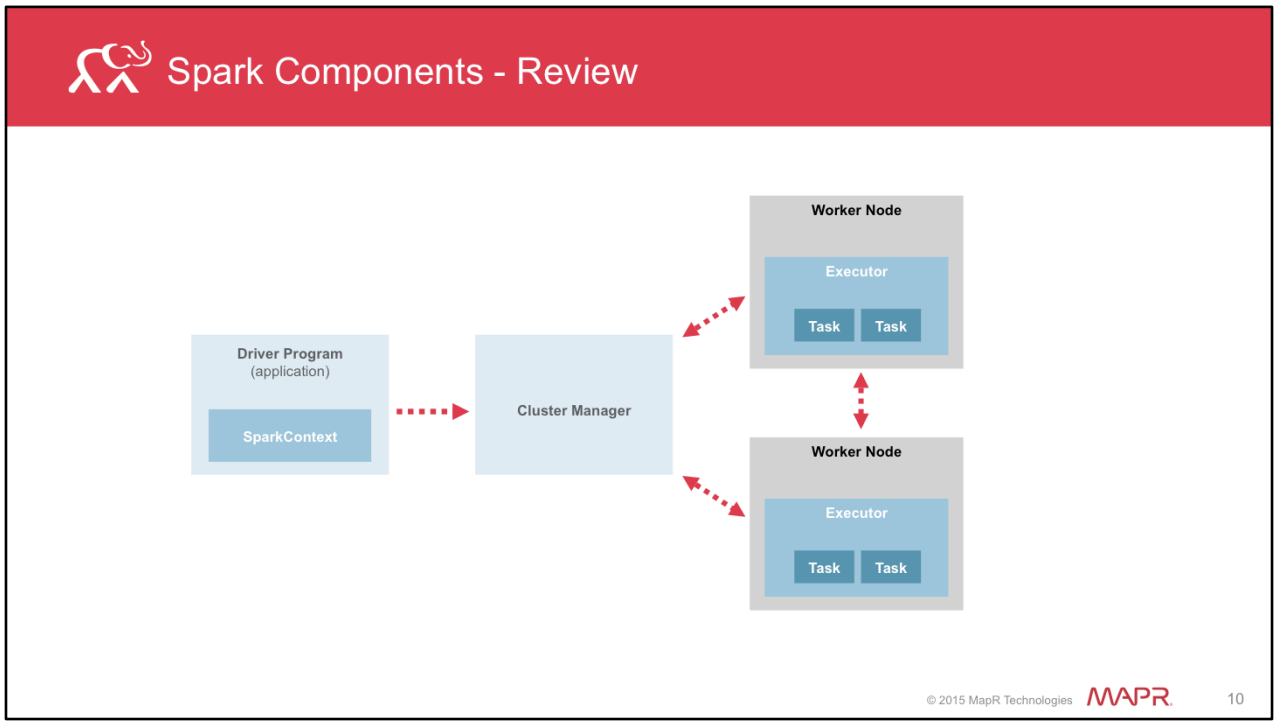
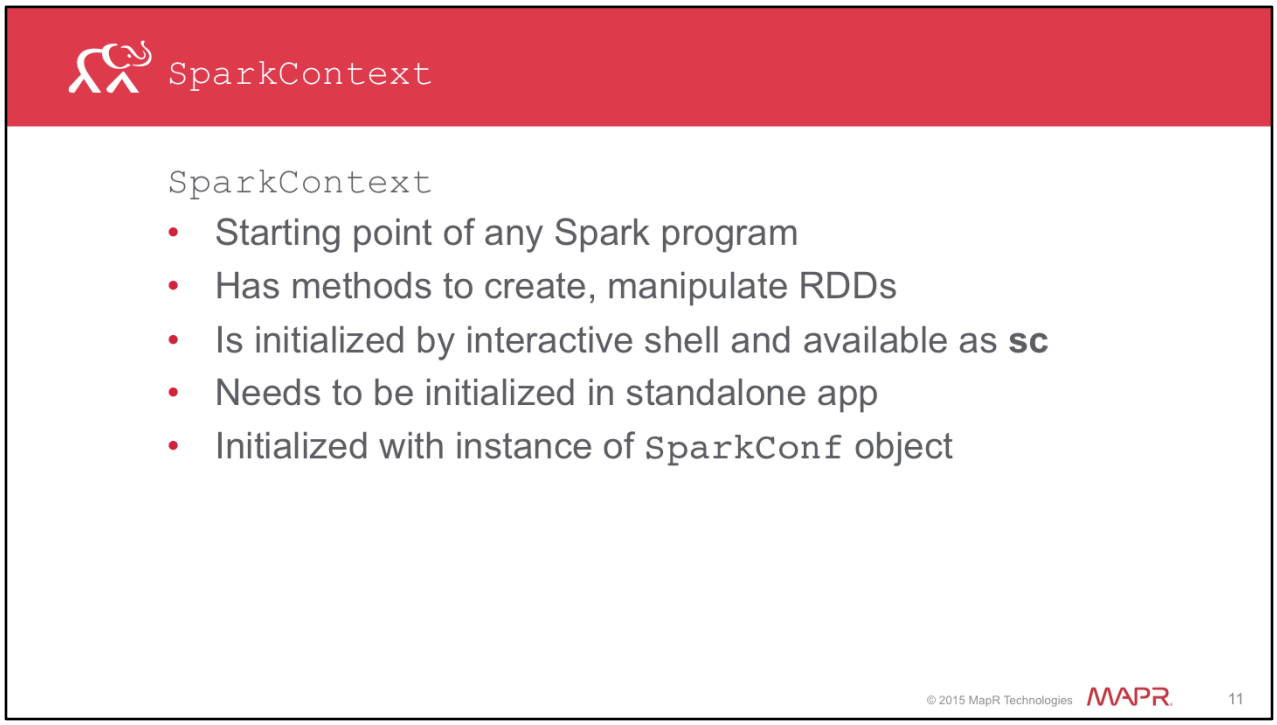


This is a review of Spark components. A Spark cluster consists of two processes, a driver program and multiple workers nodes each running an executor process.

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.

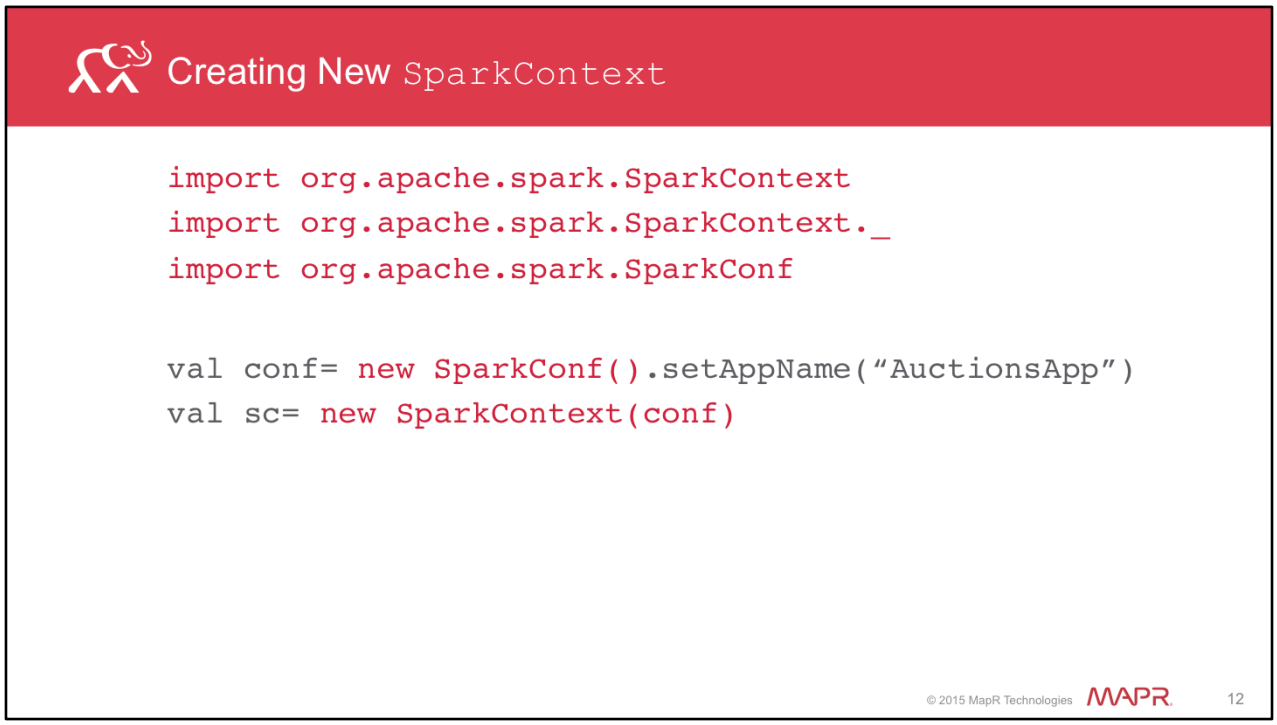


Looking at the Spark Components, SparkContext is the starting point of any Spark program. It tells Spark how and where to access the cluster.

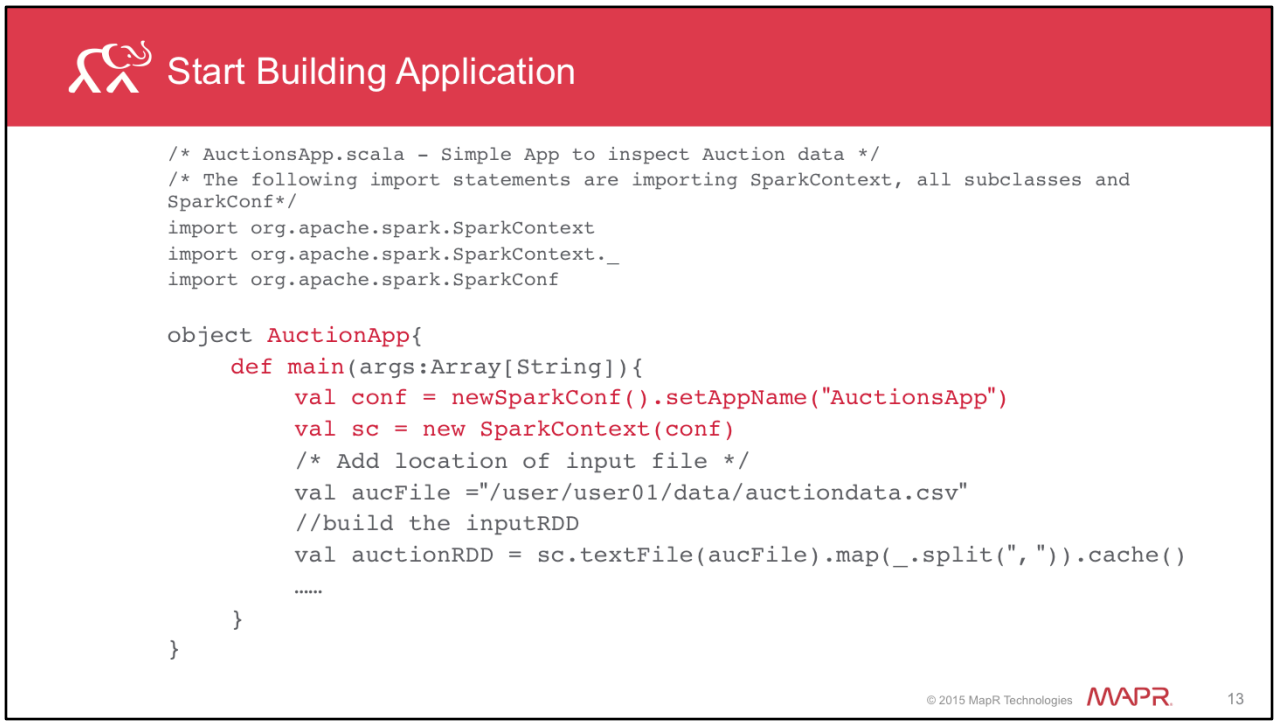
To create and manipulate RDDs, we use various methods in SparkContext.

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.



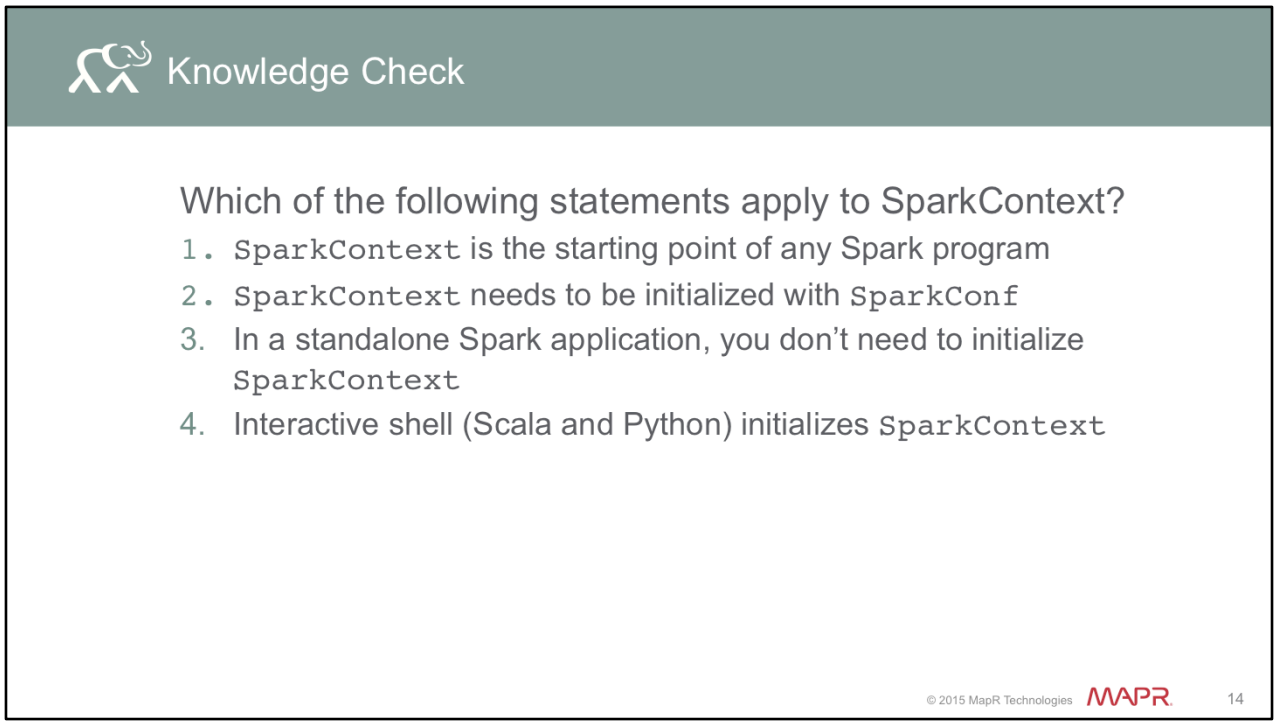
The next step is to define the class (note that this is the same name as the file). The application should define a main() method instead of extending scala.App

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.

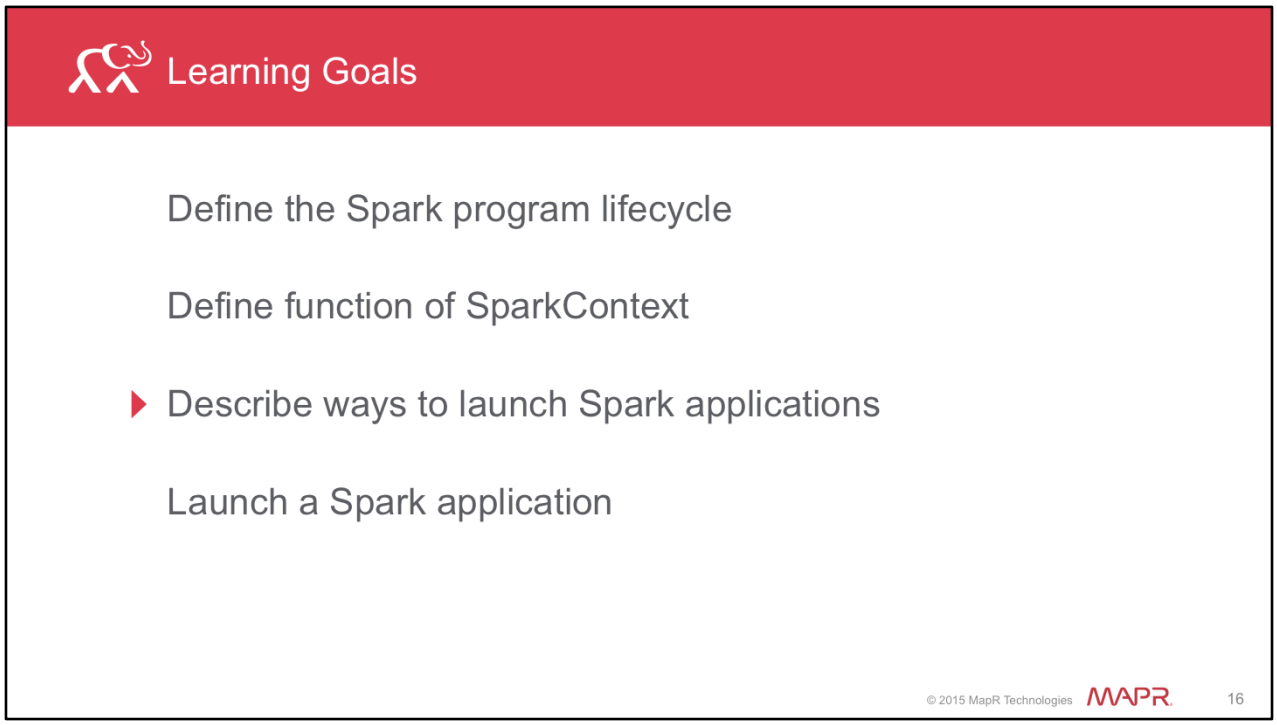
You can that more code to this file for example, to inspect the detail and print the results to the console.

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

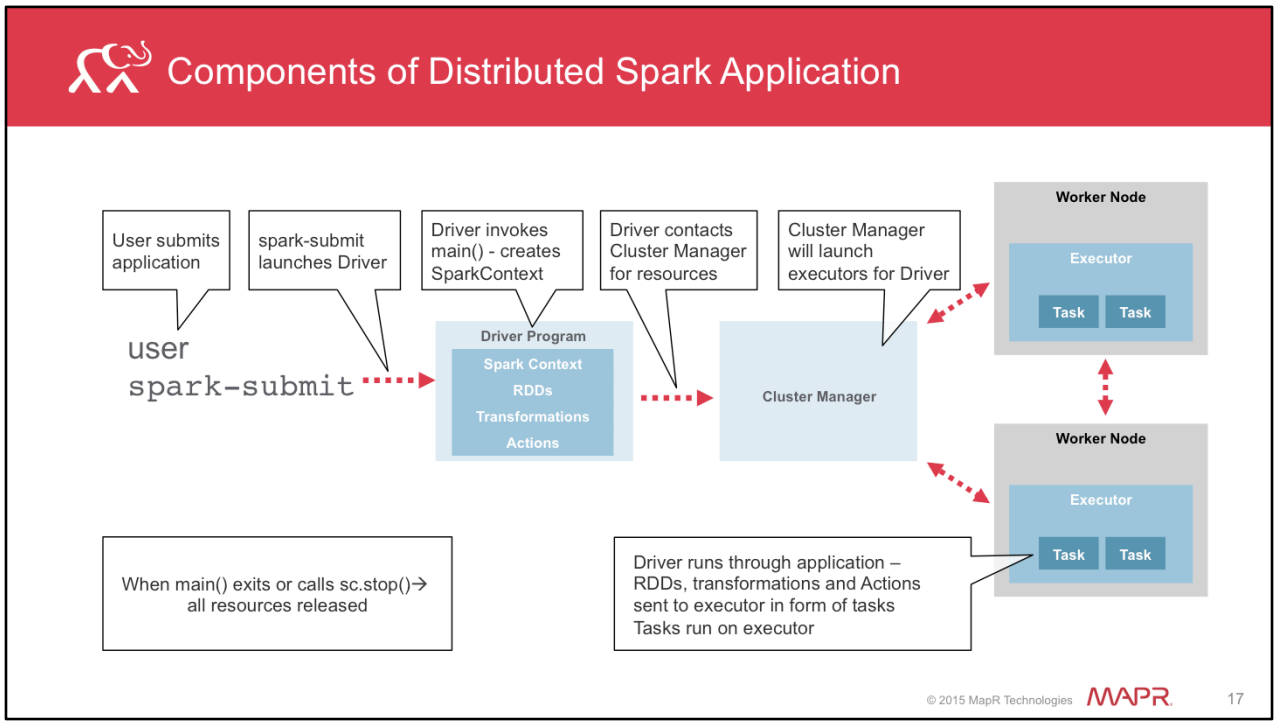

Knowledge Check
Which of the following statements apply to SparkContext?
1. SparkContext is the starting point of any Spark program
2. SparkContext needs to be initialized with SparkConf
3. In a standalone Spark application, you don't need to initialize SparkContext
4. Interactive shell (Scala and Python)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

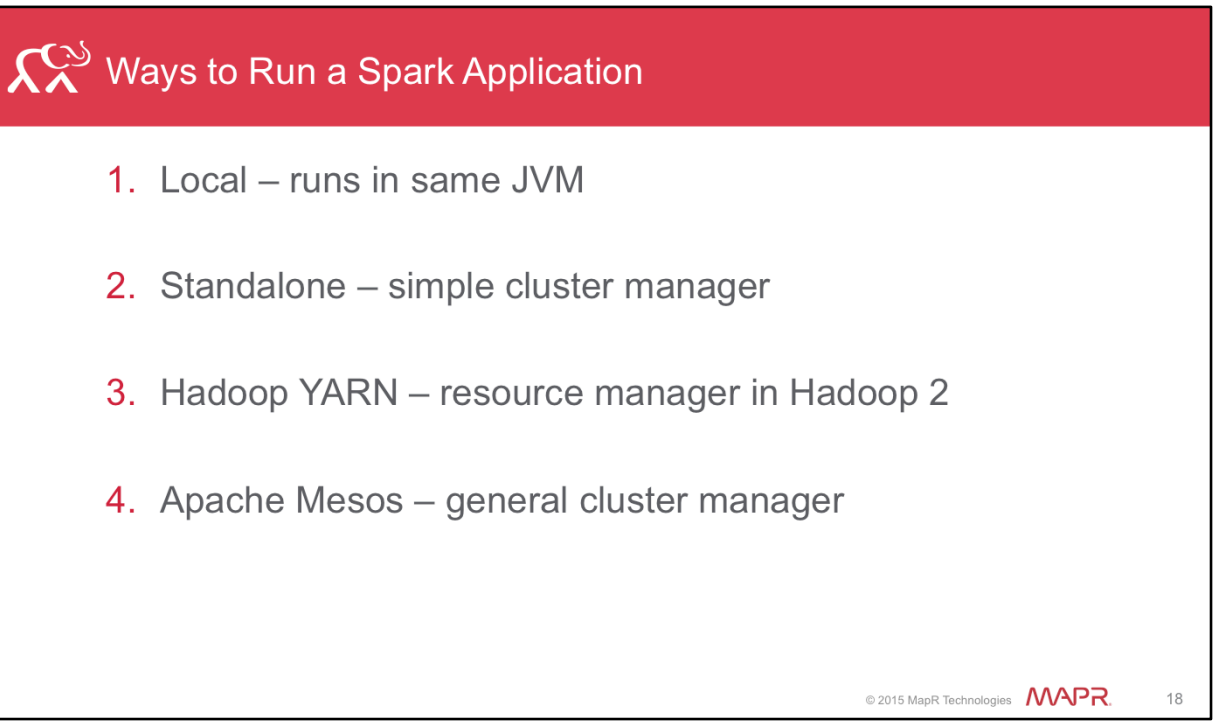


Spark uses a master-slave architecture where there is one central coordinator, the Driver, and many distributed workers called executors. The Driver runs in its own Java process. Each executor runs in its own, individual Java process.

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

To run a Spark application

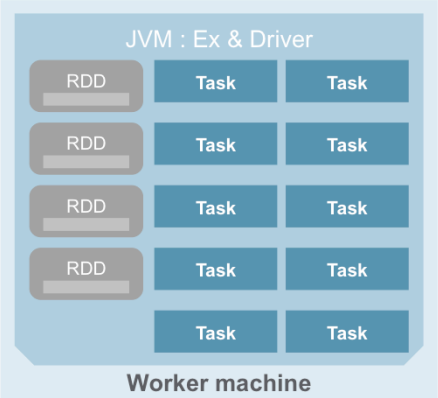
1. First, the user submits an application using spark-submit
2. spark-submit launches the driver program, which invokes the main() method. The main() method creates SparkContext which tells the driver the location of the cluster manager.
3. The driver contacts the cluster manager for resources, and to 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.

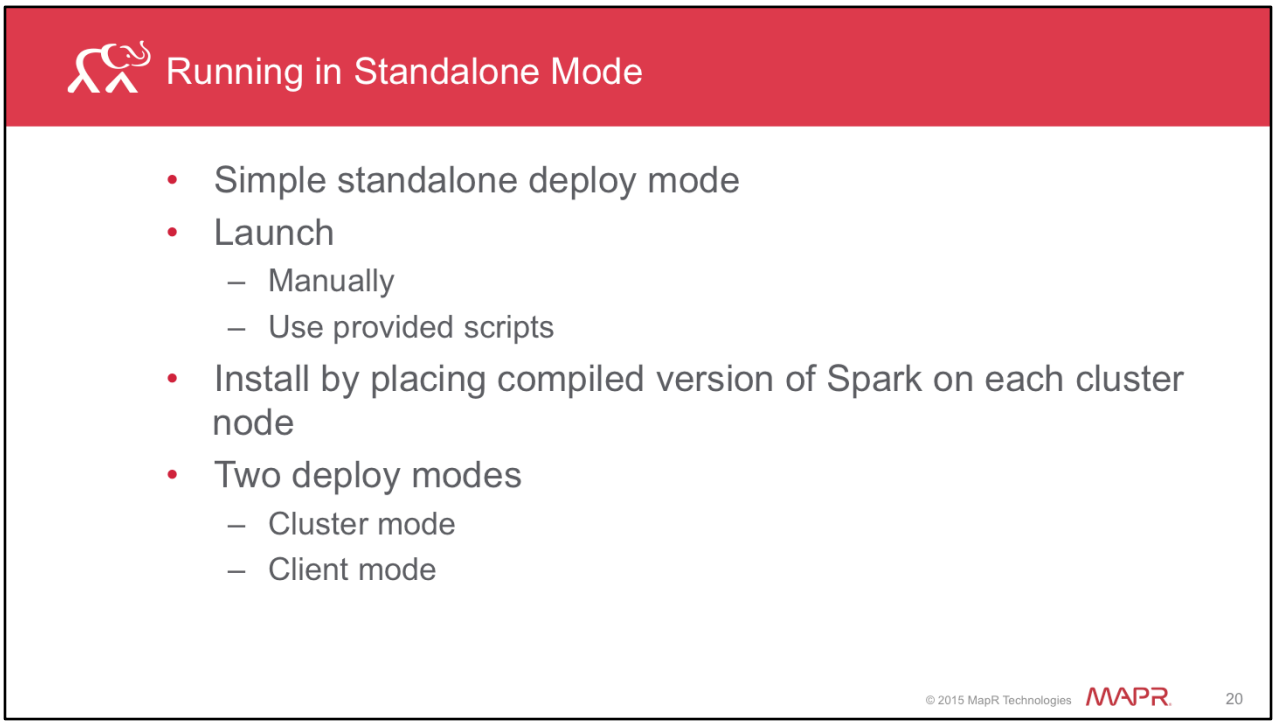
Running in Local Mode

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



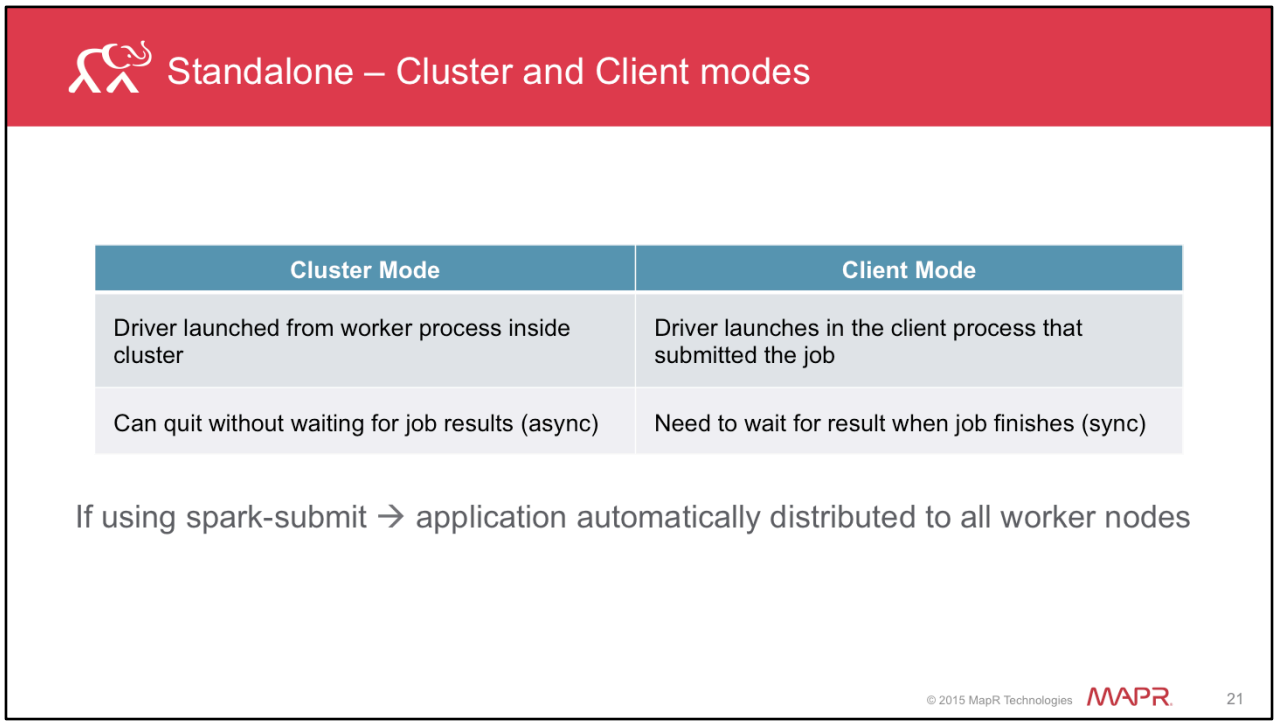
 19

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.

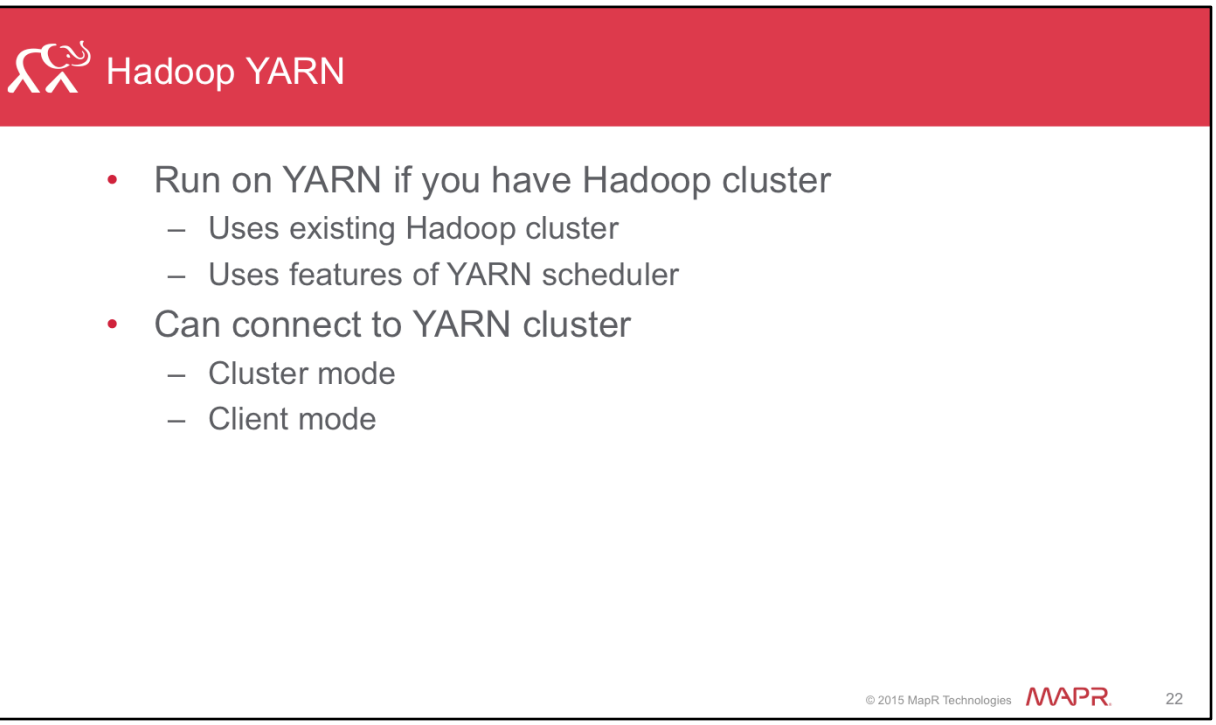


Spark provides a simple standalone deploy mode. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use launch scripts provided by Apache Spark. It is also possible to run these daemons on a single machine for testing.
To install Spark Standalone mode, you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

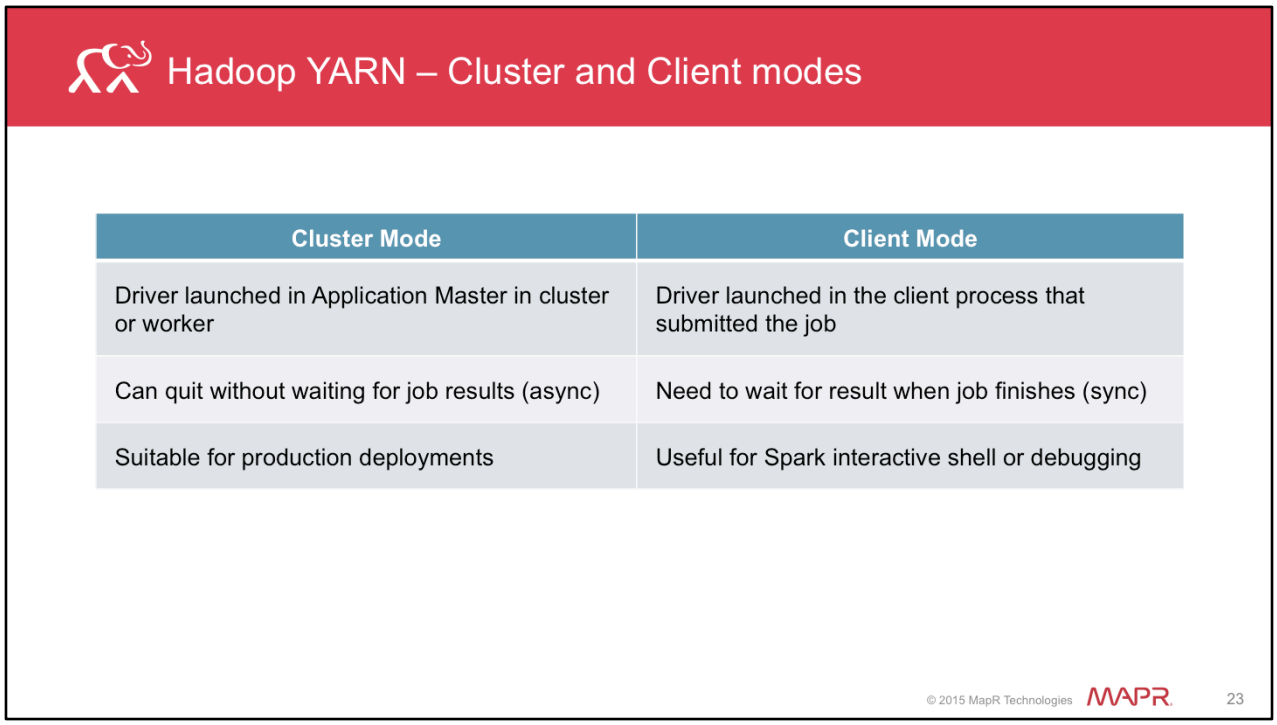
Spark will only run on those nodes on which Spark is installed.
To run an application on the Spark cluster, simply pass the spark://IP:PORT URL of the master as to the SparkContext constructor.



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.



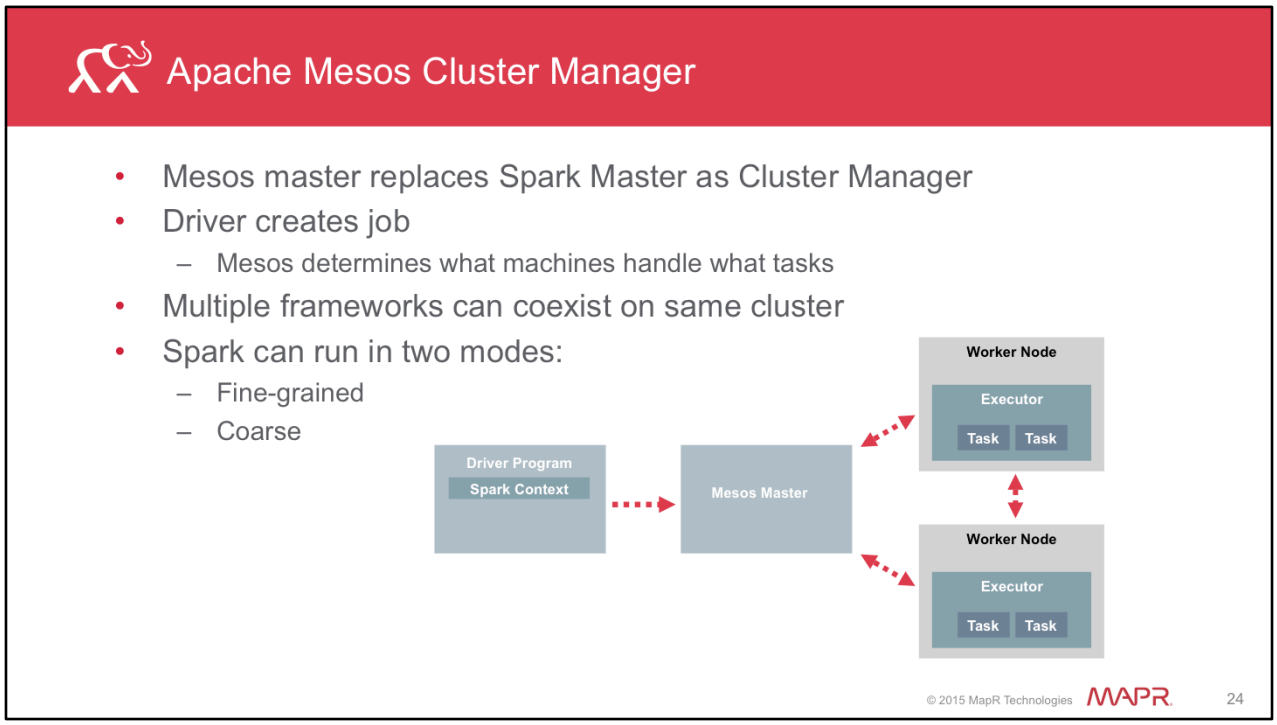
It is advantageous to run Spark on YARN if you have an existing Hadoop cluster. You can use the same Hadoop cluster without having to maintain a separate cluster. In addition, you can take advantage of he features of the YARN scheduler for categorizing, isolating and prioritizing workloads. There are 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.

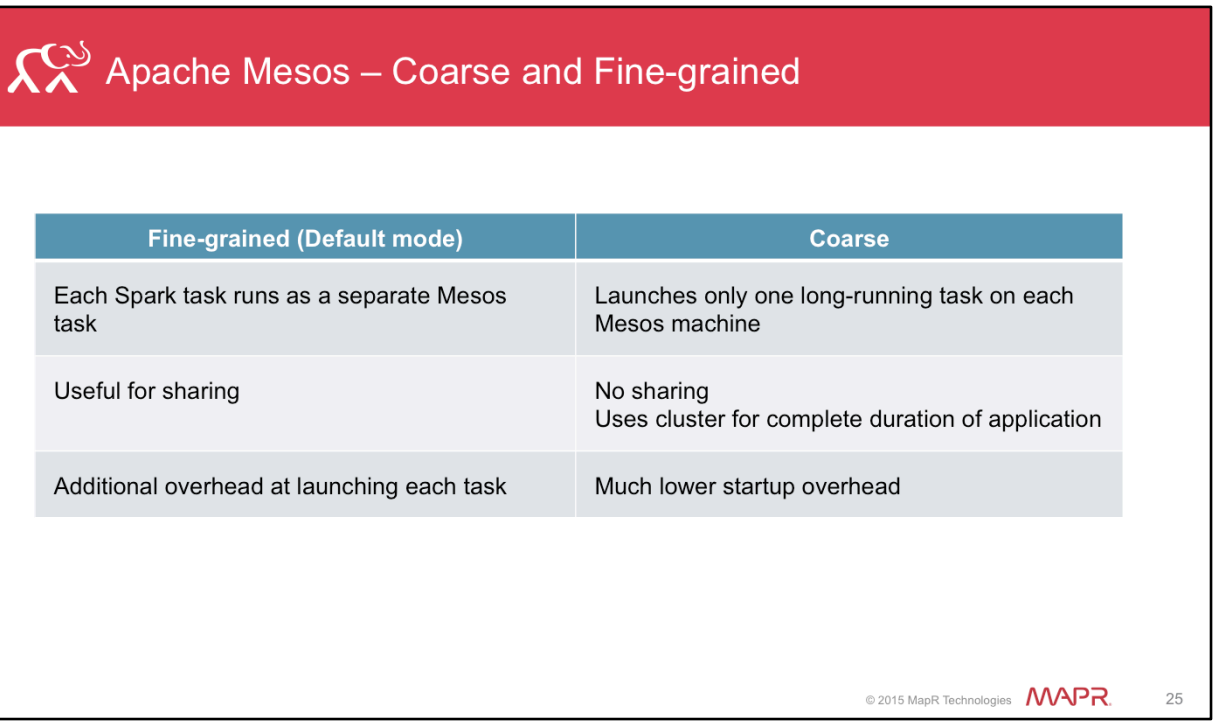
There are two deploy modes that can be used to launch Spark applications on YARN. In yarn-cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster. In yarn-client mode, the driver runs in the client process, and the application master is only used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



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

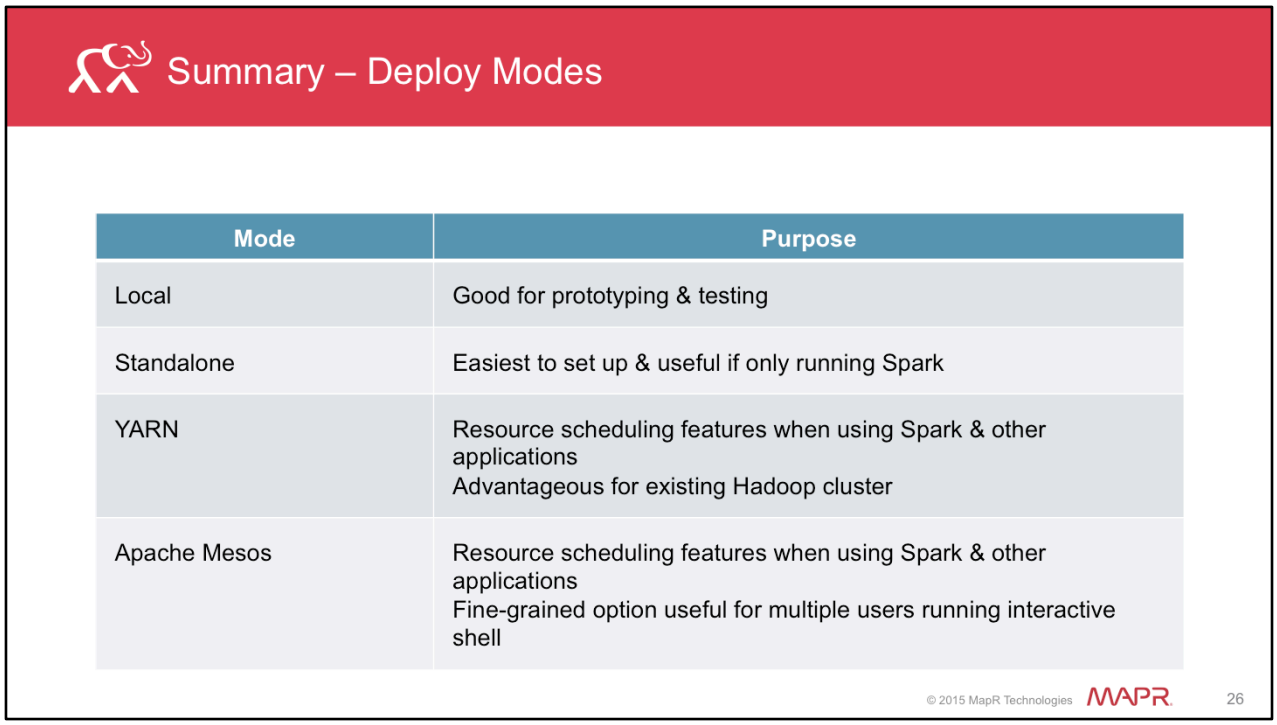
Now when a driver creates a job and starts issuing tasks for scheduling, Mesos determines what machines handle what tasks. Because it takes into account other frameworks when scheduling these many short-lived tasks, multiple frameworks can coexist on the same cluster without resorting to a static partitioning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

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

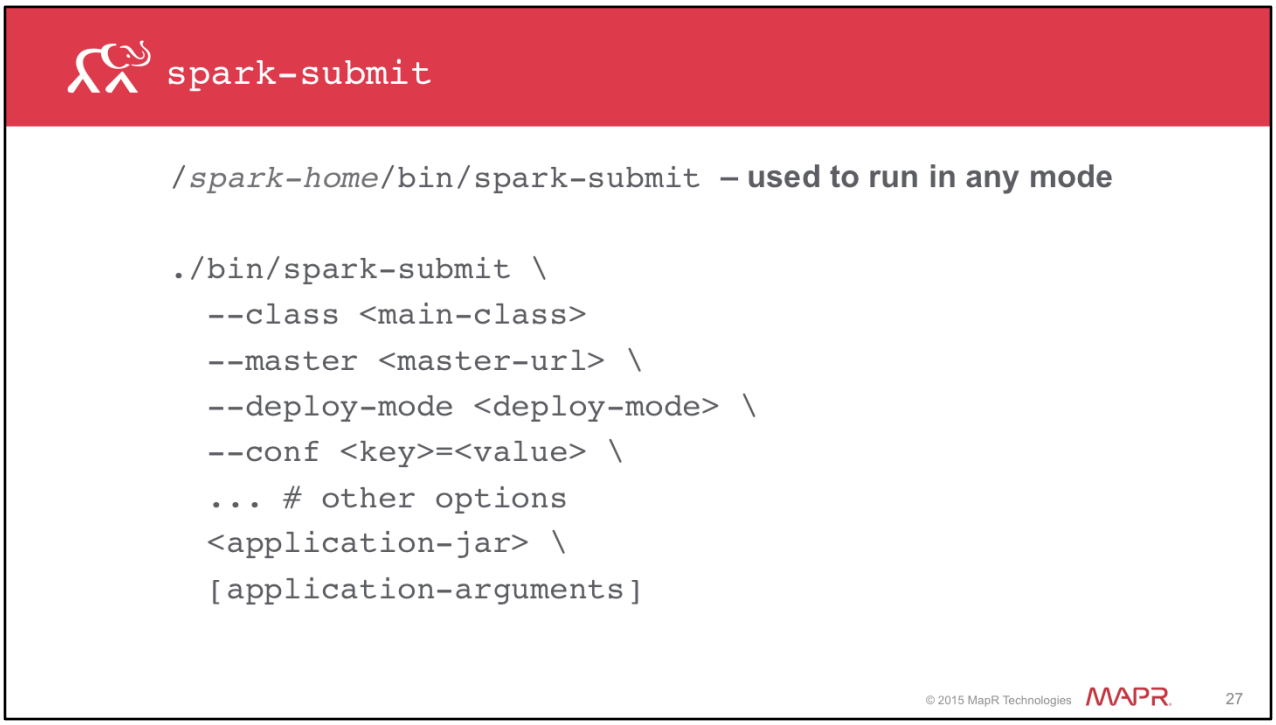


In “fine-grained” mode (default), each Spark task runs as a separate Mesos task. This allows multiple instances of Spark (and other frameworks) to share machines at a very fine granularity, where each application gets more or fewer machines as it ramps up and down, but it comes with an additional overhead 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

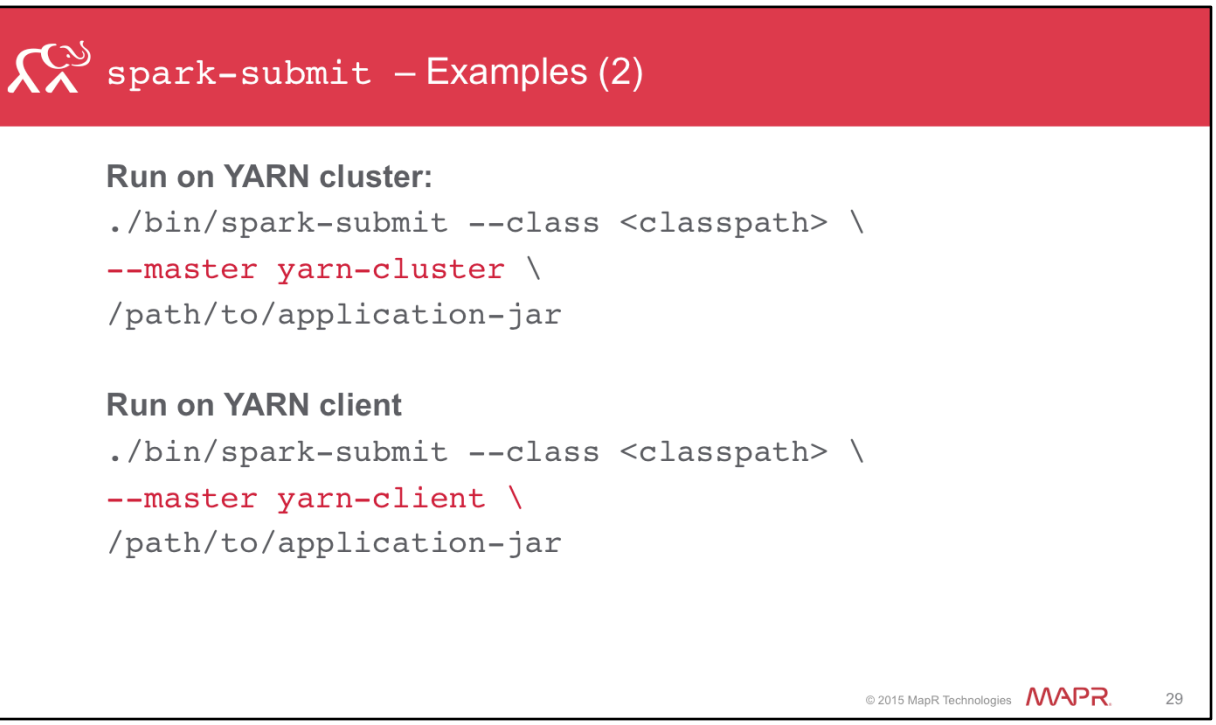
To run local mode on n cores:

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

To run standalone client mode :

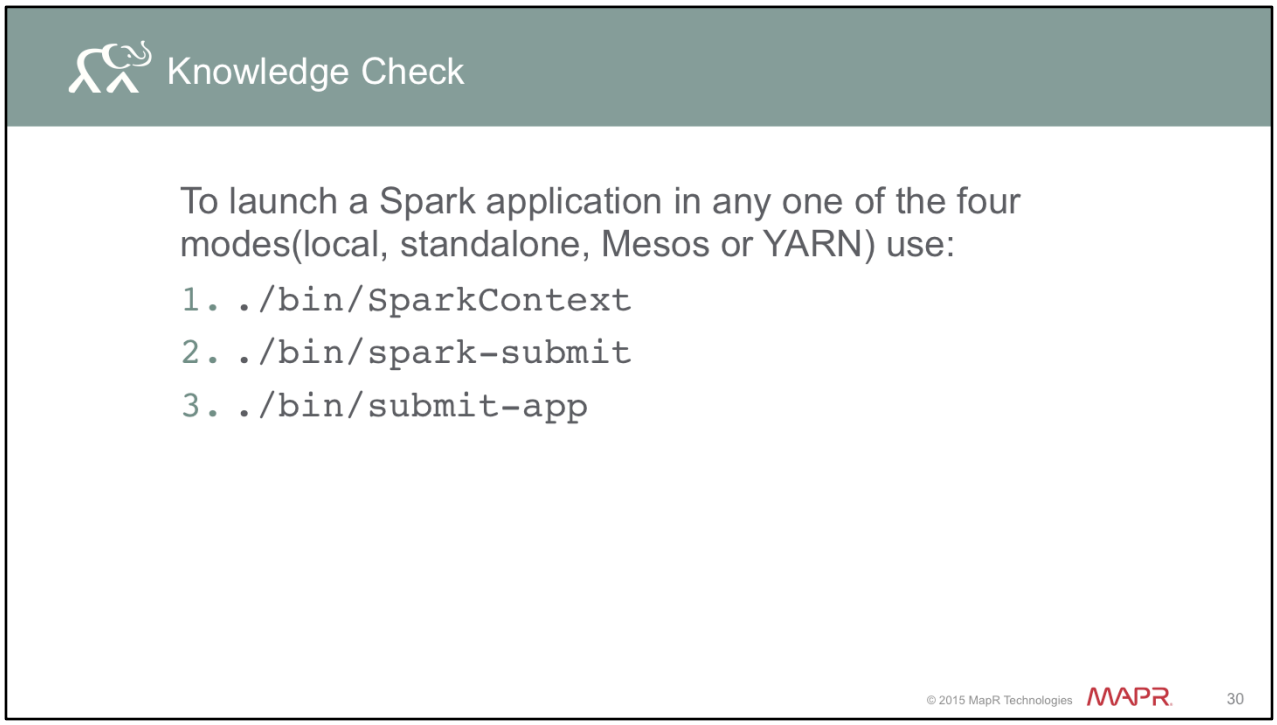
```
./bin/spark-submit --class <classpath>\
-- master spark:<master url> \
/path/to/application-jar
```


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.

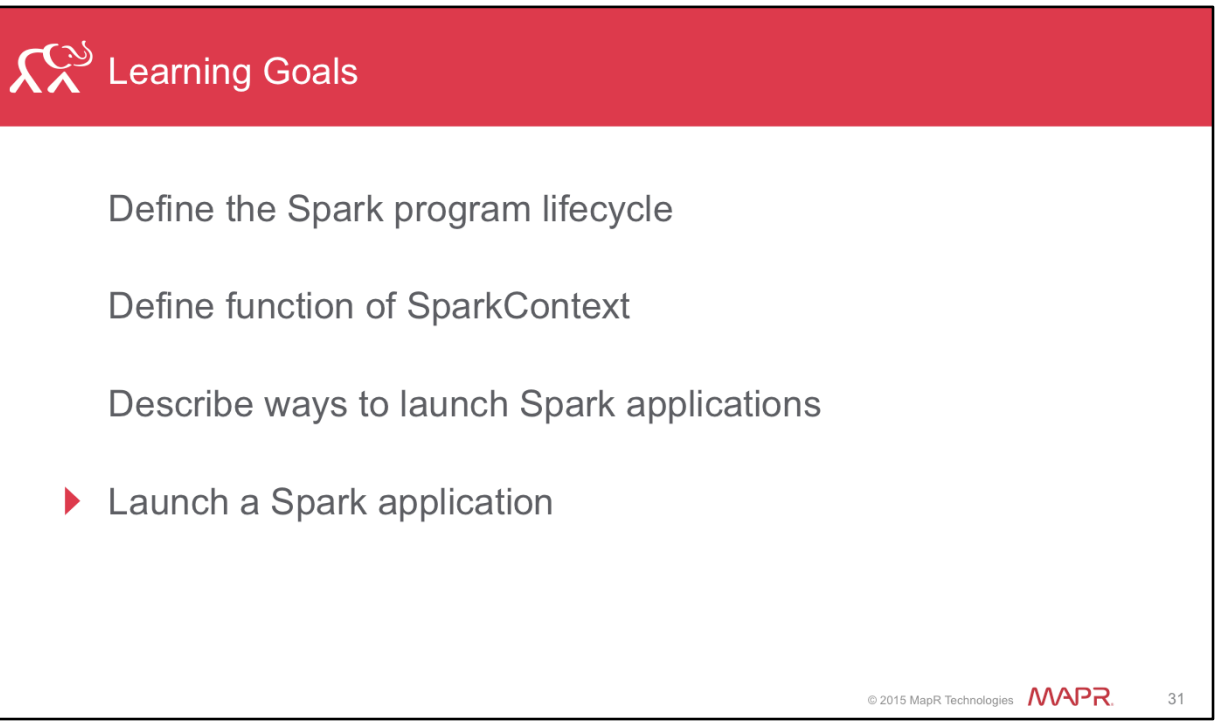


Unlike in Spark standalone and Mesos mode, in which the master's address is specified in the "master" parameter, in YARN mode the ResourceManager's address is picked up from the Hadoop configuration. Thus, the master parameter is simply "yarn-client" or "yarn-cluster".
To launch a Spark application in 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

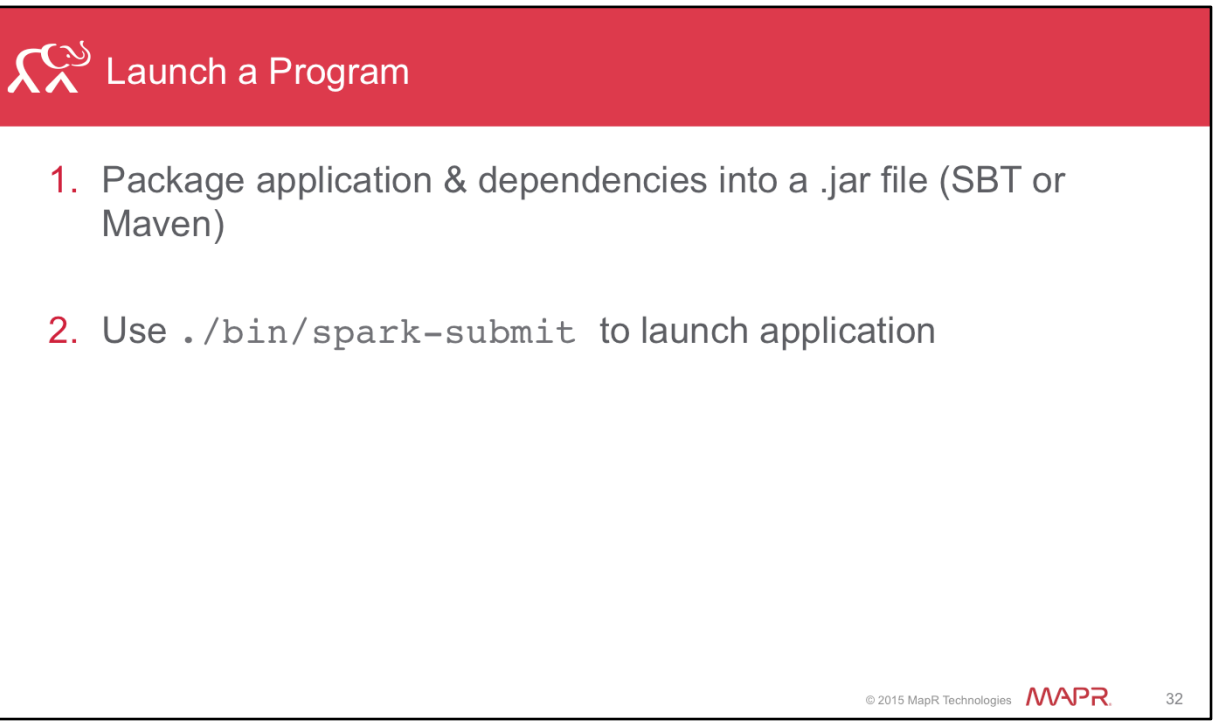
We will see next an example of how to run our application.


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

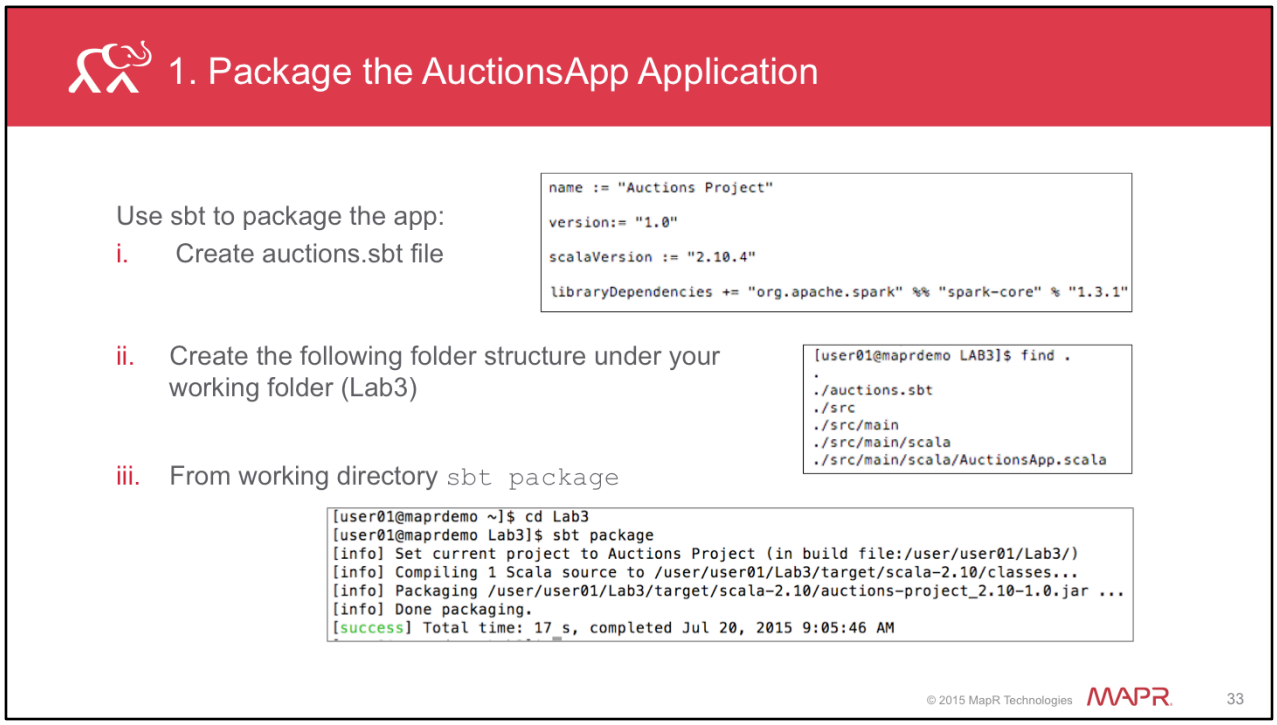
Answer: 2



We will now launch our Spark application.



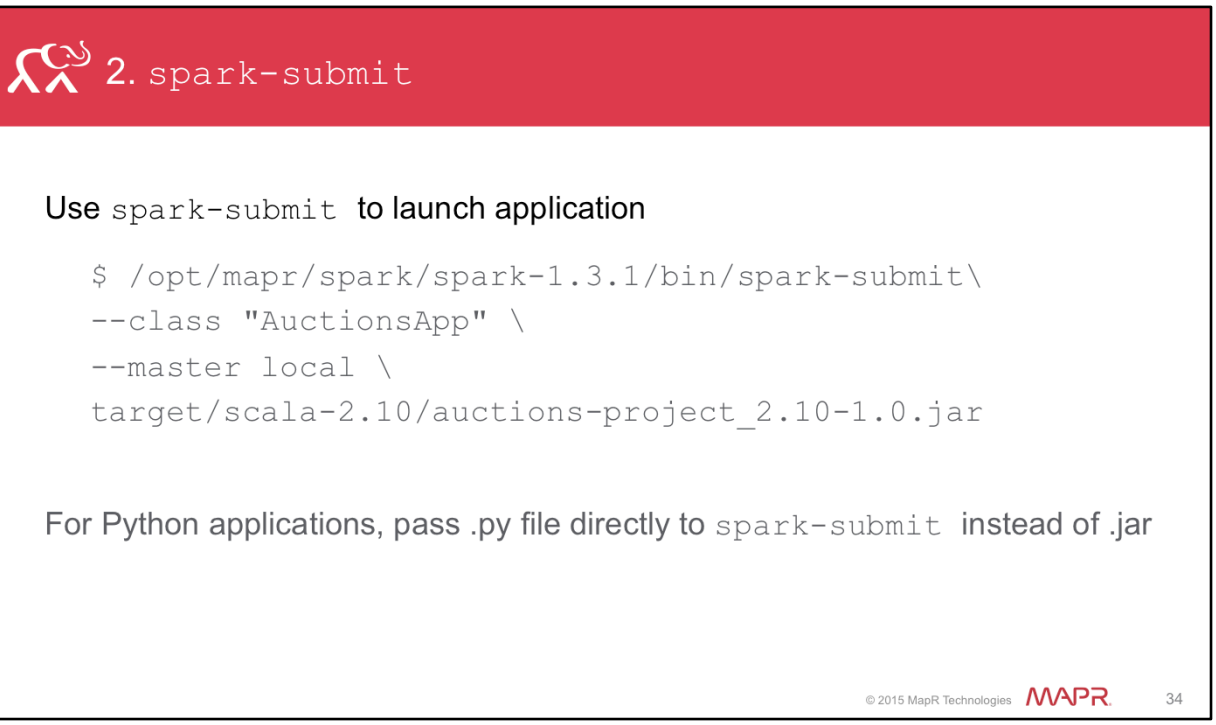
Package your application and any dependencies into a .jar file. For Scala apps, you can use Maven or SBT.



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.

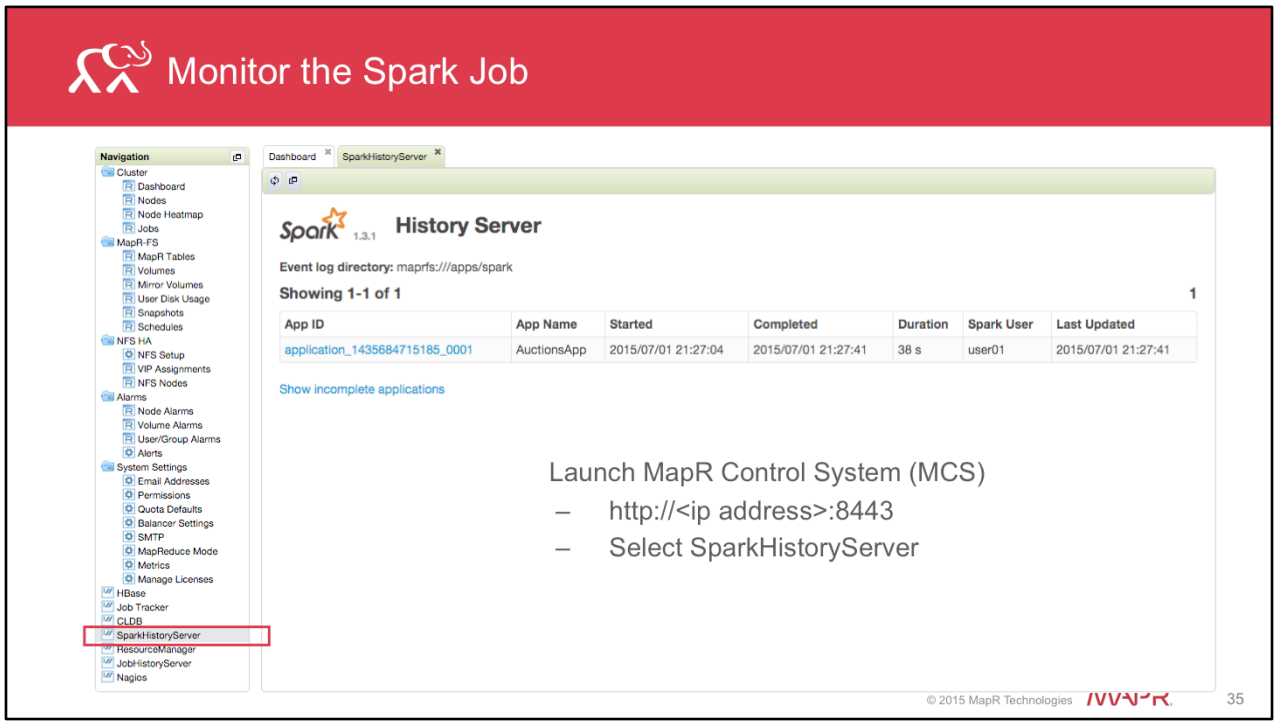
From the working directory, for example /user/user01/LAB3, run sbt package.

Note that you need to have sbt installed.

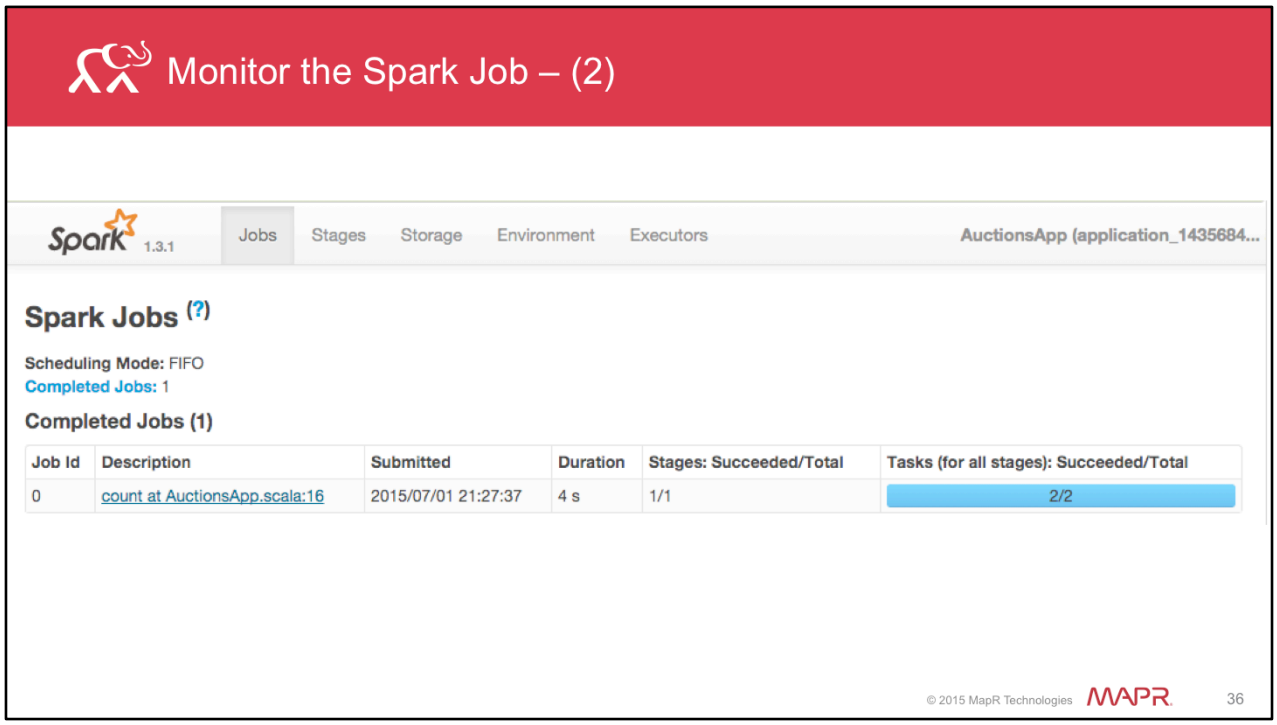


Once you have packaged your application, you should have the jar file.
You can now launch the application using spark-submit.
We are using local mode here. In the real world after testing, you would deploy to cluster mode. For example, to run on yarn-cluster, you would just change the master option to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

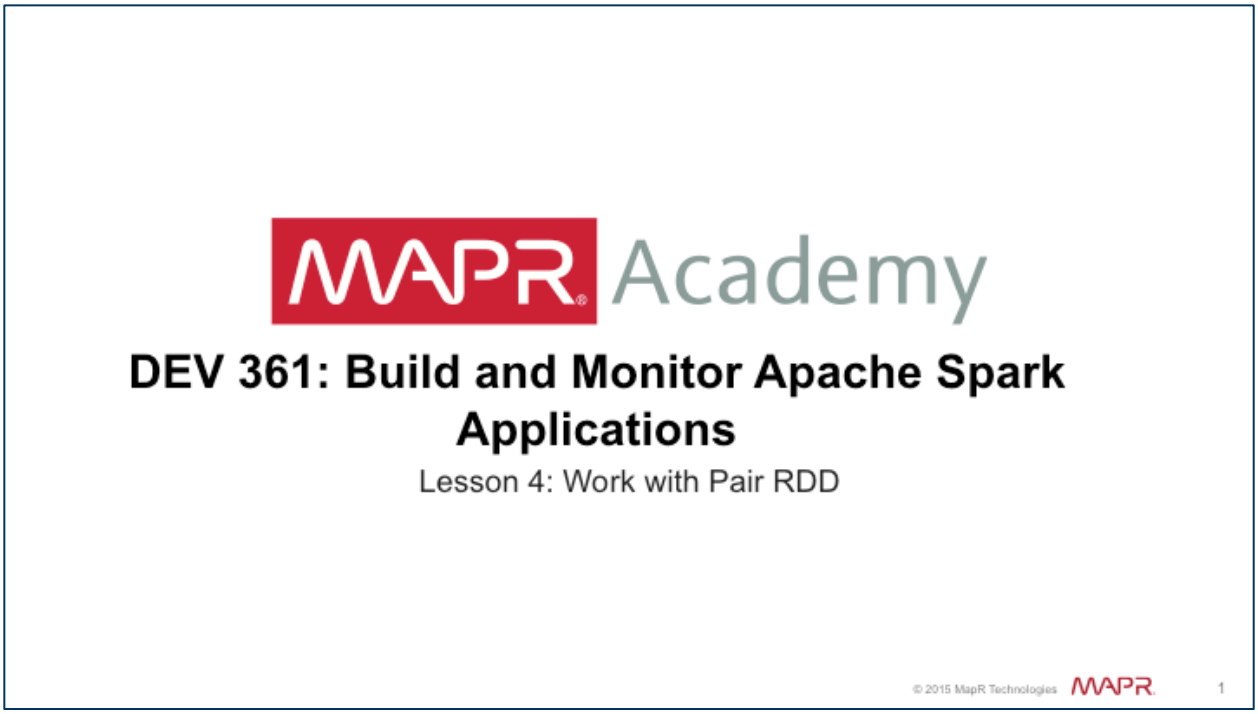
Note that you can also create the application in Python.

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.

DEV 361: Build and Monitor Apache Spark Applications
Slide Guide

Version 5.1 – Summer 2016

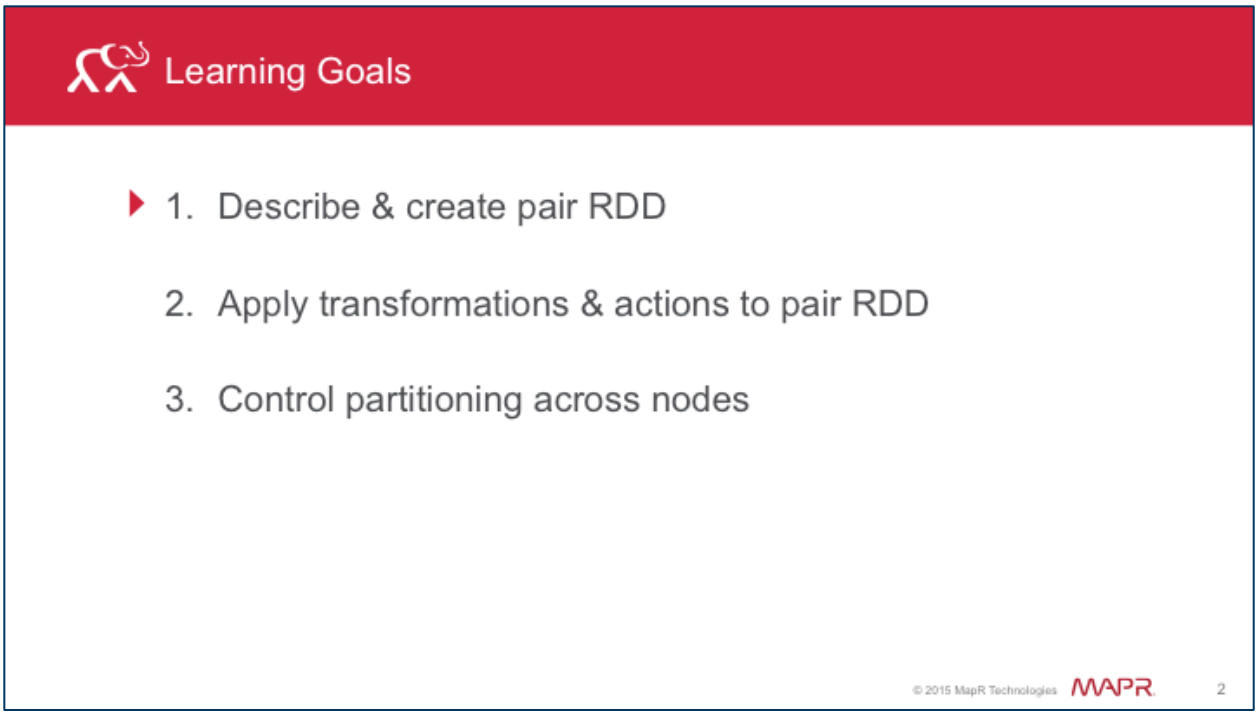
Lesson 4: Work with Pair RDD



Notes on Slide 1:

Welcome to Apache Spark: Build and Monitor Apache Spark Applications – Lesson 4: Work with Pair RDD.

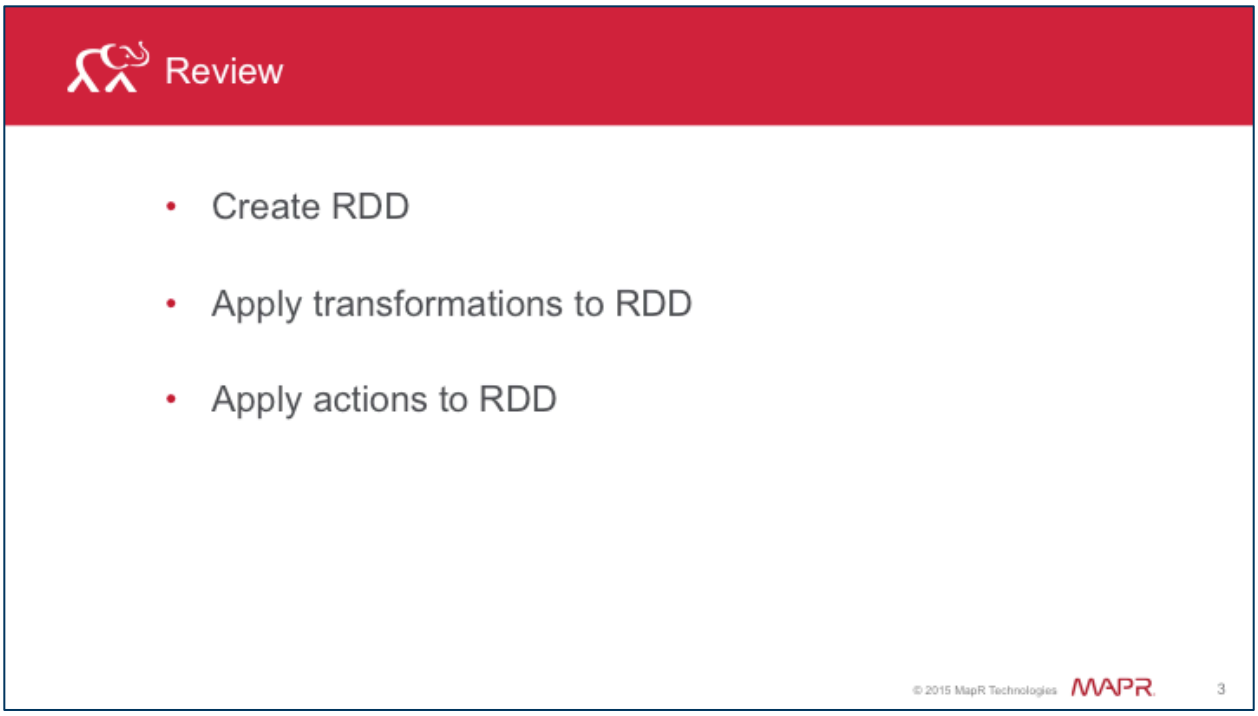
In this lesson we will describe key-value pairs as data sets and how and where they are used. We create pair RDD and apply operations to them. This lesson also describes how to control partitioning across nodes and use the partitioner property.



Notes on Slide 2:

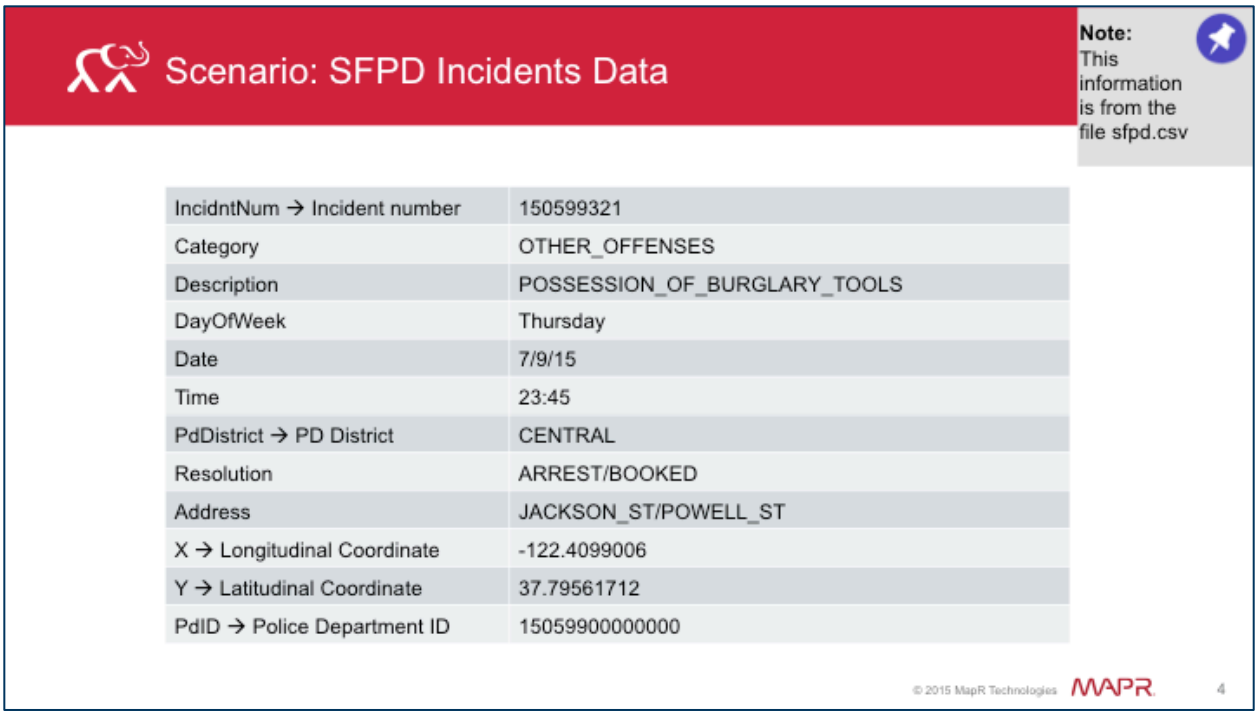
At the end of this lesson, you will be able to:

1. Describe and create pair RDD
2. Apply transformations and actions to pair RDD
3. Control partitioning across nodes



Notes on Slide 3:

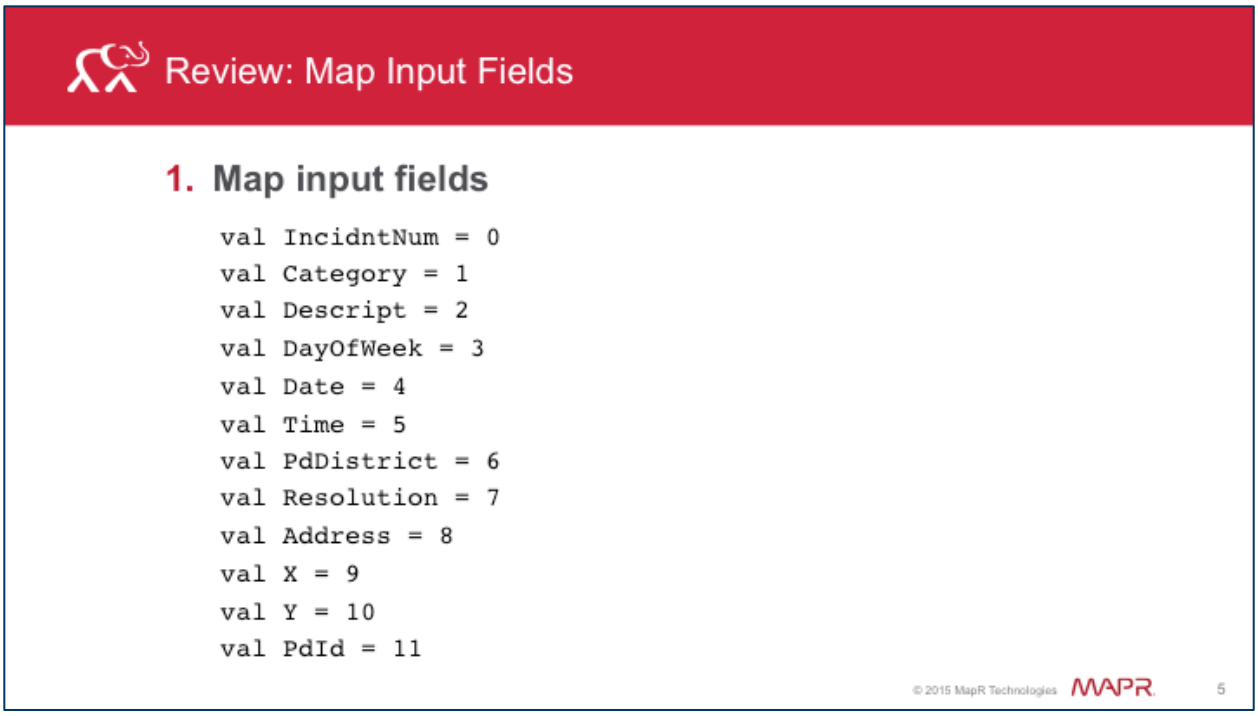
Before we start with pair RDDs, let us review creating RDD and applying actions and transformations to RDD. You should complete DEV 360 before continuing.



Notes on Slide 4:

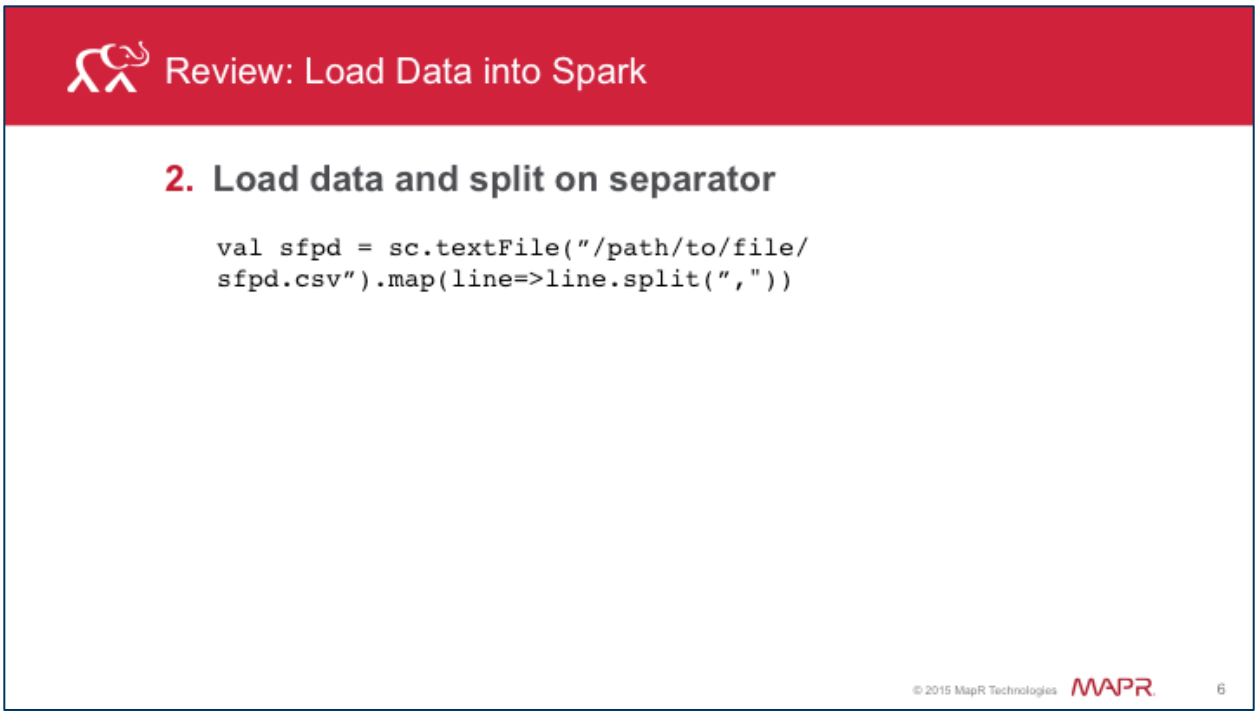
We will use this dataset in examples and Lab activities. sfpd.csv contains the fields shown here with example data points. This dataset consists of Incidents derived from the SFPD Crime Incident Reporting system between Jan 2013 - July 2015. We are going to explore this data to answer questions such as which districts have the maximum incidents or which five categories have the maximum number of incidents.

First, we will do a quick review.

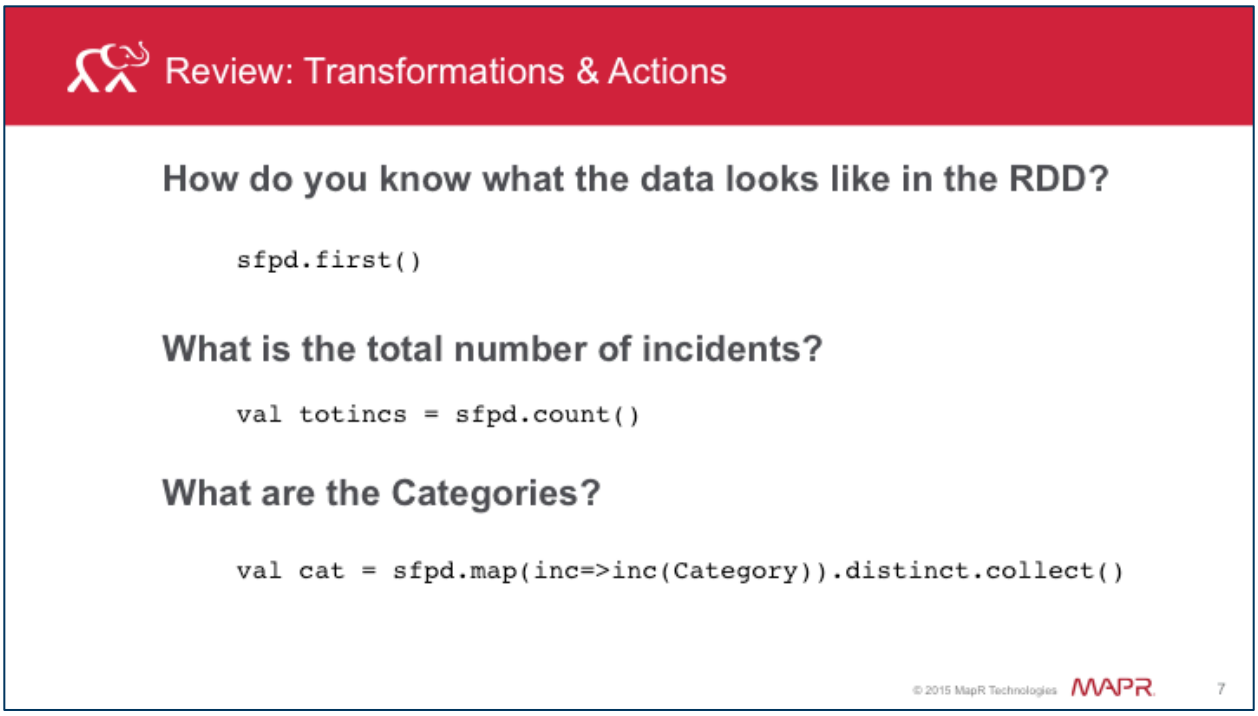


Notes on Slide 5:

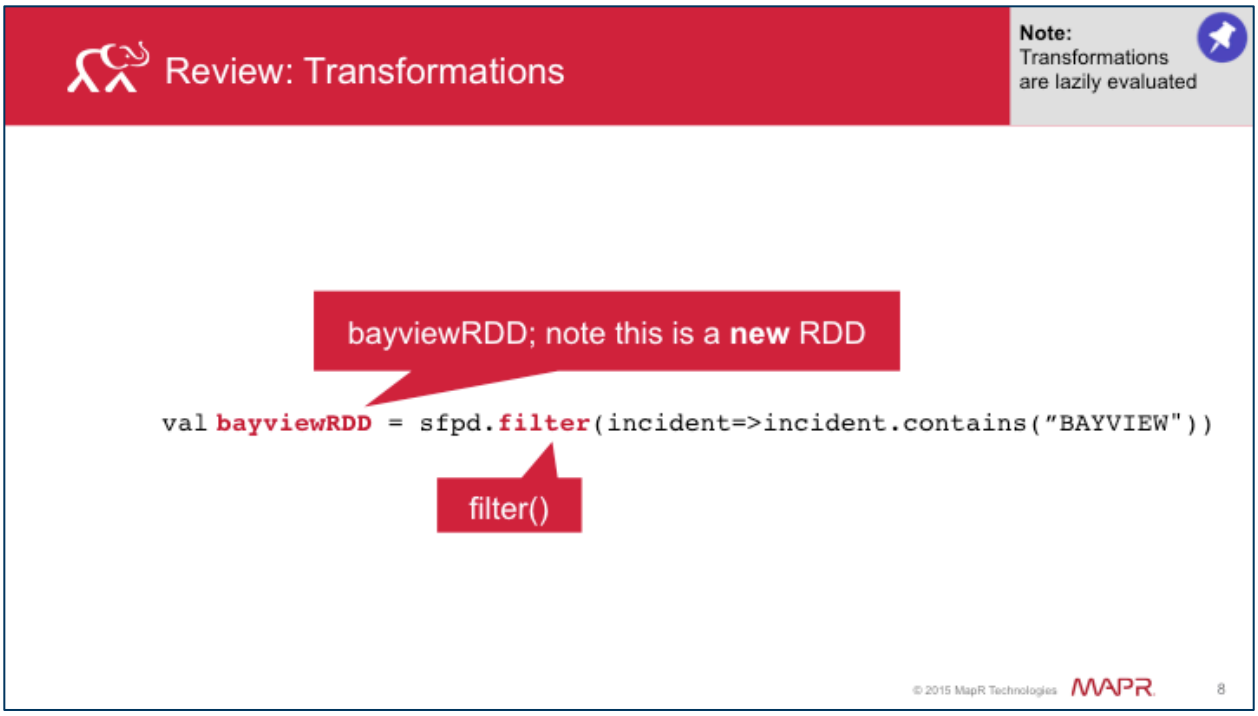
We will use the Spark Interactive Shell to load the data, create RDD and apply transformations and actions. First we map the input fields.



Notes on Slide 6:
We load the csv file using the SparkContext method textFile. We are also applying the map transformation to split on the comma.



Notes on Slide 7:
This provides a review of creating RDDs, and applying data set operations to RDD that were covered in the DEV 360 course.

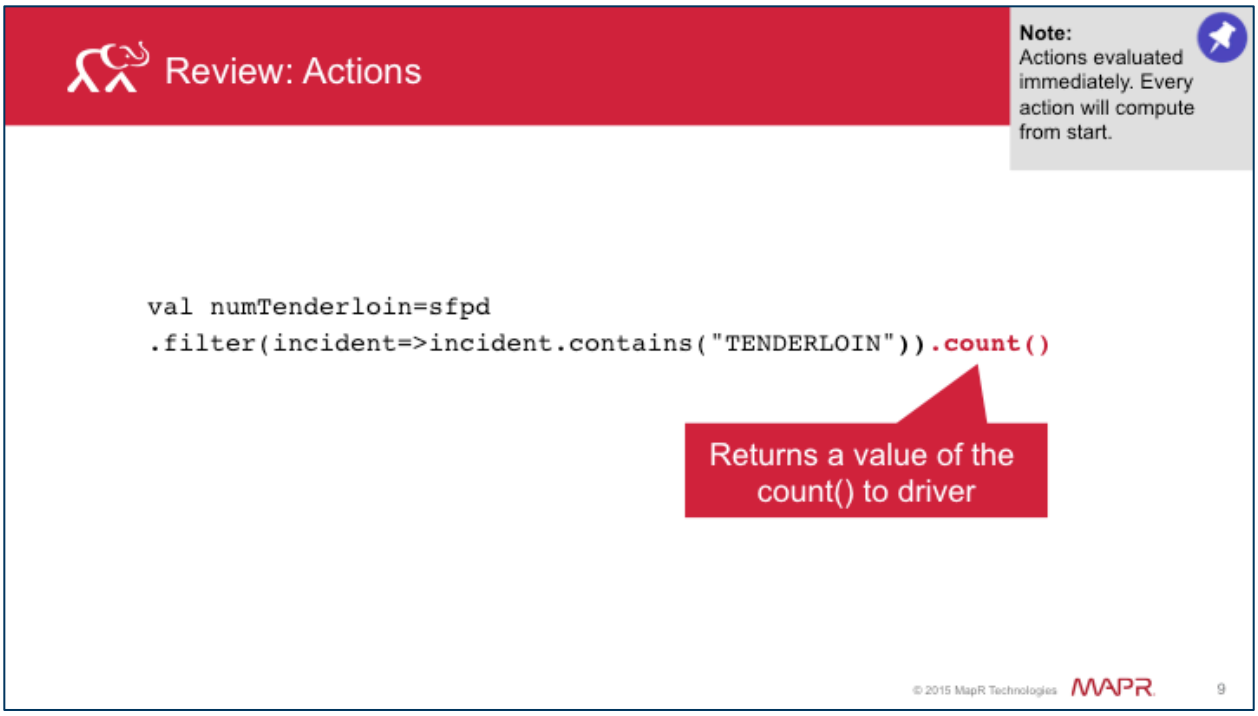


Notes on Slide 8:

Transformations are lazily evaluated. In this example, we are defining the instructions for creating the bayviewRDD. Nothing is computed until we add an action.

The filter transformation is used to filter all those elements in the sfpd RDD that contain "BAYVIEW".

Other examples of transformations include map, filter and distinct.



Notes on Slide 9:

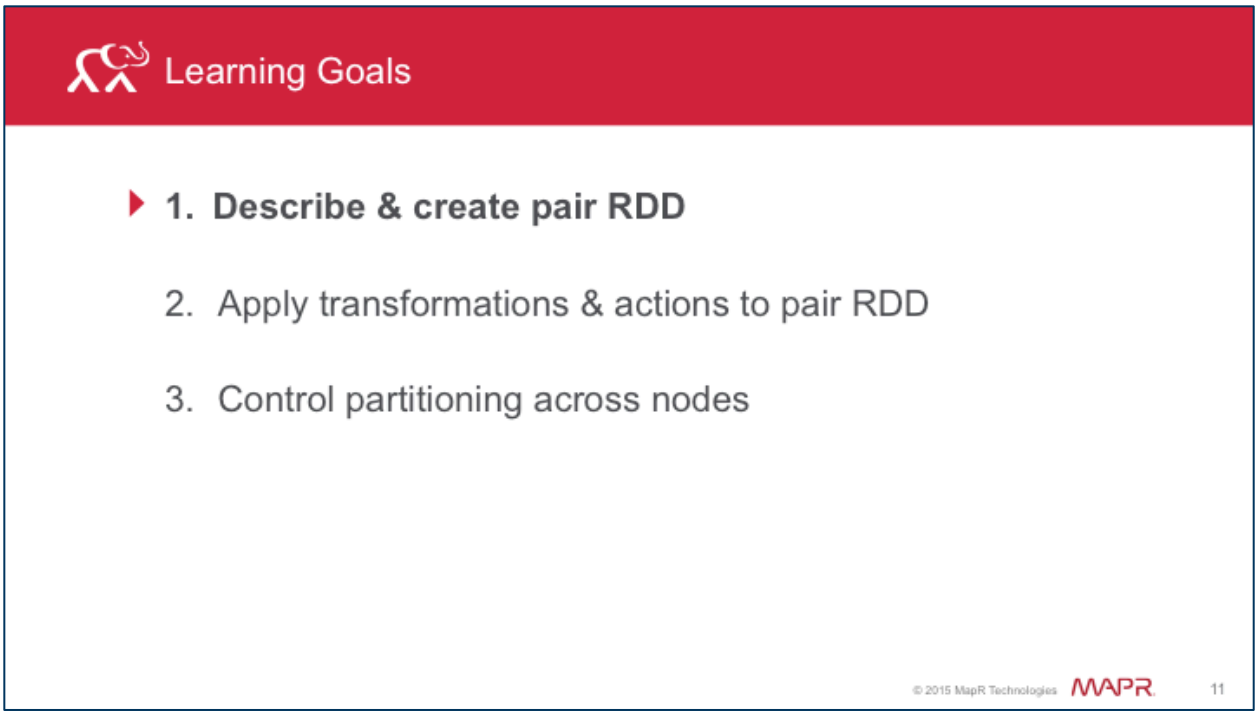
When we run the command for an action, Spark will load the data, create the inputRDD, and then compute any other defined transformations and actions.

In this example, calling the count action will result in the data being loaded into the sfpd RDD, the filter transformation being applied and then the count.

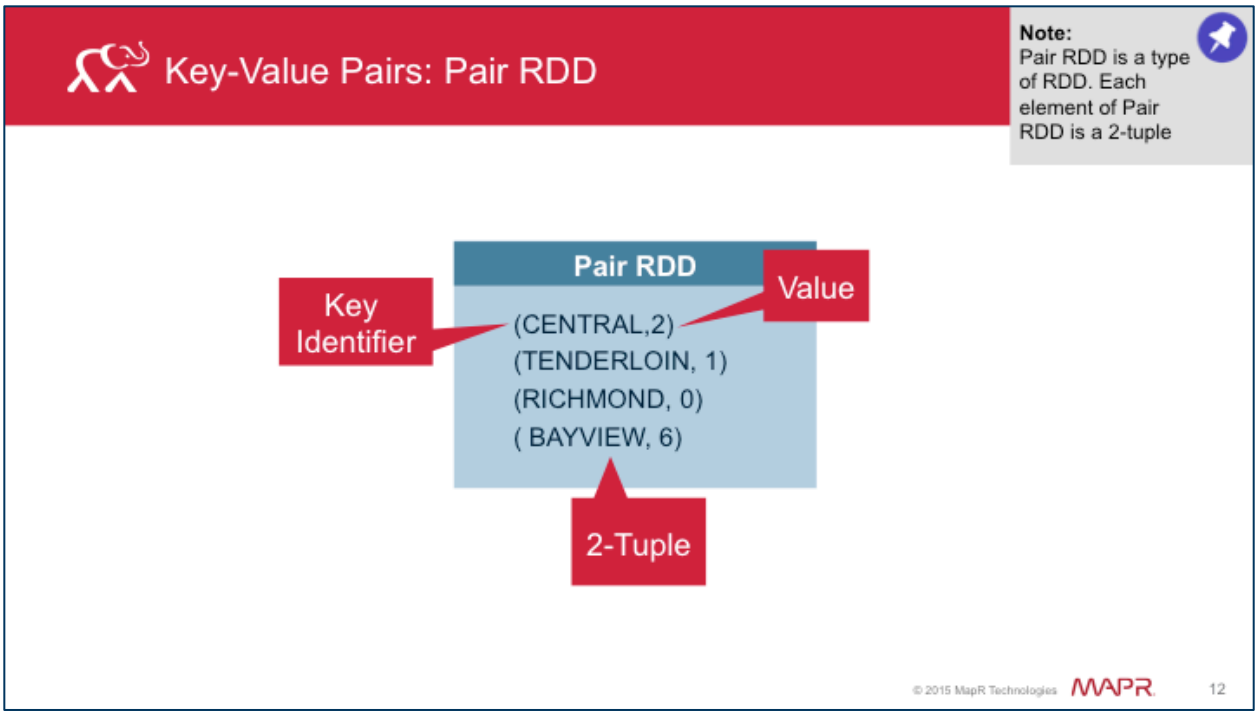
We will now look at pair RDDs that are a type of RDD.

Notes on Slide 10:

In this activity you will load data into Apache Spark and explore the data using dataset operations. This activity is a review of creating RDDs and applying transformations and actions on it.



Notes on Slide 11:
In this section we will describe key-value pairs as data sets and create pair RDDs.



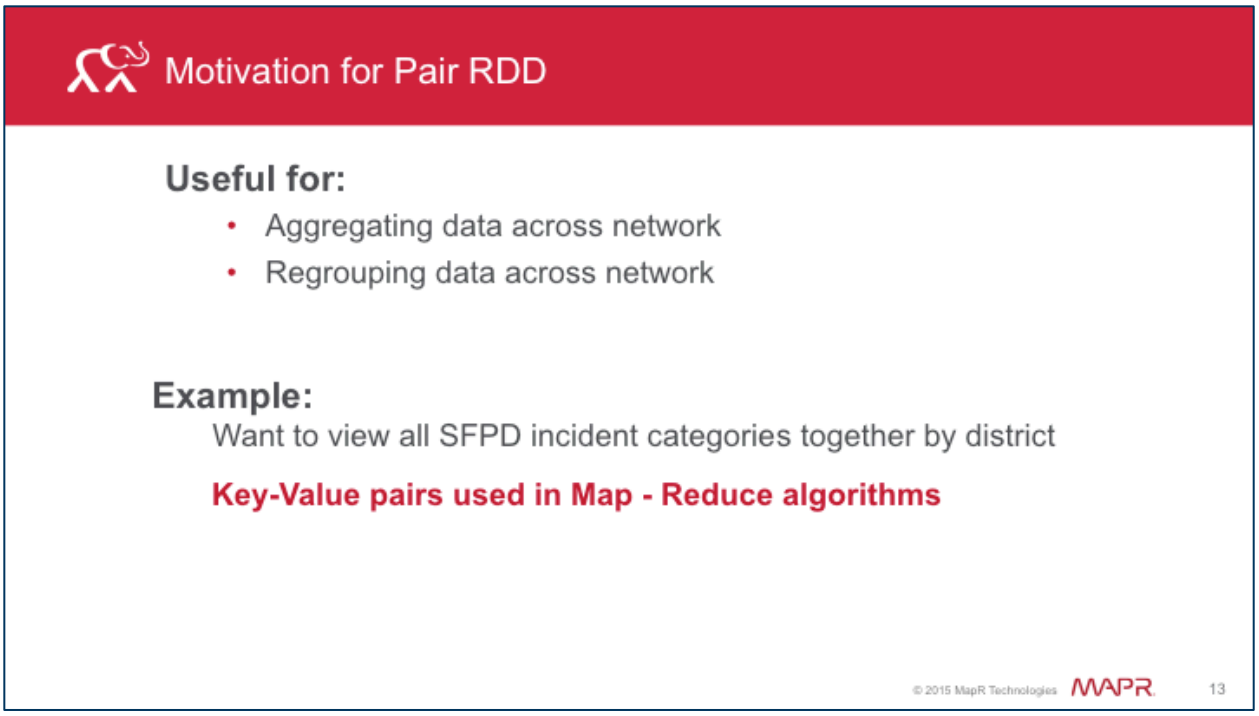
Notes on Slide 12:

The key-value pair paradigm can be found in a number of programming languages. It is a data type that consists of a group of key identifiers with a set of associated values.

When working with distributed data, it is useful to organize data into key-value pairs as it allows us to aggregate data or regroup data across the network.

Similar to MapReduce, Spark supports Key-Value pairs in the form of Pair RDD.

Spark's representation of a key-value pair is a Scala 2-tuple



Notes on Slide 13:

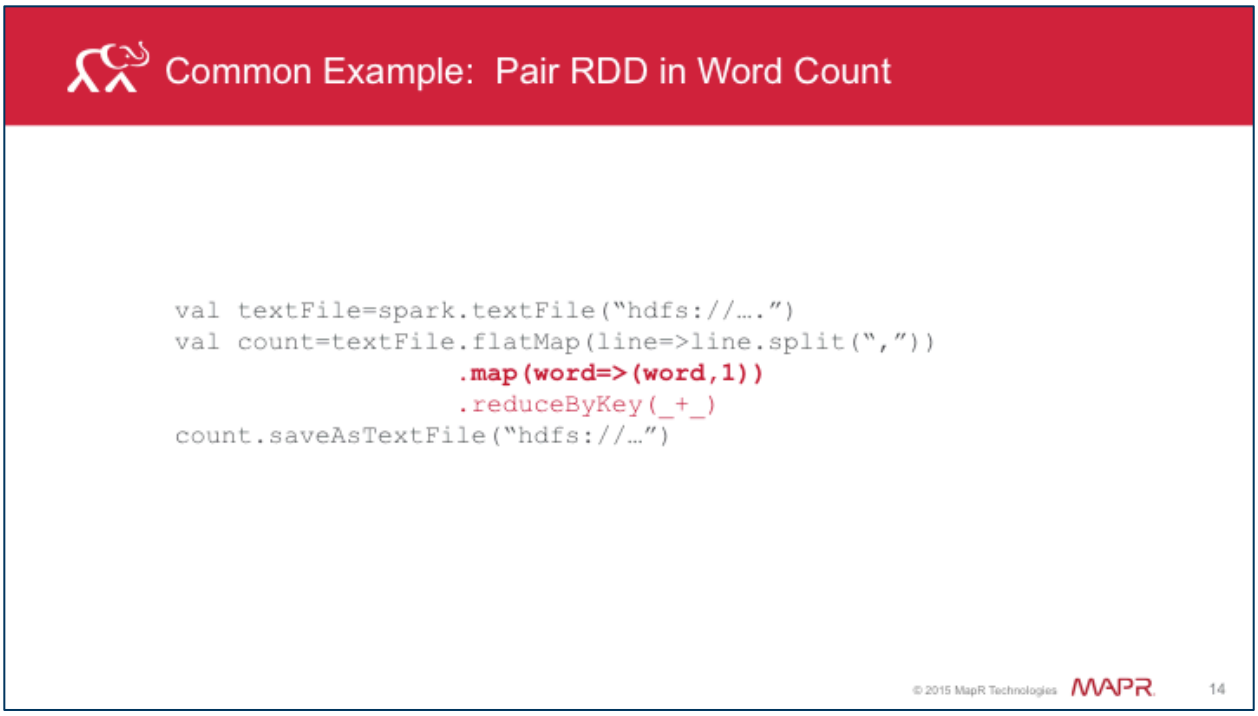
Pair RDDs are used in many Spark programs. They are useful when you want to aggregate and/or regroup data across a network.

For example, you can extract the District and use it to view all SFPD incidents together for a district.

Another example, is extracting the customerID and using it as in identifier to view all the customer's orders together.

A very common use case is in map reduce algorithms.

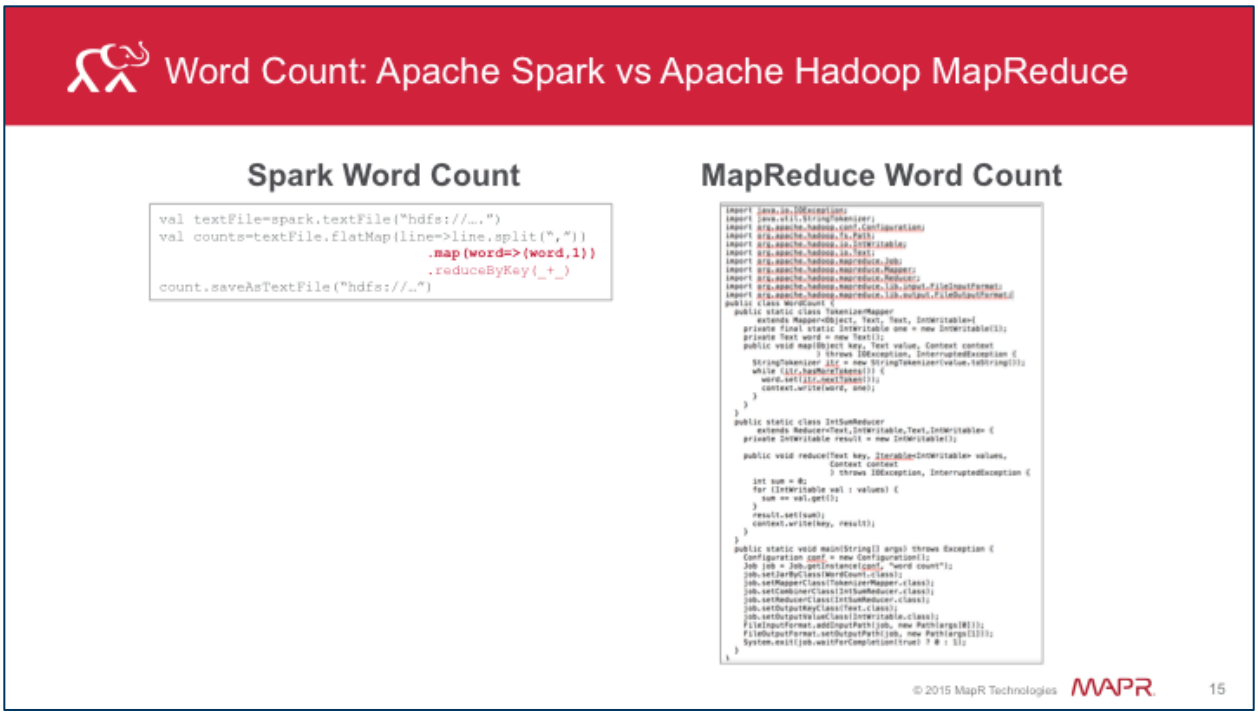
Let us take a brief look at the differences between Hadoop MapReduce and Apache Spark.



Notes on Slide 14:

Word count is a typical example used to demonstrate Hadoop MapReduce.

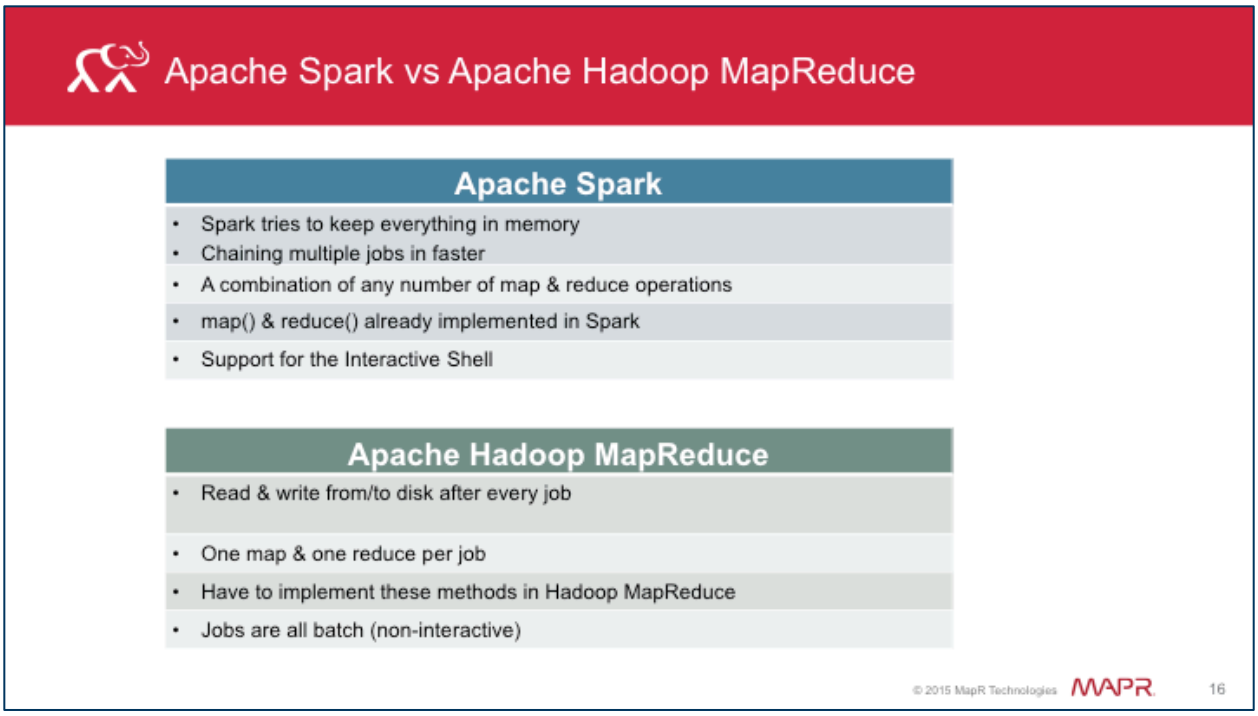
In the code block shown here, the first line defines the location of the input file. The second line loads the data into an RDD; splits the file on the separator into dataset whose elements are the words in the input file; the map transformation is applied to the out put of the flatMap to create a pair RDD consisting of tuples for each occurrence of the word – (word,1); finally the reduceByKey operation then reduces the pair RDD to give the count.



Notes on Slide 15:

On the left is the code we just saw that gives us the count of words in Scala using Spark. On the right is the Hadoop MapReduce program for word count.

We will discuss some differences between Apache Spark and Hadoop MapReduce next.

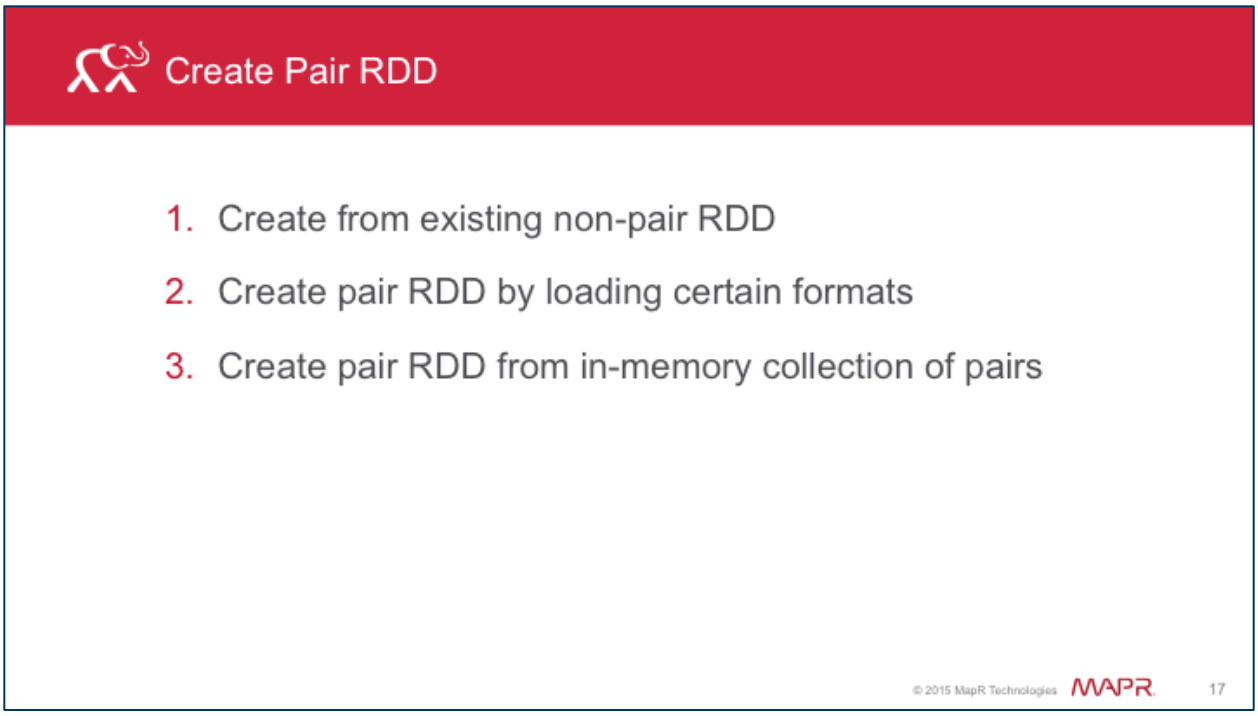


Notes on Slide 16:

This table lists differences between Apache Spark and Apache Hadoop MapReduce.

- The key differentiator between Apache Spark and MapReduce is that Spark keeps everything in memory while MapReduce reads from and writes to disk for every job.
- You can have only one map and one reduce per MapReduce job. In Spark, you can have a combination of any number of map and reduce operations.
- Furthermore, Spark provides high level operators such as map, reduce and joins whereas in MapReduce, you have to implement these methods.
- Apache Spark provides the Interactive Shell fro Scala and Python that allows you to write your code interactively. Apache Hadoop MapRedude jobs on the other hand are all batch.

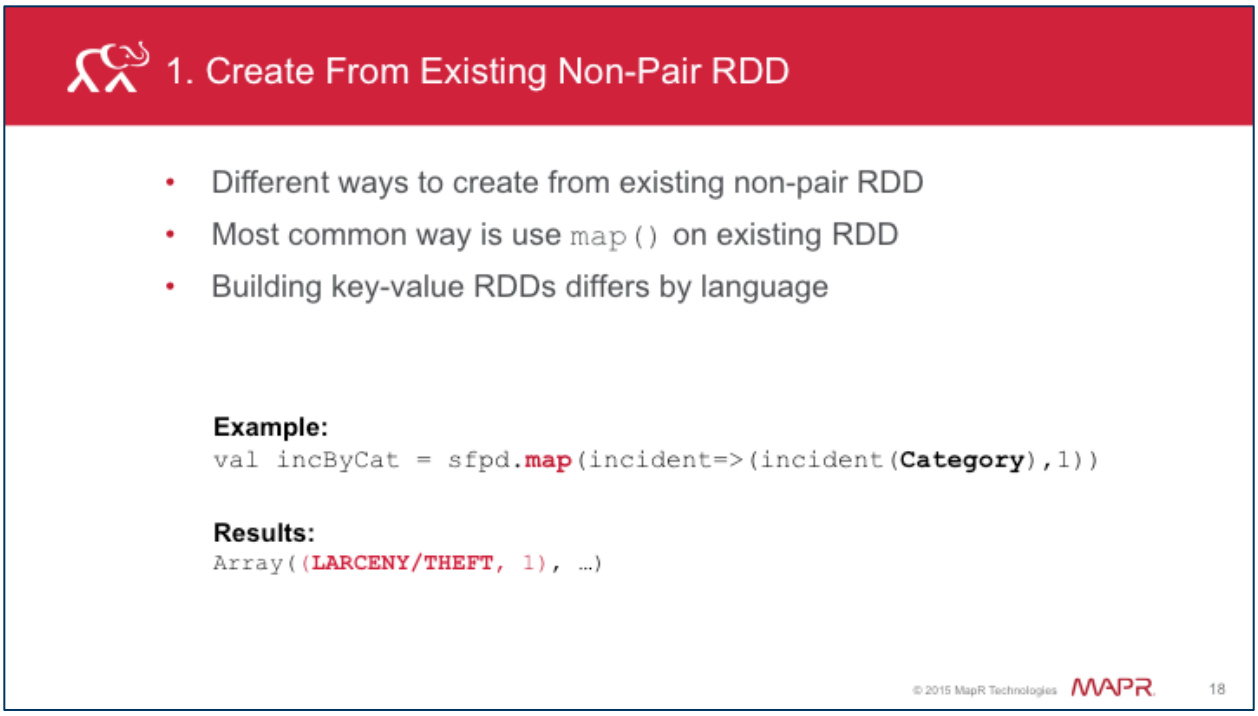
We will now take a look at creating pair RDDs.



Notes on Slide 17:

Pair RDD can be created in the following ways:

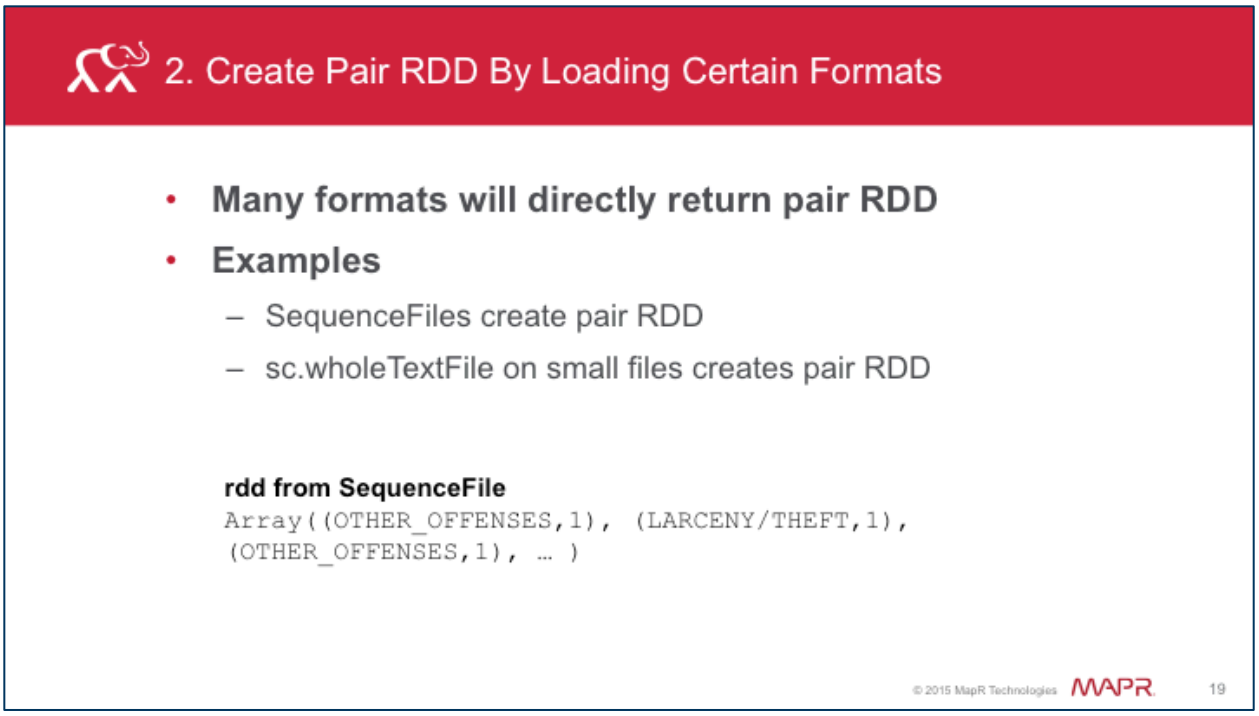
1. Pair RDDs are most often created from already-existing non-pair RDDs.
2. You can also get pair RDD when loading data. There are many data formats that when loaded, will create pair RDD.
3. Another way is to create a pair RDD from an in-memory collection of pairs.



Notes on Slide 18:

There are many ways to create a pair RDD from existing non-pair RDDs. The most common way is using a map transformation. The way to create pair RDDs is different in different languages. In Python and Scala, we need to return an RDD consisting of tuples.

In this example, we are creating a pair RDD called incByCat from the sfpd RDD by applying a map transformation. The resulting RDD contains tuples as shown.

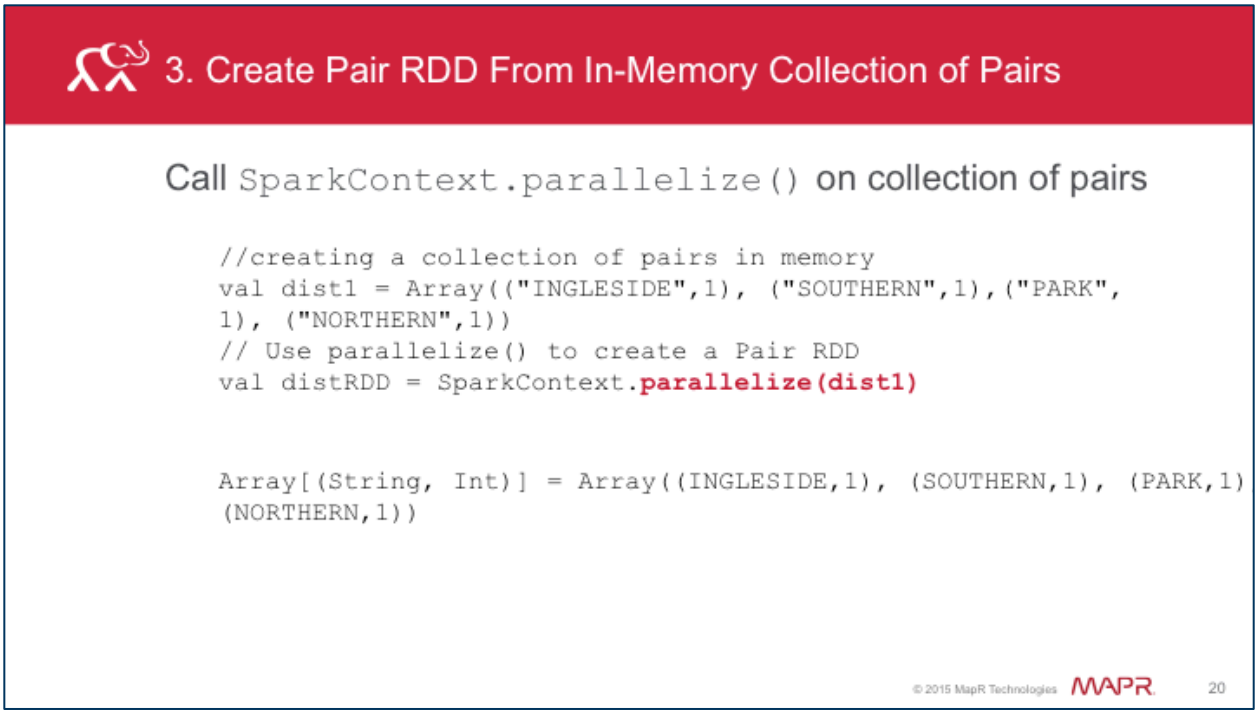


Notes on Slide 19:

Many formats will directly return pair RDDs for their key-value data.

For examples: SequenceFiles will create pair RDD. In this example, the SequenceFile consists of key-value pairs (Category,1) . When loaded into Spark it produces a pair RDD as shown.

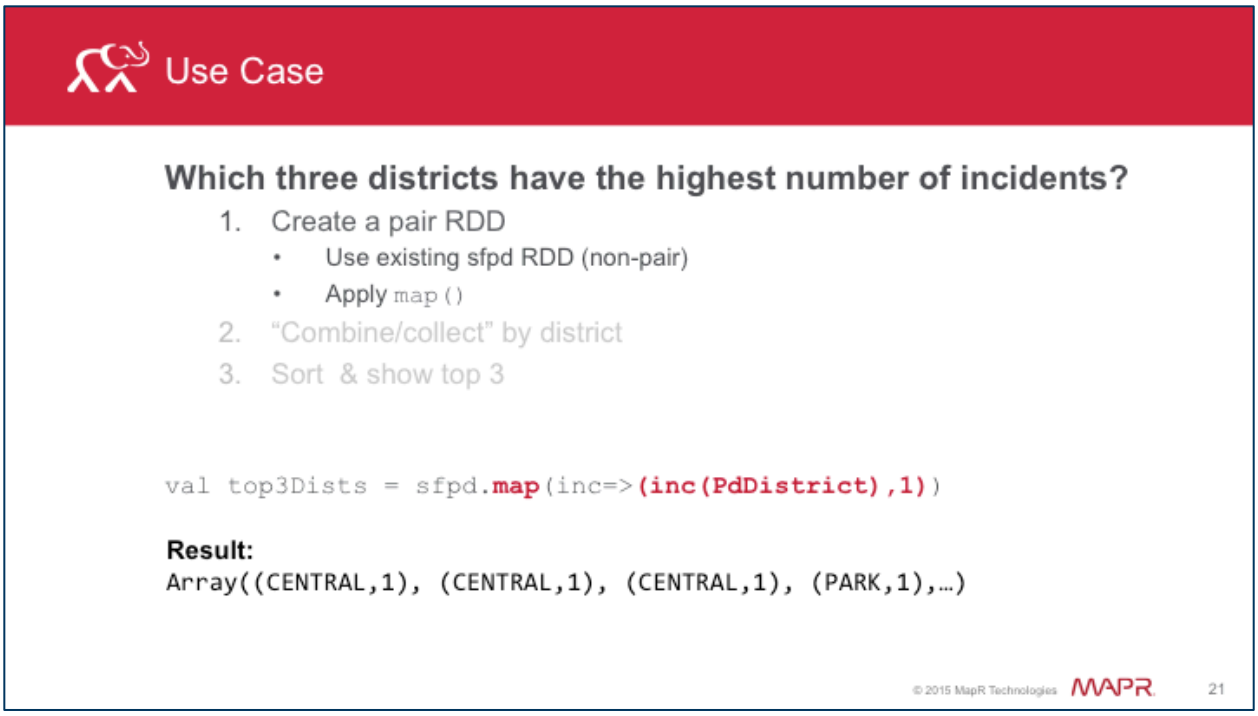
sc.wholetextFiles on a group of small text files will create pair RDD where the key is the name of the file.



Notes on Slide 20:

To create a pair RDD from an in-memory collection in Scala and Python, use SparkContext.parallelize() method on a collection of pairs.

In this example, we have a collection of of pairs, dist1, in memory. We create a pair RDD, distRDD, by applying the method SparkContext.parallelize to dist1.



Notes on Slide 21:

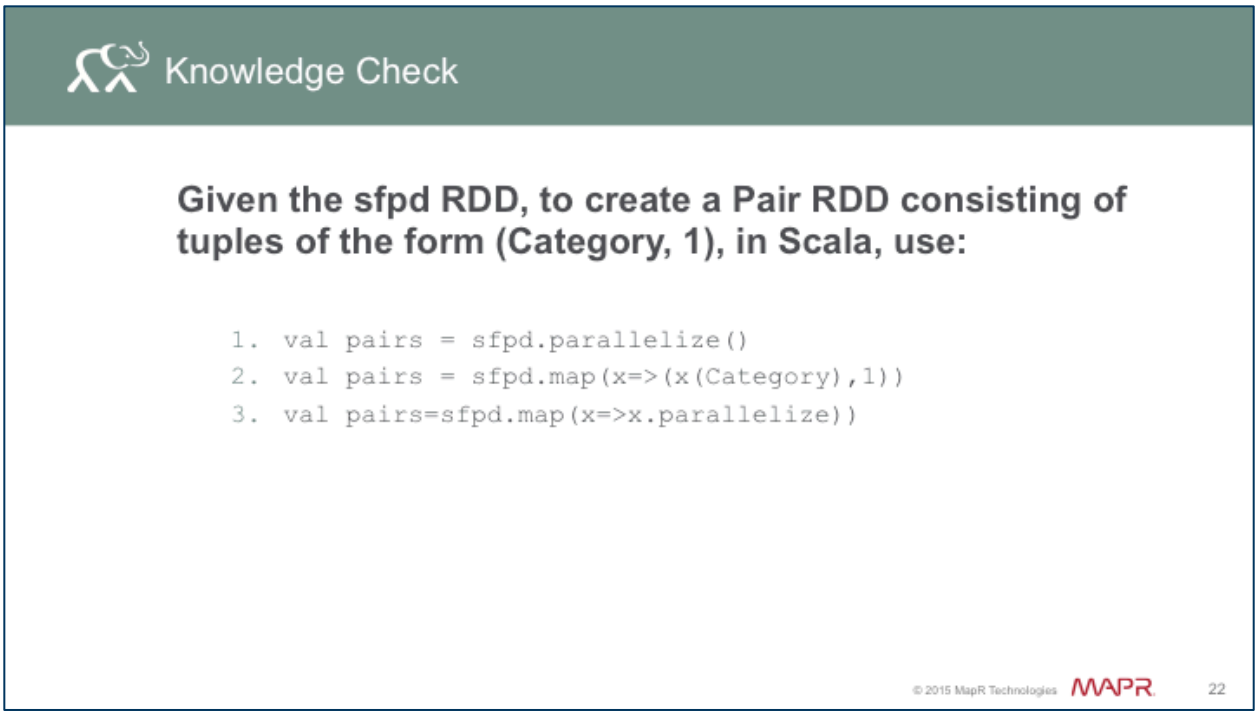
We want to answer this question of the sfpd dataset – which three districts (or Categories or Addresses) have the highest number of incidents?

Each row in the dataset represents an incident. We want to count the number of incidents in each district - i.e. how many times does the district occur in the dataset?

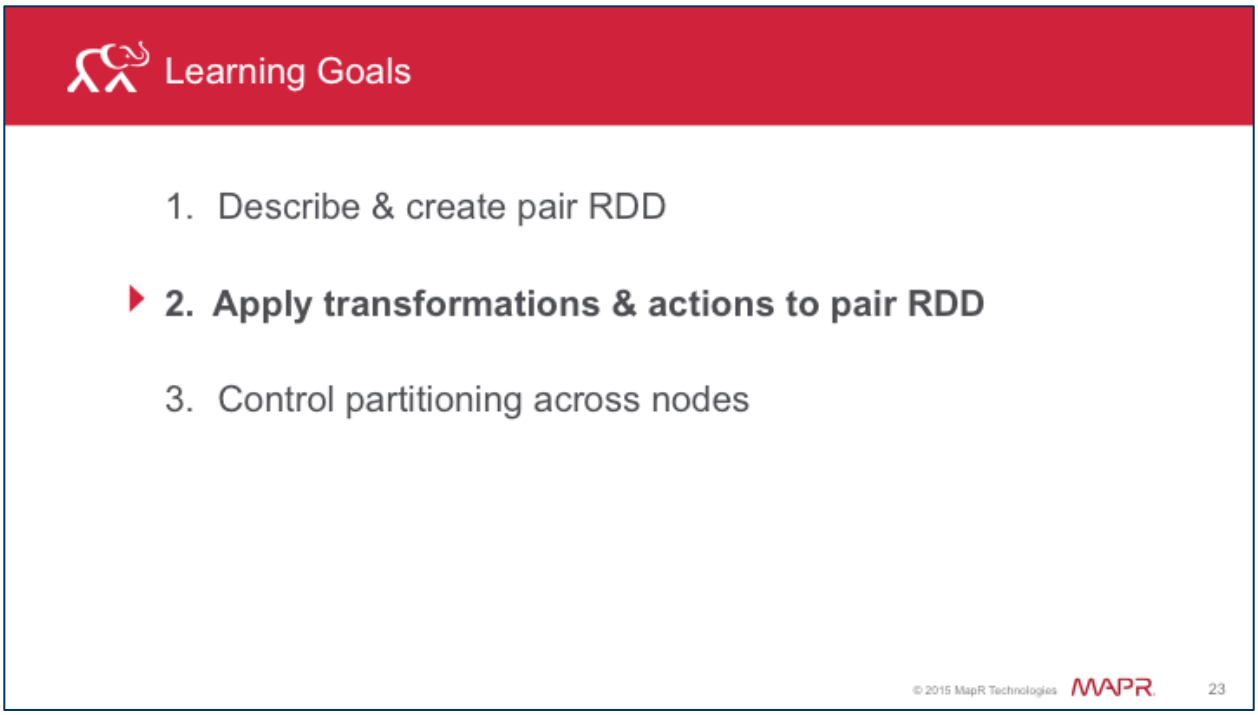
The first step is to create a pairRDD from the sfpd RDD.

We can do this by applying the map transformation on sfpd RDD. The map transformation results in pair tuples consisting of the PdDistrict and a count of 1 for each element (incident) in the sfpd RDD that occurs in that PdDistrict.

In the next section we take a look at some of the transformations and actions that we can use on pair RDD to get the final answer to the question.

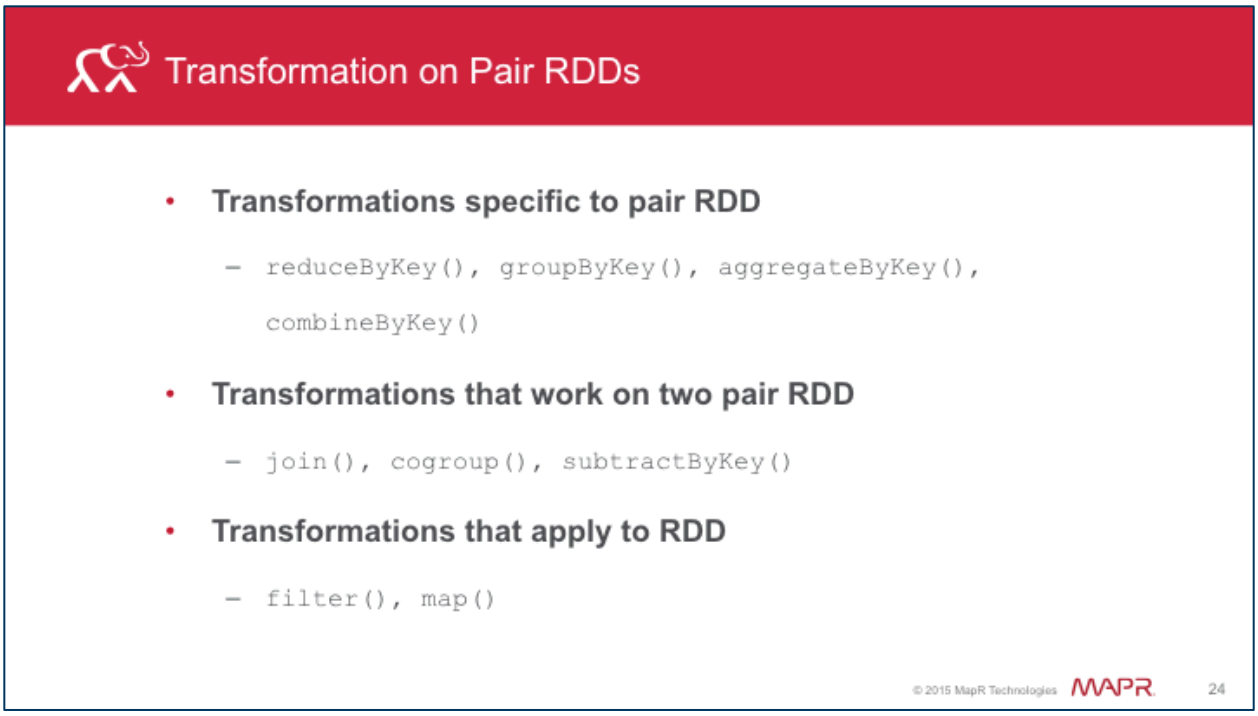


Notes on Slide 22:
Answer: 2



Notes on Slide 23:

In this section we see how to apply transformations and actions to pair RDDs.



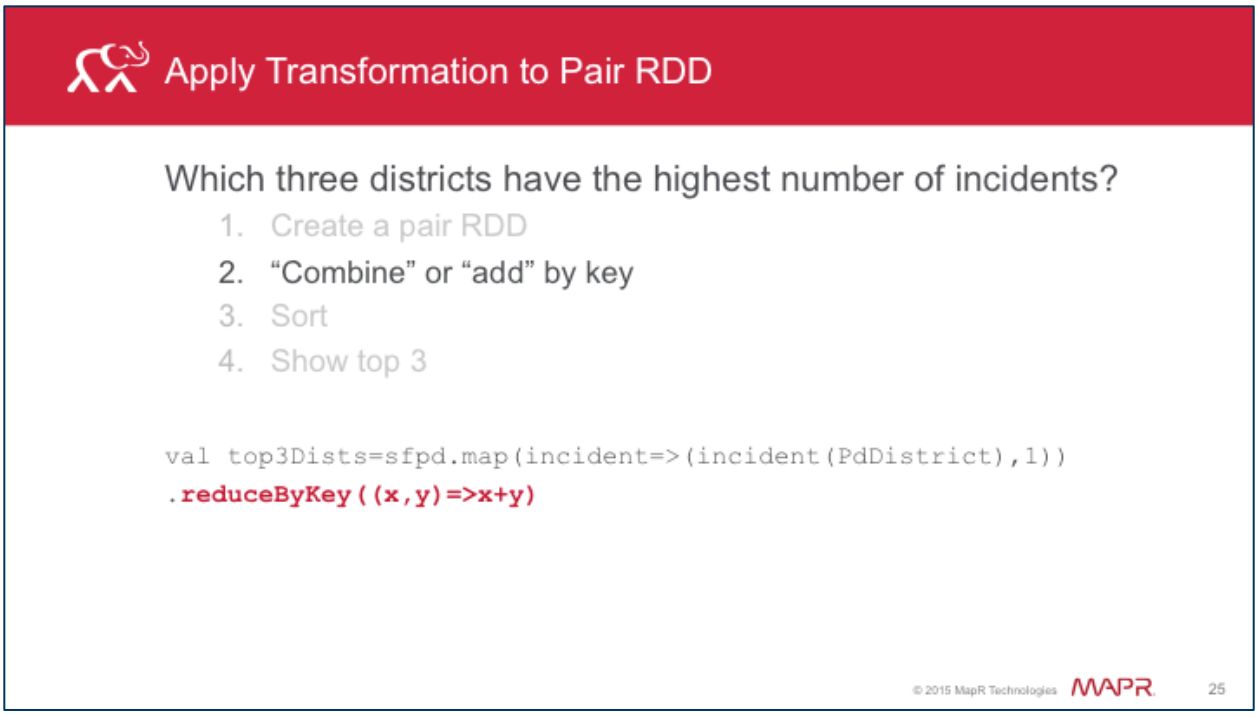
Notes on Slide 24:

When you create a pair RDD, Spark automatically adds a number of additional methods that are specific to pair RDDs such as reduceByKey, groupByKey, combineByKey.

The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.

There are also transformations that work on two pair RDDs such as join, cogroup, subtractByKey.

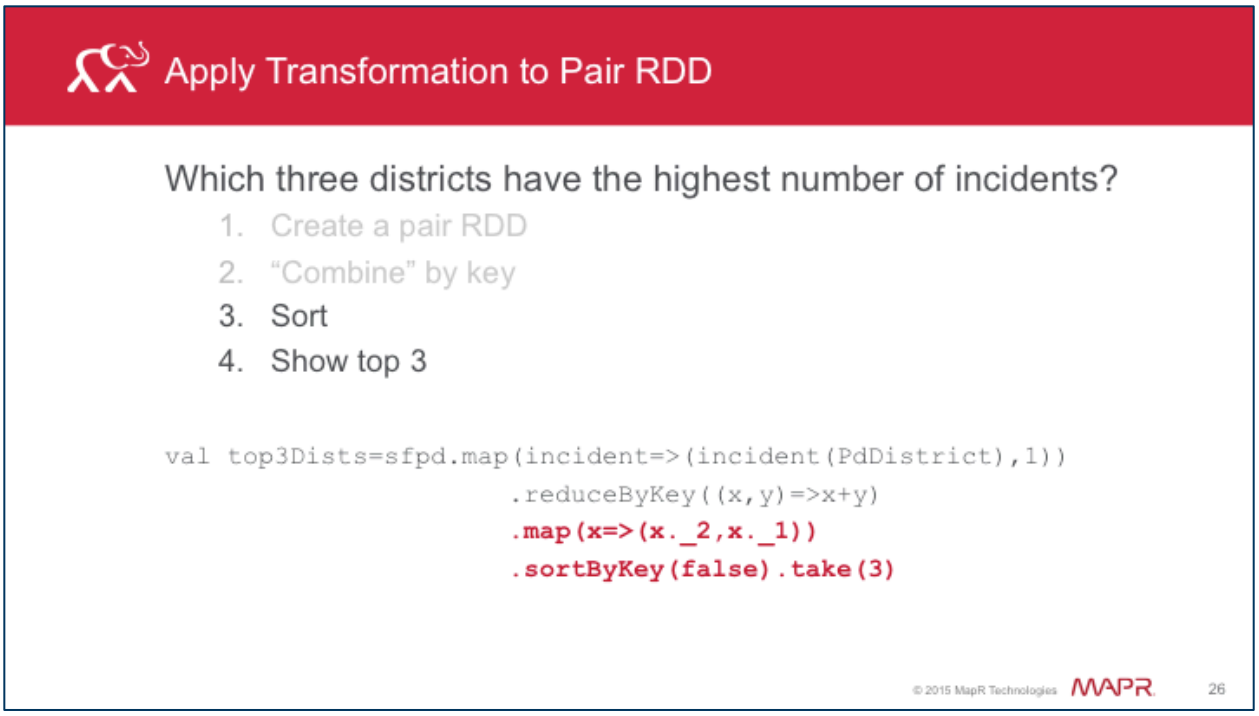
Pair RDDs are a type of RDDs and therefore also support the same operations as other RDDs such as filter() and map().



Notes on Slide 25:

We usually want to find statistics across elements with the same key. reduceByKey is an example of such a transformation. It runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key. It returns a new RDD consisting of each key and the reduced value for that key.

In this example, use the reduceByKey to apply a simple sum to the pair RDD consisting of pair tuples (district, 1). It returns a new RDD with the reduced value (in this case, the sum) for each district.



Notes on Slide 26:

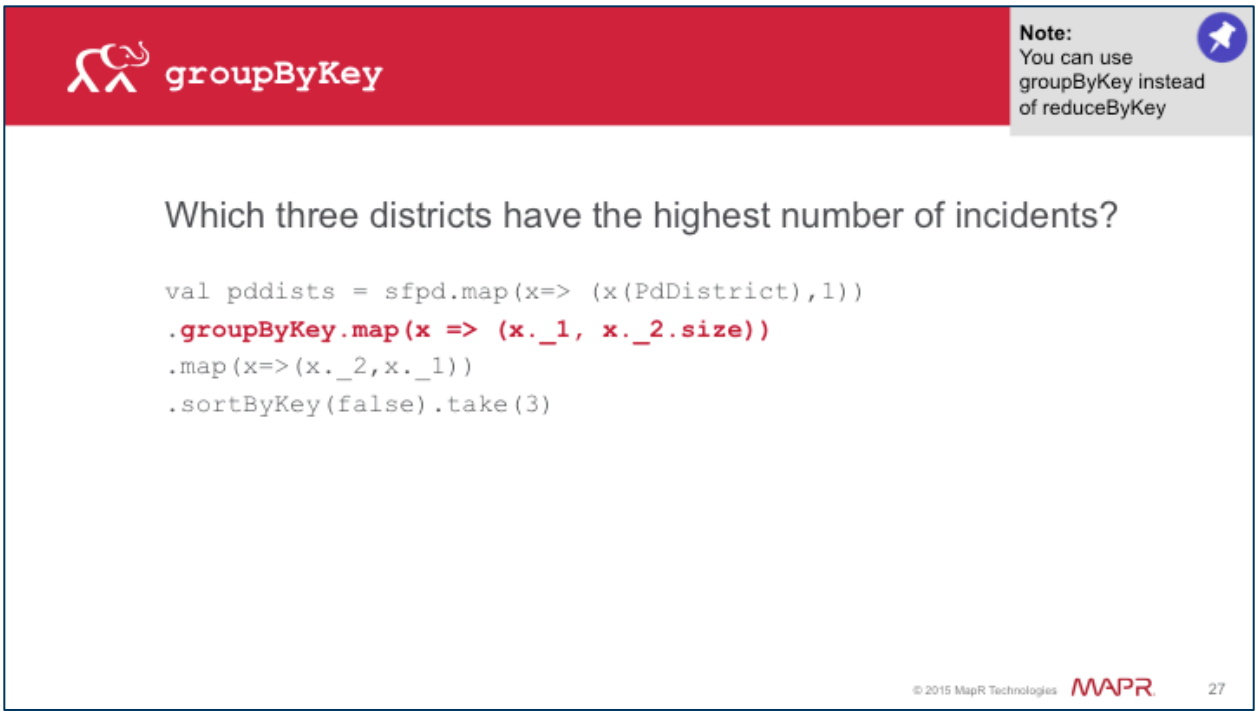
We can sort an RDD with key/value pairs, provided that there is an ordering defined on the key. Once we have sorted our data, any future calls on the sorted data to collect() or save() will result in ordered data. The sortByKey() function takes a parameter called ascending, that indicates that the results are in ascending order (by default set to true).

In our example, we apply another map operation to the results of the reduceByKey. This map operation is switching the order of the tuple – we swap district and the sum to give us a dataset of tuples(sum, district). We then apply sortByKey to this dataset. The sorting is then done on the sum which is the number of incidents for each district.

Since we want the top 3, we need the results to be in descending order. We pass in the value "false" to the "ascending" parameter to sortByKey. To return three elements, we use take(3) – this will give us the top 3 elements.

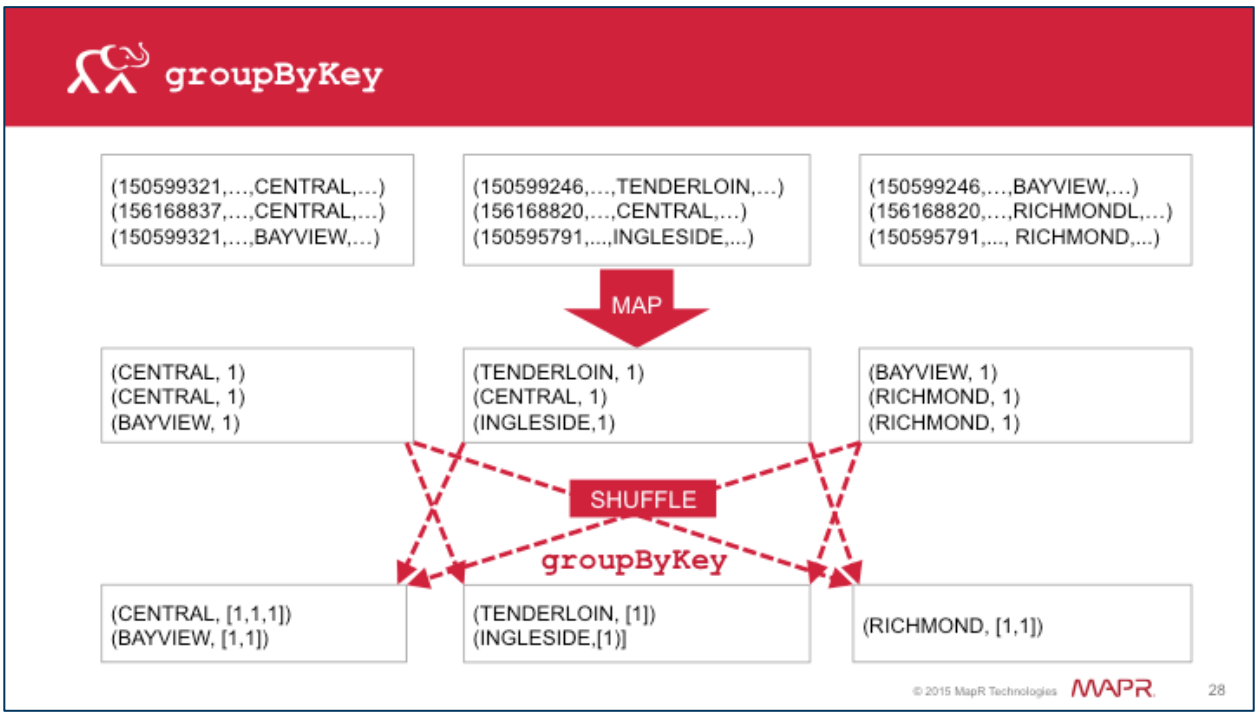
You can also use:

val top3 Dists= sfpd.map(incident=>(incident(PdDistrict),1)).reduceByKey(_ + _).top(3)(Ordering.by(_._2))



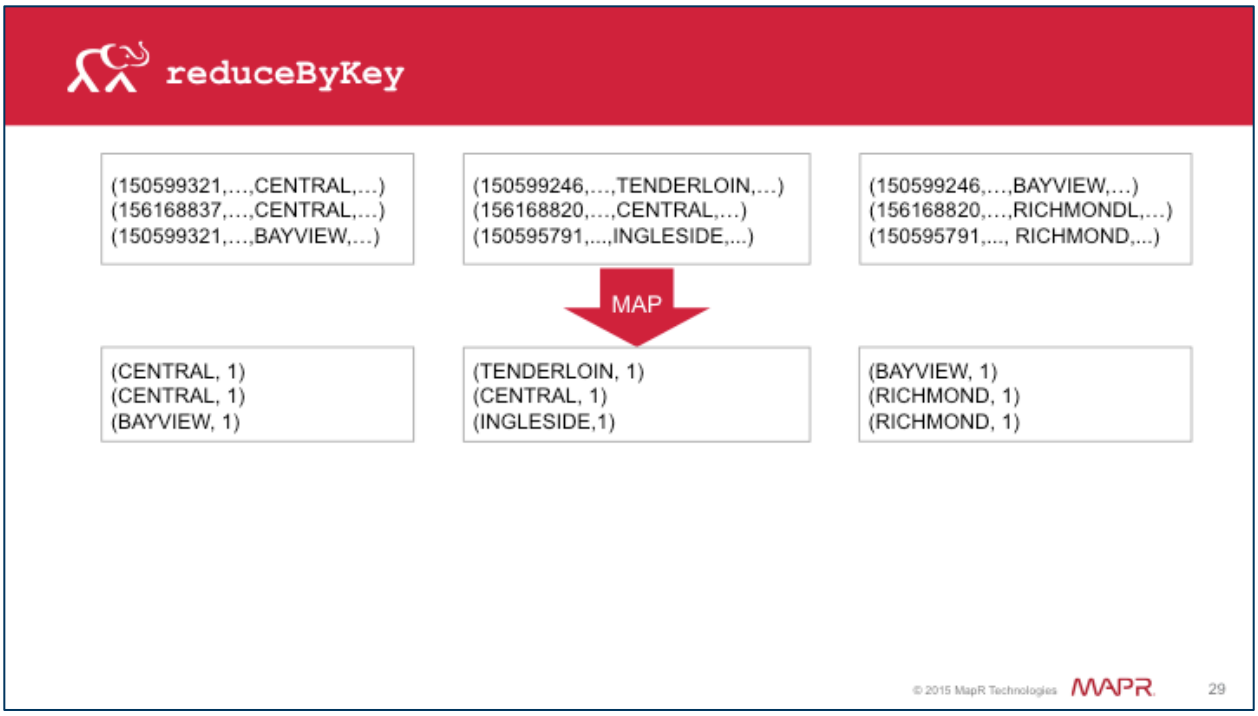
Notes on Slide 27:

This is another way to answer the question:
What are the top 3 districts with the max number of incidents?
You can use groupByKey instead of reduceByKey.
It groups all values that have the same key. As a result, it takes no argument. It results in a dataset of key/value pairs where the value is an iterable list.
Here we apply the groupByKey to the dataset consisting of the tuples (district, 1). The map transformation highlighted here is applied to every element of the result of the groupByKey transformation and it returns the district and the size of the iterable list.
Next we will see the difference between using groupByKey versus using reduceByKey.
You can also use: `val pddists = sfpd.map(x=> (x(PdDistrict),1)).groupByKey.map(x => (x._1, x._2.size)).top(3)(Ordering.by(_._2))`



Notes on Slide 28:

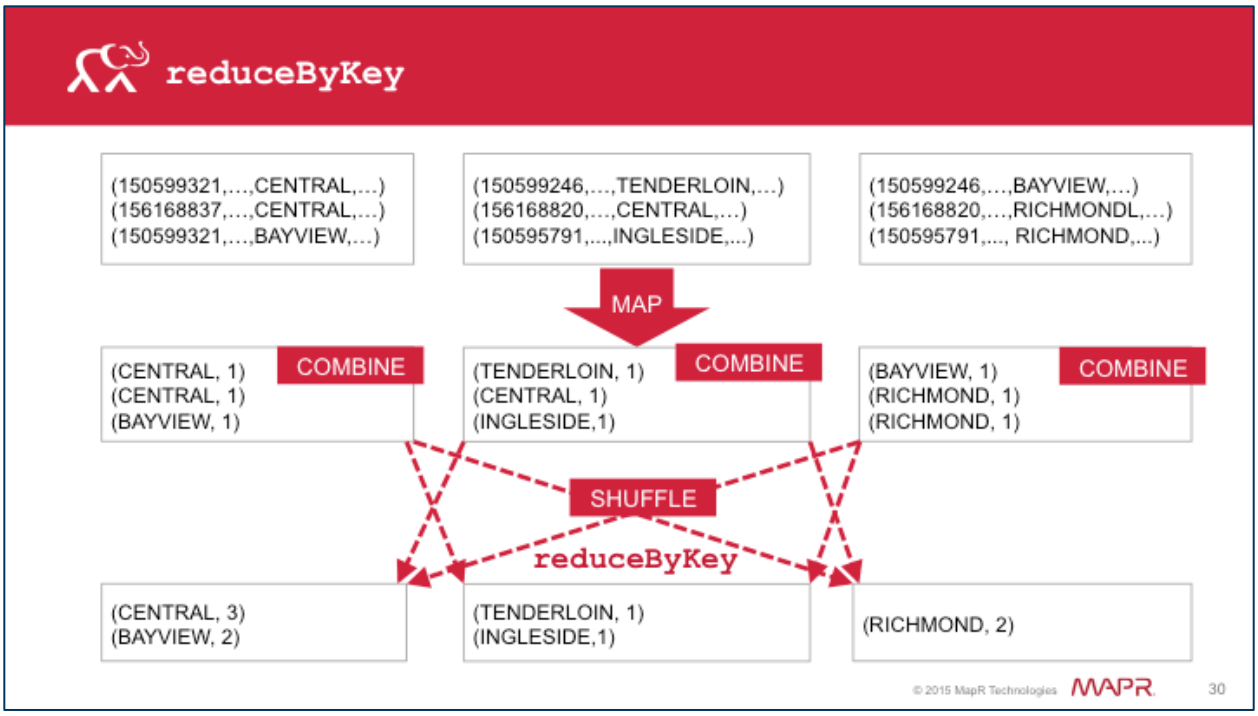
The first map transformation results in a pairRDD consisting of pairs of District name and 1.
The groupByKey groups all values that have the same key resulting in a key-iterable value pair.
It groups all values with the same key onto the same machine.
groupByKey results in one key-value pair per key. This single key-value pair cannot span across multiple worker nodes.
When handling large datasets, groupByKey causes a lot of unnecessary transfer of data over the network. When there is more data shuffled onto a single executor machine than can fit in memory, Spark spills data to disk. However, it flushes out the data to disk one key at a time. Thus, if a single key has more key-value pairs than can fit in memory, there will be an out of memory exception.



Notes on Slide 29:

reduceByKey combines values that have the same key .

The first map transformation results in a pairRDD consisting of pairs of District name and 1.



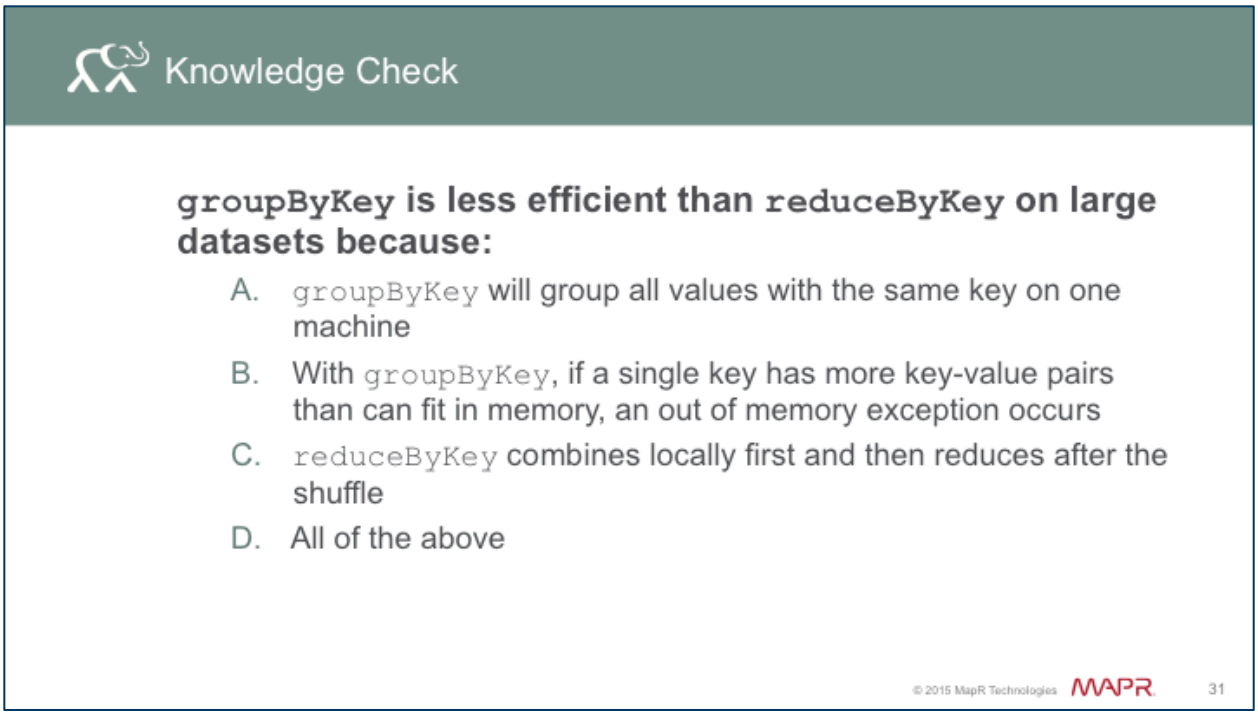
Notes on Slide 30:

reduceByKey will automatically perform a combine locally on each machine. The user does not have to specify a combiner. The data is reduced again after the shuffle.

reduceByKey can be thought of as a combination of groupByKey and a reduce on all the values per key. It is more efficient though, than using each separately. With reduceByKey, data is combined so each partition outputs at most one value for each key to send over the network resulting in better performance. In general, reduceByKey is more efficient especially for large datasets.

Not all problems that can be solved by groupByKey can be computed using reduceByKey. This is becasue reduceByKey requires combining all values into another value with the exact same type.

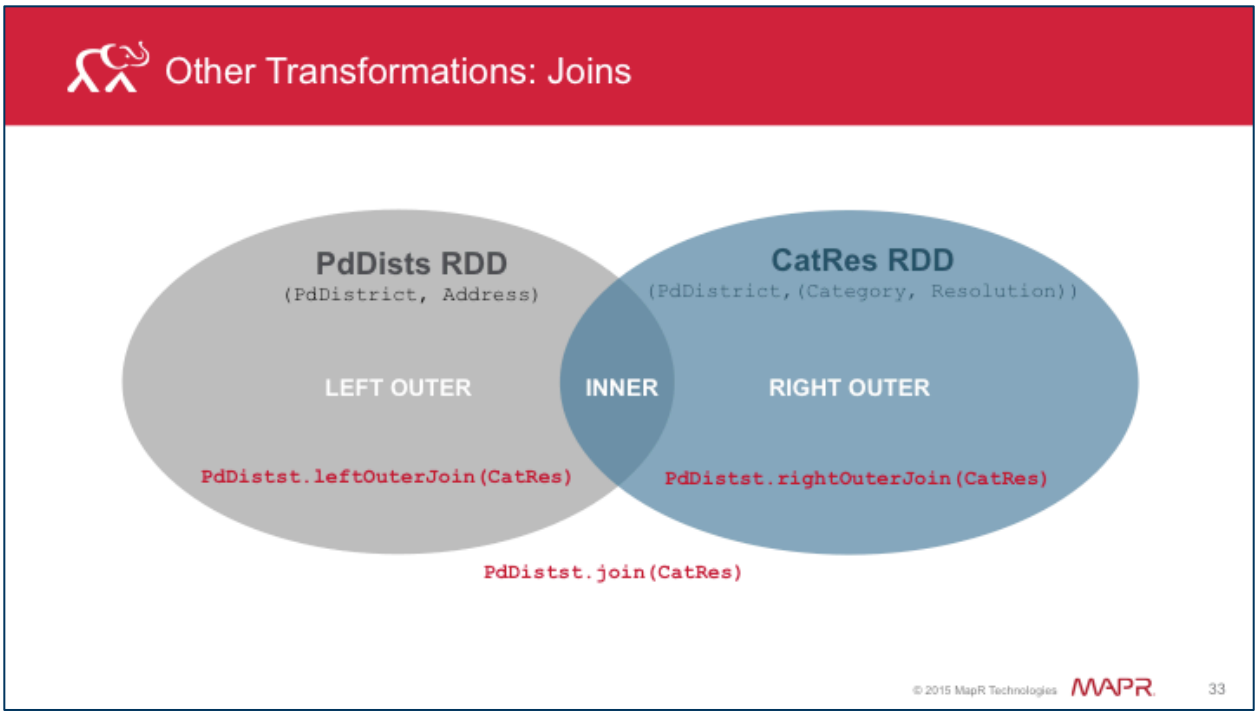
http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html



Notes on Slide 31:
Answer: 4

Notes on Slide 32:

In this activity you will create Pair RDD and apply actions and transformations on Pair RDD.

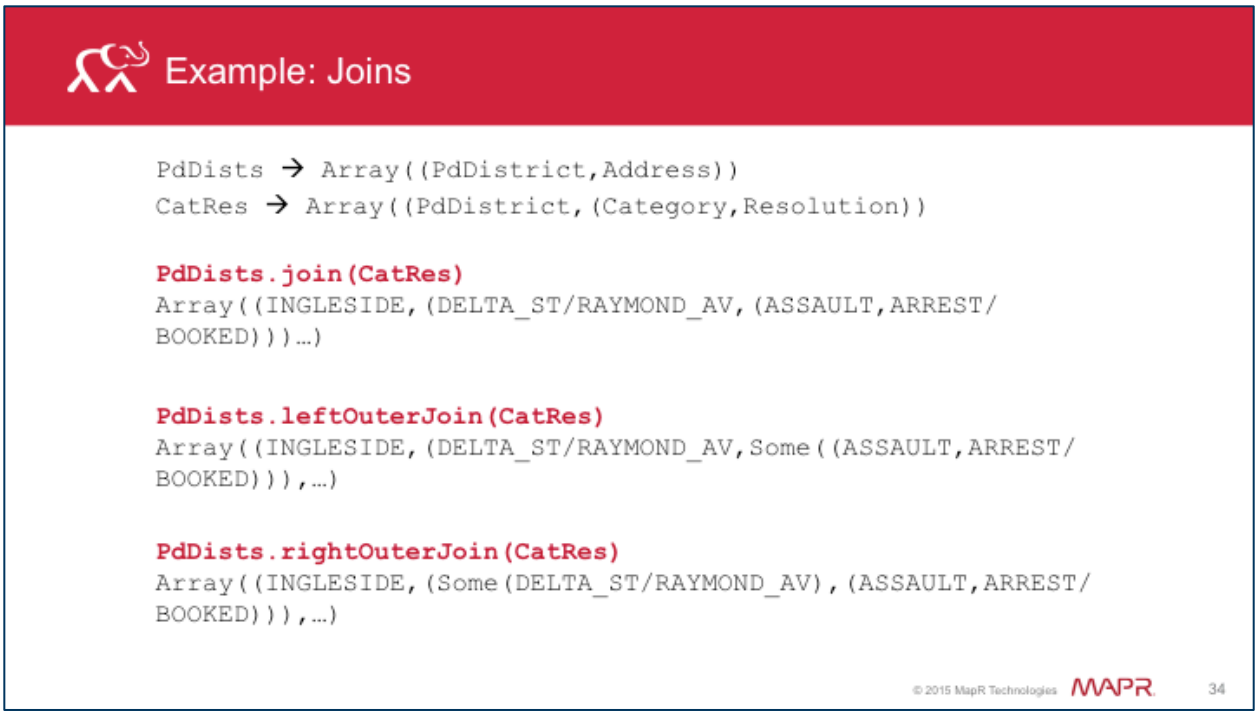


Notes on Slide 33:

Some of the most useful operations that we can use with key-value pairs is using it with other similar data. One of the most common operations on a pair RDD is to join two pair RDD. You can do right and left outer joins, cross joins, and inner joins.

You can join two datasets that have been loaded into separate RDDs by joining the resulting RDDs.

- Join is the same as an inner join, where only keys that are present in both RDDs are given in the output.
- leftOuterJoin results in a pair RDD that has entries for each key from the source RDD, joining values from just these keys included in the other RDD.
- rightOuterJoin is similar to the the leftOuterJoin but results in a pair RDD that has entries for each key in the other pair RDD, joining values from these keys included in the source RDD.



Notes on Slide 34:

In this example, PdDists is a pair RDD consisting of the pairs where the key is PdDistrict and the value is the address.

CatRes is another pair RDD with the same key – PdDistrict and the value is a tuple consisting of the Category and the Resolution.

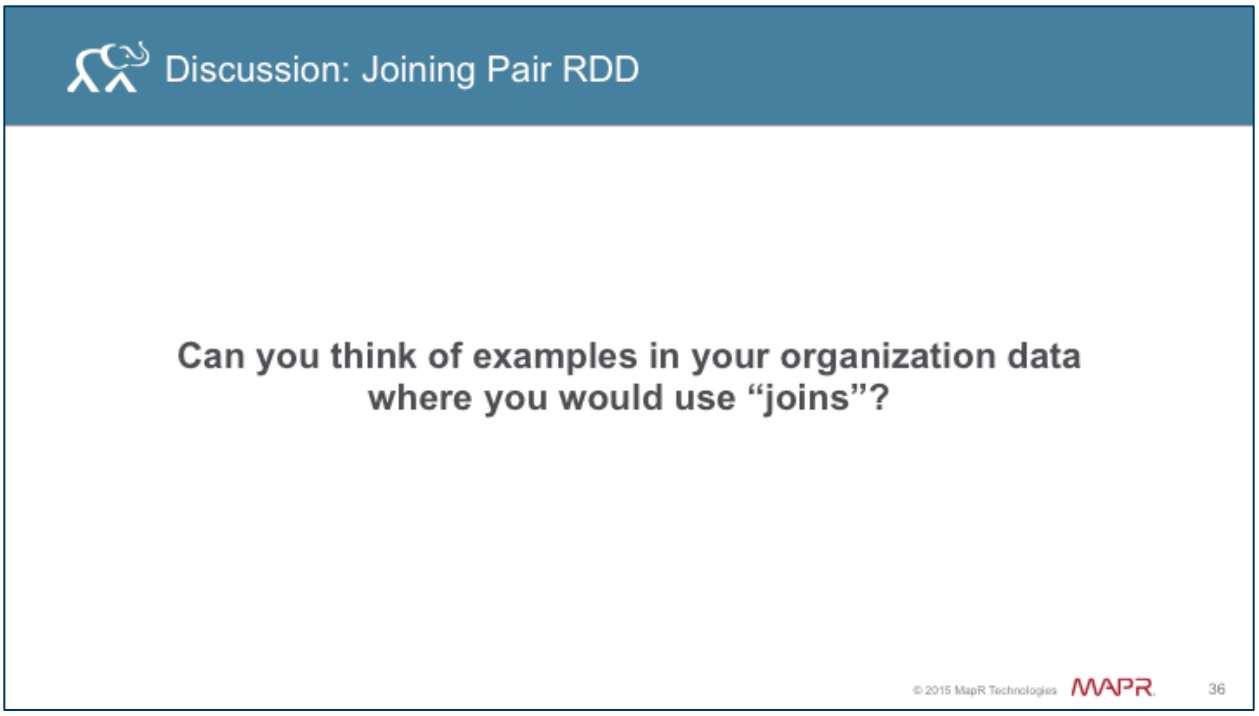
Since the key – PdDistrict- is present in both RDDs the output of the join contains all the records as shown here. The value consists of the tuple from the source RDD and an option for values from the other pair.

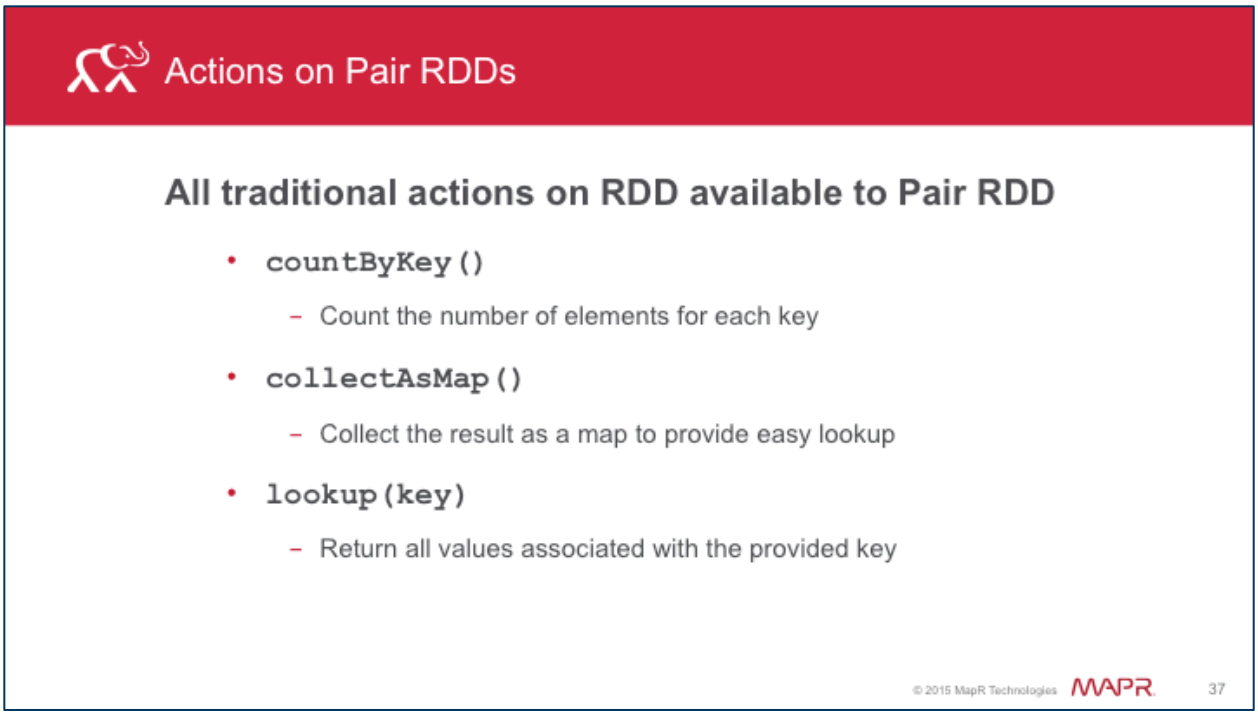
The rightOuterJoin returns pairs with the keys being the key from the other RDD (CatRes) which is the same.

As you can see here, the results in all three cases are the same because both RDD have the same set of keys.

Notes on Slide 35:

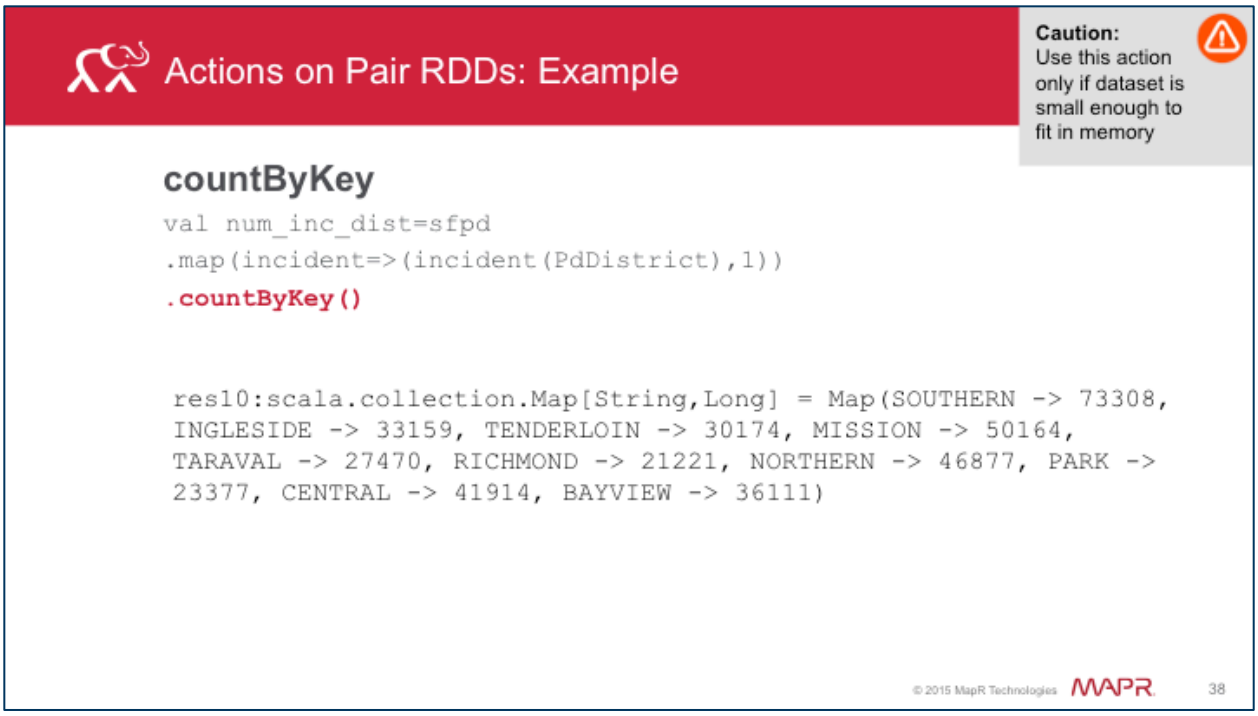
In this example, the RDD PdDists is the same as before. We have another pair RDD with IncidntNum as the key. The value is the tuple consisting of Category, Descript and Resolution.
The join returns an empty collection since the RDDs do not have any keys in common.
The leftOuterJoin returns pairs with the key being the key from the source RDD and the value consisting of the tuple from the source RDD and an option for values from the other pair.
The rightOuterJoin returns pairs with the keys being the key from the other RDD (IncCatRes).





Notes on Slide 37:
All traditional actions on RDDs are available to use on Pair RDD. In addition, the following actions are available just for pair RDD:
countByKey() will Count the number of elements for each key

collectAsMap() Collects the result as a map to provide for easy lookup
lookup(key) Return all values associated with the provided key.
There are more actions that can be performed on Pair RDDs. Refer to the API documentation.



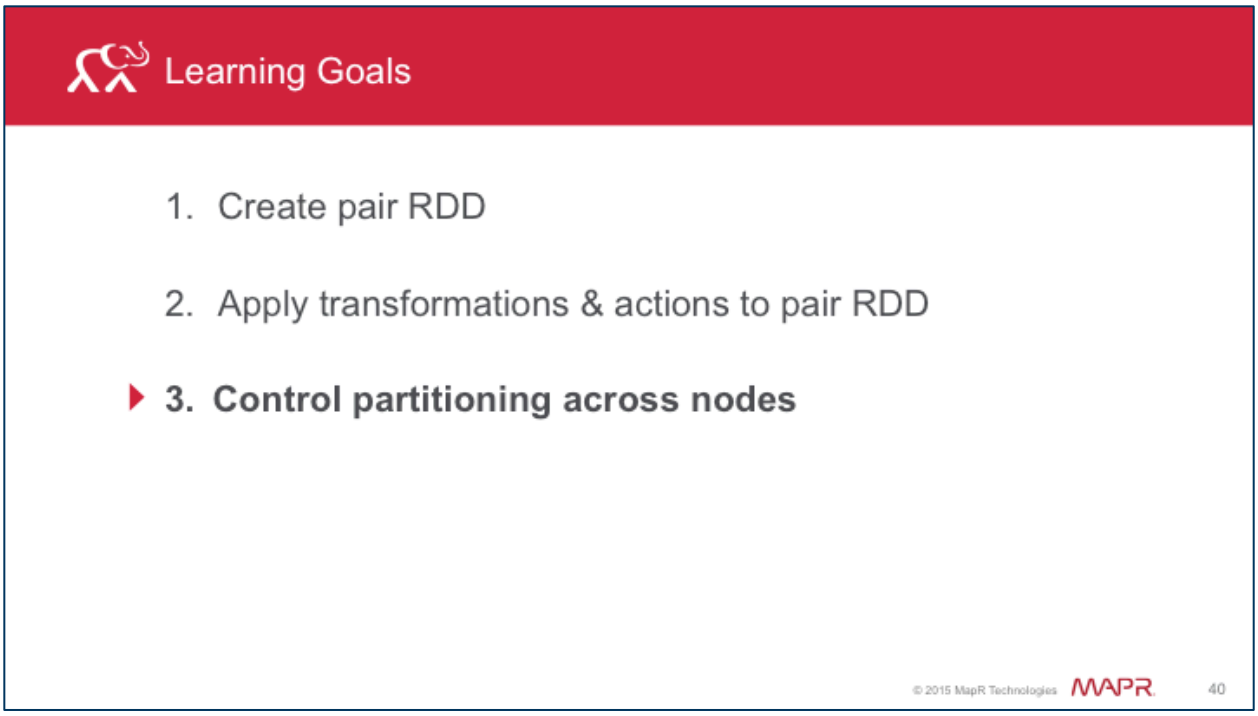
Notes on Slide 38:

Example - if we just want to return number of incidents by district:, we can use countByKey on the pair RDD consisting of pair tuples (district, 1).

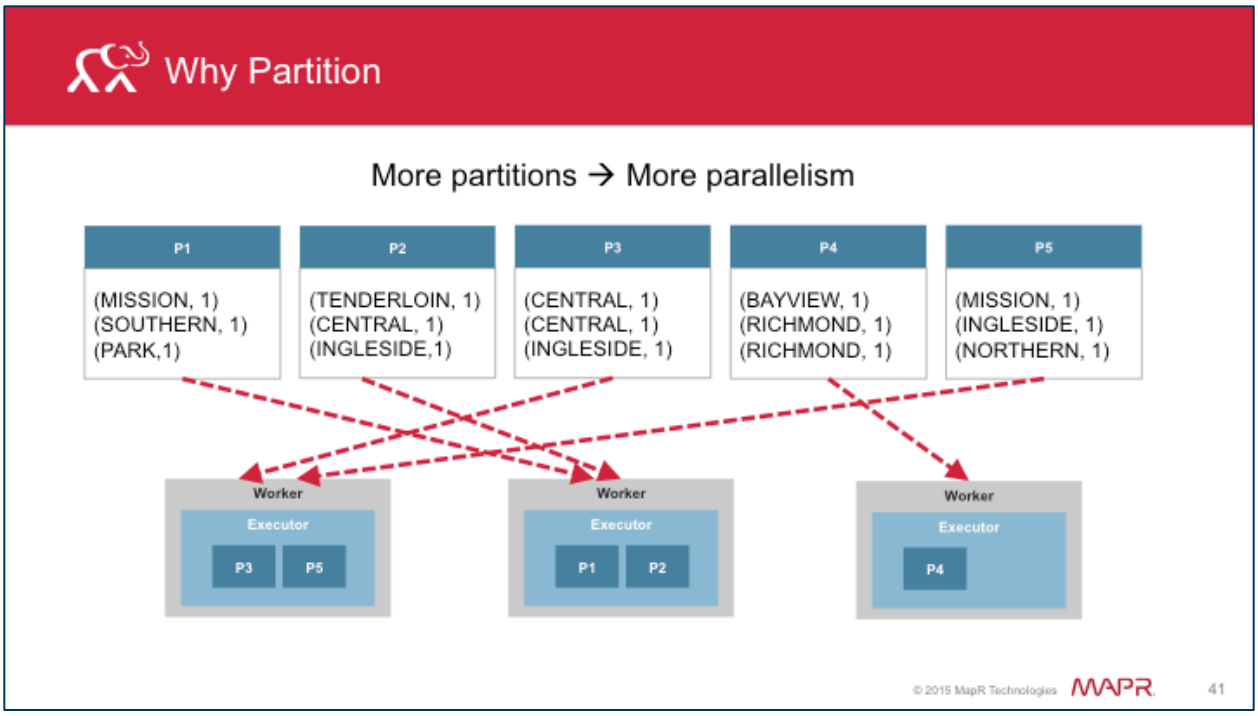
Use this operation only if the dataset size is small enough to fit in memory.

Notes on Slide 39:

In this activity you will join Pair RDD.



Notes on Slide 40:
In this section, we will look at the partitioning of RDDs and how to control the partitioning of RDDs across nodes.

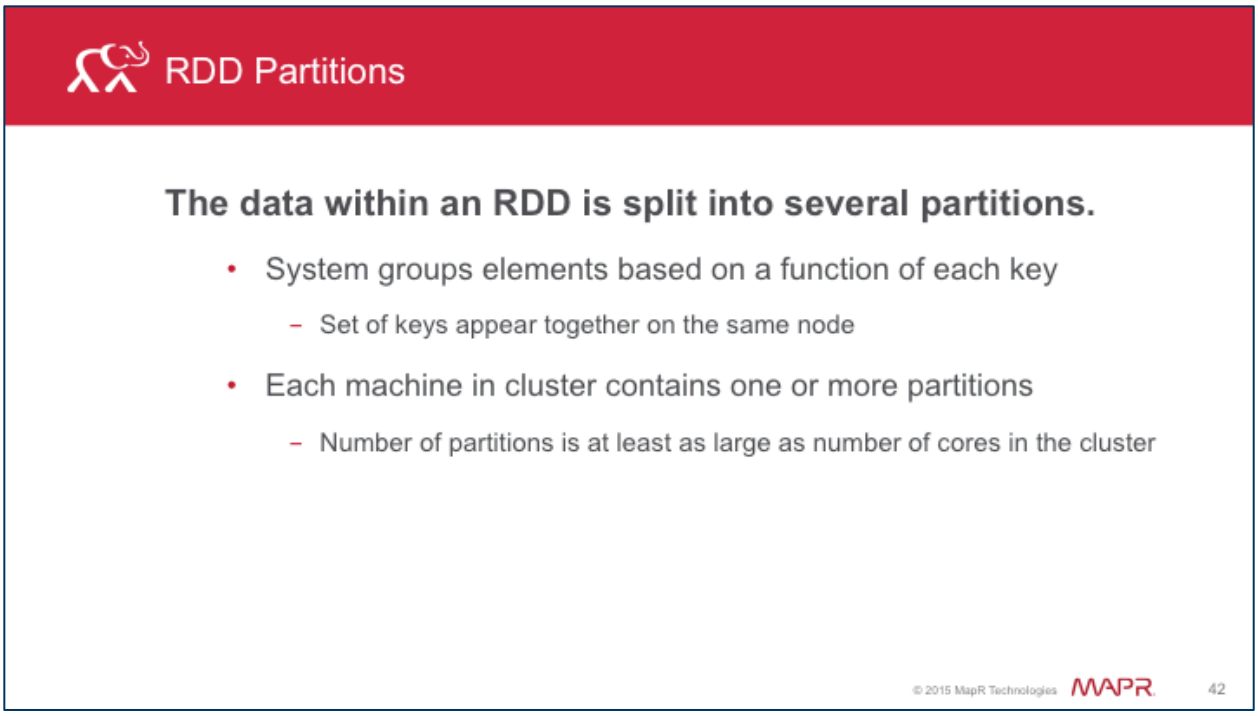


Notes on Slide 41:

In a distributed environment, how your data is laid out can affect performance. A data layout that minimizes network traffic can significantly improve performance.

The data in a Spark RDD is split into several partitions. We can control the partitioning of RDD in our Spark program to improve performance.

Note that partitioning is not necessarily helpful in all applications. If you have a dataset that is reused multiple times, it is useful to partition. However, if the RDD is scanned only once, then there is no need to partition the dataset ahead of time.



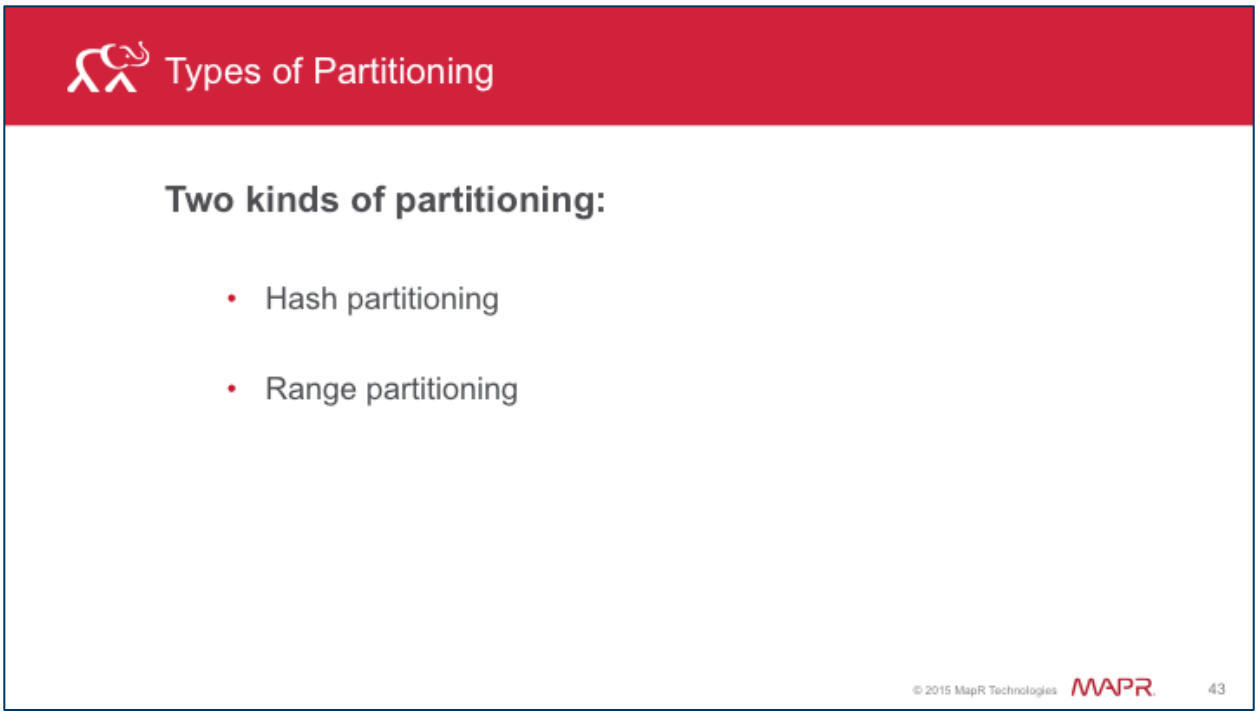
Notes on Slide 42:

The data within an RDD is split into several partitions.

Spark's partitioning is available on all RDDs of key-value pairs.

The system groups elements based on a function of each key. Functions include hash or range partitions. Set of keys will appear together on a node. Spark will group all keys from the same set on the same node.

Each machine in the cluster contains one or more partitions. The number of partitions is at least as large as the number of cores in the cluster

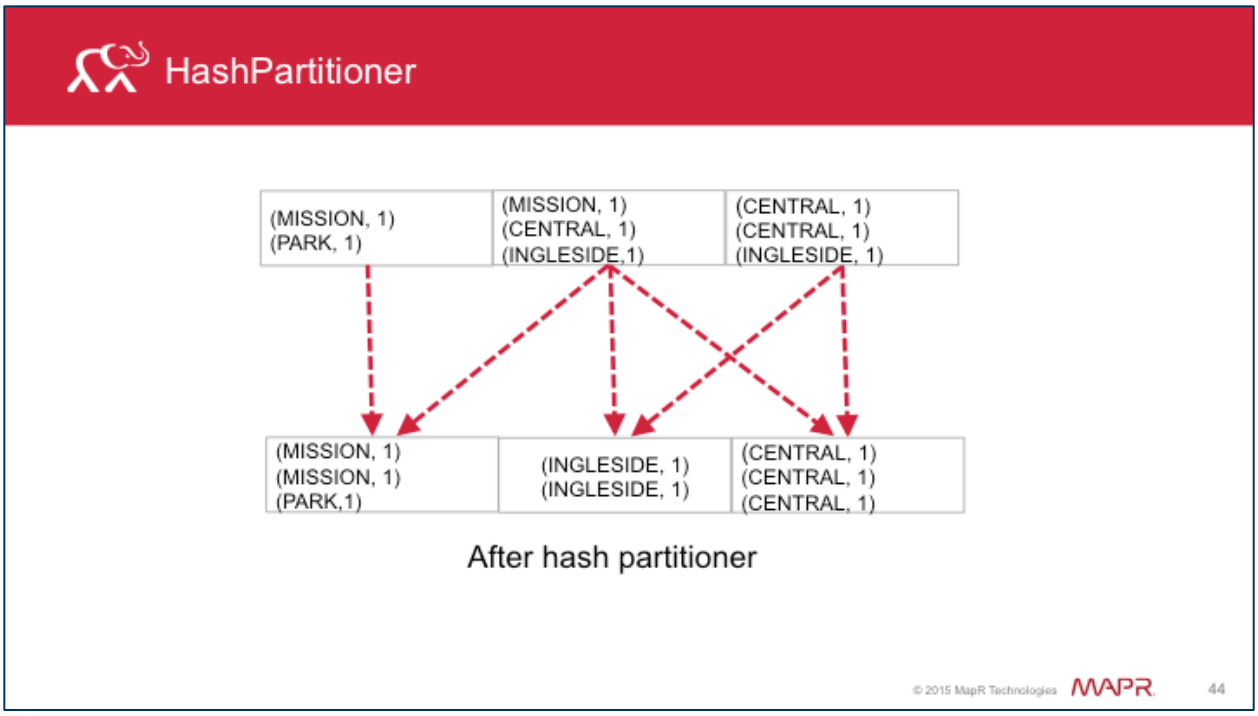


Notes on Slide 43:

There are two kinds of partitioning available in Spark:

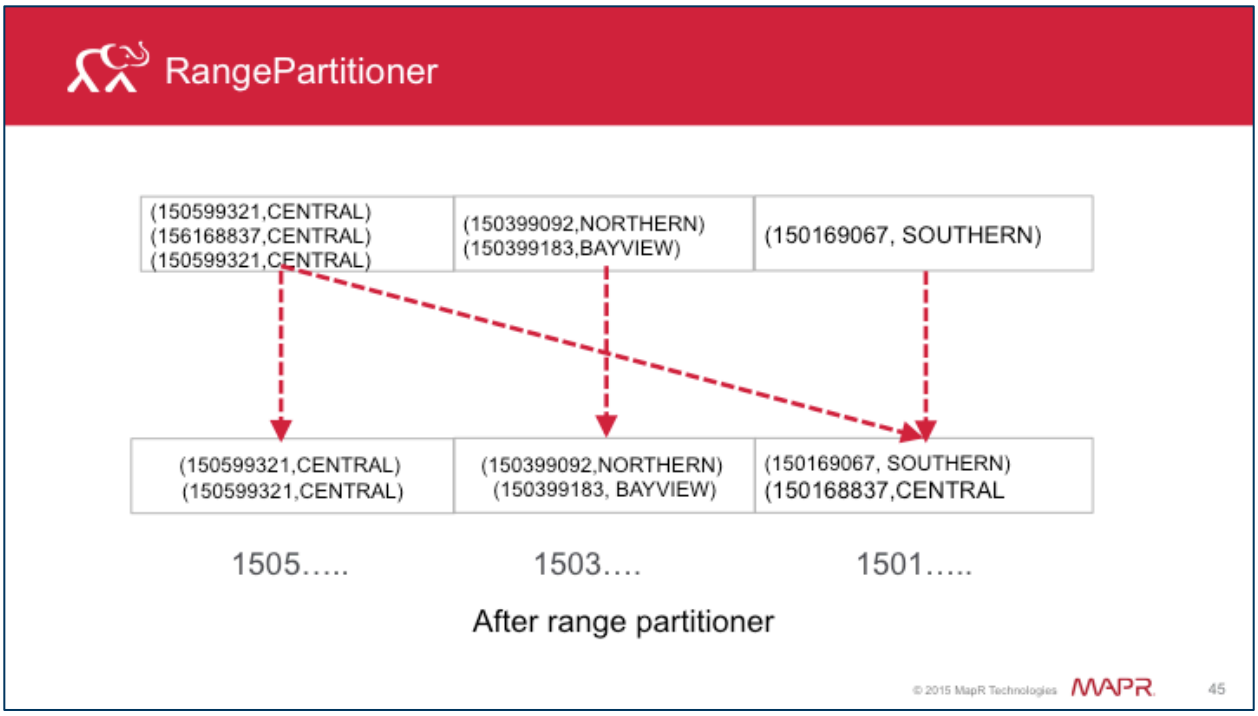
1. Hash partitioning and
2. Range partitioning

Customizing partitioning is only possible on Pair RDDs.



Notes on Slide 44:

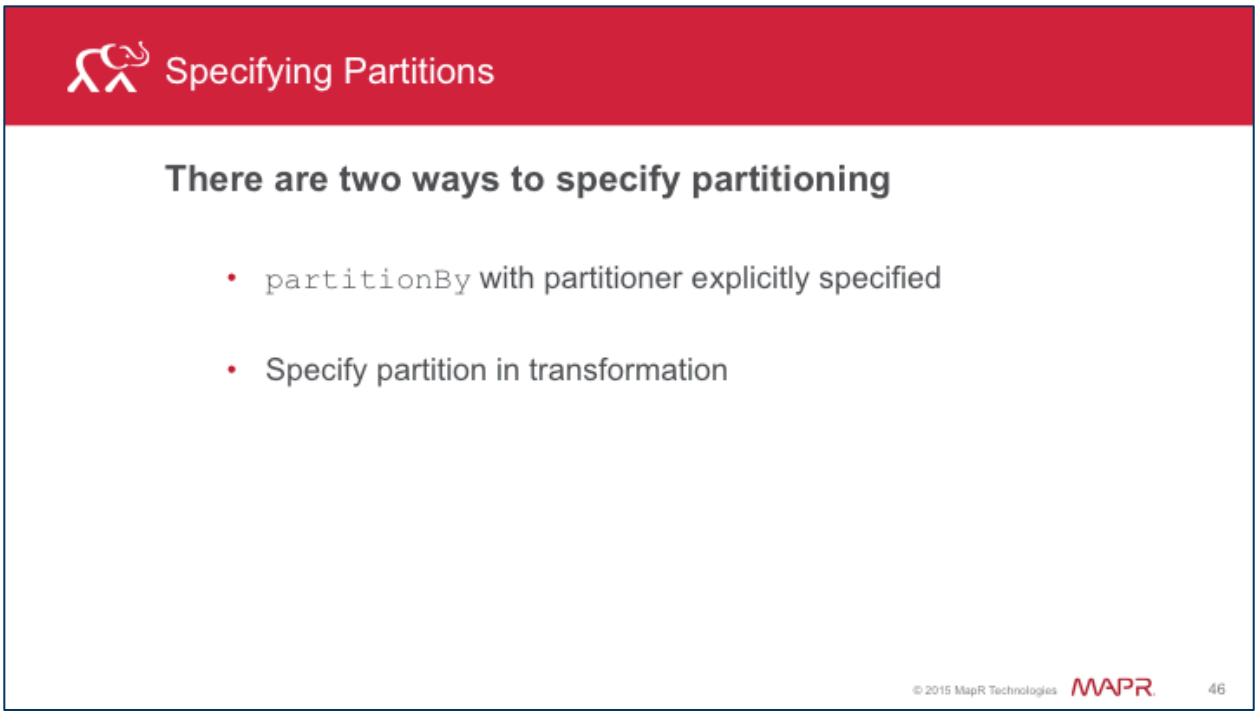
You can apply the hash partitioner to an RDD using the partitionBy transformation at the start of a program. The hash partitioner will shuffle all the data with the same key to the same worker. In this example, the data for the same keys have been shuffled to the same workers.



Notes on Slide 45:

The range partitioner will shuffle the data for keys in a range to the same workers as shown in this example.

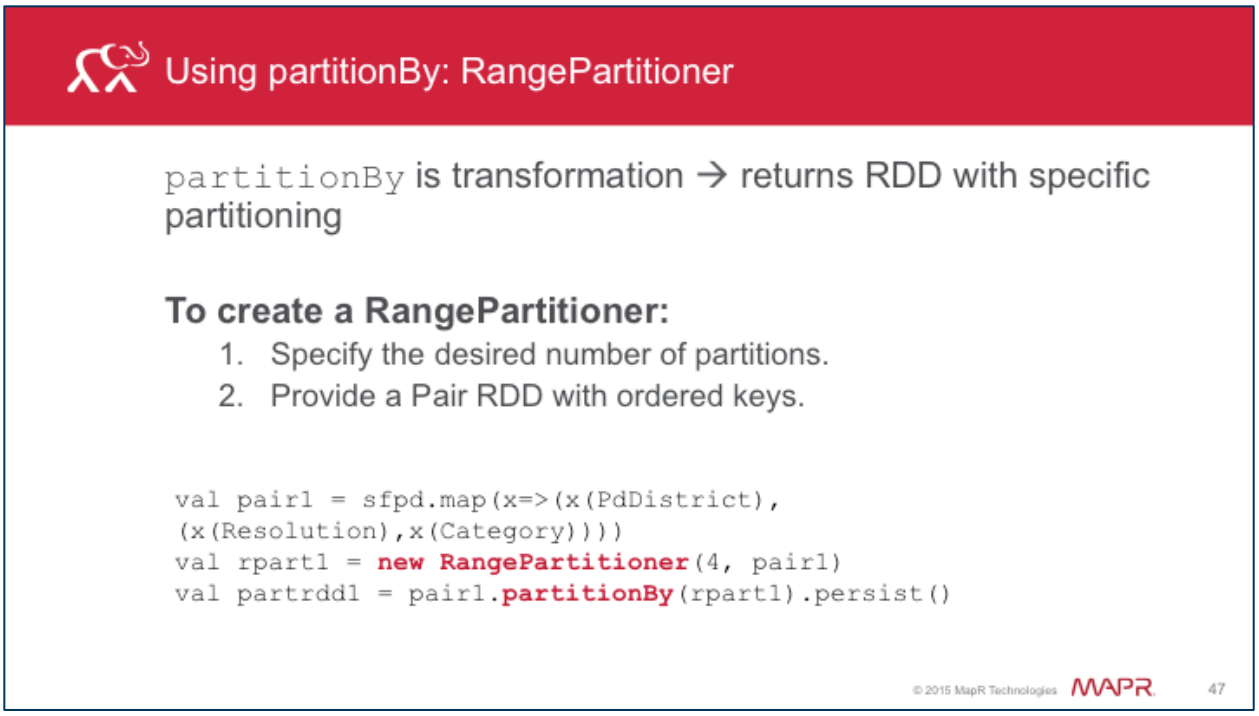
If you have a pair RDD that contains keys with an ordering defined, use a range partitioner. The range partitioner will partition the keys based on the order of the keys and the range specified. It results in tuples with keys in the same range on the same machine.



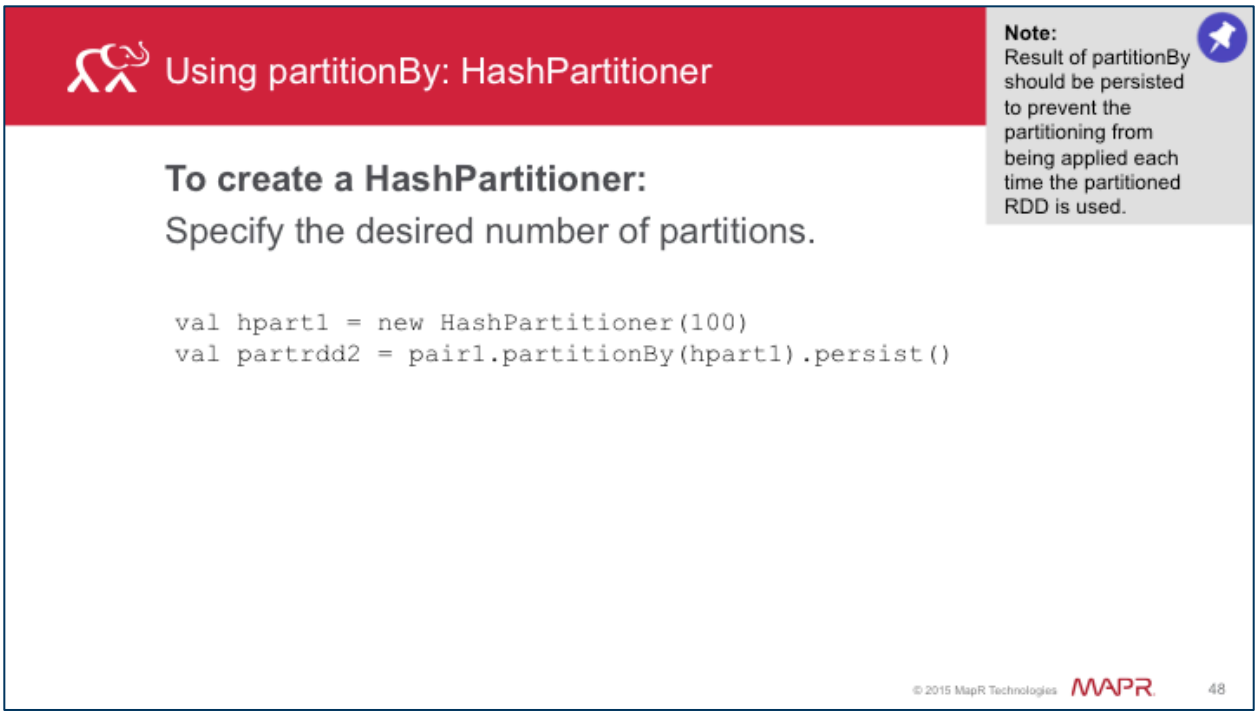
Notes on Slide 46:

There are two ways to create RDDs with specific partitioning:

1. Call partitionBy on an RDD, providing an explicit Partitioner.
2. Specify partitions in transformations –this will return RDDs partitioned

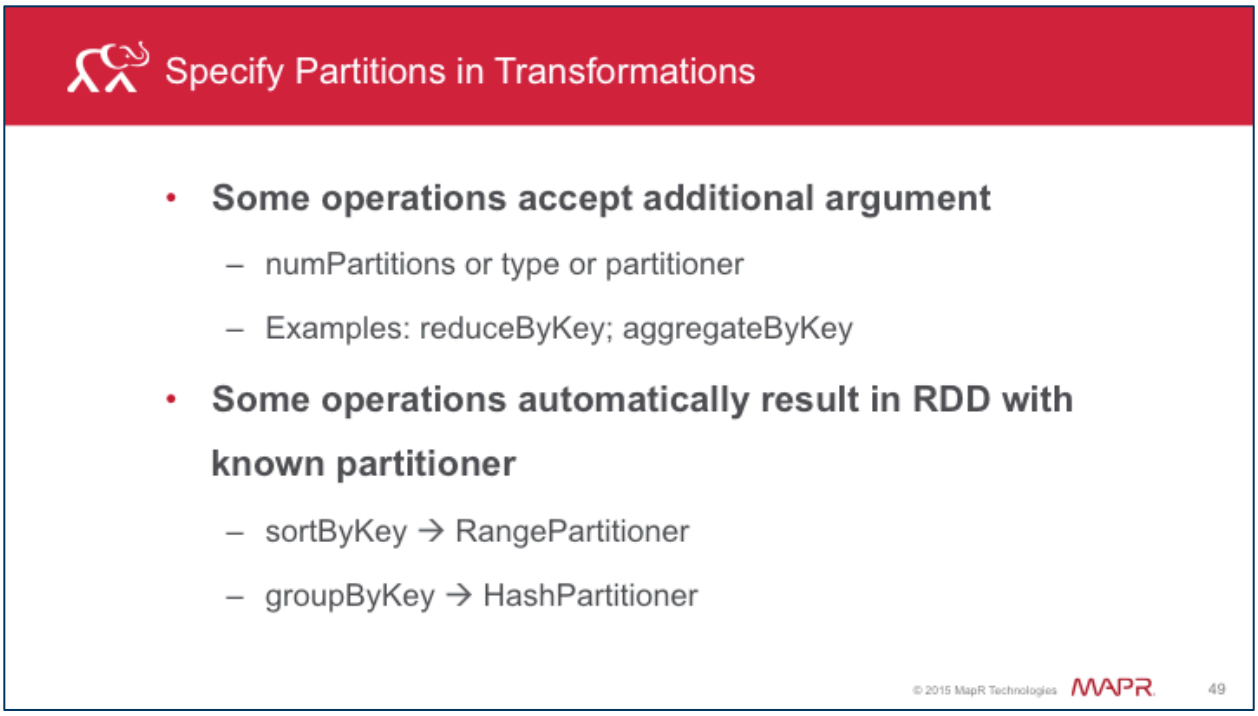


Notes on Slide 47:
partitionBy is a transformation and creates an RDD with a specified partitioner.
To create a RangePartitioner:
1. Specify the desired number of partitions.
2. Provide a Pair RDD with ordered keys.



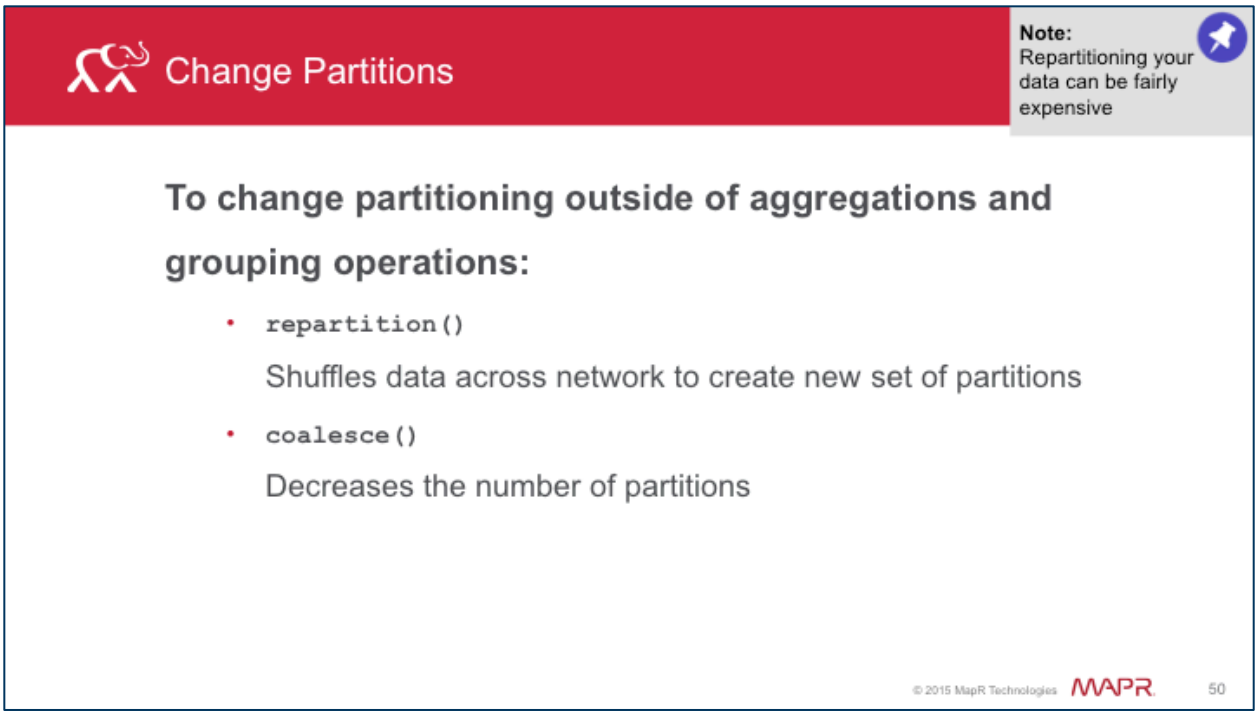
Notes on Slide 48:

HashPartitioner takes a single argument which defines the number of partitions
Values are assigned to partitions using hashCode of keys. If distribution of keys is not uniform you can end up in situations when part of your cluster is idle
Keys have to be hashable.
NOTE: When you use partition, it will create a shuffle. The result of partitionBy should be persisted to prevent reshuffling each time the partitioned RDD is used.

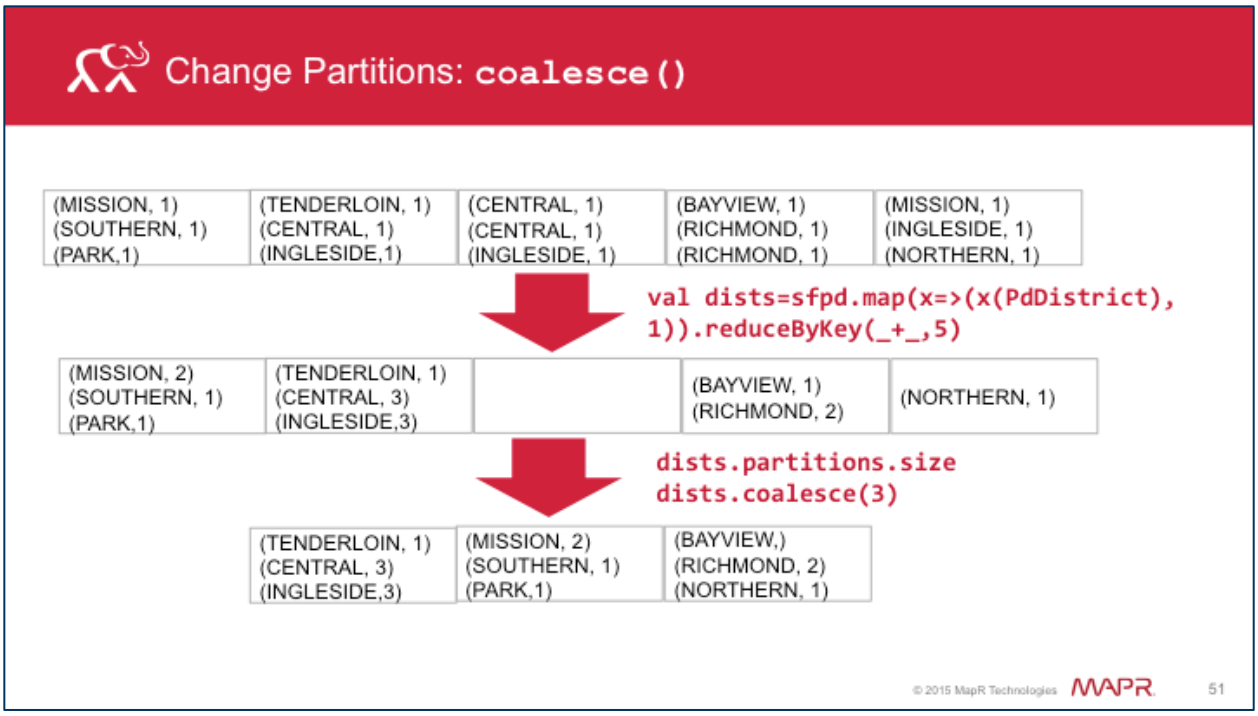


Notes on Slide 49:

A lot of the operations on Pair RDDs accept an additional argument, such as the number of partitions or the type or partitioner.
Some operations on RDDs automatically result in an RDD with a known partitioner.
For example, by default, when using sortByKey, a RangePartitioner is used, and the default partitioner used by groupByKey, is HashPartitioner.



Notes on Slide 50:
To change partitioning outside of aggregations and grouping operations, you can use the repartition() function or coalesce().
repartition(): shuffles data across the network to create a new set of partitions
coalesce(): decreases the number of partitions
NOTE: Repartitioning your data can be fairly expensive since repartitioning shuffles data across the network into new partitions.

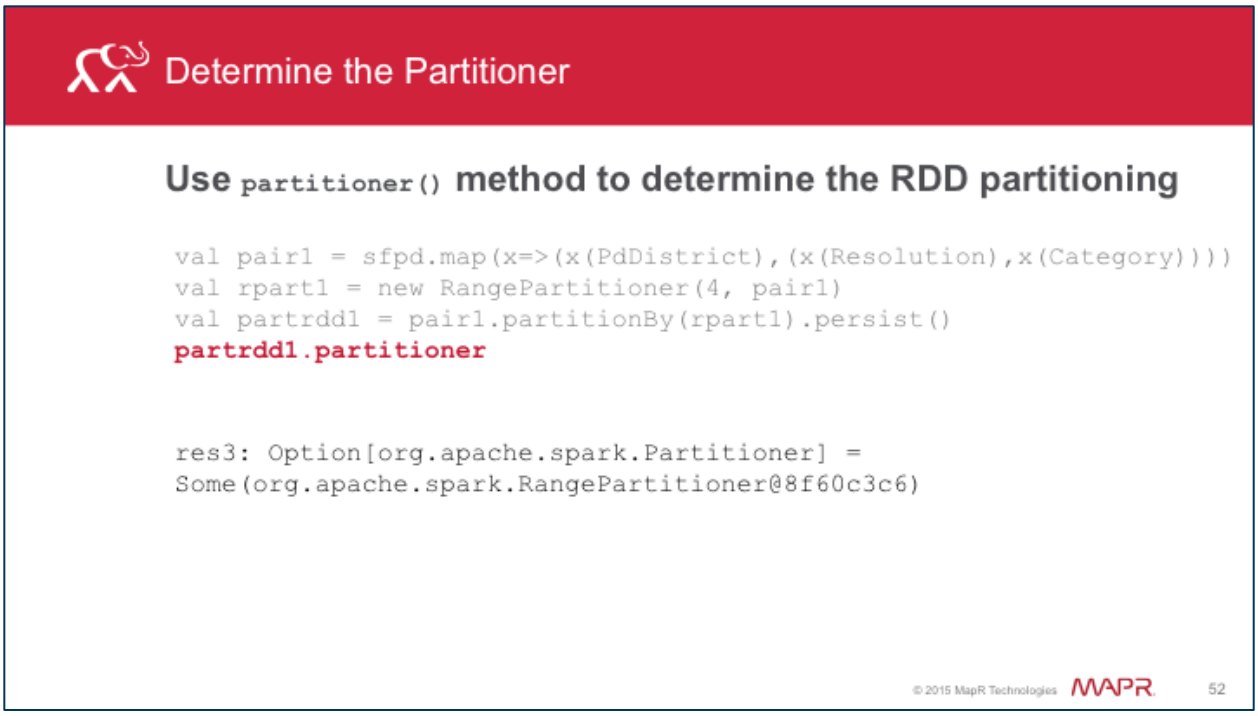


Notes on Slide 51:

You need to make sure that when using coalesce, you specify fewer number of partitions. Use rdd.partition.size() to determine the current number of partitions.

In this example, we apply the reduceByKey to the pair RDD specifying the number of partitions as a parameter.

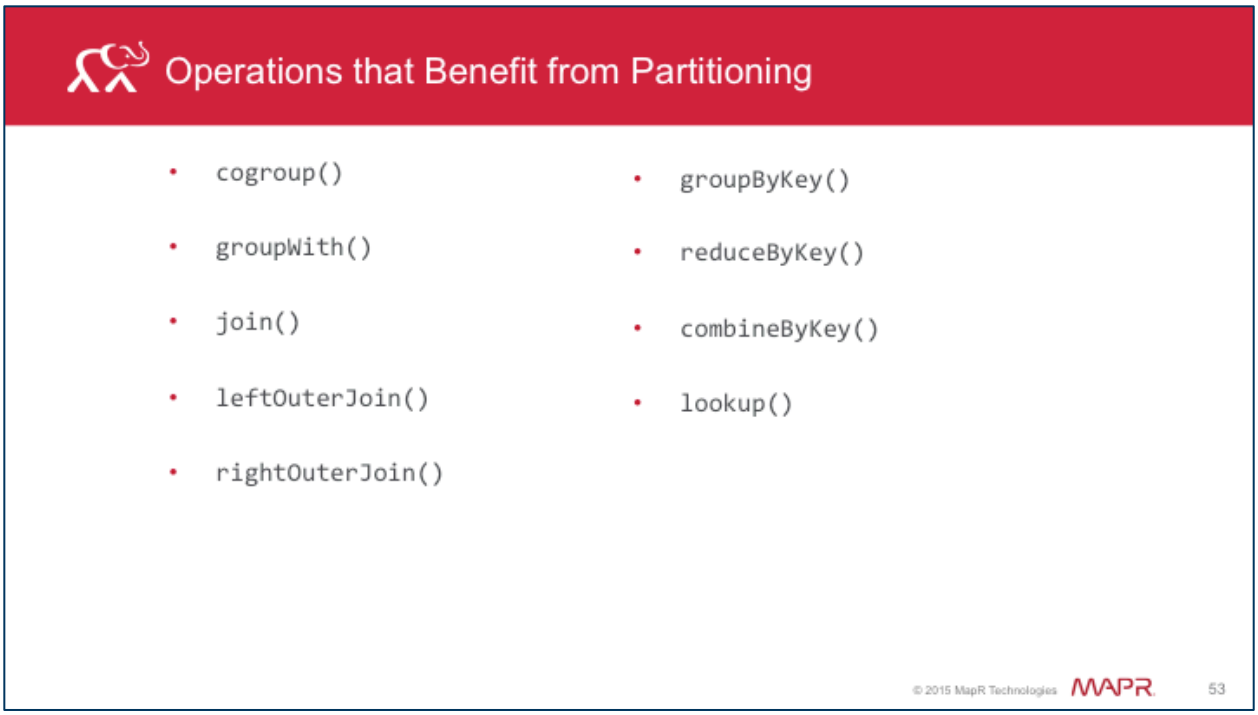
To find out the number of partitions, we use rdd.partitions.size(). Then we apply coalesce specifying the decreased number of partitions, in this case 3.



Notes on Slide 52:

In Scala and Java, you can use the method partitioner() on an RDD to determine how the RDD is partitioned.

Looking at the example used before, when you run the command, partrdd1.partitioner, it returns the type of partitioning – in this case RangePartitioner.

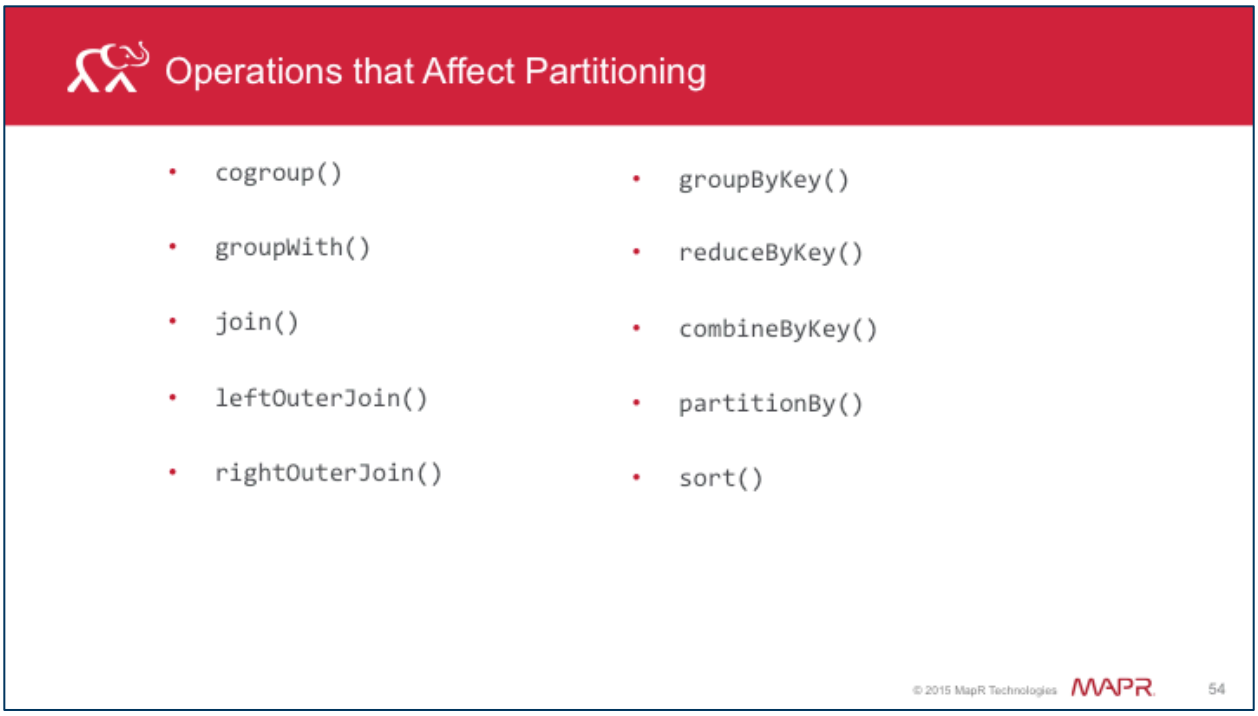


Notes on Slide 53:

Since many operations on pair RDD shuffle data by key across the network, partitioning can benefit many operations. The operations that benefit from partitioning are listed here.

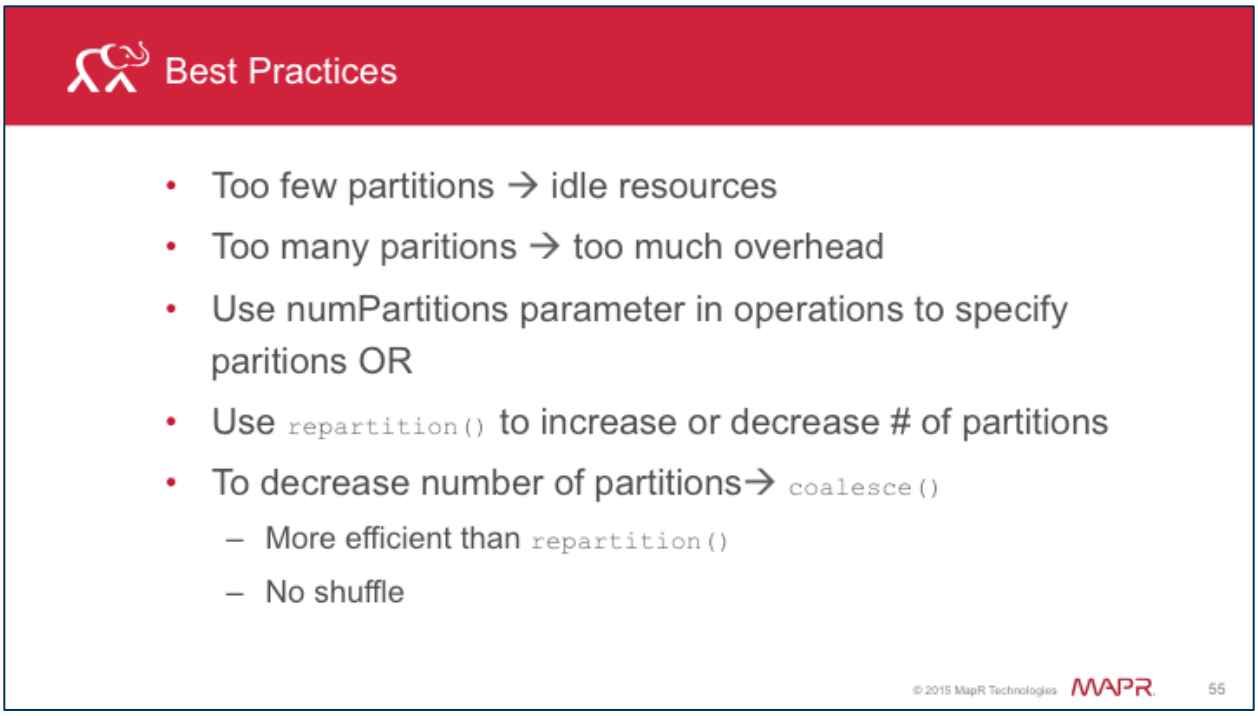
For operations such as reduceByKey, when you pre-partition, the values are computed locally for each key on a single machine. Only the final locally reduced value will be sent from each worker node back to the master.

For operations that act on two pair RDDs, when you pre-partition, at least one of the pair RDD (one with the known partitioner) will not be shuffled. If both the RDDs have the same partitioner, then there is no shuffling across the network.



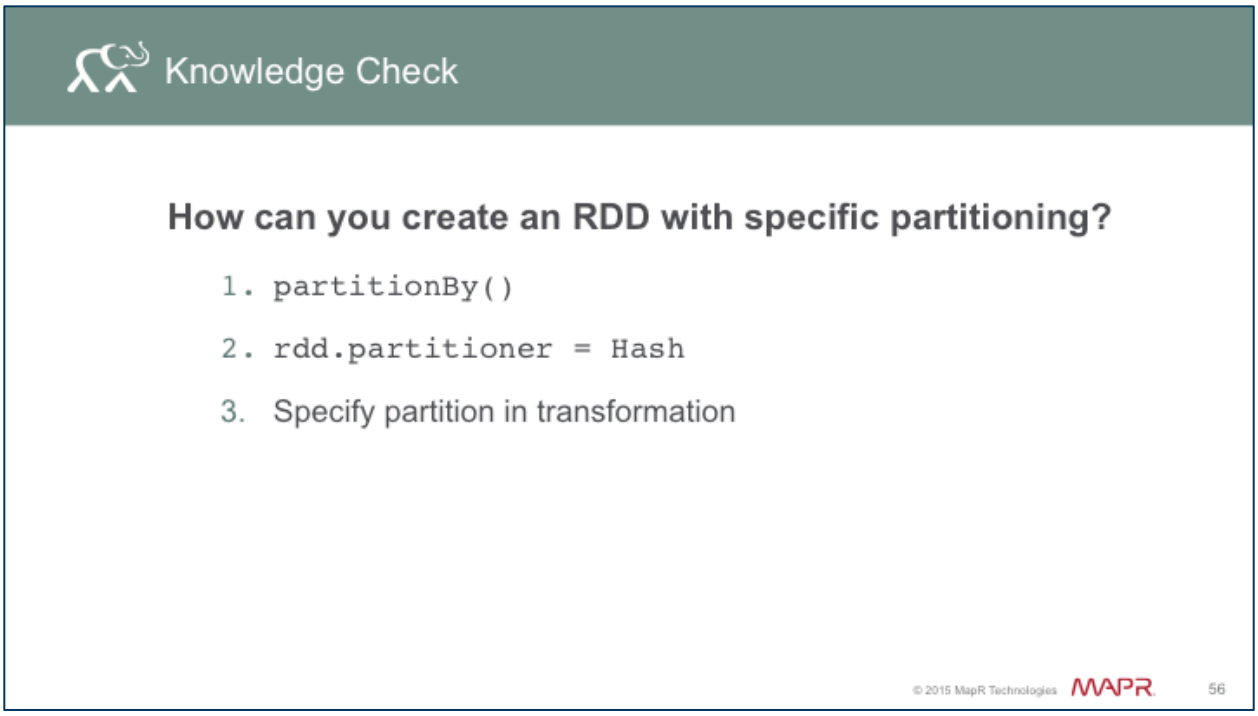
Notes on Slide 54:

These are operations that result in a partitioner being set on the output RDD.
All the other operations will produce a result with no partitioner.
Note that operations such as mapValues(), flatMapValues() and filter() will result in a partitioner being set on the output RDD only if the parent RDD has a partitioner.



Notes on Slide 55:

If you have too few number of partitions, i.e. too little parallelism, Spark will leave resources idle.
If you have too many partitions or too much parallelism, you could affect the performance as the overheads for each partition can add up to provide significant total overhead.
Use repartition() to randomly shuffle existing RDDs to have less or more partitions.
Use coalesce(N) over repartition() to decrease number of partitions as it avoids shuffle.
Note: If N is greater than your current partitions, you need to pass "shuffle=true" to coalesce. If N is less than your current partitions (i.e. you are shrinking partitions) do not set shuffle=true, otherwise it will cause additional unnecessary shuffle overhead.



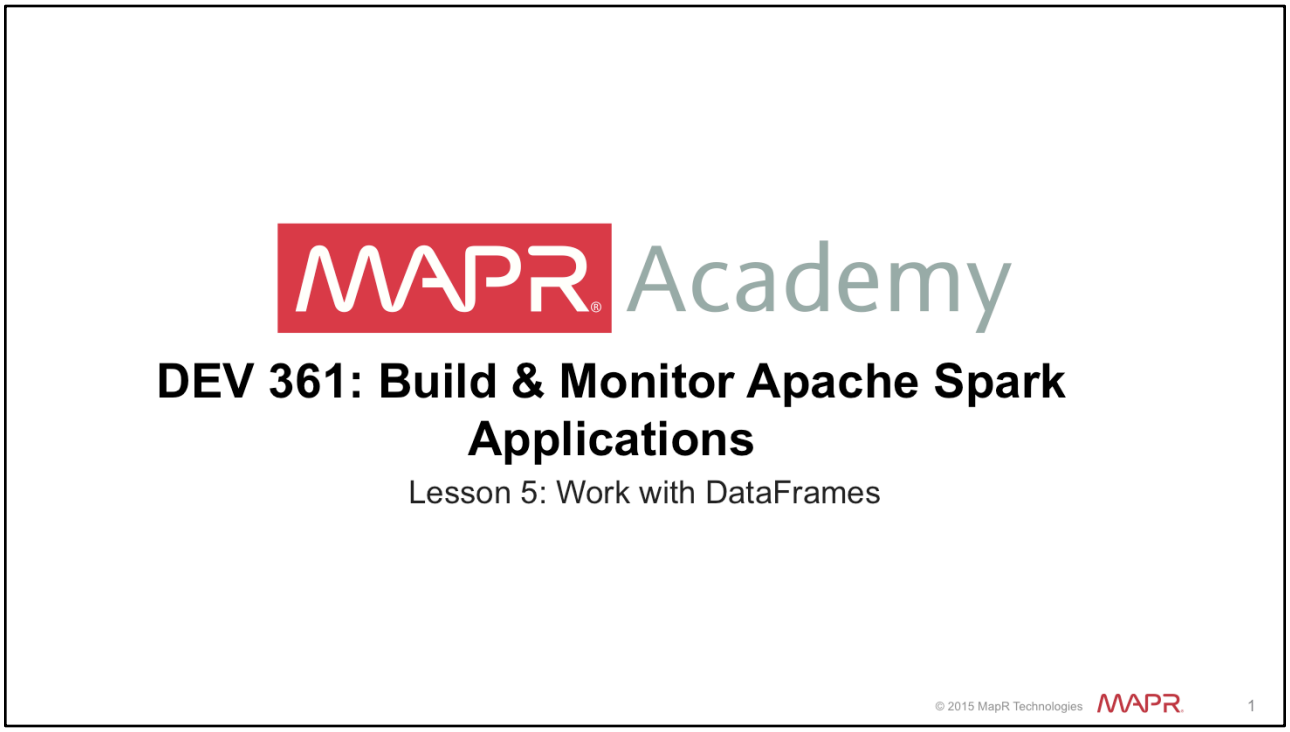
Notes on Slide 56:
Answer: 1, 3

Notes on Slide 57:

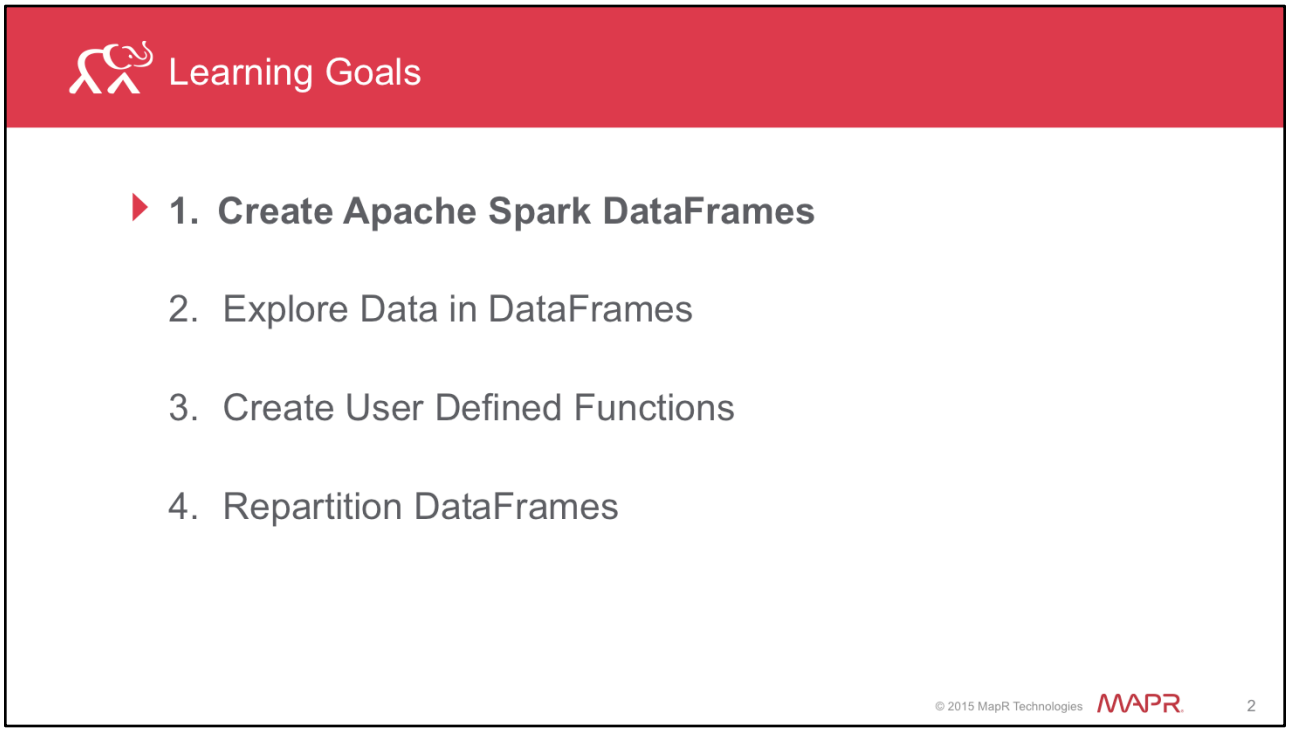
In this activity, you will see how to determine the number of partitions in an RDD, and the type of partitioner.

Notes on Slide 58:

Congratulations! You have completed Lesson 4 of this course.



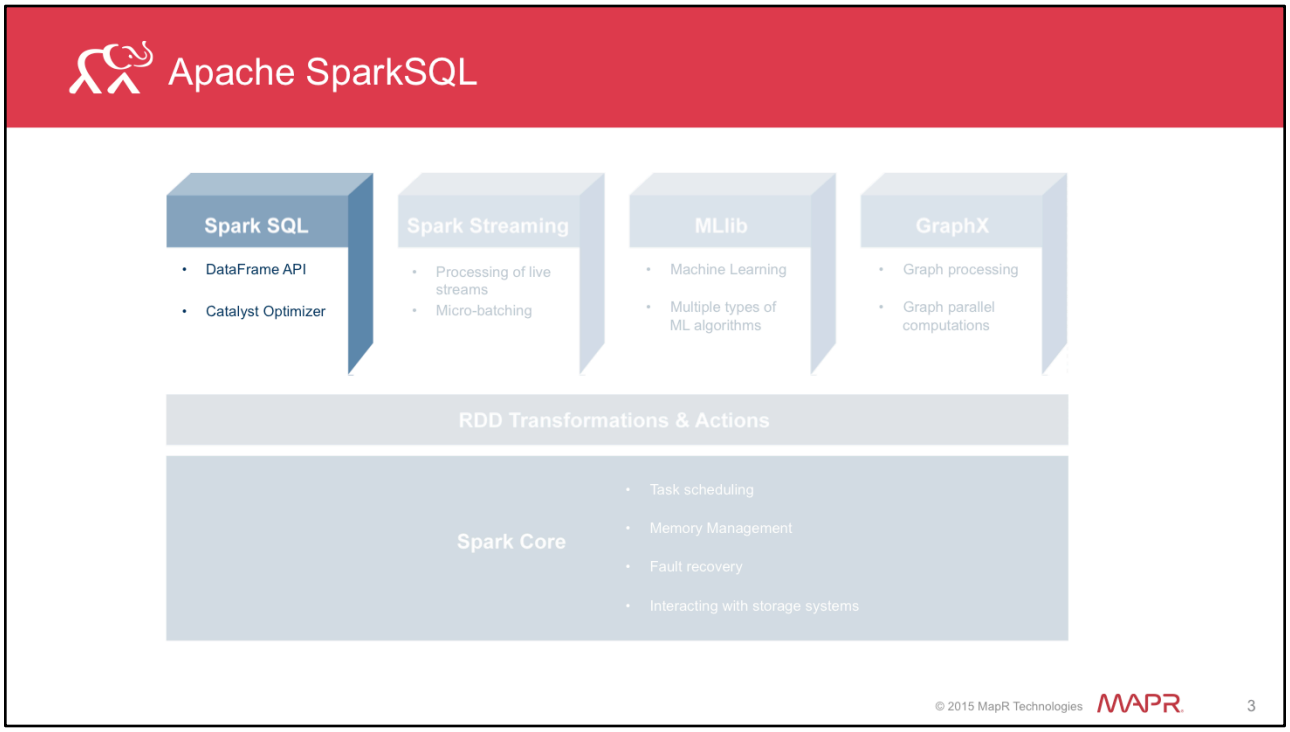
Welcome to Build and Monitor Apache Spark Applications, Lesson 5 - Work with Apache Spark DataFrames.



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

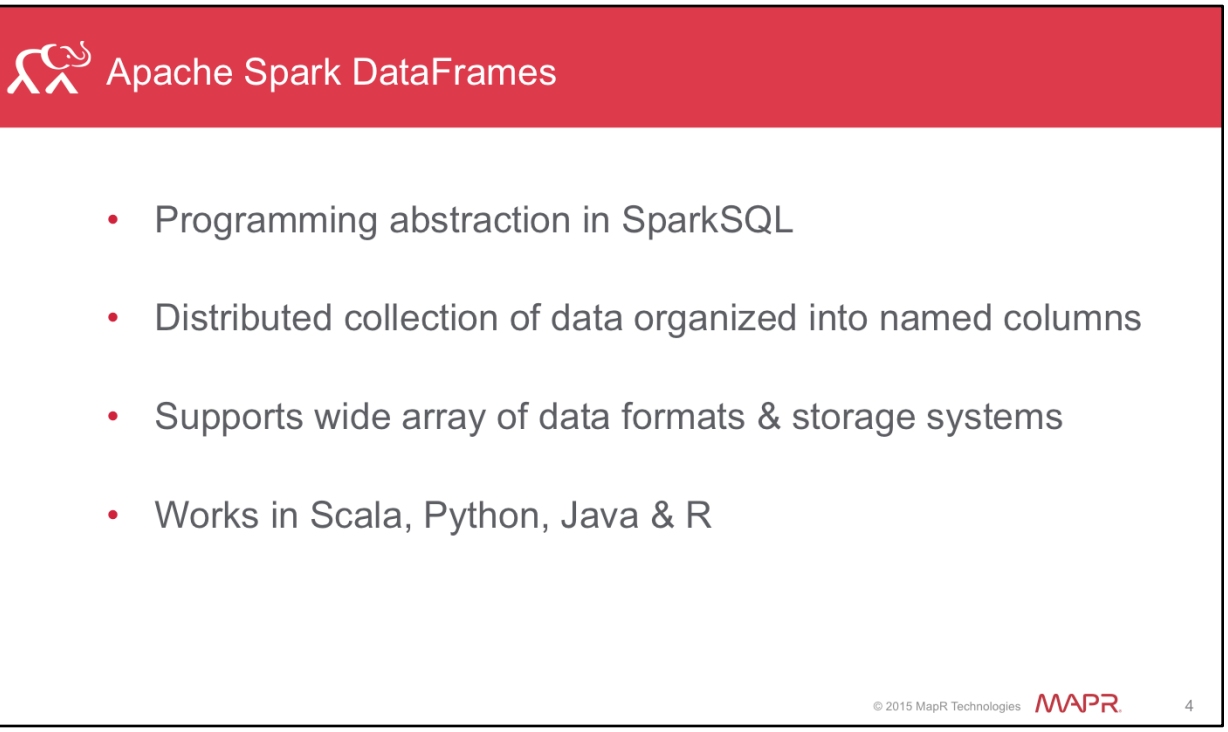
1. Create Apache Spark DataFrames
2. Work with data in DataFrames
3. Create User Defined Functions
4. Repartition DataFrames

In this section, we will create Apache Spark DataFrames.



SparkSQL is a library that runs on top of the Apache Spark core. It provides access to the SQL interface via the Interactive Shell, JDBC/ODBC or through the DataFrame API.

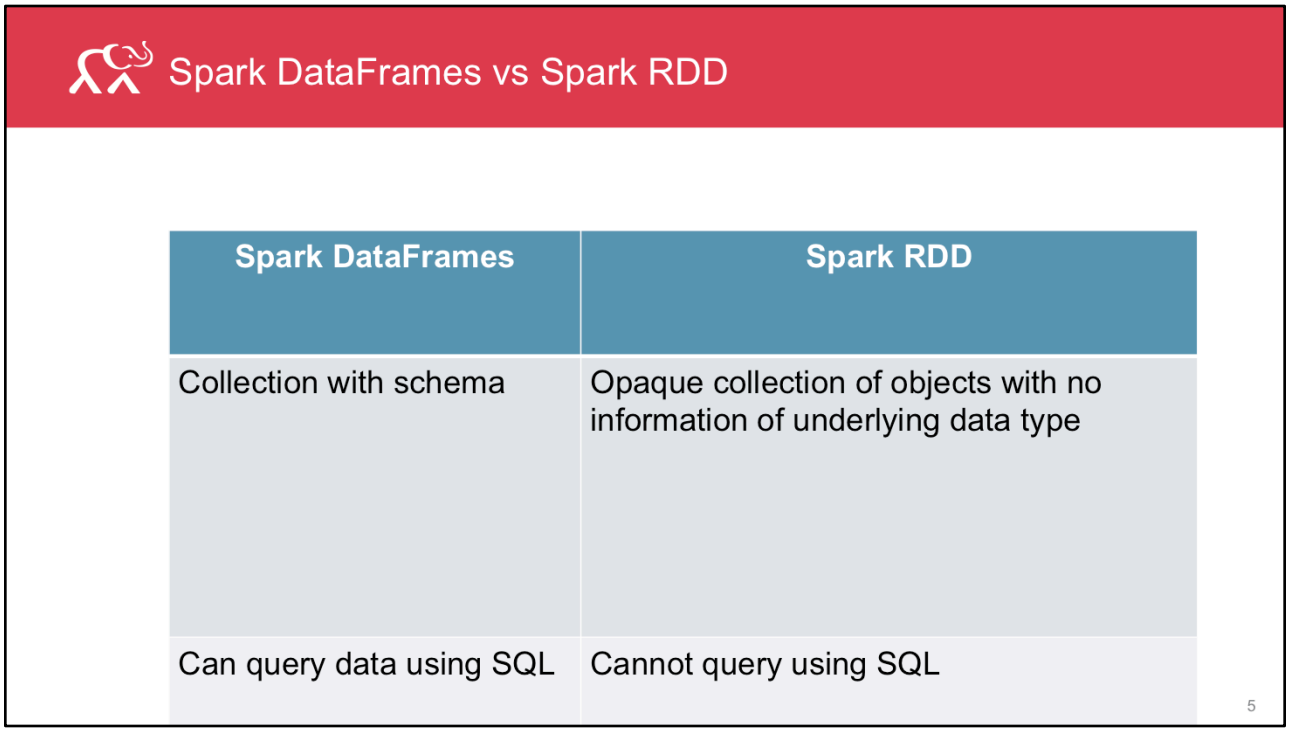
Spark DataFrames use a relational optimizer called the Catalyst optimizer. The SQL queries that are constructed against DataFrames are optimized and executed as sequences of Spark Jobs.



A DataFrame is the programming abstraction in SparkSQL. It is a distributed collection of data organized into named columns. and scales from KBs to PBs.

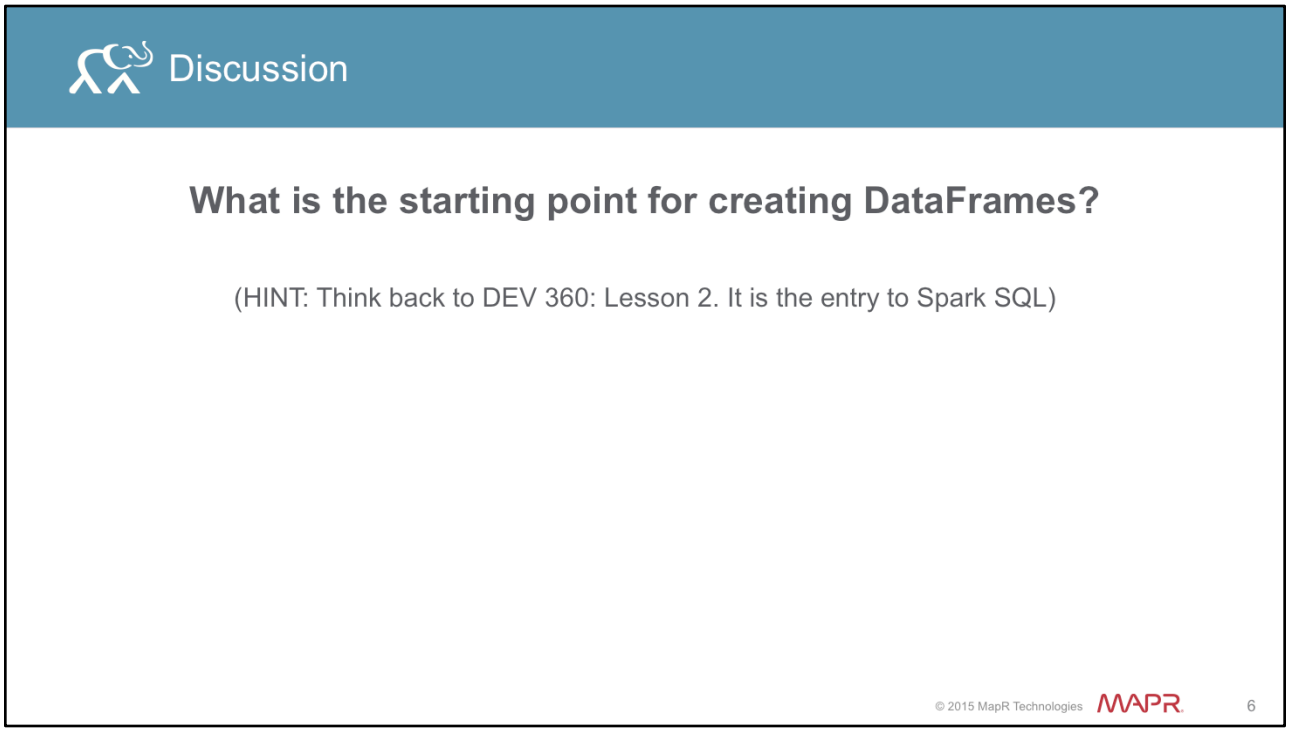
DataFrames supports a wide array of data formats. They can be constructed from structured data files, tables in Hive, external databases or existing RDDs. A DataFrame is equivalent to a Database table but provides a much finer level of optimization.

The DataFrames API is available in Scala, Python, Java and SparkR.



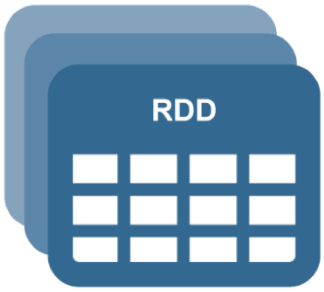
DataFrames are collections of objects that have a schema associated with them. The information about the schema makes it possible to do a lot more optimization. SQL can be used to query the data directly.

Spark RDDs on the other hand are collections of objects with no information about the underlying data format. Data can be queried using RDD transformation and actions but not directly using SQL.



SQLContext – you must create a new SQLContext.

Creating DataFrames



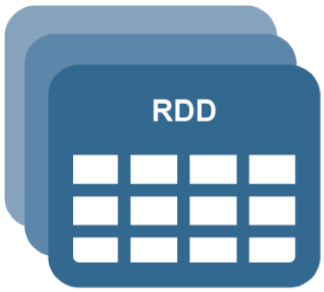
Data Sources

- Parquet
- JSON
- Hive tables
- Relational Databases

There are two ways to create Spark DataFrames. One way is to create it from existing RDDs and the other from data sources.

We will look at creating DataFrames from existing RDD first.

Creating DataFrames from Existing RDD

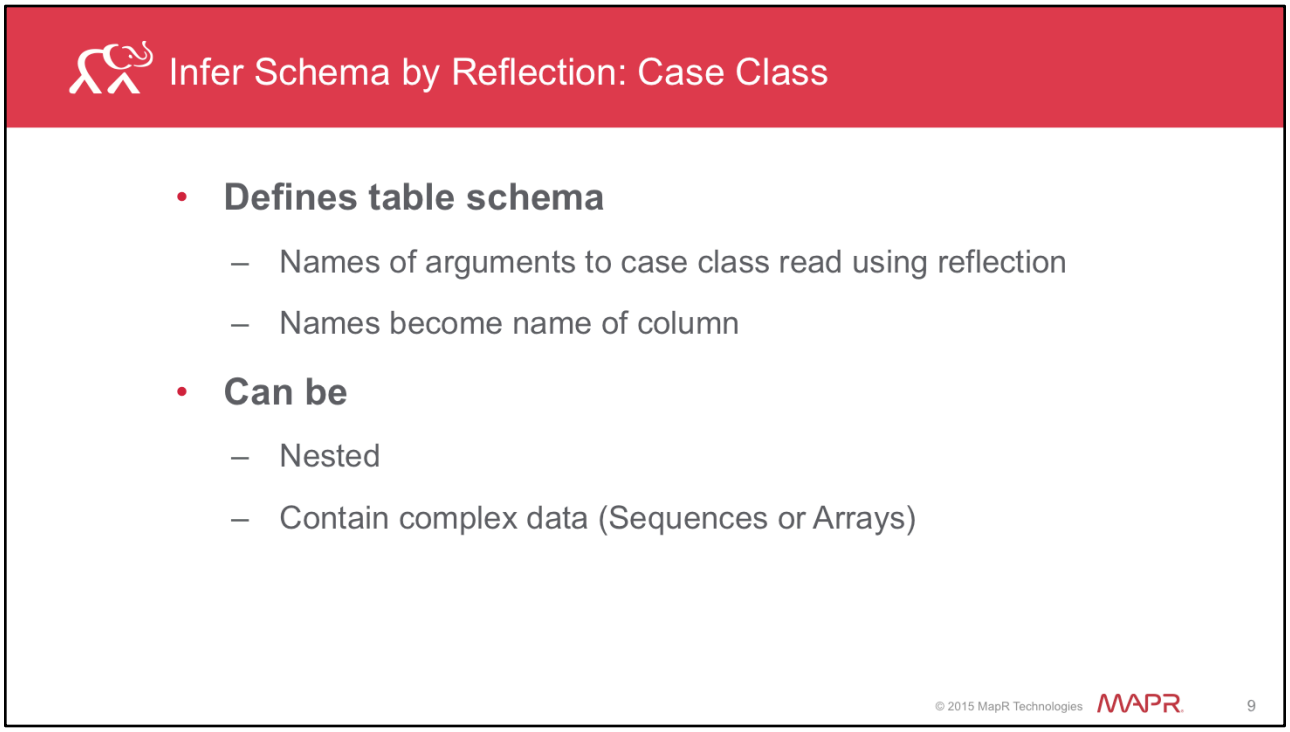


1. **Infer schema by reflection**
   - Convert RDD containing case classes
   - Use when schema is known
2. **Construct schema programmatically**
   - Use to construct DataFrames when columns & their types not known until runtime

You can create DataFrames from existing RDD in two ways. Inferring the schema by reflection is used when the RDD has case classes that define the schema. The DataFrame will infer a fixed schema from the case classes.

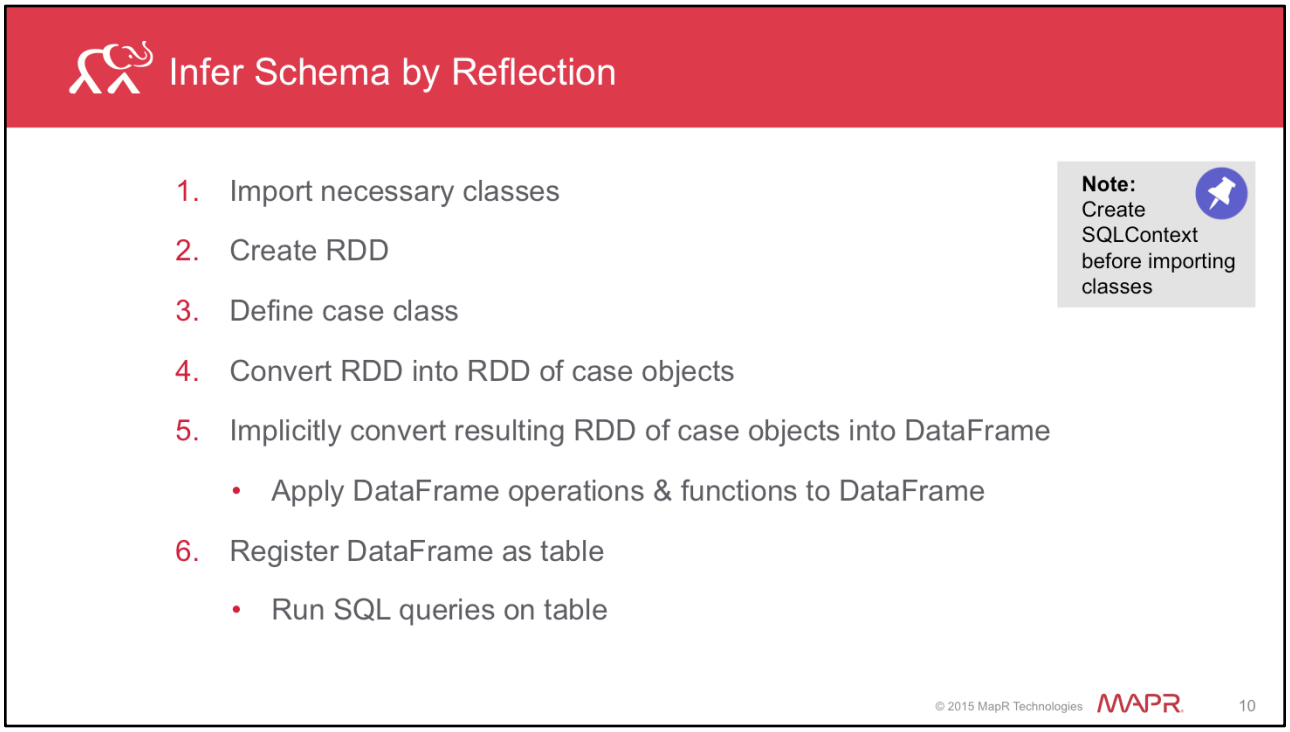
Use the programmatic interface to construct the schema and apply it to the existing RDD at runtime. This method is used when the schema is dynamic based on certain conditions. You also use this method if you have more than 22 fields as the limit to the number of fields in a Case class is 22.

Let us take a look at inferring the schema by reflection next.



The Case class defines the table schema. The names of the arguments passed to the Case class are read using reflection and the names become names of the columns.
Case classes can be nested and can also contain complex data such as sequences or arrays.

NOTE: A Case class in Scala is equivalent to plain old Java Objects (POJOs) or Java beans.



Here are the steps to create a DataFrame from an existing RDD inferring schema by reflection.

1. Import the necessary classes – for example, sqlContext.implicits and all subclasses.

Note that you need to create a SQLContext first which is the starting point for creating the DataFrame.

2. Create the RDD.
3. Define the case class.
4. Convert the RDD into an RDD of case objects using the map transformation to map the case class to every element in the RDD.
5. The resulting RDD is then implicitly converted to DataFrame. We can then apply DataFrame operations and functions to this DataFrame.
6. To run SQL queries on the data in the DataFrame, register it as a table.

Let us take a look at an example next.

DEMO

```
import org.apache.spark.sql._
import sqlContext.implicits._
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val sfpdRDD = sc.textFile("/user/user01/data/
sfpd.csv").map(inc=>inc.split(","))
case class Incidents(incidentnum:String, category:String,
description:String, dayofweek:String, date:String, time:String,
pddistrict:String, resolution:String, address:String, X:Float,
Y:Float, pdid:String)
val sfpdCase = sfpdRDD.map(inc=>Incidents(inc(0), inc(1), inc(2),
inc(3), inc(4),inc(5), inc(6), inc(7), inc(8), inc(9).toFloat,
inc(10).toFloat, inc(11)))
val sfpdDF = sfpdCase.toDF()
sfpdDF.registerTempTable("sfpd")
```