not fit many use cases
- MR spawned a lot of specialized systems (Storm, Impala, Giraph, etc.)



The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

Apache Spark

© Copyright IBM Corporation 2018

Brief history of Spark

MapReduce was developed over a decade ago as a fault tolerant framework that ran on commodity systems.

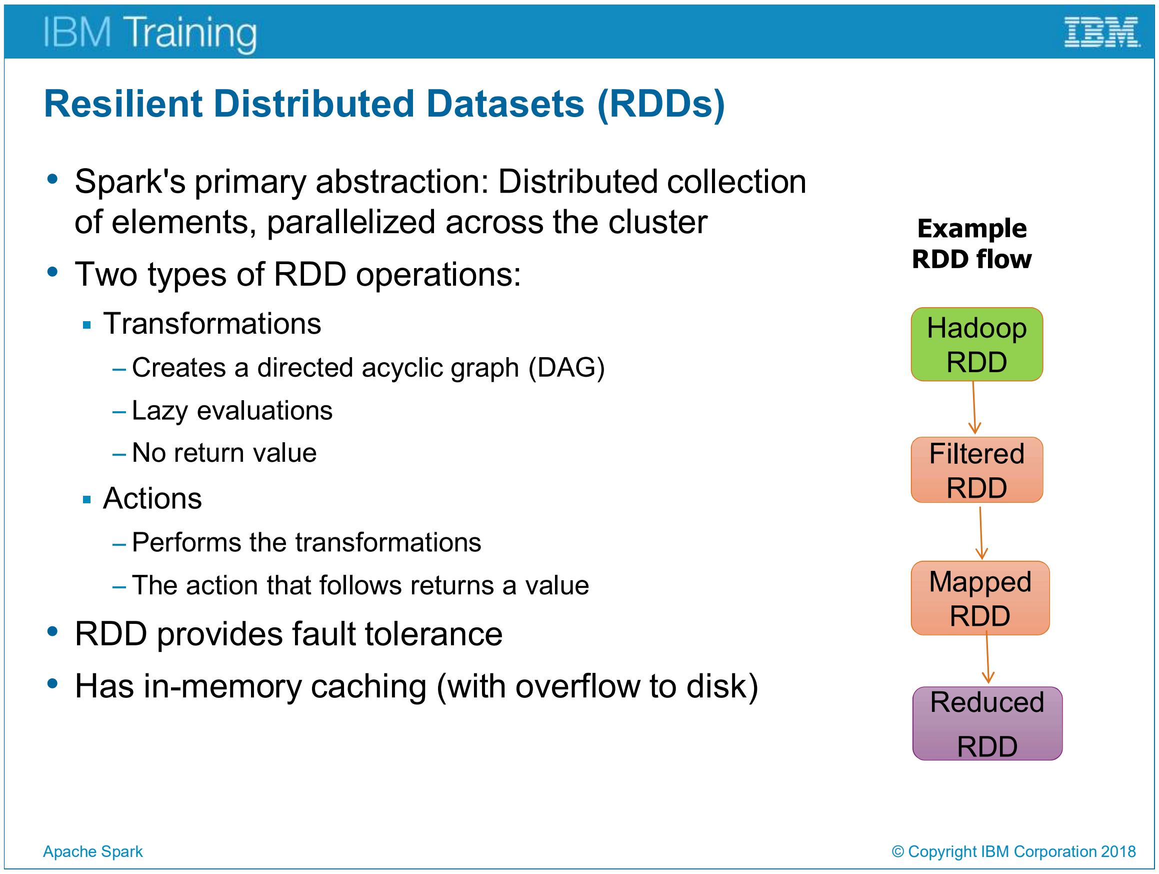
MapReduce started off as a general batch processing system, but there are two major limitations.

- Difficulty in programming directly in MR.
- Batch jobs do not fit many use cases.

This spawned specialized systems to handle the other needed use cases. When you try to combine these third party systems into your applications, there is a lot of overhead.

Spark comes out about a decade later with similar framework to run data processing on commodity systems also using a fault tolerant framework.

Taking a looking at the code size of some applications on the graph on this slide, you can see that Spark requires a considerable amount less. Even with Spark's libraries, it only adds a small amount of code due to how tightly everything is with little overhead. There is great value to be able to express a wide variety of use cases with few lines of code and the corresponding increase in programmer productivity.



IBM Training

Resilient Distributed Datasets (RDDs)

- Spark's primary abstraction: Distributed collection of elements, parallelized across the cluster
- Two types of RDD operations:
  - Transformations
    - Creates a directed acyclic graph (DAG)
    - Lazy evaluations
    - No return value
  - Actions
    - Performs the transformations
    - The action that follows returns a value
- RDD provides fault tolerance
- Has in-memory caching (with overflow to disk)

Example RDD flow

$$\begin{tikzcd} \text{Hadoop RDD} \arrow[d] \\ \text{Filtered RDD} \arrow[d] \\ \text{Mapped RDD} \arrow[d] \\ \text{Reduced RDD} \end{tikzcd}$$

Apache Spark

© Copyright IBM Corporation 2018

Resilient Distributed Datasets (RDDs)

Looking at the core of Spark, Spark's primary core abstraction is called Resilient Distributed Dataset (RDD).

RDD essentially is just a distributed collection of elements that is parallelized across the cluster.

There are two types of RDD operations: transformations and actions.

- Transformations do not return a value. In fact, nothing is evaluated during the definition of transformation statements. Spark just creates the definition of a transformation that will be evaluated later at runtime. This is called this lazy evaluation. The transformation is stored as a directed acyclic graphs (DAG).
- Actions actually perform the evaluation. The action is transformations are performed along with the work that is called for to consume or produce RDDs. Actions return values. For example, you can do a count on a RDD, to get the number of elements within and that value is returned.

The fault tolerance aspect of RDDs allows Spark to reconstruct the transformations used to build the lineage to get back any lost data.

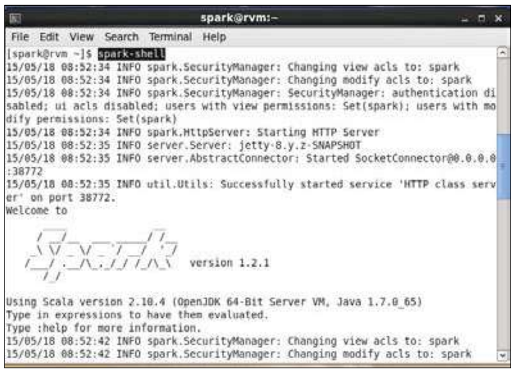
In the example RDD flow shown on the slide, the first step loads the dataset from Hadoop. Successive steps apply transformations on this data such as filter, map, or reduce. Nothing actually happens until an action is called. The DAG is just updated each time until an action is called. This provides fault tolerance; for example, if a node goes offline, all it needs to do when it comes back online is to re-evaluate the graph to where it left off.

In-memory caching is provided with Spark to enable the processing to happen in memory. If the RDD does not fit in memory, it will spill to disk.

IBM Training

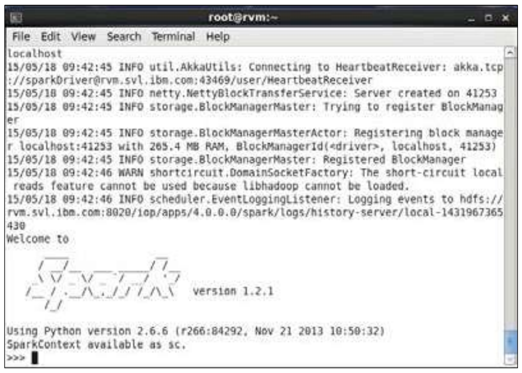
Spark jobs and shell

- Spark jobs can be written in Scala, Python, or Java. APIs are available for all three: http://spark.apache.org/docs/1.6.3/
- Spark shells are provided for Scala (spark-shell) and Python (pyspark)
- Spark's native language is Scala, so it is natural to write Spark applications using Scala



```
[spark@rvm ~]$ spark-shell
15/05/18 08:52:34 INFO spark.SecurityManager: Changing view acls to: spark
15/05/18 08:52:34 INFO spark.SecurityManager: Changing modify acls to: spark
15/05/18 08:52:34 INFO spark.SecurityManager: SecurityManager: authentication di
sabled; ui acls disabled; users with view permissions: Set(spark); users with mo
dify permissions: Set(spark)
15/05/18 08:52:34 INFO spark.HttpServer: Starting HTTP Server
15/05/18 08:52:35 INFO server.Server: jetty-8.y.z-SNAPSHOT
15/05/18 08:52:35 INFO server.AbstractConnector: Started SocketConnector@0.0.0.0
:38772
15/05/18 08:52:35 INFO util.Utils: Successfully started service 'HTTP class serv
er' on port 38772.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_65)
Type in expressions to have them evaluated.
Type :help for more information.
15/05/18 08:52:42 INFO spark.SecurityManager: Changing view acls to: spark
15/05/18 08:52:42 INFO spark.SecurityManager: Changing modify acls to: spark
```



```
localhost
15/05/18 09:42:45 INFO util.AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp
://sparkDriver@rvm.svl.ibm.com:43469/user/HeartbeatReceiver
15/05/18 09:42:45 INFO netty.NettyBlockTransferService: Server created on 41253
15/05/18 09:42:45 INFO storage.BlockManagerMaster: Trying to register BlockManag
er
15/05/18 09:42:45 INFO storage.BlockManagerMasterActor: Registering block manage
r localhost:41253 with 265.4 MB RAM, BlockManagerId(<driver>, localhost, 41253)
15/05/18 09:42:45 INFO storage.BlockManagerMaster: Registered BlockManager
15/05/18 09:42:46 WARN shortcircuit.DomainSocketFactory: The short-circuit local
 reads feature cannot be used because libhadoop cannot be loaded.
15/05/18 09:42:46 INFO scheduler.EventLoggingListener: Logging events to hdfs://
rvm.svl.ibm.com:8020/iop/apps/4.0.0.0/spark/logs/history-server/local-1431967365
430
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

Using Python version 2.6.6 (r266:84292, Nov 21 2013 10:50:32)
SparkContext available as sc.
>>> 
```

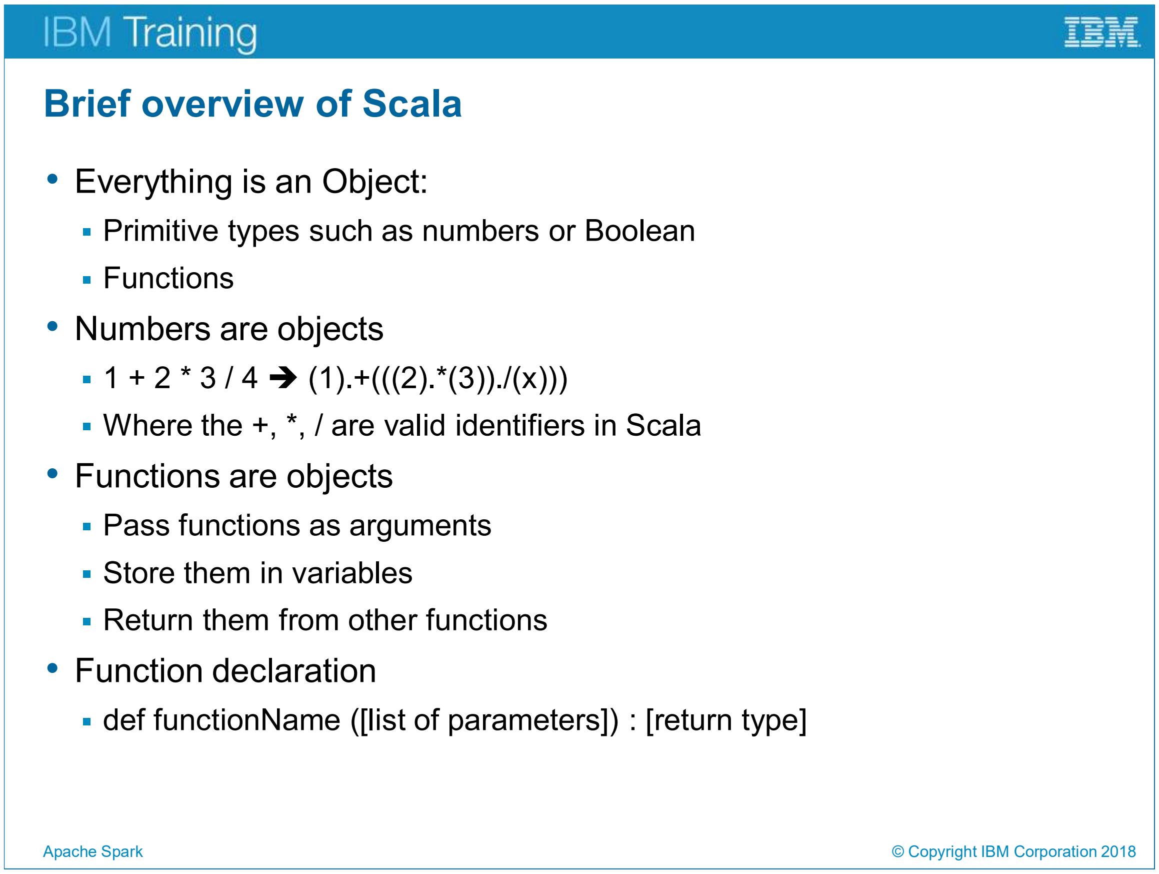
Apache Spark

Spark jobs and shell

Spark jobs can be written in Scala, Python or Java. Spark shells are available for Scala (spark-shell) and Python (pyspark). This course will not teach how to program in each specific language, but will cover how to use them within the context of Spark. It is recommended that you have at least some programming background to understand how to code in any of these.

If you are setting up the Spark cluster yourself, you will have to make sure that you have a compatible version of it. This information can be found on Spark's website. In the lab environment, everything has been set up for you - all you do is launch up the shell and you are ready to go.

Spark itself is written in the Scala language, so it is natural to use Scala to write Spark applications. This course will cover code examples from by Scala, Python, and Java. Java 8 actually supports the functional programming style to include lambdas, which concisely captures the functionality that are executed by the Spark engine. This bridges the gap between Java and Scala for developing applications on Spark. Java 6 and 7 is supported, but would require more work and an additional library to get the same amount of functionality as you would using Scala or Python.



IBM Training

Brief overview of Scala

- Everything is an Object:
  - Primitive types such as numbers or Boolean
  - Functions
- Numbers are objects
  - 1 + 2 * 3 / 4 ➔ (1).+(((2).*(3))./(x)))
  - Where the +, *, / are valid identifiers in Scala
- Functions are objects
  - Pass functions as arguments
  - Store them in variables
  - Return them from other functions
- Function declaration
  - def functionName ([list of parameters]) : [return type]

Apache Spark

© Copyright IBM Corporation 2018

Brief overview of Scala

Everything in Scala is an object. The primitive types defined by Java such as int or boolean are objects in Scala. Functions are objects in Scala and play an important role in how applications are written for Spark.

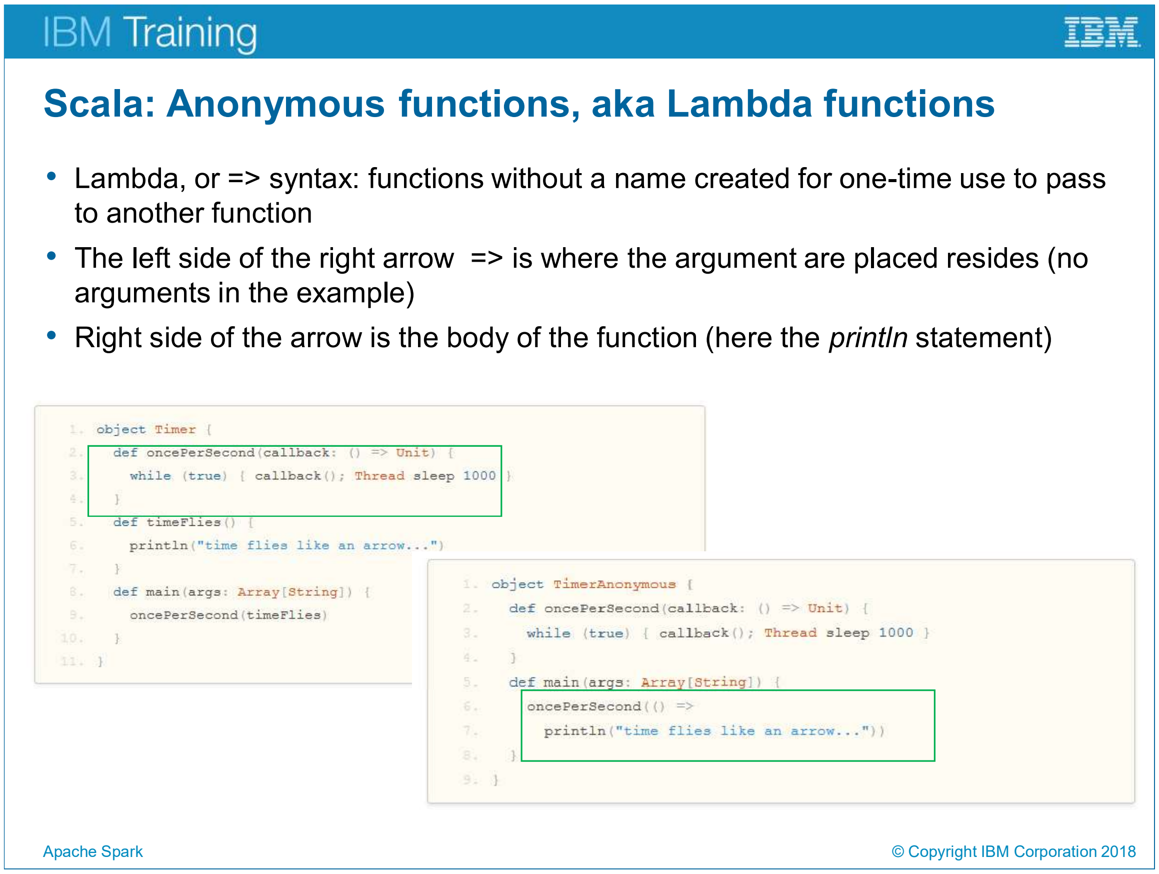
Numbers are objects. As an example, in the expression that you see here: 1 + 2 * 3 / 4 actually means that the individual numbers invoke the various identifiers +,-,*,/ with the other numbers passed in as arguments using the dot notation.

Functions are objects. You can pass functions as arguments into another function. You can store them as variables. You can return them from other functions. The function declaration is the function name followed by the list of parameters and then the return type.

If you want to learn more about Scala, check out its website for tutorials and guide. Throughout this course, you will see examples in Scala that will have explanations on what it does. Remember, the focus of this unit is on the context of Spark, and is not intended to teach Scala, Python or Java.

References for learning Scala:

- Horstmann, C. S. (2012). *Scala for the impatient*. Upper Saddle River, NJ: Addison-Wesley Professional
- Odersky, M., Spoon, L., & Venners, B. (2011). *Programming in Scala: A Comprehensive Step-by-Step Guide* (2nd ed.). Walnut Creek, CA: Artima Press



IBM Training

Scala: Anonymous functions, aka Lambda functions

- Lambda, or => syntax: functions without a name created for one-time use to pass to another function
- The left side of the right arrow => is where the argument are placed resides (no arguments in the example)
- Right side of the arrow is the body of the function (here the *println* statement)

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

Apache Spark

© Copyright IBM Corporation 2018

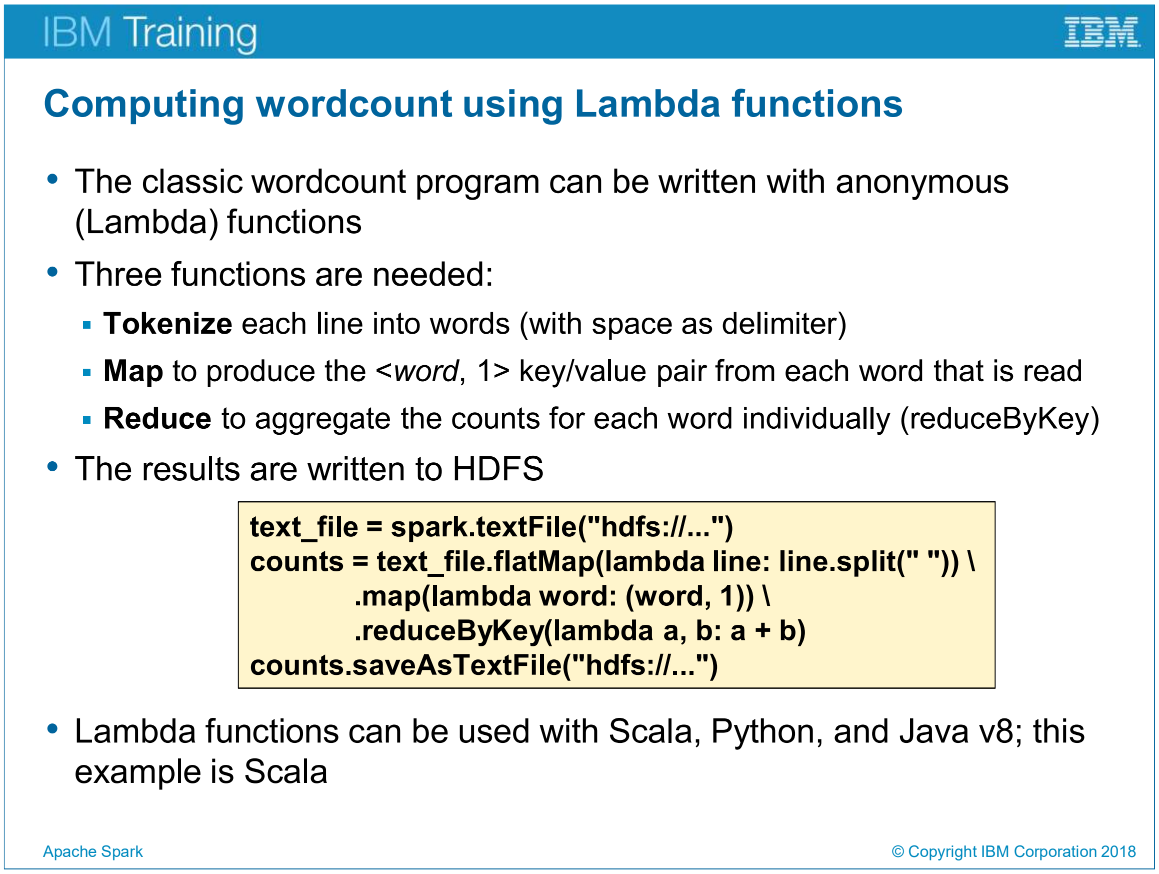
Scala: Anonymous functions, aka Lambda functions

Anonymous functions are very common in Spark applications. Essentially, if the function you need is only going to be required once, there is really no value in naming it. Just use it anonymously on the go and forget about it. For example, if you have a timeFlies function and in it, you just print a statement to the console. In another function, oncePerSecond, you need to call this timeFlies function. Without anonymous functions, you would code it like the top example by defining the timeFlies function. Using the anonymous function capability, you just provide the function with arguments, the right arrow, and the body of the function after the right arrow as in the bottom example. Because this is the only place you will be using this function, you do not need to name the function.'

Python's syntax is relatively convenient and easy to work with, but aside from the basic structure of the language Python is also sprinkled with small syntax structures that make certain tasks especially convenient. The lambda keyword/function construct is one of these, where the creators call it "syntactical candy."

References:

- http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html



IBM Training

Computing wordcount using Lambda functions

- The classic wordcount program can be written with anonymous (Lambda) functions
- Three functions are needed:
  - **Tokenize** each line into words (with space as delimiter)
  - **Map** to produce the <*word*, 1> key/value pair from each word that is read
  - **Reduce** to aggregate the counts for each word individually (reduceByKey)
- The results are written to HDFS

```
text_file = spark.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
          .map(lambda word: (word, 1)) \
          .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

- Lambda functions can be used with Scala, Python, and Java v8; this example is Scala

Apache Spark © Copyright IBM Corporation 2018

Computing wordcount using Lambda functions

The Lambda or => syntax is a shorthand way to define functions inline in Python and Scala. With Spark you can define anonymous functions separately and then pass the name to Spark. For example, in Python:

```
def hadHDP(line):

       return "HDP" in line;

HDPLines = lines.filter(hasHDP)
```

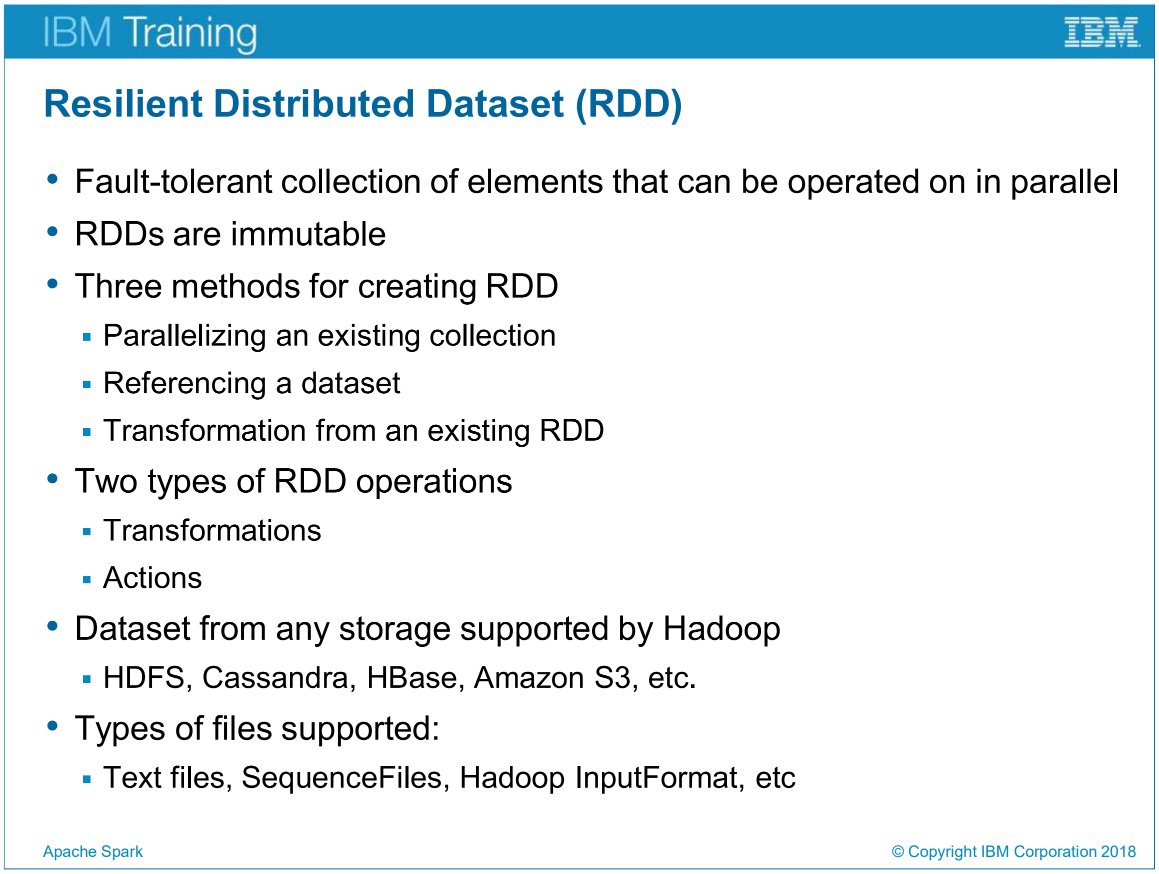
…is functionally equivalent to:

```
grep HDP inputfile
```

A common example is a MapReduce wordcount. You first split up the file by words ("tokenize") and then map each word into a key value pair with the word as the key, and the value of 1. Then you reduce by the key, which adds up all the value of the same key, effectively, counting the number of occurrences of that key. Finally, the counts are written out to a file in HDFS.

References:

- École Polytechnique Fédérale de Lausanne (EPFL). (2012). A tour of Scala: Anonymous function syntax, from http://www.scala-lang.org/old/node/133
- Apache Software Foundation,(2015). Spark Examples, from https://spark.apache.org/examples.html


IBM Training

Resilient Distributed Dataset (RDD)

- Fault-tolerant collection of elements that can be operated on in parallel
- RDDs are immutable
- Three methods for creating RDD
  - Parallelizing an existing collection
  - Referencing a dataset
  - Transformation from an existing RDD
- Two types of RDD operations
  - Transformations
  - Actions
- Dataset from any storage supported by Hadoop
  - HDFS, Cassandra, HBase, Amazon S3, etc.
- Types of files supported:
  - Text files, SequenceFiles, Hadoop InputFormat, etc

Apache Spark

© Copyright IBM Corporation 2018

Resilient Distributed Dataset (RDD)

Major points of interest here:

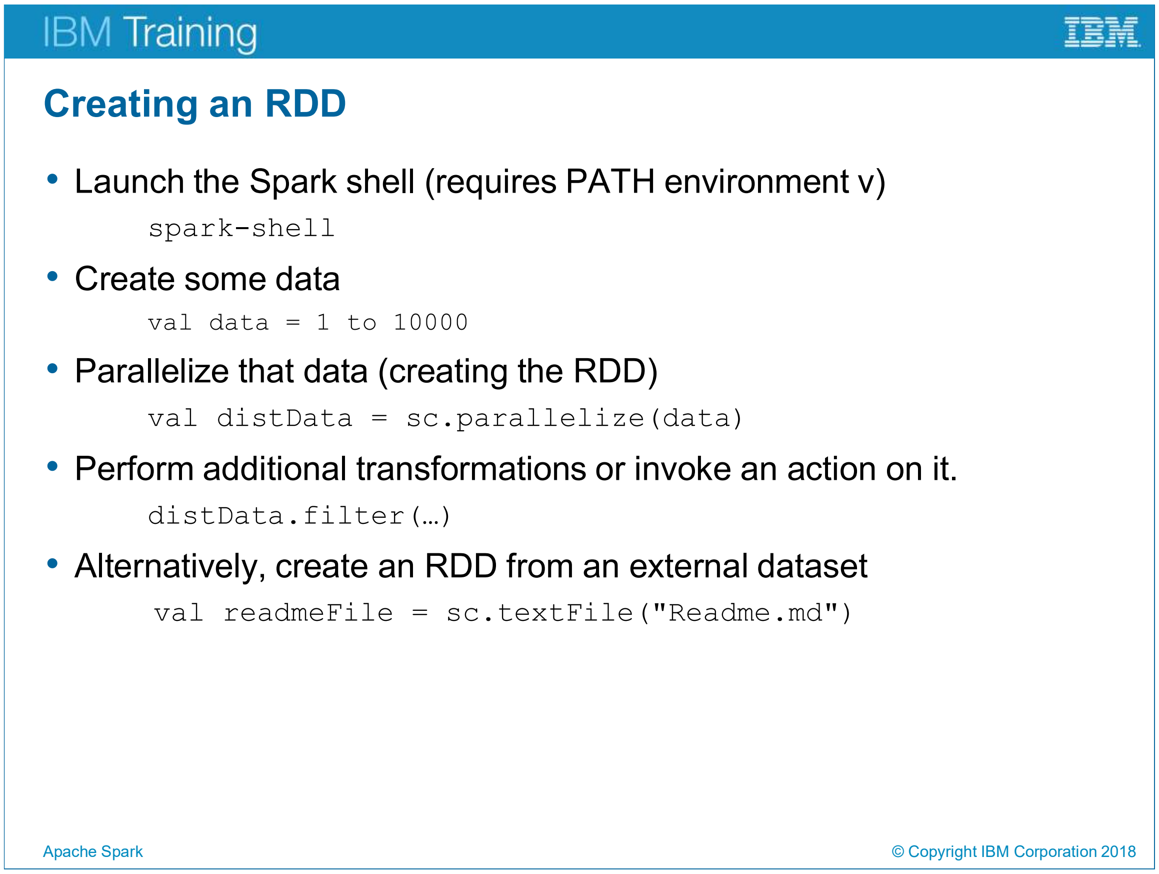
- Resilient Distributed Dataset (RDD) is Spark's primary abstraction.
- An RDD is a fault tolerant collection of elements that can be parallelized. In other words, they can be made to be operated on in parallel. They are immutable.
- These are the fundamental primary units of data in Spark.
- When RDDs are created, a direct acyclic graph (DAG) is created. This type of operation is called transformations. Transformations makes updates to that graph, but nothing actually happens until some action is called. Actions are another type of operations. We'll talk more about this shortly. The notion here is that the graphs can be replayed on nodes that need to get back to the state it was before it went offline, thus providing fault tolerance. The elements of the RDD can be operated on in parallel across the cluster. Remember, transformations return a pointer to the RDD created and actions return values that comes from the action.

There are three methods for creating a RDD. You can parallelize an existing collection. This means that the data already resides within Spark and can now be operated on in parallel. As an example, if you have an array of data, you can create a RDD out of it by calling the parallelized method. This method returns a pointer to the RDD. So this new distributed dataset can now be operated upon in parallel throughout the cluster.

The second method to create a RDD, is to reference a dataset. This dataset can come from any storage source supported by Hadoop such as HDFS, Cassandra, HBase, Amazon S3, etc.

The third method to create a RDD is from transforming an existing RDD to create a new RDD. In other words, if you have the array of data that you parallelized earlier, and you want to filter out the records available. A new RDD is created using the filter method.

The final point on this slide: Spark supports text files, SequenceFiles, and any other Hadoop InputFormat.



IBM Training

Creating an RDD

- Launch the Spark shell (requires PATH environment v)

```
spark-shell
```

- Create some data

```
val data = 1 to 10000
```

- Parallelize that data (creating the RDD)

```
val distData = sc.parallelize(data)
```

- Perform additional transformations or invoke an action on it.

```
distData.filter(…)
```

- Alternatively, create an RDD from an external dataset

```
val readmeFile = sc.textFile("Readme.md")
```

Apache Spark © Copyright IBM Corporation 2018

Creating an RDD

Here is a quick example of how to create an RDD from an existing collection of data. In the examples throughout the course, unless otherwise indicated, you will be using Scala to show how Spark works. In the lab exercises, you will get to work with Python and Java as well.

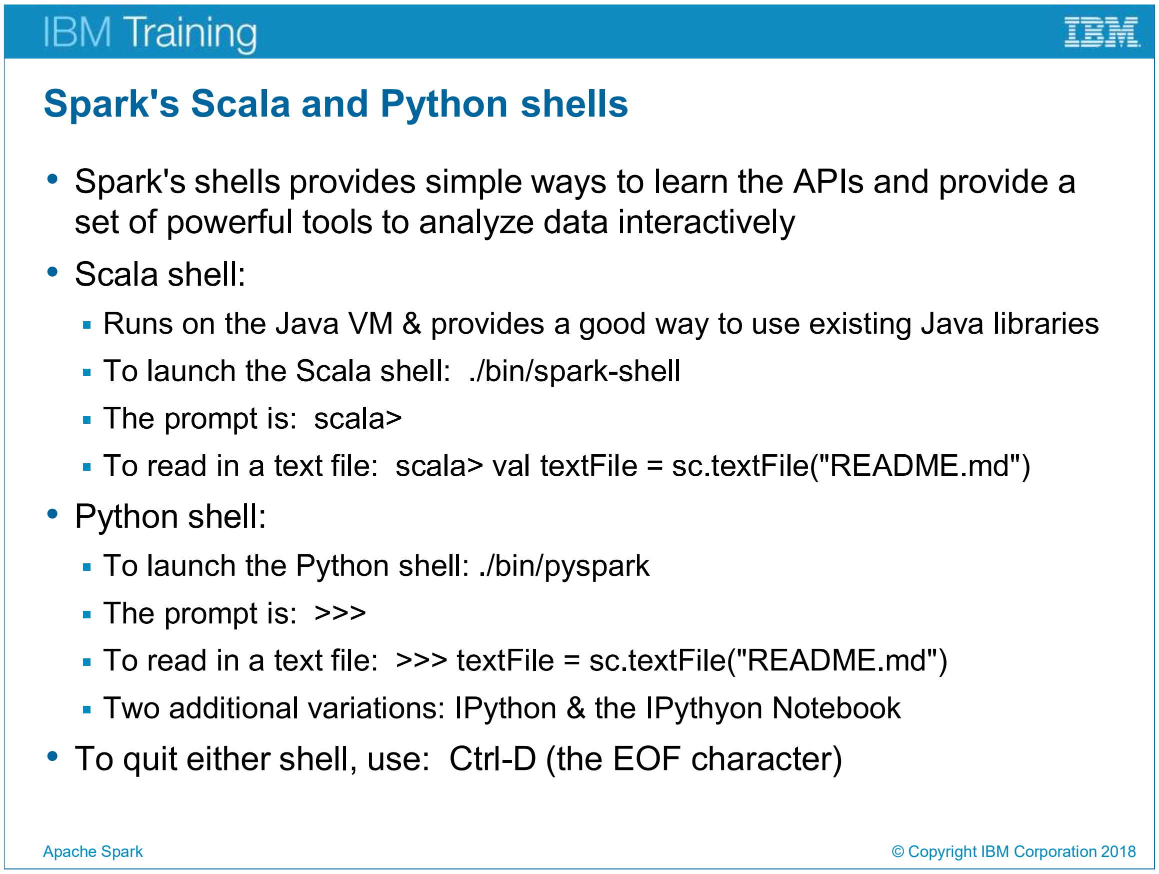
First, launch the Spark shell. This command is located under the /usr/bin directory.

Once the shell is up (with the prompt "scala>"), create some data with values from 1 to 10,000. Then, create an RDD from that data using the parallelize method from the SparkContext, shown as sc on the slide; this means that the data can now be operated on in parallel.

More will be covered on the SparkContext, the sc object that is invoking the parallelized function later, so for now, just know that when you initialize a shell, the SparkContext, sc, is initialized for you to use.

The parallelize method returns a pointer to the RDD. Remember, transformations operations such as parallelize, only returns a pointer to the RDD. It actually will not create that RDD until some action is invoked on it. With this new RDD, you can perform additional transformations or actions on it such as the filter transformation. This is important with large amounts of data (big data), because you do not want to duplicate the date until needed, and certainly not cache it in memory until needed.

Another way to create an RDD is from an external dataset. In the example here, you are creating a RDD from a text file using the textFile method of the SparkContext object. You will see more examples of how to create RDD throughout this course.



IBM Training

Spark's Scala and Python shells

- Spark's shells provides simple ways to learn the APIs and provide a set of powerful tools to analyze data interactively
- Scala shell:
  - Runs on the Java VM & provides a good way to use existing Java libraries
  - To launch the Scala shell: ./bin/spark-shell
  - The prompt is: scala>
  - To read in a text file: scala> val textFile = sc.textFile("README.md")
- Python shell:
  - To launch the Python shell: ./bin/pyspark
  - The prompt is: >>>
  - To read in a text file: >>> textFile = sc.textFile("README.md")
  - Two additional variations: IPython & the IPythyon Notebook
- To quit either shell, use: Ctrl-D (the EOF character)

Apache Spark

© Copyright IBM Corporation 2018

Spark's Scala and Python shells

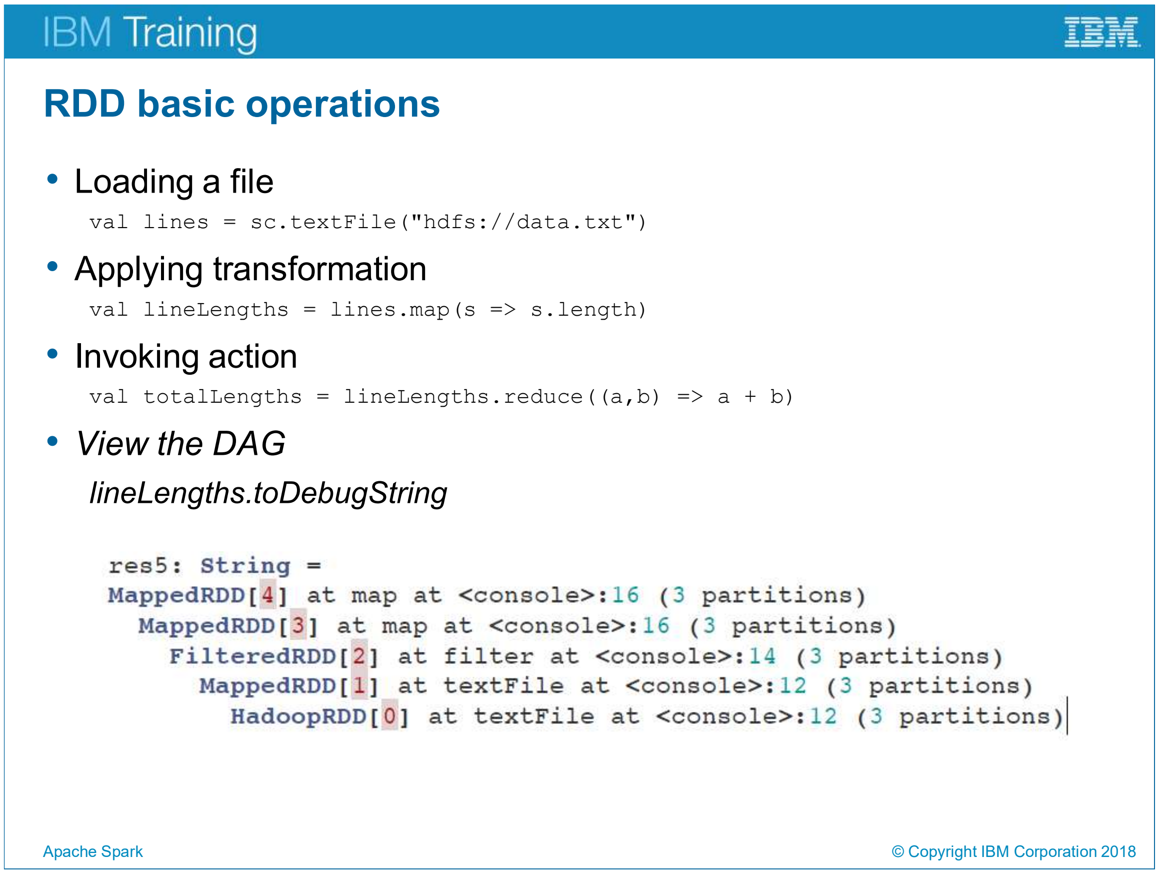
The Spark shell provides a simple way to learn Spark's API. It is also a powerful tool analyze data interactively. The Shell is available in either Scala, which runs on the Java VM, or Python. To start up Scala, execute the command spark-shell from within the Spark's bin directory. To create a RDD from a text file, invoke the textFile method with the **sc** object, which is the SparkContext.

To start up the shell for Python, you would execute the pyspark command from the same bin directory. Then, invoking the textFile command will also create a RDD for that text file.

In the lab exercise later, you will start up either of the shells and run a series of RDD transformations and actions to get a feel of how to work with Spark. Later you will get to dive deeper into RDDs.

IPython offers features such as tab completion. For further details: http://ipython.org

IPython Notebook is a web-browser based version of IPython.


IBM Training

RDD basic operations

- Loading a file

```
val lines = sc.textFile("hdfs://data.txt")
```

- Applying transformation

```
val lineLengths = lines.map(s => s.length)
```

- Invoking action

```
val totalLengths = lineLengths.reduce((a,b) => a + b)
```

- *View the DAG*

lineLengths.toDebugString

```
res5: String =
MappedRDD[4] at map at <console>:16 (3 partitions)
  MappedRDD[3] at map at <console>:16 (3 partitions)
    FilteredRDD[2] at filter at <console>:14 (3 partitions)
      MappedRDD[1] at textFile at <console>:12 (3 partitions)
        HadoopRDD[0] at textFile at <console>:12 (3 partitions)
```

Apache Spark © Copyright IBM Corporation 2018

RDD basic operations

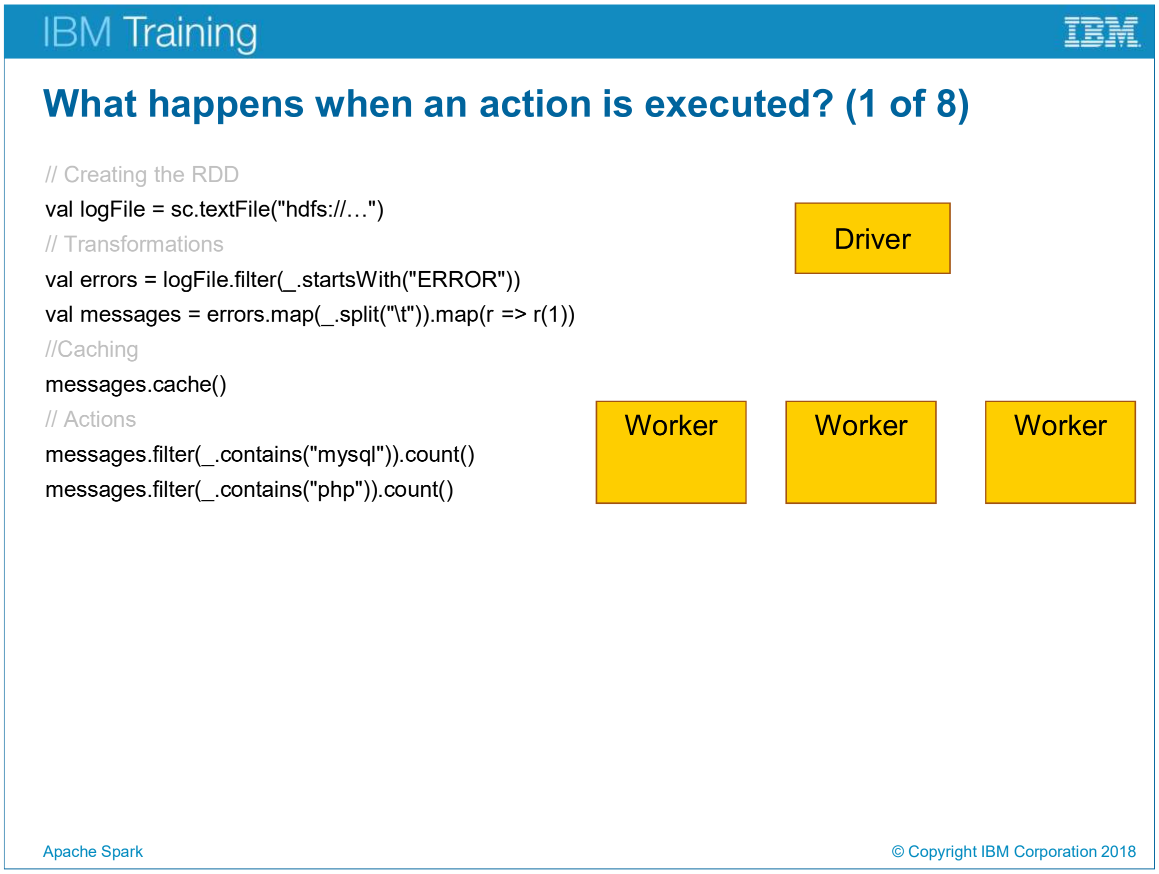
You have seen how to load a file from an external dataset. This time, however, you are loading a file from the hdfs. Loading the file creates a RDD, which is only a pointer to the file. The dataset is not loaded into memory yet. Nothing will happen until some action is called. The transformation basically updates the direct acyclic graph (DAG).

So the transformation here is saying map each line - ‘s’ - to the length of that line. Then, the action operation is reducing it to get the total length of all the lines. When the action is called, Spark goes through the DAG and applies all the transformation up until that point, followed by the action and then a value is returned back to the caller.

Directed Acyclic Graph (DAG)

On this slide, you will see how to view the DAG of any particular RDD. A DAG is essentially a graph of the business logic and does not get executed until an action is called - often called lazy evaluation.

To view the DAG of a RDD after a series of transformation, use the toDebugString method as you see here on the slide. It will display the series of transformation that Spark will go through once an action is called. You read it from the bottom up. In the sample DAG shown on the slide, you can see that it starts as a textFile and goes through a series of transformation such as map and filter, followed by more map operations. Remember, that it is this behavior that allows for fault tolerance. If a node goes offline and comes back on, all it has to do is just grab a copy of this from a neighboring node and rebuild the graph back to where it was before it went offline.



IBM Training

What happens when an action is executed? (1 of 8)

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Driver
Worker
Worker
Worker

Apache Spark

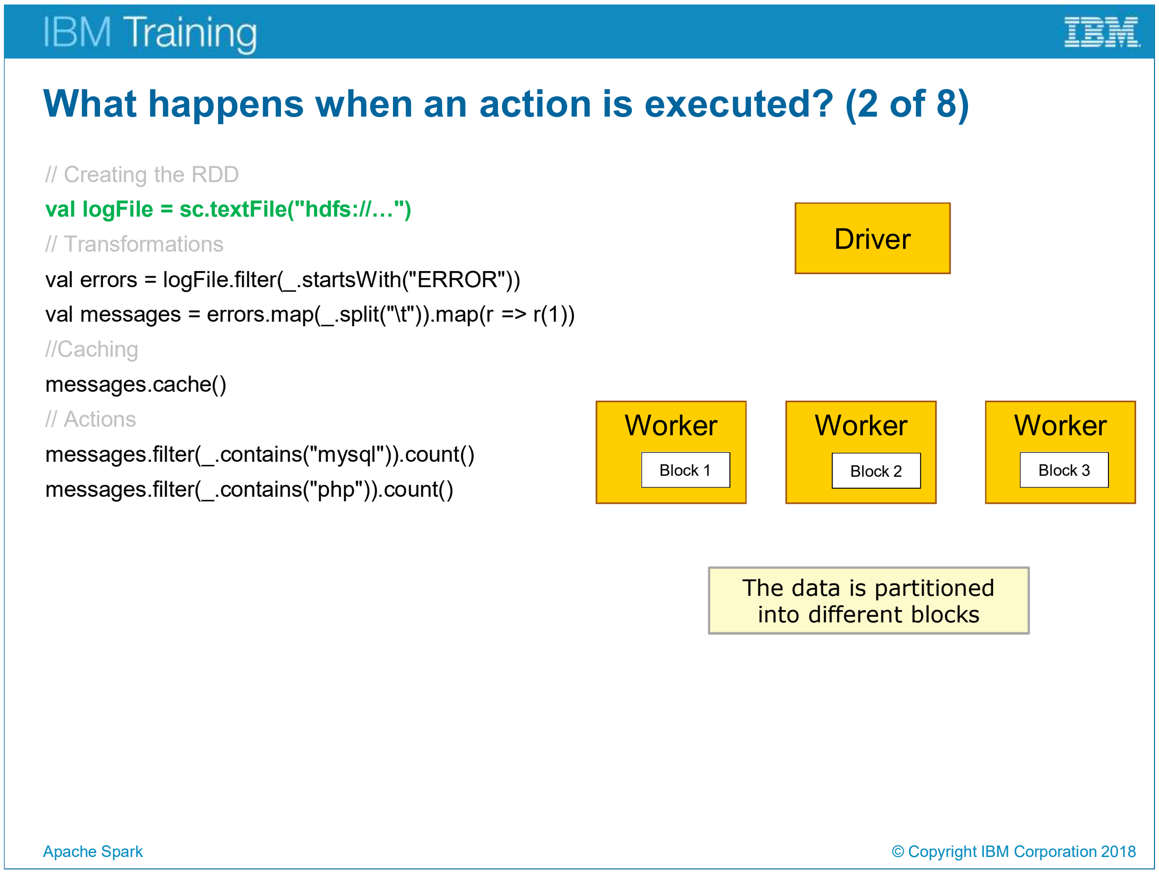
© Copyright IBM Corporation 2018

What happens when an action is executed?

In the next several slides, you will see at a high level what happens when an action is executed.

Let's look at the code first. The goal here is to analyze some log files. The first line you load the log from the hadoop file system. The next two lines you filter out the messages within the log errors. Before you invoke some action on it, you tell it to cache the filtered dataset; it doesn't actually cache it yet as nothing has been done up until this point.

Then you do more filters to get specific error messages relating to mysql and php followed by the count action to find out how many errors were related to each of those filters.



IBM Training

What happens when an action is executed? (2 of 8)

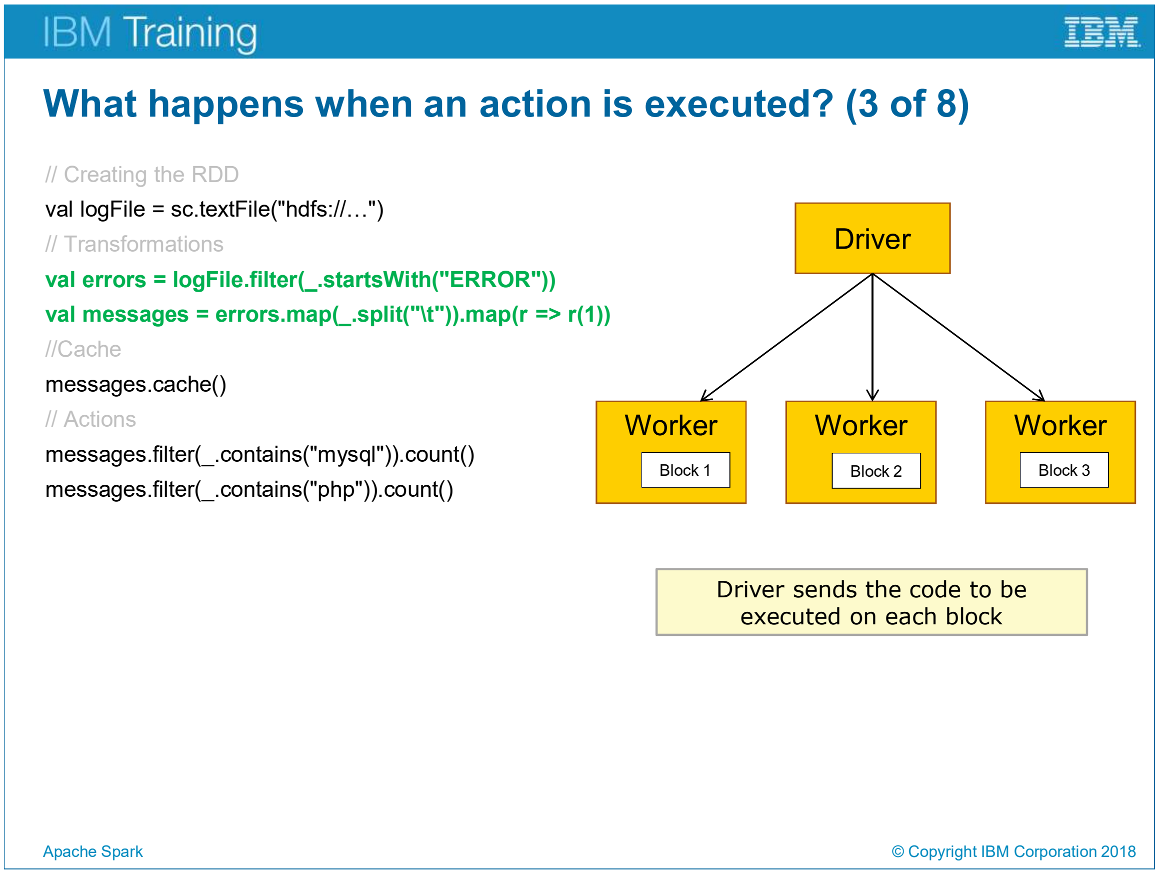
```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

The data is partitioned into different blocks

Apache Spark

© Copyright IBM Corporation 2018

In reviewing the steps, the first thing that happens when you load in the text file is the data is partitioned into different blocks across the cluster.



IBM Training

What happens when an action is executed? (3 of 8)

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Cache
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

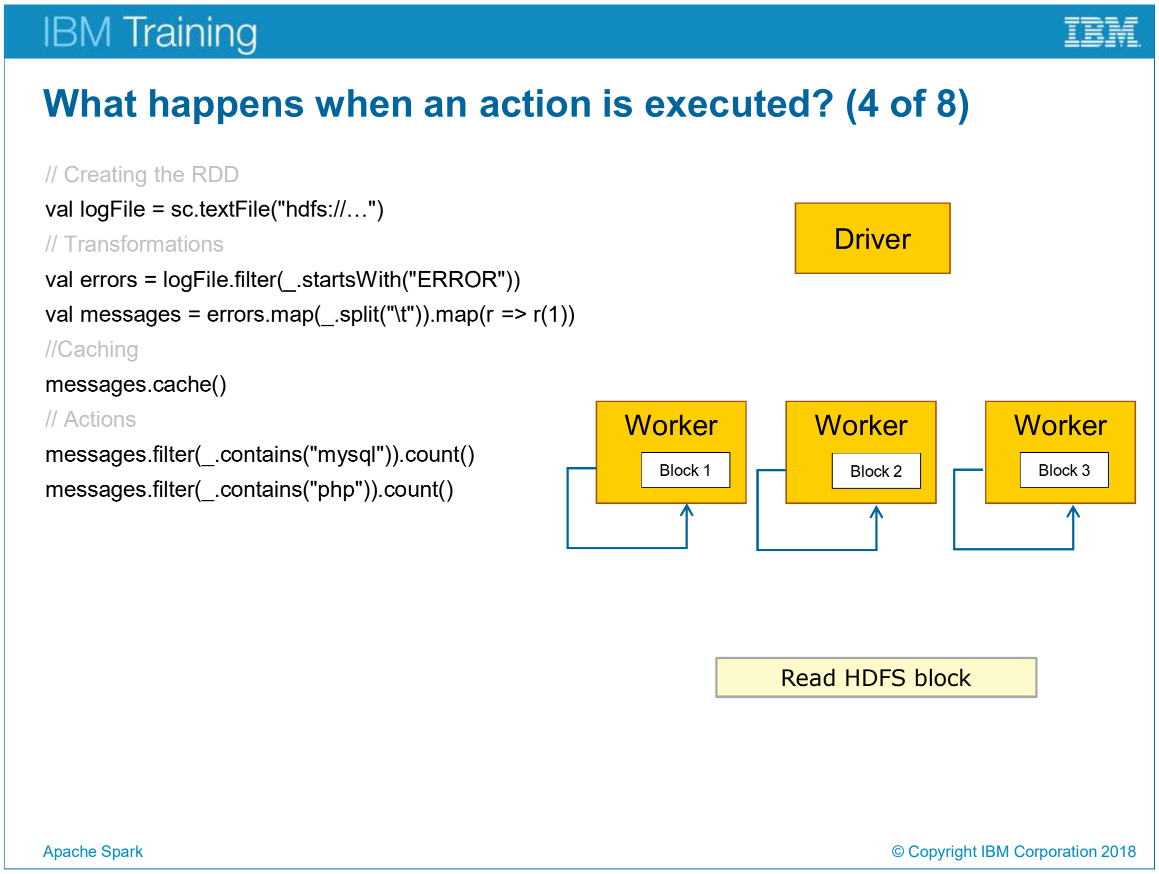
Driver
Worker
Block 1
Worker
Block 2
Worker
Block 3

Driver sends the code to be executed on each block

Apache Spark

© Copyright IBM Corporation 2018

Then the driver sends the code to be executed on each block. In the example, it would be the various transformations and actions that will be sent out to the workers. Actually, it is the executor on each workers that is going to be performing the work on each block. You will see a bit more on executors later.



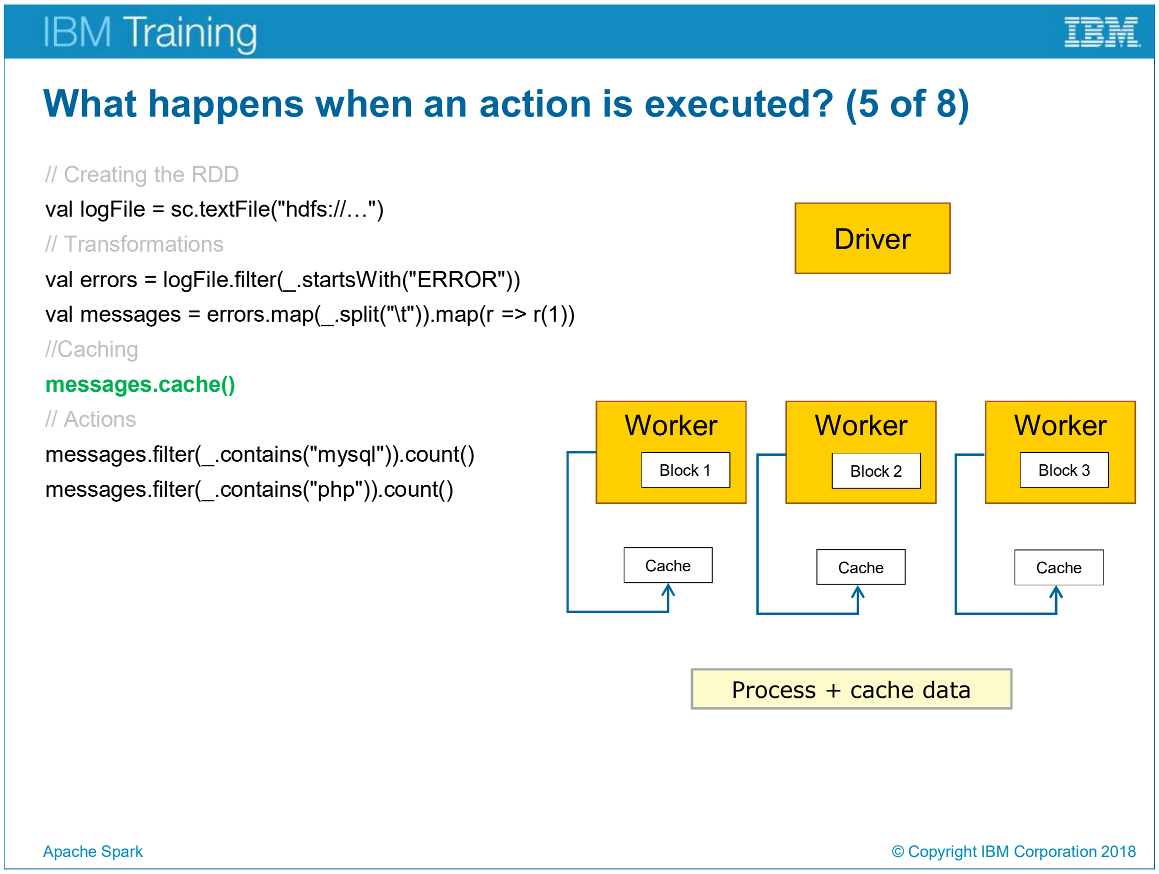
IBM Training

What happens when an action is executed? (4 of 8)

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Apache Spark © Copyright IBM Corporation 2018

The executors read the HDFS blocks to prepare the data for the operations in parallel.



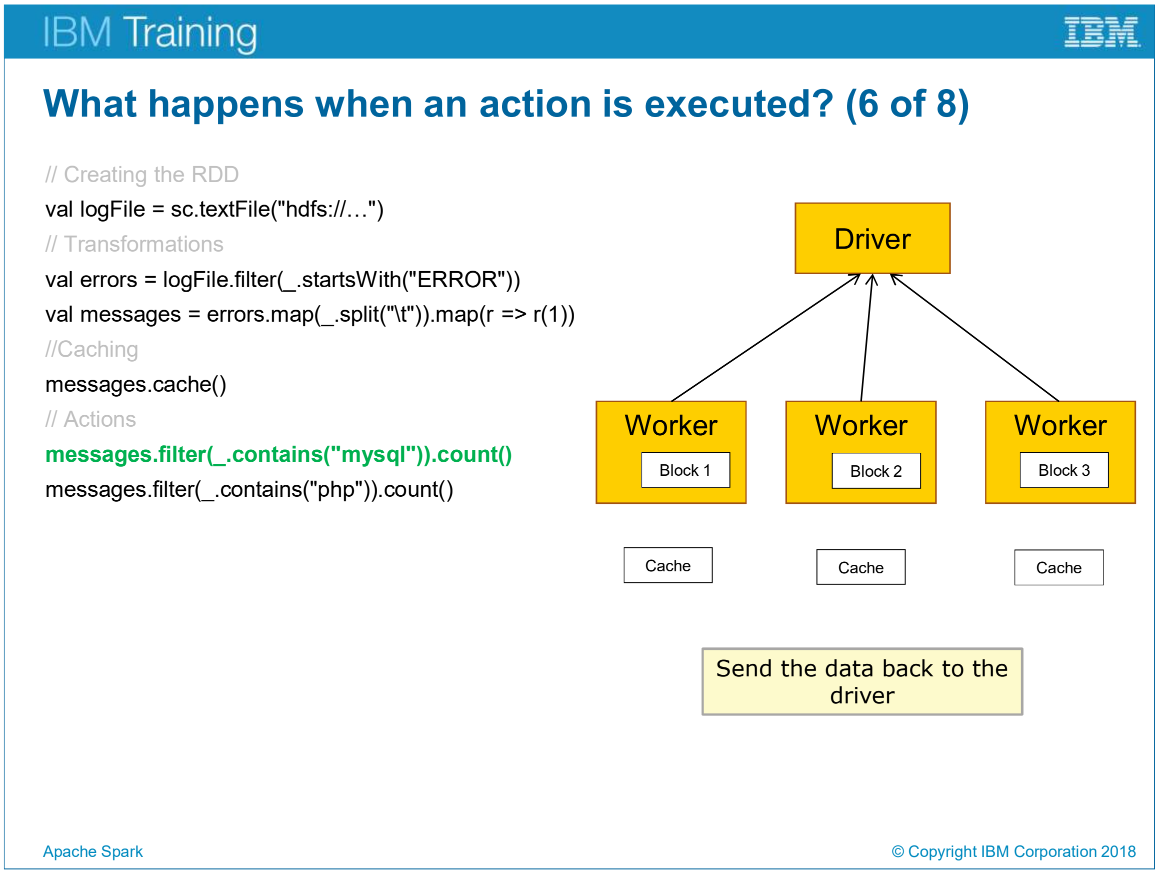
IBM Training

What happens when an action is executed? (5 of 8)

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Apache Spark © Copyright IBM Corporation 2018

After a series of transformations, you want to cache the results up until that point into memory. A cache is created.



IBM Training

What happens when an action is executed? (6 of 8)

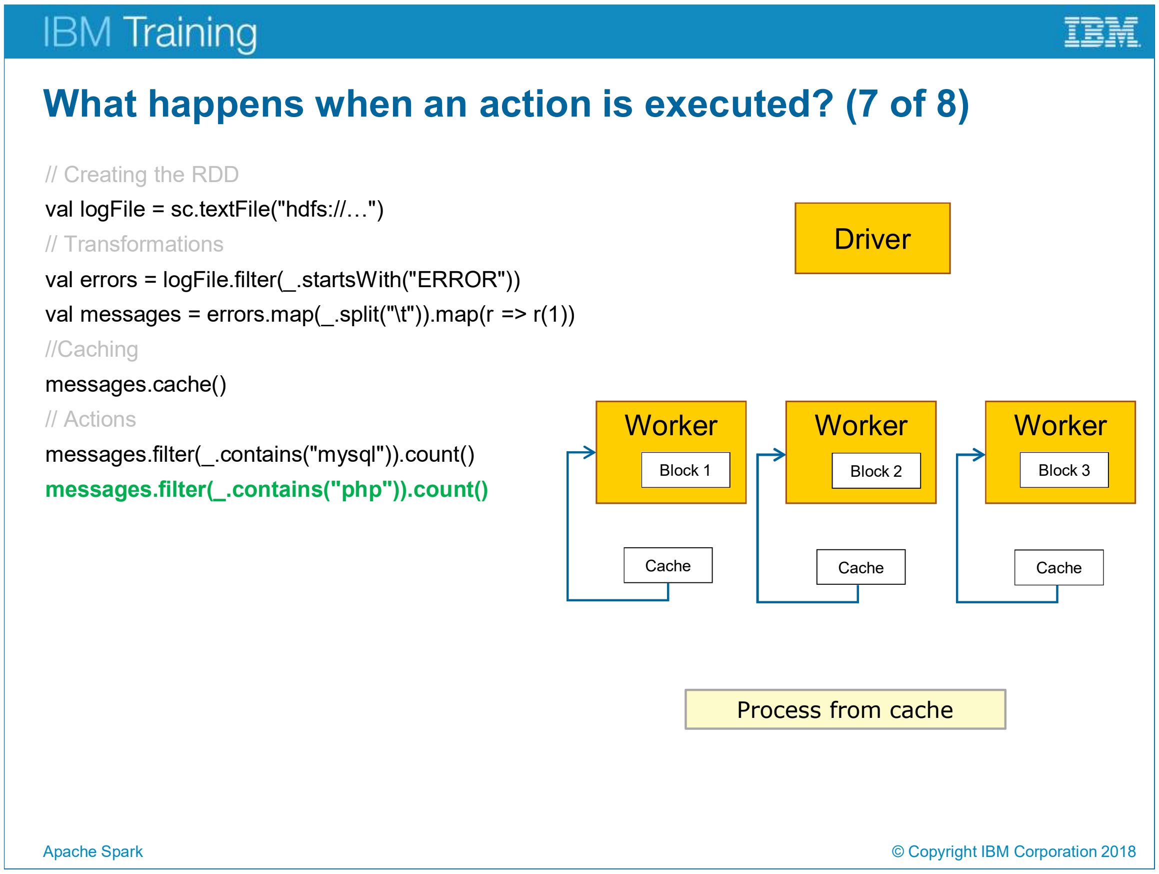
```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Driver
Worker
Block 1
Worker
Block 2
Worker
Block 3
Cache
Cache
Cache
Send the data back to the driver

Apache Spark

© Copyright IBM Corporation 2018

After the first action completes, the results are sent back to the driver. In this case, you are looking for messages that relate to mysql. This is then returned back to the driver.



IBM Training

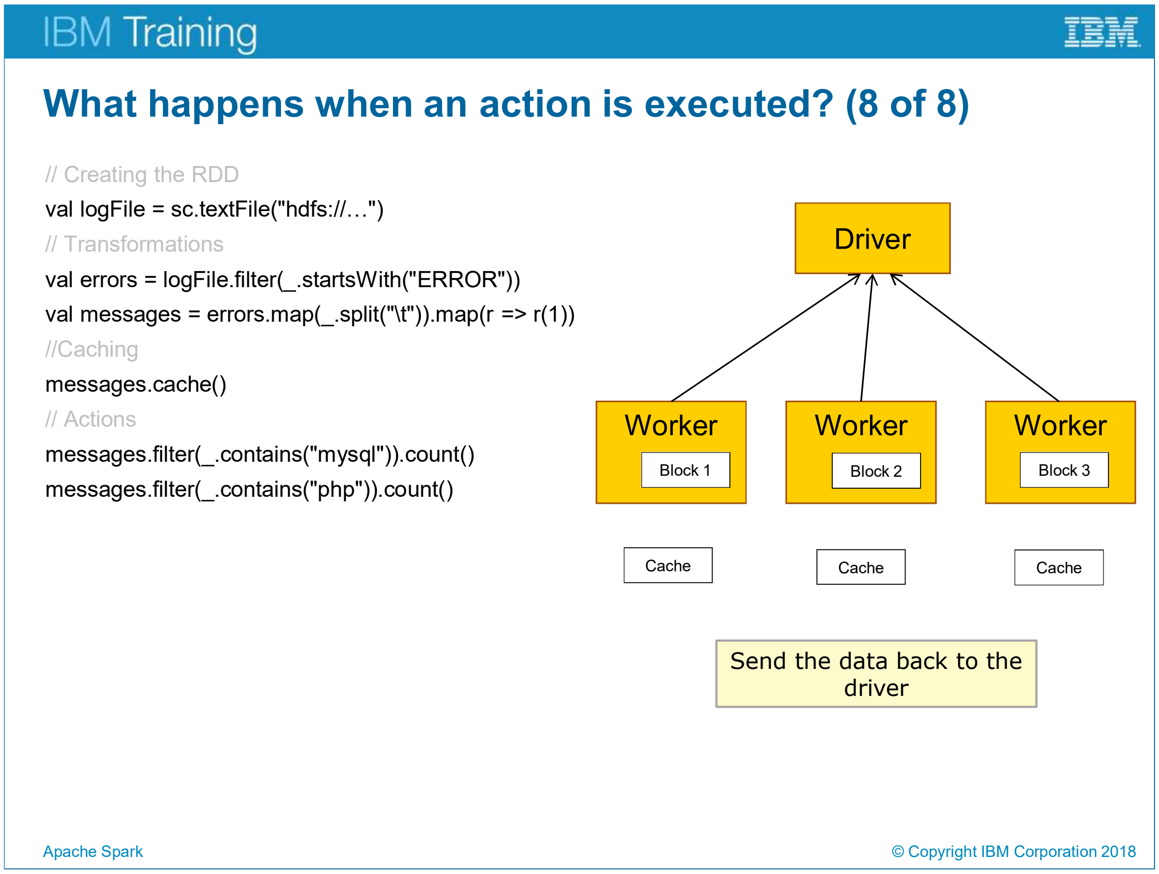
What happens when an action is executed? (7 of 8)

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Apache Spark

© Copyright IBM Corporation 2018

To process the second action, Spark will use the data on the cache; it does not need to go to the HDFS data again. It just reads it from the cache and processes the data from there.



IBM Training

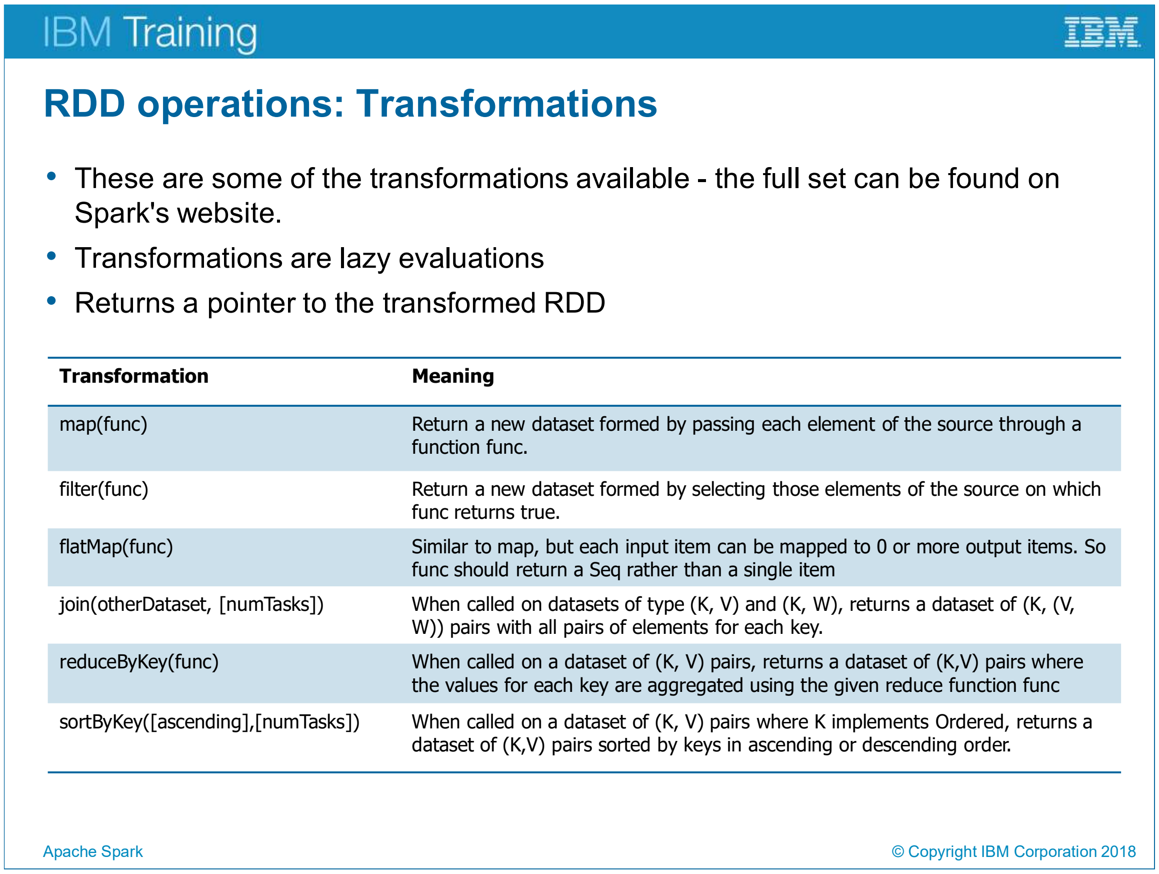
What happens when an action is executed? (8 of 8)

```
// Creating the RDD
val logFile = sc.textFile("hdfs://…")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Apache Spark

© Copyright IBM Corporation 2018

Finally the results are sent back to the driver and you have completed a full cycle.



IBM Training

RDD operations: Transformations

- These are some of the transformations available - the full set can be found on Spark's website.
- Transformations are lazy evaluations
- Returns a pointer to the transformed RDD

Transformation	Meaning
map(func)	Return a new dataset formed by passing each element of the source through a function func.
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
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.
reduceByKey(func)	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 func
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.

Apache Spark

© Copyright IBM Corporation 2018

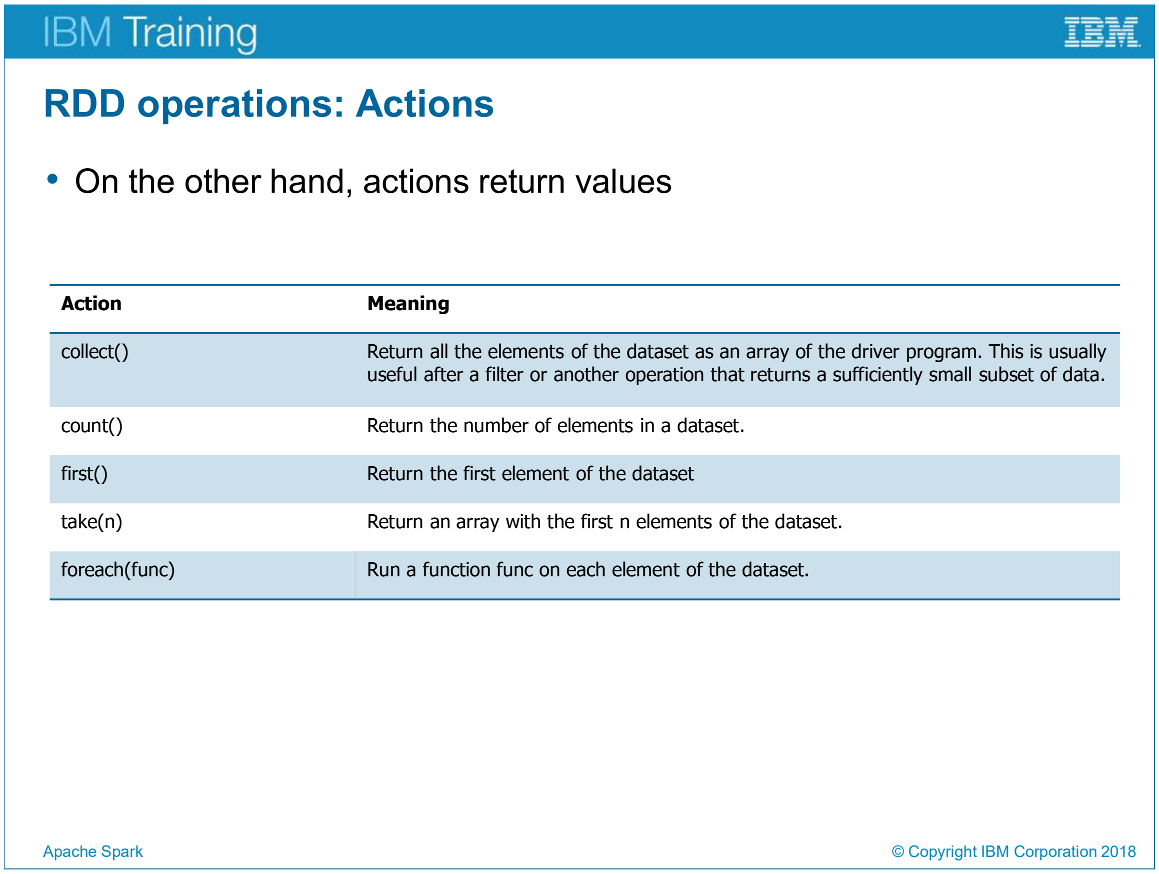
RDD operations: Transformations

These are some of the transformations available; the full set can be found on Spark's website. The Spark Programming Guide can be found at https://spark.apache.org/docs/latest/programming-guide.html and transformations can be found at https://spark.apache.org/docs/latest/programming-guide.html#transformations.

Remember that Transformations are essentially lazy evaluations. Nothing is executed until an action is called. Each transformation function basically updates the graph and when an action is called, the graph is executed. Transformation returns a pointer to the new RDD.

Some of the less obvious are:

- The **flatMap** function is similar to map, but each input can be mapped to 0 or more output items. What this means is that the returned pointer of the func method, should return a sequence of objects, rather than a single item. It would mean that the flatMap would flatten a list of lists for the operations that follows. Basically this would be used for MapReduce operations where you might have a text file and each time a line is read in, you split that line up by spaces to get individual keywords. Each of those lines ultimately is flatten so that you can perform the map operation on it to map each keyword to the value of one.
- The **join** function combines two sets of key value pairs and return a set of keys to a pair of values from the two initial set. For example, you have a K,V pair and a K,W pair. When you join them together, you will get a K, (V,W) set.
- The **reduceByKey** function aggregates on each key by using the given reduce function. This is something you would use in a WordCount to sum up the values for each word to count its occurrences.



IBM Training

RDD operations: Actions

- On the other hand, actions return values

Action	Meaning
collect()	Return all the elements of the dataset as an array of the driver program. This is usually useful after a filter or another operation that returns a sufficiently small subset of data.
count()	Return the number of elements in a dataset.
first()	Return the first element of the dataset
take(n)	Return an array with the first n elements of the dataset.
foreach(func)	Run a function func on each element of the dataset.

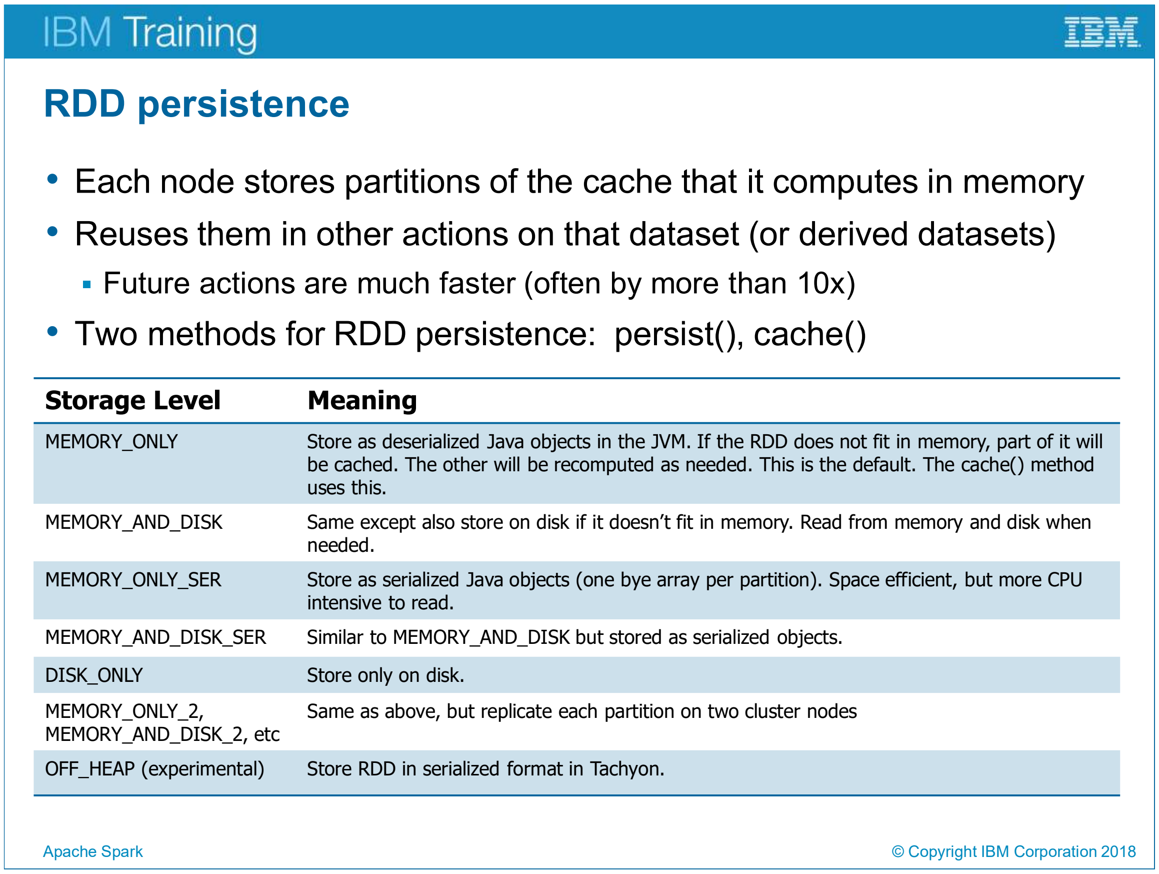
Apache Spark © Copyright IBM Corporation 2018

RDD operations: Actions

Action returns values. Again, you can find more information on Spark's website. The full set of functions are available at https://spark.apache.org/docs/latest/programming-guide.html#actions.

The slide shows a subset:

- The collect function returns all the elements of the dataset as an array of the driver program. This is usually useful after a filter or another operation that returns a significantly small subset of data to make sure your filter function works correctly.
- The count function returns the number of elements in a dataset and can also be used to check and test transformations.
- The take(n) function returns an array with the first n elements. Note that this is currently not executed in parallel. The driver computes all the elements.
- The foreach(func) function run a function func on each element of the dataset.



IBM Training

RDD persistence

- Each node stores partitions of the cache that it computes in memory
- Reuses them in other actions on that dataset (or derived datasets)
  - Future actions are much faster (often by more than 10x)
- Two methods for RDD persistence: persist(), cache()

Storage Level	Meaning
MEMORY_ONLY	Store as deserialized Java objects in the JVM. If the RDD does not fit in memory, part of it will be cached. The other will be recomputed as needed. This is the default. The cache() method uses this.
MEMORY_AND_DISK	Same except also store on disk if it doesn't fit in memory. Read from memory and disk when needed.
MEMORY_ONLY_SER	Store as serialized Java objects (one bye array per partition). Space efficient, but more CPU intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_AND_DISK but stored as serialized objects.
DISK_ONLY	Store only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as above, but replicate each partition on two cluster nodes
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon.

Apache Spark © Copyright IBM Corporation 2018

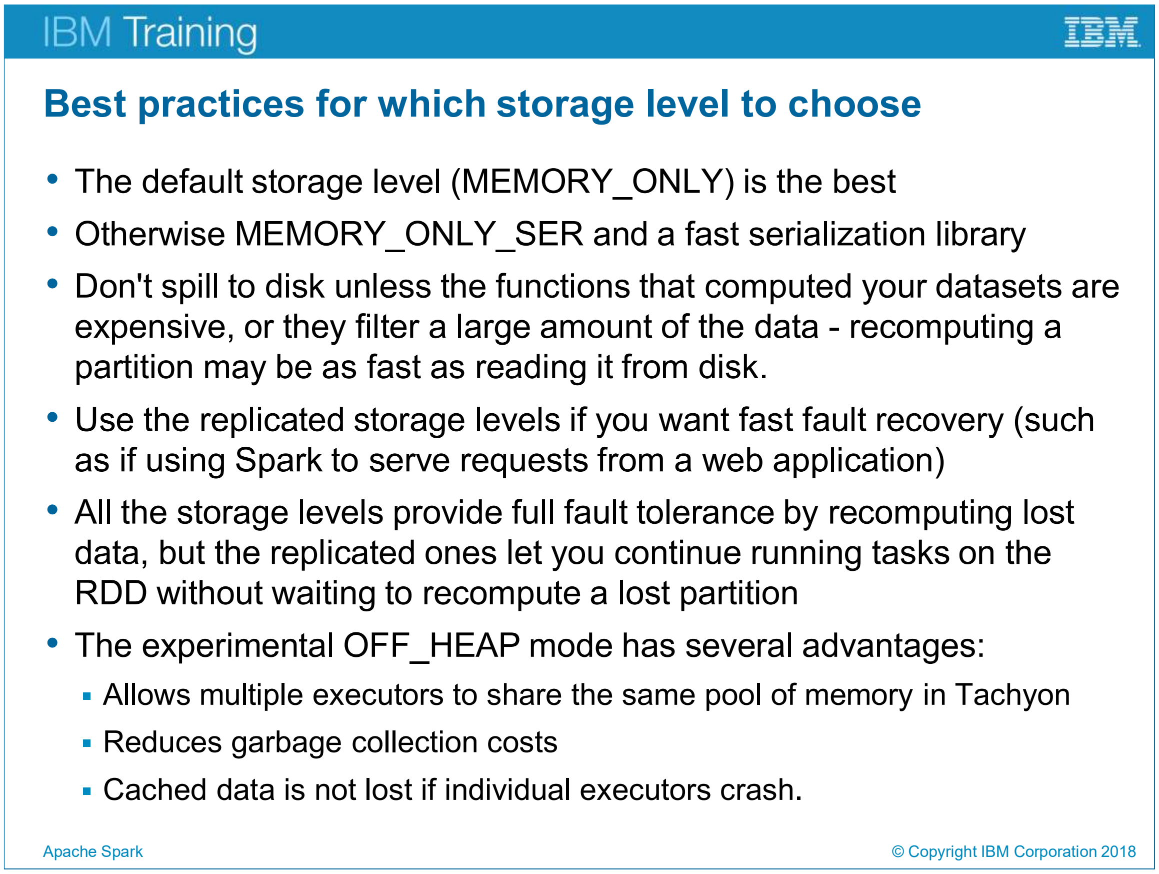
RDD persistence

Now to want to get a bit into RDD persistence. You have seen this used already; it is the cache function. The cache function is actually the default of the persist function; cache() is essentially just persist with MEMORY_ONLY storage.

One of the key capability of Spark is its speed through persisting or caching. Each node stores any partitions of the cache and computes it in memory. When a subsequent action is called on the same dataset, or a derived dataset, it uses it from memory instead of having to retrieve it again. Future actions in such cases are often 10 times faster. The first time a RDD is persisted, it is kept in memory on the node. Caching is fault tolerant because if it any of the partition is lost, it will automatically be recomputed using the transformations that originally created it.

There are two methods to invoke RDD persistence. persist() and cache(). The persist() method allows you to specify a different storage level of caching. For example, you can choose to persist the data set on disk, persist it in memory but as serialized objects to save space, etc. Again the cache() method is just the default way of using persistence by storing deserialized objects in memory.

The table shows the storage levels and what it means. Basically, you can choose to store in memory or memory and disk. If a partition does not fit in the specified cache location, then it will be recomputed on the fly. You can also decide to serialized the objects before storing this. This is space efficient, but will require the RDD to deserialized before it can be read, so it takes up more CPU workload. There's also the option to replicate each partition on two cluster nodes. Finally, there is an experimental storage level storing the serialized object in Tachyon. This level reduces garbage collection overhead and allows the executors to be smaller and to share a pool of memory. You can read more about this on Spark's website.



IBM Training

Best practices for which storage level to choose

- The default storage level (MEMORY_ONLY) is the best
- Otherwise MEMORY_ONLY_SER and a fast serialization library
- Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data - recomputing a partition may be as fast as reading it from disk.
- Use the replicated storage levels if you want fast fault recovery (such as if using Spark to serve requests from a web application)
- All the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition
- The experimental OFF_HEAP mode has several advantages:
  - Allows multiple executors to share the same pool of memory in Tachyon
  - Reduces garbage collection costs
  - Cached data is not lost if individual executors crash.

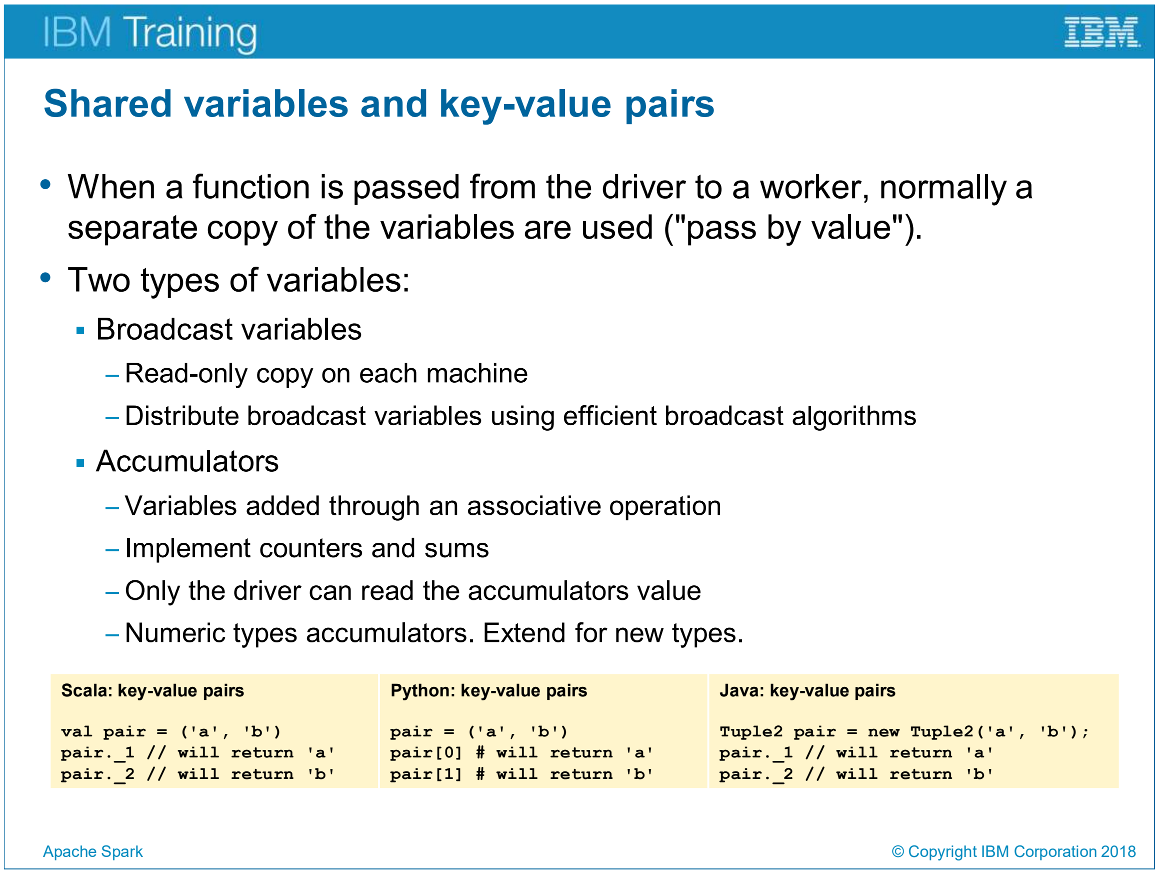
Apache Spark © Copyright IBM Corporation 2018

Best practices for which storage level to choose

There are a lot of rules on this page; use them as a reference when you have to decide the type of storage level. There are tradeoffs between the different storage levels. You should analyze your current situation to decide which level works best. You can find this information on Spark's website: https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

The primary rules are:

- If the RDDs fit comfortably with the default storage level (MEMORY_ONLY), leave them that way - the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible. Basically if your RDD fits within the default storage level, by all means, use that. It is the fastest option to fully take advantage of Spark's design.
- Otherwise use MEMORY_ONLY_SER and a fast serialization library to make objects more space-efficient, but still reasonably fast to access.
- Do not spill to disk unless the functions that compute your datasets are expensive or it requires a large amount of space.
- If you want fast recovery, use the replicated storage levels. Note that all levels of storage are fully fault tolerant, but would still require the recomputing of the data. If you have a replicated copy, you can continue to work while Spark is recomputing a lost partition.
- In environments with high amounts of memory or multiple applications, the experimental OFF_HEAP mode has several advantages. Use Tachyon if your environment has high amounts of memory or multiple applications. It allows you to share the same pool of memory and significantly reduces garbage collection costs. Also, the cached data is not lost if the individual executors crash.



IBM Training

Shared variables and key-value pairs

- When a function is passed from the driver to a worker, normally a separate copy of the variables are used ("pass by value").
- Two types of variables:
  - Broadcast variables
    - Read-only copy on each machine
    - Distribute broadcast variables using efficient broadcast algorithms
  - Accumulators
    - Variables added through an associative operation
    - Implement counters and sums
    - Only the driver can read the accumulators value
    - Numeric types accumulators. Extend for new types.

Scala: key-value pairs

```
val pair = ('a', 'b')
pair._1 // will return 'a'
pair._2 // will return 'b'
```

Python: key-value pairs

```
pair = ('a', 'b')
pair[0] # will return 'a'
pair[1] # will return 'b'
```

Java: key-value pairs

```
Tuple2 pair = new Tuple2('a', 'b');
pair._1 // will return 'a'
pair._2 // will return 'b'
```

Apache Spark

© Copyright IBM Corporation 2018

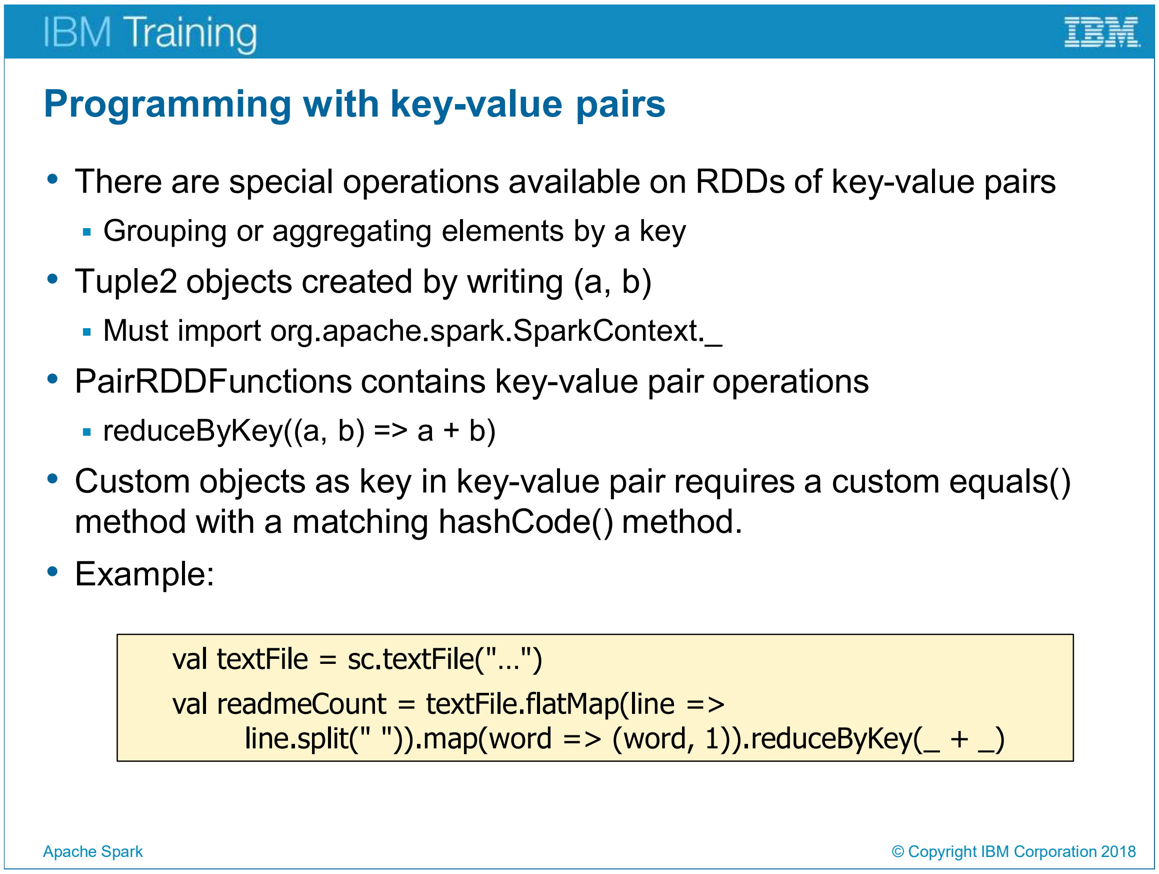
Shared variables and key-value pairs

On this page and the next, you will review Spark shared variables and the type of operations you can do on key-value pairs.

Spark provides two limited types of shared variables for common usage patterns: broadcast variables and accumulators. Normally, when a function is passed from the driver to a worker, a separate copy of the variables are used for each worker. Broadcast variables allow each machine to work with a read-only variable cached on each machine. Spark attempts to distribute broadcast variables using efficient algorithms. As an example, broadcast variables can be used to give every node a copy of a large dataset efficiently.

The other shared variables are accumulators. These are used for counters in sums that works well in parallel. These variables can only be added through an associated operation. Only the driver can read the accumulators value, not the tasks. The tasks can only add to it. Spark supports numeric types but programmers can add support for new types. As an example, you can use accumulator variables to implement counters or sums, as in MapReduce.

Last, but not least, key-value pairs are available in Scala, Python and Java. In Scala, you create a key-value pair RDD by typing val pair = ('a', 'b'). To access each element, invoke the **._** notation. This is not zero-index, so the **._1** will return the value in the first index and **._2** will return the value in the second index. Java is also very similar to Scala where it is not zero-index. You create the Tuple2 object in Java to create a key-value pair. In Python, it is a zero-index notation, so the value of the first index resides in index 0 and the second index is 1.



IBM Training

Programming with key-value pairs

- There are special operations available on RDDs of key-value pairs
  - Grouping or aggregating elements by a key
- Tuple2 objects created by writing (a, b)
  - Must import org.apache.spark.SparkContext._
- PairRDDFunctions contains key-value pair operations
  - reduceByKey((a, b) => a + b)
- Custom objects as key in key-value pair requires a custom equals() method with a matching hashCode() method.
- Example:

```
val textFile = sc.textFile("...")
val readmeCount = textFile.flatMap(line =>
        line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
```

Apache Spark © Copyright IBM Corporation 2018

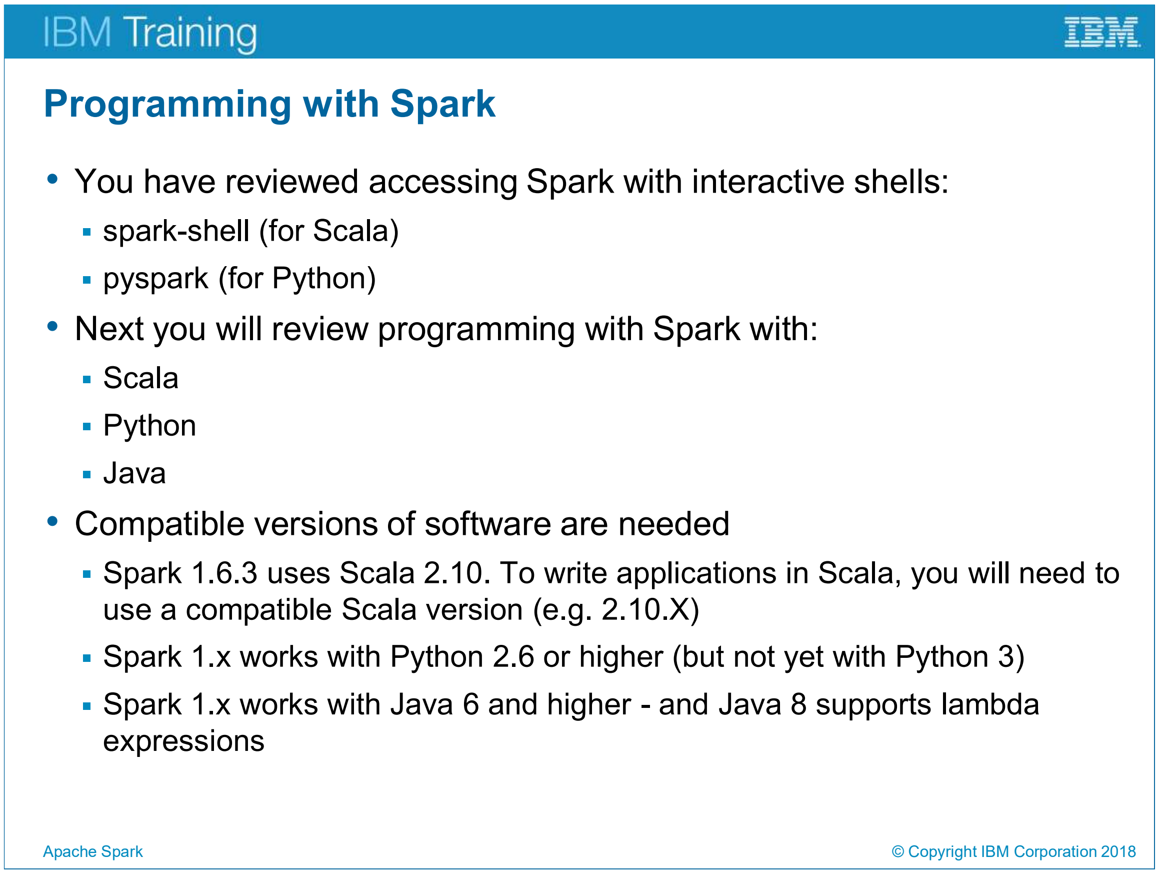
Programming with key-value pairs

There are special operations available to RDDs of key-value pairs. In an application, you must remember to import the **SparkContext** package to use **PairRDDFunctions** such as **reduceByKey**.

The most common ones are those that perform grouping or aggregating by a key. RDDs containing the Tuple2 object represents the key-value pairs. Tuple2 objects are simple created by writing (a, b) as long as you import the library to enable Spark's implicit conversion.

If you have custom objects as the key inside your key-value pair, remember that you will need to provide your own equals() method to do the comparison as well as a matching hashCode() method.

In the example, you have a **textFile** that is just a normal RDD. You then perform some transformations on it and it creates a PairRDD which allows it to invoke the **reduceByKey** method that is part of the PairRDDFunctions API.



IBM Training

Programming with Spark

- You have reviewed accessing Spark with interactive shells:
  - spark-shell (for Scala)
  - pyspark (for Python)
- Next you will review programming with Spark with:
  - Scala
  - Python
  - Java
- Compatible versions of software are needed
  - Spark 1.6.3 uses Scala 2.10. To write applications in Scala, you will need to use a compatible Scala version (e.g. 2.10.X)
  - Spark 1.x works with Python 2.6 or higher (but not yet with Python 3)
  - Spark 1.x works with Java 6 and higher - and Java 8 supports lambda expressions

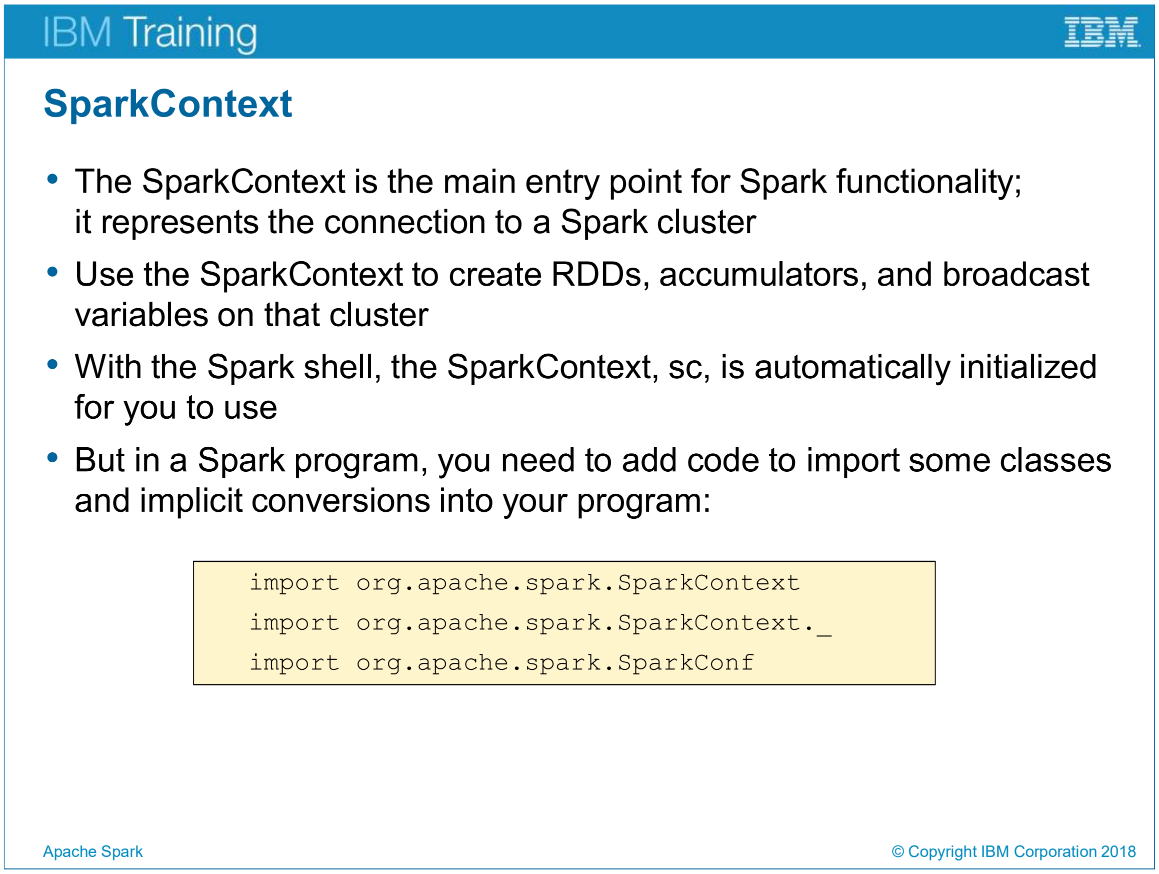
Apache Spark

© Copyright IBM Corporation 2018

Programming with Spark

Compatibility of Spark with various versions of the programming languages is important.

Note also that as new releases of the HDP are released, you should revisit the issue of compatibility of languages to work with the new versions of Spark. View all versions of Spark and compatible software at: http://spark.apache.org/documentation.html



IBM Training

SparkContext

- The SparkContext is the main entry point for Spark functionality; it represents the connection to a Spark cluster
- Use the SparkContext to create RDDs, accumulators, and broadcast variables on that cluster
- With the Spark shell, the SparkContext, sc, is automatically initialized for you to use
- But in a Spark program, you need to add code to import some classes and implicit conversions into your program:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

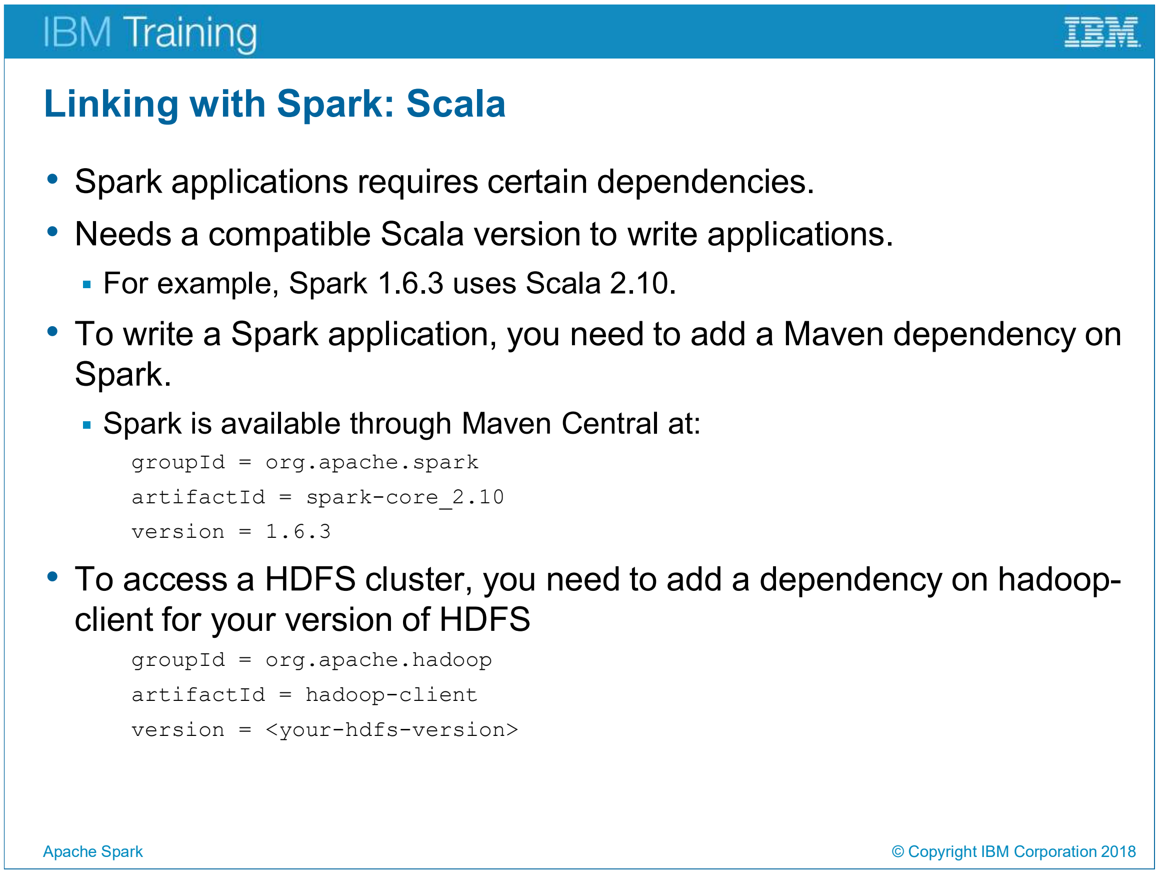
Apache Spark

© Copyright IBM Corporation 2018

SparkContext

The **SparkContext** is the main entry point to everything Spark. It can be used to create RDDs and shared variables on the cluster. When you start up the Spark Shell, the SparkContext is automatically initialized for you with the variable sc. For a Spark application, you must first import some classes and implicit conversions and then create the SparkContext object.

The three import statements for Scala are shown on the slide.



IBM Training

Linking with Spark: Scala

- Spark applications requires certain dependencies.
- Needs a compatible Scala version to write applications.
  - For example, Spark 1.6.3 uses Scala 2.10.
- To write a Spark application, you need to add a Maven dependency on Spark.
  - Spark is available through Maven Central at:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.6.3
```

- To access a HDFS cluster, you need to add a dependency on hadoop-client for your version of HDFS

```
groupId = org.apache.hadoop
artifactId = hadoop-client
version = <your-hdfs-version>
```

Apache Spark © Copyright IBM Corporation 2018

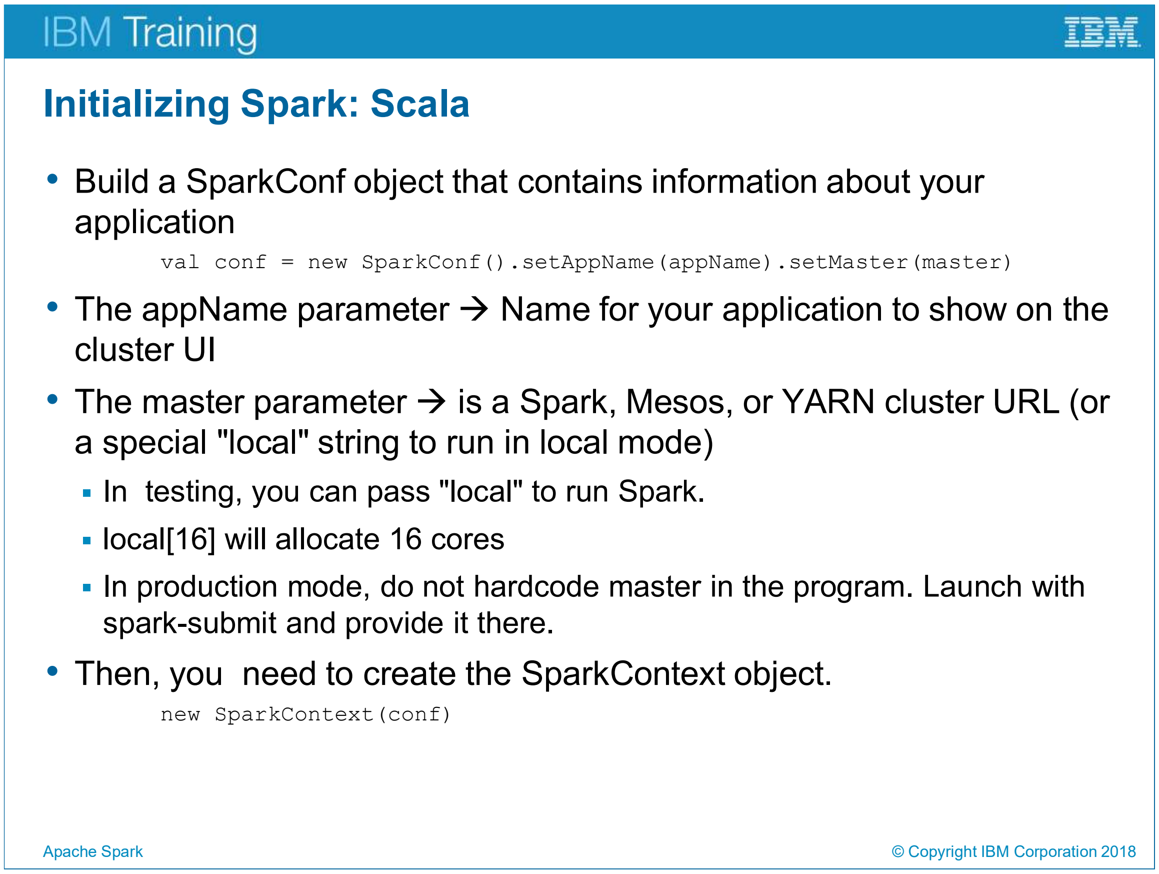
Linking with Spark: Scala

Each Spark application you create requires certain dependencies. Over the next three slides you will review how to link to those dependencies depending on which programming language you decide to use.

To link with Spark using Scala, you must have a compatible version of Scala with the Spark you choose to use. For example, Spark 1.6.3 uses Scala 2.10, so make sure that you have Scala 2.10 if you wish to write applications for Spark 1.6.3.

To write a Spark application, you must add a Maven dependency on Spark. The information is shown on the slide here. If you wish to access a Hadoop cluster, you need to add a dependency to that as well.

In the lab environment, this will already be set up for you. The information on this page is important if you want to set up a Spark stand-alone environment or your own Spark cluster. Visit the site for more information on Spark versions and dependencies: https://mvnrepository.com/artifact/org.apache.spark/spark-core?repo=hortonworks-releases



IBM Training

Initializing Spark: Scala

- Build a SparkConf object that contains information about your application

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
```

- The appName parameter → Name for your application to show on the cluster UI
- The master parameter → is a Spark, Mesos, or YARN cluster URL (or a special "local" string to run in local mode)
  - In testing, you can pass "local" to run Spark.
  - local[16] will allocate 16 cores
  - In production mode, do not hardcode master in the program. Launch with spark-submit and provide it there.
- Then, you need to create the SparkContext object.

```
new SparkContext(conf)
```

Apache Spark © Copyright IBM Corporation 2018

Initializing Spark: Scala

Once you have the dependencies established, the first thing is to do in your Spark application before you can initialize Spark is to build a SparkConf object. This object contains information about your application.

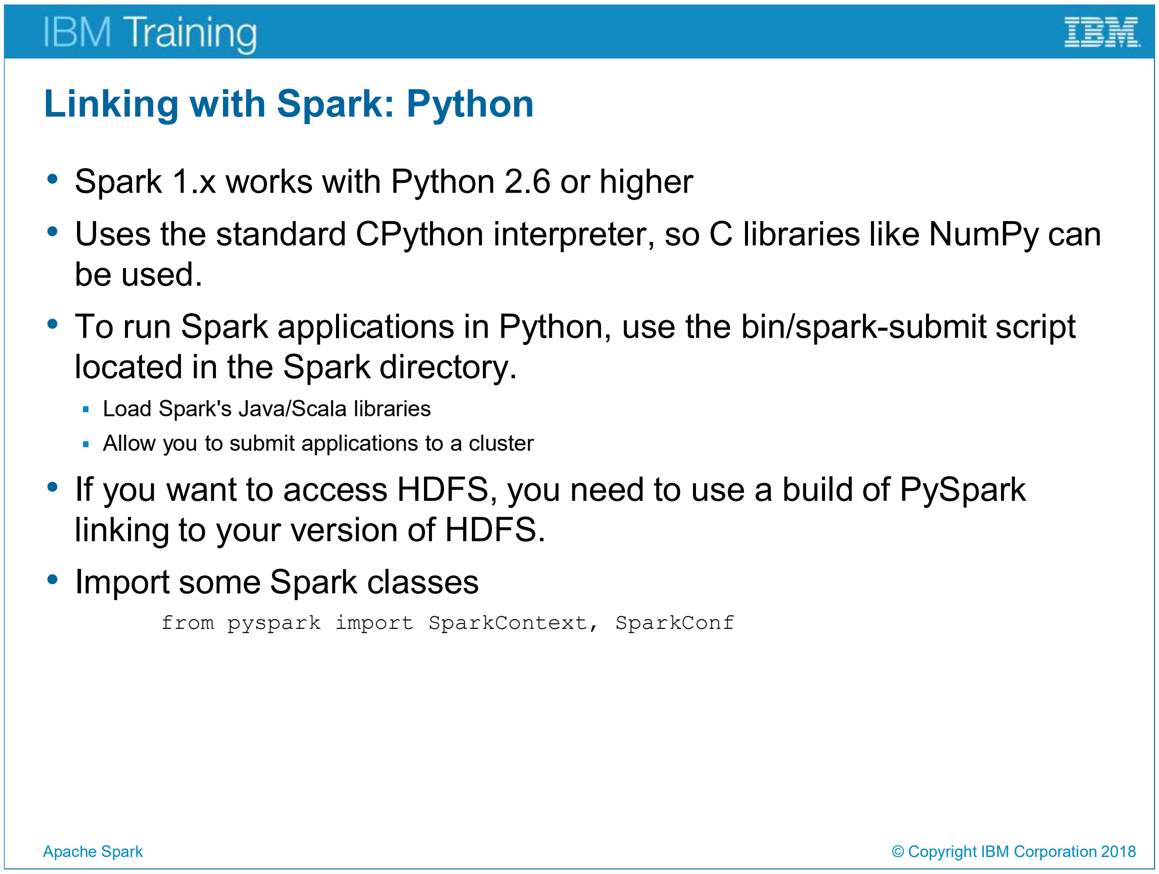
For example,

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
```

You set the application name and tell it which is the master node. The master parameter can be a standalone Spark distribution, Mesos, or a YARN cluster URL. You can also decide to use the local keyword string to run it in local mode. In fact, you can run local[16] to specify the number of cores to allocate for that particular job or Spark shell as 16.

For production mode, you would not want to hardcode the master path in your program. Instead, launch it as an argument to the spark-submit command.

Once you have the SparkConf all set up, you pass it as a parameter to the SparkContext constructor to create it.



IBM Training

Linking with Spark: Python

- Spark 1.x works with Python 2.6 or higher
- Uses the standard CPython interpreter, so C libraries like NumPy can be used.
- To run Spark applications in Python, use the bin/spark-submit script located in the Spark directory.
  - Load Spark's Java/Scala libraries
  - Allow you to submit applications to a cluster
- If you want to access HDFS, you need to use a build of PySpark linking to your version of HDFS.
- Import some Spark classes

```
from pyspark import SparkContext, SparkConf
```

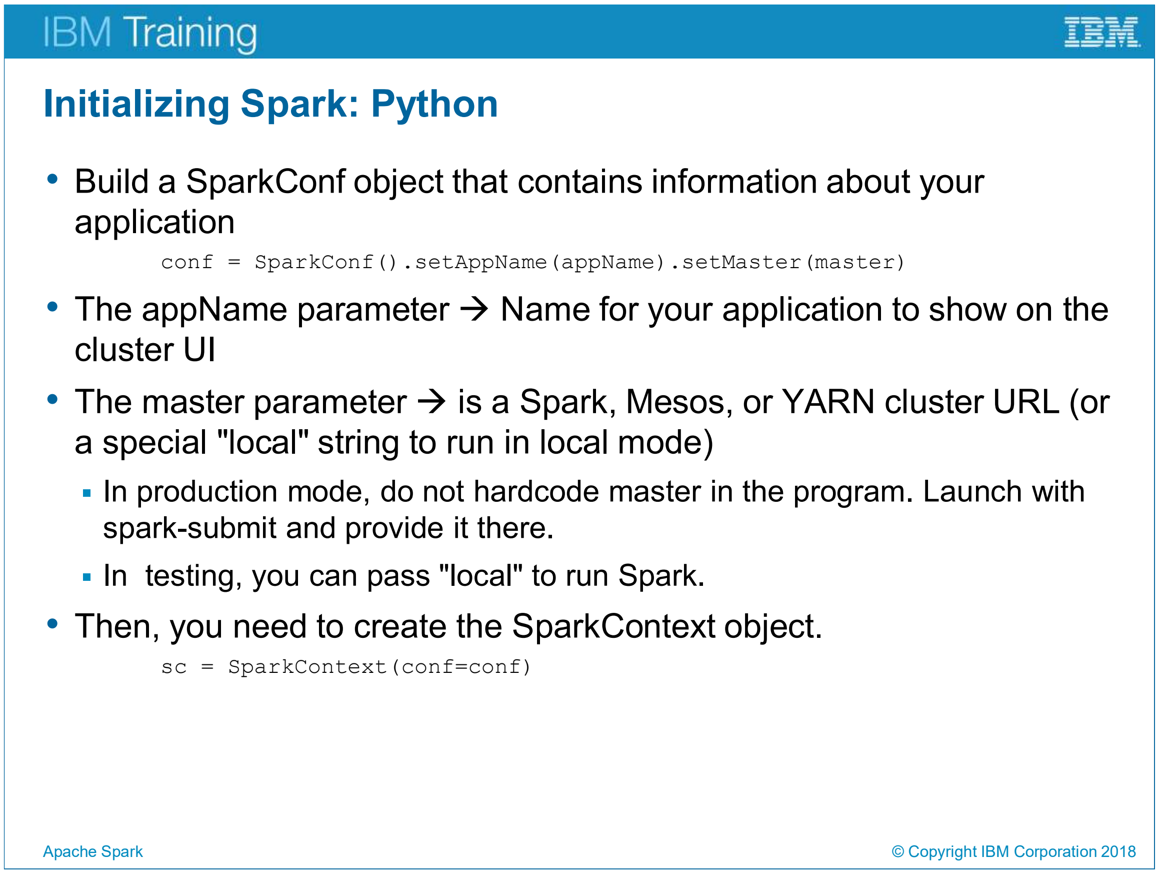
Apache Spark © Copyright IBM Corporation 2018

Linking with Spark: Python

Spark 1.6.3 works with Python 2.6 or higher. It uses the standard CPython interpreter, so C libraries like NumPy can be used.

Check which version of Spark you have when you enter an environment that uses it.

To run Spark applications in Python, use the bin/spark-submit script located in the Spark's home directory. This script will load the Spark's Java/Scala libraries and allow you to submit applications to a cluster. If you want to use HDFS, you will have to link to it as well. In the lab environment, you will not need to do this as Spark is bundled with it. You also need to import some Spark classes, as shown.



IBM Training

Initializing Spark: Python

- Build a SparkConf object that contains information about your application

```
conf = SparkConf().setAppName(appName).setMaster(master)
```

- The appName parameter → Name for your application to show on the cluster UI
- The master parameter → is a Spark, Mesos, or YARN cluster URL (or a special "local" string to run in local mode)
  - In production mode, do not hardcode master in the program. Launch with spark-submit and provide it there.
  - In testing, you can pass "local" to run Spark.
- Then, you need to create the SparkContext object.

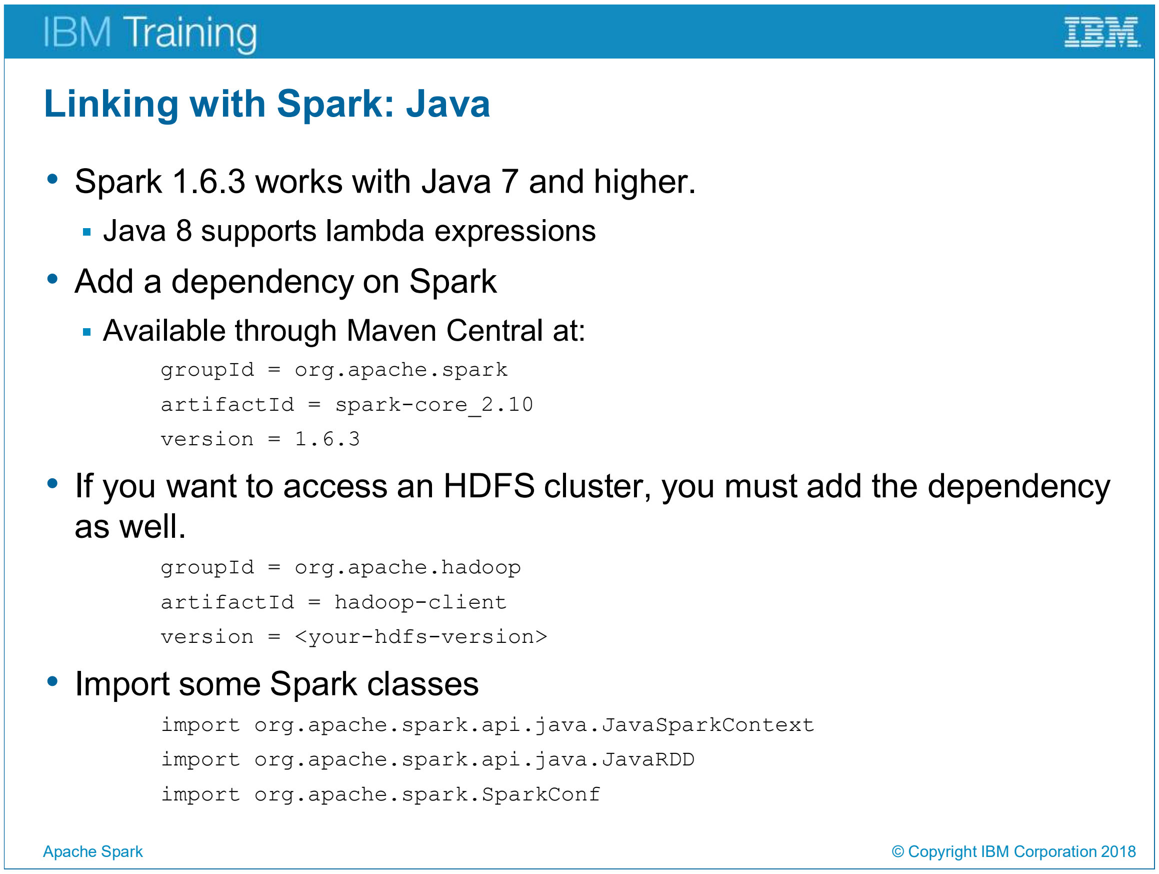
```
sc = SparkContext(conf=conf)
```

Apache Spark

© Copyright IBM Corporation 2018

Initializing Spark: Python

Here's the information for Python. It is pretty much the same information as Scala. The syntax here is slightly different, otherwise, you are required to set up a SparkConf object to pass as a parameter to the SparkContext object. You are also recommended to pass the master parameter as an argument to the spark-submit operation.



IBM Training

Linking with Spark: Java

- Spark 1.6.3 works with Java 7 and higher.
  - Java 8 supports lambda expressions
- Add a dependency on Spark
  - Available through Maven Central at:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.6.3
```

- If you want to access an HDFS cluster, you must add the dependency as well.

```
groupId = org.apache.hadoop
artifactId = hadoop-client
version = <your-hdfs-version>
```

- Import some Spark classes

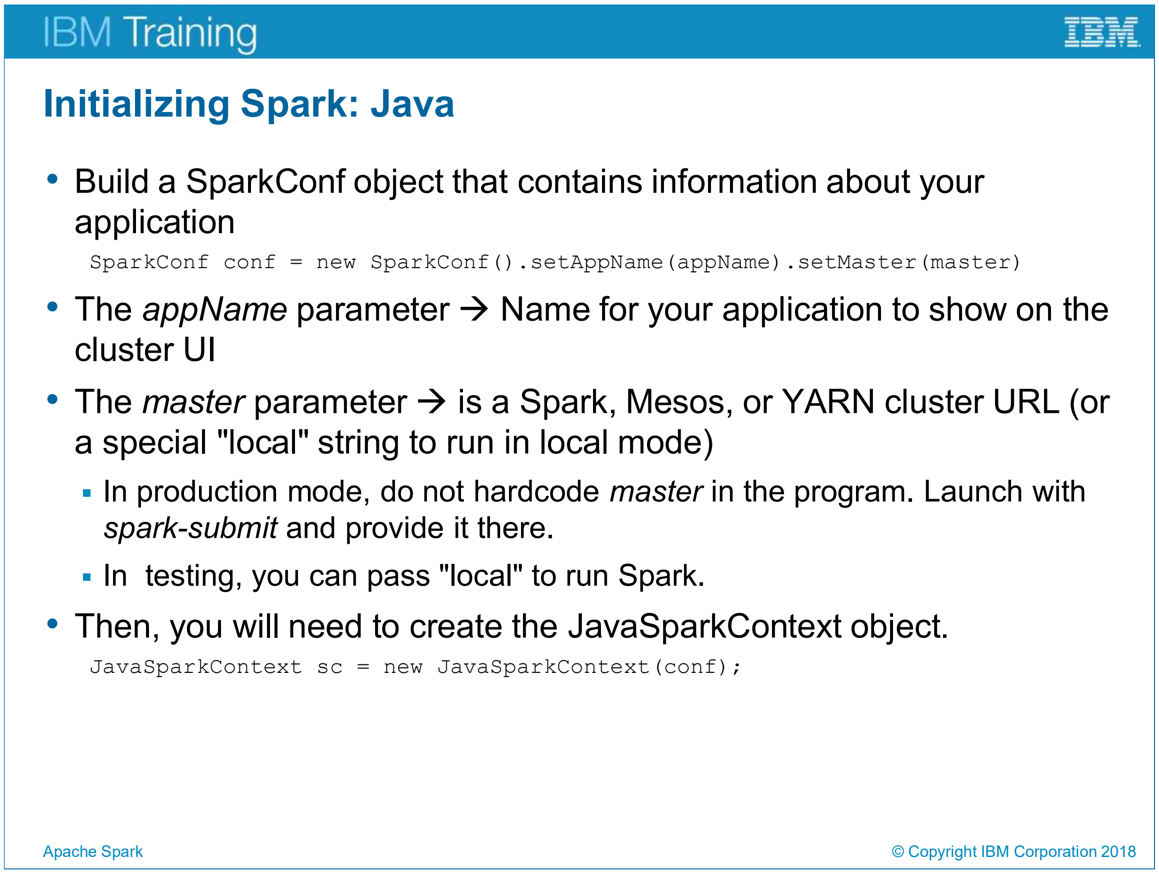
```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf
```

Apache Spark © Copyright IBM Corporation 2018

Linking with Spark: Java

Spark 1.6.3 works with Java 6 and higher. If you are using Java 8, Spark supports lambda expressions for concisely writing functions. Otherwise, you can use the org.apache.spark.api.java.function package with older Java versions.

As with Scala, you need to a dependency on Spark, which is available through Maven Central. If you wish to access an HDFS cluster, you must add the dependency there as well. Last, but not least, you need to import some Spark classes.



IBM Training

Initializing Spark: Java

- Build a SparkConf object that contains information about your application

  ```
  SparkConf conf = new SparkConf().setAppName(appName).setMaster(master)
  ```
- The *appName* parameter → Name for your application to show on the cluster UI
- The *master* parameter → is a Spark, Mesos, or YARN cluster URL (or a special "local" string to run in local mode)
  - In production mode, do not hardcode *master* in the program. Launch with *spark-submit* and provide it there.
  - In testing, you can pass "local" to run Spark.
- Then, you will need to create the JavaSparkContext object.

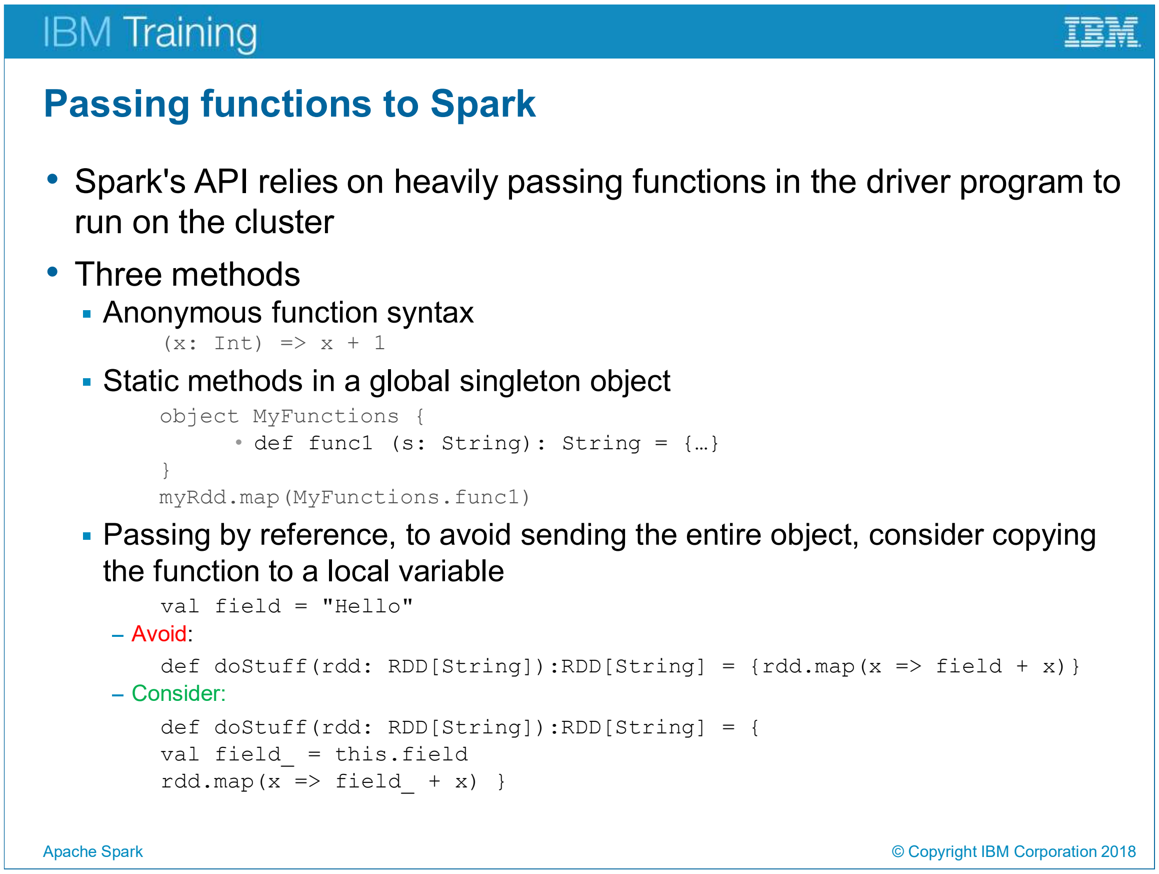
  ```
  JavaSparkContext sc = new JavaSparkContext(conf);
  ```

Apache Spark

© Copyright IBM Corporation 2018

Initializing Spark: Java

Here is the same information for Java. Following the same idea, you need to create the SparkConf object and pass that to the SparkContext, which in this case, would be a JavaSparkContext. Remember, when you imported statements in the program, you imported the JavaSparkContext libraries.


IBM Training

Passing functions to Spark

- Spark's API relies on heavily passing functions in the driver program to run on the cluster
- Three methods
  - Anonymous function syntax

    ```
    (x: Int) => x + 1
    ```
  - Static methods in a global singleton object

    ```
    object MyFunctions {
        • def func1 (s: String): String = {…}
    }
    myRdd.map(MyFunctions.func1)
    ```
  - Passing by reference, to avoid sending the entire object, consider copying the function to a local variable

    ```
    val field = "Hello"
    ```
    - Avoid:

      ```
      def doStuff(rdd: RDD[String]):RDD[String] = {rdd.map(x => field + x)}
      ```
    - Consider:

      ```
      def doStuff(rdd: RDD[String]):RDD[String] = {
      val field_ = this.field
      rdd.map(x => field_ + x) }
      ```

Apache Spark © Copyright IBM Corporation 2018

Passing functions to Spark

Passing functions to Spark is important to understand as you begin to think about the business logic of your application.

The design of Spark's API relies heavily on passing functions in the driver program to run on the cluster. When a job is executed, the Spark driver needs to tell its worker how to process the data.

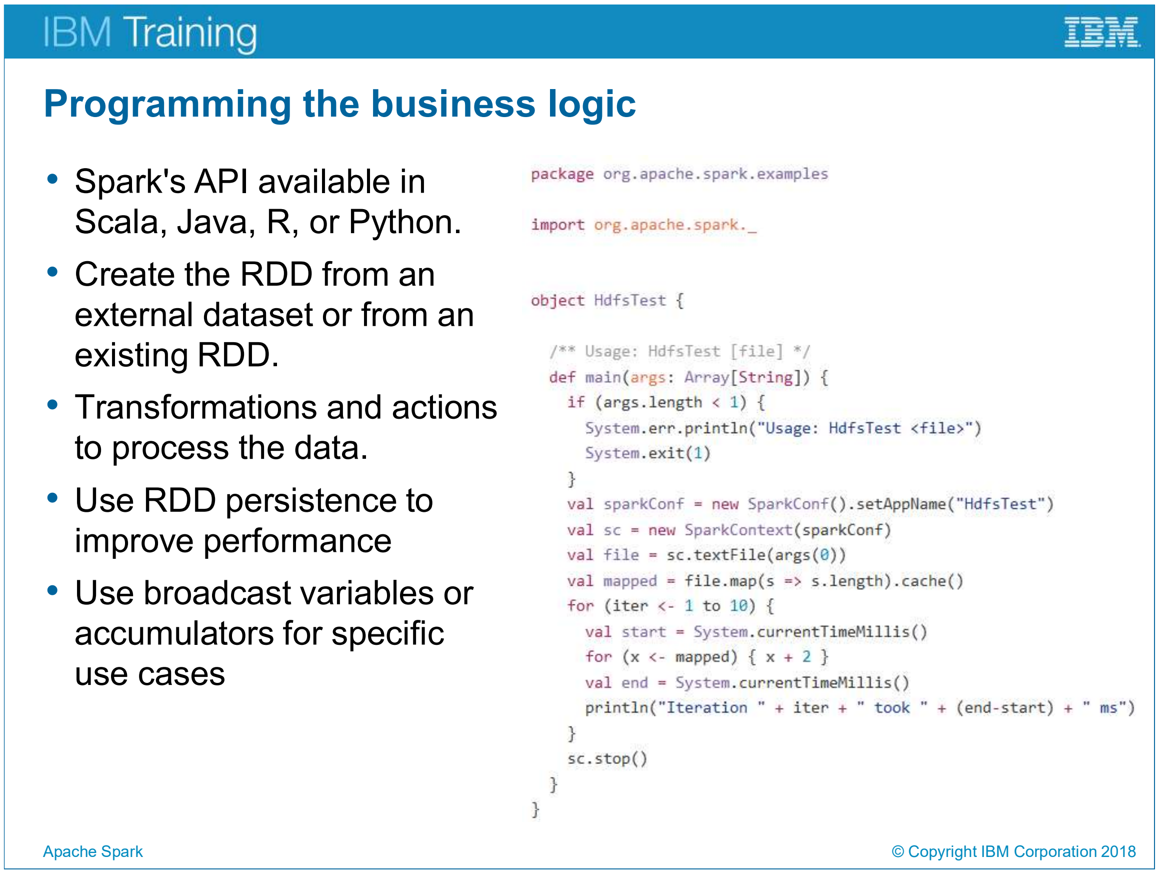
There are three methods that you can use to pass functions.

- The first method to do this is using an anonymous function syntax. You saw briefly what an anonymous function is in the first lesson. This is useful for short pieces of code. For example, here we define the anonymous function that takes in a parameter x of type Int and add one to it. Essentially, anonymous functions are useful for one-time use of the function. In other words, you do not need to explicitly define the function to use it. You define as you go. Again, the left side of the => are the parameters or the argument. The right side of the => is the body of the function.
- Another method to pass functions around Spark is to use static methods in a global singleton object. This means that you can create a global object, in the example, it is the object MyFunctions. Inside that object, you basically define the function func1. When the driver requires that function, it only needs to send out the object - the worker will be able to access it. In this case, when the driver sends out the instructions to the worker, it just has to send out the singleton object.
- It is possible to pass reference to a method in a class instance, as opposed to a singleton object. This would require sending the object that contains the class along with the method. To avoid this consider copying it to a local variable within the function instead of accessing it externally.

Example, say you have a field with the string Hello. You want to avoid calling that directly inside a function as shown on the slide as x => field + x.

Instead, assign it to a local variable so that only the reference is passed along and not the entire object shown `val field_ = this.field`

For an example such as this, it may seem trivial, but imagine if the field object is not a simple text Hello, but is something much larger, say a large log file. In that case, passing by reference will have greater value by saving a lot of storage by not having to pass the entire file.



IBM Training

Programming the business logic

- Spark's API available in Scala, Java, R, or Python.
- Create the RDD from an external dataset or from an existing RDD.
- Transformations and actions to process the data.
- Use RDD persistence to improve performance
- Use broadcast variables or accumulators for specific use cases

```
package org.apache.spark.examples

import org.apache.spark._


object HdfsTest {

  /** Usage: HdfsTest [file] */
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: HdfsTest <file>")
      System.exit(1)
    }
    val sparkConf = new SparkConf().setAppName("HdfsTest")
    val sc = new SparkContext(sparkConf)
    val file = sc.textFile(args(0))
    val mapped = file.map(s => s.length).cache()
    for (iter <- 1 to 10) {
      val start = System.currentTimeMillis()
      for (x <- mapped) { x + 2 }
      val end = System.currentTimeMillis()
      println("Iteration " + iter + " took " + (end-start) + " ms")
    }
    sc.stop()
  }
}
```

Apache Spark

Programming the business logic

At this point, you should know how to link dependencies with Spark and also know how to initialize the SparkContext. I also touched a little bit on passing functions with Spark to give you a better view of how you can program your business logic. This course will not focus too much on how to program business logics, but there are examples available for you to see how it is done. The purpose is to show you how you can create an application using a simple, but effective examples which demonstrates Spark's capabilities.

Once you have the beginning of your application ready by creating the SparkContext object, you can start to program in the business logic using Spark's API available in Scala, Java, or Python. You create the RDD from an external dataset or from an existing RDD. You use transformations and actions to compute the business logic. You can take advantage of RDD persistence, broadcast variables and/or accumulators to improve the performance of your jobs.

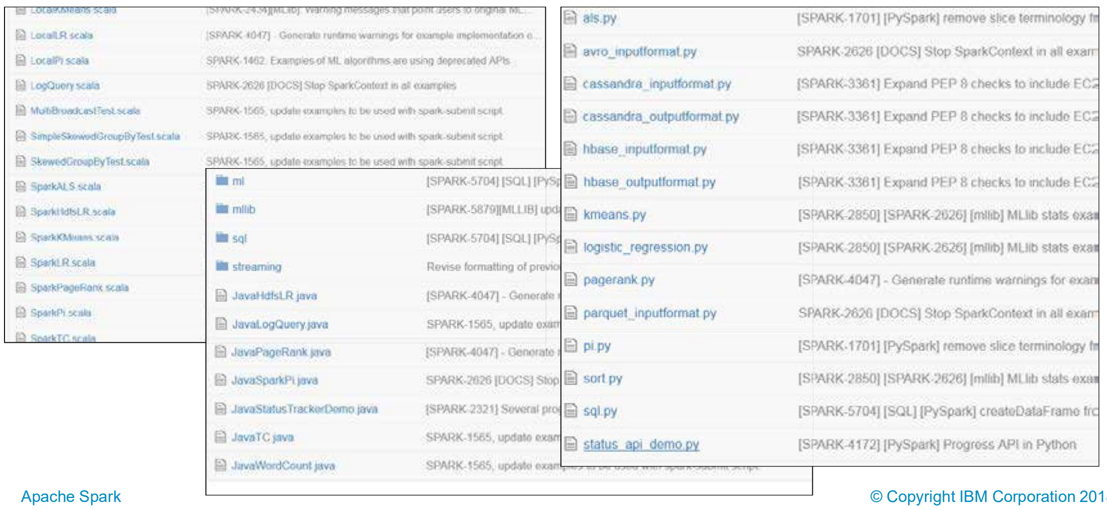
Here's a sample Scala application. You have your import statement. After the beginning of the object, you see that the SparkConf is created with the application name. Then a SparkContext is created. The several lines of code after is creating the RDD from a text file and then performing the Hdfs test on it to see how long the iteration through the file takes. Finally, at the end, you stop the SparkContext by calling the stop() function.

Again, just a simple example to show how you would create a Spark application. You will get to practice this in an exercise.

IBM Training

Running Spark examples

- Spark samples available in the examples directory
- Run the examples: `./bin/run-example SparkPi`
  where SparkPi is the name of the sample application
- In Python: `./bin/spark-submit examples/src/main/python/pi.py`



LocalKMeans.scala	[SPARK-2434][MLlib]: Warning messages that point users to original ML...
LocalLR.scala	[SPARK-4047] - Generate runtime warnings for example implementation o...
LocalPi.scala	SPARK-1462: Examples of ML algorithms are using deprecated APIs
LogQuery.scala	SPARK-2626 [DOCS] Stop SparkContext in all examples
MultiBroadcastTest.scala	SPARK-1565, update examples to be used with spark-submit script
SimpleSkewedGroupByTest.scala	SPARK-1565, update examples to be used with spark-submit script
SkewedGroupByTest.scala	SPARK-1565, update examples to be used with spark-submit script
SparkALS.scala	
SparkHdfsLR.scala	
SparkKMeans.scala	
SparkLR.scala	
SparkPageRank.scala	
SparkPi.scala	
SparkTC.scala	

ml	[SPARK-5704] [SQL] [PySp
mllib	[SPARK-5879][MLLIB] upd
sql	[SPARK-5704] [SQL] [PySp
streaming	Revise formatting of previo
JavaHdfsLR.java	[SPARK-4047] - Generate
JavaLogQuery.java	SPARK-1565, update exam
JavaPageRank.java	[SPARK-4047] - Generate
JavaSparkPi.java	SPARK-2626 [DOCS] Stop
JavaStatusTrackerDemo.java	[SPARK-2321] Several pro
JavaTC.java	SPARK-1565, update exam
JavaWordCount.java	SPARK-1565, update examples to be used with spark-submit script

als.py	[SPARK-1701] [PySpark] remove slice terminology fr
avro_inputformat.py	SPARK-2626 [DOCS] Stop SparkContext in all exam
cassandra_inputformat.py	[SPARK-3361] Expand PEP 8 checks to include EC2
cassandra_outputformat.py	[SPARK-3361] Expand PEP 8 checks to include EC2
hbase_inputformat.py	[SPARK-3361] Expand PEP 8 checks to include EC2
hbase_outputformat.py	[SPARK-3361] Expand PEP 8 checks to include EC2
kmeans.py	[SPARK-2850] [SPARK-2626] [mllib] MLlib stats exa
logistic_regression.py	[SPARK-2850] [SPARK-2626] [mllib] MLlib stats exa
pagerank.py	[SPARK-4047] - Generate runtime warnings for exam
parquet_inputformat.py	SPARK-2626 [DOCS] Stop SparkContext in all exam
pi.py	[SPARK-1701] [PySpark] remove slice terminology fr
sort.py	[SPARK-2850] [SPARK-2626] [mllib] MLlib stats exa
sql.py	[SPARK-5704] [SQL] [PySpark] createDataFrame fr
status_api_demo.py	[SPARK-4172] [PySpark] Progress API in Python

Apache Spark

© Copyright IBM Corporation 2018

Running Spark examples

There are many example programs available that shows the various usage of Spark. Depending on your programming language preference, there are examples in three languages that work with Spark. You can view the source code of the examples on the Spark website, or on Github, or within the Spark distribution itself. The slide is a partial overview.

For a full list of the examples available in GitHuB:

Scala:
https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples

Python:
https://github.com/apache/spark/tree/master/examples/src/main/python

Java:
https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples

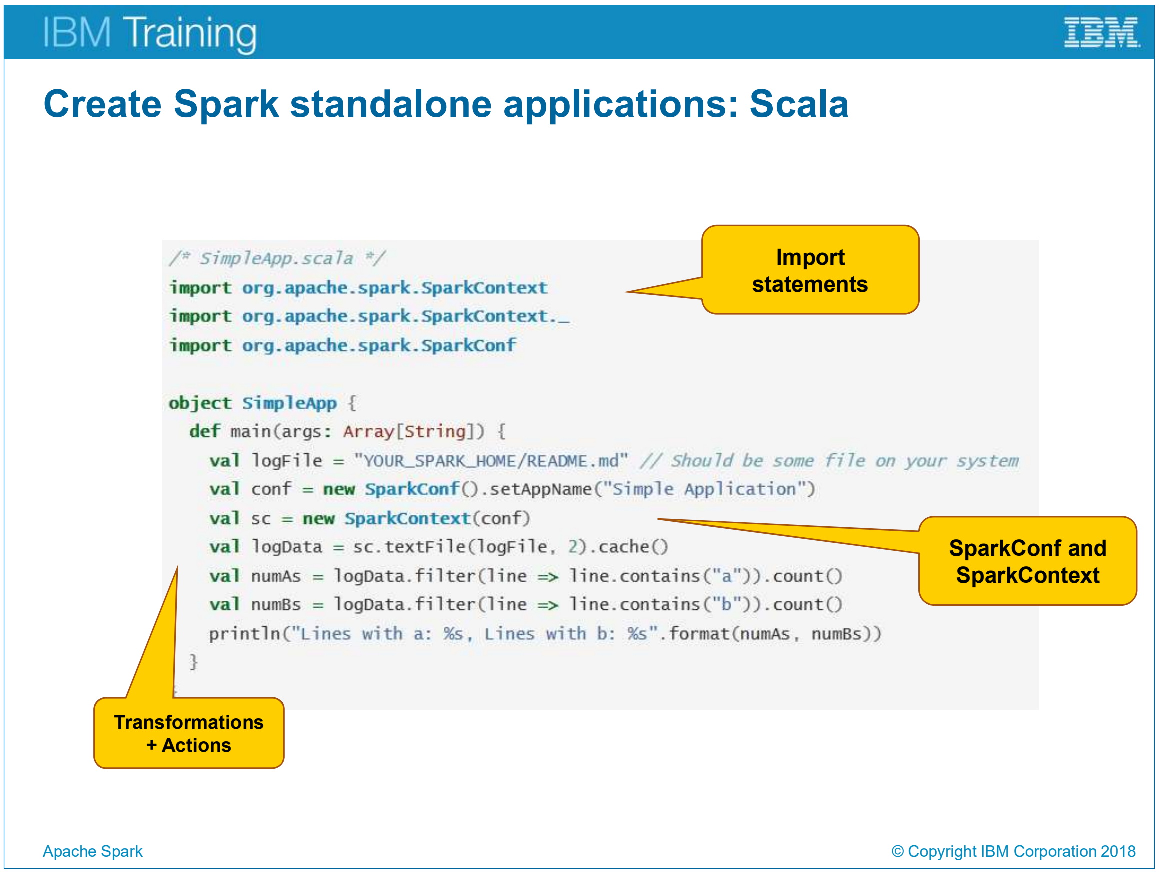
R:
https://github.com/apache/spark/tree/master/examples/src/main/r

Spark Streaming:
https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/
examples/streaming

Java Streaming:
https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/
examples/streaming

The slide lists the steps to run some of these examples. To run Scala or Java examples, you would execute the run-example script under the Spark's bin directory. So for example, to run the SparkPi application, execute run-example SparkPi, where SparkPi would be the name of the application. Substitute that with a different application name to run that other applications.

To run the sample Python applications, use the spark-submit command and provide the path to the application.



Create Spark standalone applications: Scala

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

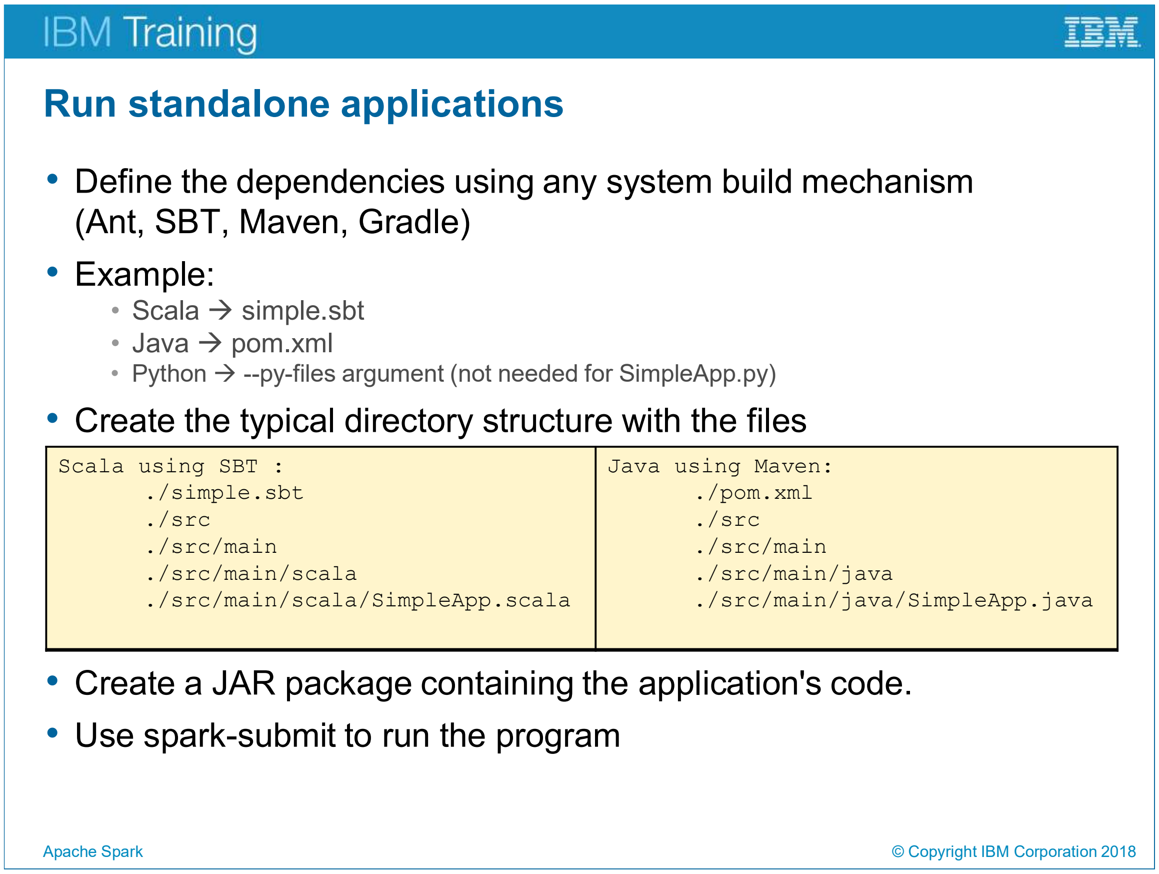
Import statements
SparkConf and SparkContext
Transformations + Actions
Apache Spark
© Copyright IBM Corporation 2018

Create Spark standalone applications - Scala

This is an example application using Scala. Similarly programs can be written in Python or Java.

The application shown here counts the number of lines with 'a' and the number of lines with 'b' - you need to replace the YOUR_SPARK_HOME with the directory where Spark is installed.

Unlike the Spark shell, you have to initialize the SparkContext in a program. First you must create a SparkConf to set up your application's name. Then you create the SparkContext by passing in the SparkConf object. Next, you create the RDD by loading in the textFile, and then caching the RDD. Since we will be apply a couple of transformation on it, caching will help speed up the process, especially if the logData RDD is large. Finally, you get the values of the RDD by executing the count action on it. End the program by printing it out onto the console.



IBM Training

Run standalone applications

- Define the dependencies using any system build mechanism (Ant, SBT, Maven, Gradle)
- Example:
  - Scala → simple.sbt
  - Java → pom.xml
  - Python → --py-files argument (not needed for SimpleApp.py)
- Create the typical directory structure with the files

```
Scala using SBT :
       ./simple.sbt
       ./src
       ./src/main
       ./src/main/scala
       ./src/main/scala/SimpleApp.scala
```

```
Java using Maven:
       ./pom.xml
       ./src
       ./src/main
       ./src/main/java
       ./src/main/java/SimpleApp.java
```

- Create a JAR package containing the application's code.
- Use spark-submit to run the program

Apache Spark © Copyright IBM Corporation 2018

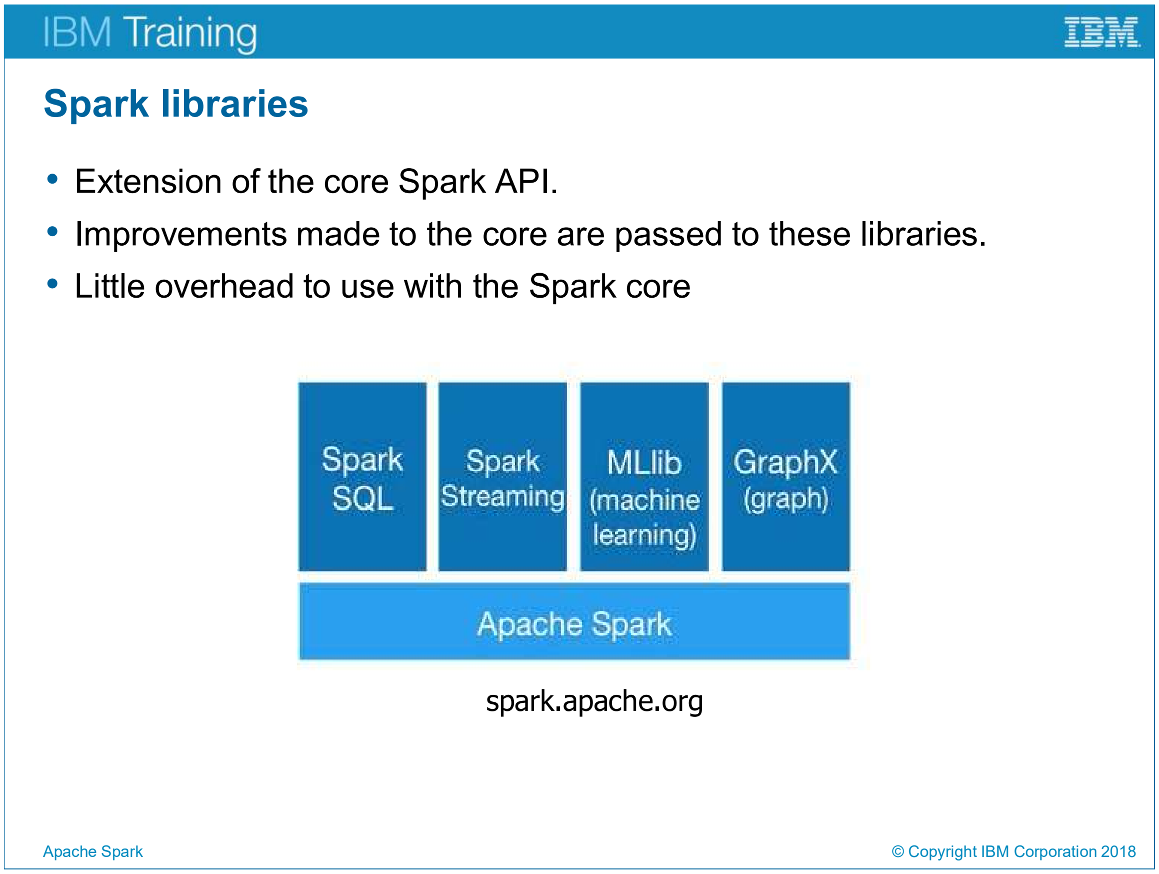
Run standalone applications

At this point, you should know how to create a Spark application using any of the supported programming language. Now you get to explore how to run the application. You will need to first define the dependencies. Then you have to package the application together using system build tools such as Ant, sbt, or Maven. The examples here show how you would do it using various tools. You can use any tool for any of the programming languages. For Scala, the example is shown using sbt, so you would have a simple.sbt file. In Java, the example shows using Maven so you would have the pom.xml file. In Python, if you need to have dependencies that requires third party libraries, then you can use the -py-files argument to handle that.

Again, shown here are examples of what a typical directory structure would look like for the tool that you choose.

Finally, once you have the JAR packaged created, run the spark-submit to execute the application:

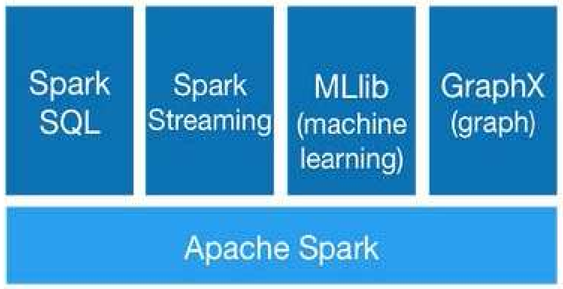
- Scala: sbt
- Java: mvn
- Python: submit-spark



IBM Training

Spark libraries

- Extension of the core Spark API.
- Improvements made to the core are passed to these libraries.
- Little overhead to use with the Spark core



spark.apache.org

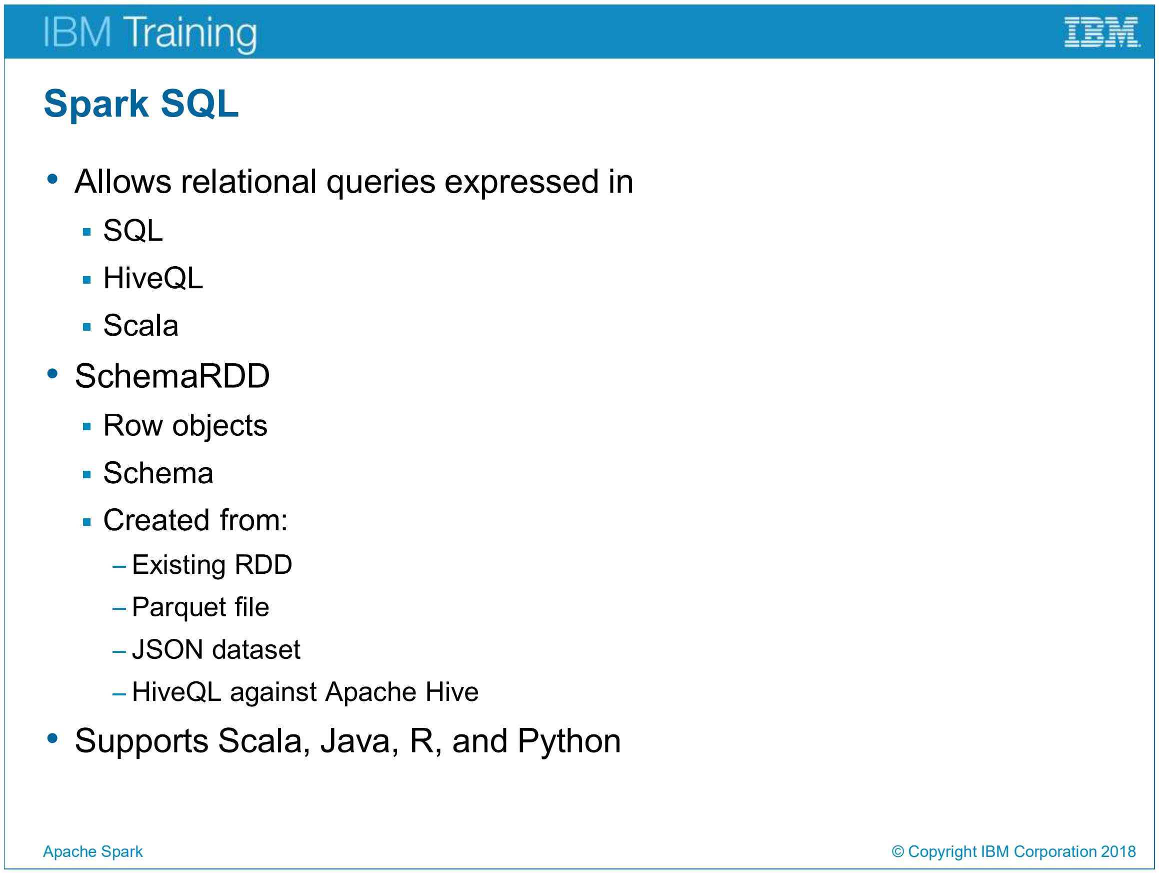
Apache Spark

© Copyright IBM Corporation 2018

Spark libraries

Spark comes with libraries that you can utilize for specific use cases. These libraries are an extension of the Spark Core API. Any improvements made to the core will automatically take effect with these libraries. One of the big benefits of Spark is that there is little overhead to use these libraries with Spark as they are tightly integrated. The rest of this lesson will cover on a high level, each of these libraries and their capabilities. The main focus will be on Scala with specific callouts to Java or Python if there are major differences.

The four libraries are Spark SQL, Spark Streaming, Mllib, and GraphX. The remainder of this course will cover these libraries.



IBM Training

Spark SQL

- Allows relational queries expressed in
  - SQL
  - HiveQL
  - Scala
- SchemaRDD
  - Row objects
  - Schema
  - Created from:
    - Existing RDD
    - Parquet file
    - JSON dataset
    - HiveQL against Apache Hive
- Supports Scala, Java, R, and Python

Apache Spark

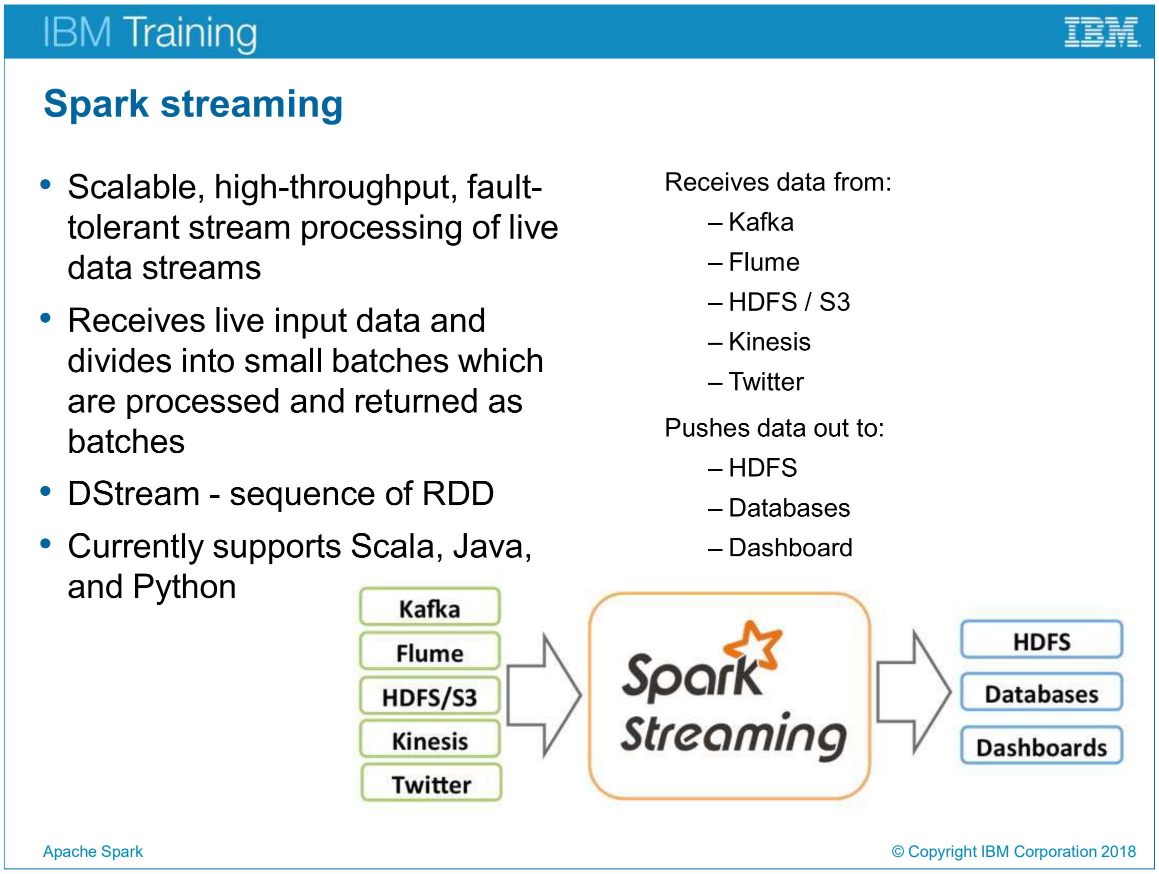
© Copyright IBM Corporation 2018

Spark SQL

Spark SQL allows you to write relational queries that are expressed in either SQL, HiveQL, or Scala to be executed using Spark. Spark SQL has a new RDD called the SchemaRDD. The SchemaRDD consists of rows objects and a schema that describes the type of data in each column in the row. You can think of this as a table in a traditional relational database.

You create a SchemaRDD from existing RDDs, a Parquet file, a JSON dataset, or using HiveQL to query against the data stored in Hive. Spark SQL is currently an alpha component, so some APIs may change in future releases.

Spark SQL supports Scala, Java, R, and Python.



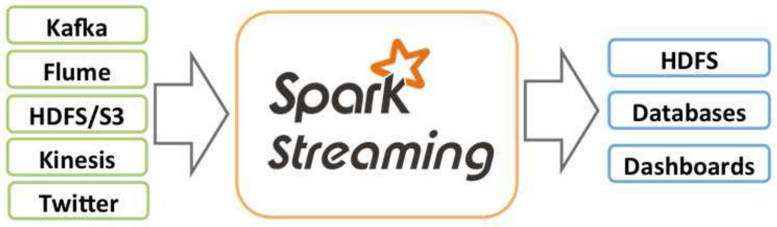
IBM Training

Spark streaming

- Scalable, high-throughput, fault-tolerant stream processing of live data streams
- Receives live input data and divides into small batches which are processed and returned as batches
- DStream - sequence of RDD
- Currently supports Scala, Java, and Python

Receives data from:
- Kafka
- Flume
- HDFS / S3
- Kinesis
- Twitter

Pushes data out to:
- HDFS
- Databases
- Dashboard



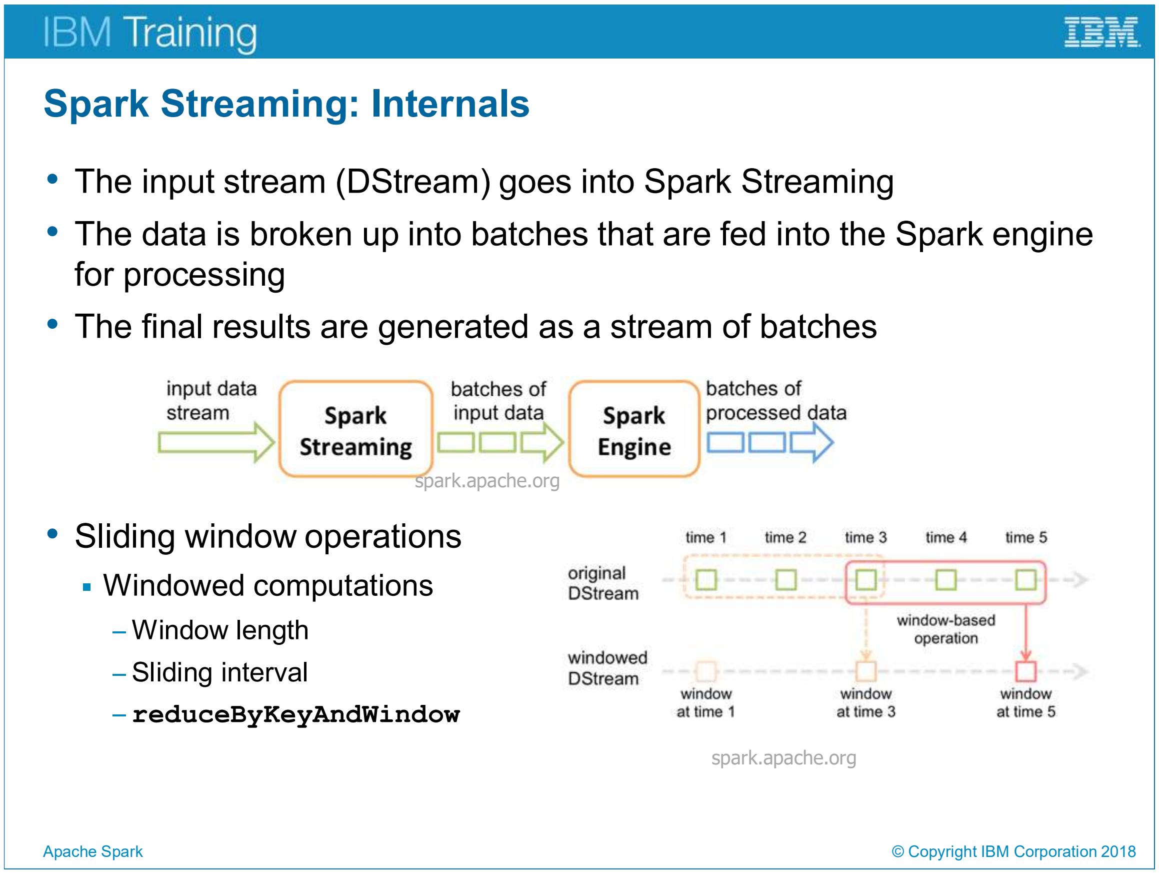
Apache Spark

© Copyright IBM Corporation 2018

Spark streaming

Spark streaming gives you the capability to process live streaming data in small batches. Utilizing Spark's core, Spark Streaming is scalable, high-throughput and fault-tolerant. You write Stream programs with DStreams, which is a sequence of RDDs from a stream of data. There are various data sources that Spark Streaming receives from including, Kafka, Flume, HDFS, Kinesis, or Twitter. It pushes data out to HDFS, databases, or some sort of dashboard.

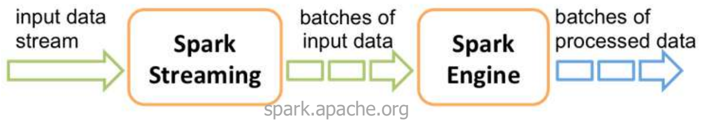
Spark Streaming supports Scala, Java and Python. Python was actually introduced with Spark 1.2. Python has all the transformations that Scala and Java have with DStreams, but it can only support text data types. Support for other sources such as Kafka and Flume will be available in future releases for Python.



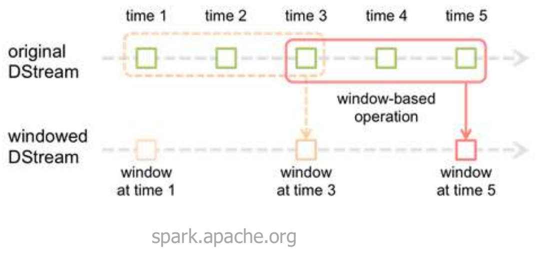
IBM Training

Spark Streaming: Internals

- The input stream (DStream) goes into Spark Streaming
- The data is broken up into batches that are fed into the Spark engine for processing
- The final results are generated as a stream of batches



- Sliding window operations
  - Windowed computations
    - Window length
    - Sliding interval
    - `reduceByKeyAndWindow`



Apache Spark

© Copyright IBM Corporation 2018

Spark Streaming: Internals

Here's a quick view of how Spark Streaming works. First the input stream comes in to Spark Streaming. That data stream is broken up into batches of data that are fed into the Spark engine for processing. Once the data has been processed, it is sent out in batches.

Spark Stream support sliding window operations. In a windowed computation, every time the window slides over a source of DStream, the source RDDs that falls within the window are combined and operated upon to produce the resulting RDD.

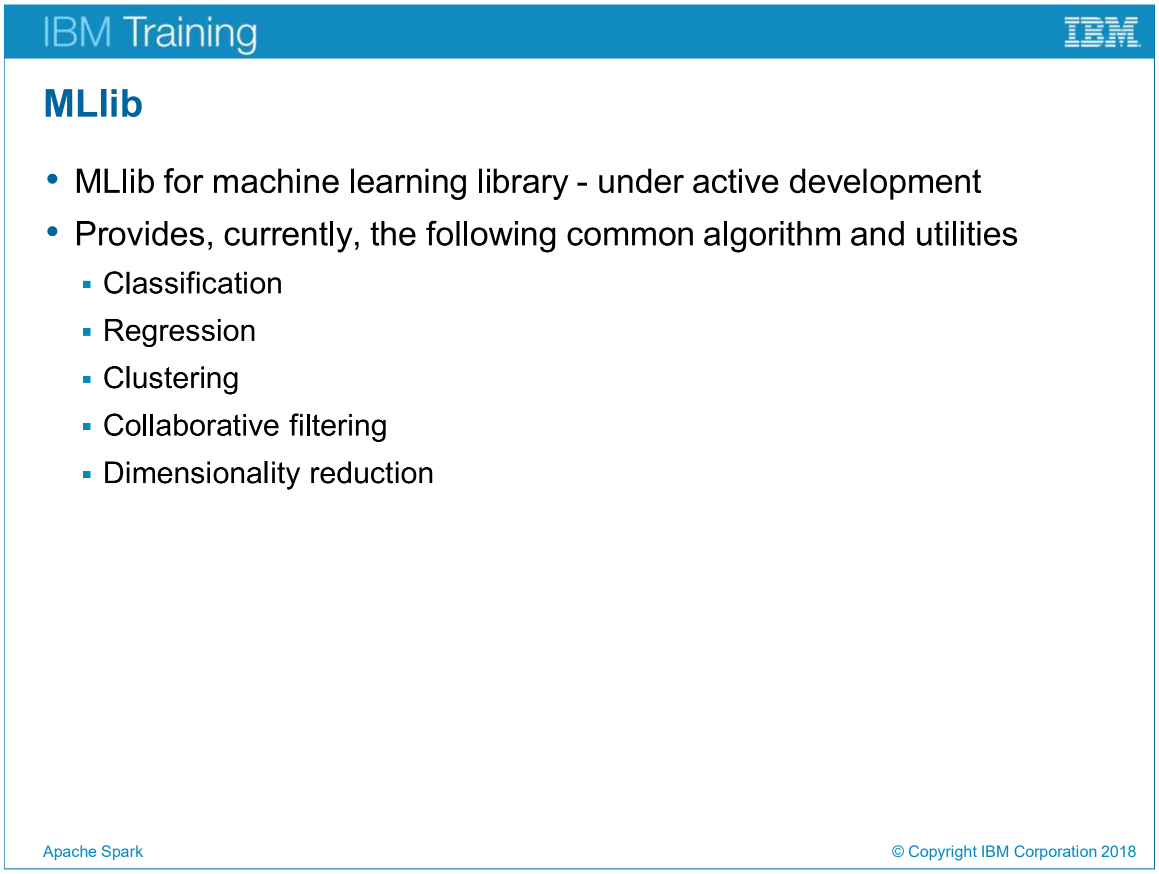
There are two parameters for a sliding window:

- The **window length** is the duration of the window
- The **sliding interval** is the interval in which the window operation is performed.

Both of these parameters must be in multiples of the batch interval of the source DStream.

In the second diagram, the window length is 3 and the sliding interval is 2. To put it into perspective, maybe you want to generate word counts over last 30 seconds of data, every 10 seconds. To do this, you would apply the reduceByKeyAndWindow operation on the pairs of DStream of (Word,1) pairs over the last 30 seconds of data.

Doing wordcount in this manner is provided as an example program NetworkWordCount, available on GitHub at https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/NetworkWordCount.scala



IBM Training

MLlib

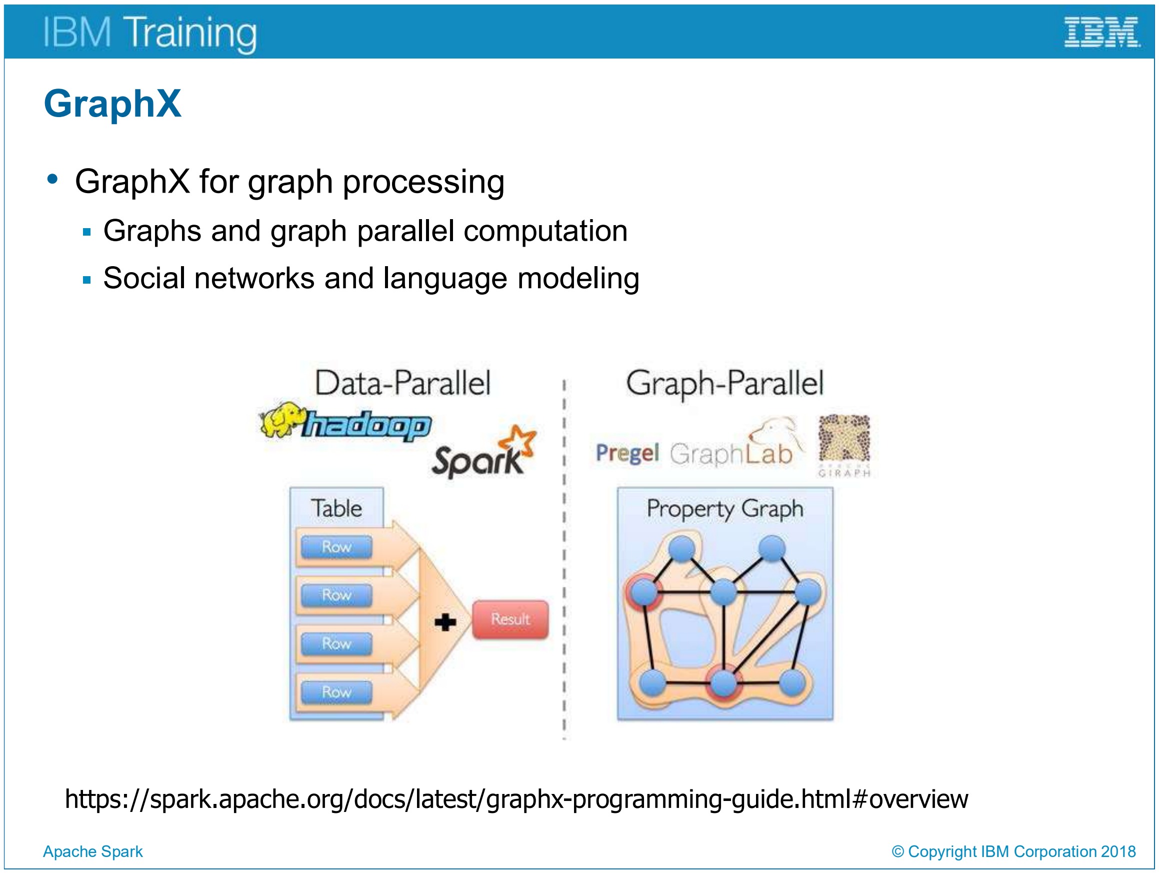
- MLlib for machine learning library - under active development
- Provides, currently, the following common algorithm and utilities
  - Classification
  - Regression
  - Clustering
  - Collaborative filtering
  - Dimensionality reduction

Apache Spark

© Copyright IBM Corporation 2018

MLib

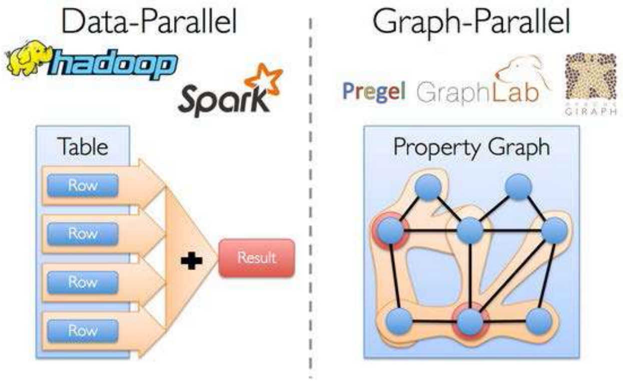
Here is a really short overview of the machine learning library. The MLlib library contains algorithms and utilities for classification, regression, clustering, collaborative filtering and dimensionality reduction. Essentially, you would use this for specific machine learning use cases that requires these algorithms.



IBM Training

GraphX

- GraphX for graph processing
  - Graphs and graph parallel computation
  - Social networks and language modeling



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

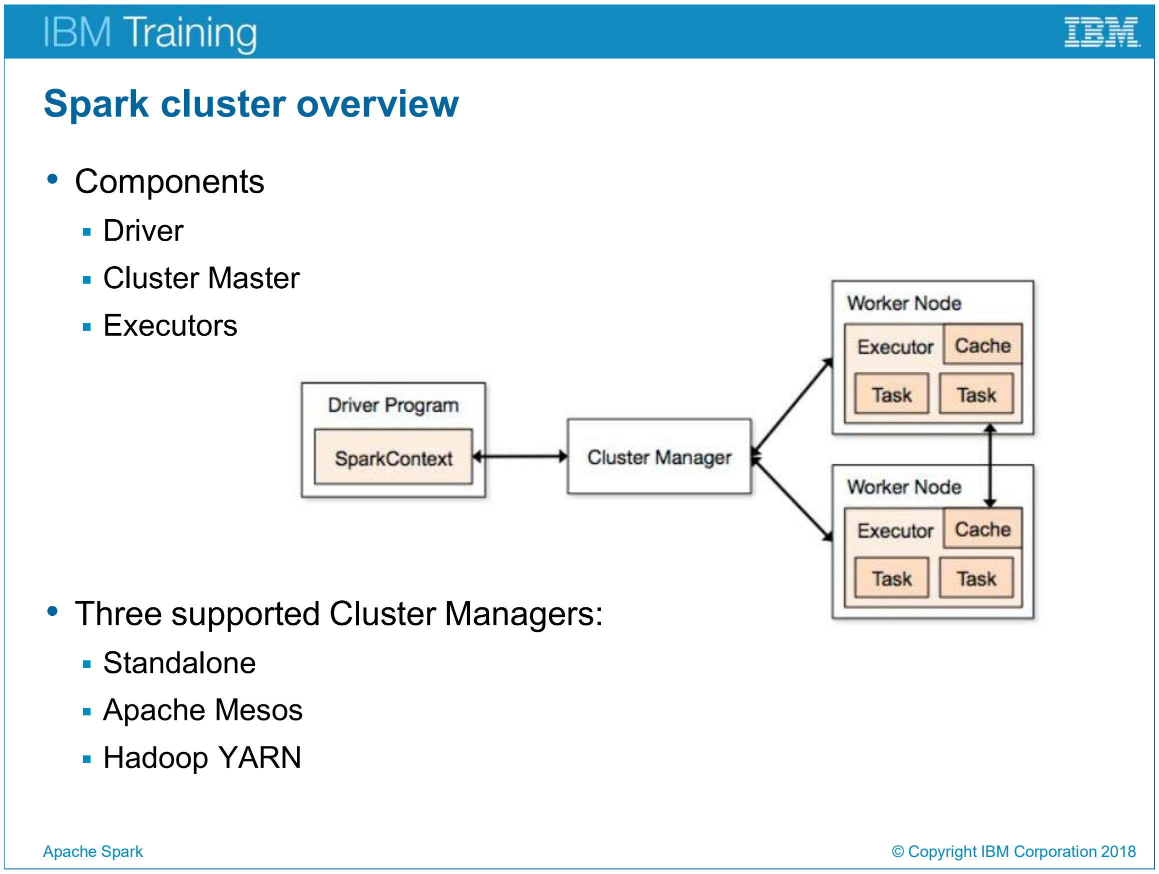
Apache Spark © Copyright IBM Corporation 2018

GraphX

The GraphX is a library that sits on top of the Spark Core. It is basically a graph processing library which can used for social networks and language modeling. Graph data and the requirement for graph parallel systems is becoming more common, which is why the GraphX library was developed. Specific scenarios would not be efficient if it is processed using the data-parallel model. A need for the graph-parallel model is introduced with new graph-parallel systems like Giraph and GraphLab to efficiently execute graph algorithms much faster than general data-parallel systems.

There are new inherent challenges that comes with graph computations, such as constructing the graph, modifying its structure, or expressing computations that span several graphs. As such, it is often necessary to move between table and graph views depending on the objective of the application and the business requirements.

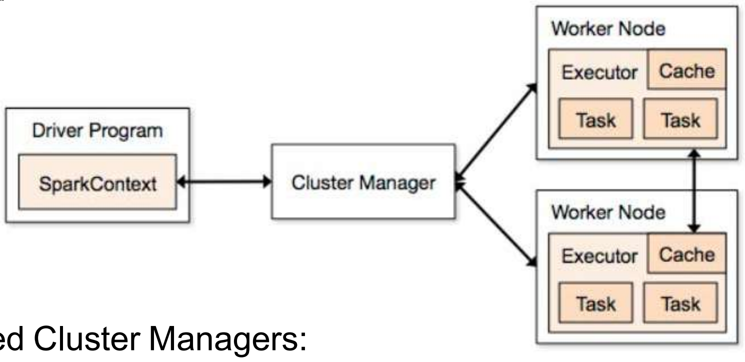
The goal of GraphX is to optimize the process by making it easier to view data both as a graph and as collections, such as RDD, without data movement or duplication.



IBM Training

Spark cluster overview

- Components
  - Driver
  - Cluster Master
  - Executors



- Three supported Cluster Managers:
  - Standalone
  - Apache Mesos
  - Hadoop YARN

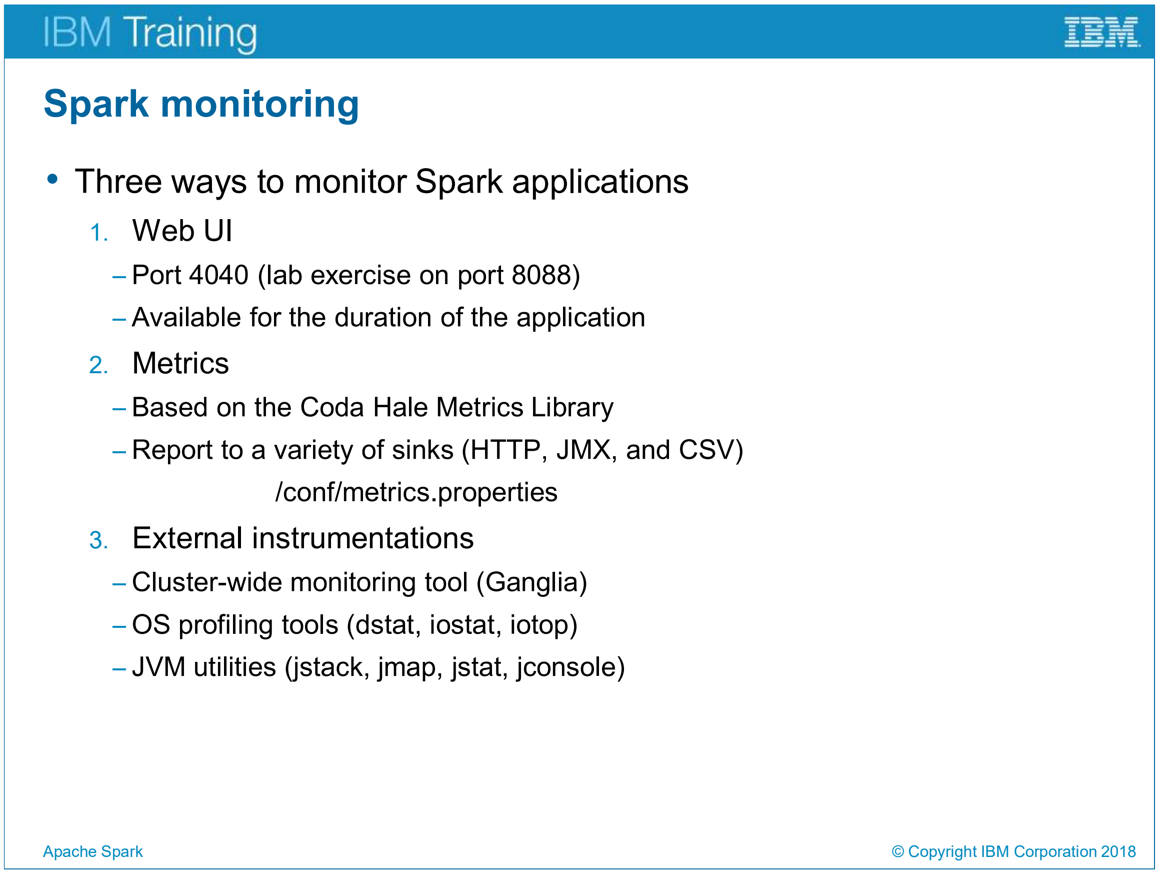
Apache Spark © Copyright IBM Corporation 2018

Spark cluster overview

There are three main components of a Spark cluster. You have the driver, where the SparkContext is located within the main program. To run on a cluster, you would need some sort of cluster manager. This could be either Spark's standalone cluster manager, or Mesos or Yarn. Then you have your worker nodes where the executors reside. The executors are the processes that run computations and store the data for the application. The SparkContext sends the application, defined as JAR or Python files to each executor. Finally, it sends the tasks for each executor to run.

Several things to understand this architecture.

- Each application gets its own executor processes. The executor stays up for the entire duration that the application is running. The benefit of this is that the applications are isolated from each other, on both the scheduling side, and running on different JVMs. However, this means that you cannot share data across applications. You would need to externalize the data if you wish to share data between different applications, instances of SparkContext.
- Spark applications don't care about the underlying cluster manager. As long as it can acquire executors and communicate with each other, it can run on any cluster manager.
- Because the driver program schedules tasks on the cluster, it should run close to the worker nodes on the same local network. If you like to send remote requests to the cluster, it is better to use a RPC and have it submit operations from nearby.
- There are currently three supported cluster managers. Spark comes with a standalone manager that you can use to get up and running. You can use Apache Mesos, a general cluster manager that can run and service Hadoop jobs. Finally, you can also use Hadoop YARN, the resource manager in Hadoop 2. In the lab exercise, you will be using HDP with Yarn to run your Spark applications.



IBM Training

Spark monitoring

- Three ways to monitor Spark applications
  1. Web UI
     - Port 4040 (lab exercise on port 8088)
     - Available for the duration of the application
  2. Metrics
     - Based on the Coda Hale Metrics Library
     - Report to a variety of sinks (HTTP, JMX, and CSV) /conf/metrics.properties
  3. External instrumentations
     - Cluster-wide monitoring tool (Ganglia)
     - OS profiling tools (dstat, iostat, iotop)
     - JVM utilities (jstack, jmap, jstat, jconsole)

Apache Spark © Copyright IBM Corporation 2018

Spark monitoring

There are three ways to monitor Spark applications. The first way is the Web UI. The default port is 4040. The port in the lab environment is 8088. The information on this UI is available for the duration of the application. If you want to see the information after the fact, set the spark.eventLog.enabled to true before starting the application. The information will then be persisted to storage as well.

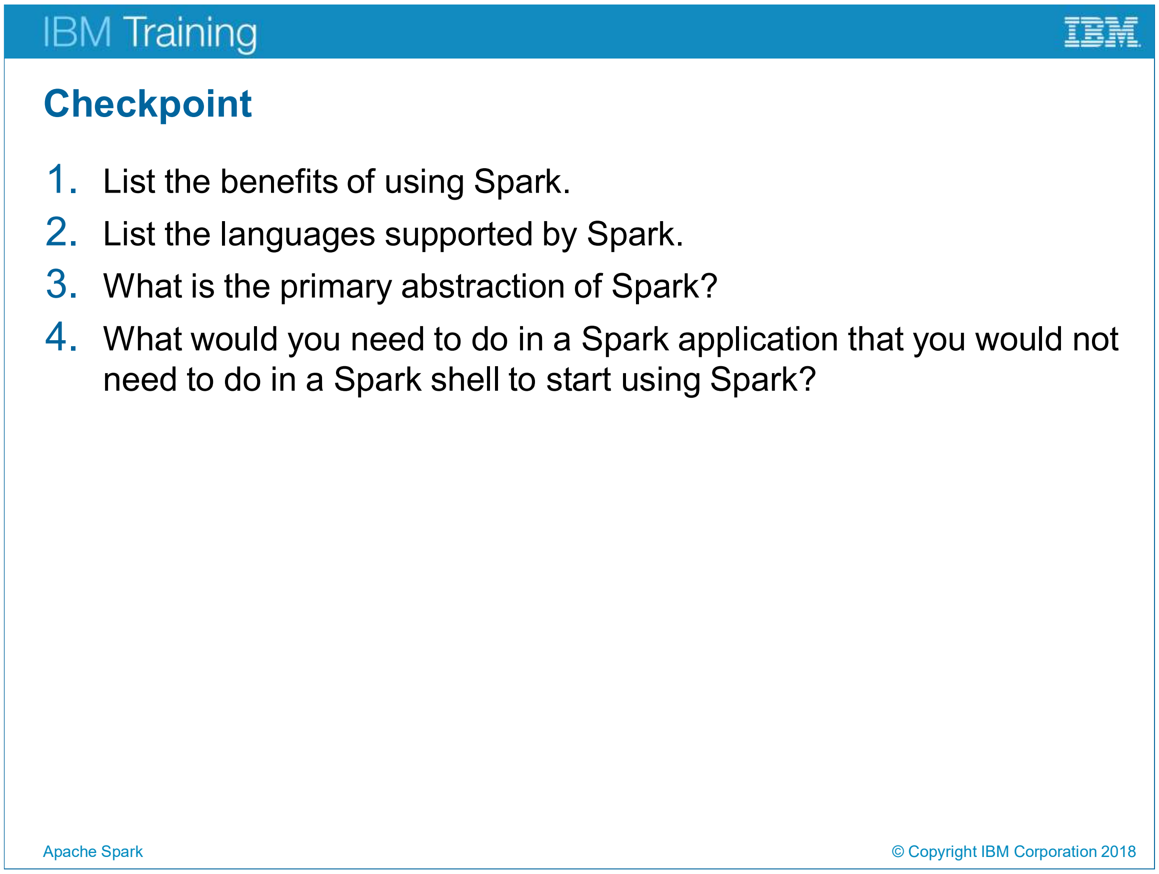
The **Web UI** has the following information.

- A list of scheduler stages and tasks.
- A summary of RDD sizes and memory usage.
- Environmental information and information about the running executors.

To view the history of an application after it has running, you can start up the history server. The history server can be configured on the amount of memory allocated for it, the various JVM options, the public address for the server, and a number of properties.

Metrics are a second way to monitor Spark applications. The metric system is based on the Coda Hale Metrics Library. You can customize it so that it reports to a variety of sinks such as CSV. You can configure the metrics system in the metrics.properties file under the conf directory.

In addition, you can also use external instrumentations to monitor Spark. Ganglia is used to view overall cluster utilization and resource bottlenecks. Various OS profiling tools and JVM utilities can also be used for monitoring Spark.



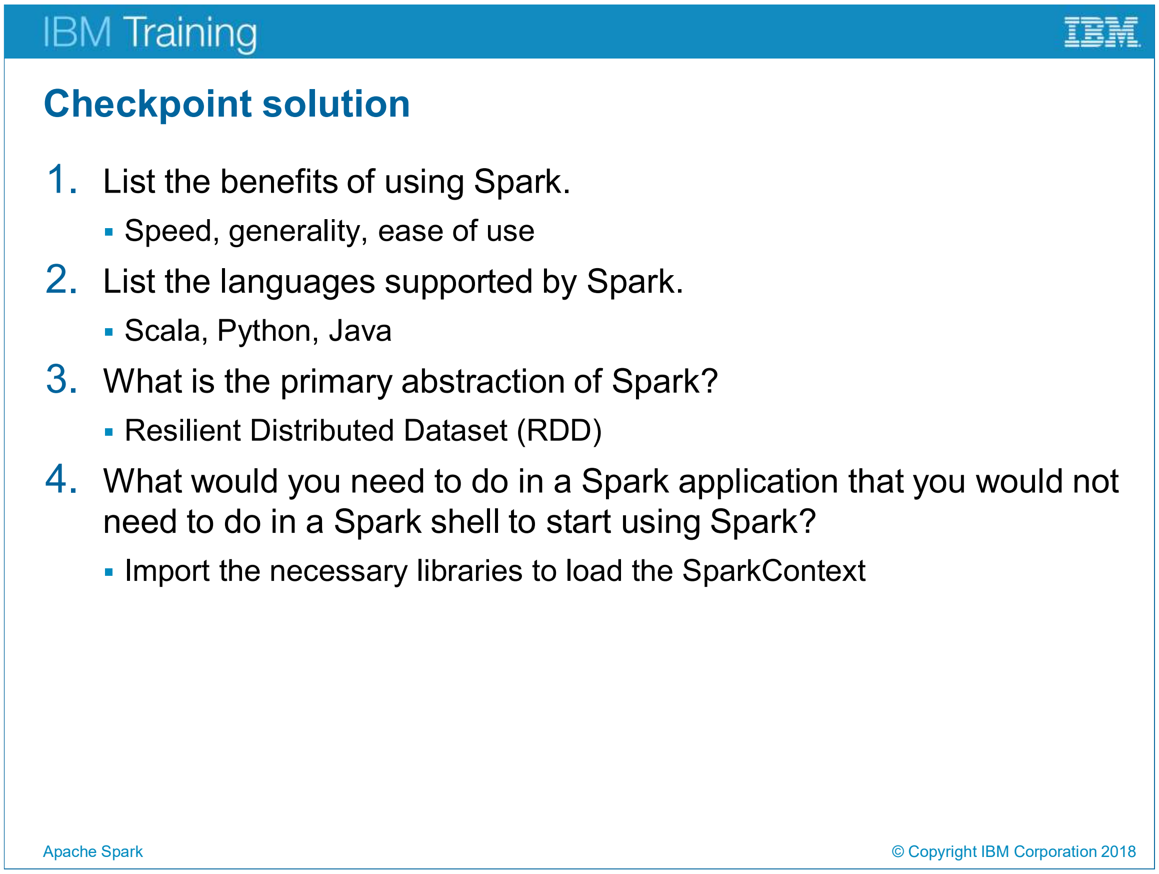
IBM Training

Checkpoint

1. List the benefits of using Spark.
2. List the languages supported by Spark.
3. What is the primary abstraction of Spark?
4. What would you need to do in a Spark application that you would not need to do in a Spark shell to start using Spark?

Apache Spark © Copyright IBM Corporation 2018

Checkpoint



IBM Training

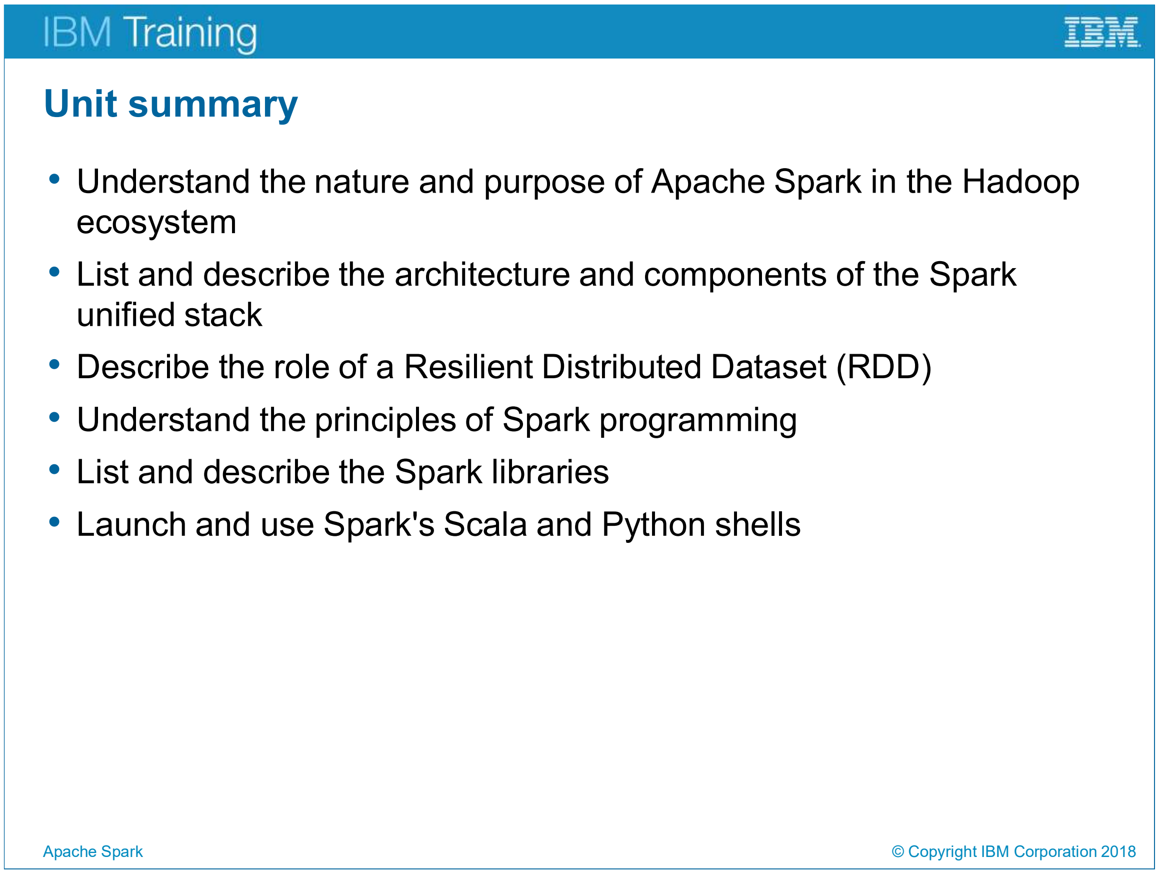
Checkpoint solution

1. List the benefits of using Spark.
   - Speed, generality, ease of use
2. List the languages supported by Spark.
   - Scala, Python, Java
3. What is the primary abstraction of Spark?
   - Resilient Distributed Dataset (RDD)
4. What would you need to do in a Spark application that you would not need to do in a Spark shell to start using Spark?
   - Import the necessary libraries to load the SparkContext

Apache Spark

© Copyright IBM Corporation 2018

Checkpoint solution



IBM Training

Unit summary

- Understand the nature and purpose of Apache Spark in the Hadoop ecosystem
- List and describe the architecture and components of the Spark unified stack
- Describe the role of a Resilient Distributed Dataset (RDD)
- Understand the principles of Spark programming
- List and describe the Spark libraries
- Launch and use Spark's Scala and Python shells

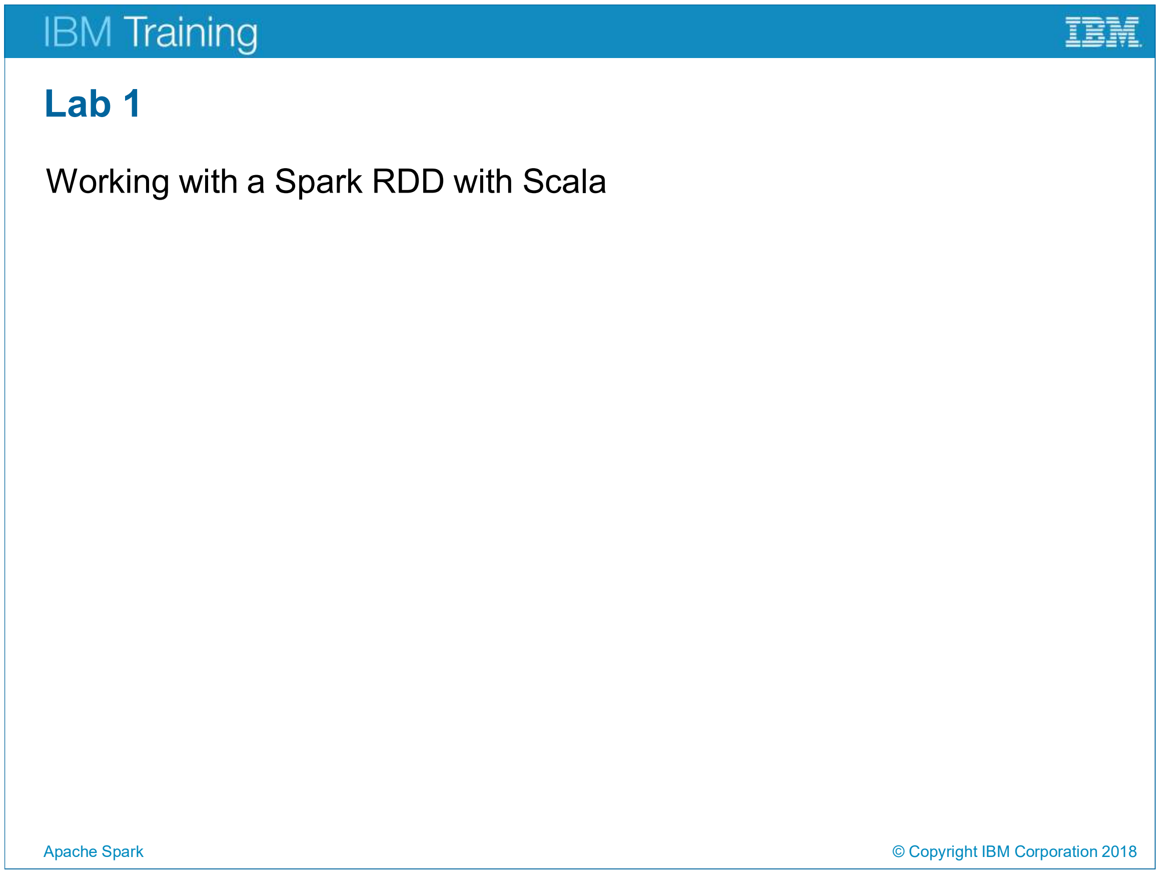
Apache Spark © Copyright IBM Corporation 2018

Unit summary

Additional information and exercises are available in the Cognitive Class course *Spark Fundamentals* available at https://cognitiveclass.ai/learn/spark/. That course gets into further details on most of the topics covered here, especially Spark configuration, monitoring, and tuning.

Cognitive Class has been chosen by IBM as one of the issuers of badges as part of the IBM Open Badge program.

Cognitive Class leverages the services of Pearson VUE Acclaim to assist in the administration of the IBM Open Badge program. By enrolling into this course, you agree to Cognitive Class sharing your details with Pearson VUE Acclaim for the strict use of issuing your badge upon completion of the badge criteria. The course comes with a final quiz where the minimum passing mark for the course is 60%, where the final test is worth 100% of the course mark.



IBM Training

Lab 1

Working with a Spark RDD with Scala

Apache Spark

© Copyright IBM Corporation 2018

Lab 1: Working with a Spark RDD with Scala

Lab 1: Working with a Spark RDD with Scala

Purpose:
In this lab, you will learn to use some of the fundamental aspects of running Spark in the HDP environment.

Additional information on Spark and additional lab exercises using Spark (with Scala, Python, and Java) are available in the Cognitive Class course *Spark Fundamentals* available at https://cognitiveclass.ai/learn/spark/.

Good locations for additional information on Scala and introductory exercises in Spark with Scala and Python can be found at:

Scala Community: http://www.scala-lang.org

Quick Start: https://spark.apache.org/docs/latest/quick-start.html

Blogs, e.g.: http://blog.ajduke.in/2013/05/31/various-ways-to-run-scala-code

Note:

You may need verify that your hostname and IP address are setup correctly as noted in earlier units. Note that if you shut down your lab environment, this verification of hostname and IP address should be repeated. This is particularly important if you find that you cannot connect to Spark. Resetting the IP values may require that you reboot the VMware Image (as root: `reboot`).

Task 1. Connect to the VMware Image & to the Spark server.

1. Connect to and login to your lab environment with user **student** and password **student** credentials.
2. In a new terminal window, type `cd` to change to your home directory.
3. To set an environmental variable $SPARK_HOME, type `export SPARK_HOME=/usr/hdp/current/spark-client.`

4. To start the Spark shell, type **`$SPARK_HOME/bin/spark-shell`**.

```
[biadmin@ibmclass Desktop]$ cd
[biadmin@ibmclass ~]$ $SPARK_HOME/bin/spark-shell
15/06/05 11:06:04 INFO spark.SecurityManager: Changing view acls to: biadmin
15/06/05 11:06:04 INFO spark.SecurityManager: Changing modify acls to: biadmin
15/06/05 11:06:04 INFO spark.SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(biadmin); users with
modify permissions: Set(biadmin)
15/06/05 11:06:04 INFO spark.HttpServer: Starting HTTP Server
15/06/05 11:06:04 INFO server.Server: jetty-8.y.z-SNAPSHOT
15/06/05 11:06:04 INFO server.AbstractConnector: Started
SocketConnector@0.0.0.0:34520
15/06/05 11:06:04 INFO util.Utils: Successfully started service 'HTTP class server'
on port 34520.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.3
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_65)
Type in expressions to have them evaluated.
Type :help for more information.
15/06/05 11:06:07 INFO spark.SecurityManager: Changing view acls to: biadmin
15/06/05 11:06:07 INFO spark.SecurityManager: Changing modify acls to: biadmin

. . .

15/06/05 11:06:10 INFO scheduler.EventLoggingListener: Logging events to
hdfs://ibmclass.localdomain:8020/hdp/apps/4.0.0.0/spark/logs/history-server/local-
1433520369438
15/06/05 11:06:10 INFO repl.SparkILoop: Created spark context..
Spark context available as sc.

scala>
```

You now have a Scala prompt where you can enter Scala interactively.

Note that the Spark context is available as sc.

To exit from Scala at any time, you type **`sys.exit`** and press **Enter** (the official approach) or use Ctrl-D.

The tab key provides code completion.

5. Type `sc.` (the period is needed!), and then press **Tab**.

```
scala> sc.
accumulable                accumulableCollection
accumulator                addFile
addJar                     addSparkListener
appName                    applicationId
asInstanceOf               binaryFiles
binaryRecords              broadcast
cancelAllJobs              cancelJobGroup
clearCallSite              clearFiles
clearJars                  clearJobGroup
defaultMinPartitions       defaultMinSplits
defaultParallelism         emptyRDD
files                      getAllPools
getCheckpointDir           getConf
getExecutorMemoryStatus    getExecutorStorageStatus
getLocalProperty           getPersistentRDDs
getPoolForName             getRDDStorageInfo
getSchedulingMode          hadoopConfiguration
hadoopFile                 hadoopRDD
initLocalProperties        isInstanceOf
isLocal                    jars
killExecutor               killExecutors
makeRDD                    master
metricsSystem              newAPIHadoopFile
newAPIHadoopRDD            objectFile
parallelize                requestExecutors
runApproximateJob          runJob
sequenceFile               setCallSite
setCheckpointDir           setJobDescription
setJobGroup                setLocalProperty
sparkUser                  startTime
statusTracker              stop
submitJob                  tachyonFolderName
textFile                   toString
union                      version
wholeTextFiles
```

Note, if your type `sc` and press **Tab**, without the period after *sc*, you will get an abbreviated output, since only three keywords start with *sc*, whereas a lot of functionality is provided by the Spark context ("sc.").

```
scala> sc
sc       scala    schema
```

Task 2. Load data into an RDD and perform transformations and actions on that data.

1. To do an RDD transformation by reading in a file that was previously loaded to HDFS, type the following:
   `val pp = sc.textFile("Gutenberg/Pride_and_Prejudice.txt")`

```
scala> val pp = sc.textFile("Gutenberg/Pride_and_Prejudice.txt")
15/06/05 12:05:48 INFO storage.MemoryStore: ensureFreeSpace(274073) called with
curMem=0, maxMem=278302556
15/06/05 12:05:48 INFO storage.MemoryStore: Block broadcast_0 stored as values in
memory (estimated size 267.6 KB, free 265.1 MB)
15/06/05 12:05:49 INFO storage.MemoryStore: ensureFreeSpace(41821) called with
curMem=274073, maxMem=278302556
15/06/05 12:05:49 INFO storage.MemoryStore: Block broadcast_0_piece0 stored as bytes
in memory (estimated size 40.8 KB, free 265.1 MB)
15/06/05 12:05:49 INFO storage.BlockManagerInfo: Added broadcast_0_piece0 in memory
on localhost:46250 (size: 40.8 KB, free: 265.4 MB)
15/06/05 12:05:49 INFO storage.BlockManagerMaster: Updated info of block
broadcast_0_piece0
15/06/05 12:05:49 INFO spark.SparkContext: Created broadcast 0 from textFile at
<console>:12
pp: org.apache.spark.rdd.RDD[String] = Gutenberg/Pride_and_Prejudice.txt MappedRDD[1]
at textFile at <console>:12
```

   The result is a pointer to the file. The file is not actually read at this time, as is evidenced by noting that you do not get any errors if you misspell the file name. Now **pp** is a pointer to the RDD.

   We can perform some RDD actions on this data. One simple action is to count the number of items (lines, records) in the RDD.

2. To count the number of items in the RDD, type **`pp.count()`**.

```
scala> pp.count()
15/06/05 12:02:27 INFO mapred.FileInputFormat: Total input paths to process : 1
15/06/05 12:02:27 INFO spark.SparkContext: Starting job: count at <console>:15
15/06/05 12:02:27 INFO scheduler.DAGScheduler: Got job 0 (count at <console>:15) with
2 output partitions (allowLocal=false)
15/06/05 12:02:27 INFO scheduler.DAGScheduler: Final stage: Stage 0(count at
<console>:15)
15/06/05 12:02:27 INFO scheduler.DAGScheduler: Parents of final stage: List()
15/06/05 12:02:27 INFO scheduler.DAGScheduler: Missing parents: List()
15/06/05 12:02:27 INFO scheduler.DAGScheduler: Submitting Stage 0
(Gutenberg/Pride_and_Prejudice.txt MappedRDD[7] at textFile at <console>:12), which
has no missing parents
15/06/05 12:02:27 INFO storage.MemoryStore: ensureFreeSpace(2536) called with
curMem=1263720, maxMem=278302556

...

15/06/05 12:02:28 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
15/06/05 12:02:28 INFO scheduler.DAGScheduler: Stage 0 (count at <console>:15)
finished in 0.600 s
res2: Long = 13030

scala> 15/06/05 12:02:28 INFO scheduler.DAGScheduler: Job 0 finished: count at
<console>:15, took 0.958171 s
```

   The number of lines in the file is 13030.

3. In a new terminal window, to verify the number of lines in the file, use the Linux command **`wc`** on the original file that we uploaded to HDFS:
   `wc -l /home/biadmin/labfiles/Pr*`

```
[biadmin@ibmclass ~]$ wc -l /home/labfiles/Pr*
13030 /home/labfiles/Pride_and_Prejudice.txt
[biadmin@ibmclass ~]$
```

4. Restart the Spark Shell, and then to read the first record of the RDD, type **`pp.first()`.**

```
scala> pp.first()
15/06/05 12:40:14 INFO mapred.FileInputFormat: Total input paths to process : 1
15/06/05 12:40:14 INFO spark.SparkContext: Starting job: first at <console>:15
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Got job 1 (first at <console>:15) with
1 output partitions (allowLocal=true)
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Final stage: Stage 1(first at
<console>:15)
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Parents of final stage: List()
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Missing parents: List()
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Submitting Stage 1
(Gutenberg/Pride_and_Prejudice.txt MappedRDD[3] at textFile at <console>:12), which
has no missing parents
15/06/05 12:40:14 INFO storage.MemoryStore: ensureFreeSpace(2560) called with
curMem=631860, maxMem=278302556
15/06/05 12:40:14 INFO storage.MemoryStore: Block broadcast_3 stored as values in
memory (estimated size 2.5 KB, free 264.8 MB)
15/06/05 12:40:14 INFO storage.MemoryStore: ensureFreeSpace(1901) called with
curMem=634420, maxMem=278302556
15/06/05 12:40:14 INFO storage.MemoryStore: Block broadcast_3_piece0 stored as bytes
in memory (estimated size 1901.0 B, free 264.8 MB)
15/06/05 12:40:14 INFO storage.BlockManagerInfo: Added broadcast_3_piece0 in memory
on localhost:46250 (size: 1901.0 B, free: 265.3 MB)
15/06/05 12:40:14 INFO storage.BlockManagerMaster: Updated info of block
broadcast_3_piece0
15/06/05 12:40:14 INFO spark.SparkContext: Created broadcast 3 from broadcast at
DAGScheduler.scala:838
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Submitting 1 missing tasks from Stage
1 (Gutenberg/Pride_and_Prejudice.txt MappedRDD[3] at textFile at <console>:12)
15/06/05 12:40:14 INFO scheduler.TaskSchedulerImpl: Adding task set 1.0 with 1 tasks
15/06/05 12:40:14 INFO scheduler.TaskSetManager: Starting task 0.0 in stage 1.0 (TID
1, localhost, ANY, 1343 bytes)
15/06/05 12:40:14 INFO executor.Executor: Running task 0.0 in stage 1.0 (TID 1)
15/06/05 12:40:14 INFO rdd.HadoopRDD: Input split:
hdfs://ibmclass.localdomain:8020/user/biadmin/Gutenberg/Pride_and_Prejudice.txt:0+348
901
15/06/05 12:40:14 INFO mapred.LineRecordReader: Found UTF-8 BOM and skipped it
15/06/05 12:40:14 INFO executor.Executor: Finished task 0.0 in stage 1.0 (TID 1).
1796 bytes result sent to driver
15/06/05 12:40:14 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 1.0 (TID
1) in 21 ms on localhost (1/1)
15/06/05 12:40:14 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks
have all completed, from pool
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Stage 1 (first at <console>:15)
finished in 0.021 s
15/06/05 12:40:14 INFO scheduler.DAGScheduler: Job 1 finished: first at <console>:15,
took 0.030293 s
res2: String = PRIDE AND PREJUDICE

scala>
```