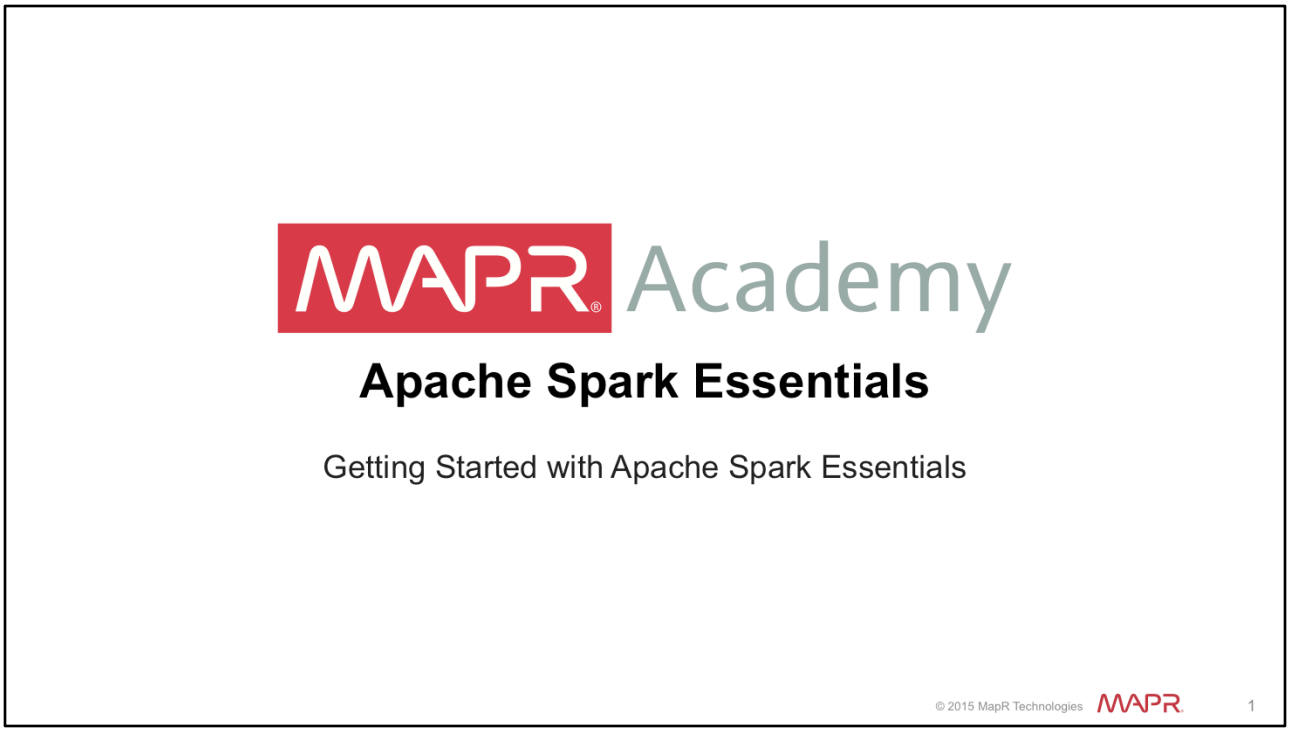
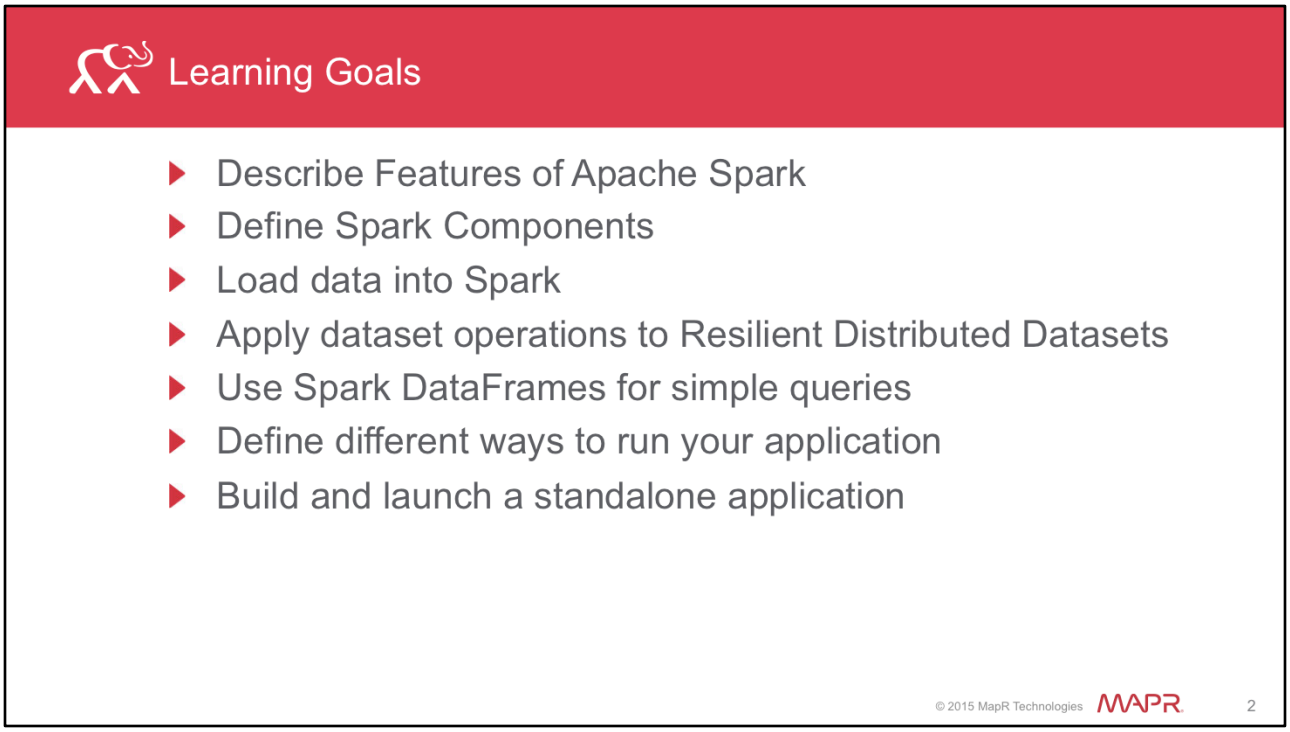
DEV 360 - Apache Spark Essentials

Slide Guide

Version 5.0 – Septemeber 2015

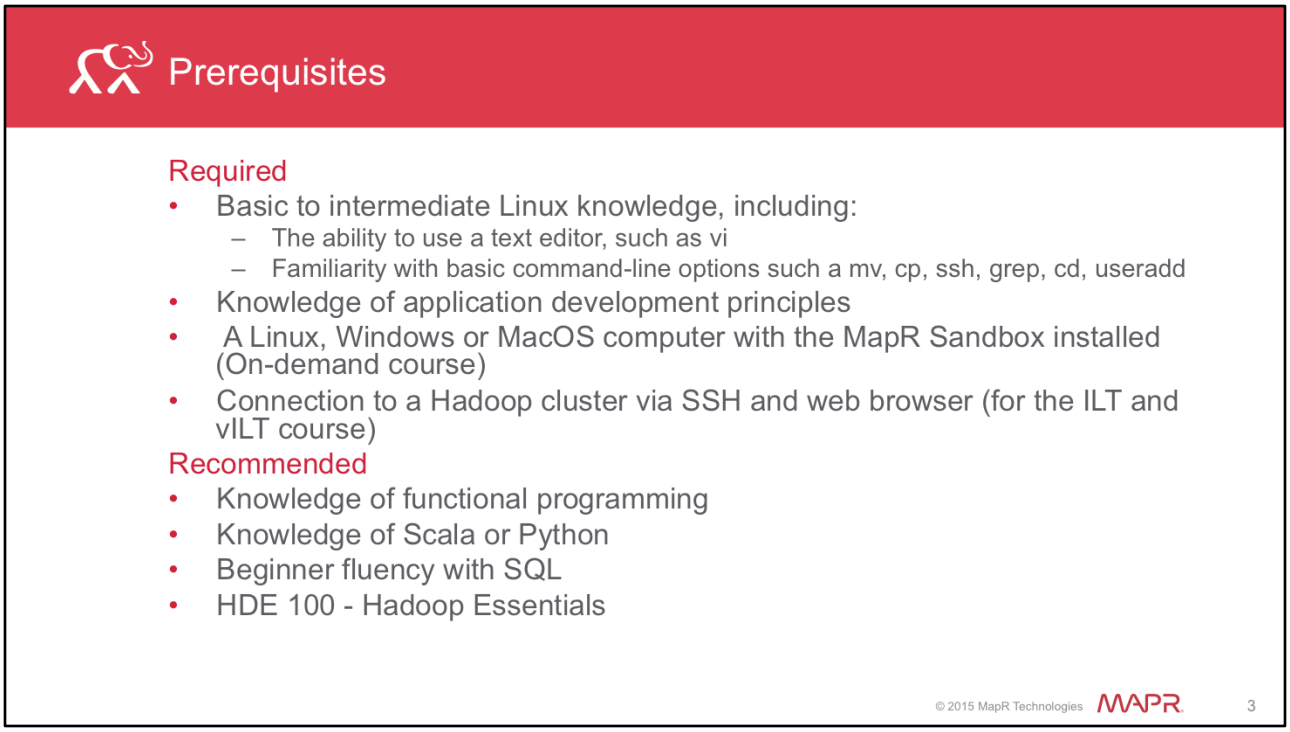


Welcome to Apache Spark Essentials. This introductory course enables developers to get started with Apache Spark. It introduces the benefits of Apache Spark for developing big data applications. In the first part of the course, you will use Spark's interactive shell to load and inspect data. You will then go on to build and launch a standalone Spark application. The concepts are taught using scenarios that also form the basis of hands-on labs.

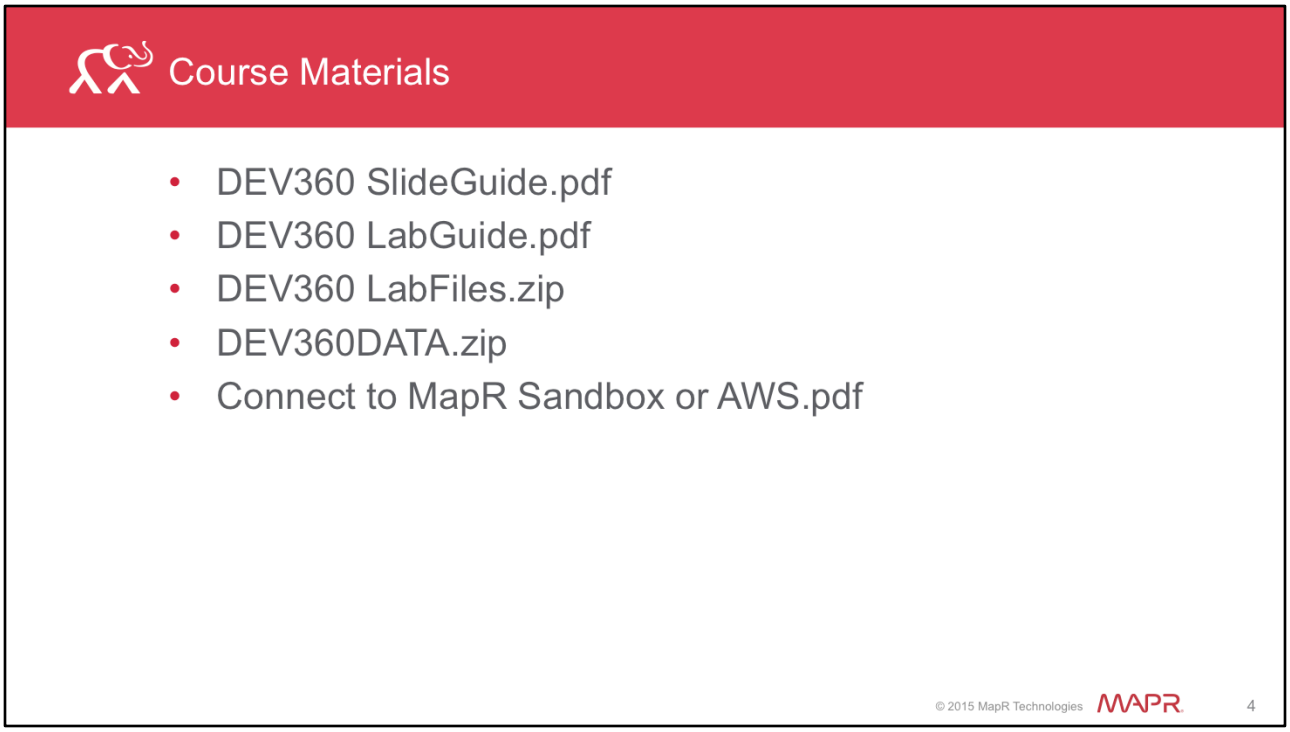


At the end of this course, you will be able to:

- Describe Features of Apache Spark
- Define Spark Components
- Load data into Spark
- Apply dataset operations to Resilient Distributed Datasets
- Use Spark DataFrames for simple queries
- Define different ways to run your application
- Build and launch a standalone application



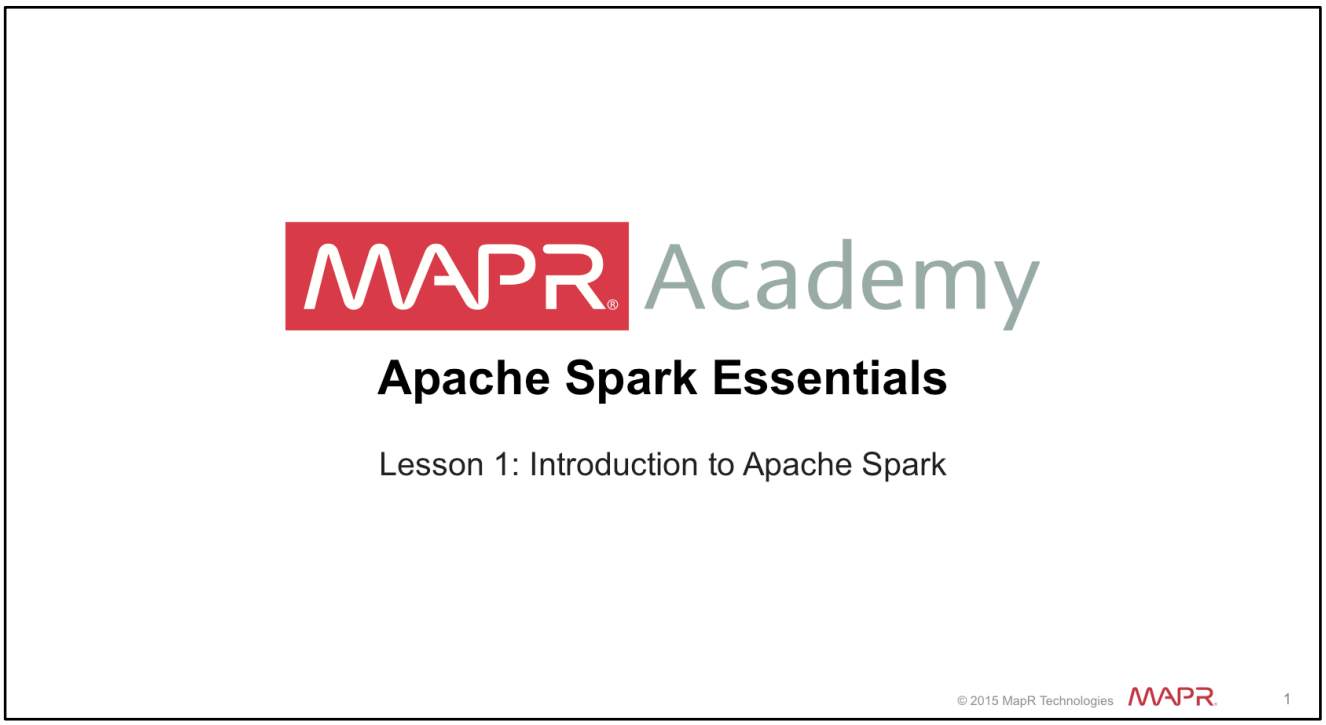
The prerequisites for this course include a basic to intermediate knowledge of Linux, basic knowledge of application development principles. For the on-demand course, you will also need the MapR Sandbox installed. Knowledge of functional programming, Scala or Python and SQL is recommended.



The following course materials are provided:

- The slide guide provides the slides and notes.
- The Lab guide contains lab instructions for the lab activities.
- LabFiles.zip contains the files and solutions for the labs.
- DEV360DATA.zip contains the data files that you need for the labs.
- Connect to MapR Sandbox or AWS.pdf provides connection instructions.

You are now ready to start this course. Proceed to Lesson 1: Introduction to Apache Spark.



Welcome to Apache Spark Essentials, Lesson 1 – Introduction to Apache Spark. This lesson describes Apache Spark, lists the benefits of using Spark and defines the Spark components.

At the end of this lesson, you will be able to:

- Describe the features of Apache Spark, such as:
  - How Spark fits in the Big Data ecosystem, and
  - Why Spark & Hadoop fit together
- Also, you will be able to define the Spark Components

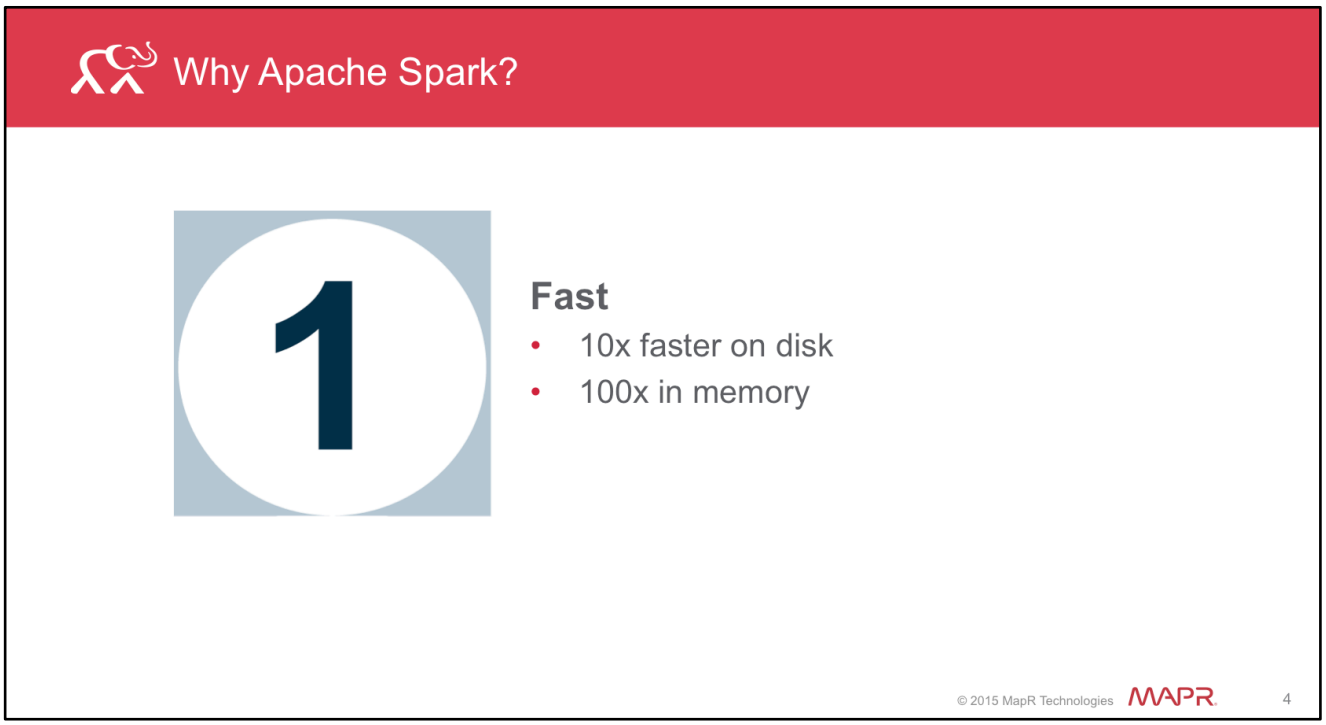
In this section, we will take a look at the features of Apache Spark, how it fits in the Big Data ecosystem

What is Apache Spark?

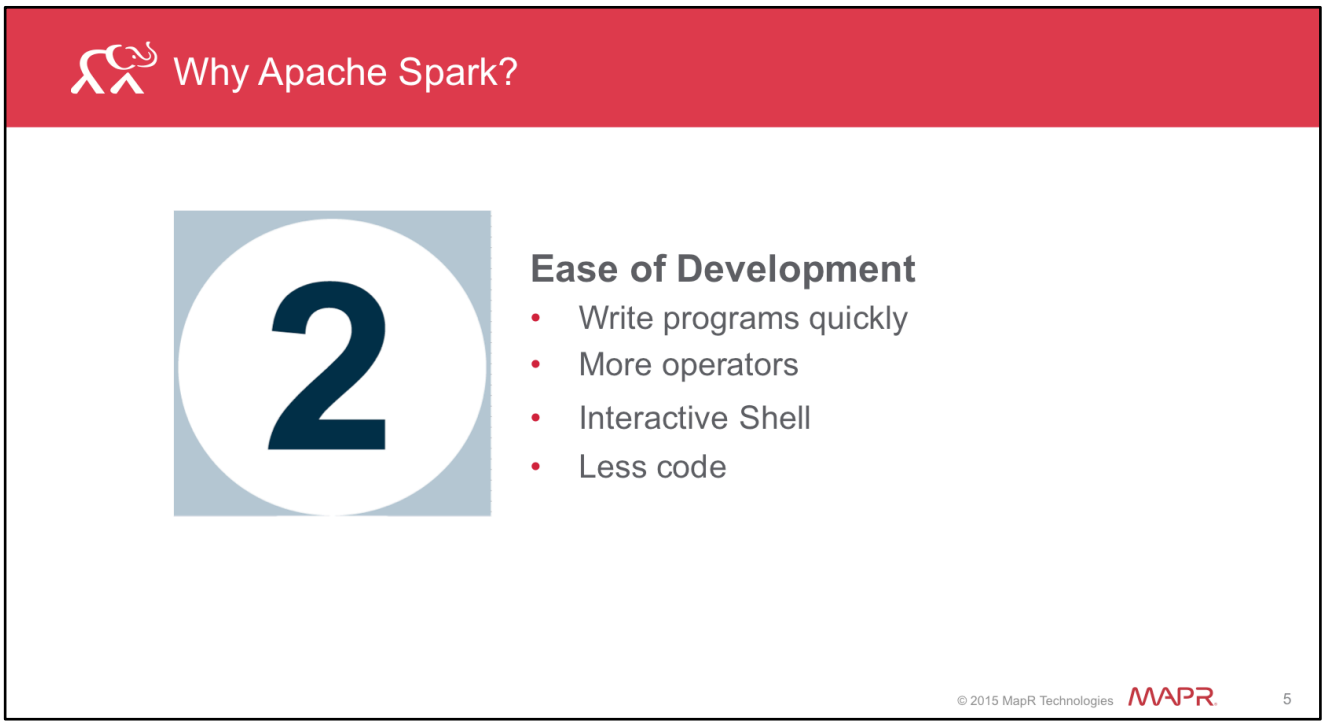
- Cluster computing platform on top of storage layer
- Extends MapReduce with support for more components
  - Streaming
  - Interactive analytics
- Runs in memory

 3

- Apache Spark is a cluster computing platform on top of a storage layer.
- It extends MapReduce with support for more types of components such as streaming and interactive analytics.
- Spark offers the ability to run computations in memory, but is also more efficient than MapReduce for running on disk



- Spark provides reliable in-memory performance. Iterative algorithms are faster as data is not being written to disk between jobs.
- In-memory data sharing across DAGs make it possible for different jobs to work with the same data quickly.
- Spark processes data 10 times faster than MapReduce on disk and 100 times faster in memory.



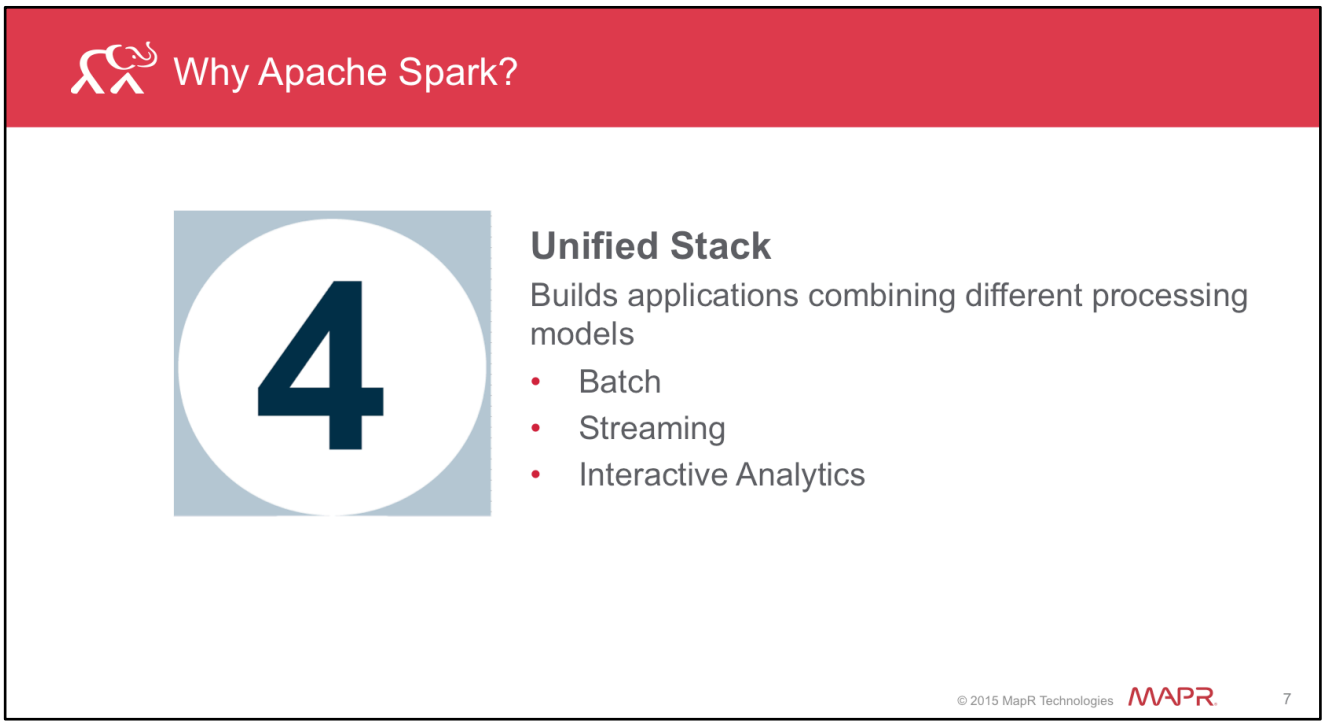
- You can build complex algorithms for data processing very quickly in Spark.
- Spark provides support for many more operators than MapReduce such as Joins, reduceByKey, combineByKey.
- Spark also provides the ability to write programs interactively using the Spark Interactive Shell available for Scala and Python.
- You can compose non-trivial algorithms with little code.

Why Apache Spark?

Deployment Flexibility

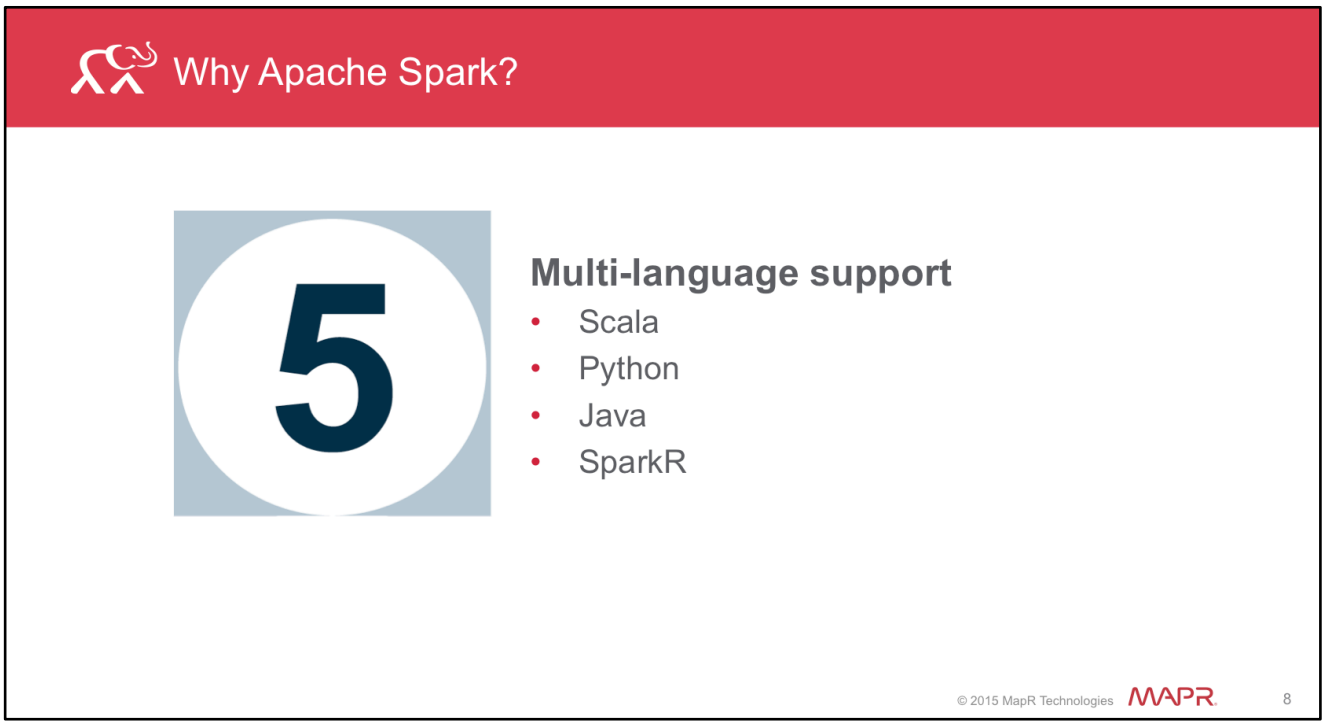
- Deployment
  - Mesos
  - YARN
  - Standalone
  - Local
- Storage
  - HDFS
  - S3

- You can continue to use your existing big data investments.
- Spark is fully compatible with Hadoop.
  - It can run in YARN, and access data from sources including HDFS, MapR-FS, HBase and HIVE.
- In addition, spark can also use the more general resource manager Mesos.

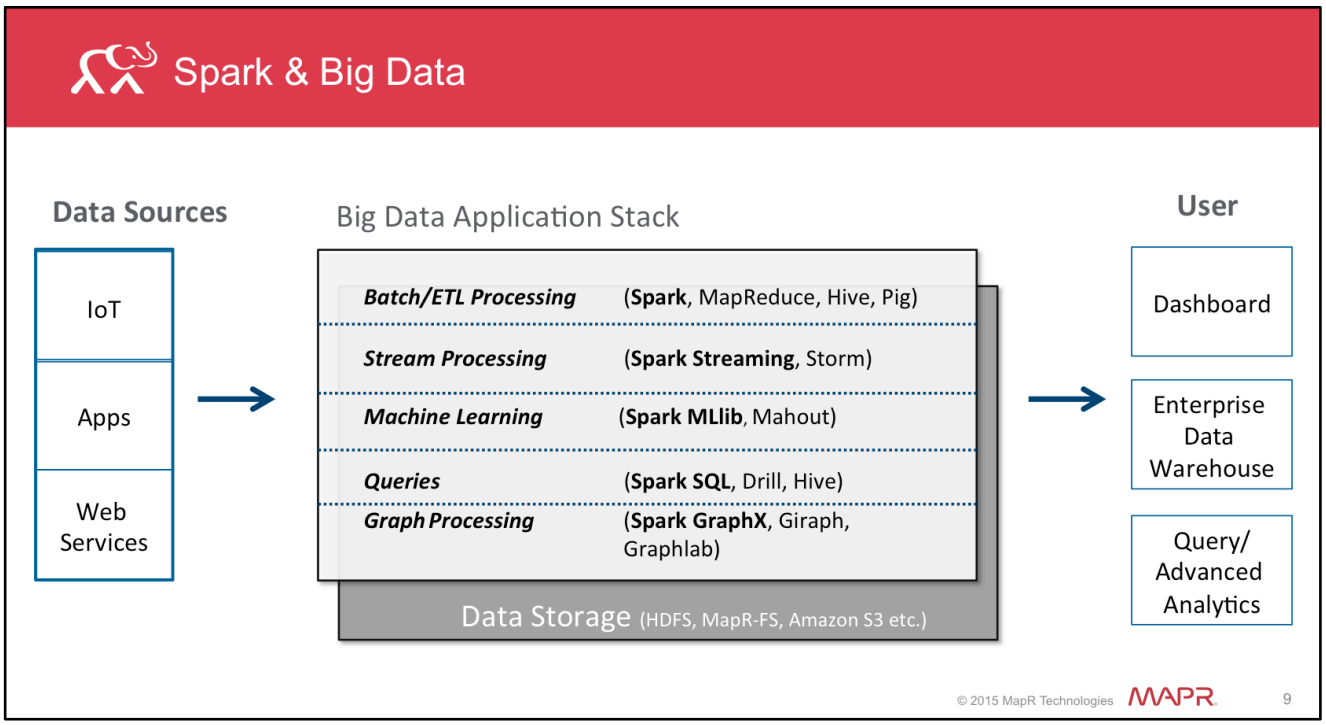


Spark has an integrated framework for advanced analytics like Graph processing, advanced queries, stream processing and machine learning.

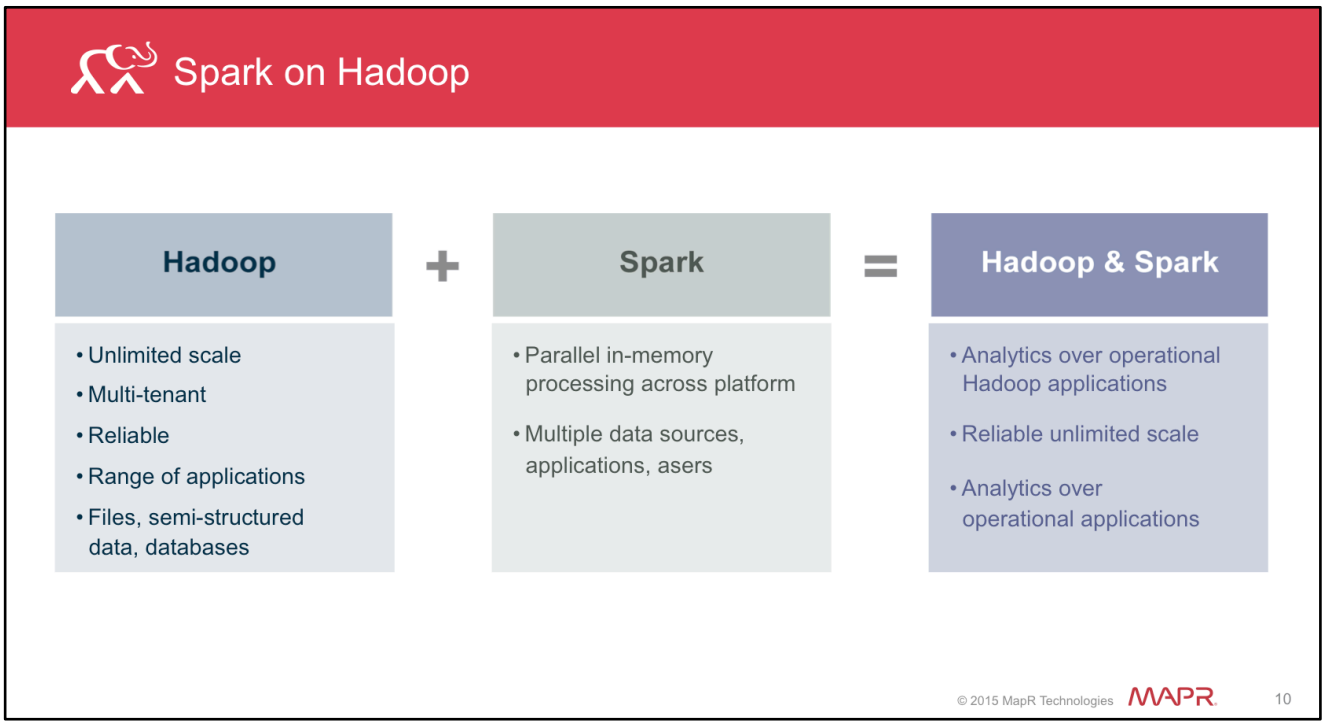
You can combine these libraries into the same application and use a single programming language through the entire workflow



Developers have the choice of using Scala, a relatively new language, Python, Java and as of 1.4.1, SparkR.



This graphic depicts where Spark fits in the Big Data Application stack.
On the left we see different data sources. There are multiple ways of getting data in using different industry standards such as NFS or existing Hadoop tools.
The stack in the middle represents various Big Data processing workflows and tools that are commonly used. You may have just one of these workflows in your application, or a combination of many. Any of these workflows could read/write to or from the storage layer.
As you can see here, with Spark, you have a unified stack. You can use Spark for any of the workflows.
The output can then be used to create real-time dashboards and alerting systems for querying and advanced analytics; and loading into an enterprise data warehouse.



Hadoop
Hadoop has grown into a multi-tenant, reliable, enterprise-grade platform with a wide range of applications that can handle files, databases, and semi-structured data.

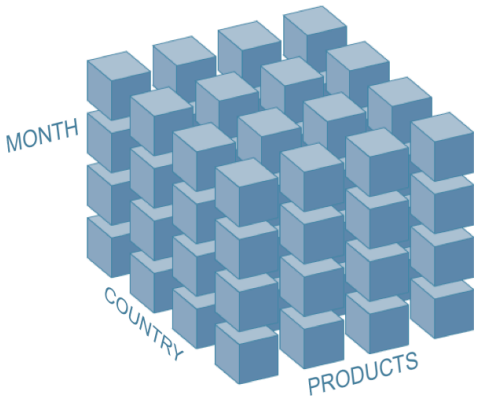
Spark
Spark provides parallel, in-memory processing across this platform, and can accommodate multiple data sources, applications and users.

Hadoop + Spark
The combination of Spark and Hadoop takes advantage of both platforms, providing reliable, scalable, and fast parallel, in-memory processing. Additionally, you can easily combine different kinds of workflows to provide analytics over your Hadoop and other operational applications.

Use Case

OLAP Analytics

- Service provider using MapR & Spark delivers real-time multi-dimensional OLAP analytics
- Must accept data of any type/format
- Perform rigorous analytics across large datasets



A service provider offers services in customer analytics, technology and decision support to their clients. They are providing real-time multi-dimensional OLAP analytics. They should be able to accept data of any type/format and perform rigorous analytics across datasets that can go up to 1-2 TBs in size.

Use Case

Operational Analytics

- Health Insurance company uses MapR to store patient information
- Spark computes patient re-admittance probability
- Real-time analytics over NoSQL
- Spark with MapR-DB

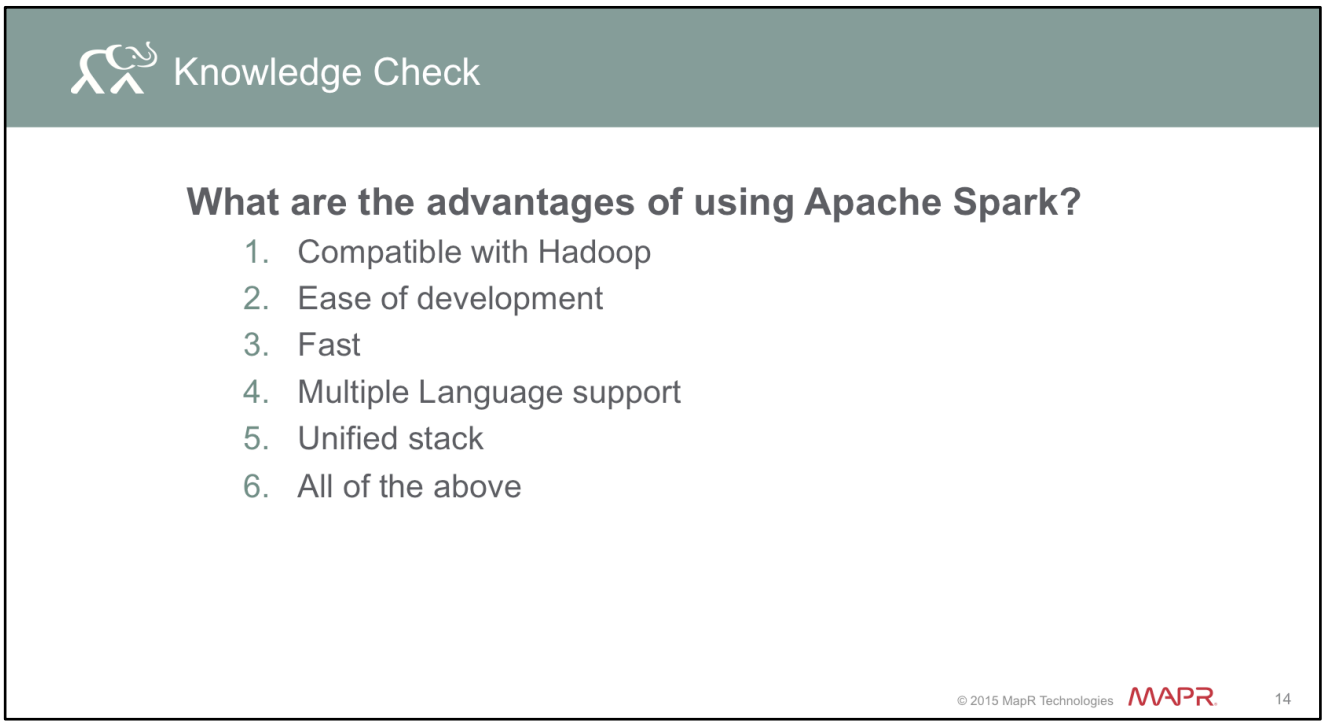
A health insurance company is using MapR to store patient information, which is combined with clinical records. Using real time analytics over NoSQL, they are able to compute re-admittance probability. If this probability is high enough, additional services, such as in home nursing, are deployed.

Use Case

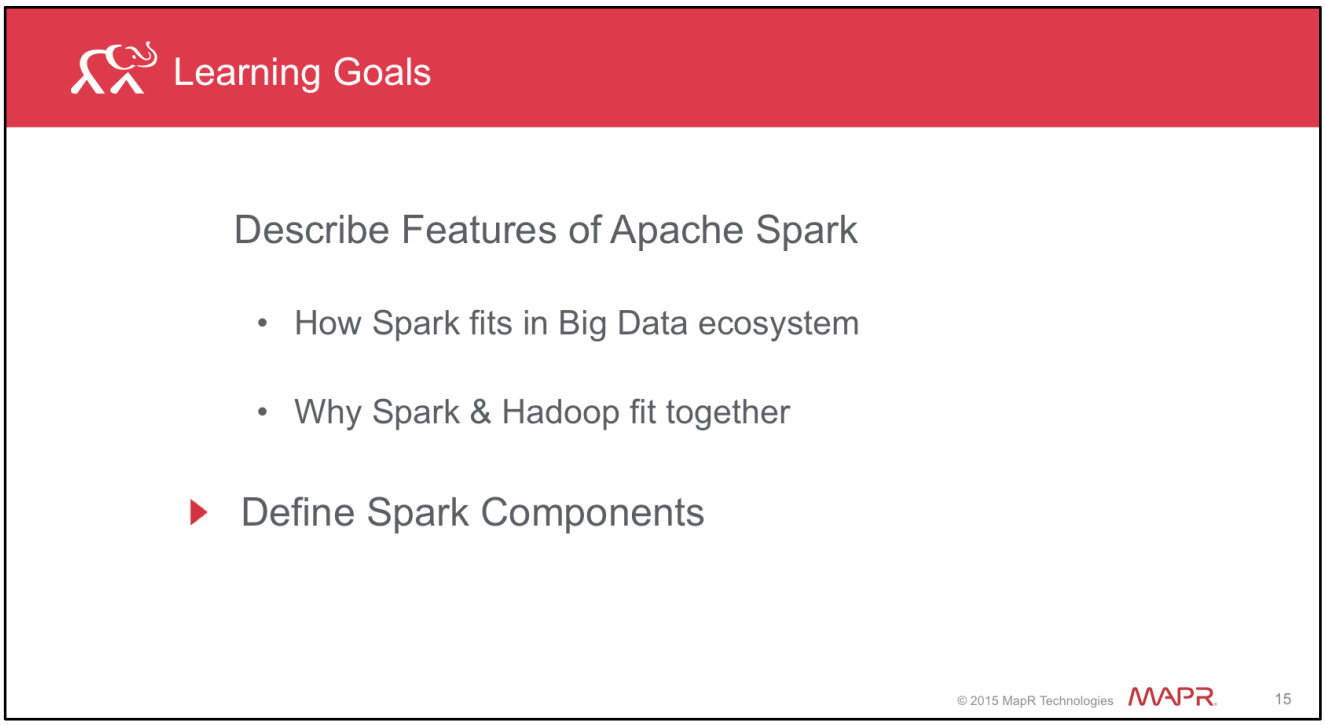
Complex Data Pipelining

- Pharmaceutical company uses Spark on MapR for gene sequencing analysis
- Spark reduces processing from weeks to hours
- Complex machine learning without MapReduce

A leading pharmaceutical company uses Spark to improve gene sequencing analysis capabilities, resulting in faster time to market. Before Spark, it would take several weeks to align chemical compounds with genes. With ADAM running on Spark, gene alignment only takes a matter of a few hours.

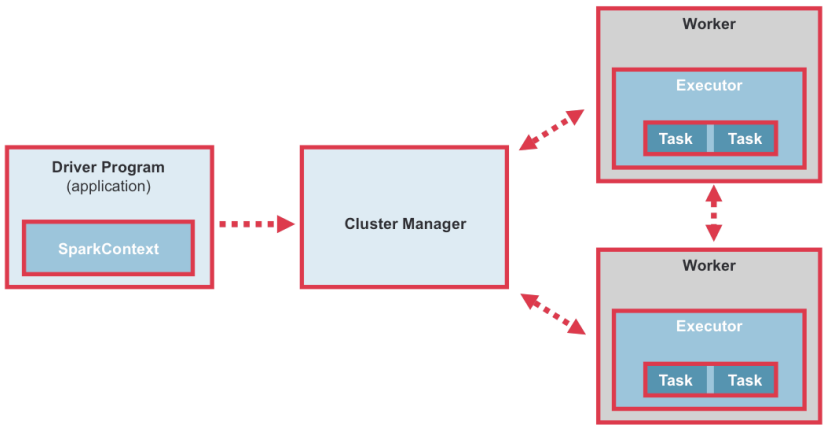

Knowledge Check
What are the advantages of using Apache Spark?
1. Compatible with Hadoop
2. Ease of development
3. Fast
4. Multiple Language support
5. Unified stack
6. All of the above
© 2015 MapR Technologies
14

Answer: 6



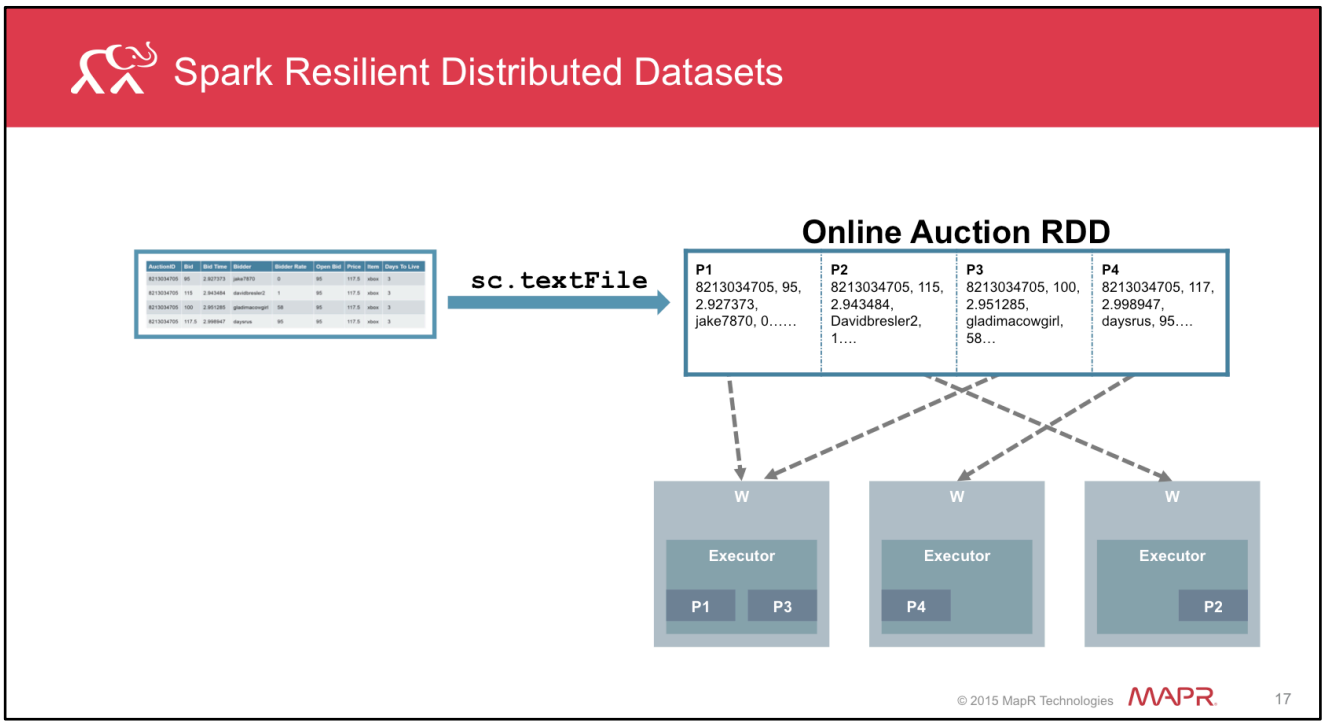
In this section, we take a look at Spark components and libraries.

Spark Components



A Spark cluster consists of two processes, a driver program and multiple workers nodes each running an executor process. The driver program runs on the driver machine, the worker programs run on cluster nodes or in local threads.

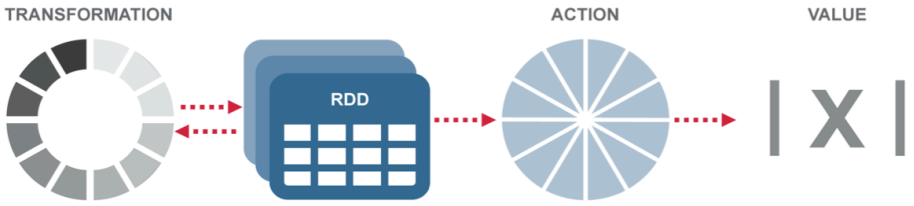
1. The first thing a program does is to create a SparkContext object. This tells Spark how and where to access a cluster
2. SparkContext connects to cluster manager. Cluster manager allocates resources across applications
3. Once connected, Spark acquires executors in the worker nodes (an executor is a process that runs computations and stores data for your application)
4. Jar or python files passed to the SparkContext are then sent to the executors.
5. SparkContext will then send the tasks for the executor to run.
6. The worker nodes can access data storage sources to ingest and output data as needed.



Resilient distributed datasets, or RDD, are the primary abstraction in Spark. They are a collection of objects that is distributed across nodes in a cluster, and data operations are performed on RDD.

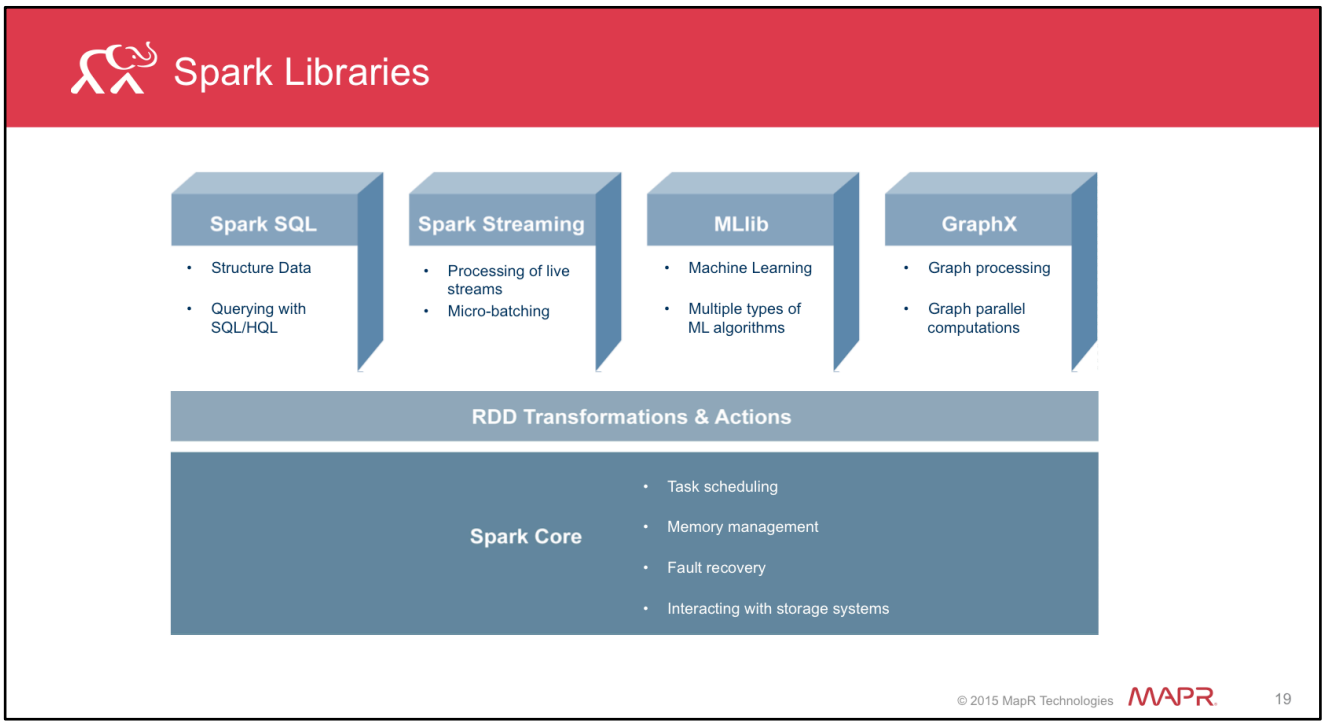
- Once created, RDD are immutable.
- You can also persist, or cache, RDDs in memory or on disk.
- Spark RDDs are fault-tolerant. If a given node or task fails, the RDD can be reconstructed automatically on the remaining nodes and the job will complete.

Spark Resilient Distributed Datasets



There are two types of data operations you can perform on an RDD, transformations and actions.

- A transformation will return an RDD. Since RDD are immutable, the transformation will return a new RDD.
- An action will return a value.



The **Spark core** is a computational engine that is responsible for task scheduling, memory management, fault recovery and interacting with storage systems. The Spark core contains the functionality of Spark. It also contains the APIs that are used to define RDDs and manipulate them.

The higher level components are integrated into this stack.

- Spark SQL can be used for working with structured data. You can query this data via SQL or HiveQL. Spark SQL supports many types of data sources such as structured Hive tables and complex JSON data.
- Spark streaming enables processing of live streams of data and doing real-time analytics.
- MLlib is a machine learning library that provides multiple types of machine learning algorithms such as classification, regression, clustering.
- GraphX is a library for manipulating graphs and performing graph-parallel computations.

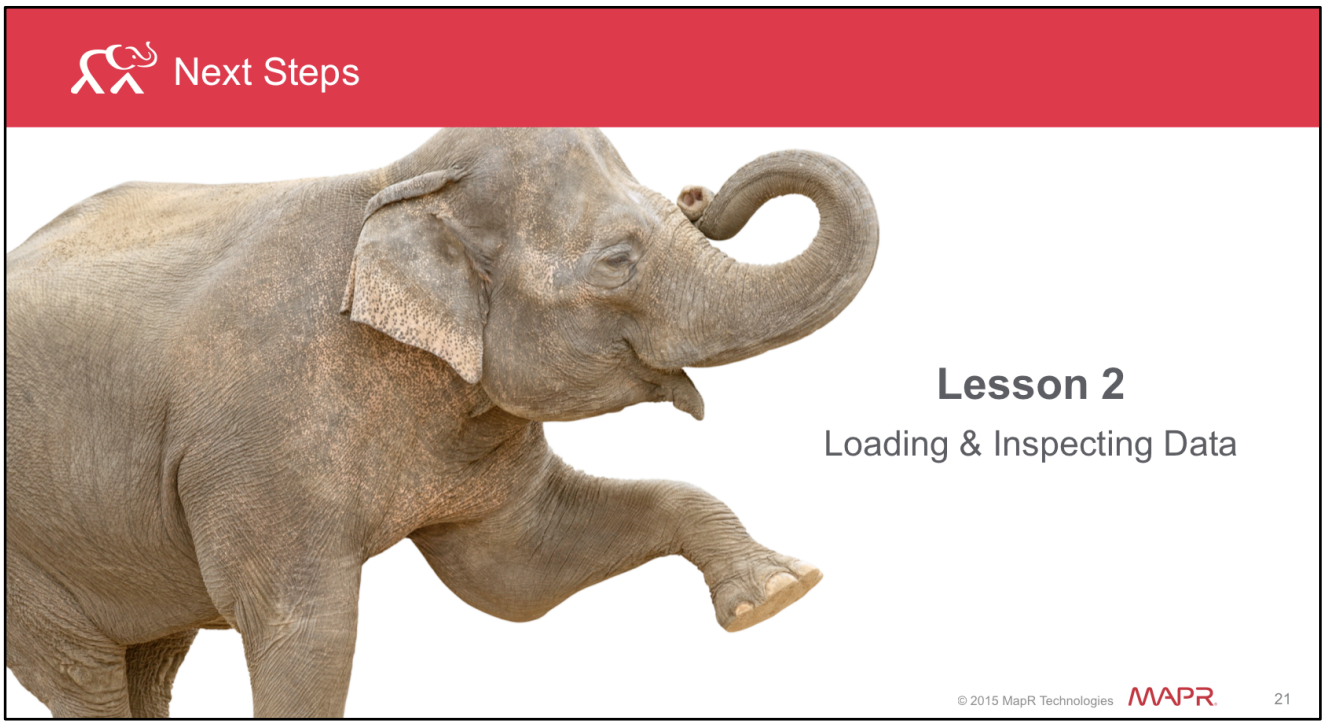
Knowledge Check

Match the following:

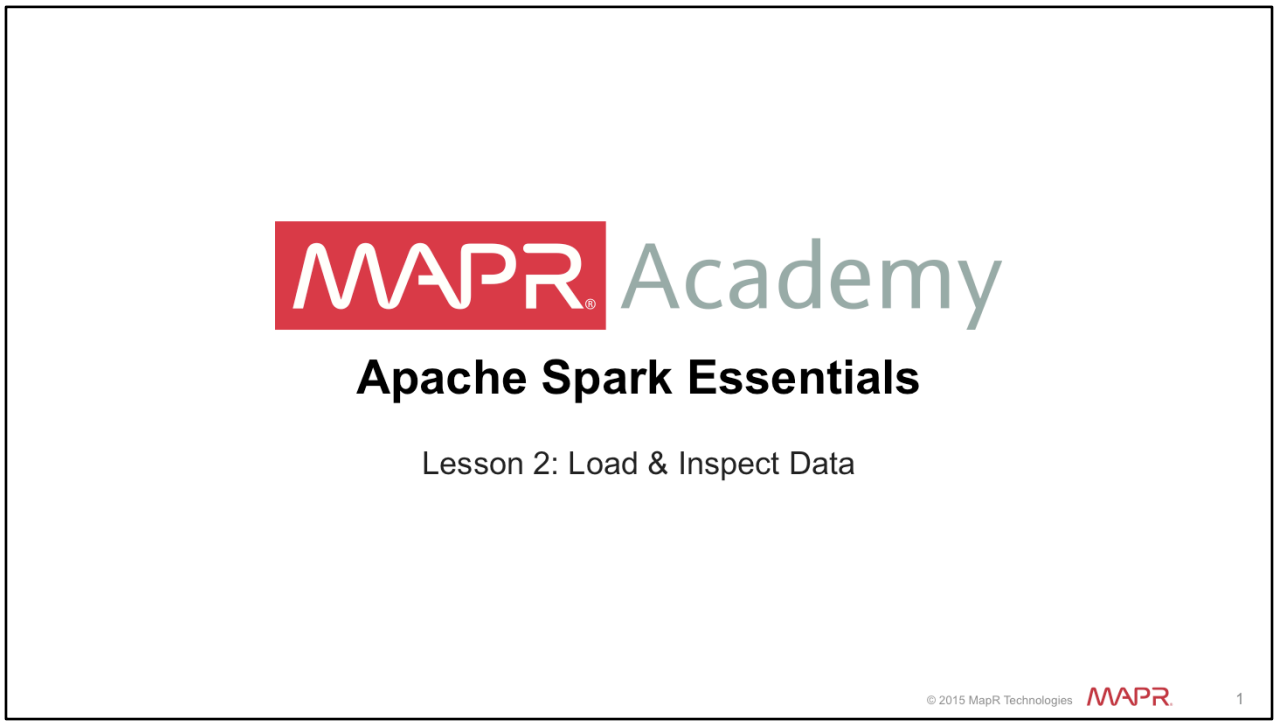
A. Responsible for task scheduling, memory management
B. Tells Spark how & where to access a cluster
C. Collection of objects distributed across many nodes in a cluster

1	SparkContext	2	RDD	3	Spark Core

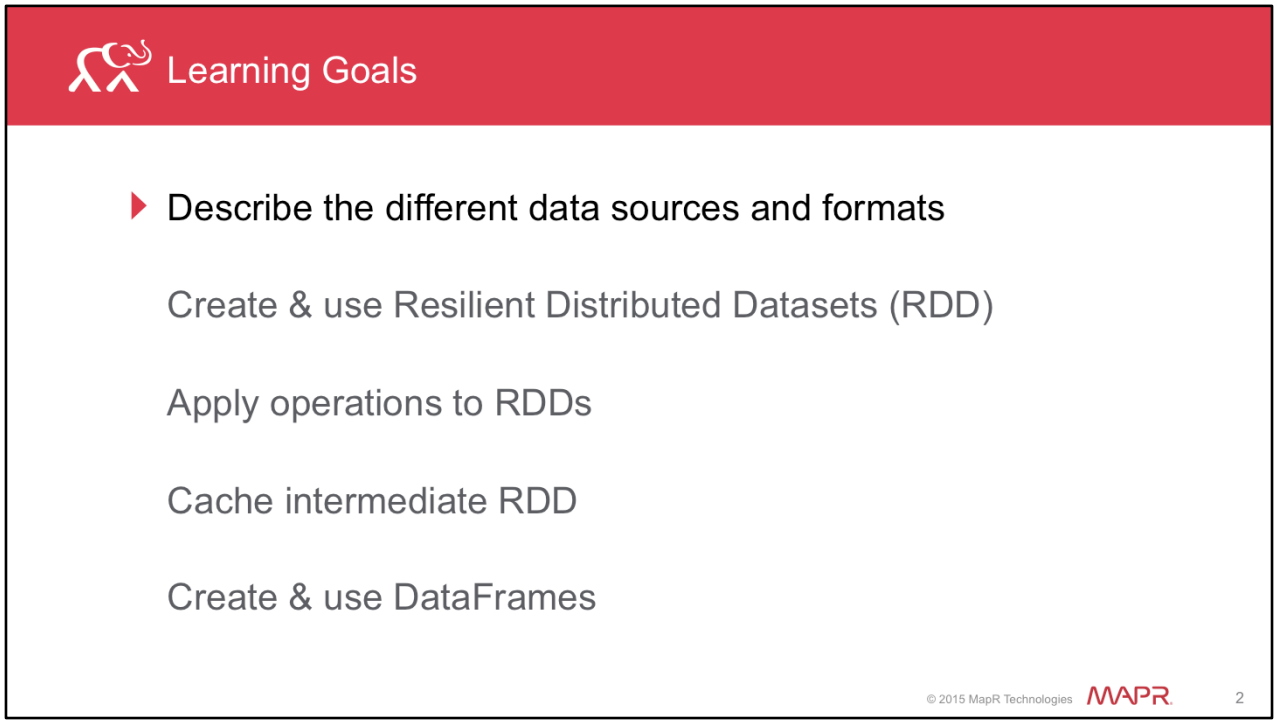
1-b, 2-c,3-a



Congratulations! You have completed Lesson 1: Introduction to Apache Spark. Go onto Lesson 2 to learn about loading and inspecting data.



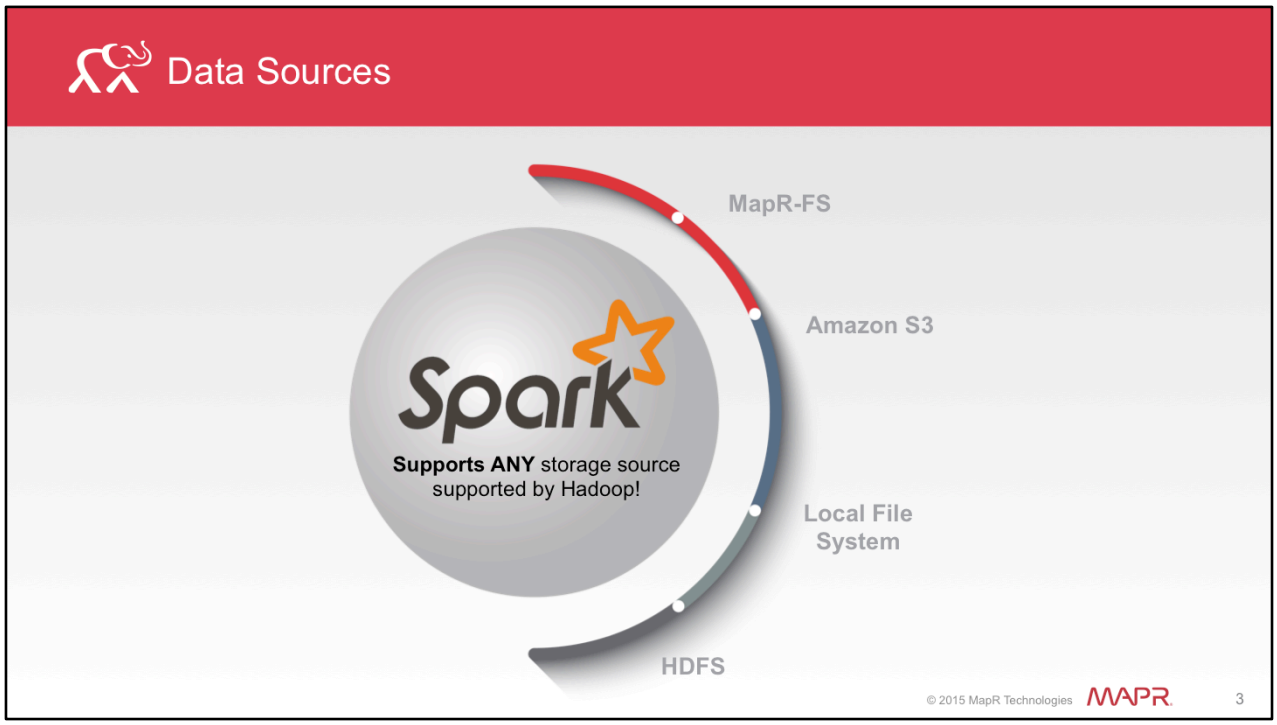
In this lesson, we look at loading data into Spark using Resilient Distributed Datasets (RDDs) and DataFrames. We will use transformations and actions to explore and process this data



At the end of this lesson, you will be able to

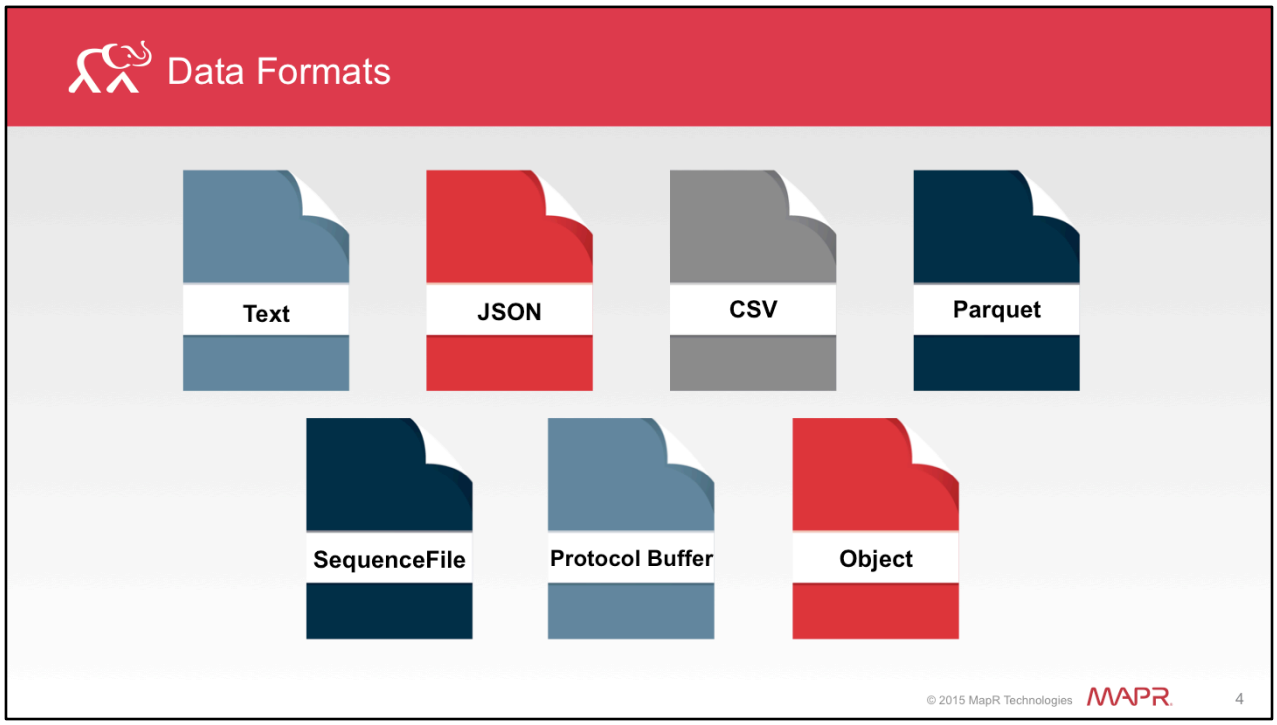
- Describe the different data sources and formats available to use with Apache Spark
- Create and use RDDs
- Apply dataset operations to RDDs
- Cache intermediate RDDs
- Use Apache Spark DataFrames

In the first section, we will look at the different data sources available to use with Spark.

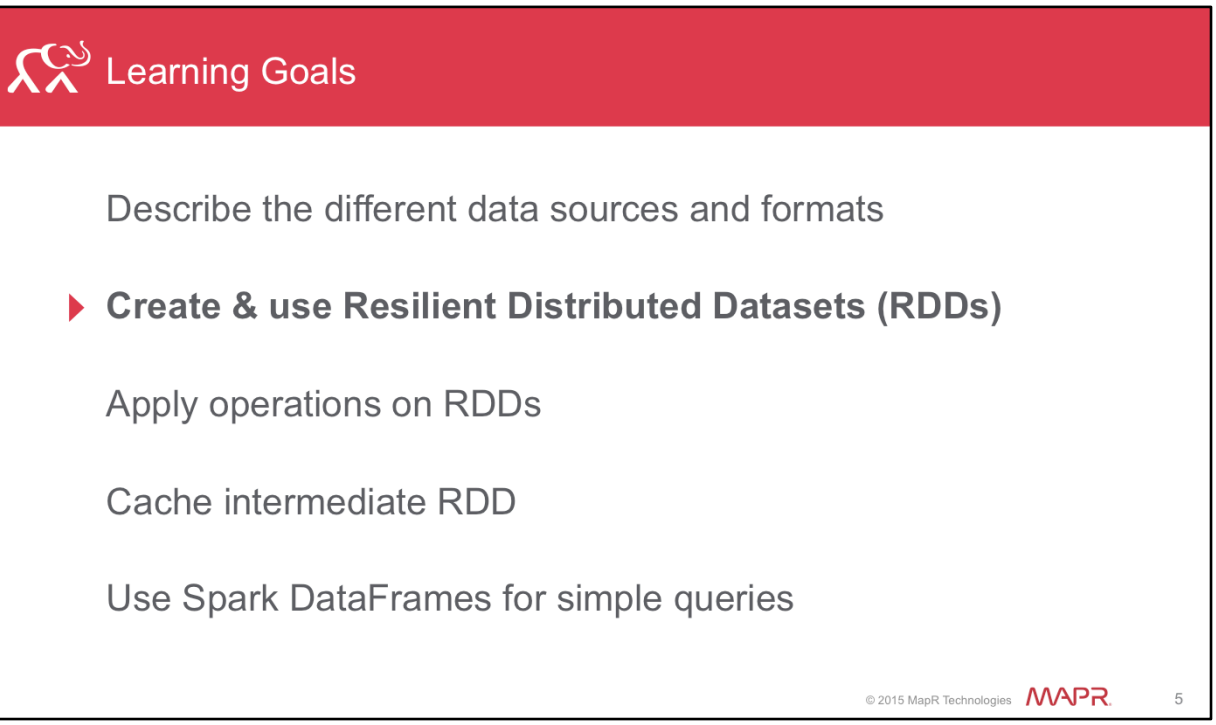


Before we load data into Spark, take a look at the data sources and data that Spark supports.

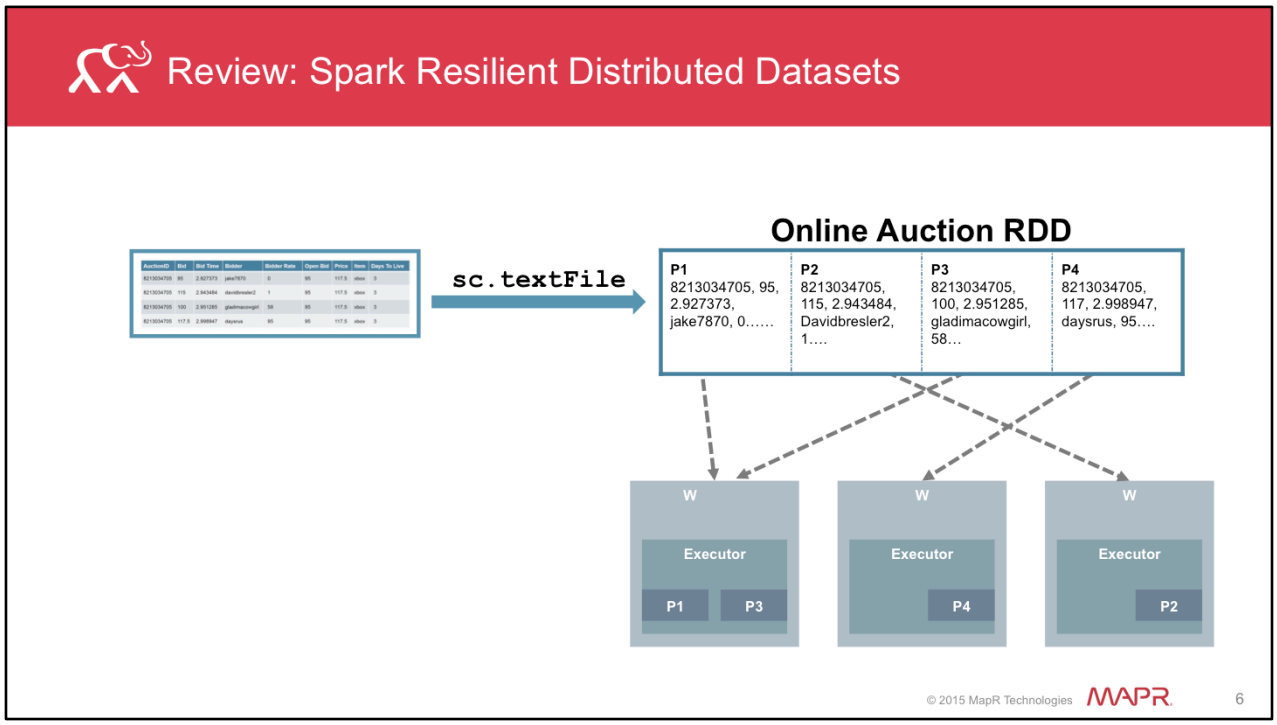
You can load data from any storage source that is supported by Hadoop. You can upload data from the local file system, HDFS, MapR-FS, Amazon S3, Hive, HBase or MapR-DB, JDBC databases, and any other data source that you are using with your Hadoop cluster.



Spark provides wrappers for text files, JSON, CSV, Parquet files, SequenceFiles, protocol buffers and object files. In addition, Spark can also interact with any Hadoop-supported formats.



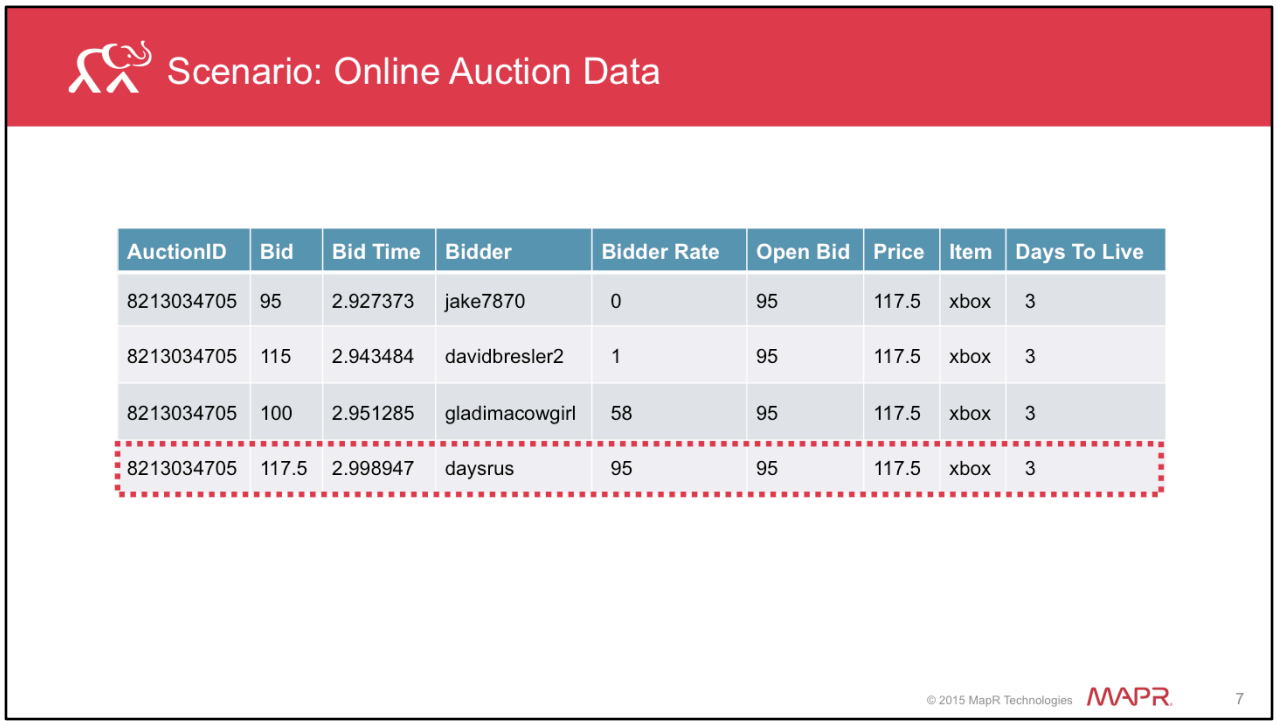
In this section, we will take a look at RDDs and load data into a Spark RDD.



Resilient distributed datasets are the primary abstraction in Spark. They are a collection of objects that is distributed across many nodes in a cluster. They are immutable once created and cannot be changed.

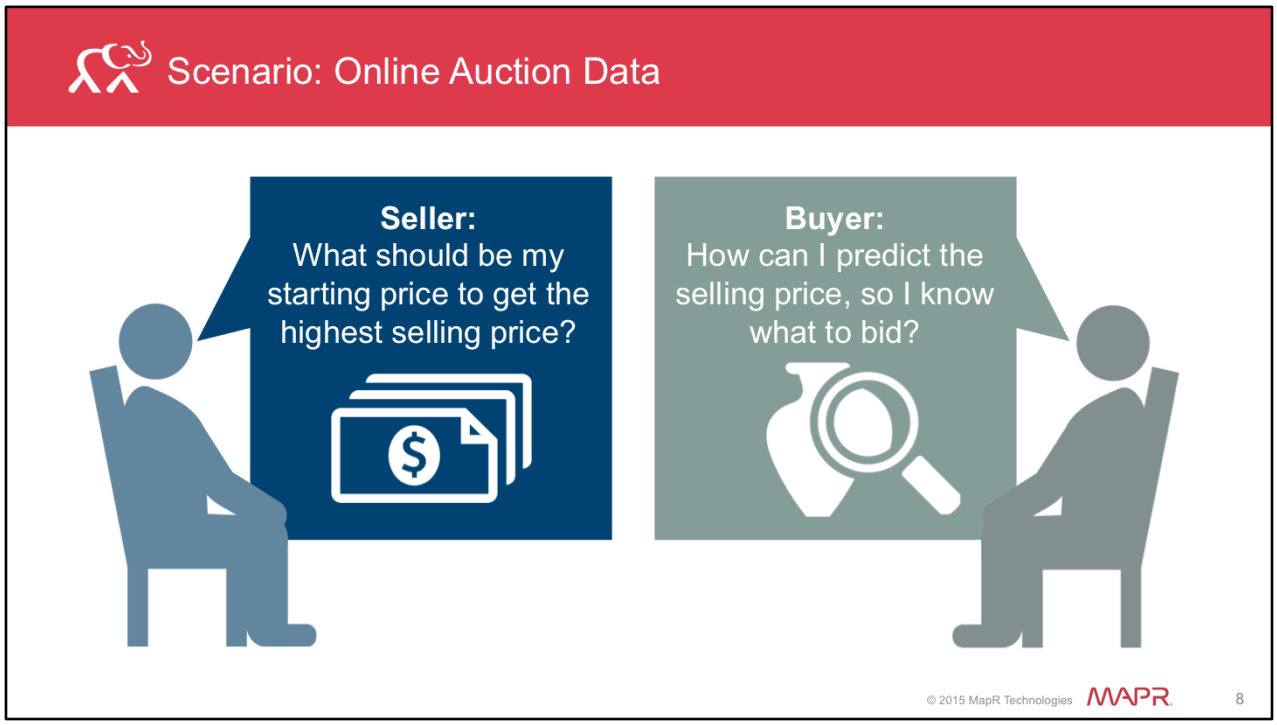
There are two types of data operations you can perform on an RDD--transformations and actions.
Transformations are lazily evaluated i.e. They're not computed immediately and will only execute when an action runs on it.
You can also persist, or cache, RDDs in memory or on disk.

Spark RDDs are fault-tolerant. If a given node or task fails, the RDD can be reconstructed automatically on the remaining nodes and the job will complete.



We are working with online Auction data that is in a CSV file in the local file system.
The data contains three types of items – xbox, cartier and palm.

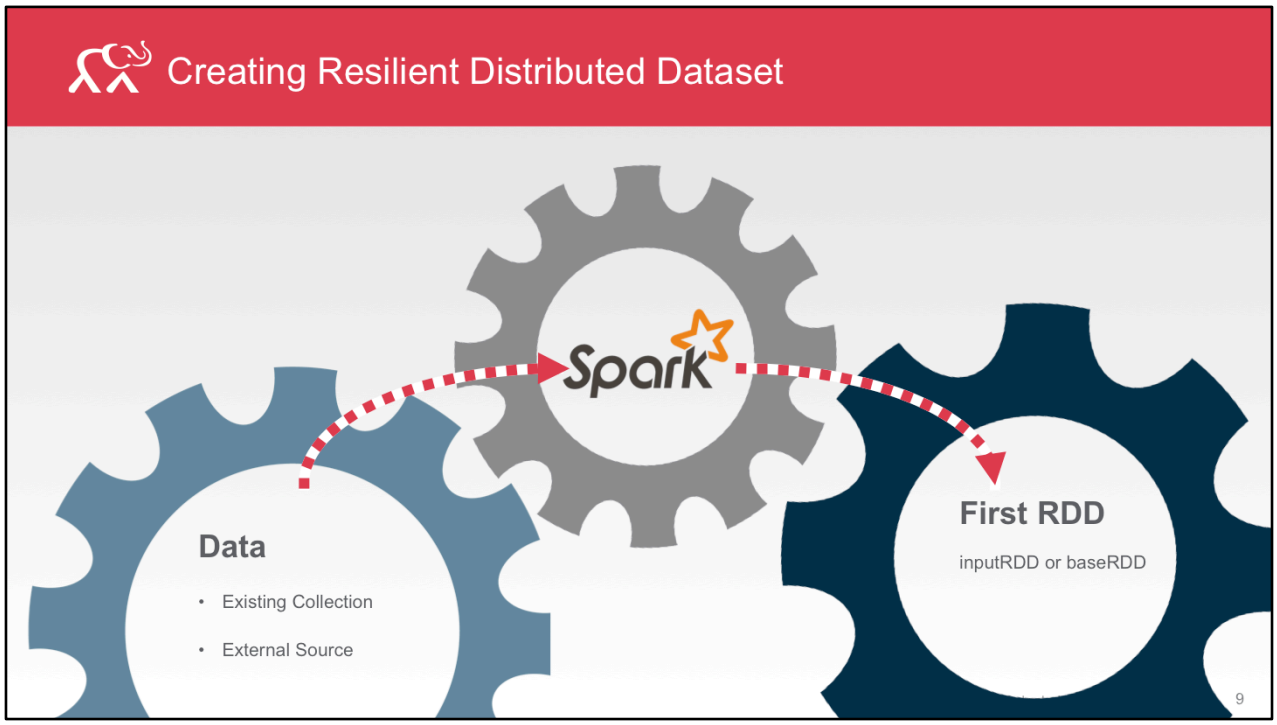
Each row in this file represents an individual bid.
Every bid has the auctionID, the bid amount, the bid time in days from when the auction started, the bidder userid, bidder rating, the opening bid for that auction, the final selling price, the item type, and the length of time for the auction in days.



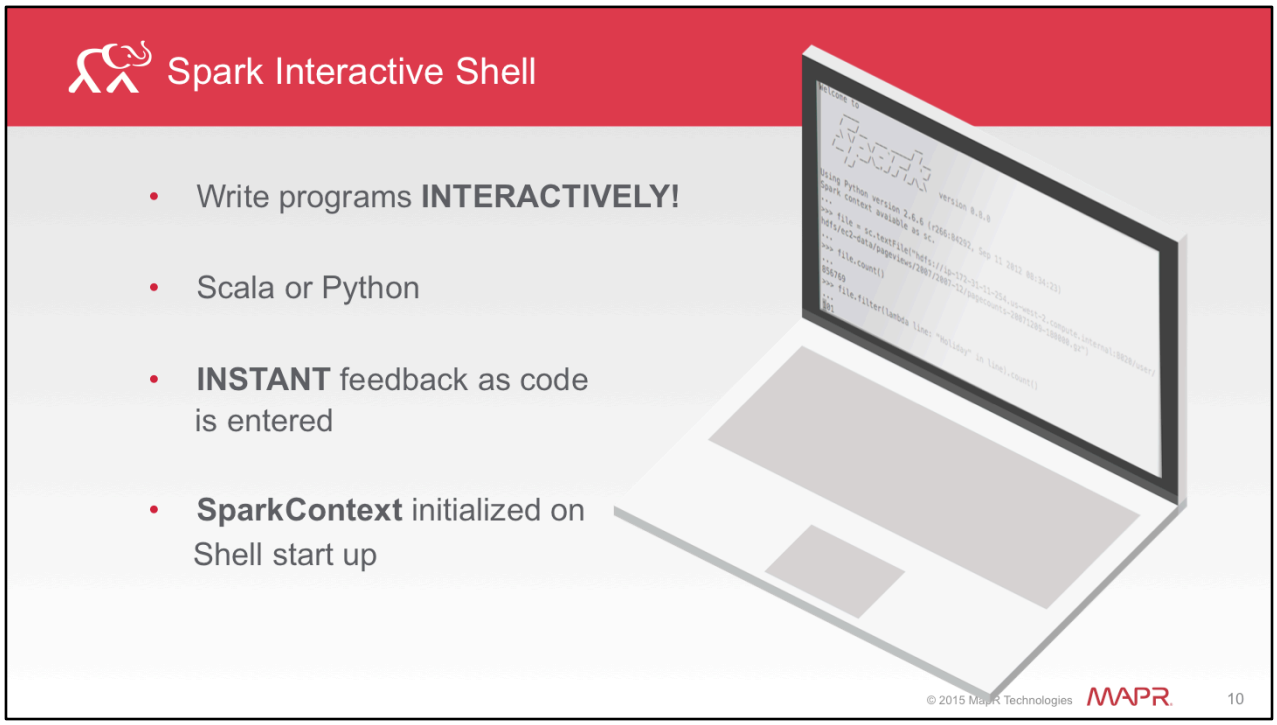
These are some common questions to ask of auction data.

- One use case is ad-targeting. Advertising companies auction ad space and companies vie to have their ads seen.
- Another use case is telephony bandwidth where spectrums are auctioned off in a closed bid.

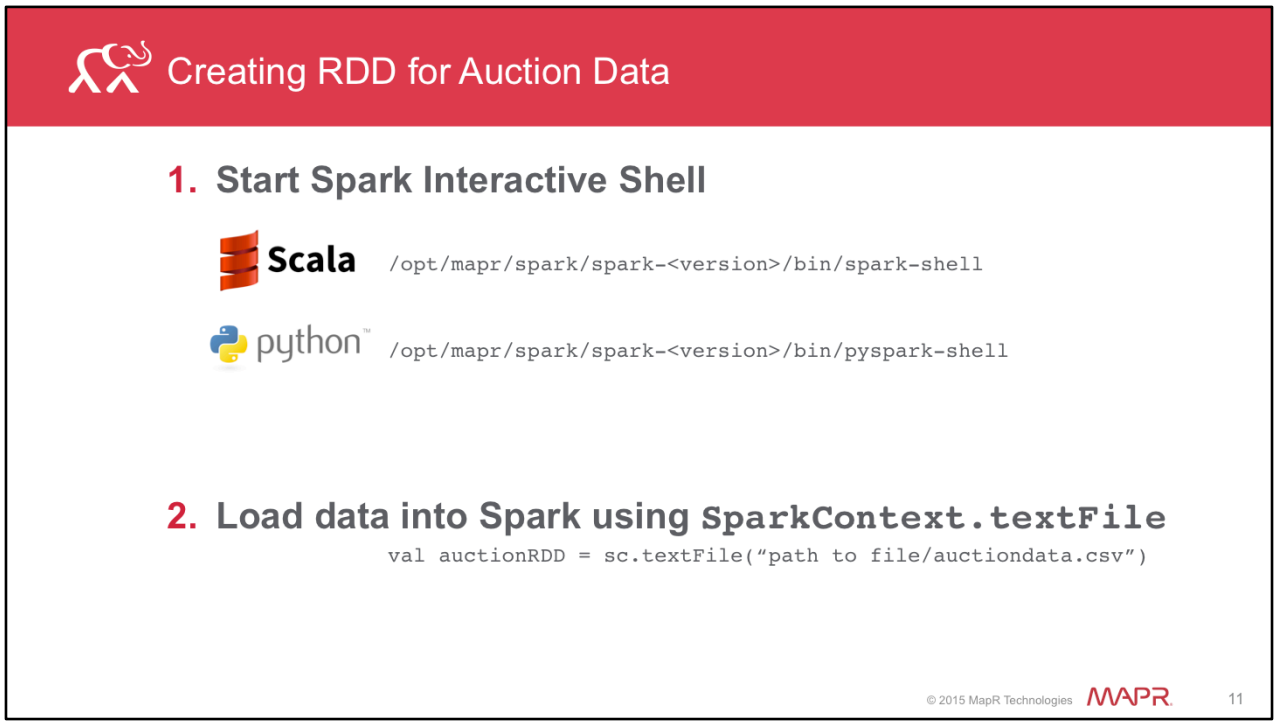
In this course, we are going to load and inspect the data. Later, we will use machine learning code to answer such questions.



You can create an RDD from an existing collection or from external data sources.
When you load the data into Spark, you are creating an RDD.
The first RDD you create is usually called the **input RDD** or **base RDD**.
We are going to use the Spark Interactive Shell to load the auction data into Spark.



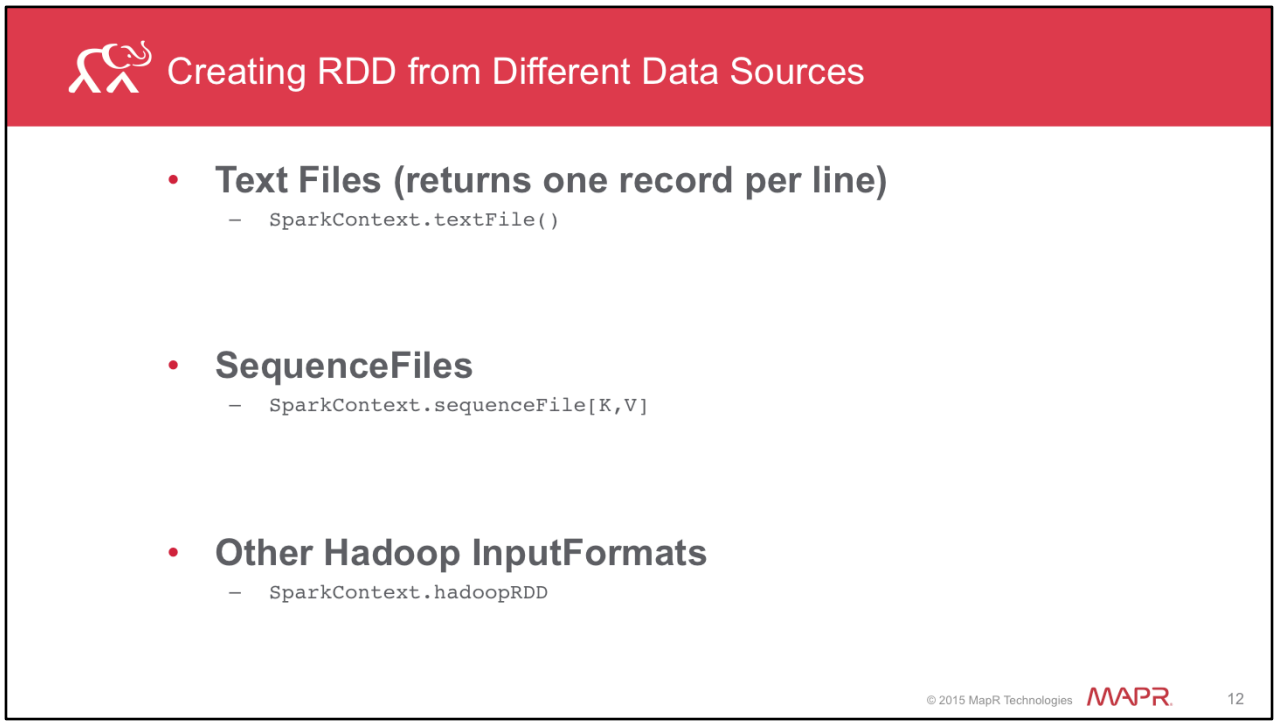
- The Spark interactive shell allows you to write programs interactively.
- It uses the Scala or Python REPL.
- The Interactive shell provides instant feedback as code is entered.
- When the shell starts, SparkContext is initialized and is then available to use as the variable sc. As a reminder, the first thing a program does is create a SparkContext. The SparkContext tells Spark where and how to access the Spark cluster.



1. First we start the interactive shell. This can be done by running spark-shell from bin in the Spark install directory. You can also use the Python interactive shell.

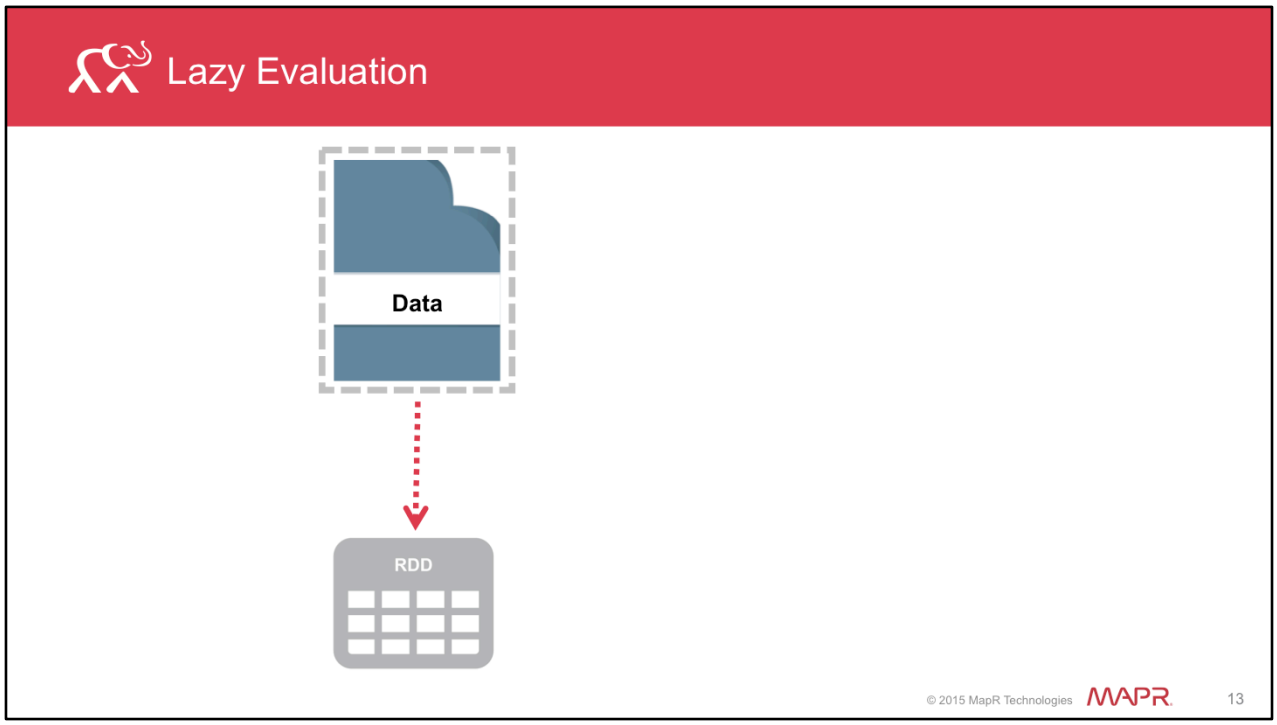
NOTE: During the course of the lesson, we will demonstrate the Scala code. However, in the lab activities, you have the option to use Python, and solutions are provided for both Python and Scala.

2. Once we have started our shell, we next load the data into Spark using the SparkContext.textFile method. Note that "sc" in the code refers to SparkContext.



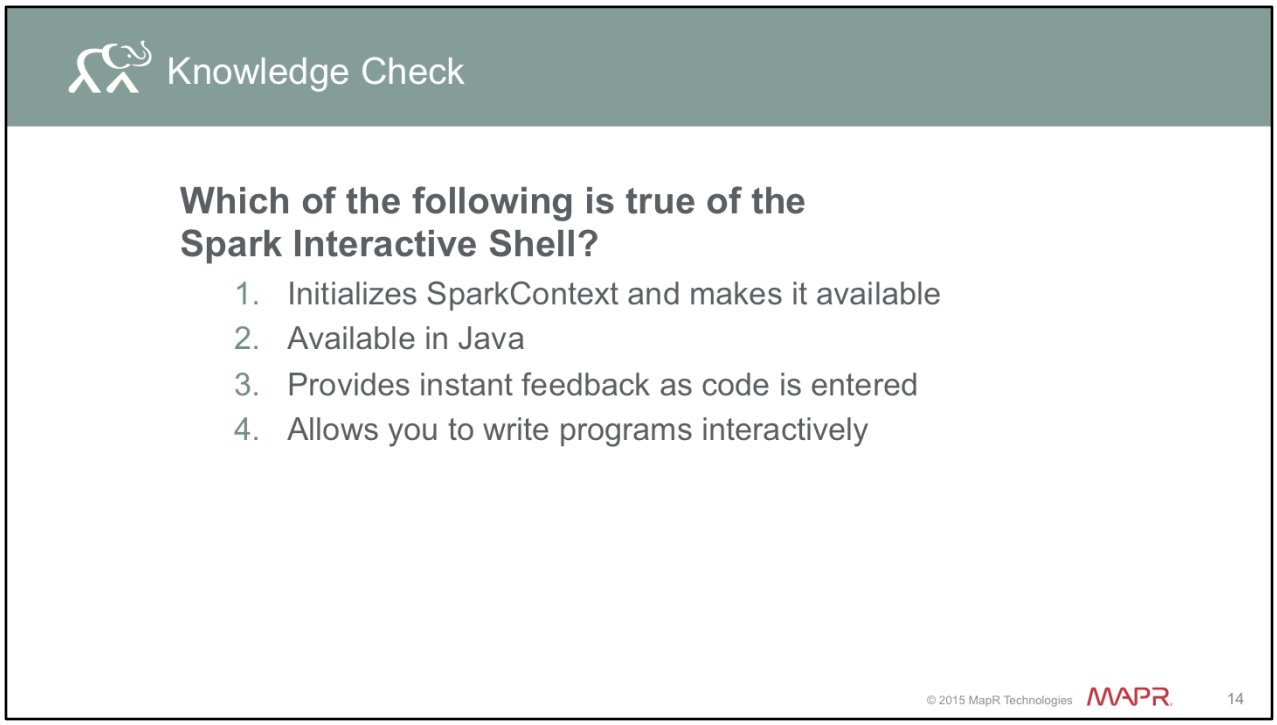
The textFile method returns an RDD that contains one record per line. Apart from text files, Spark's Scala API also supports several other data formats:

- SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file.
- For SequenceFiles, use SparkContext's sequenceFile[K, V] method where K and V are the types of key and values in the file.
- For other Hadoop InputFormats, you can use the SparkContext.hadoopRDD method, which takes an arbitrary JobConf and input format class, key class and value class. Set these the same way you would for a Hadoop job with your input source.

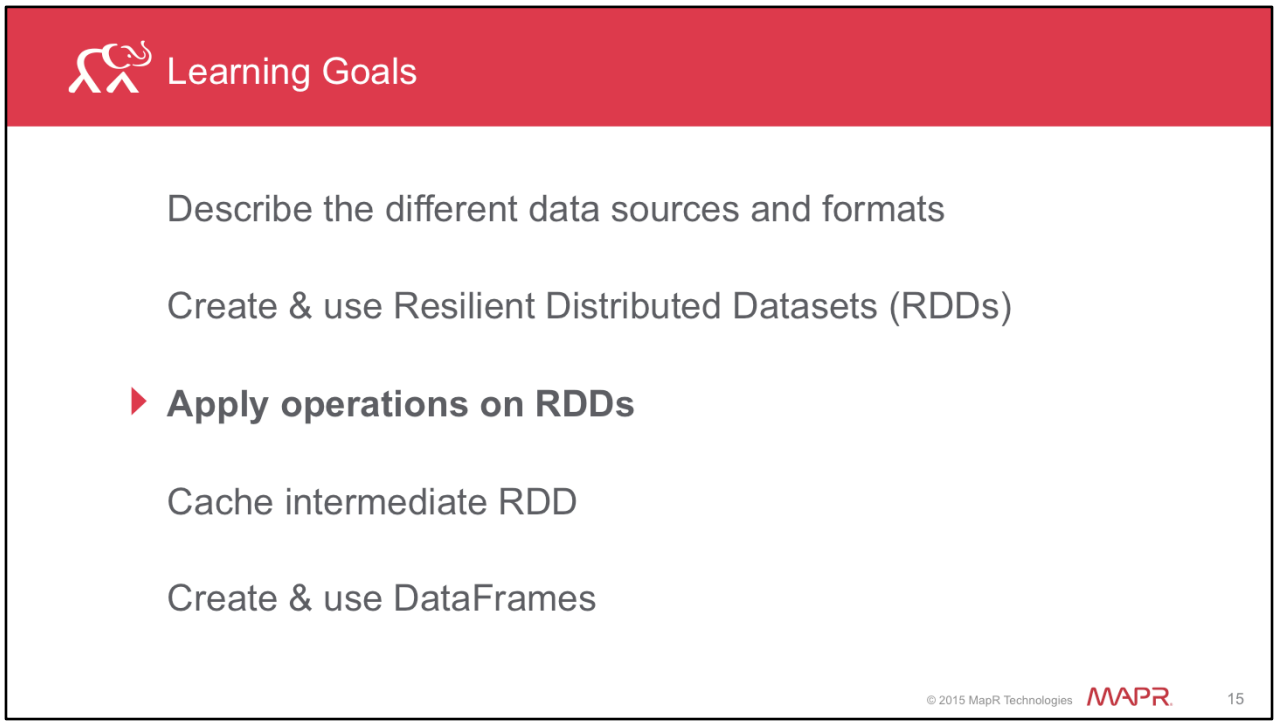


RDDs are lazily evaluated. We have defined the instructions to create our RDD and defined what data we will use. The RDD has not actually been created yet.

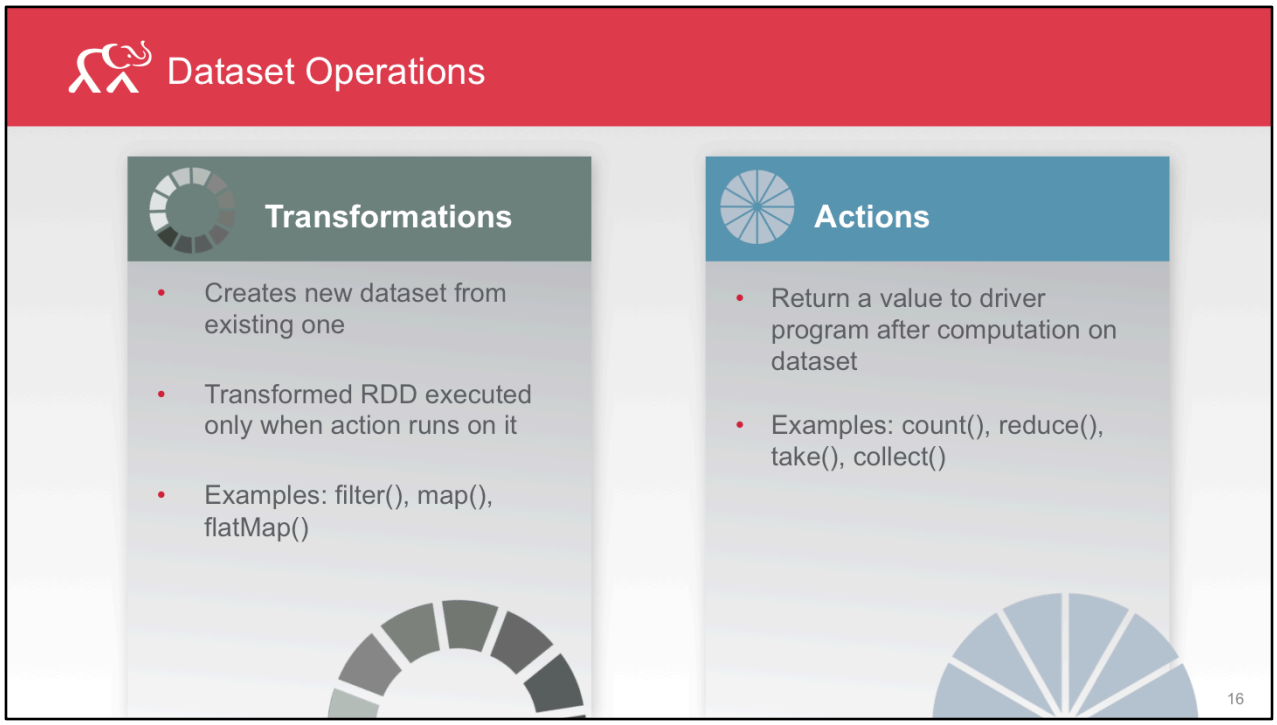
The RDD will be created, the data loaded into it and the results returned only when Spark encounters an action.


Knowledge Check
Which of the following is true of the Spark Interactive Shell?
1. Initializes SparkContext and makes it available
2. Available in Java
3. Provides instant feedback as code is entered
4. Allows you to write programs interactively
© 2015 MapR Technologies
14

Answers: 1,3,4



Now that we have created an RDD, we can apply some operations on it. In this section, we will discuss the two types of operations: transformations and actions that can be applied to RDDs.



There are two types of operations you can perform on an RDD:

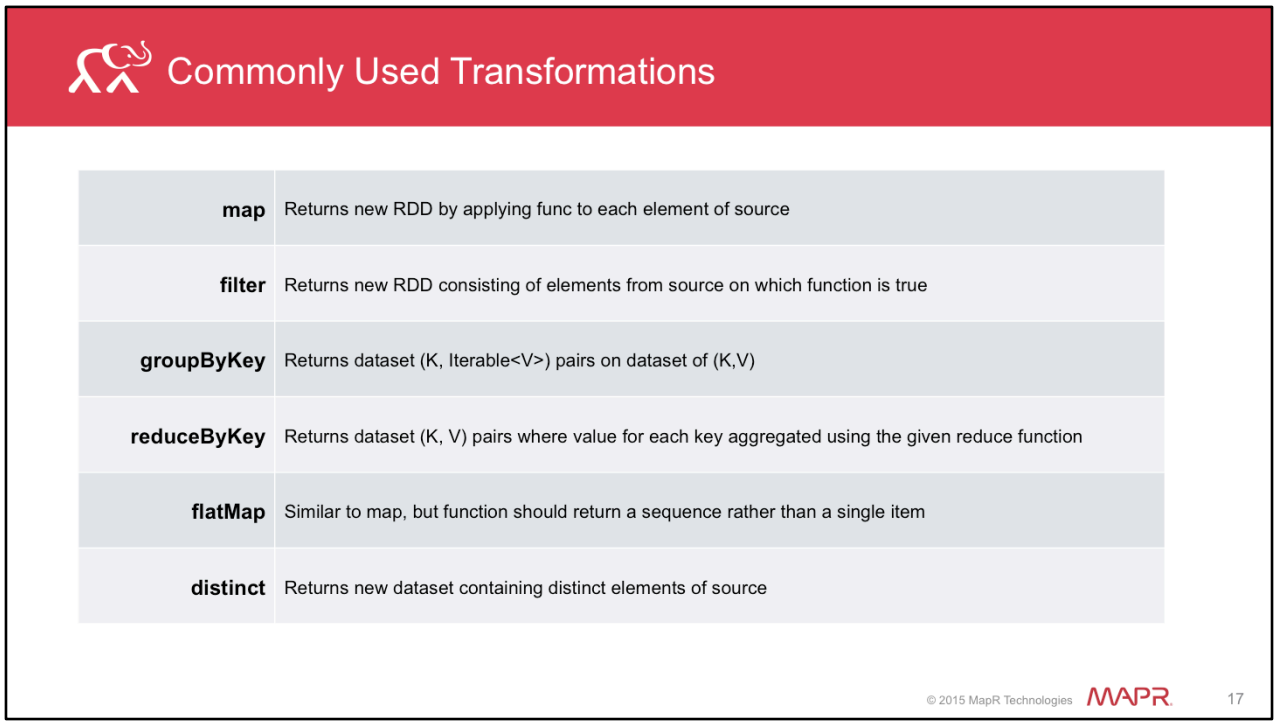
- *transformations*, which create a new dataset from an existing one
- and *actions*, which return a value to the driver program after running a computation on the dataset

Transformations are lazily evaluated, which means they are not computed immediately. A transformed RDD is executed only when an action runs on it.

Some examples of Transformations are filter and map

Examples of Actions include count() and reduce().

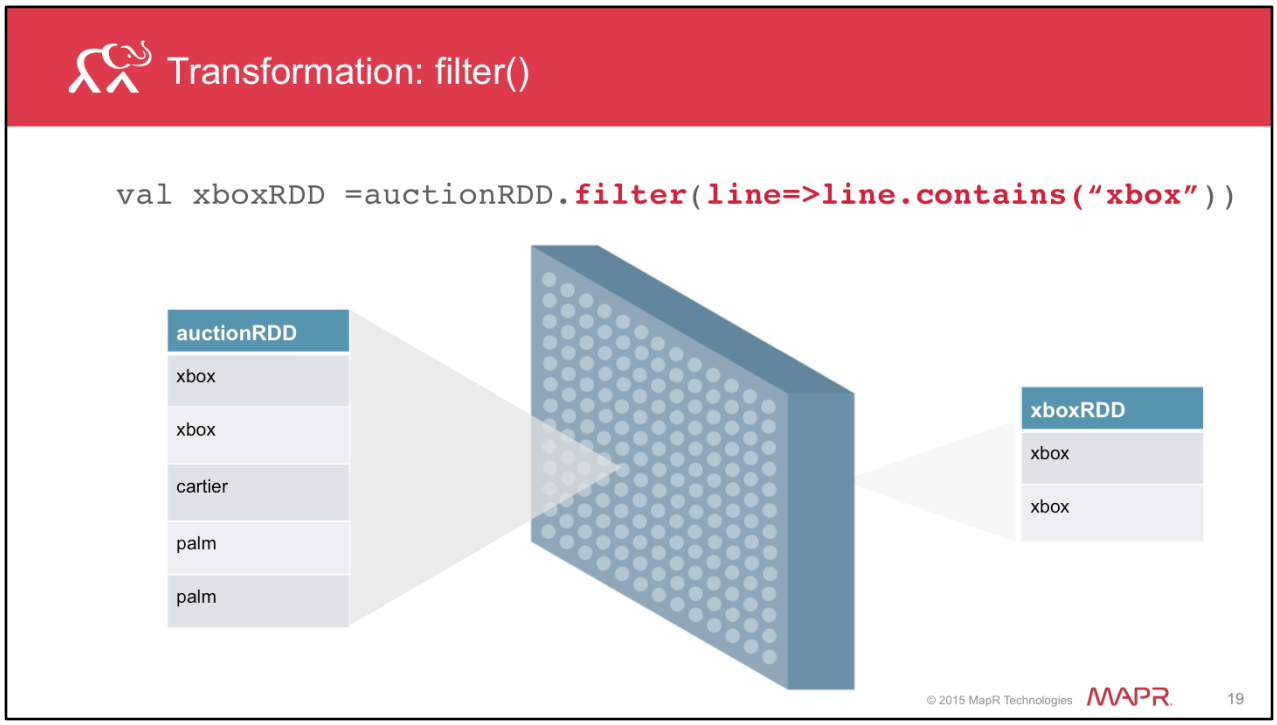
Let us talk about transformations first.



Here is a list of some of the commonly used transformations. You can visit the link provided in the resources included with this course for more transformations.

A sample of data in auctiondata.csv is shown here. Each line represents a bid on a particular item such as xbox, cartier, palm, etc. We only want to look at the bids on xbox.

Q. What transformation could we use to get bids only on Xboxes?



To do this we can use the filter transformation on the auctionRDD. Each element in the auctionRDD is a line. So we apply the filter to each line of the RDD.

The filter transformation filters out based on the specified condition. We are applying the filter transformation to the auctionRDD where the condition checks to see if the line contains the word "xbox". If the condition is true, then that line is added to the resulting RDD, in this case, xboxRDD. The filter transformation is using an **anonymous function**.

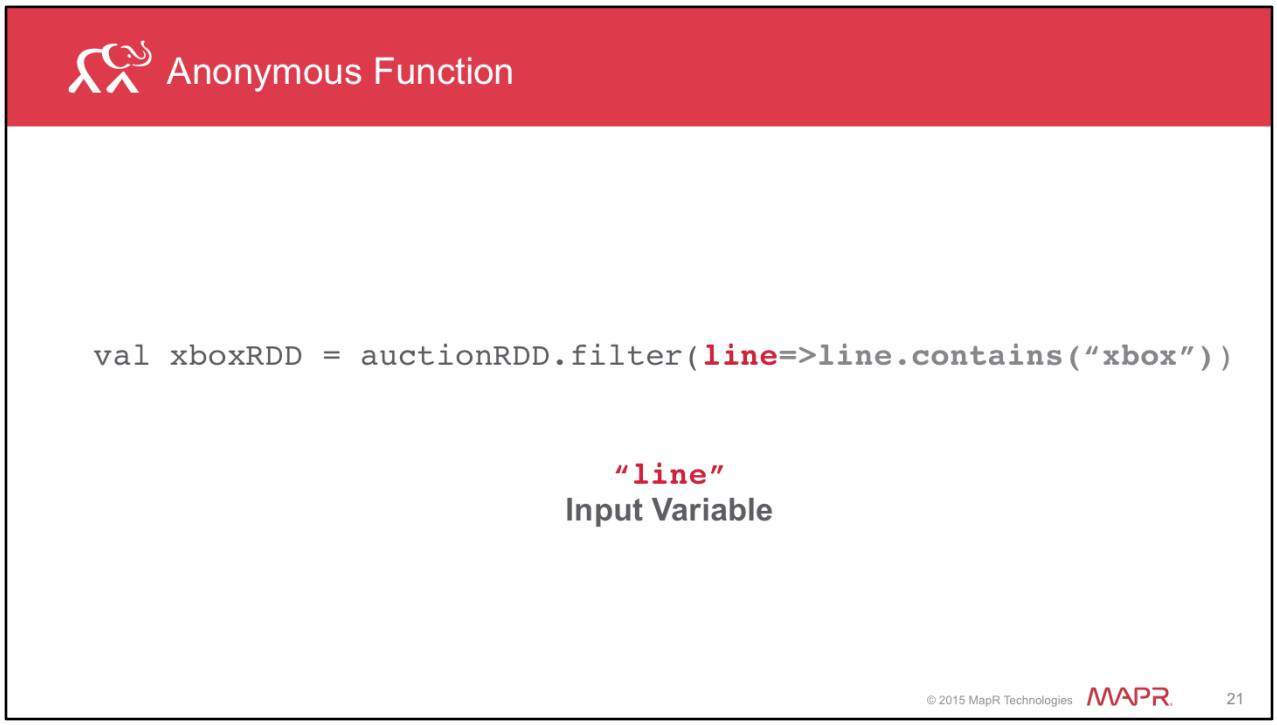
Anonymous Function

```
val xboxRDD = auctionRDD.filter(line=>line.contains("xbox"))
```

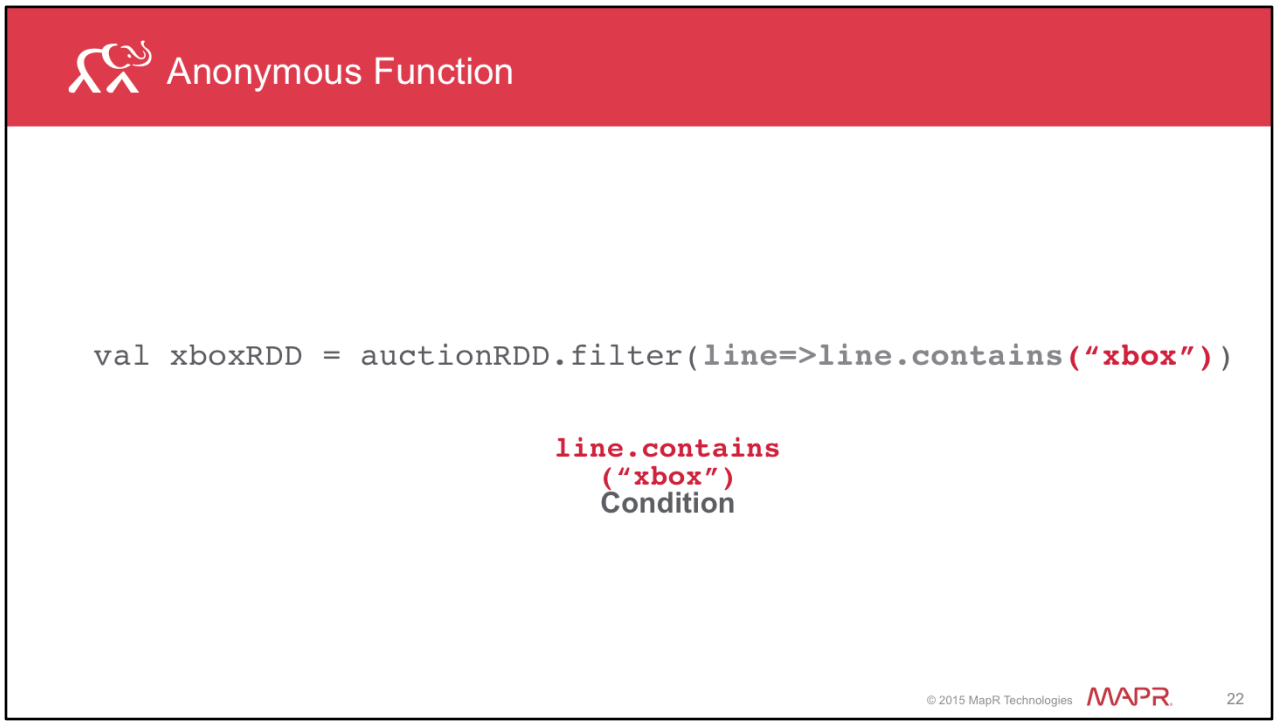
"=>"

Anonymous Syntax

The filter transformation is applying an anonymous function to each element of the RDD which is a line.
This is the Scala syntax for an anonymous function, which is a function that is not a named method.



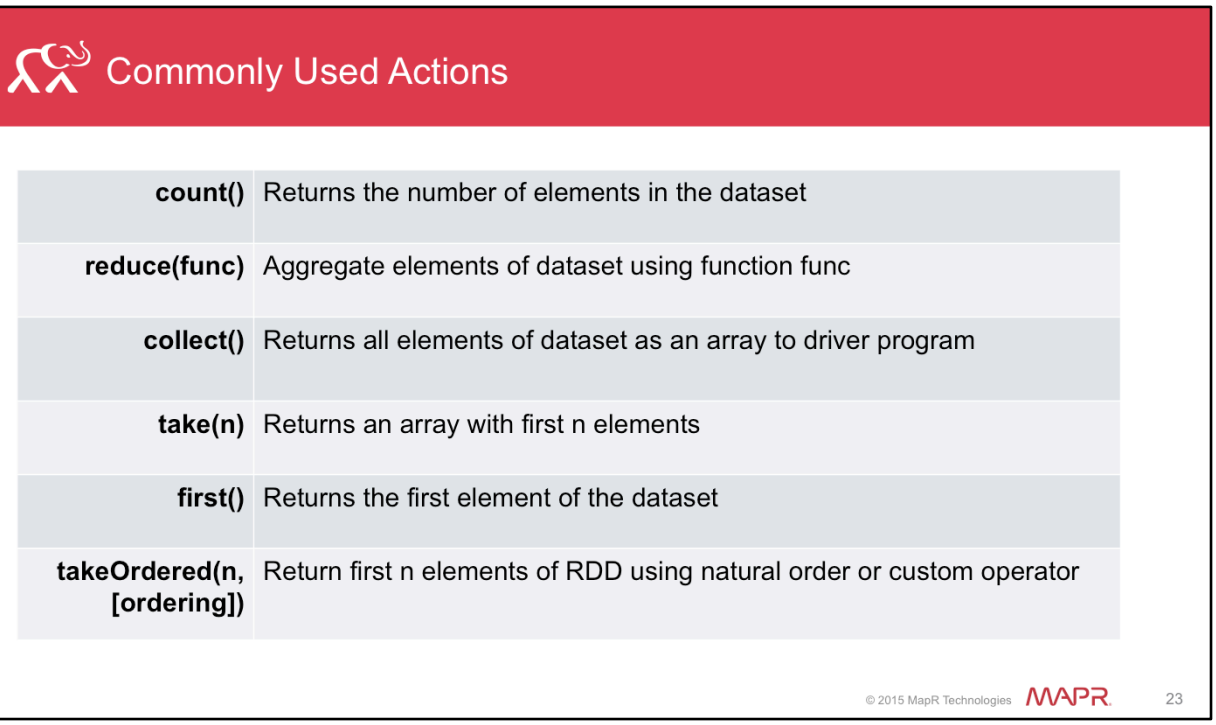
The `line => line.contains` syntax means that we are applying a function where the input variable is to the left of the => operator. In this case, the input is line,



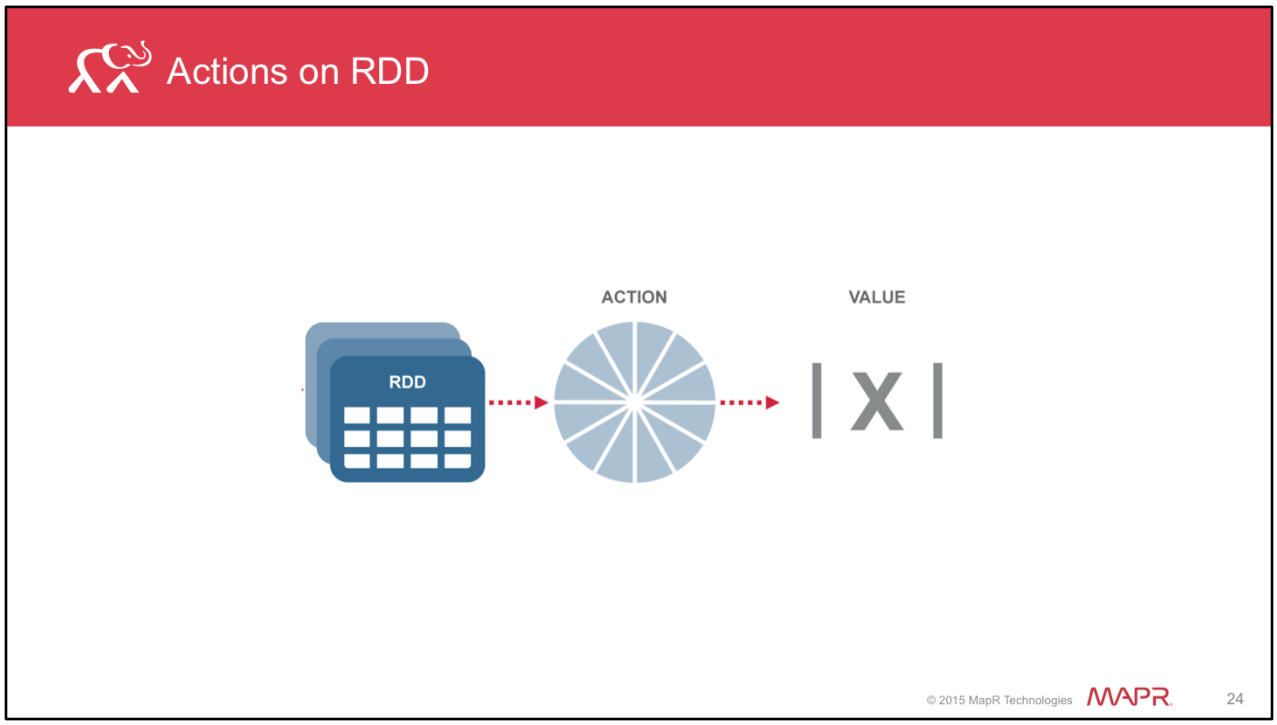
The filter will return the result of the code to the right of the function operator. In this example the output is the result of calling line.contains with the condition, does it contain “xbox”.

Anonymous functions can be used for short pieces of code.

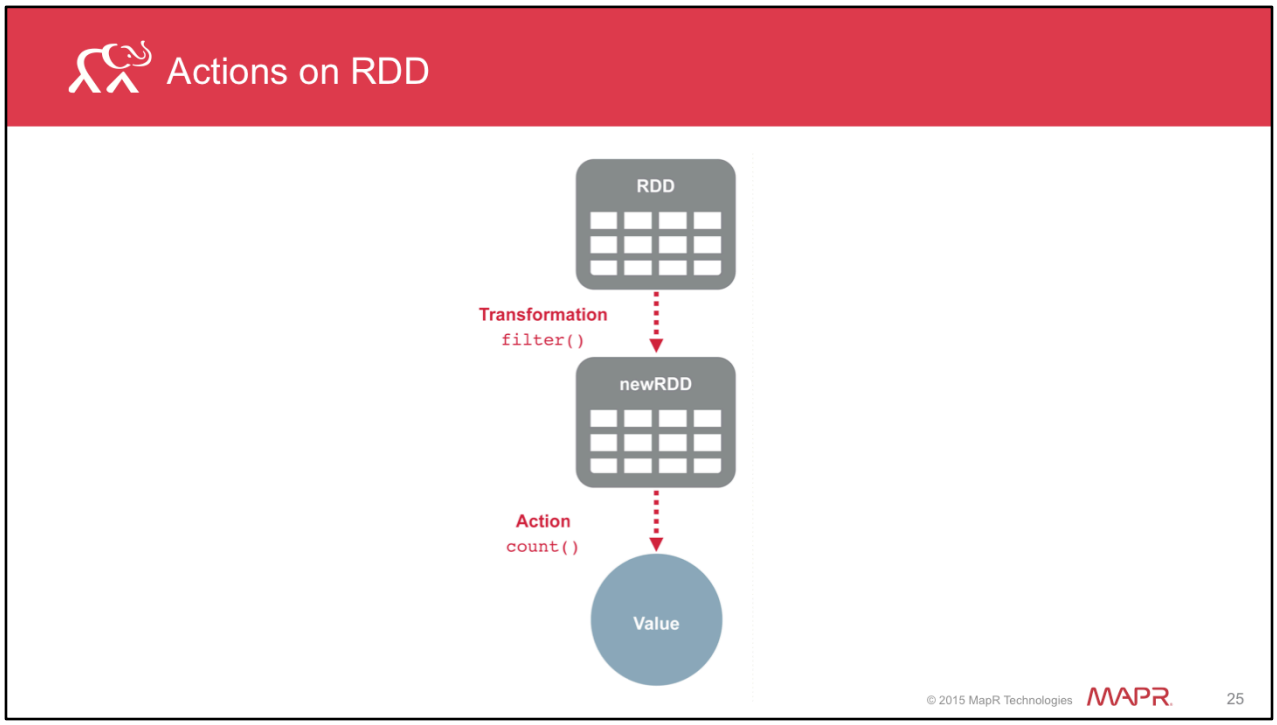
We will now take a look at applying actions.



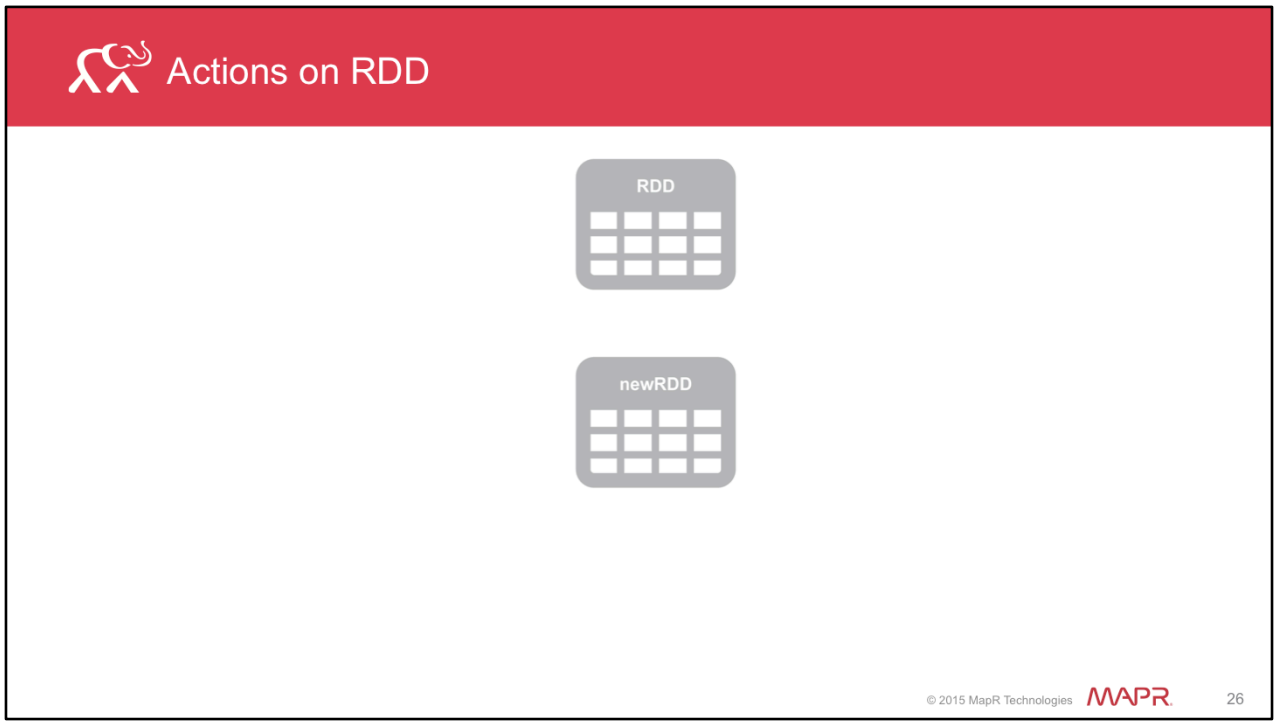
Here is a list of some of the commonly used actions. You can visit the link provided in the resources included with this course for more actions.



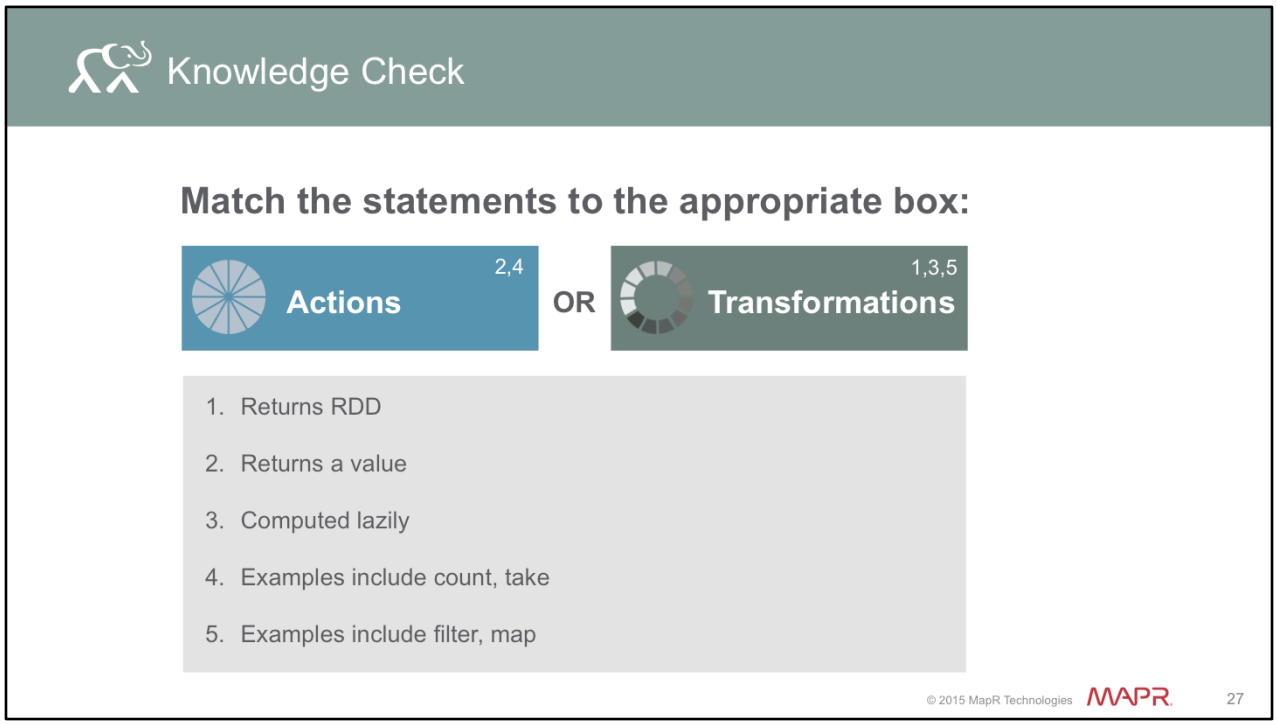
An action on an RDD returns values to the driver program, after running the computation on the dataset.
As mentioned earlier, transformations are lazy. They are only computed when an action requires a result to be returned to the Driver program.



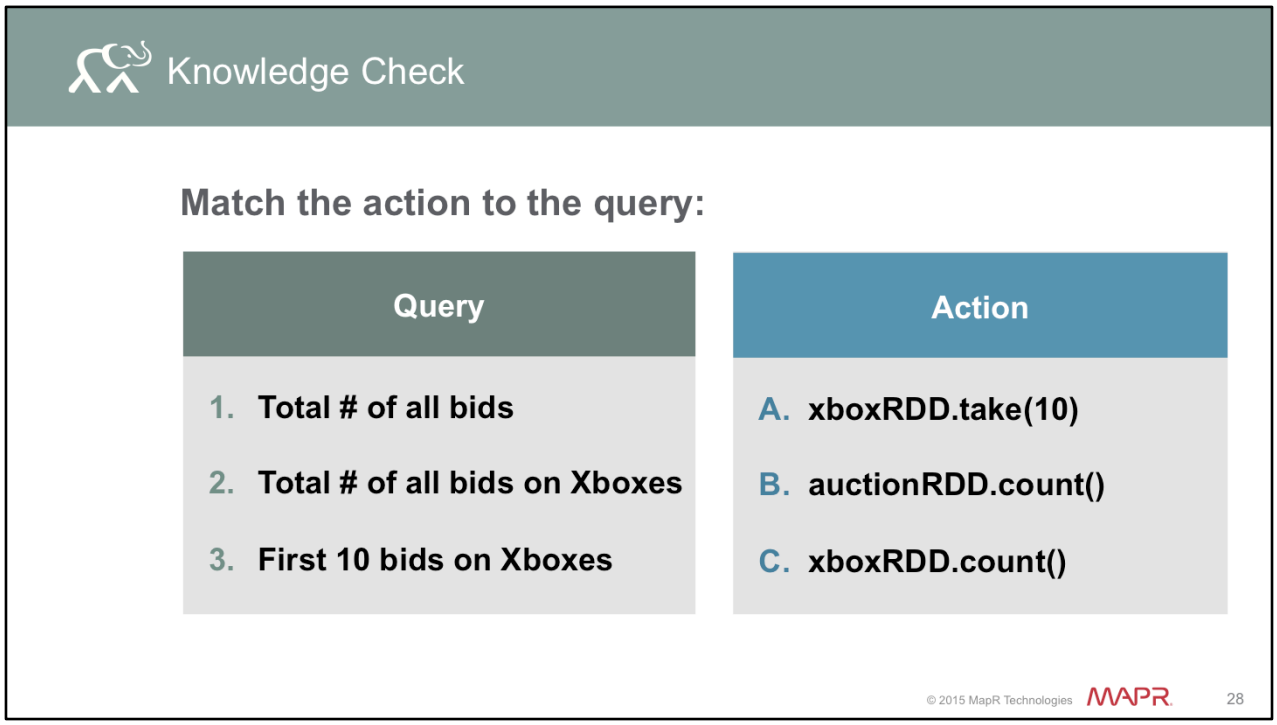
Here the baseRDD and new RDD are not materialized till we add an action – count().
When we run the command for the action, Spark will read from the text file and create the base RDD. Then the filter transformation is applied to the baseRDD to produce the newRDD. Then count runs on the newRDD sending the value of the total number of elements in the RDD to the driver program.



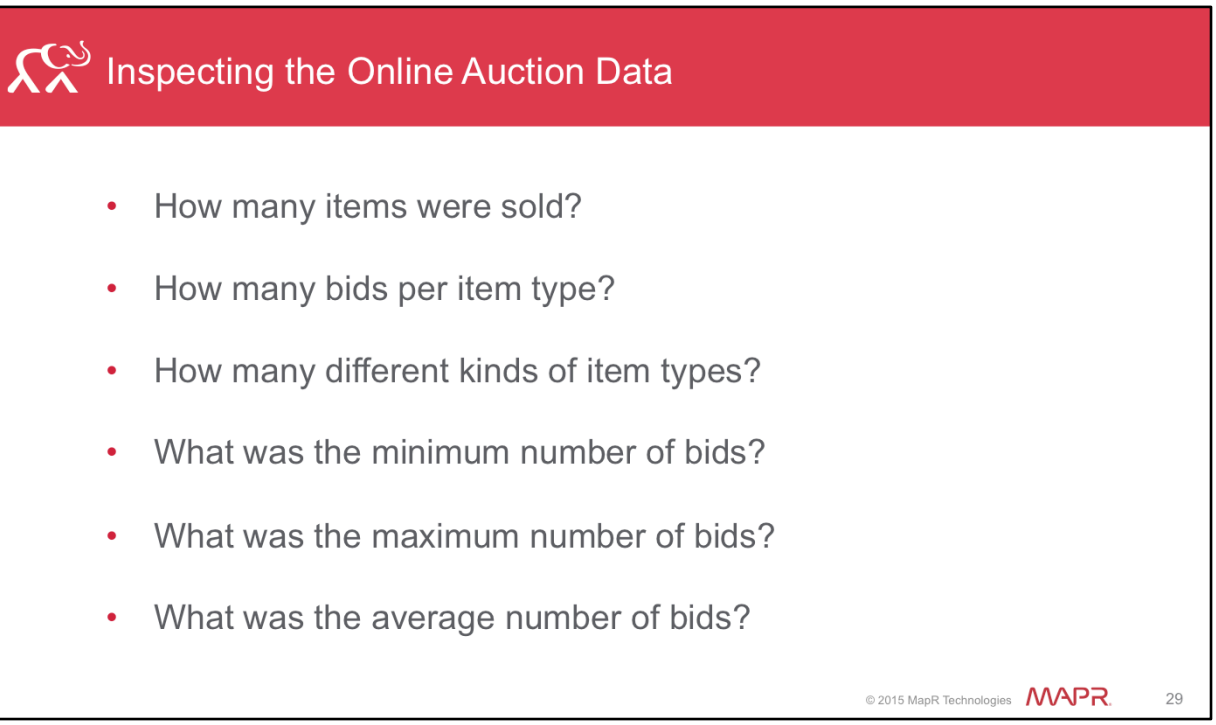
Once the action has run, the data is no longer in memory.



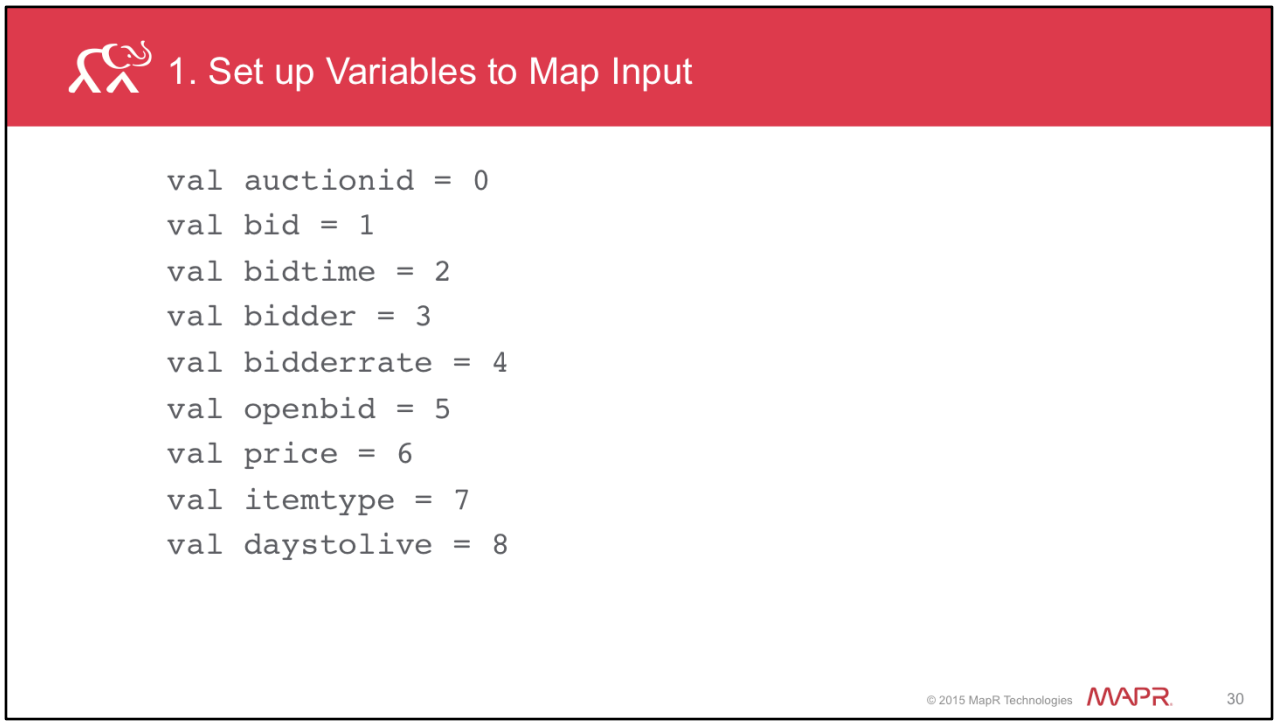
Actions: 2,4
Transformations: 1,3,5


Knowledge Check
Match the action to the query:
Query
1. Total # of all bids
2. Total # of all bids on Xboxes
3. First 10 bids on Xboxes
Action
A. xboxRDD.take(10)
B. auctionRDD.count()
C. xboxRDD.count()
© 2015 MapR Technologies
28

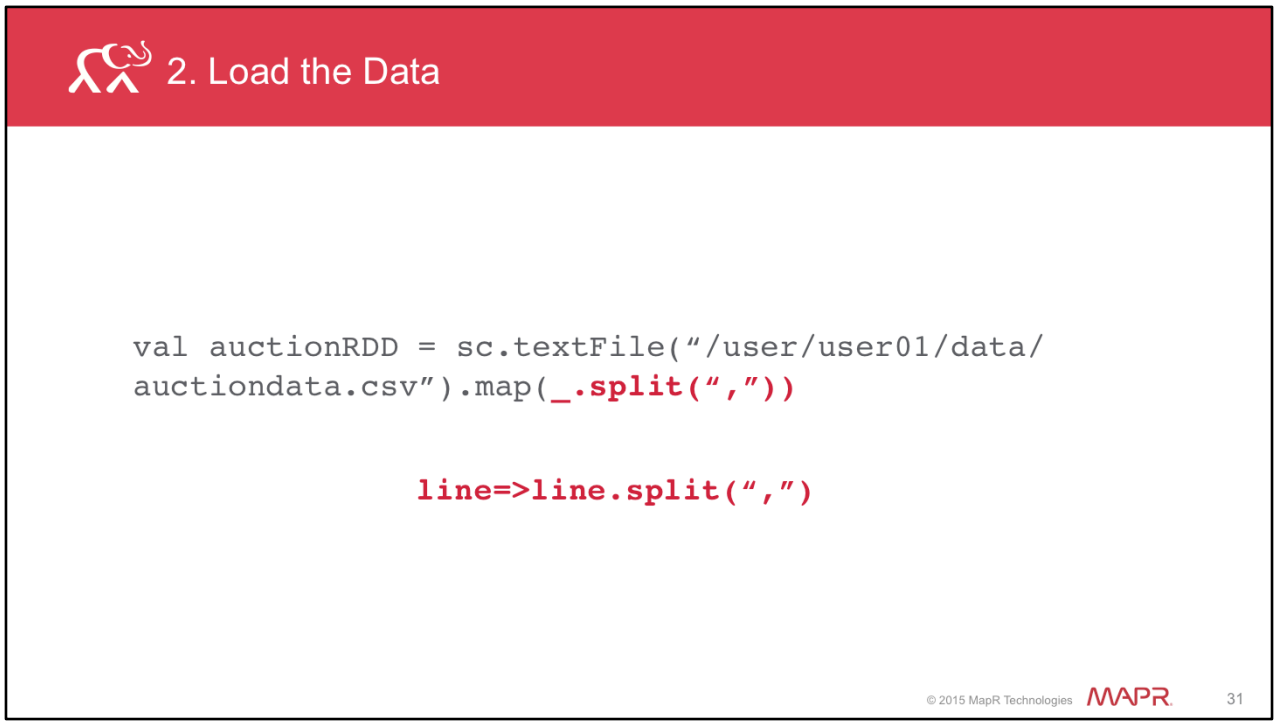
1 → b
2 → c
3 → a



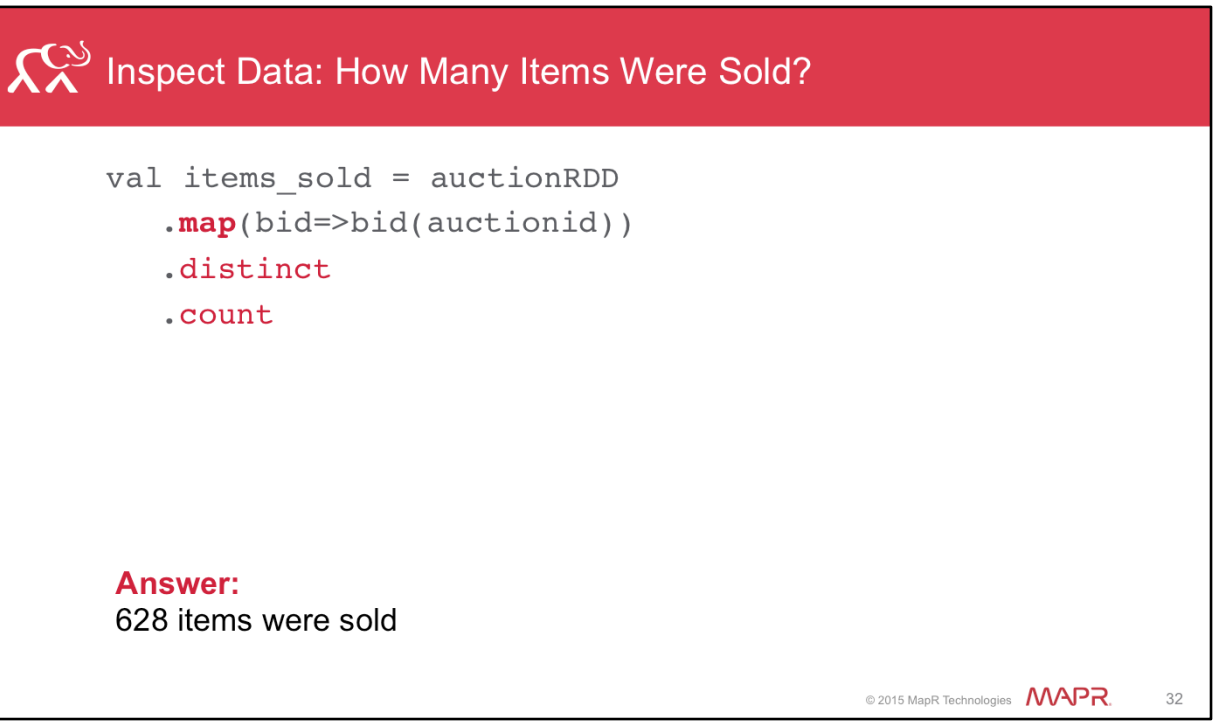
Now that we have looked at the definitions transformations and actions, let us apply these to the base RDD, to inspect the data. We want to find answers to the questions listed here. We will discuss some of these questions in this section, and you will answer the remaining questions in a Lab activity.



We are setting up the variables to map the input based on our auction data. This is one way to load the data into an RDD in a 2-D array, which makes it easier to reference individual “columns”.



As we have seen before, we are using the SparkContext method `textFile` to load the data in .csv format from the local file system. We use the map transformation that applies the split function to each line to split it on the ",".

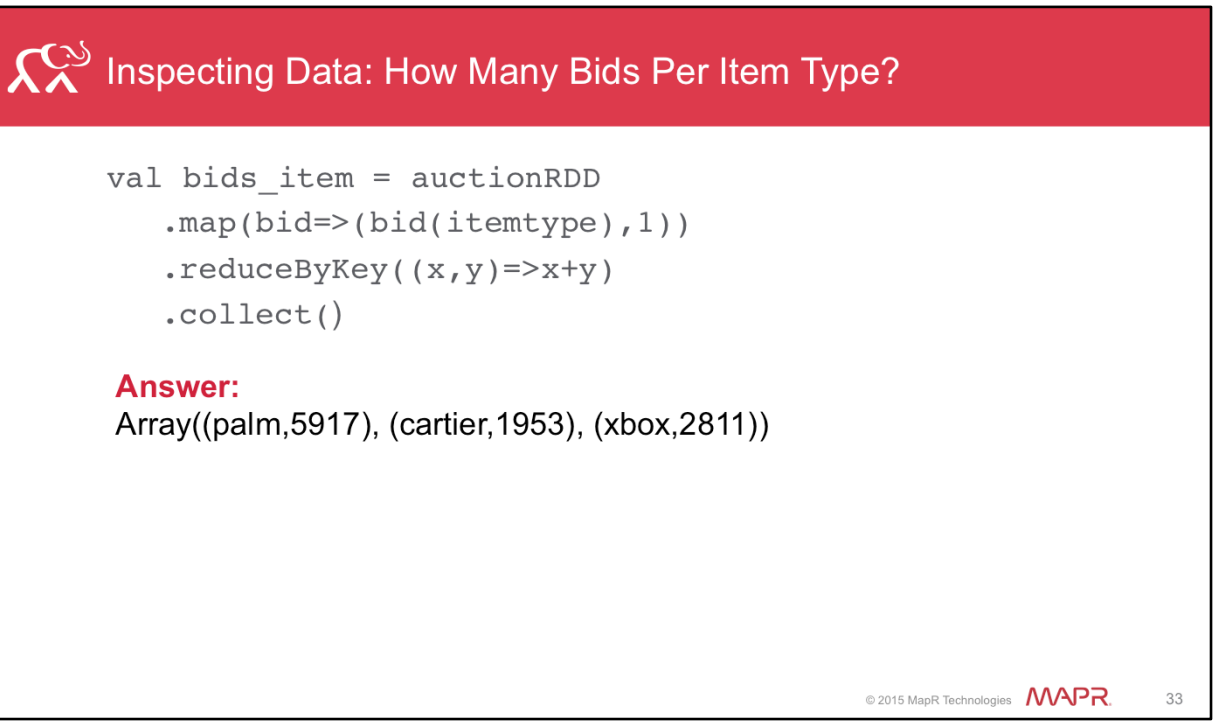


The first question that we want to ask of our data, is how many items were sold.

Each auctionID represents an item for sale. Every bid on that item will be one complete line in the baseRDD dataset, therefore we have many lines in our data for the same item.

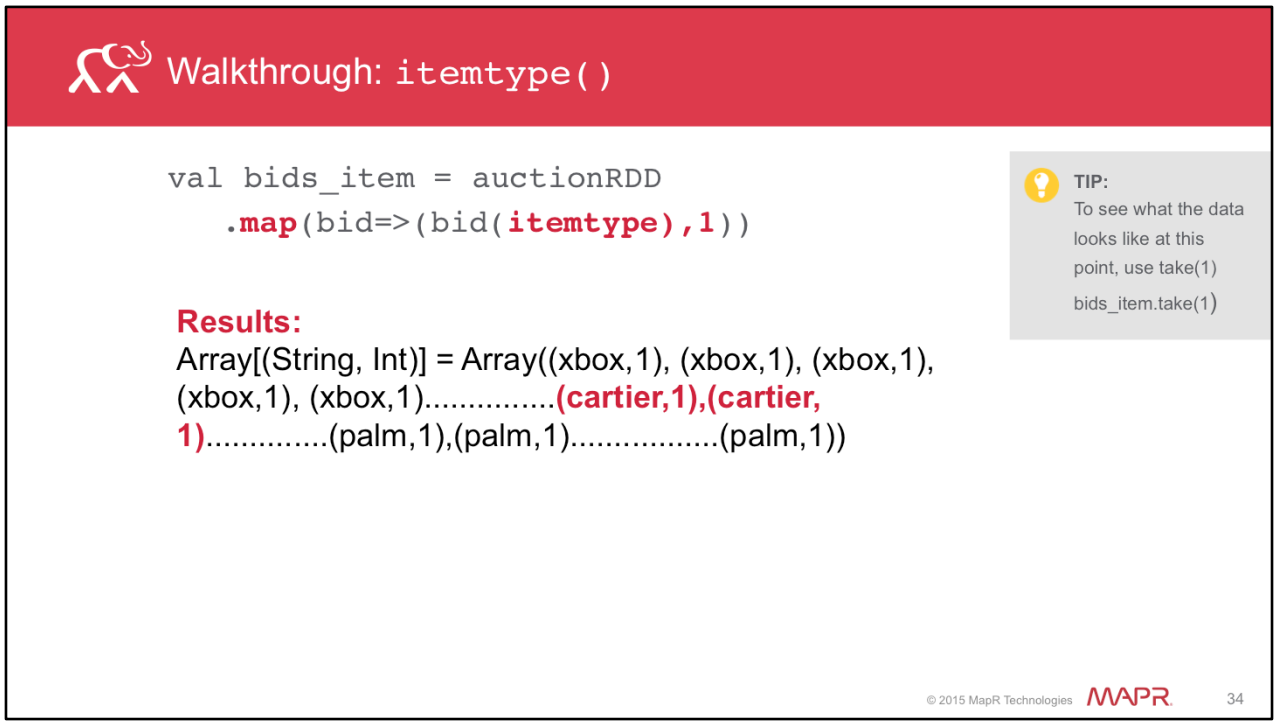
The map transformation shown here will return a new dataset that contains the auctionID from each line. The distinct transformation is run on the "auctionID" dataset, and will return another new dataset, which contains the distinct auctionIDs. The count action is run on the "distinct" dataset, and will return the number of distinct auctionIDs.

Each of the datasets that are created by a transformation is temporary, and stored in memory until the final action is complete.

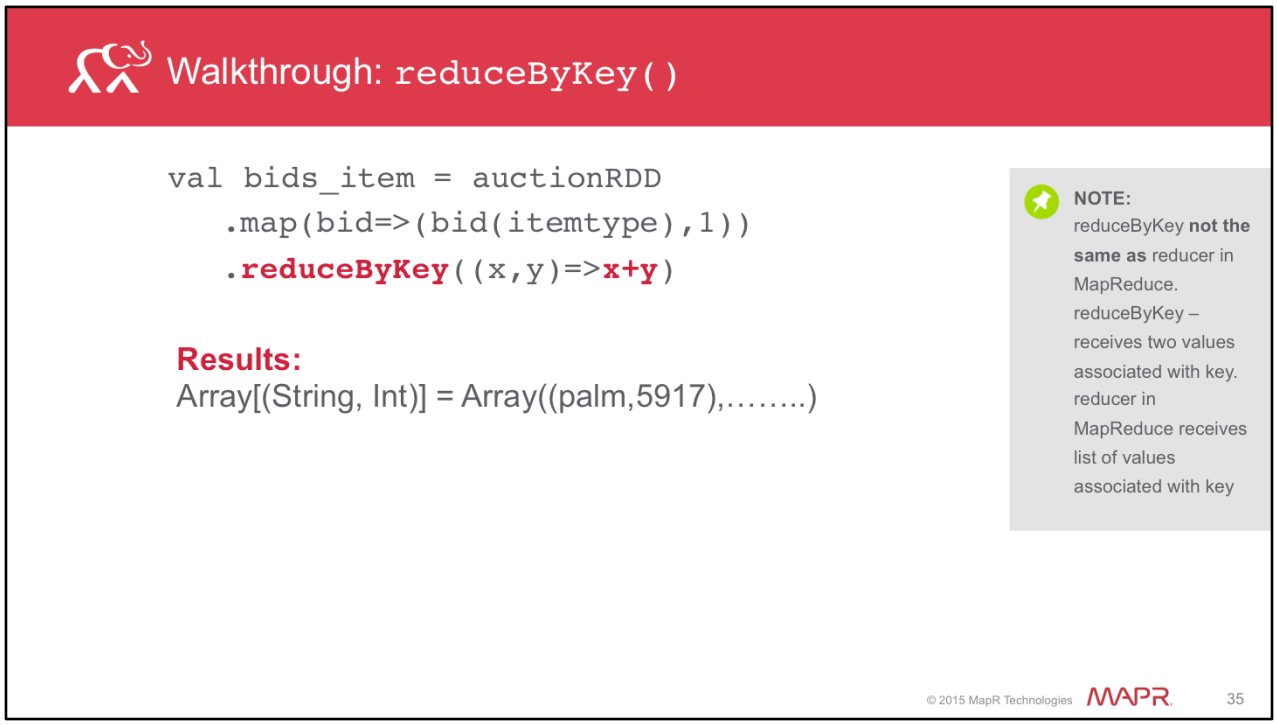


Question 2 – How many bids are there per item type? The item type in our data could be xbox, cartier or palm. Each item type, may have a number of different items.
This line of code is explained in more detail next.

A: Array((palm,5917), (cartier,1953), (xbox,2811))



The function in the first map transformation is mapping each record (bid) of the dataset to an ordered pair (2-tuple) consisting of (itemtype, 1). If we want to see what the data looks like at this point, apply take(1). i.e. bids_item.take(1).



In this example, reduceByKey runs on the key-value pairs from the map transformation – (itemtype,1) and aggregates on the key based on the function. The function that we have defined here for the reduceByKey transformation is sum of the values i.e reduceByKey returns key-value pairs which consists of the sum of the values by key.
Note that reduceByKey here does not work the same way as the reducer in MapReduce. reduceByKey receives two values associated with a key. The reducer in MapReduce receives a list of values associated with the key.

Walkthrough: `collect()`

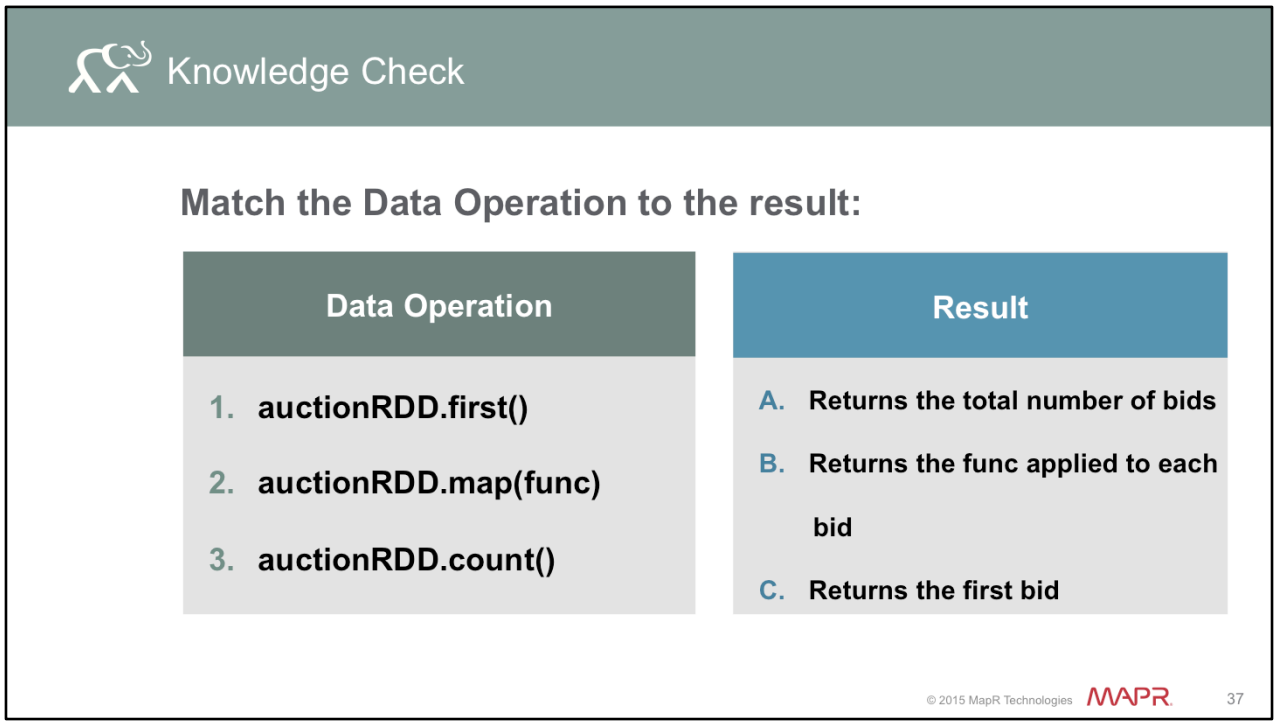
```
val bids_item = auctionRDD
                          .map(x=>(x(item),1))
                          .reduceByKey((x,y)=>x+y)
                          .collect()
```

Results:

Array[(String, Int)] = Array((palm,5917), (cartier,1953), (xbox,2811))

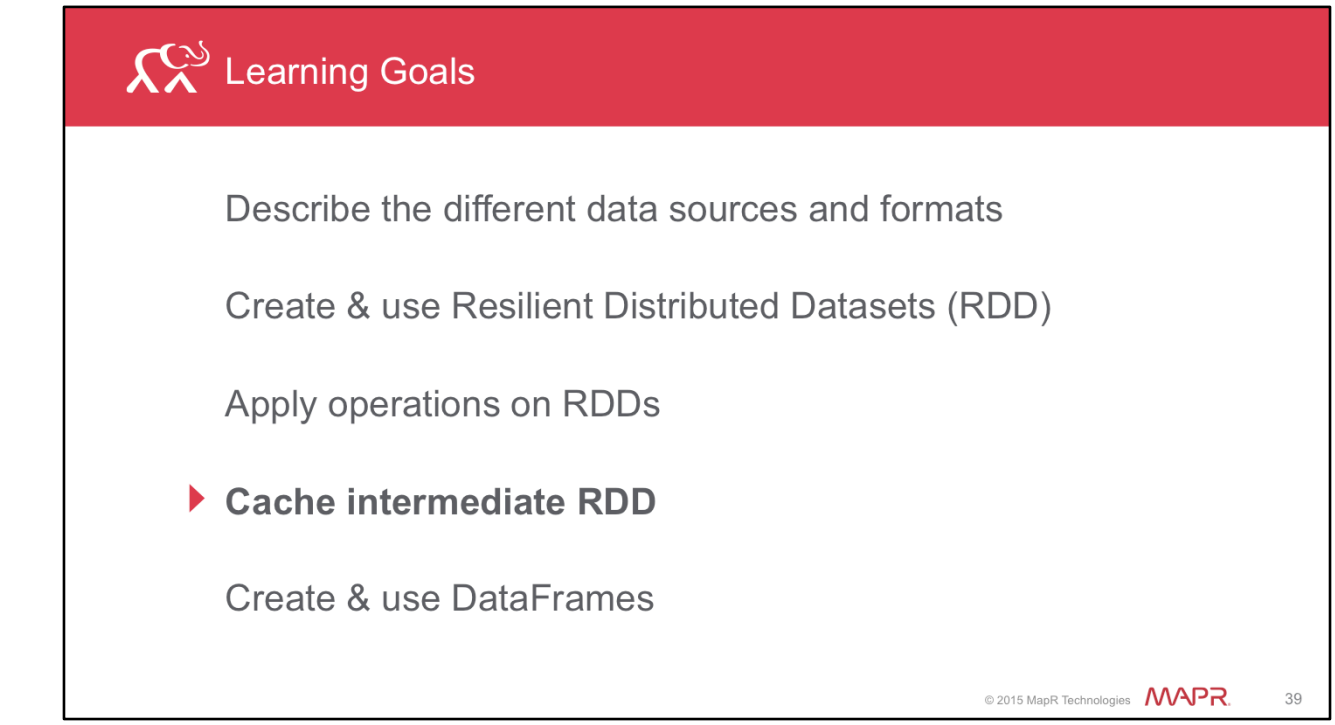
The collect() action, collects the results and sends it to the driver. We see the result as shown here – i.e. the number of bids per item type.

A: Array((palm,5917), (cartier,1953), (xbox,2811))

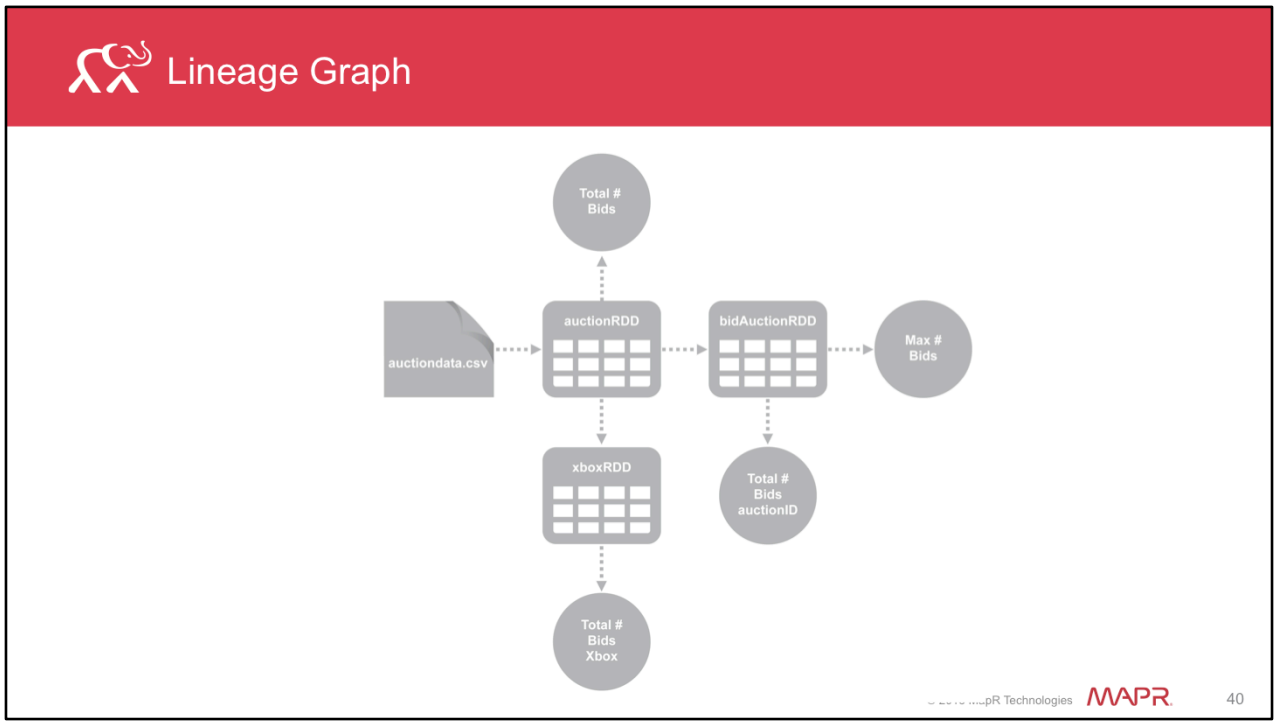

Knowledge Check
Match the Data Operation to the result:
Data Operation
1. auctionRDD.first()
2. auctionRDD.map(func)
3. auctionRDD.count()
Result
A. Returns the total number of bids
B. Returns the func applied to each bid
C. Returns the first bid
© 2015 MapR Technologies
37

1 → c
2 → b
3 → a

In this lab, you use the interactive shell to load the online auction data. You use transformations and actions to answer the questions about the data.

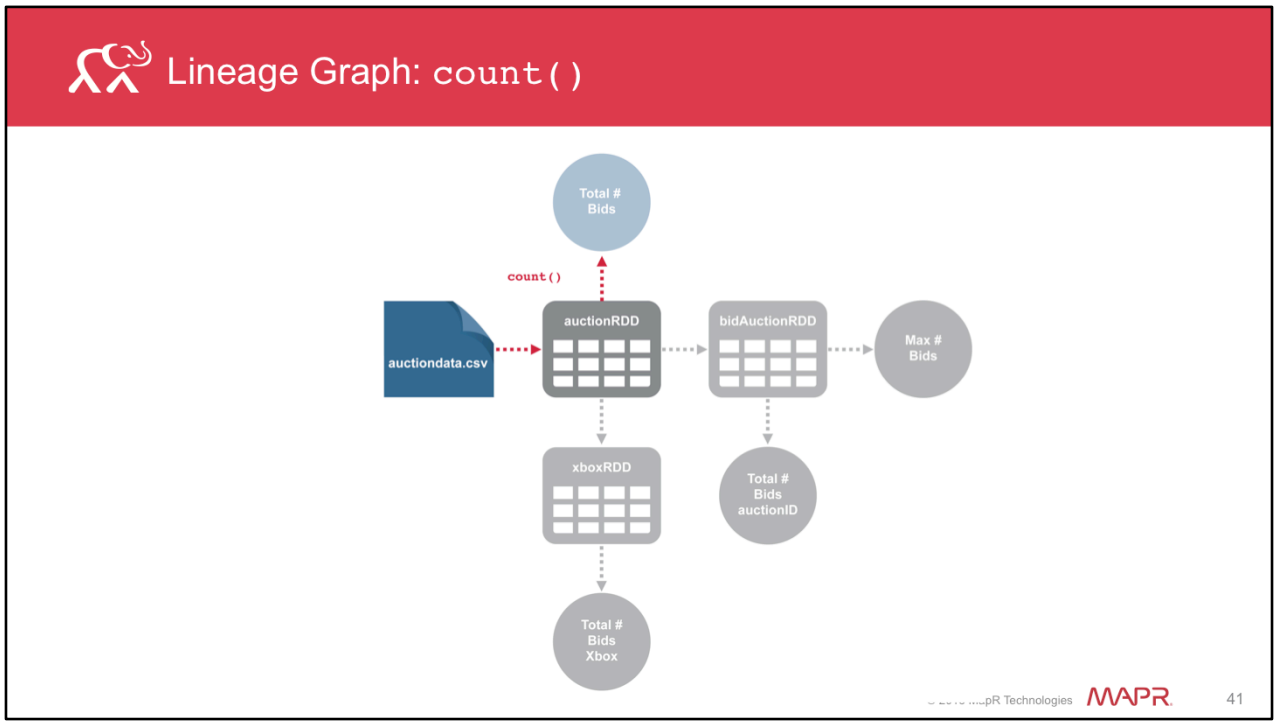


In the next section, we will discuss reasons for caching an RDD.

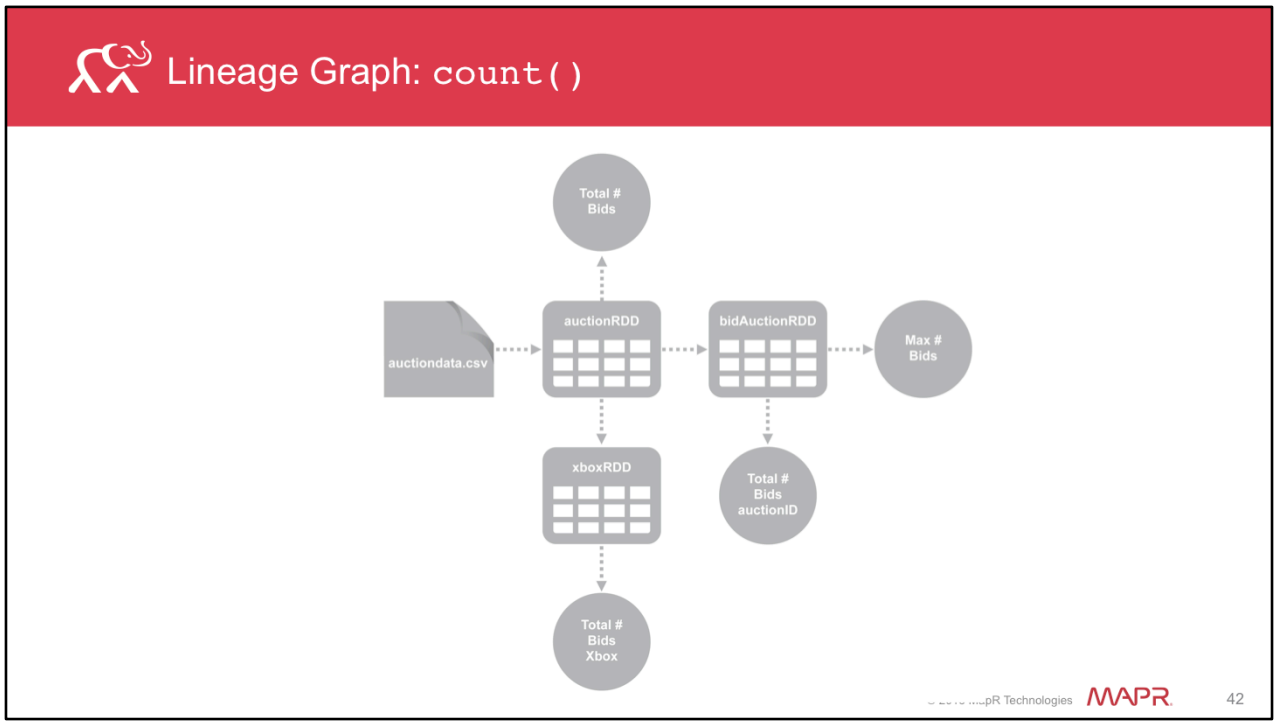


As mentioned earlier, RDDs are lazily evaluated. What this means is that even the base RDD (auctionRDD) is not created till we call an action on an RDD.

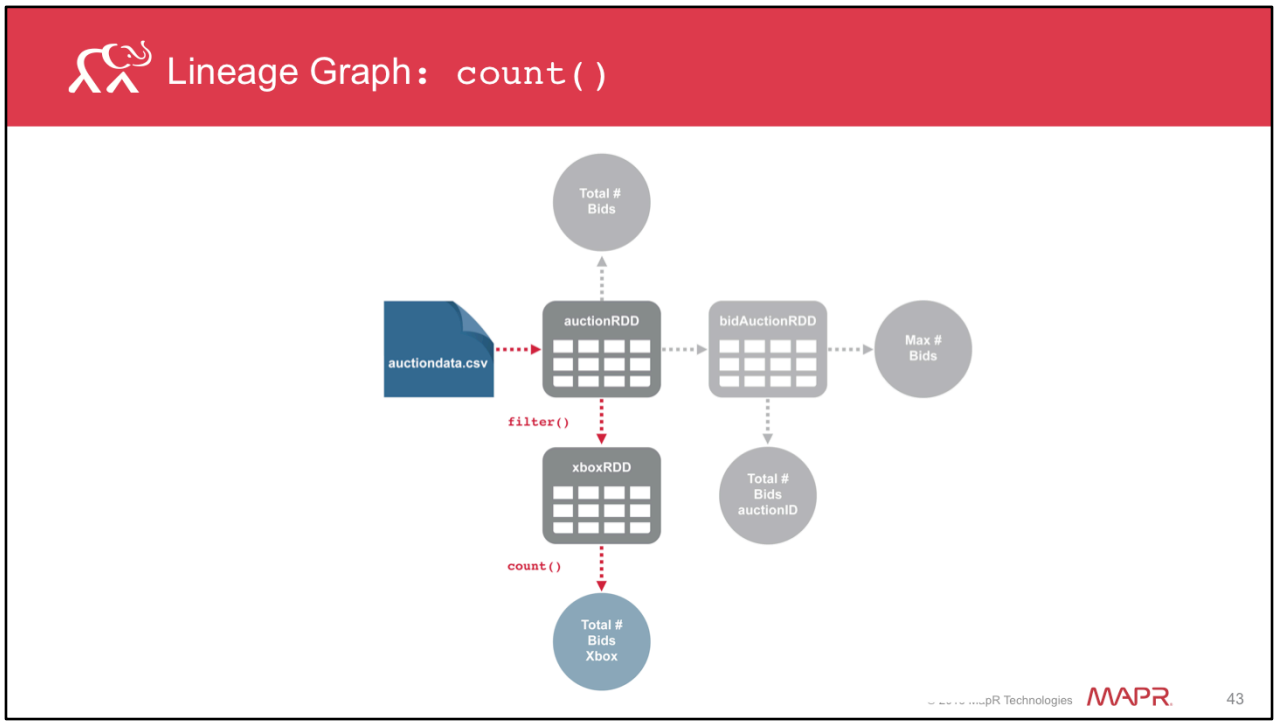
Every time we call an action on an RDD, Spark will recompute the RDD and all its dependencies.



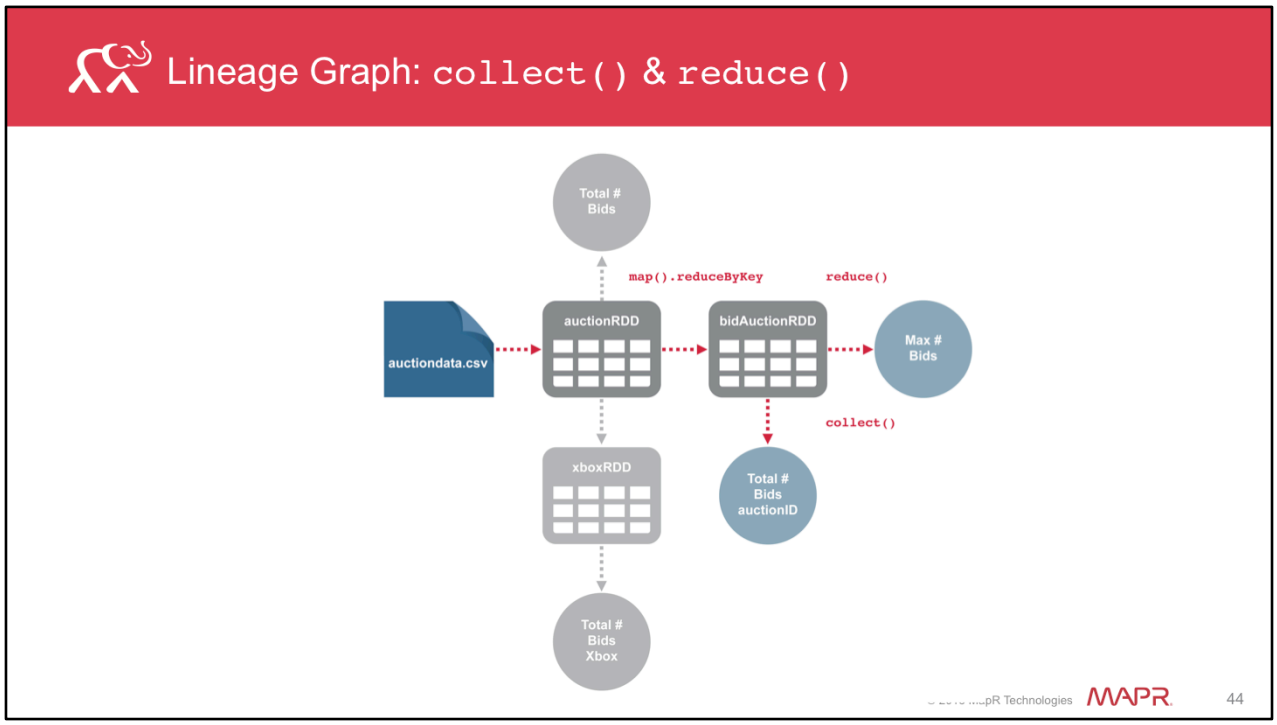
For example, when we call the count for the total number of bids, data will be loaded into the auctionRDD and then count action is applied.



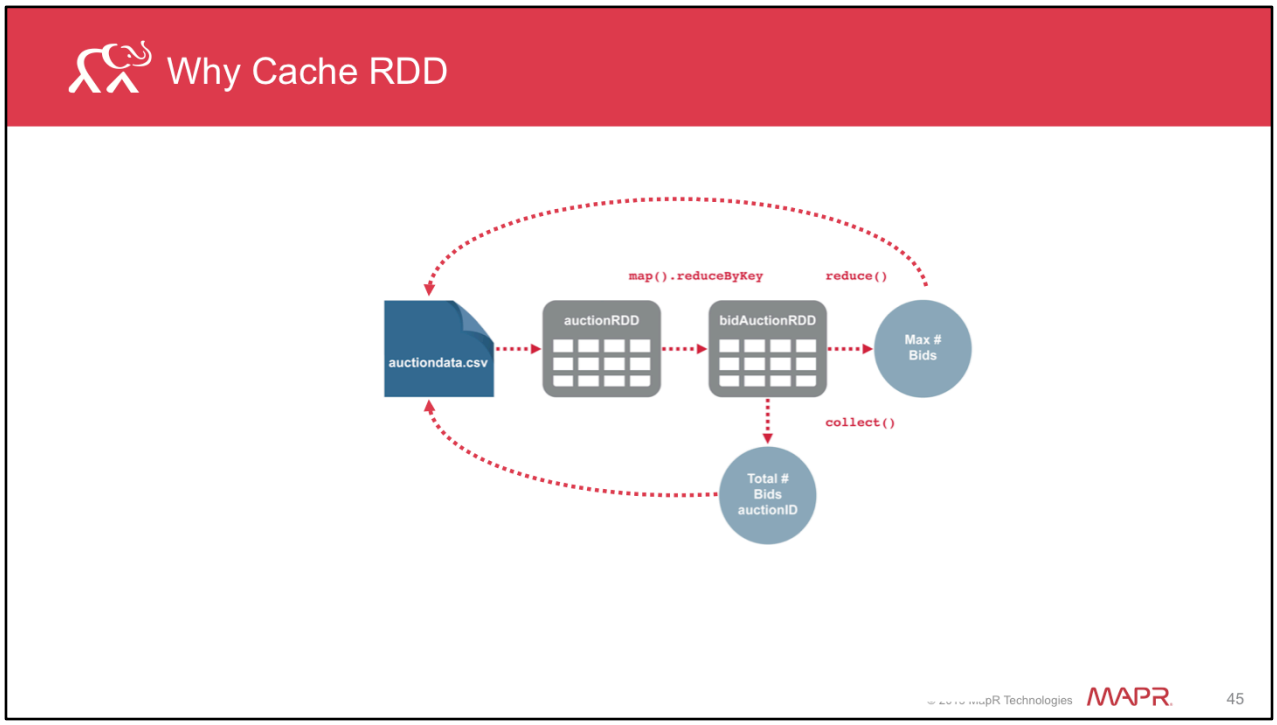
The data is no longer in memory.



Similarly for the count action for total number of bids on xbox will compute the auctionRDD and xboxRDD and then the count.

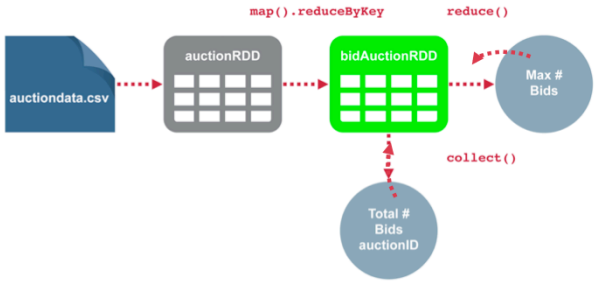


When you call the collect for the total bids per auctionid or the reduce to find the max number of bids, the data is loaded into the auctionRDD, bidAuctionRDD is computed and then the respective action operates on the RDD and the results returned. The reduce and collect will each recompute from the start.



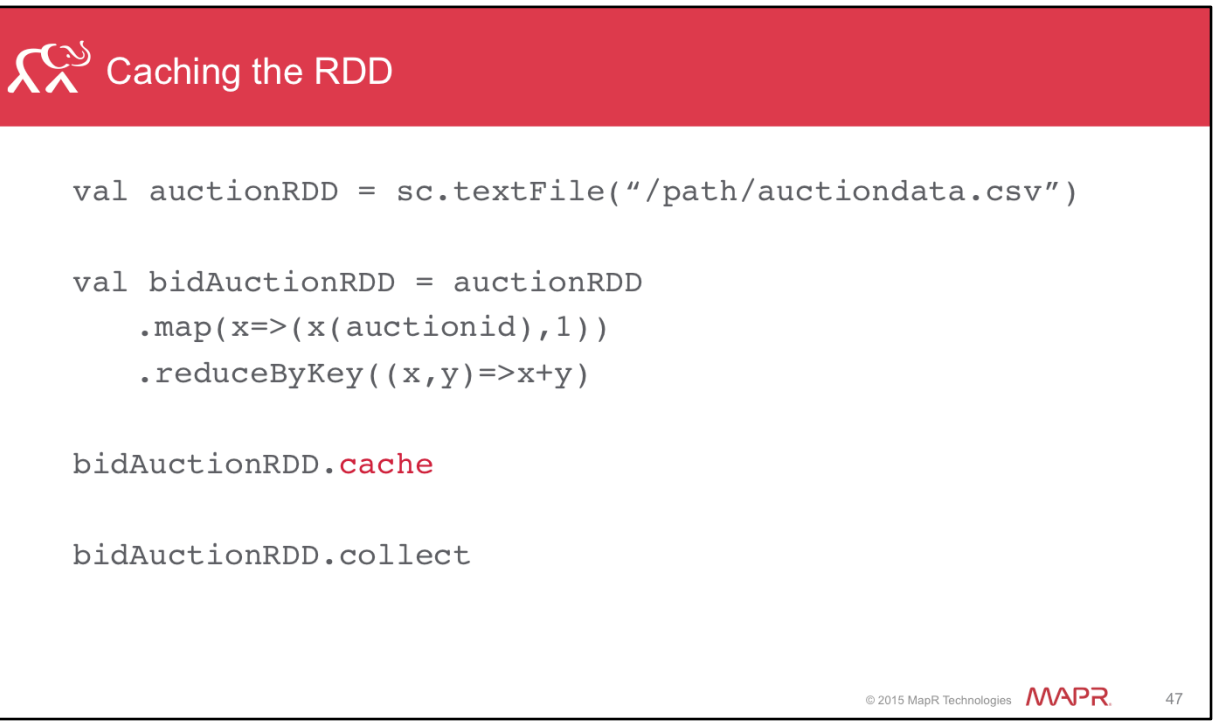
Each action is called and computed independently. In tabs 3 and 4 in the previous example, when we call reduce() or collect(), each time the process is completed and the data is removed from memory. When we call one of these actions again, the process starts from the beginning, loading the data into memory. Recomputing every time we run an action can be expensive especially for iterative algorithms.

Why Cache RDD



When there is a branching in the lineage, and we are using the same RDD multiple times, it is advisable to cache or persist the RDD(s). The RDD has already been computed and the data is already in memory. We can reuse this RDD without using any additional compute or memory resources.

We need to be careful not to cache everything, and only those that are being iterated. The cache behavior depends on the available memory. If the file does not fit in the memory, then the count reloads the data file, and then proceeds as normal.



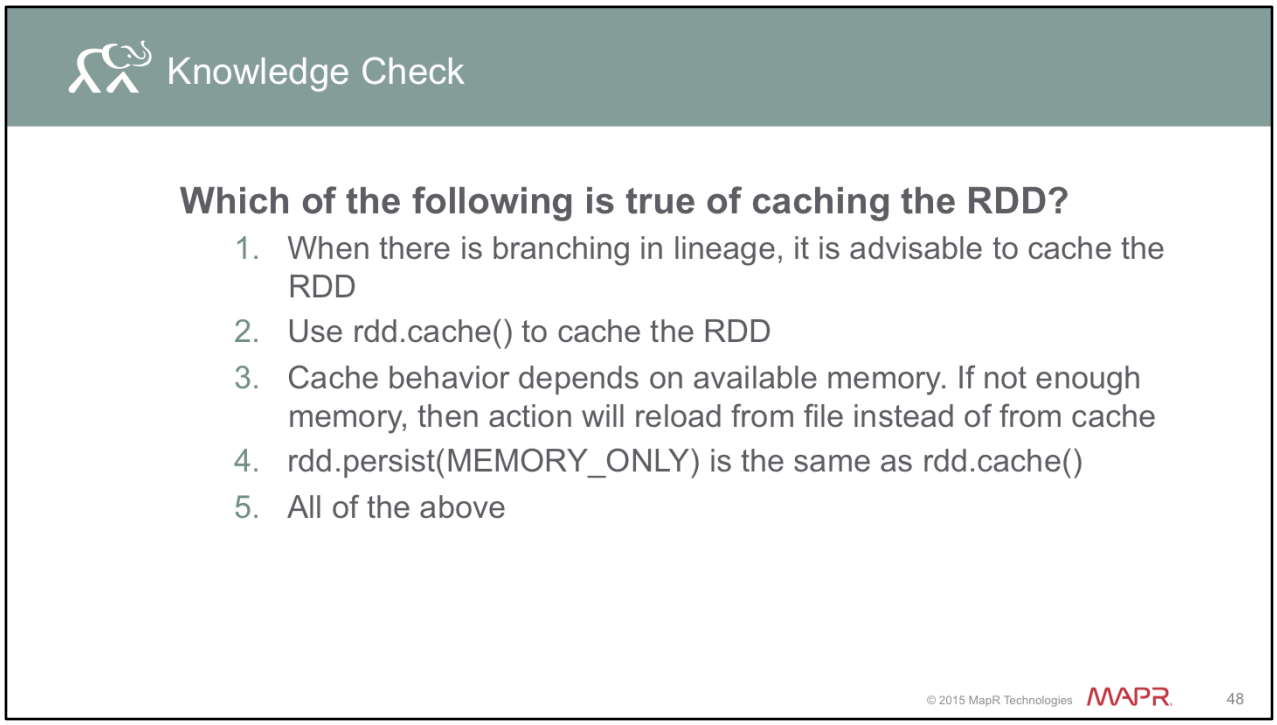
1. The first line defines the instructions on how to create the RDD, though the file is not, yet read.

2. The second line is says to apply a transformation to the base RDD.

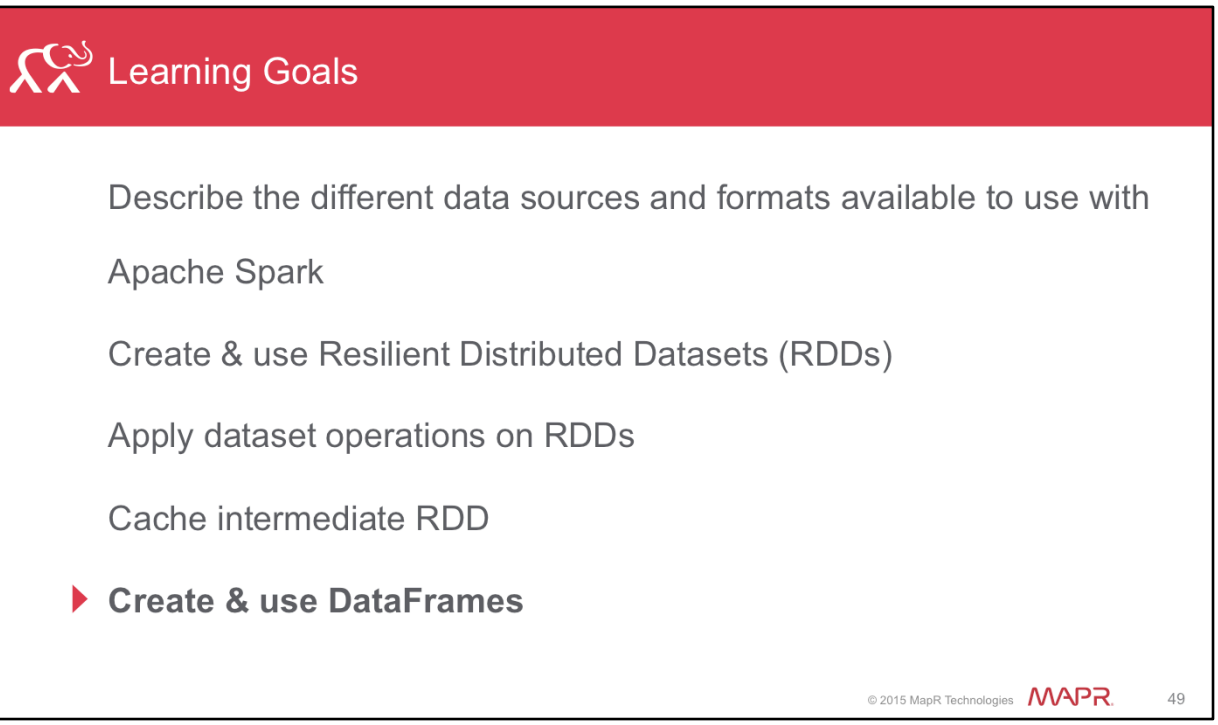
3. The third line says the cache the contents.

4. Again, nothing will happen until you to get the fourth line, the collect action. At this point, the auctiondata.csv is read, the transformation is applied, and the data is cached and collected

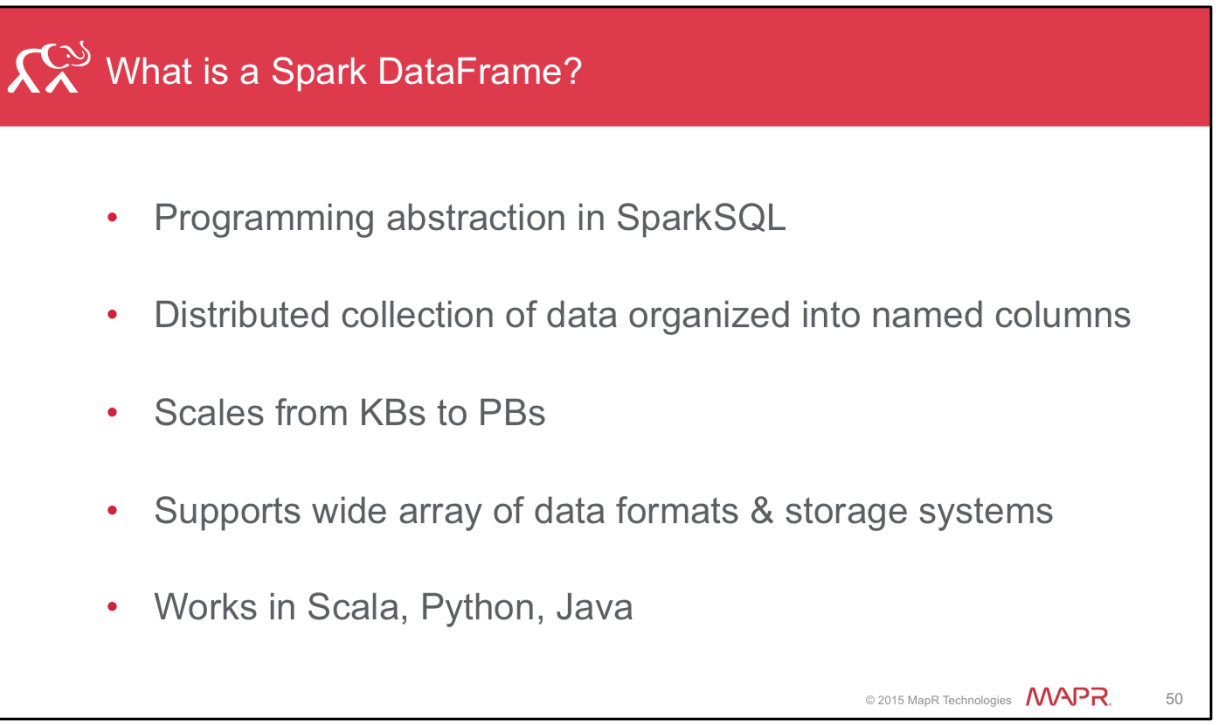
The next count or other action, will now just use the data from the cache, rather than re-loading the file and performing the first transformation.


Knowledge Check
Which of the following is true of caching the RDD?
1. When there is branching in lineage, it is advisable to cache the RDD
2. Use rdd.cache() to cache the RDD
3. Cache behavior depends on available memory. If not enough memory, then action will reload from file instead of from cache
4. rdd.persist(MEMORY_ONLY) is the same as rdd.cache()
5. All of the above
© 2015 MapR Technologies
48

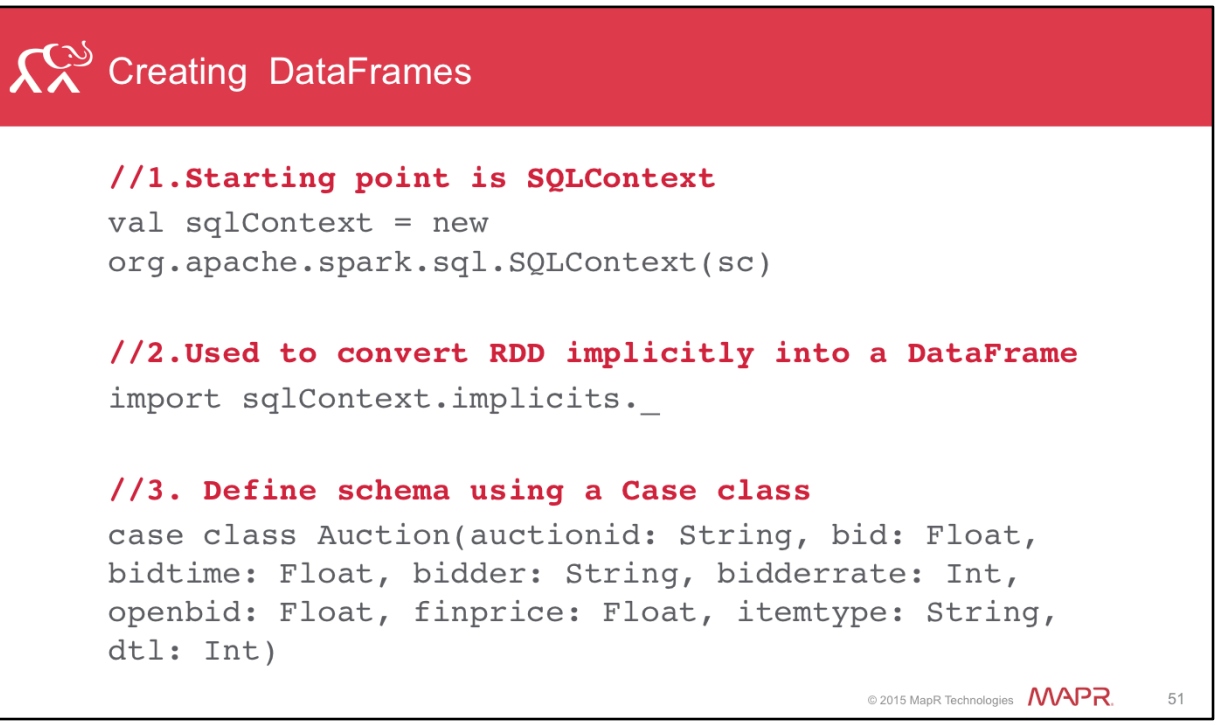
Answers: 5



In this section we are going to look at constructing and using Spark DataFrames.



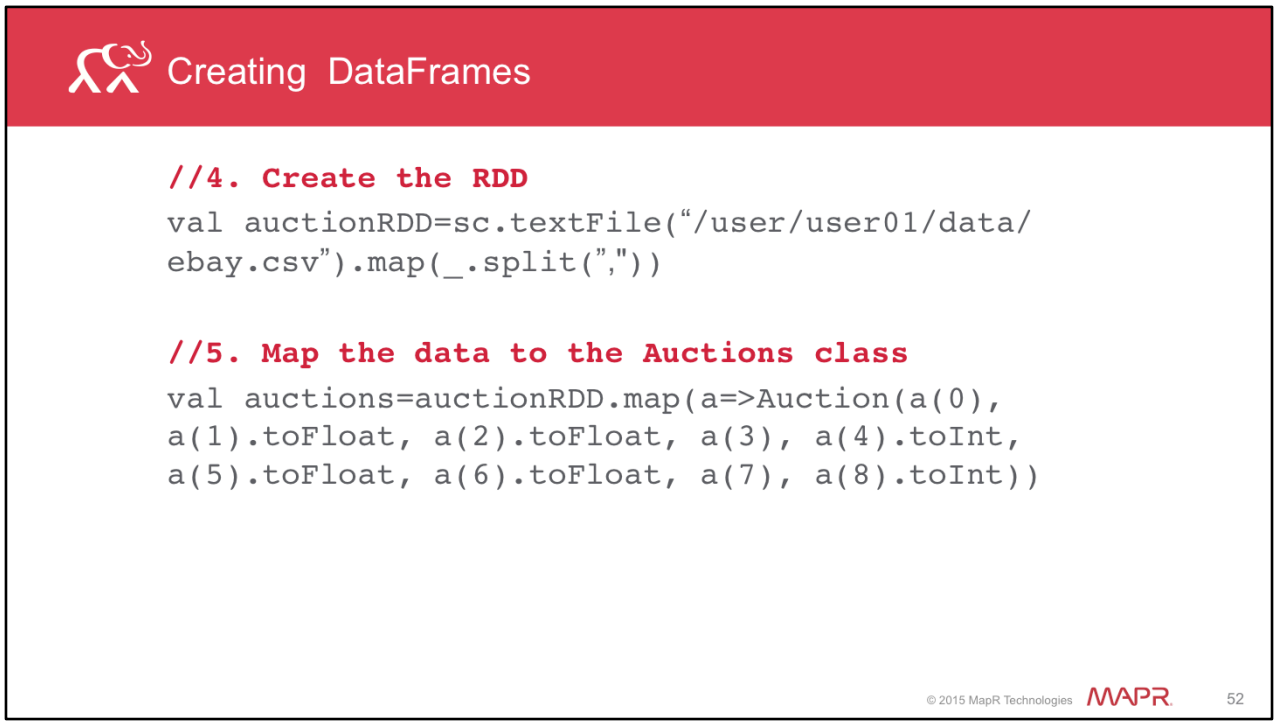
A DataFrame is distributed collection of data organized into named columns. It scales from KBs to PBs. It supports a wide array of data formats. They can be constructed from structured data files, tables in Hive, external databases or existing RDDs.
The DataFrames API is available in Scala, Python and Java.



Spark SQL supports two different methods for converting existing RDDs into DataFrames, reflection and programmatic.

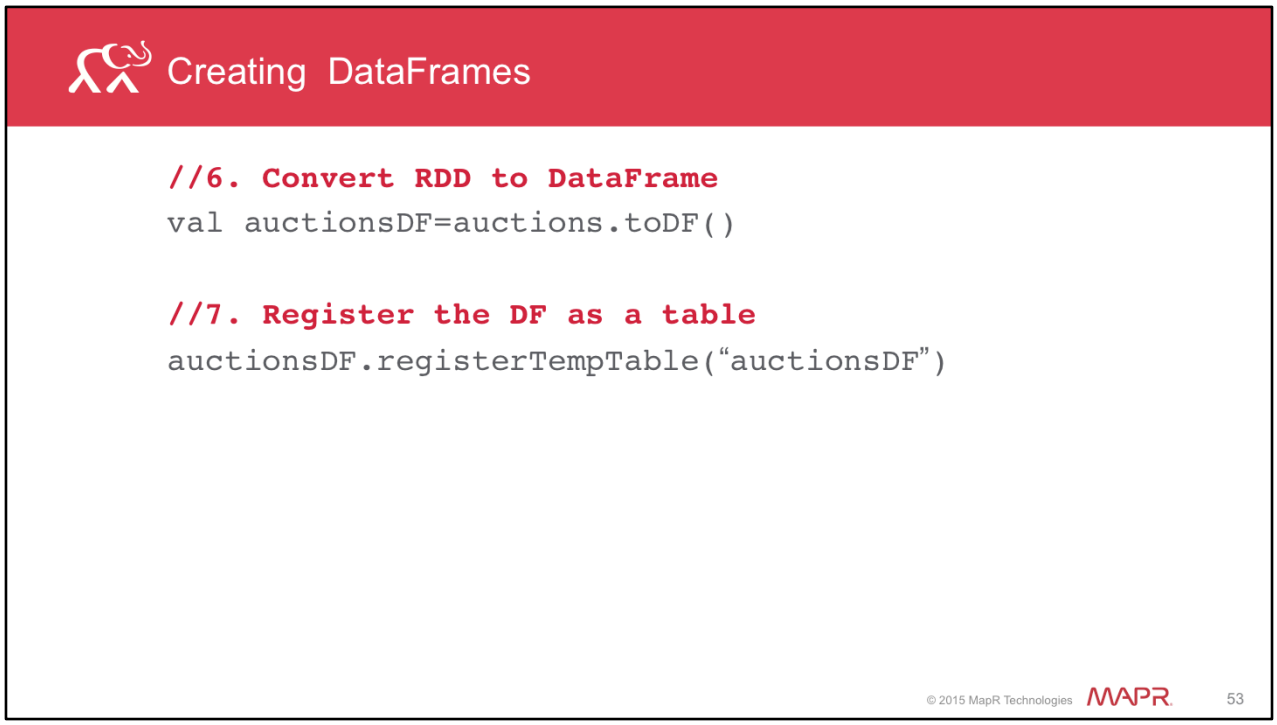
In this example, we are creating a DataFrame from an RDD using the reflection approach.

1. We first need to create a basic SQLcontext, which is the starting point for creating the DataFrame.
2. We then import sqlContext.implicits._, since we want to create a DataFrame from an existing RDD.
3. Next, we define the schema for the dataset using a Case class.



4. We create the RDD next using the SparkContext textFile method. In this example, we are creating a DataFrame from an RDD.

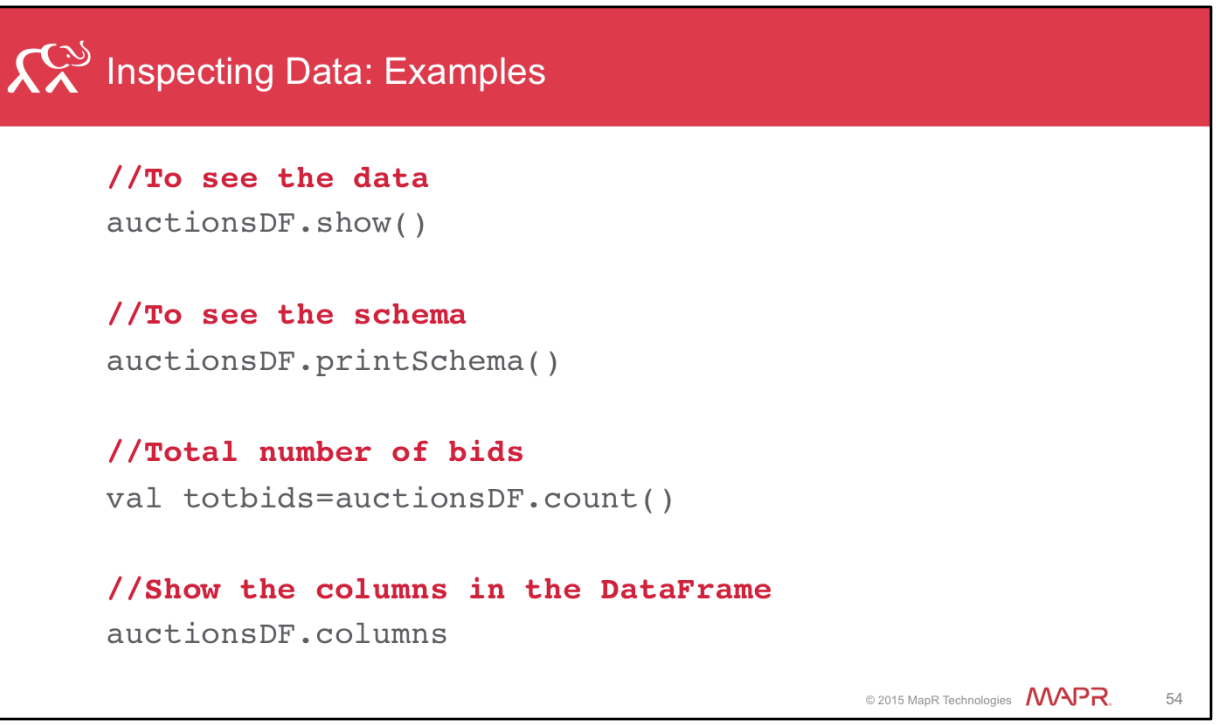
5. Once the RDD is created, we map the data to the schema i.e. the Auction class created earlier.



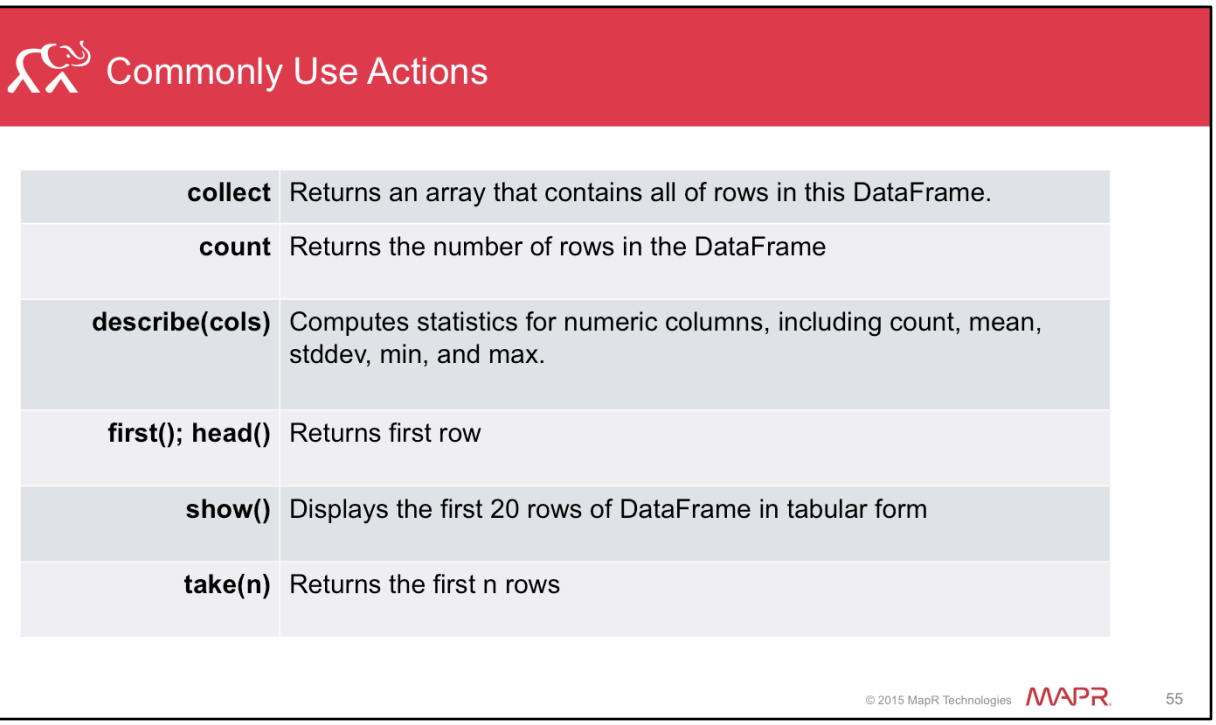
6. We then convert the RDD to a DataFrame using the toDF() method.

7. In the last step, we register the DataFrame as a table. Registering it as a table allows us to use it in subsequent SQL statements.

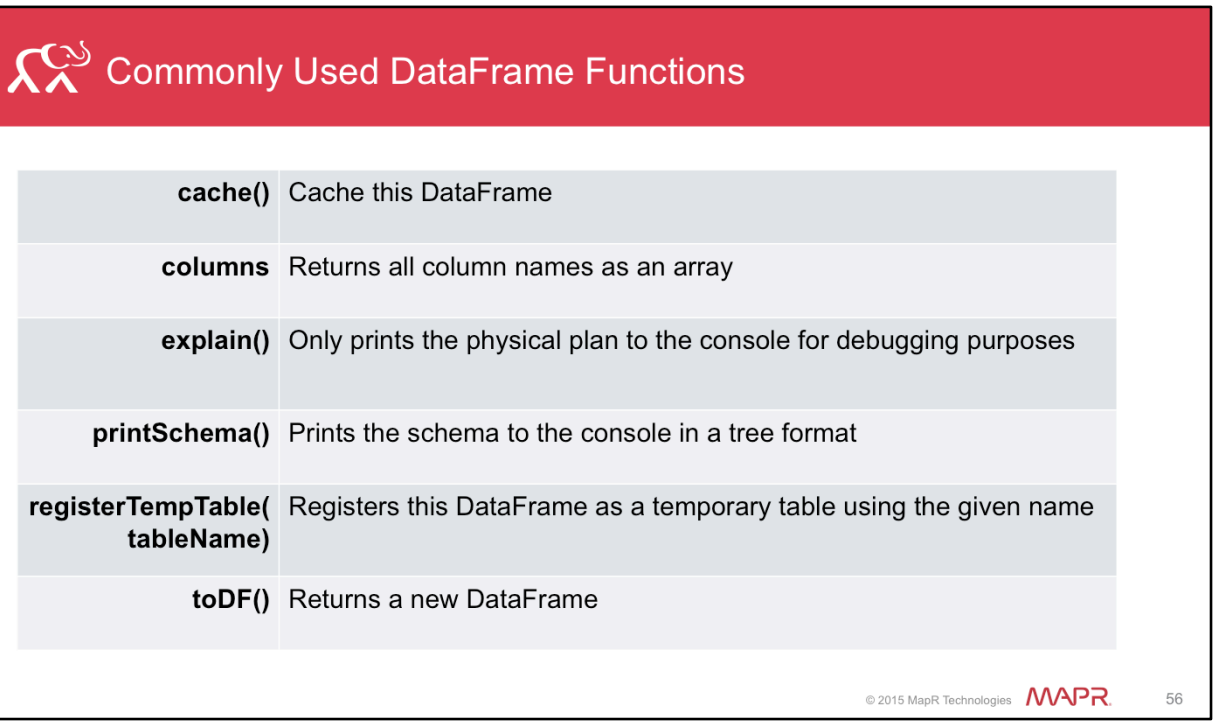
Now we can inspect the data.



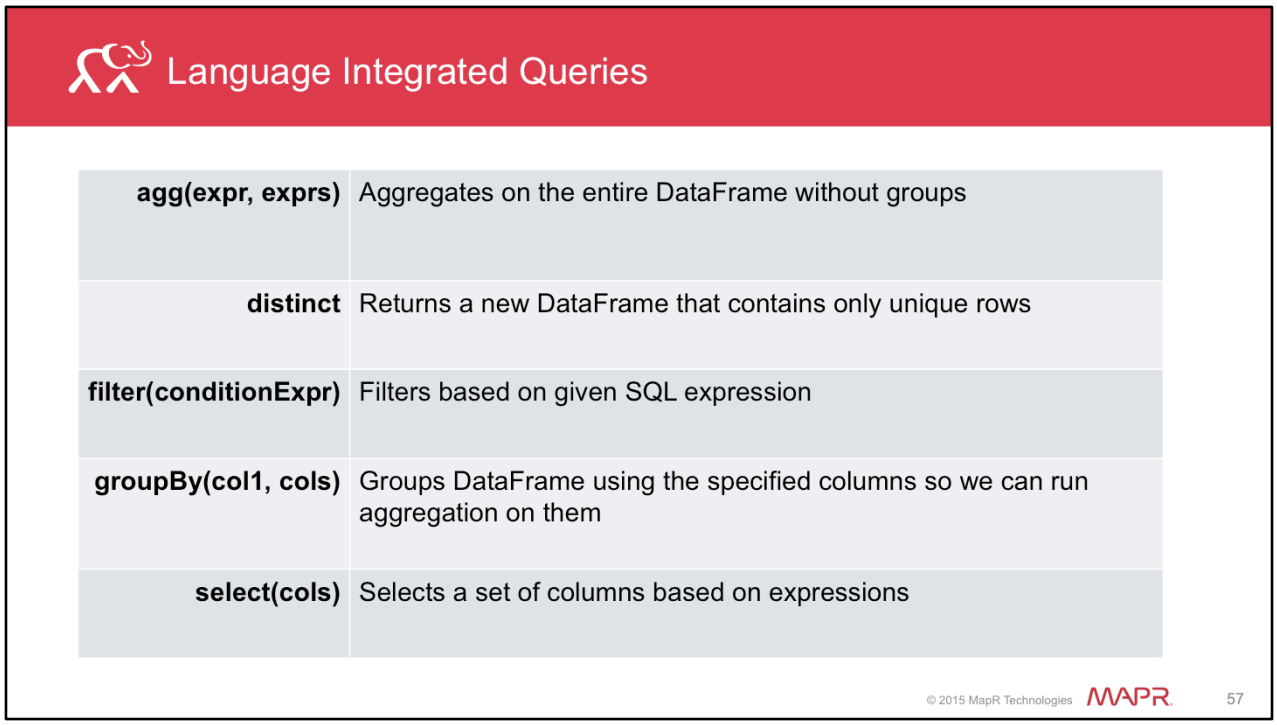
Here are some examples of using DataFrame functions and actions.



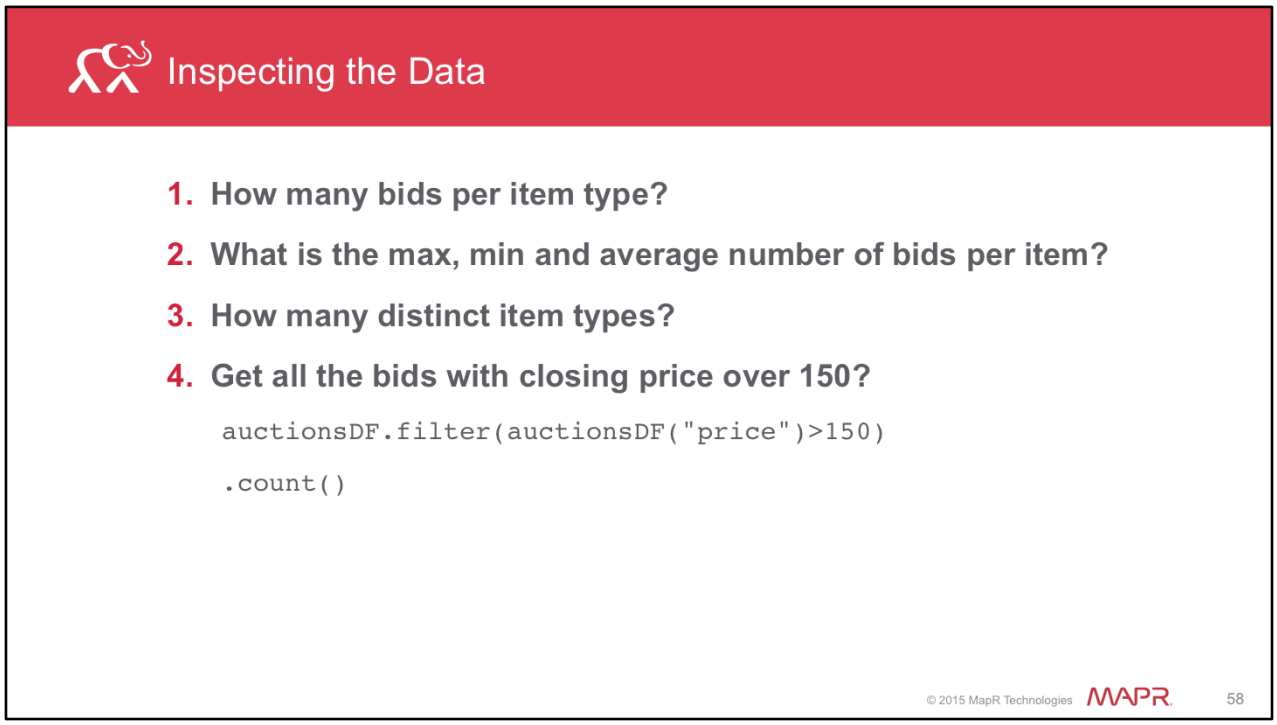
This table lists commonly used DataFrame actions. Refer to the link provided for more actions.



Here are some commonly used DataFrame functions. Refer to the link provided for more functions

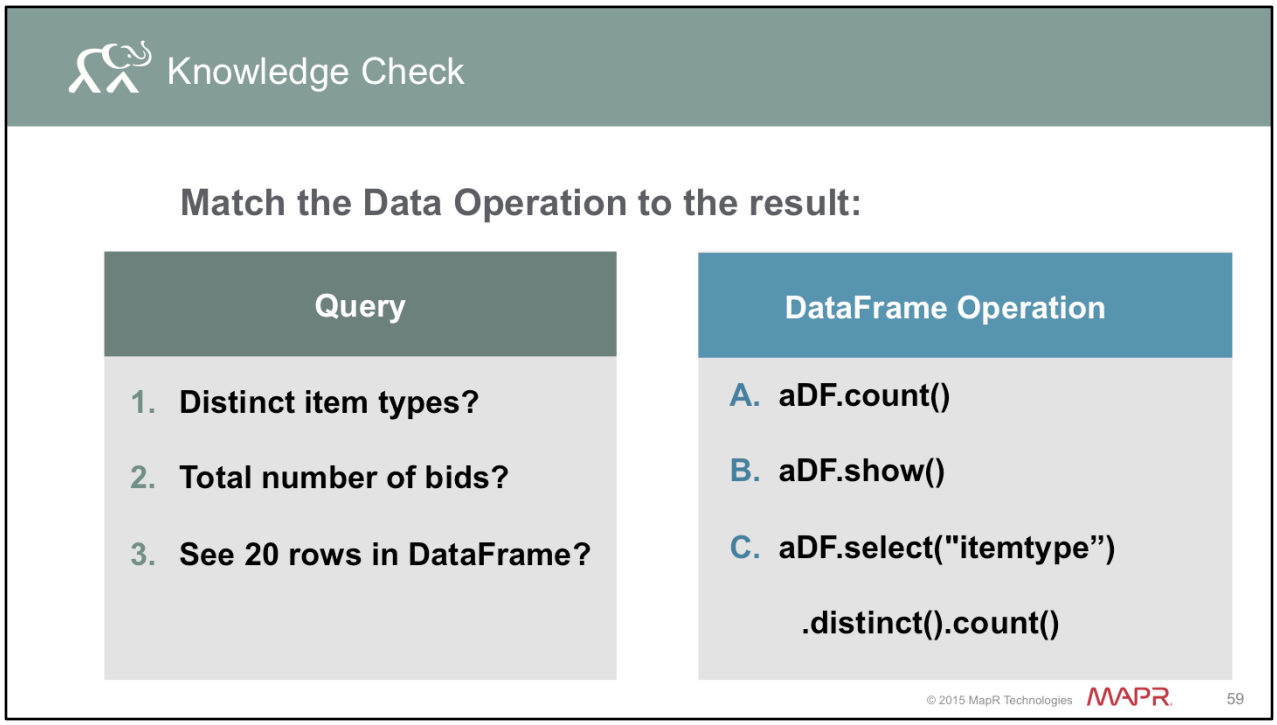


This table lists commonly used language integrated queries. Refer to the link here for more information.



Can you answer the following?

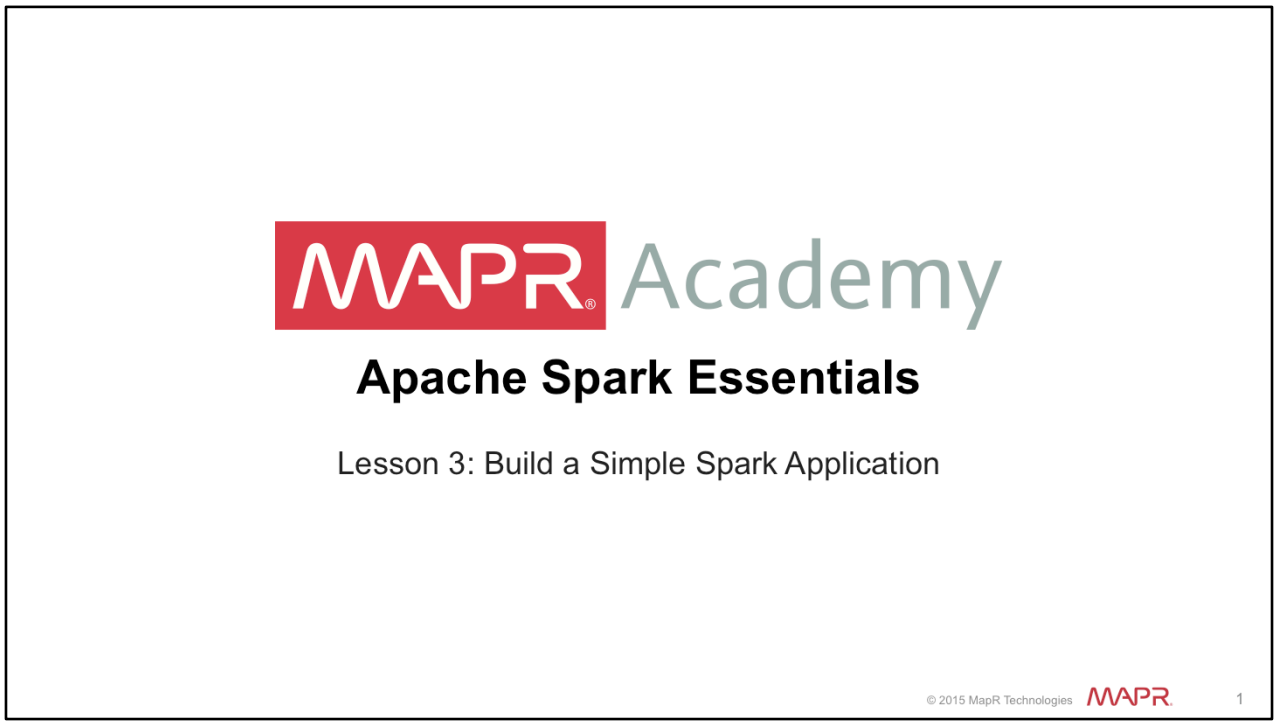
For the fourth question: we use filter with a condition (price > 150) and then apply count.


Knowledge Check
Match the Data Operation to the result:
Query
1. Distinct item types?
2. Total number of bids?
3. See 20 rows in DataFrame?
DataFrame Operation
A. aDF.count()
B. aDF.show()
C. aDF.select("itemtype")
.distinct().count()
© 2015 MapR Technologies
59

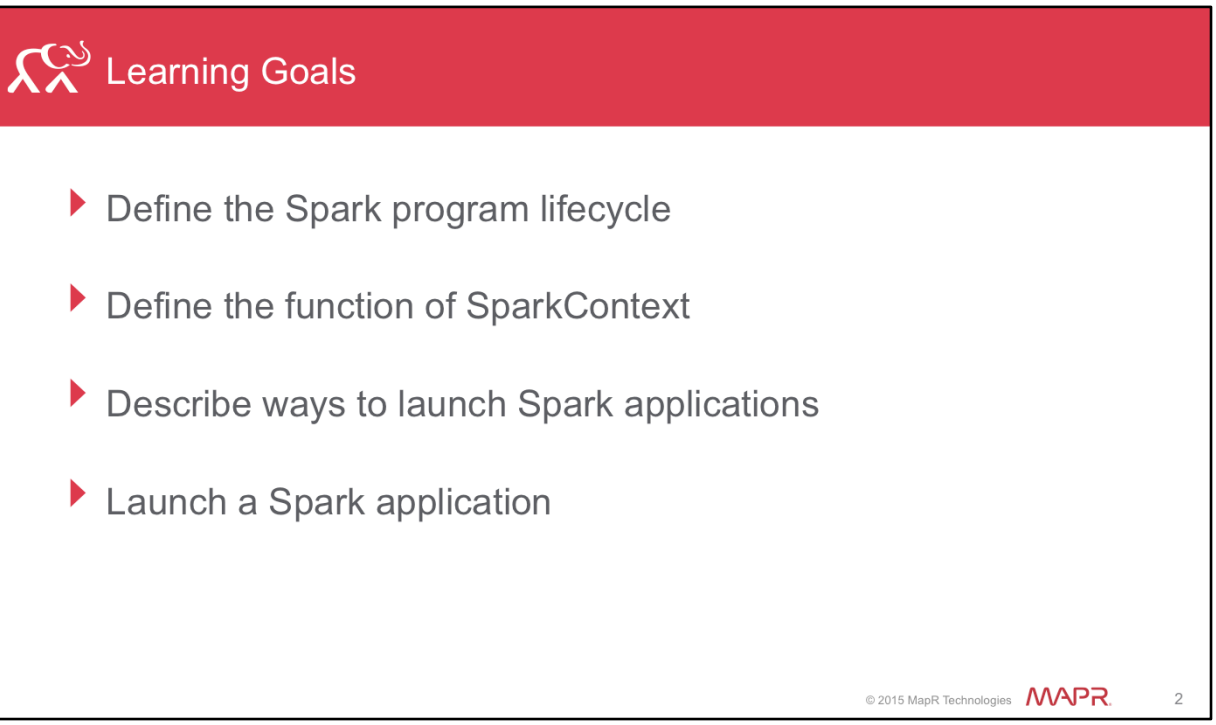
1 → c
2 → a
3 → b

In this lab, you will use the interactive shell to inspect the data in dataframes which is also based on the online auction data.

Congratulation! You have completed Lesson 2. Go on to Lesson 3 to learn about building a simple Spark application.

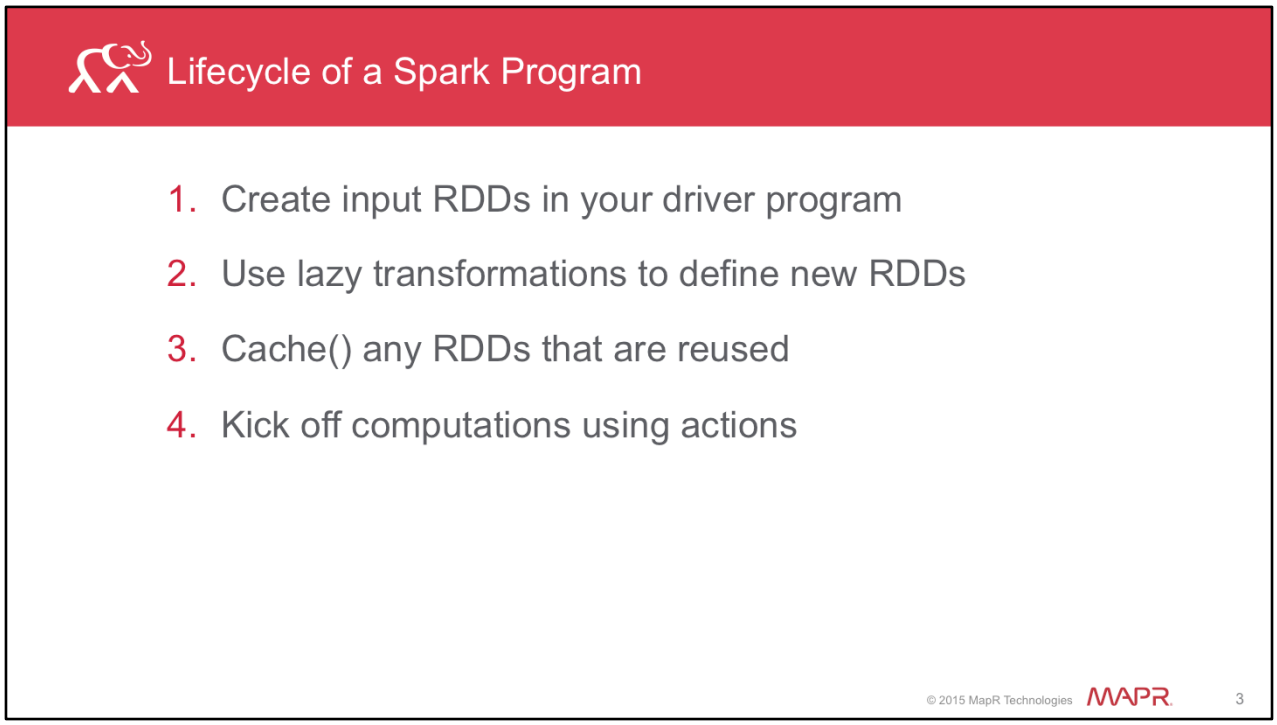


Welcome to Apache Spark Essentials – Lesson 3: Build a Simple Spark Application. In this lesson, we will build a simple standalone Spark application.



At the end of this lesson, you will be able to:

1. Define the Spark program lifecycle
2. Define function of SparkContext
3. Describe ways to launch Spark applications
4. Launch a Spark application

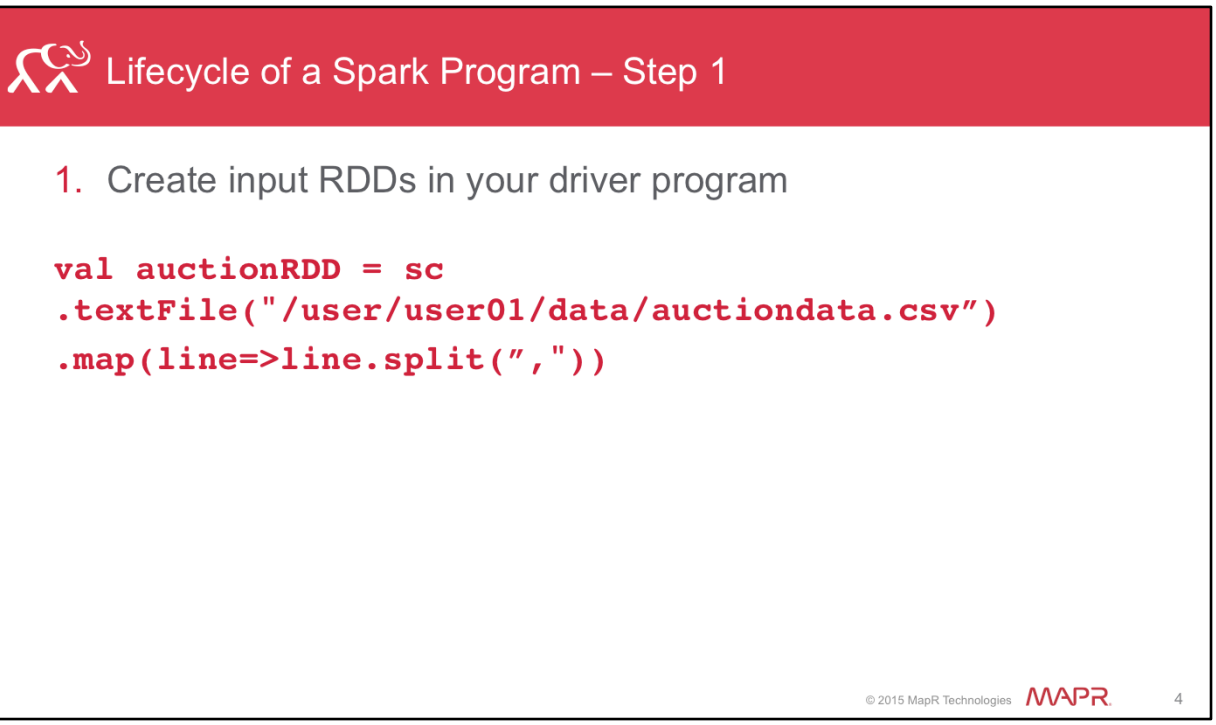


Create some input RDDs from external data or parallelize a collection in your driver program.

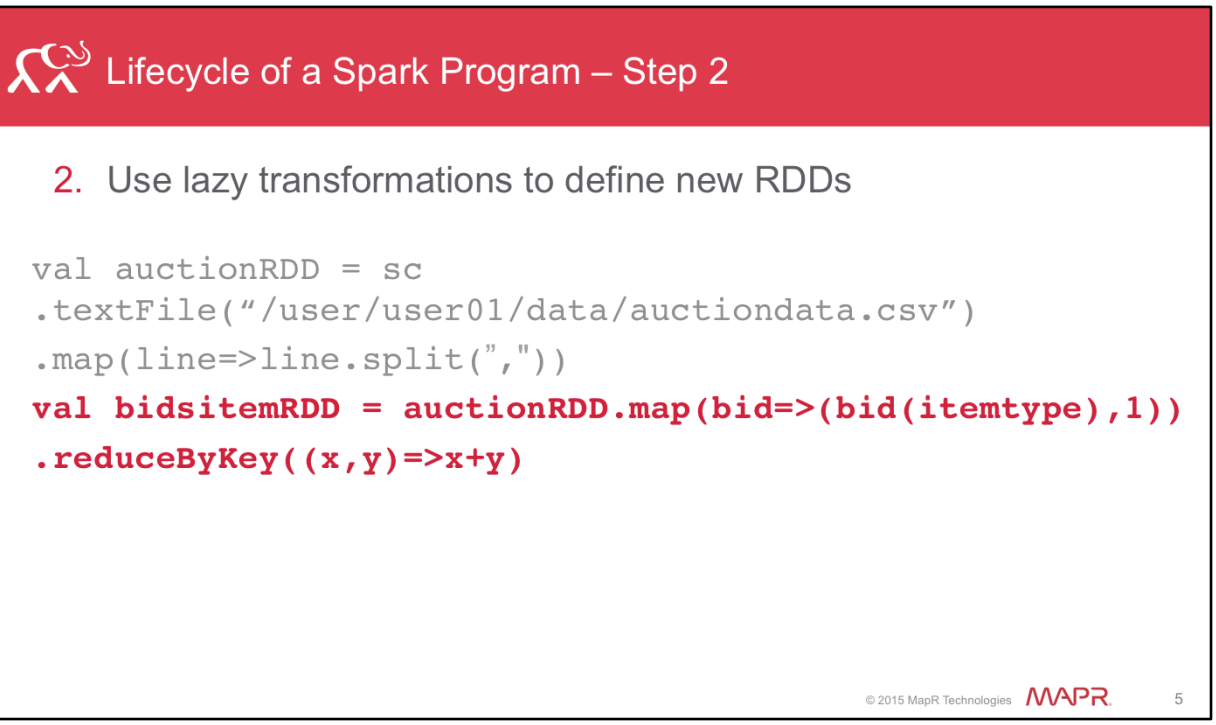
Lazily transform them to define new RDDs using transformations like filter()or map()

Ask Spark to cache()any intermediate RDDs that will need to be reused.

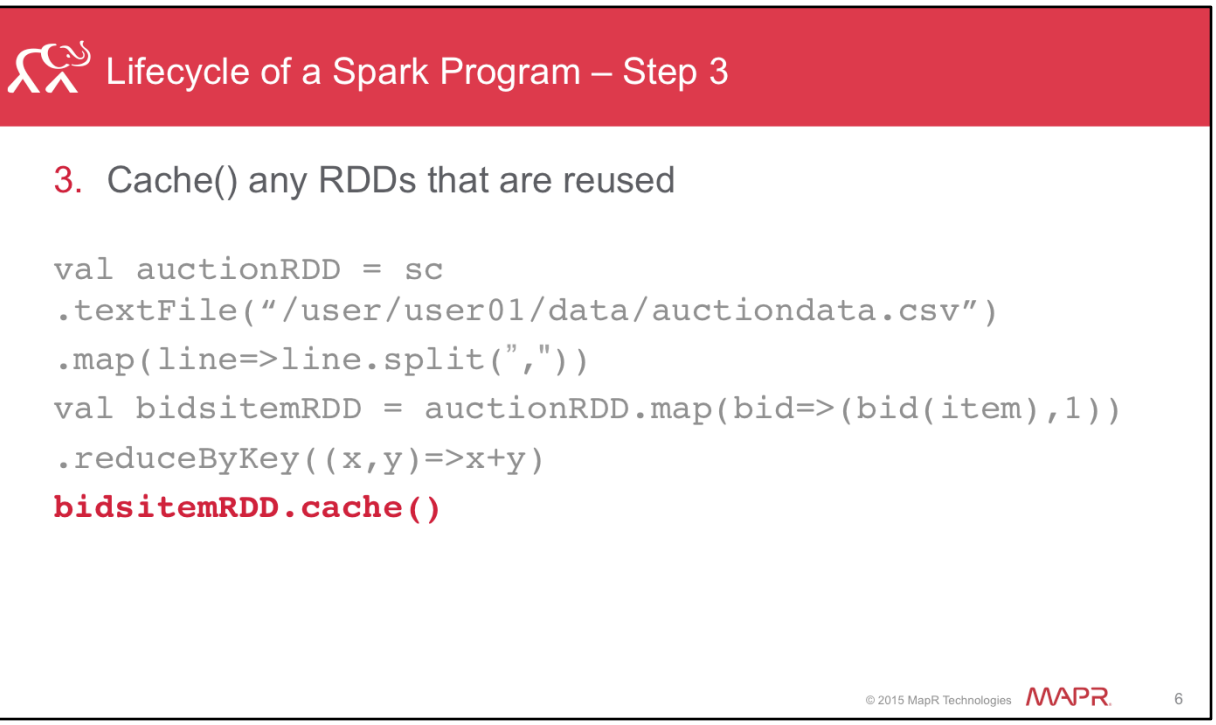
Launch actions such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark.



We did this in the previous lesson where we created an RDD using the SparkContext textFile method to load a csv file.

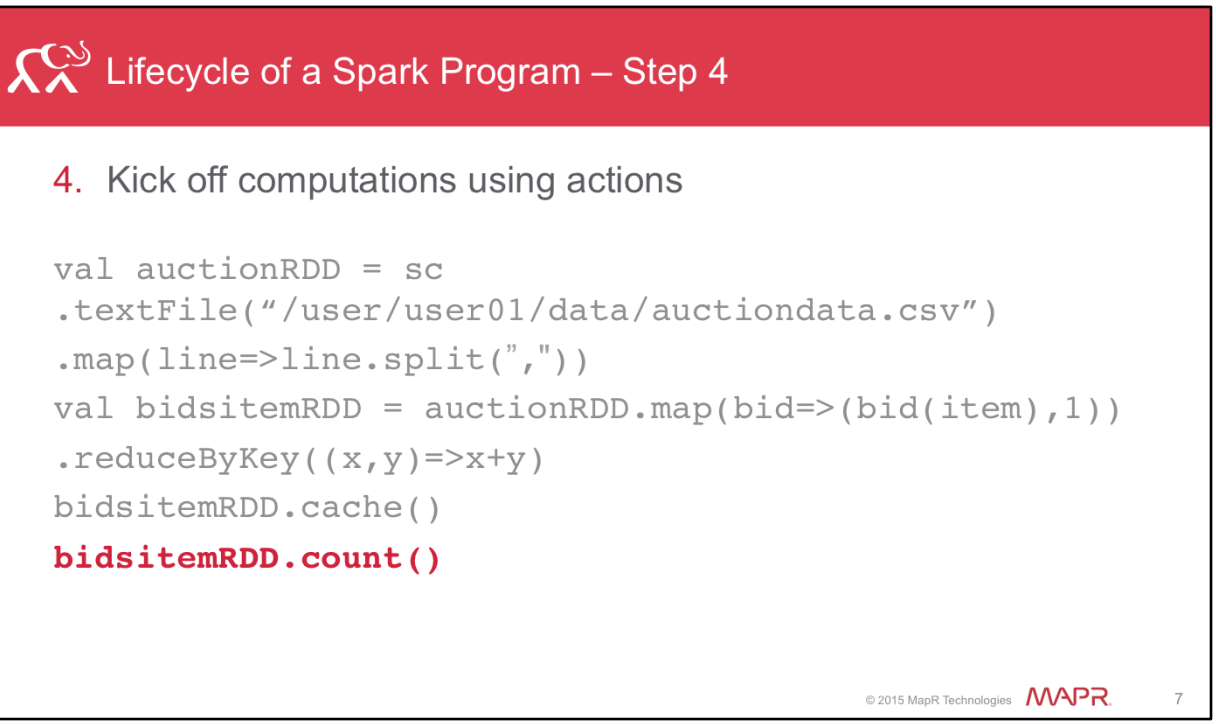


We can apply a transformation to the input RDD, which results in another RDD. In this example, we apply a map and a reduceByKey transformation on auctionRDD to create bidsitemRDD



When we run the command for an action, Spark will load the data, create the inputRDD, and then compute any other defined transformations and actions.

In the example here, if we are going to apply actions and transformations to the bidsitemRDD, we should cache this RDD so that we do not have to recompute the inputRDD each time we perform an action on bidsitemRDD.



Once you cache the intermediate RDD, you can launch actions on this RDD without having to recompute everything from the start.

In this example, we launch the count action on the cached bidsitemRDD.

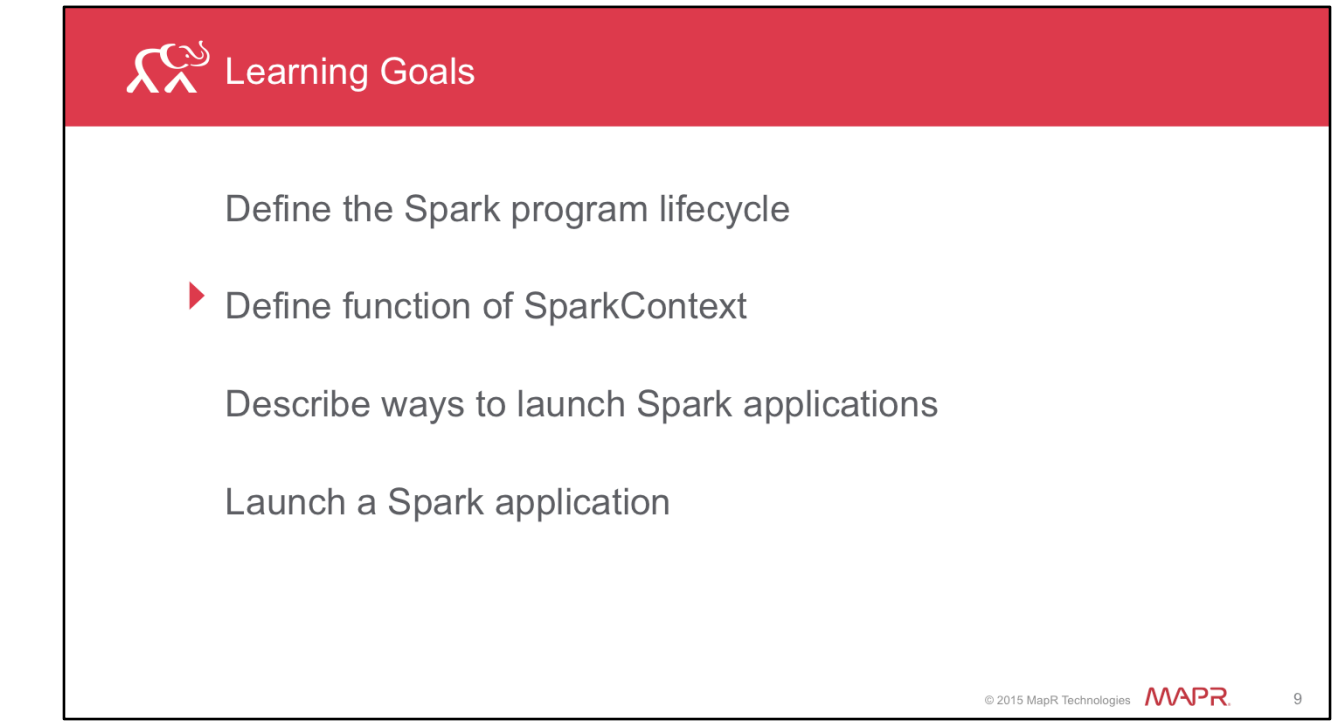
Knowledge Check

Place the steps of a Spark application lifecycle in the right order:

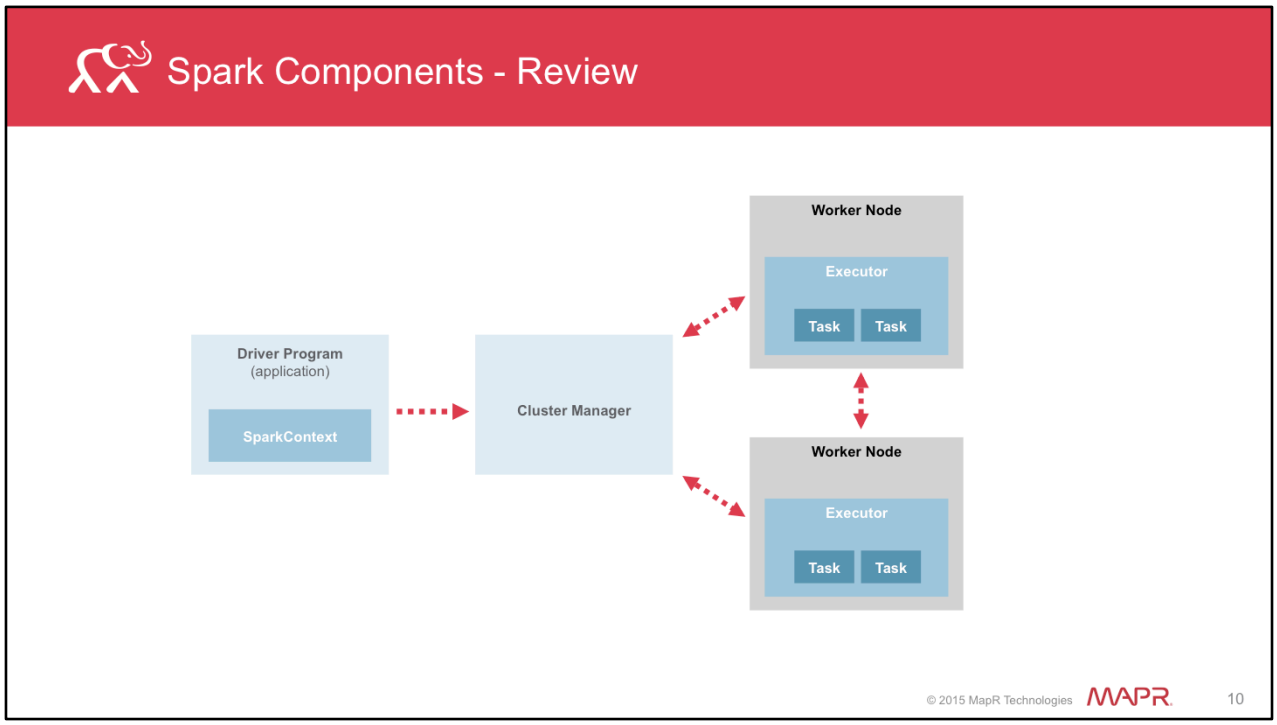
1. `val burglaries=sfpdRDD.filter(line=>line.contains("BURGLARIES"))`
2. `val sfpdRDD=sc.textFile("/user/user01/data/sfpd.csv")`
3. `val totburglaries=burglaries.count()`
4. `burglaries.cache()`

1	1,2,4,3
2	2,1,4,3
3	3,1,4,2

Answers:



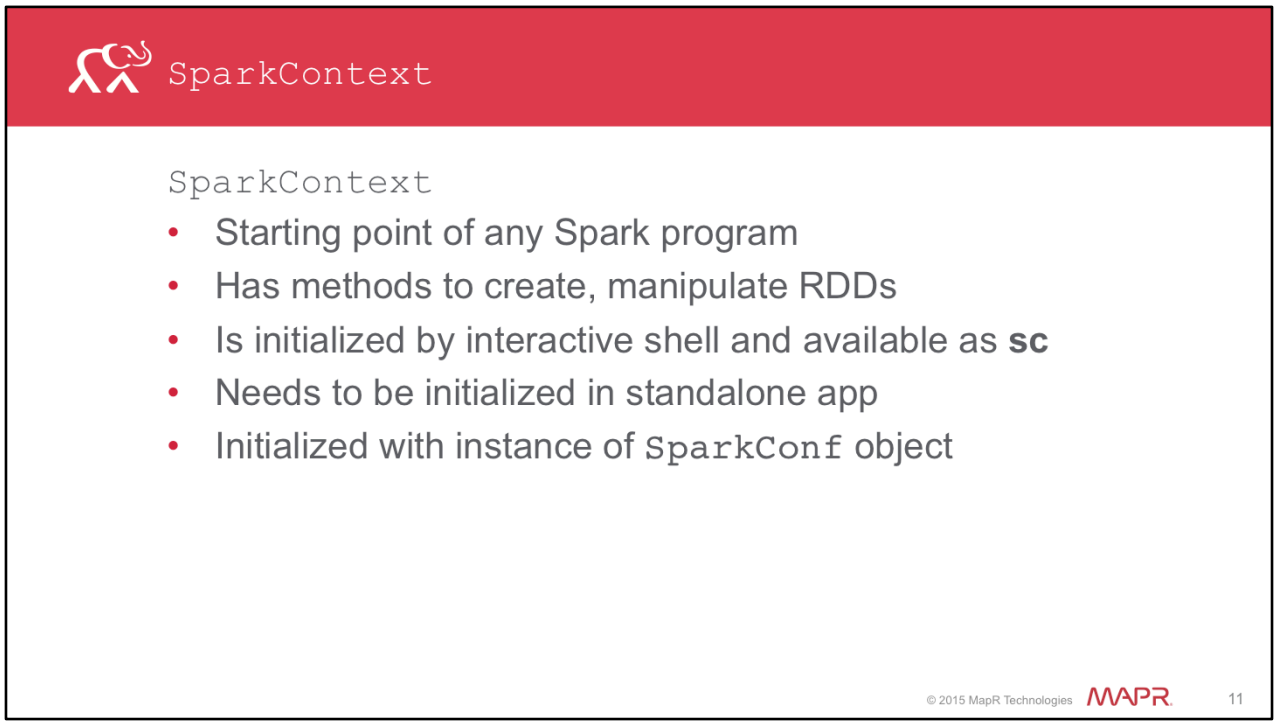
In this section. we take a look at the function of SparkContext.



This is a review of Spark components. A Spark cluster consists of two processes, a driver program and multiple workers nodes each running an executor process.

The driver program runs on the driver machine, the worker programs run on cluster nodes or in local threads.

1. The first thing a program does is to create a SparkContext object. This tells Spark how and where to access a cluster
2. SparkContext connects to cluster manager. Cluster manager allocates resources across applications
3. Once connected, Spark acquires executors in the worker nodes (an executor is a process that runs computations and stores data for your application)
4. Jar or python files passed to the SparkContext are then sent to the executors.
5. SparkContext will then send the tasks for the executor to run.

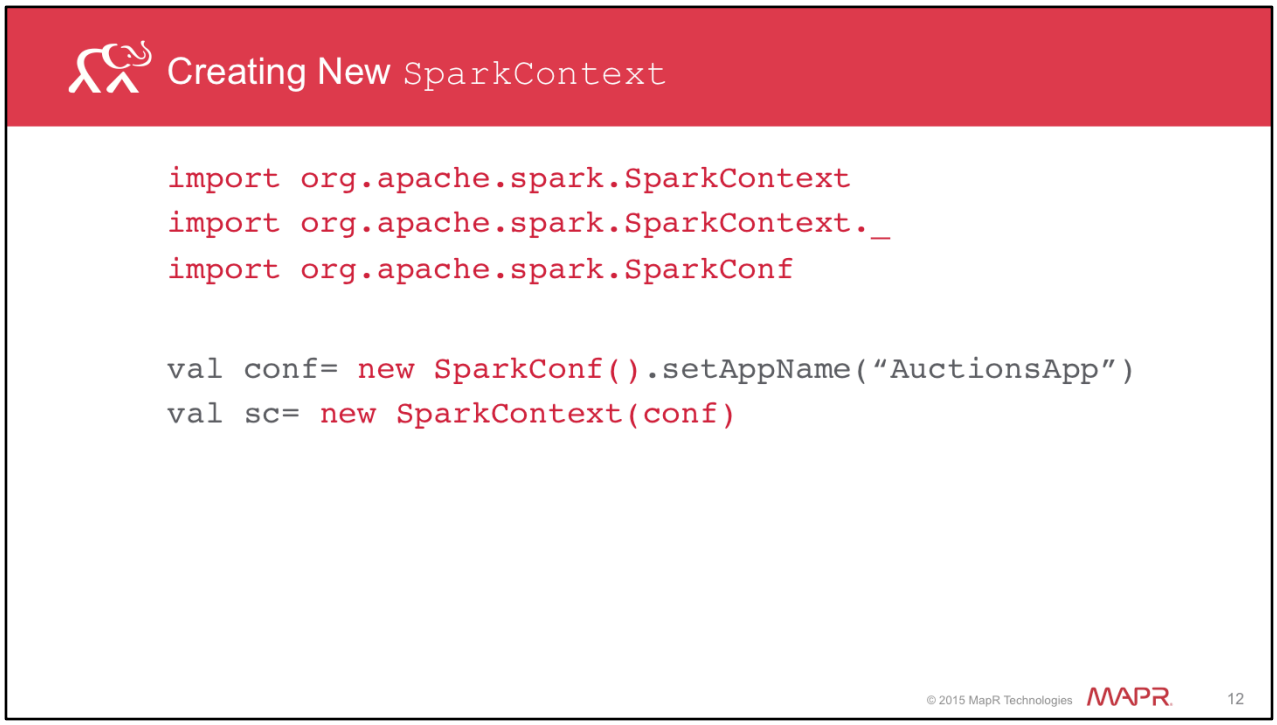


Looking at the Spark Components, SparkContext is the starting point of any Spark program. It tells Spark how and where to access the cluster.

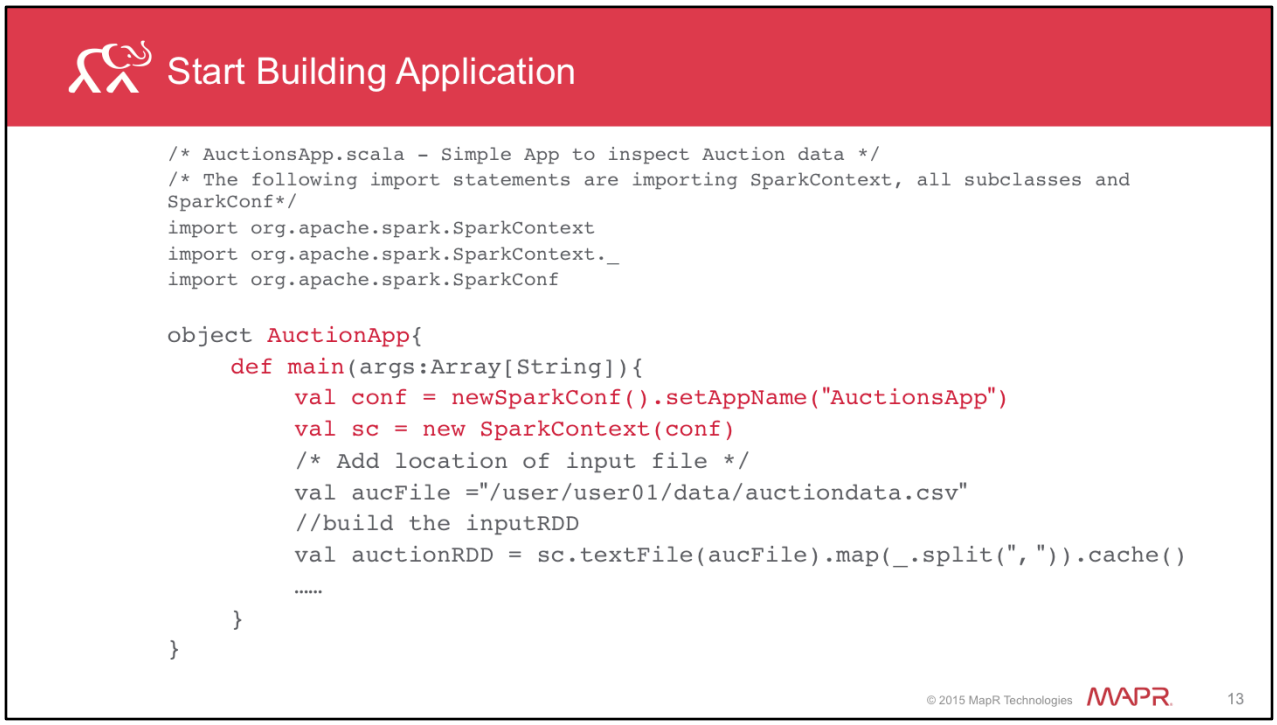
To create and manipulate RDDs, we use various methods in SparkContext.

The interactive shell, in either Scala or Python, initializes SparkContext and makes it available as a variable "sc." However, we need to initialize SparkContext when we build applications.

We can initialize the SparkContext with an instance of the SparkConf object.



If you are building a standalone application, you will need to create the SparkContext. In order to do this, you import the classes listed here.
You create a new SparkConf object and you can set the application name.
You then create a new SparkContext. As you will see next, you create the SparkConf and SparkContext within the main method.



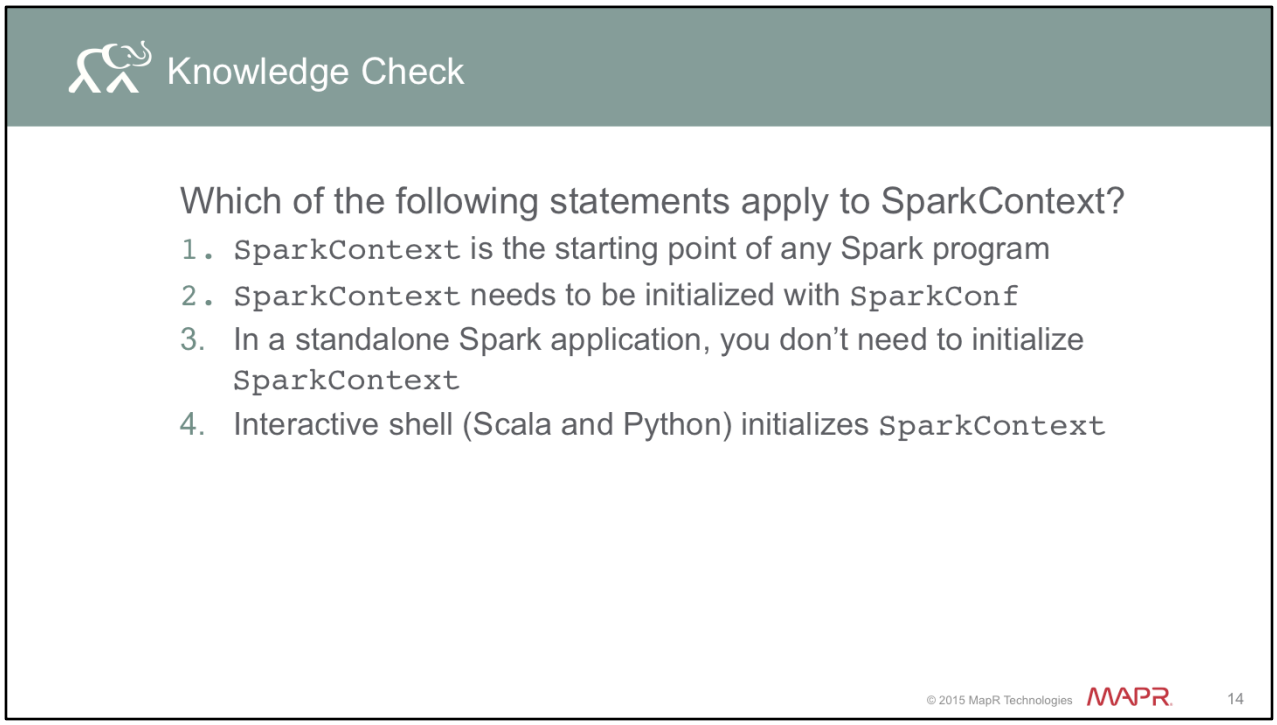
The next step is to define the class (note that this is the same name as the file). The application should define a main() method instead of extending scala.App

Within the main method here, we are creating the SparkContext.

We declare aucFile that points to the location of the auctionsdata.csv and then use the SparkContext textFile method to create the RDD. Note that we are caching this RDD.

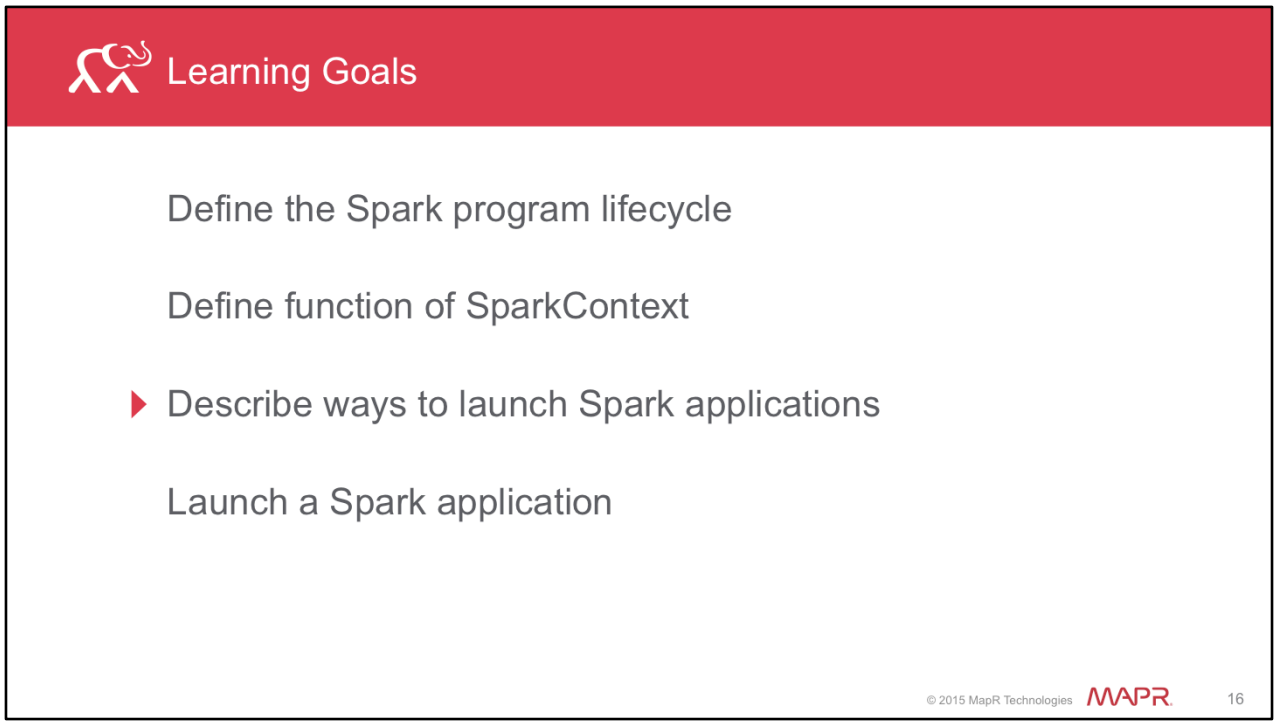
You can that more code to this file for example, to inspect the detail and print the results to the console.

Note that you can also build the application in Java or Python.

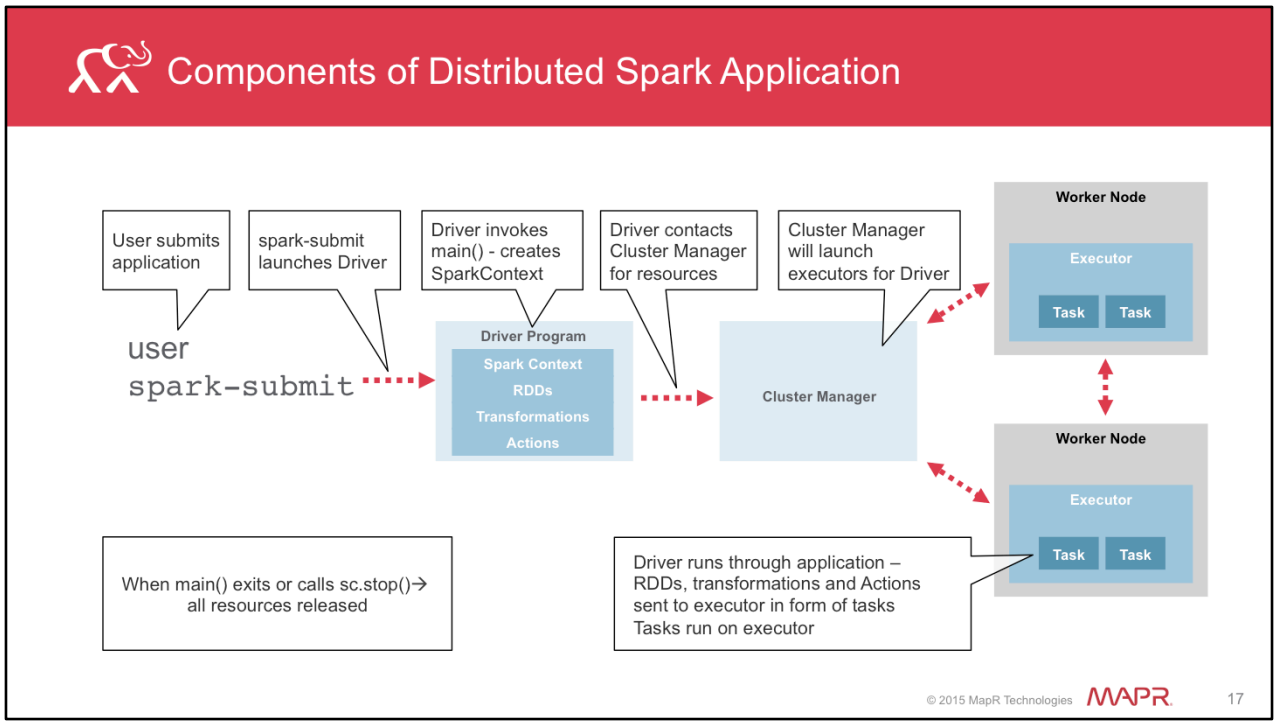

Knowledge Check
Which of the following statements apply to SparkContext?
1. SparkContext is the starting point of any Spark program
2. SparkContext needs to be initialized with SparkConf
3. In a standalone Spark application, you don't need to initialize SparkContext
4. Interactive shell (Scala and Python) initializes SparkContext
© 2015 MapR Technologies
14

Answers: 1,2,4

In this lab, you will start building your application. You will create an application that is going to read data from the auctiondata.csv file and load it into an RDD.



In this section we will take a look at the different ways to launch Spark applications.

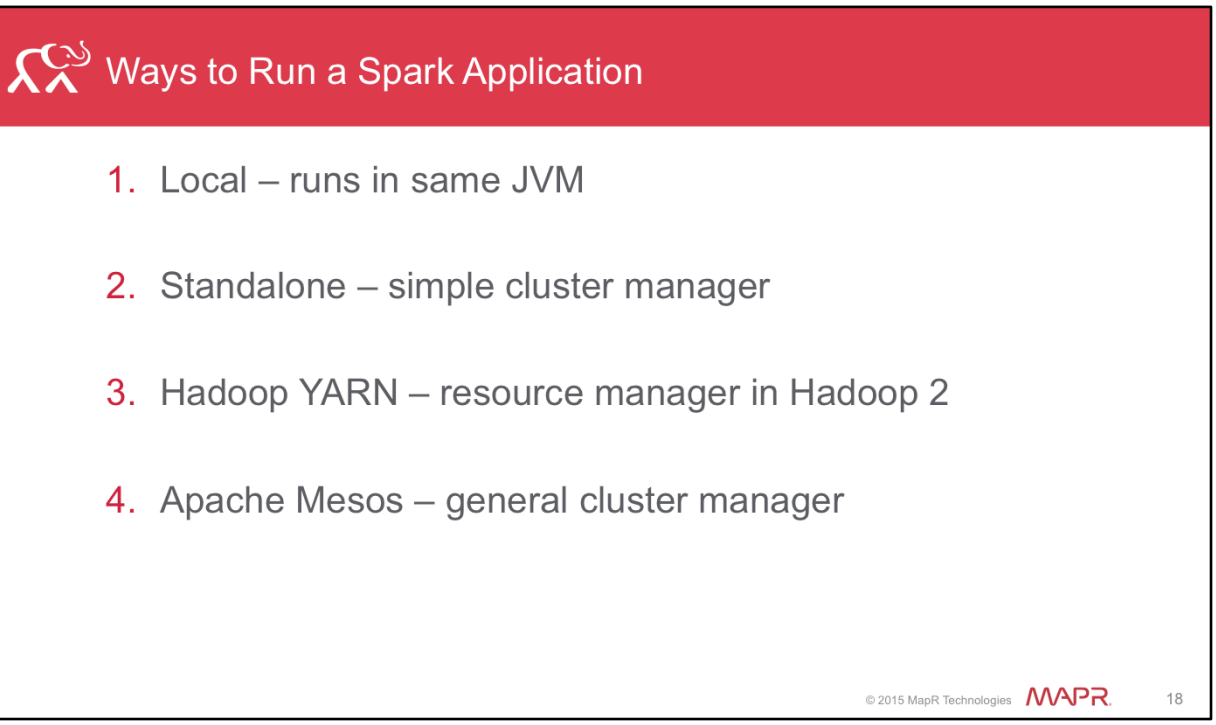


Spark uses a master-slave architecture where there is one central coordinator, the Driver, and many distributed workers called executors. The Driver runs in its own Java process. Each executor runs in its own, individual Java process.

The driver, together with its executors, are referred to as a Spark application.

To run a Spark application

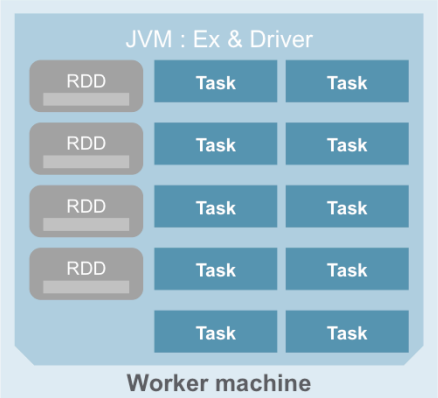
1. First, the user submits an application using spark-submit
2. spark-submit launches the driver program, which invokes the main() method. The main() method creates SparkContext which tells the driver the location of the cluster manager.
3. The driver contacts the cluster manager for resources, and to launch executors
4. Cluster manager will launch executors for the driver program
5. Driver runs through the application (RDDs, Transformations and Actions) sending work to the executors in the form of tasks



We have seen what occurs when you run a Spark application on a cluster. Spark uses the cluster manager to launch executors. The cluster manager is a pluggable component in Spark, which is why Spark can run on various external managers such as Hadoop YARN, Apache Mesos. Spark also has its own built-in standalone cluster manager.

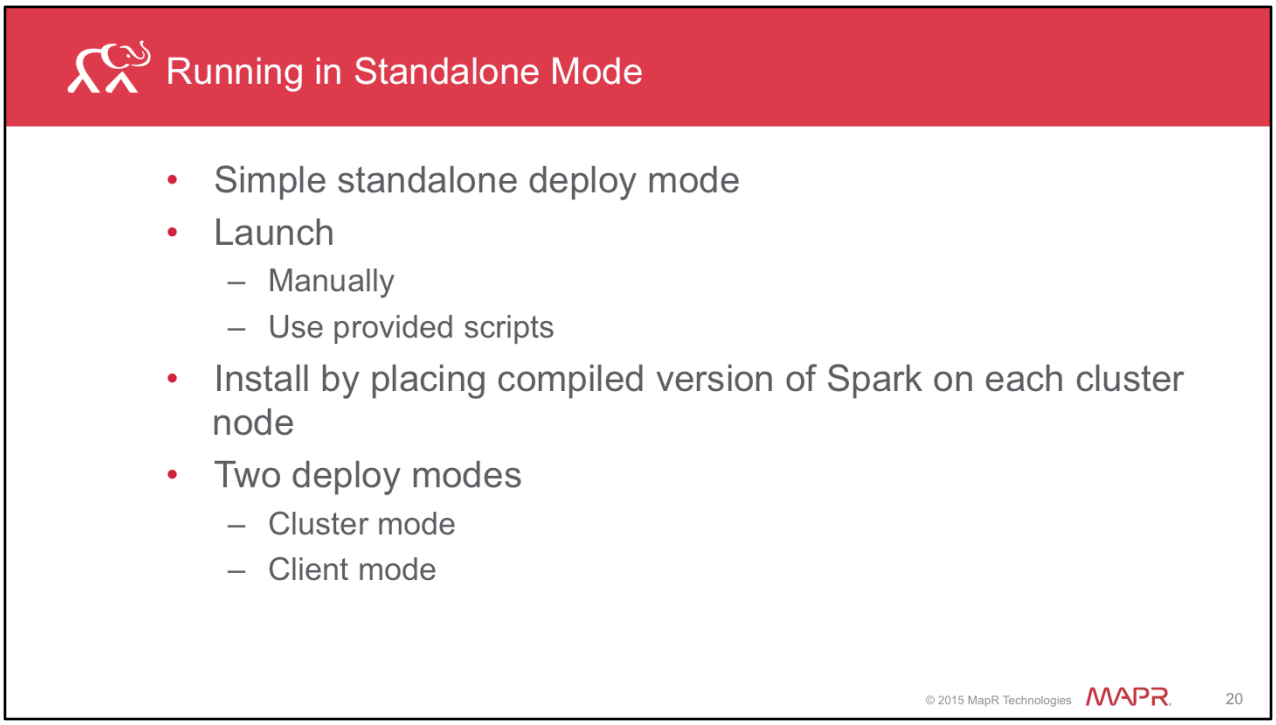
Running in Local Mode

- Driver program & workers in same JVM
- RDDs & variables in same memory space
- No central master
- Execution started by user
- Good for prototyping, testing



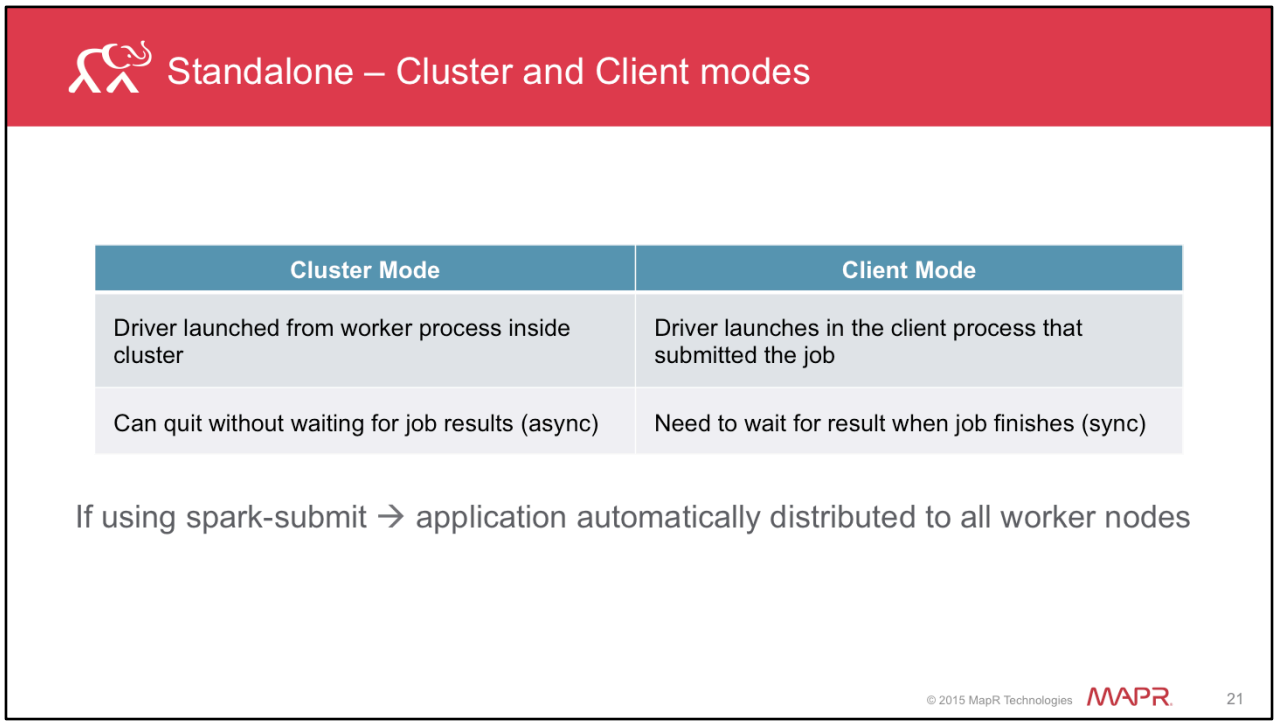
 19

In the local mode, there is only one JVM. The driver program and the workers are in the same JVM.
Within the program, any RDDs, variables that are created are in the the same memory space. There is no central master in this case and the execution is started by the user.
The local mode is useful for prototyping, development, debugging and testing.

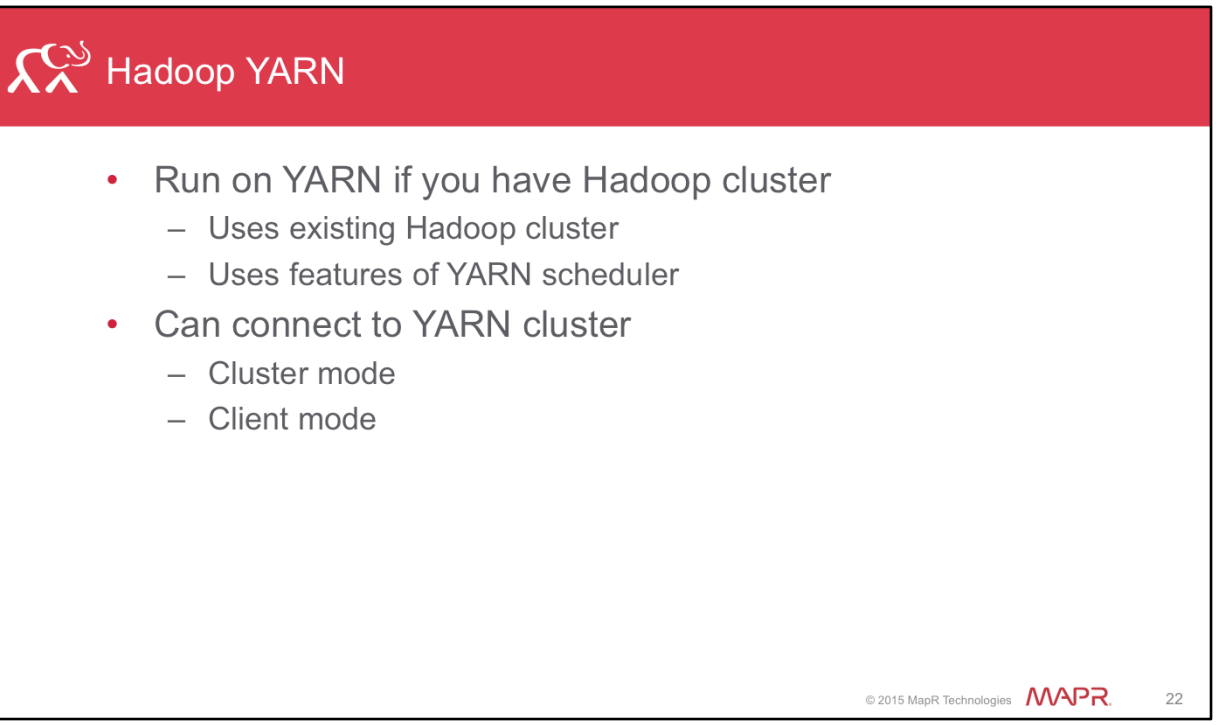


Spark provides a simple standalone deploy mode. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use launch scripts provided by Apache Spark. It is also possible to run these daemons on a single machine for testing.
To install Spark Standalone mode, you simply place a compiled version of Spark on each node on the cluster. For standalone clusters, Spark currently supports two deploy modes.

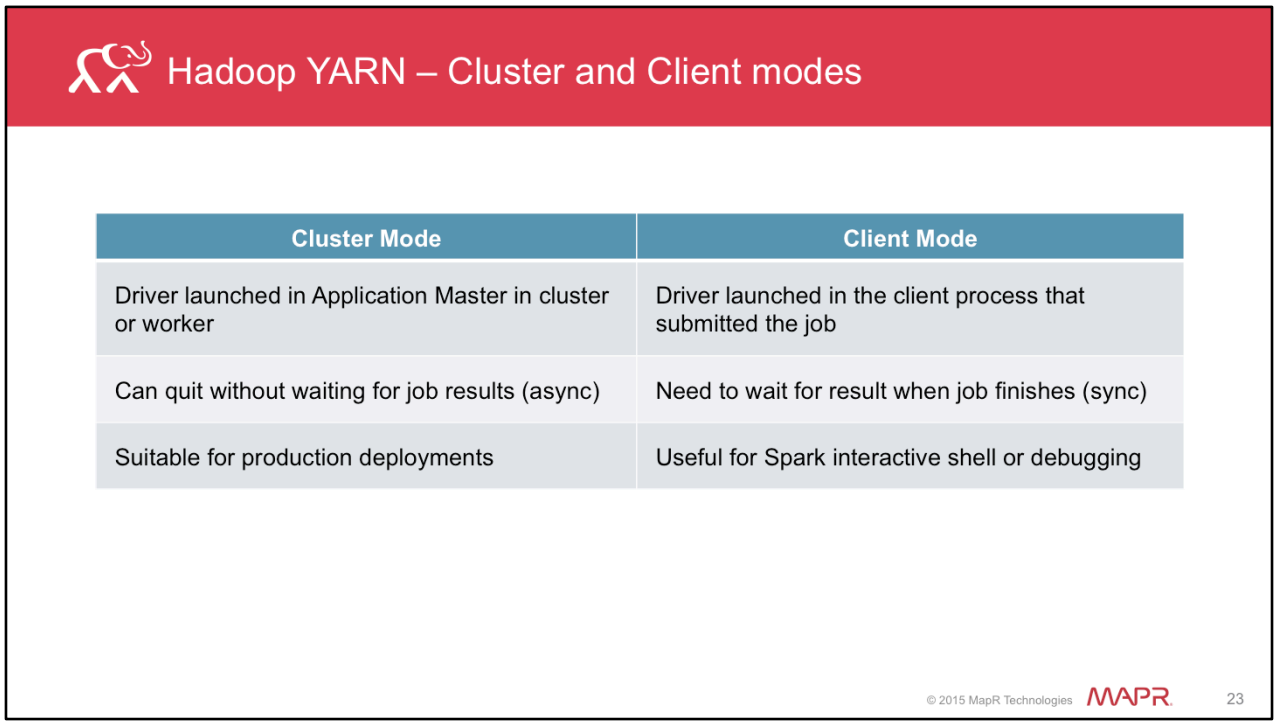
Spark will only run on those nodes on which Spark is installed.
To run an application on the Spark cluster, simply pass the spark://IP:PORT URL of the master as to the SparkContext constructor.



In cluster mode, however, the driver is launched from one of the Worker processes inside the cluster, and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish. In client mode, the driver is launched in the same process as the client that submits the application.
If your application is launched through Spark submit, then the application jar is automatically distributed to all worker nodes.



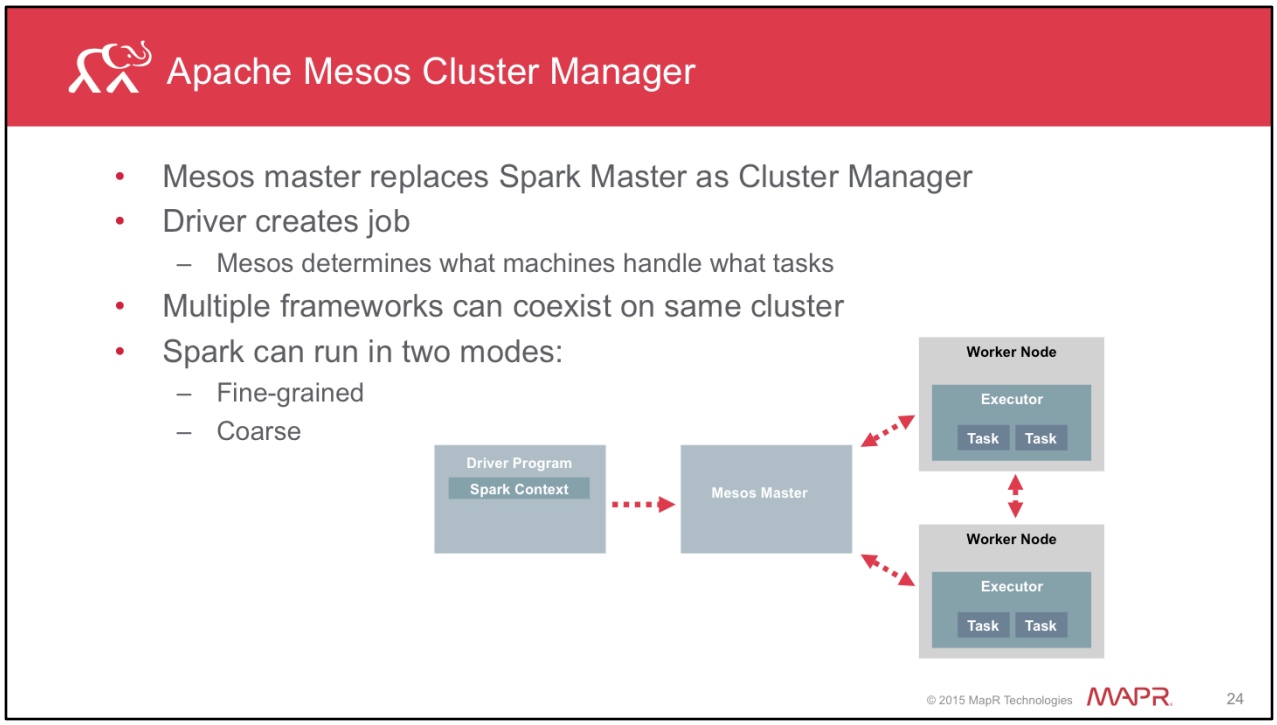
It is advantageous to run Spark on YARN if you have an existing Hadoop cluster. You can use the same Hadoop cluster without having to maintain a separate cluster. In addition, you can take advantage of he features of the YARN scheduler for categorizing, isolating and prioritizing workloads. There are two deploy modes that can be used to launch Spark applications on YARN. – cluster mode and client mode. Let us take a look at these next.



Running in YARN cluster mode is an async process – you can quit without waiting for the job results. On the other hand, running in YARN client mode is a sync process – you have to wait for the result when the job finishes.

The YARN cluster mode is suitable for long running jobs i.e. for production deployments. The YARN client mode is useful when using the interactive shell, for debugging and testing.

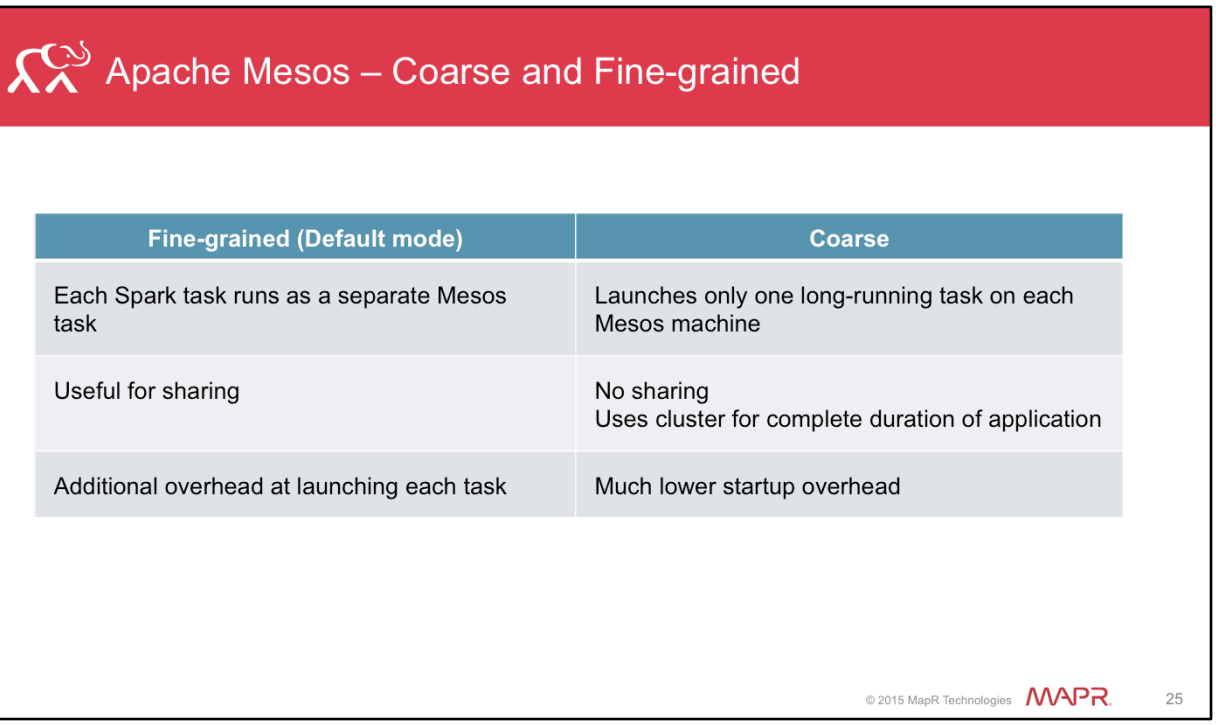
There are two deploy modes that can be used to launch Spark applications on YARN. In yarn-cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster. In yarn-client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.



When using Mesos, the Mesos master replaces the Spark master as the cluster manager.

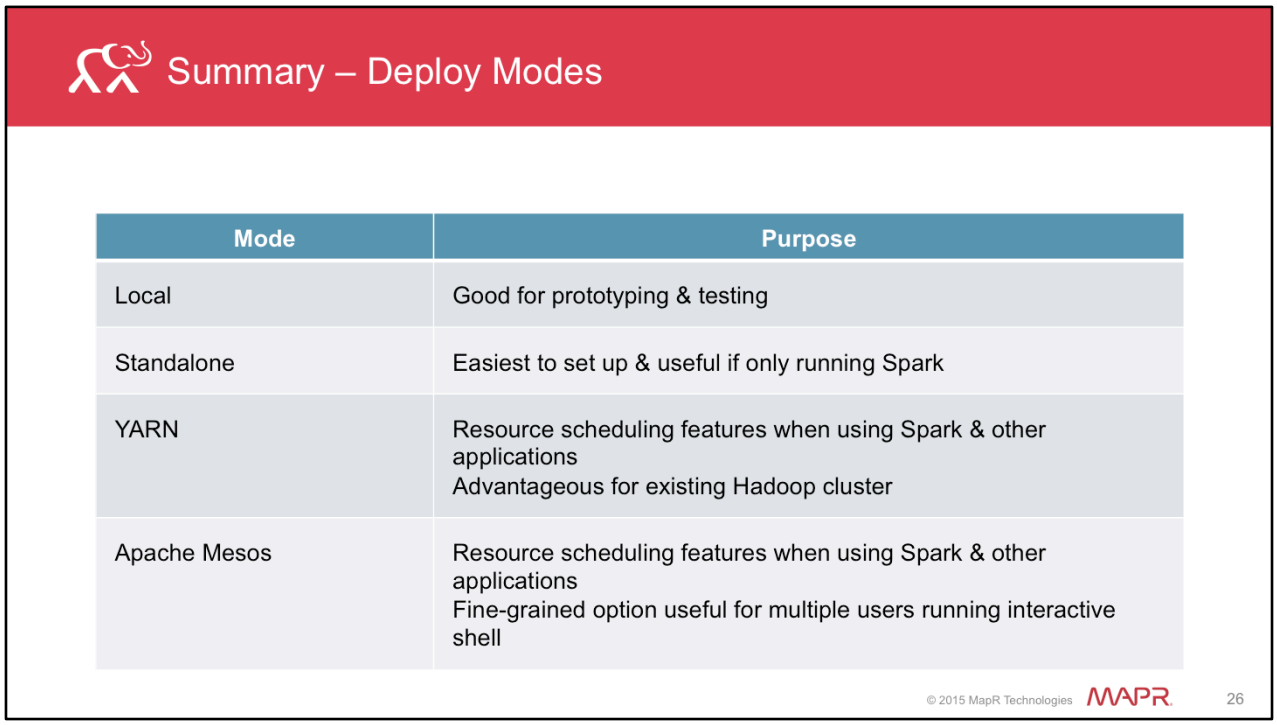
Now when a driver creates a job and starts issuing tasks for scheduling, Mesos determines what machines handle what tasks. Because it takes into account other frameworks when scheduling these many short-lived tasks, multiple frameworks can coexist on the same cluster without resorting to a static partitioning of resources.

Spark can run over Mesos in two modes: “fine-grained” (default) and “coarse-grained”.

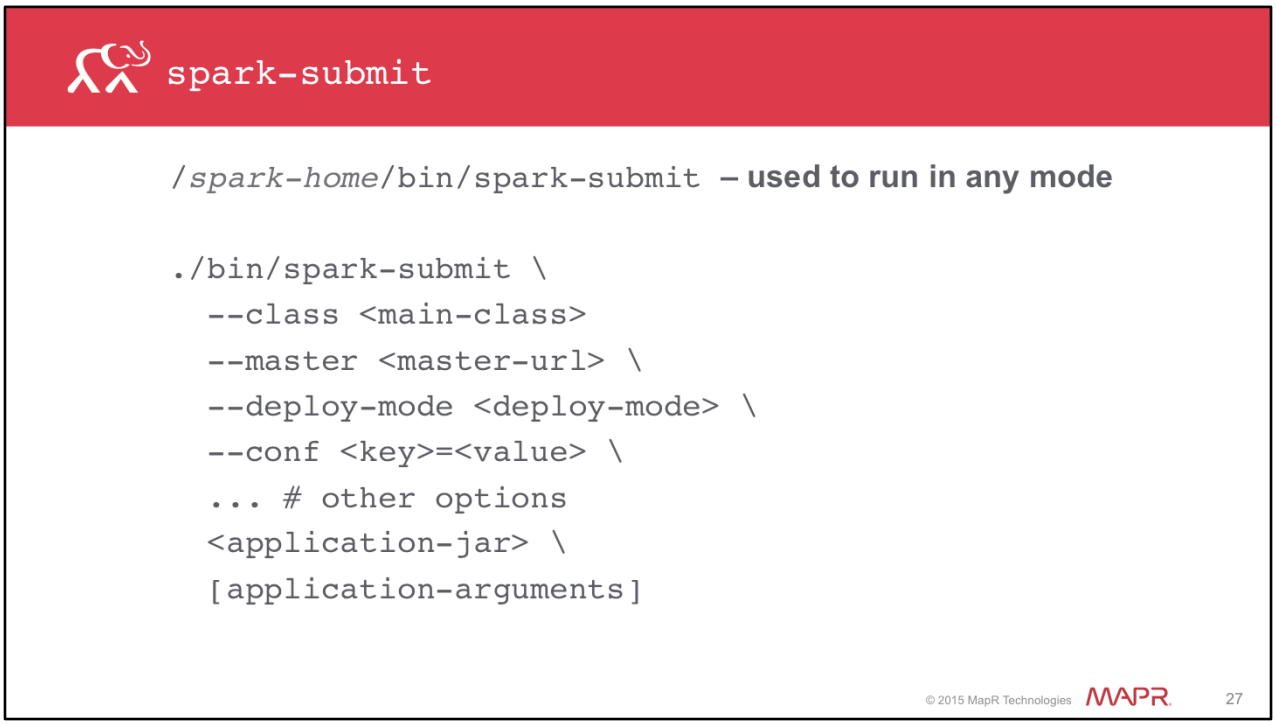


In “fine-grained” mode (default), each Spark task runs as a separate Mesos task. This allows multiple instances of Spark (and other frameworks) to share machines at a very fine granularity, where each application gets more or fewer machines as it ramps up and down, but it comes with an additional overhead in launching each task. This mode may be inappropriate for low-latency requirements like interactive queries or serving web requests.

The “coarse-grained” mode will instead launch only *one* long-running Spark task on each Mesos machine, and dynamically schedule its own “mini-tasks” within it. The benefit is much lower startup overhead, but at the cost of reserving the Mesos resources for the complete duration of the application.



Here is a summary of the different deploy modes. The local mode is useful for prototyping, development, debugging and testing.
The standalone mode is the easiest to set up and will provide almost all of the functionality of the other cluster managers if you are only running Spark.
YARN and Mesos both provide rich resources scheduling if you want to run Spark with other applications. However, YARN is advantageous if you have an existing Hadoop cluster as you can then use the same cluster for Spark. You can also use the the features of the YARN scheduler. The advantage of Apache Mesos is the fine grained sharing option that allows multiple users to run the interactive shell.



The spark-submit script in Spark home/bin director is used to run an application in any mode.
We can specify different options when calling spark-submit.
-- class→the entry point for your application – the main class
-- master → the Master URL for the cluster
-- deploy-mode (whether to deply the driver on the worker nodes (cluster) or locally
-- conf- arbitrary Spark configuration property in key=value format
application jar – path to the bundled jar including all dependencie. The URL must be globally visible inside the cluster i.e. an hdfs://path or a file:// path present on all nodes.
application arguments – arguments passed to the main method of the main class
Depending on the mode in which you are deploying, you may have other options.

spark-submit - Examples

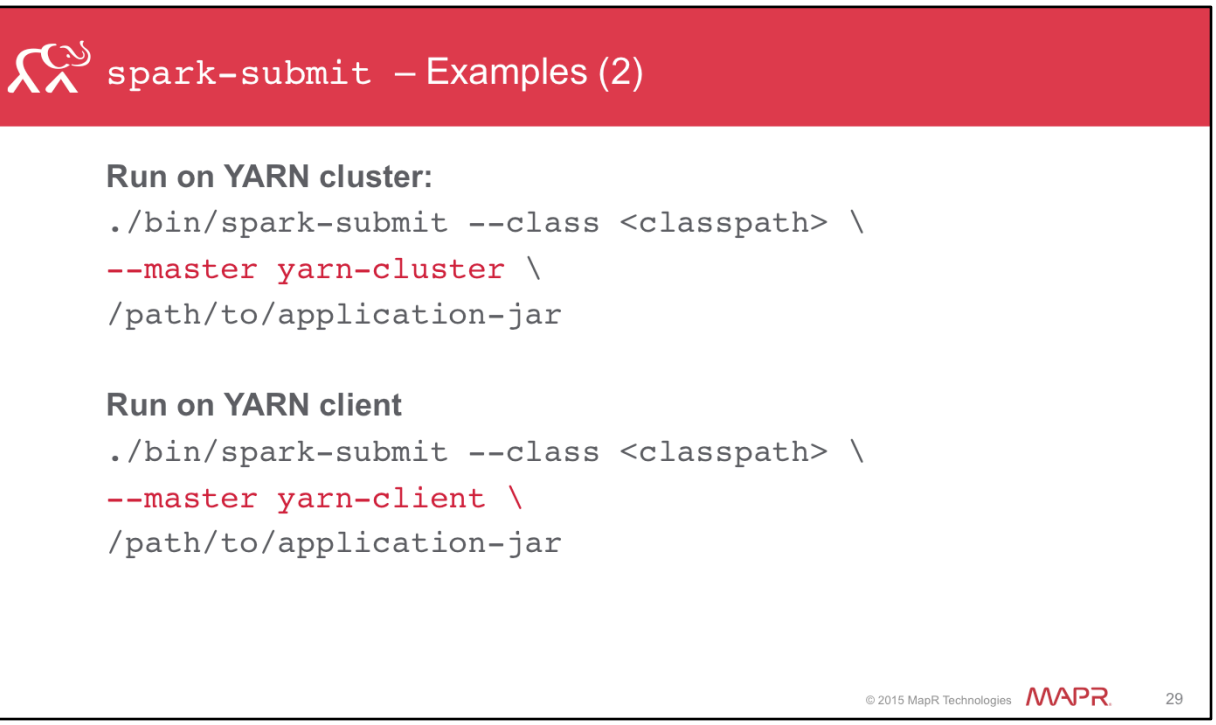
To run local mode on n cores:

```
./bin/spark-submit --class <classpath> \
--master local[n] \
/path/to/application-jar
```

To run standalone client mode :

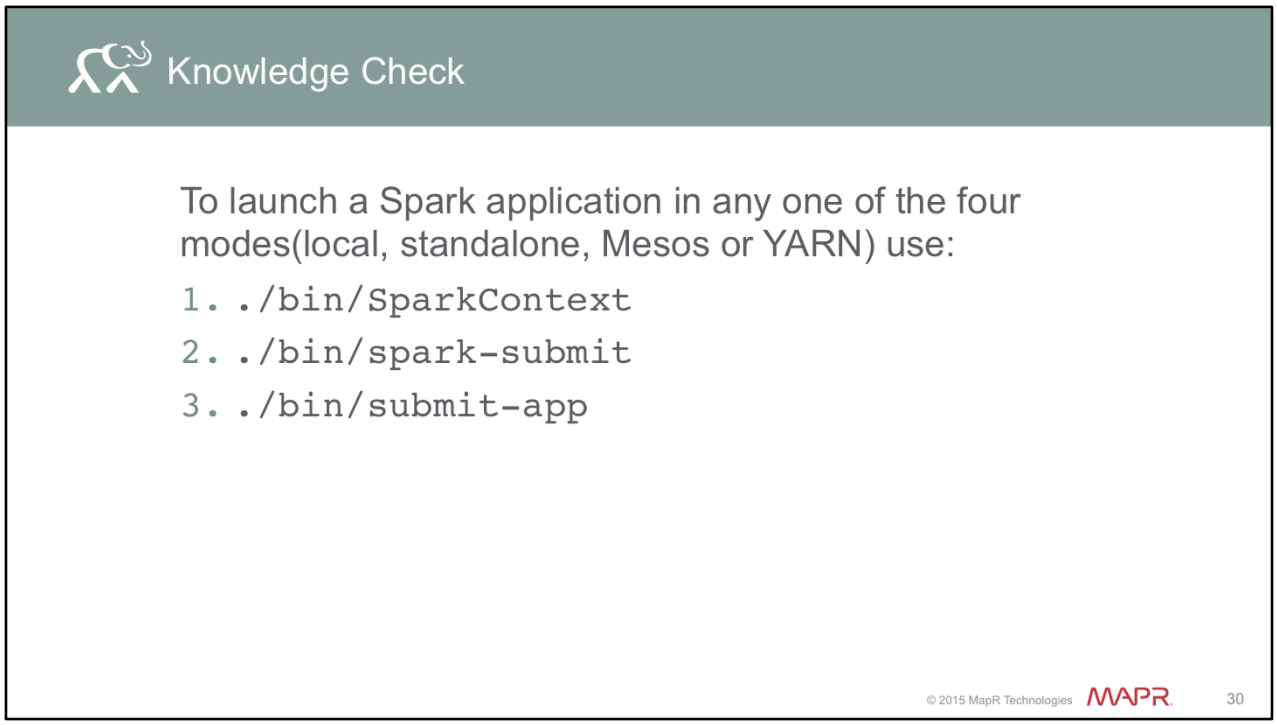
```
./bin/spark-submit --class <classpath>\
-- master spark:<master url> \
/path/to/application-jar
```


Here is an example to run in local mode on n cores.
The second example shows you how to run in the Standalone client mode.

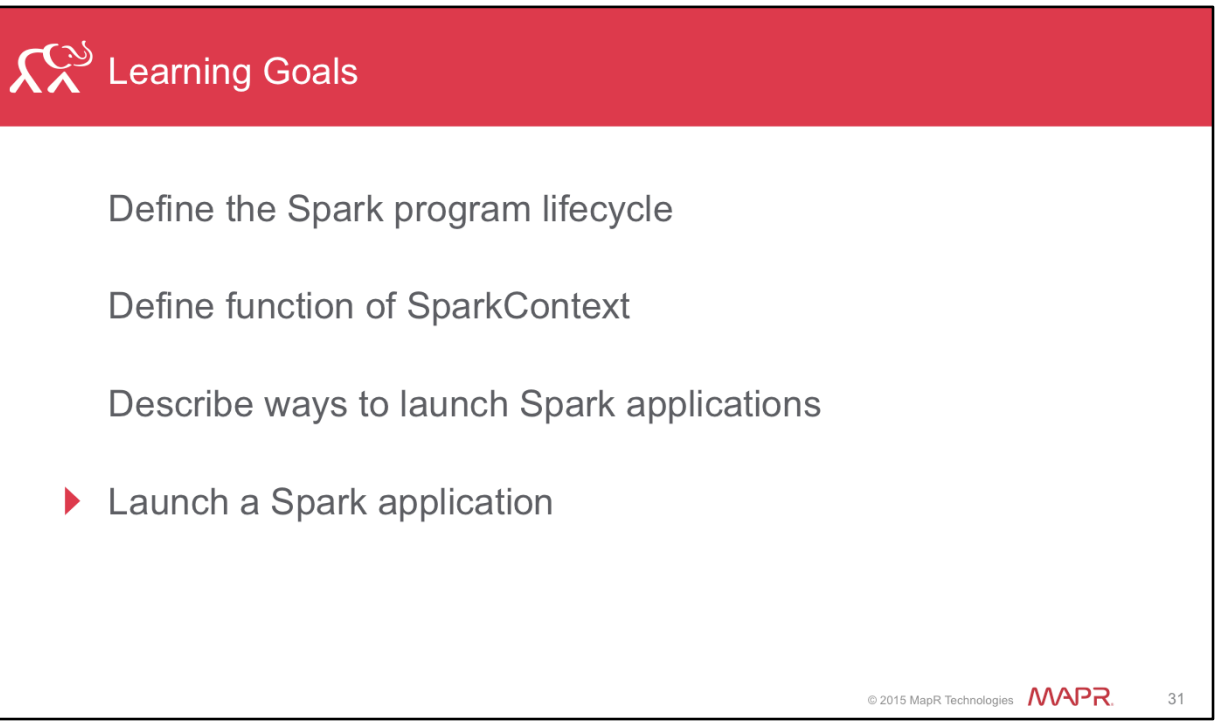


Unlike in Spark standalone and Mesos mode, in which the master's address is specified in the "master" parameter, in YARN mode the ResourceManager's address is picked up from the Hadoop configuration. Thus, the master parameter is simply "yarn-client" or "yarn-cluster".
To launch a Spark application in yarn-cluster mode:
./bin/spark-submit --class path.to.your.Class --master yarn-cluster [options] <app jar> [app options]
To launch a Spark application in yarn-client mode, do the same, but replace "yarn-cluster" with "yarn-client". To run spark-shell:
$ ./bin/spark-shell --master yarn-client

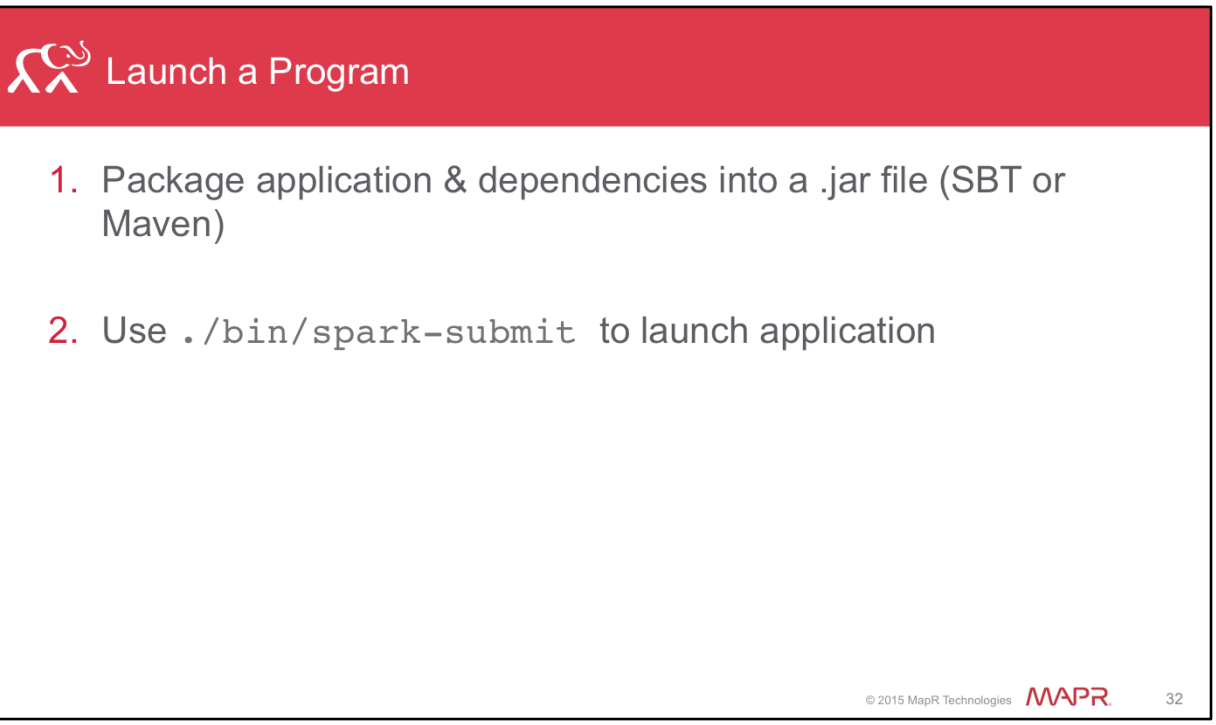
We will see next an example of how to run our application.


Knowledge Check
To launch a Spark application in any one of the four modes(local, standalone, Mesos or YARN) use:
1. ./bin/SparkContext
2. ./bin/spark-submit
3. ./bin/submit-app
© 2015 MapR Technologies
30

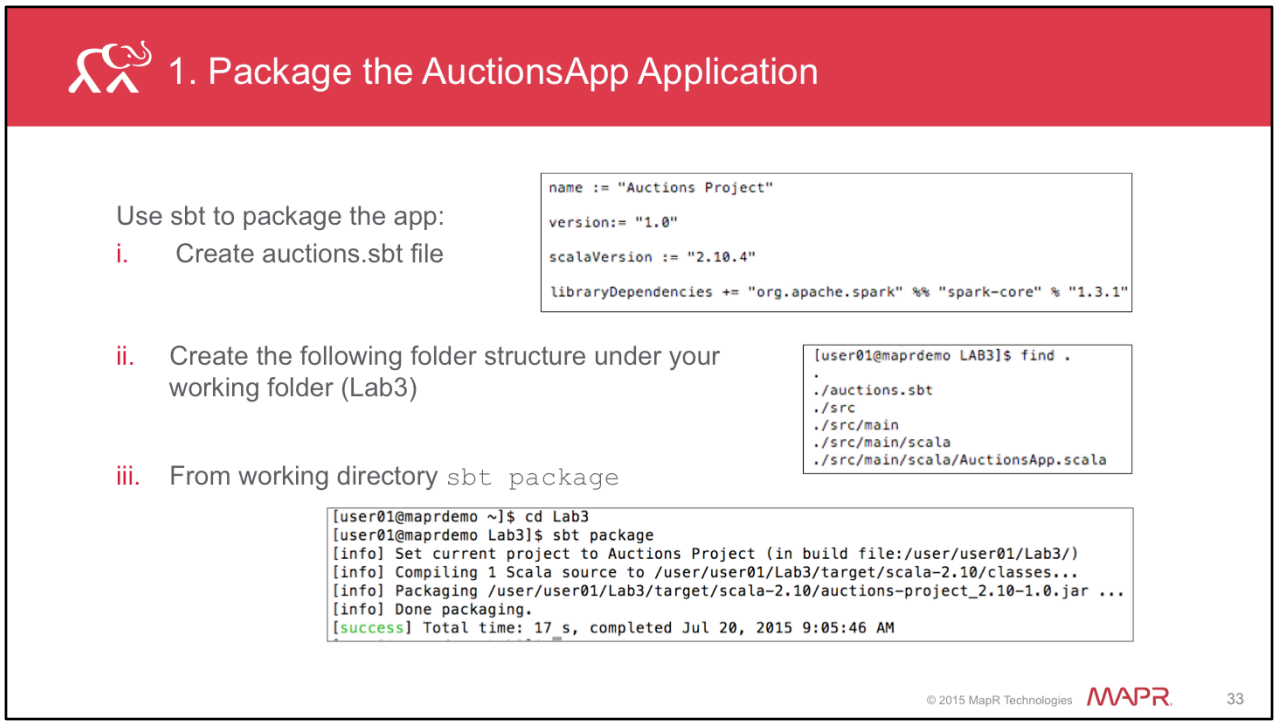
Answer: 2



We will now launch our Spark application.



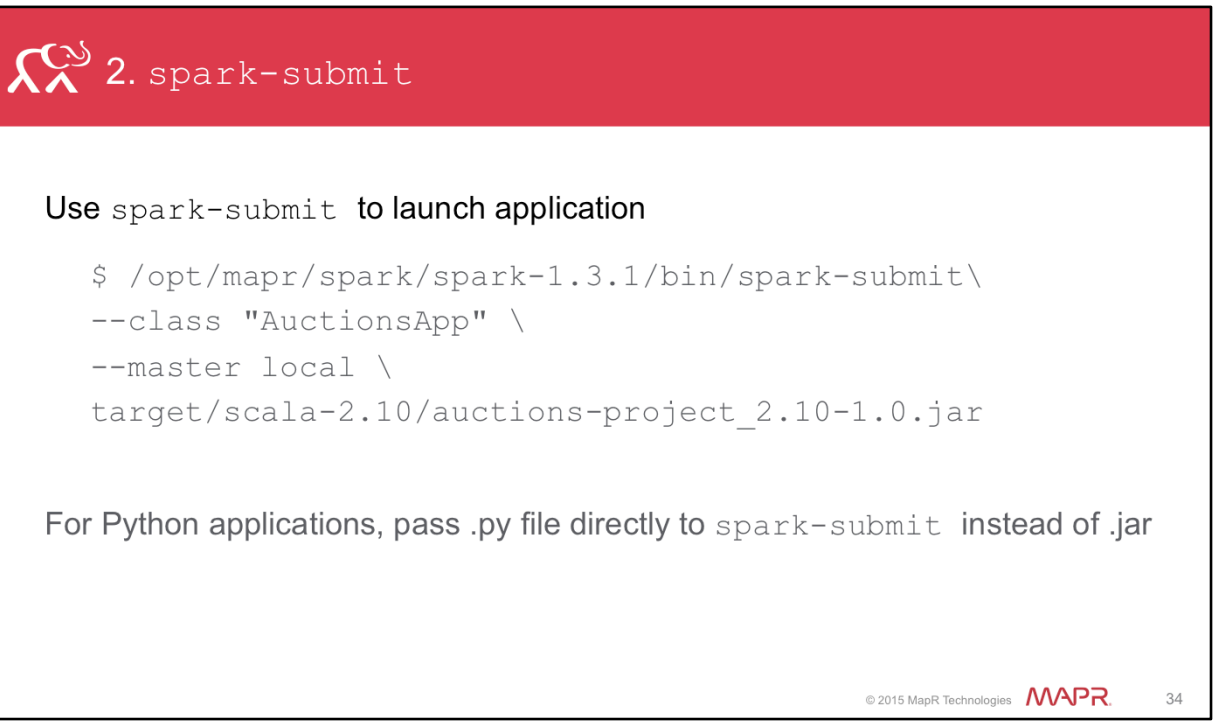
Package your application and any dependencies into a .jar file. For Scala apps, you can use Maven or SBT.



In this example, we package the Scala application using sbt. You can also use maven. In the lab, you can use either method to package the application.
Use vi or your favorite editor to create the file auctions.sbt containing the lines shown here. You specify the project name, version, the Scala version and the library dependencies – spark core version.

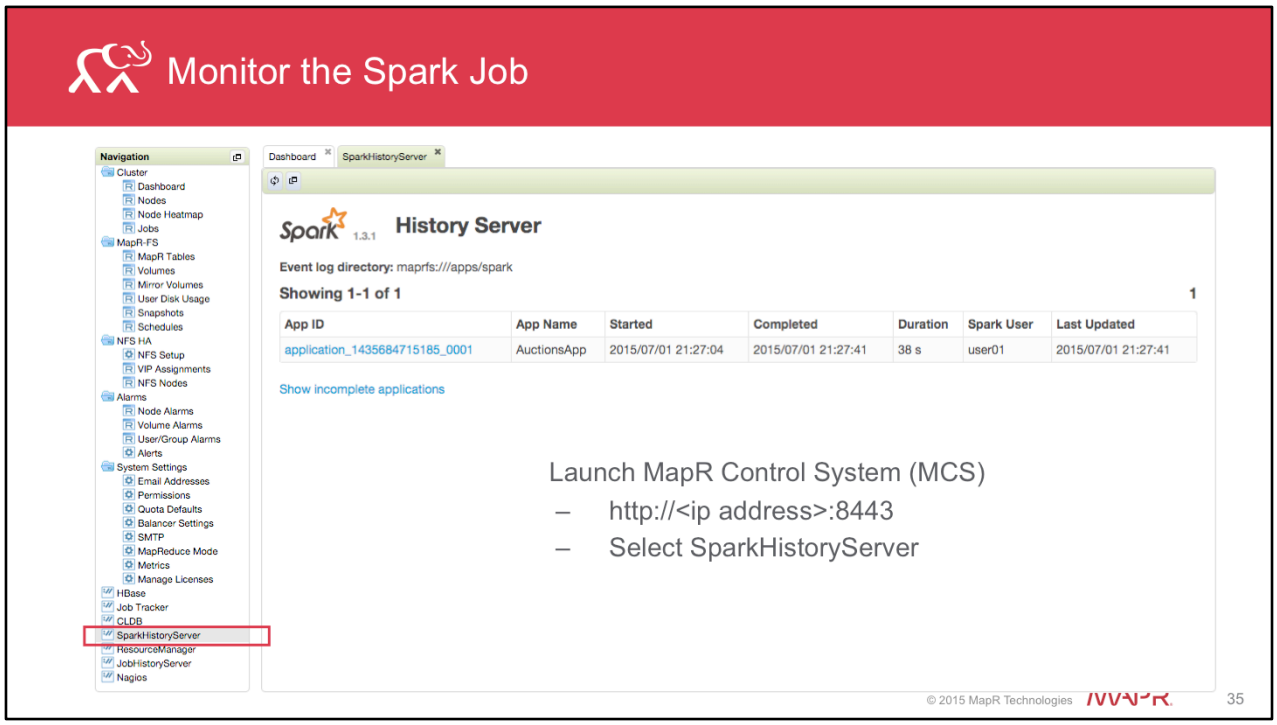
From the working directory, for example /user/user01/LAB3, run sbt package.

Note that you need to have sbt installed.

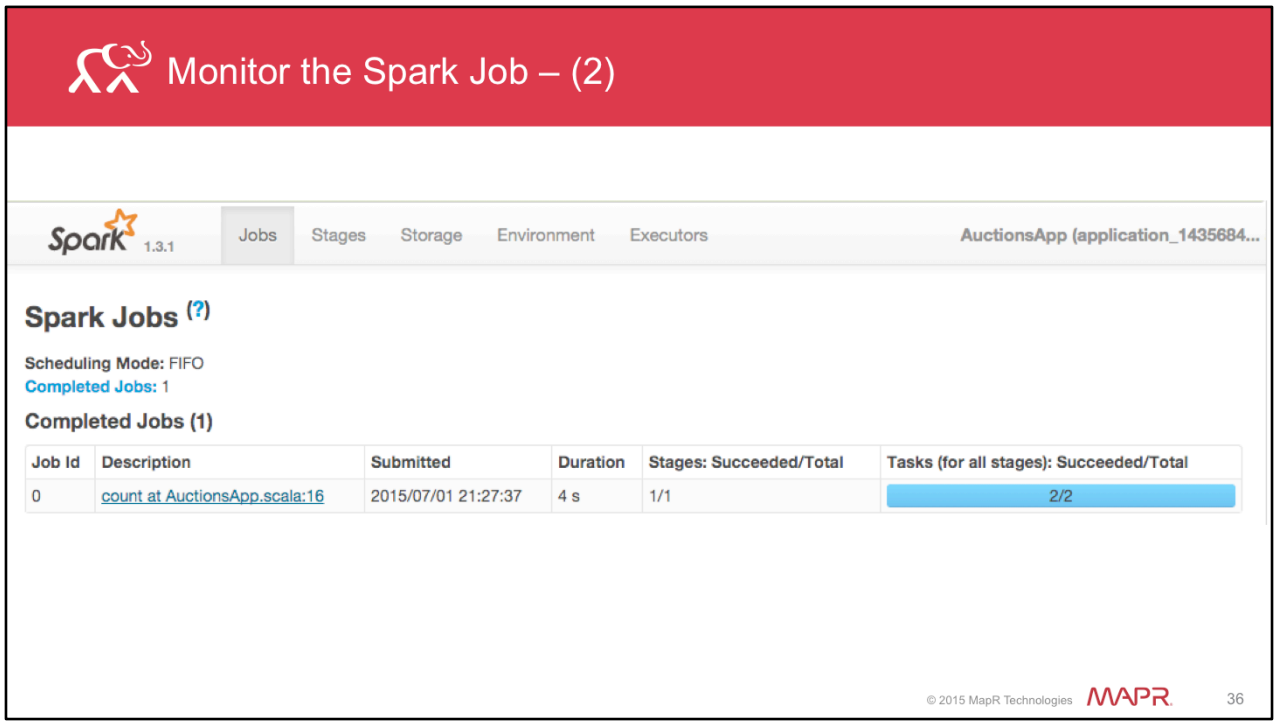


Once you have packaged your application, you should have the jar file.
You can now launch the application using spark-submit.
We are using local mode here. In the real world after testing, you would deploy to cluster mode. For example, to run on yarn-cluster, you would just change the master option to yarn-cluster.

For python applications, pass the .py file directly to spark-submit.



You can monitor the Spark application using the MapR Control System (MCS). To launch the MCS, open a browser and navigate to the following URL: http://<ip address>:8443. Select SparkHistoryServer from the left navigation pane.



You can drill down the application name and keep drilling to get various metrics. Monitoring is covered in more detail in a later course.

Knowledge Check

Select the most appropriate command to launch your Scala application, “IncidentsApp” on a YARN cluster, where the application with its dependencies is packaged to “/path/to/file/incidentsapp.jar” :

1. `./bin/spark-submit --class IncidentsApp --master cluster / path/to/file/incidentsapp.jar`
2. `./bin/spark-submit --class IncidentsApp spark:// 100.10.60.120:7077 /path/to/file/incidentsapp.jar`
3. `./bin/spark-submit --class IncidentsApp --master yarn-cluster /path/to/file/incidentsapp.jar`

3

In this lab, you will run the standalone Spark application. You can use sbt or maven to package the application. Directions for both are provided.

Note that you can also create the application in Python.

Congratulations! You have completed Lesson 3 and this course –DEV 360 –Spark Essentials. Visit the MapR Academy for more courses or go to doc.mapr.com for more information on Hadoop and Spark and other ecosystem components.