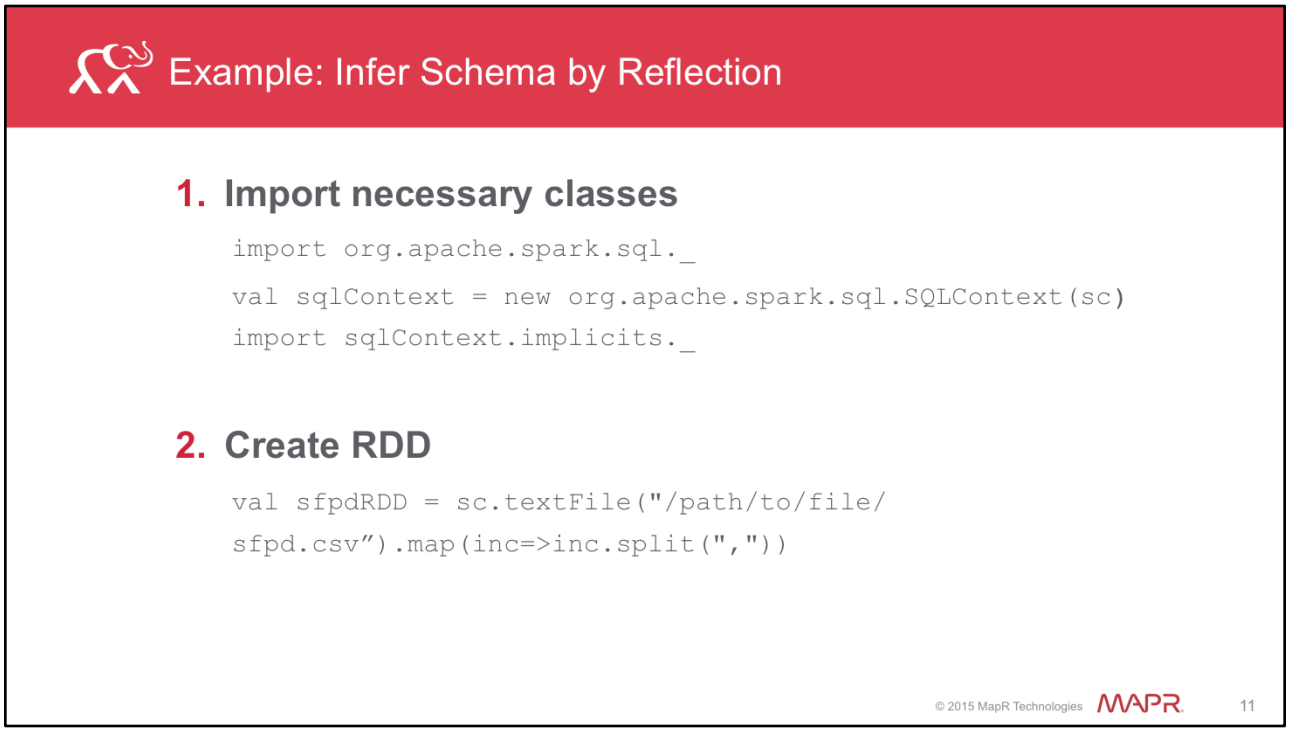
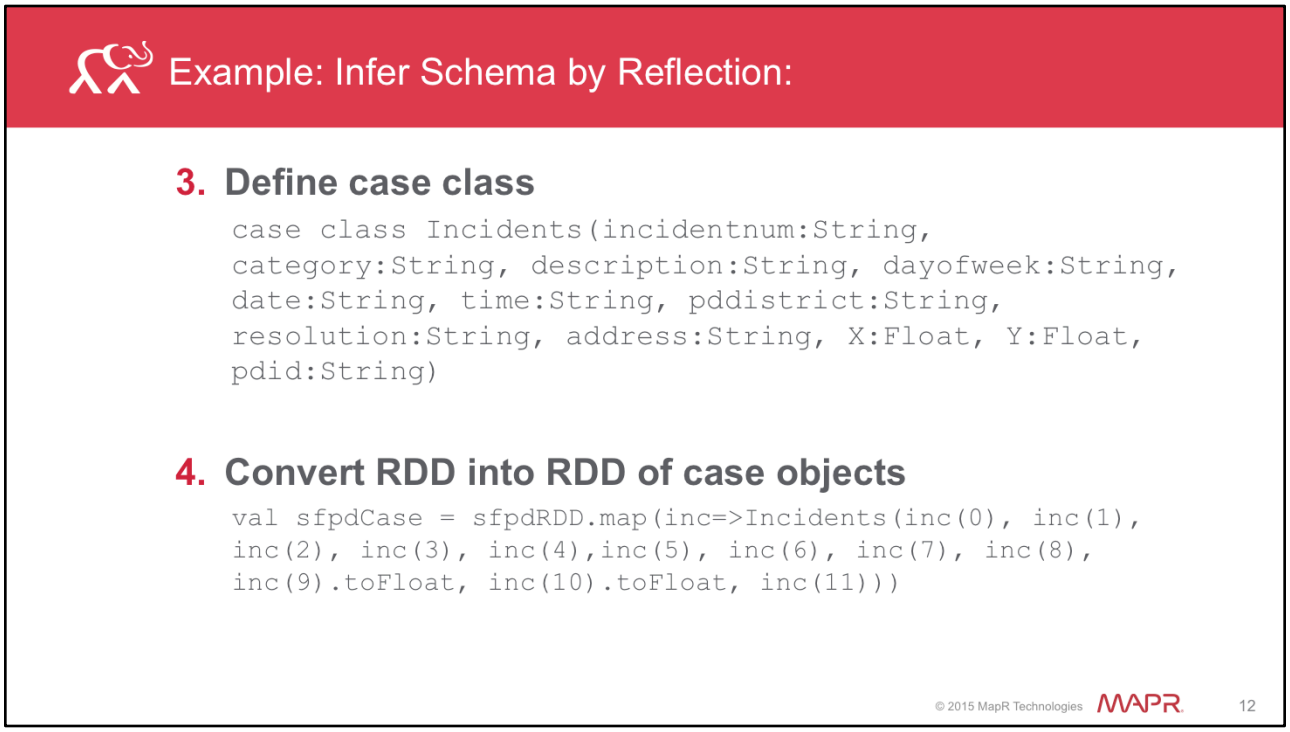


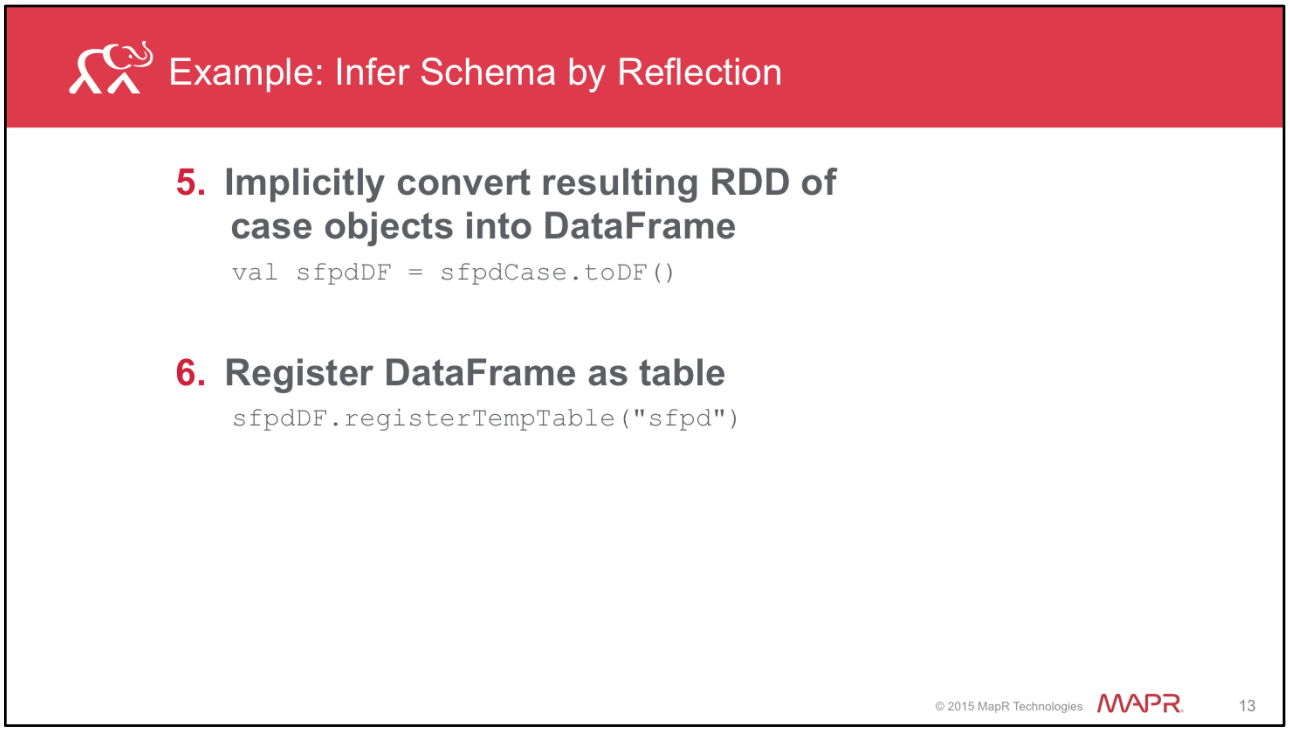
1. First, Import the necessary classes – for example, sqlContext.implicits and all subclasses.

2. The data is in a csv file. Create the base RDD by importing the csv file and splitting on the delimiter ",". We are using the SFPD data in this example.



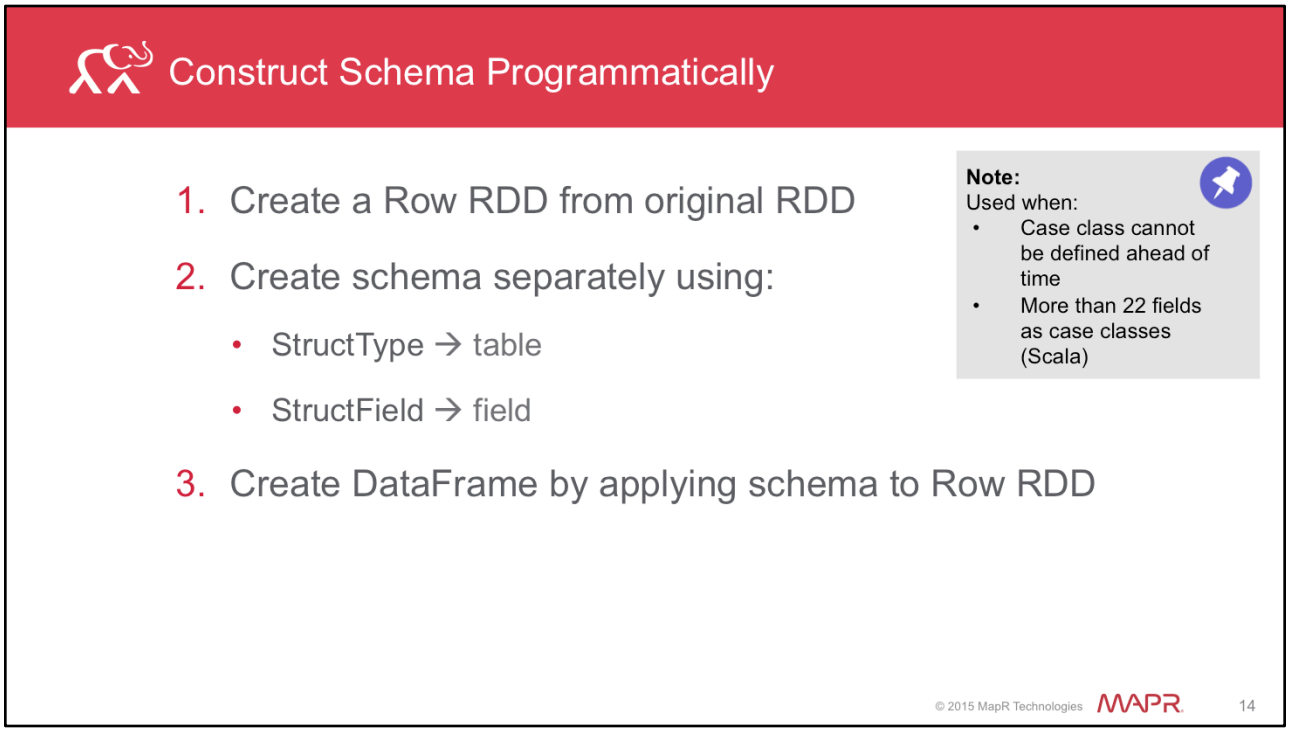
3. Define the case class.

4. Convert the base RDD into an RDD on case objects – sfpdCase. We apply the map transformation to each element of the RDD mapping the case class to every element in the RDD.



5. We then implicitly convert sfpdCase into a DAtaFframe using the toDF() method.. We can apply DataFrame operations to sfpdDF now.

6. Finally we register the DataFrame as a table so we can query it using SQL. We can now query the table – sfpd – using SQL.

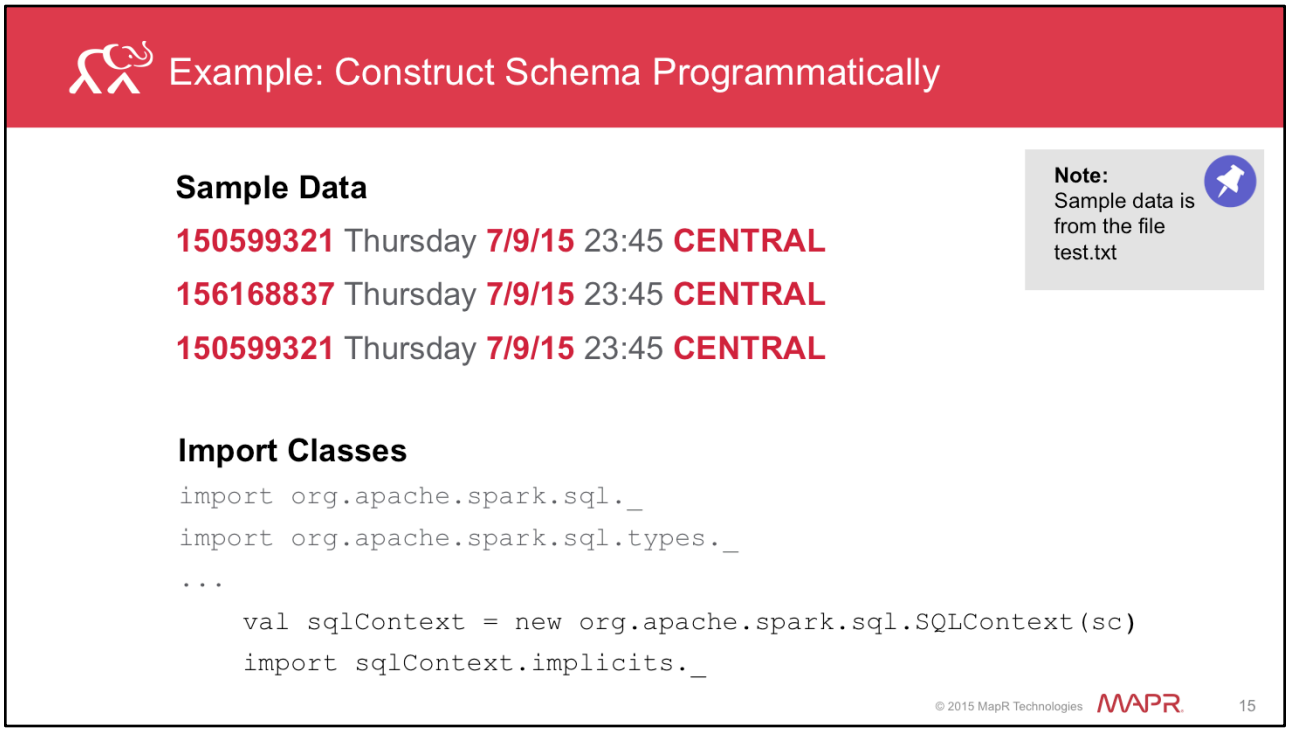


We now take a look at the other method to create DataFrames from existing RDDs. When case classes cannot be defined ahead of time. For example, we want parts of a string to represent different fields, or we want to parse a text dataset based on the user. In this case a DataFrame can be created programmatically with three steps.

1. Create an RDD of Rows from the original RDD;
2. Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
3. Apply the schema to the RDD of Rows via createDataFrame method provided by SQLContext.

Another reason for using this method is when there are more than 22 fields as there is a limit of 22 fields in a case class in Scala.

The schema for the SFPD data is known and it has less than 22 fields. We will take a look at another simple example to demonstrate how to programmatically construct the schema.



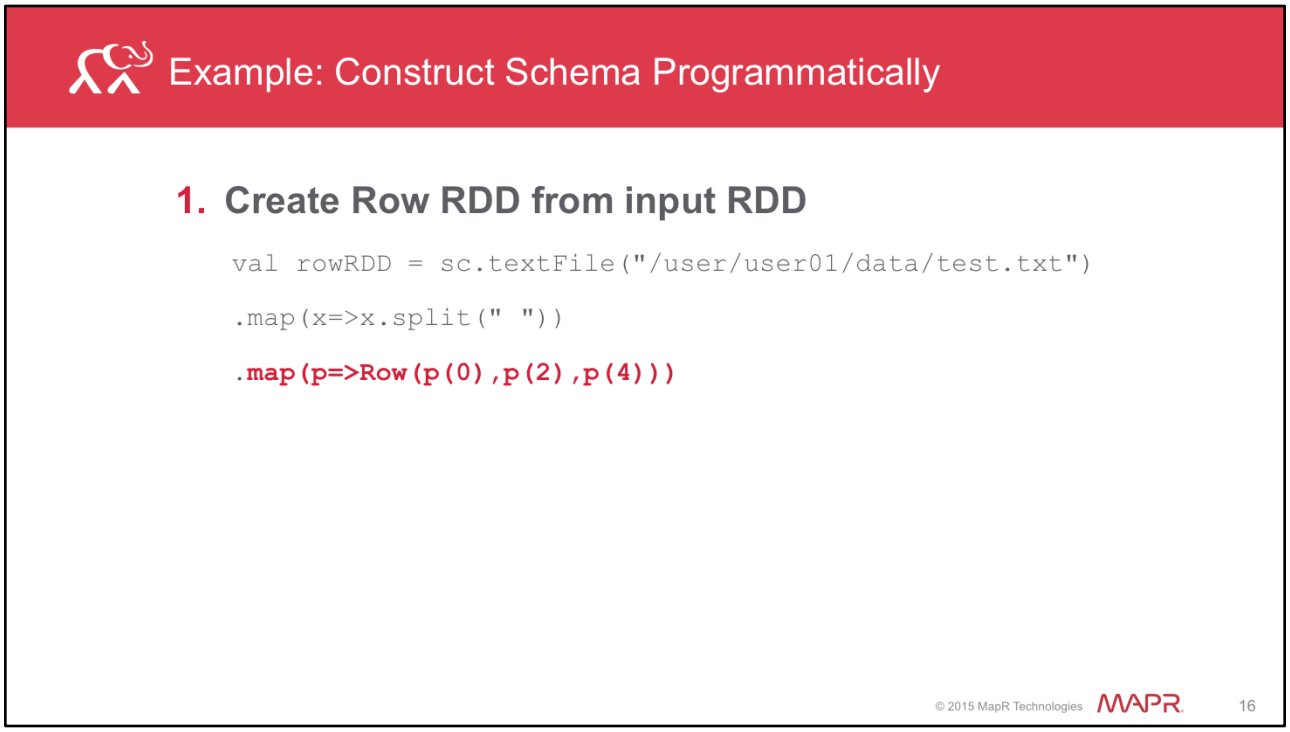
Here is sample data that we will use to create a DataFrame. We have a group of users that is only interested in a DataFrame that has data from the first, third and last "columns" – incident number, date of incident and the district.

First we need to import the necessary classes.

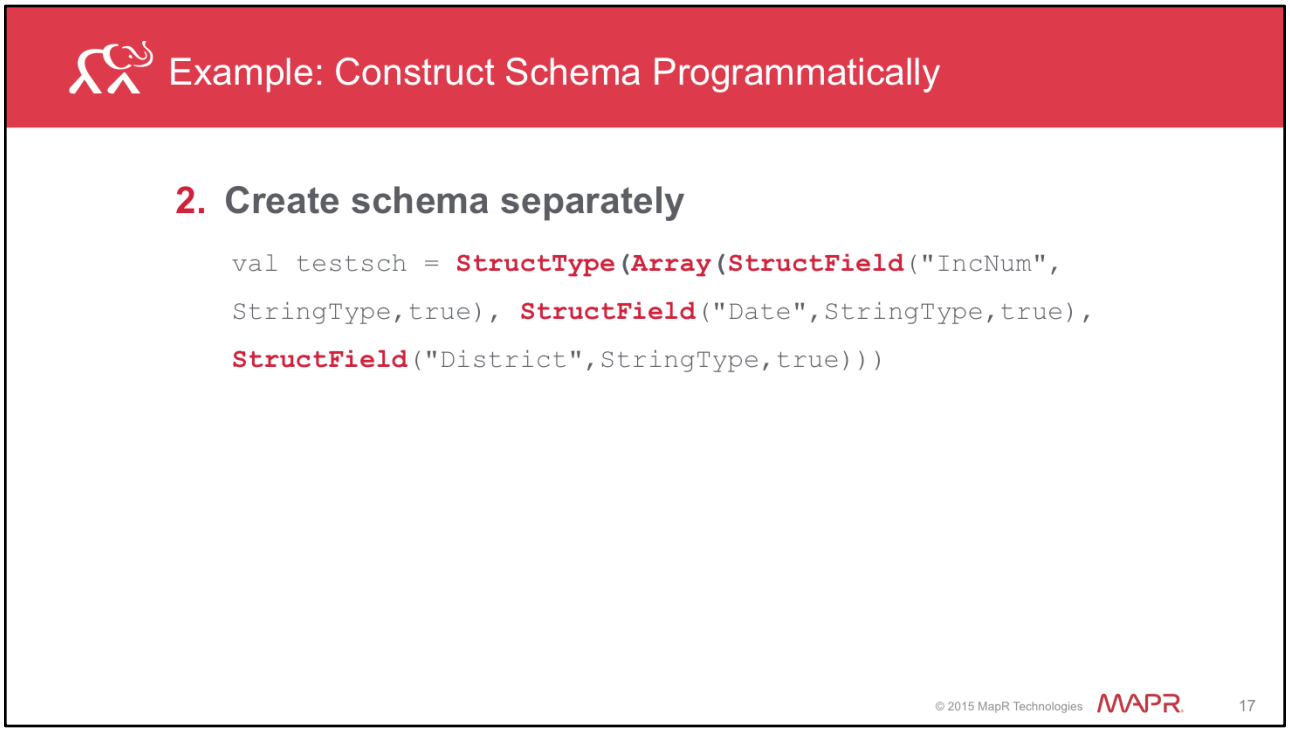
If you want to demo this, this sample data is in a file test.txt.
DEMO

```
import sqlContext.implicits._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
val rowRDD = sc.textFile("/user/user01/data/
test.txt".map(x=>x.split(" ")).map(p=>Row(p(0),p(2),p(4)))
val testsch = StructType(Array(StructField("IncNum",
StringType,true), StructField("Date",StringType,true),
StructField("District",StringType,true)))
val testDF = sqlContext.createDataFrame(rowRDD,schema)
testDF.registerTempTable("test")

val incs = sql("SELECT * FROM test")
```



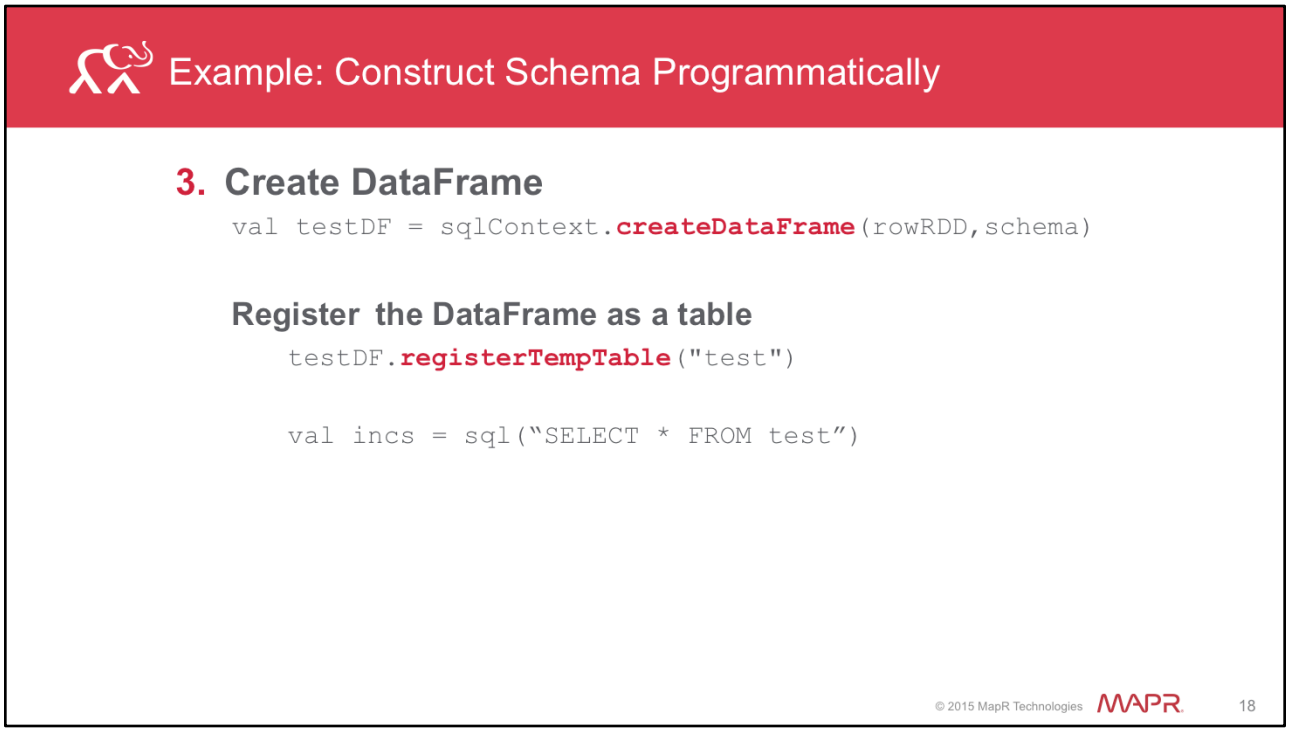
In this step, we load the data into an RDD, apply map to split on space and then convert that RDD into a Row RDD with the last map transformation.



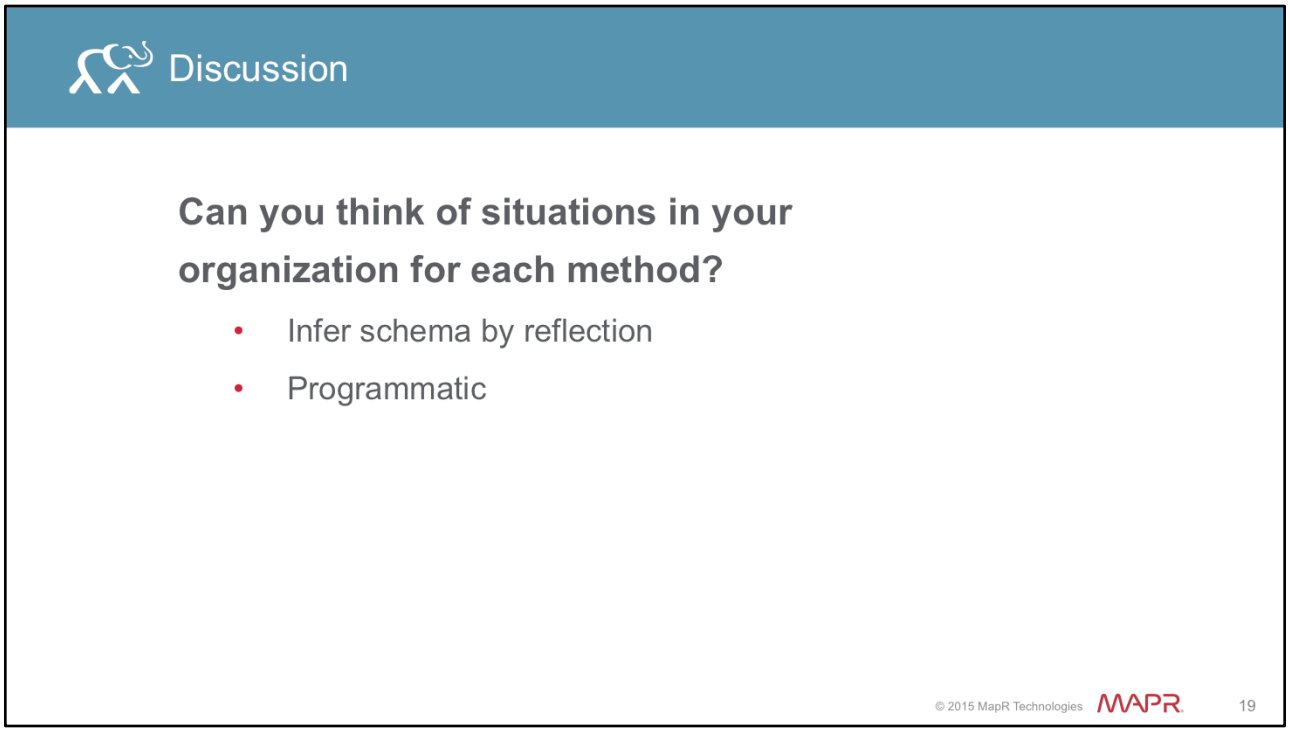
The StructType object defines the schema. It takes an array of StructField objects.

- StructType takes the arguments: (fields, Array[StructField])
- StructField takes the following arguments: (name, dataType , nullable – Boolean, metadata).

In this example, we are building a schema called testsch. We are defining fields IncNum, Date and District. Each field here is a String (StringType) and can be null (nullable = true).

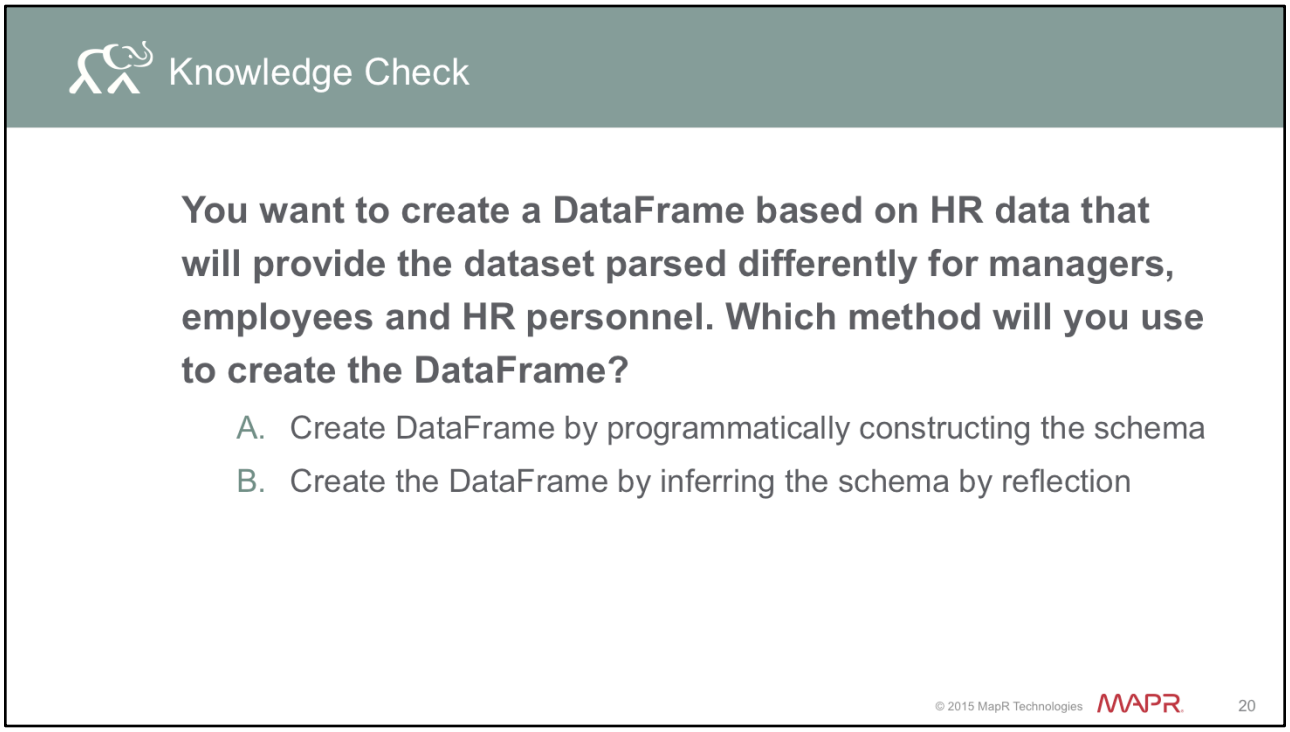


Create the DataFrame from the Row RDD by applying the schema to it.
Once the DataFrame is created, it can be registered as a table and the table can be queried using SQL as shown here.

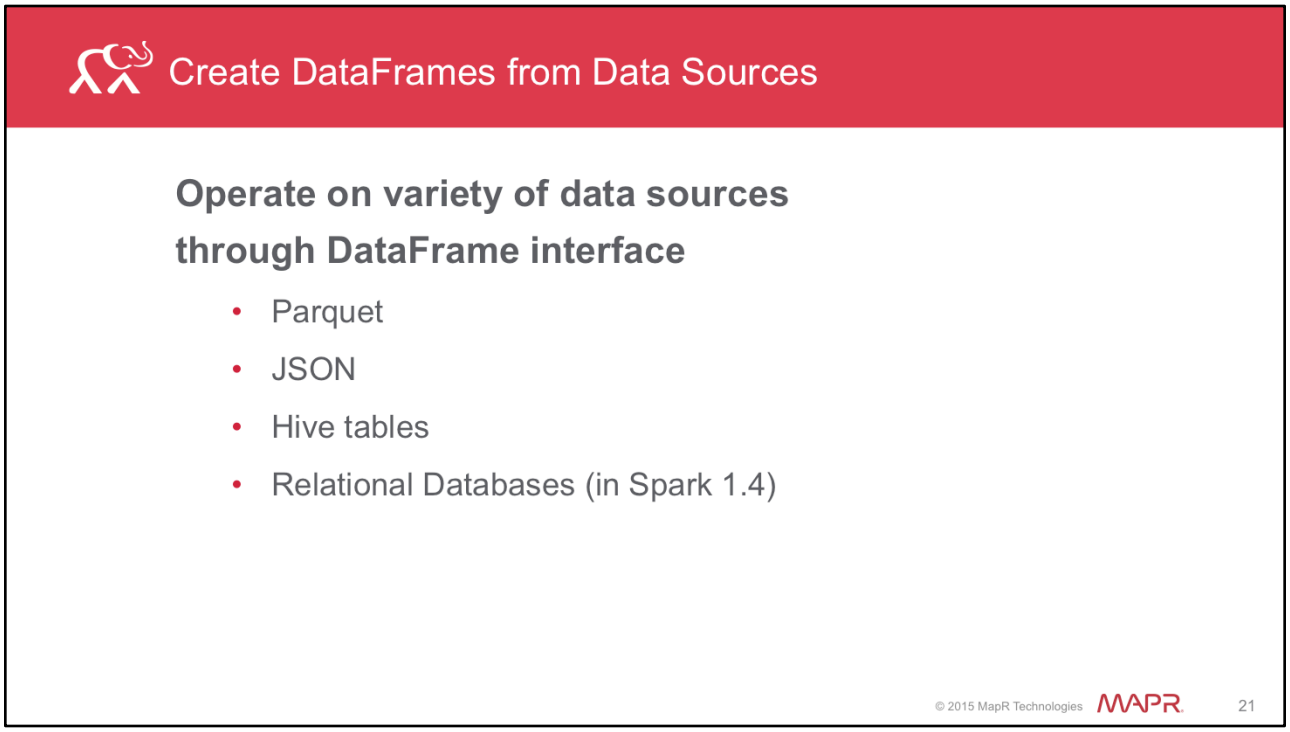


Another way to think about this:
Think of situations where you have data whose schema is known and all your users are going to see the same fields in the same way?

Are there situations where some of users may require the field to be parsed differently?


Knowledge Check
You want to create a DataFrame based on HR data that will provide the dataset parsed differently for managers, employees and HR personnel. Which method will you use to create the DataFrame?
A. Create DataFrame by programmatically constructing the schema
B. Create the DataFrame by inferring the schema by reflection
© 2015 MapR Technologies
20

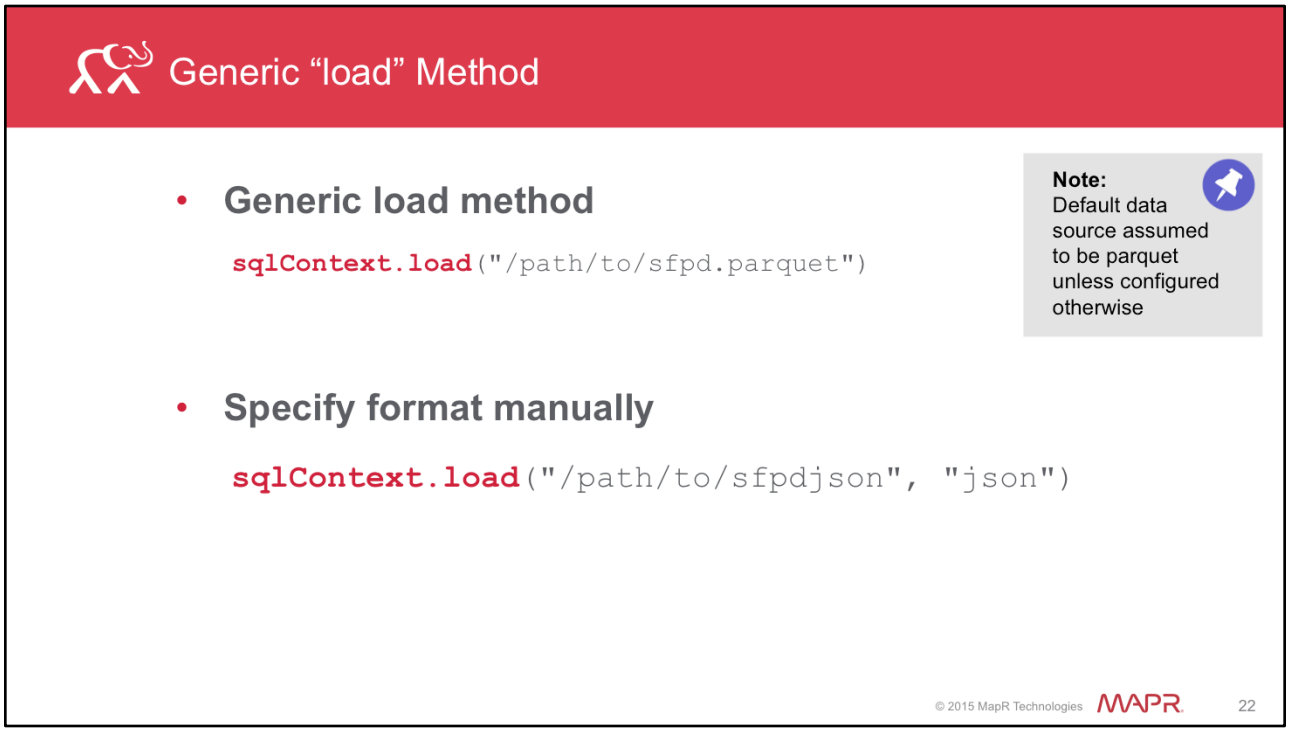
Answer: A



Spark SQL can operate on a variety of data sources through the DataFrame interface.

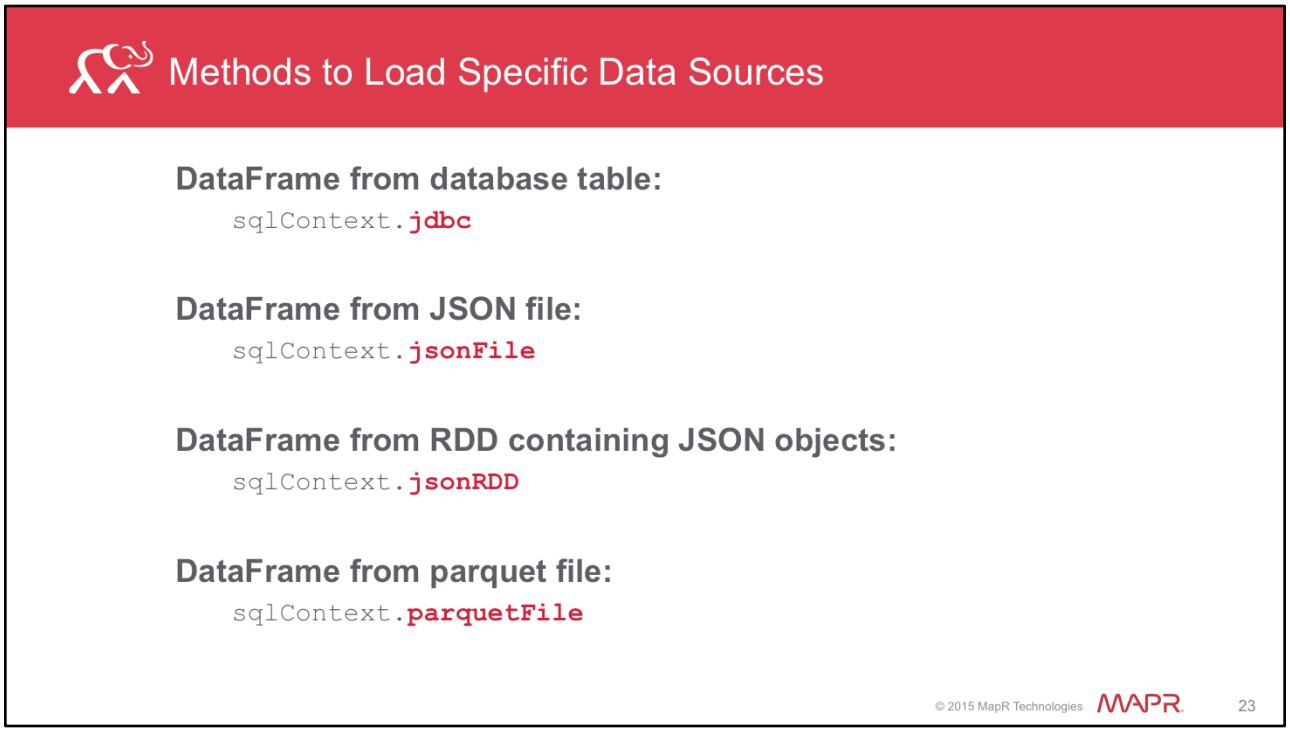
Spark SQL includes a schema inference algorithm for JSON and other semistructured data that enables users to query the data right away.

Spark 1.4 provides support to load data directly from Relational Databases into DataFrames.

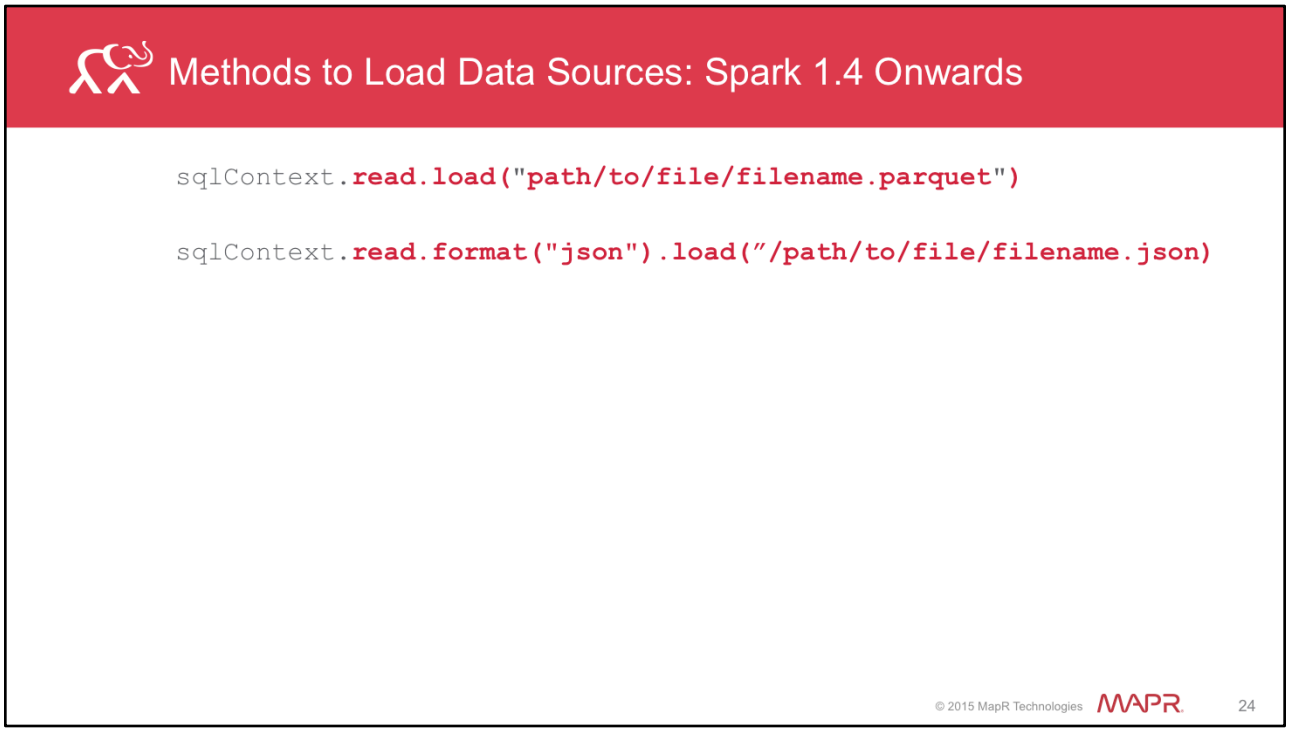


There are generic load and save functions. The default data source - parquet will be used for all operations unless otherwise configured by spark.sql.sources.default.

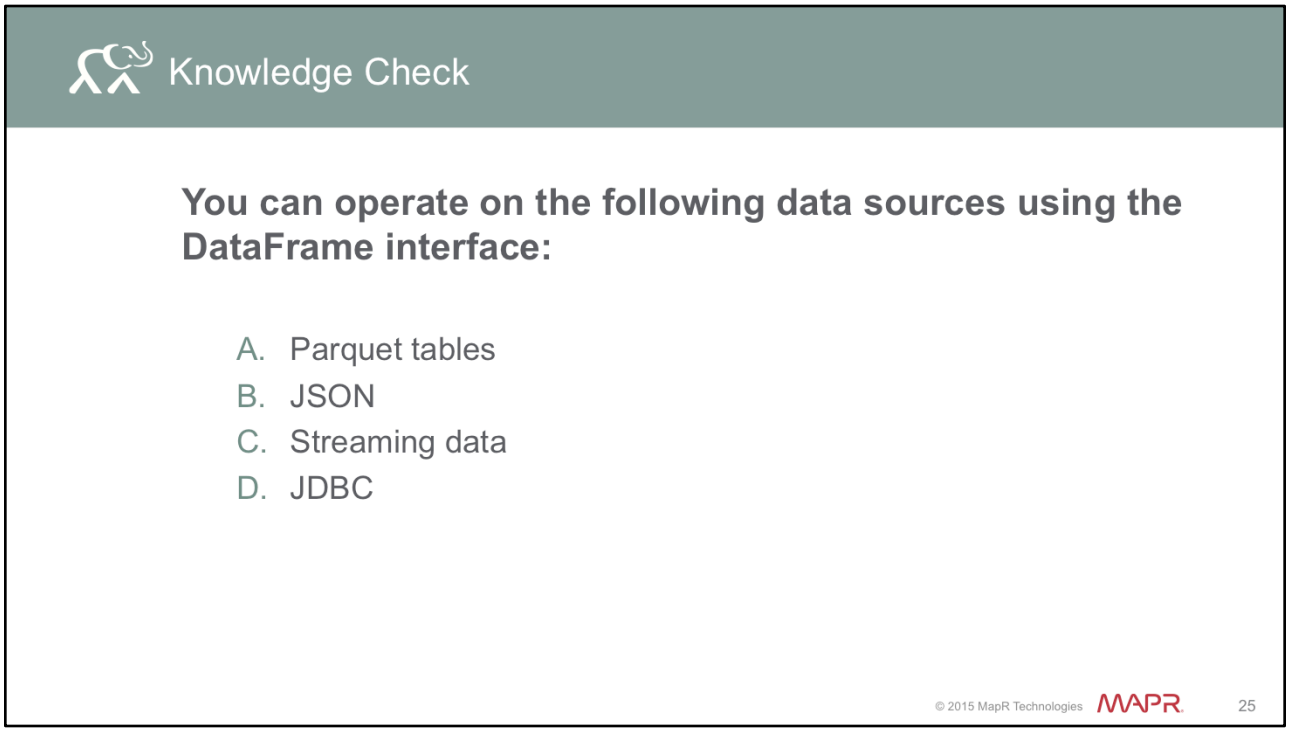
You can also specify options manually as shown here. Data sources are specified by their fully qualified name (eg. org.apache.spark.sql.parquet), for built in resources use the shortened form (parquet, json, jdbc)



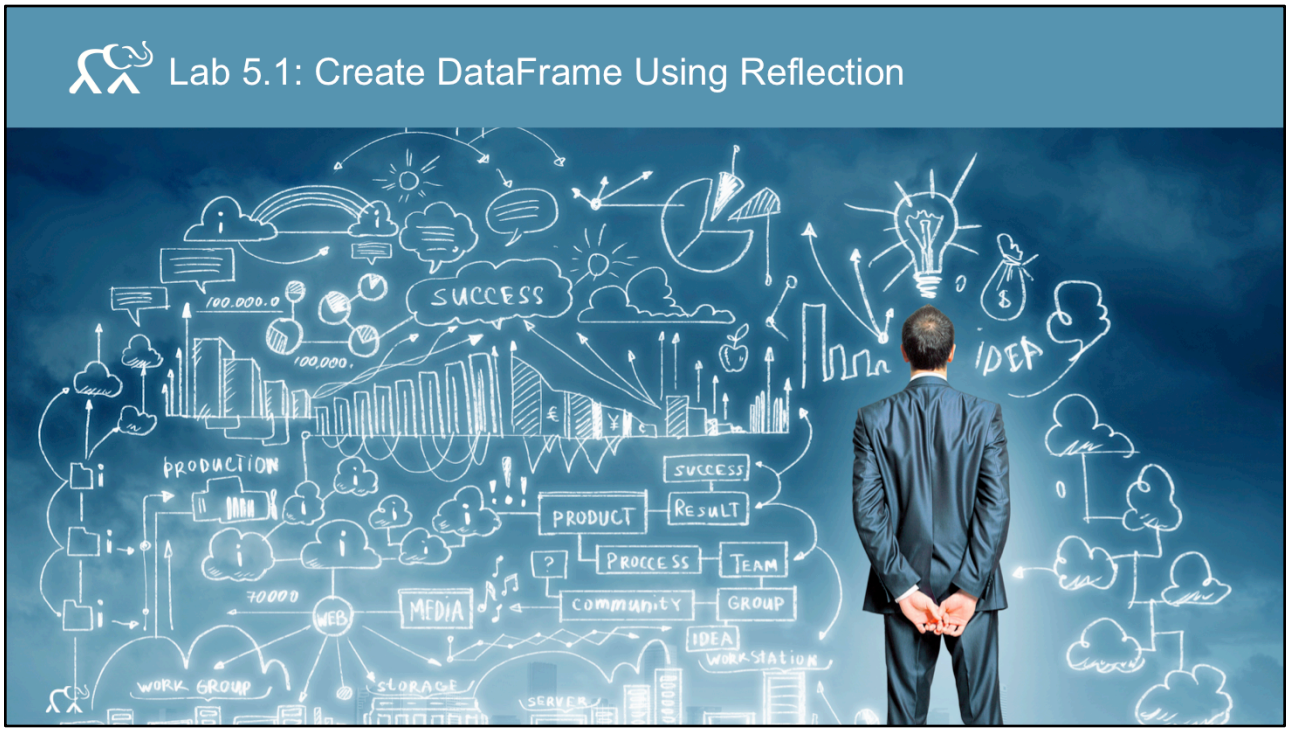
There are methods available to load specific data sources into DataFrames. These methods have been deprecated in Spark 1.4. Refer to the Apache Spark documentation.



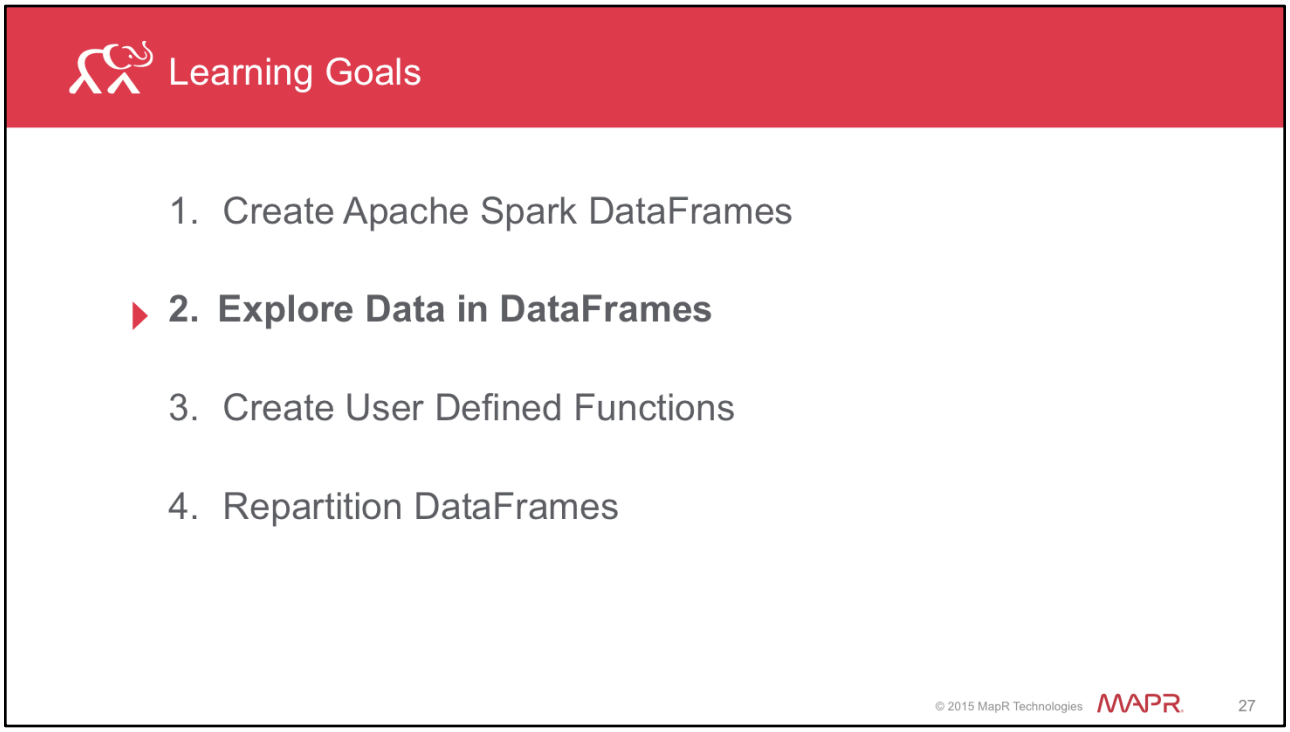
This is the method to load from data sources to DataFrames in Spark 1.4 onwards.


Knowledge Check
You can operate on the following data sources using the DataFrame interface:
A. Parquet tables
B. JSON
C. Streaming data
D. JDBC
© 2015 MapR Technologies
25

Answer: A,B,D

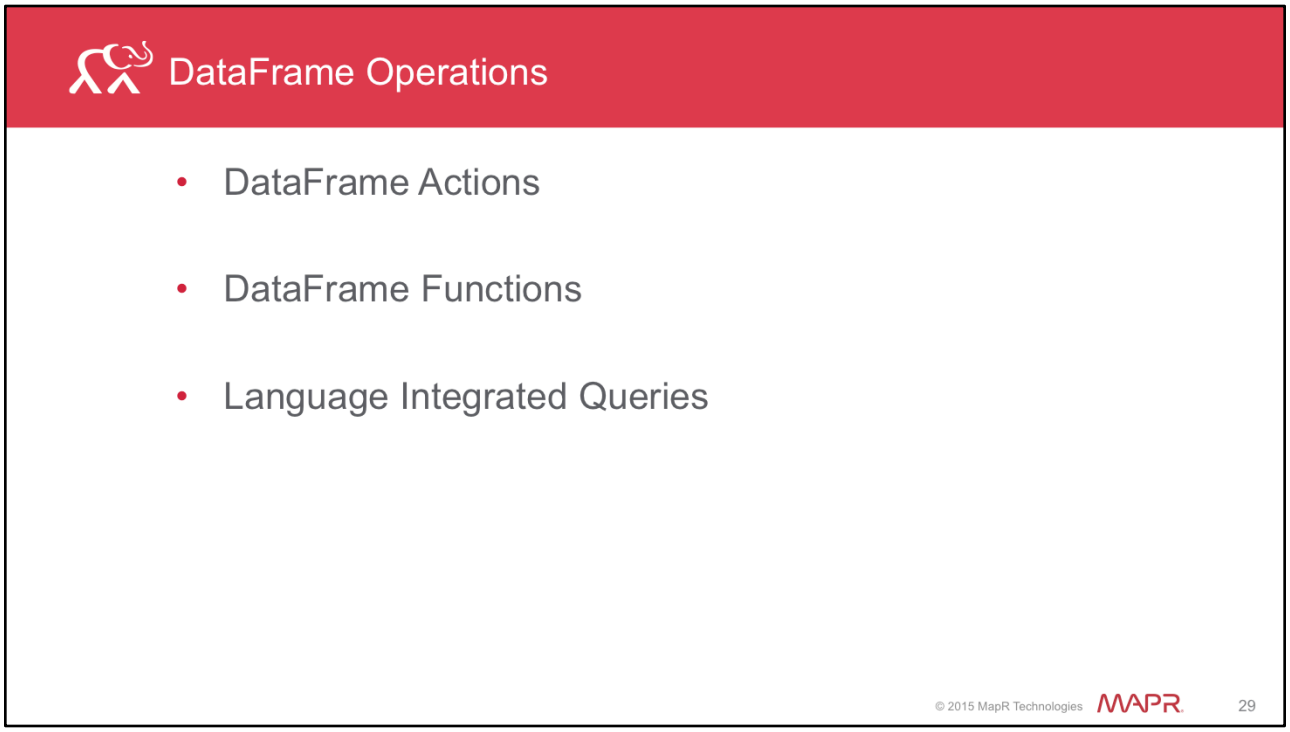


In this lab, you will create the DataFrame based on the SFPD dataset using reflection.

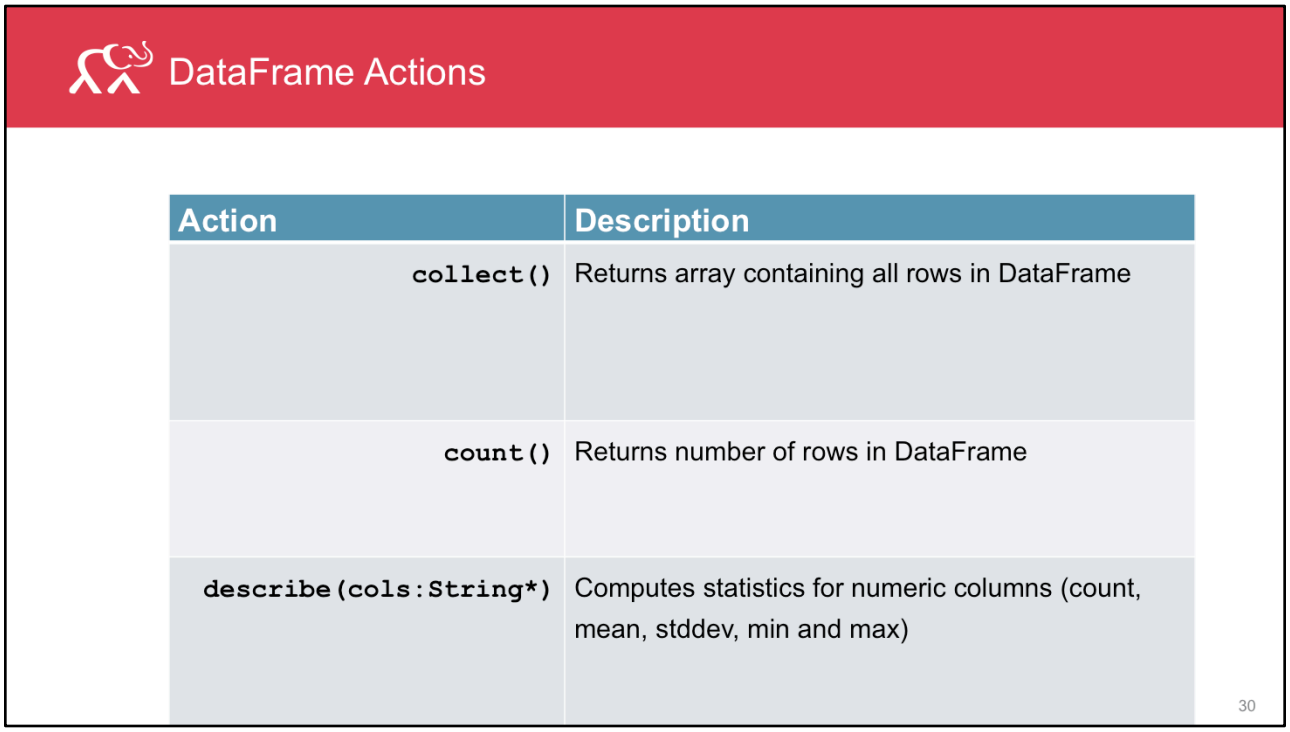


In this section we will explore data in DataFrames using DataFrame operations and SQL.

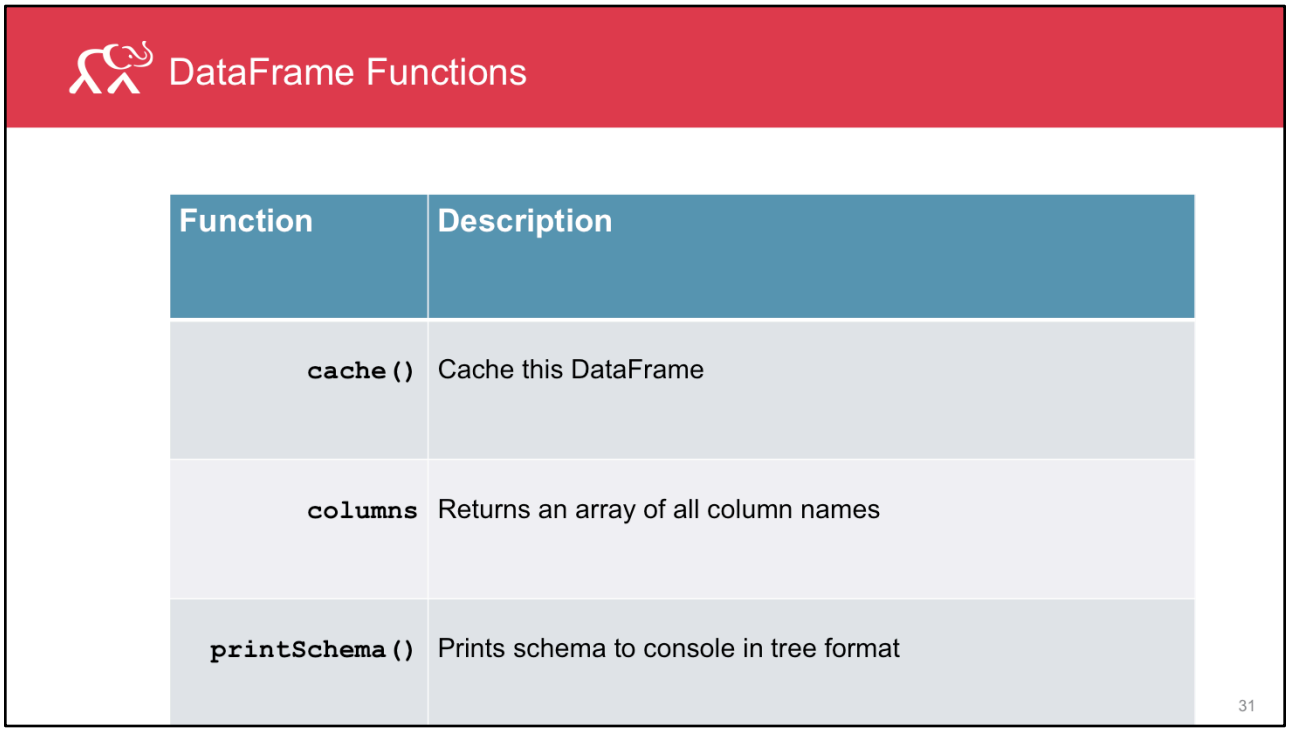
Here are some questions we can ask of the SFPD data. Next we will discuss how to apply operations to the DataFrame to answer these questions.



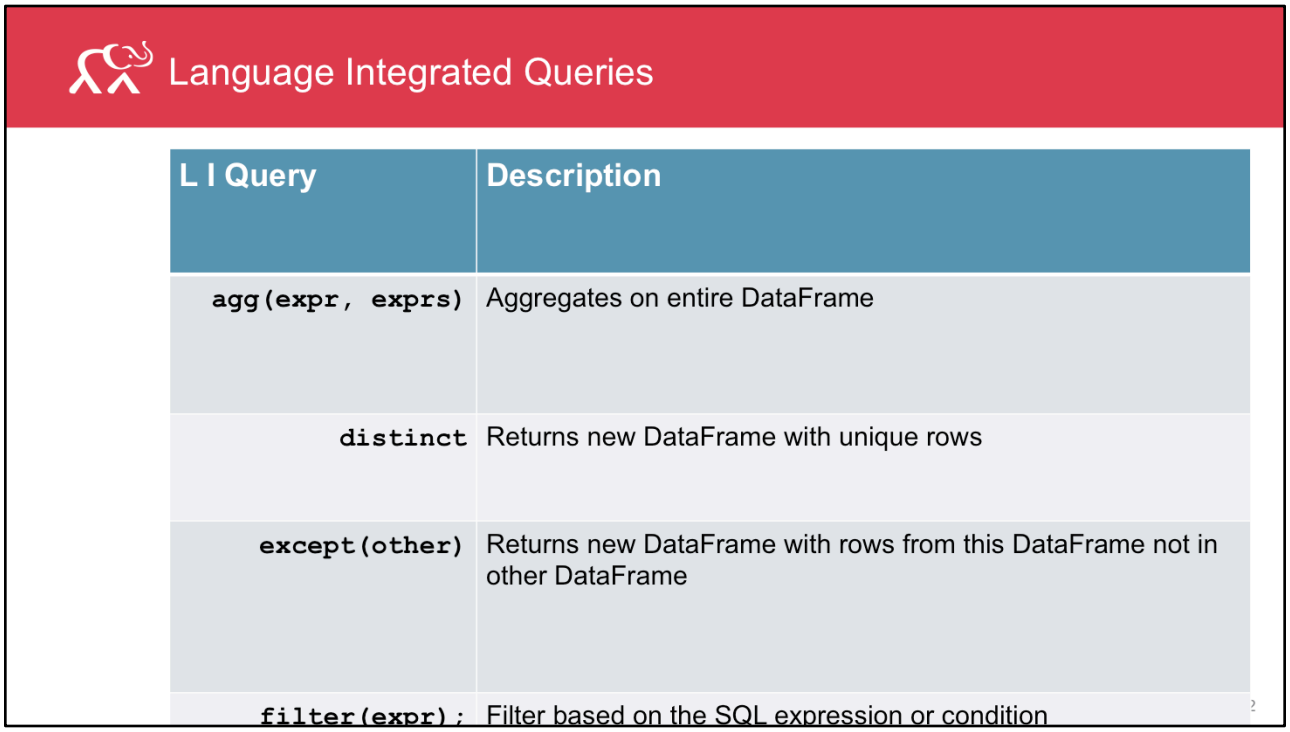
There are different categories of operations that can be performed on DataFrames. In addition to the ones listed here, you can also use some RDD operations on DataFrames. You can also output data from DataFrames to tables and files.



This table lists DataFrame actions.



This table lists DataFrame functions.



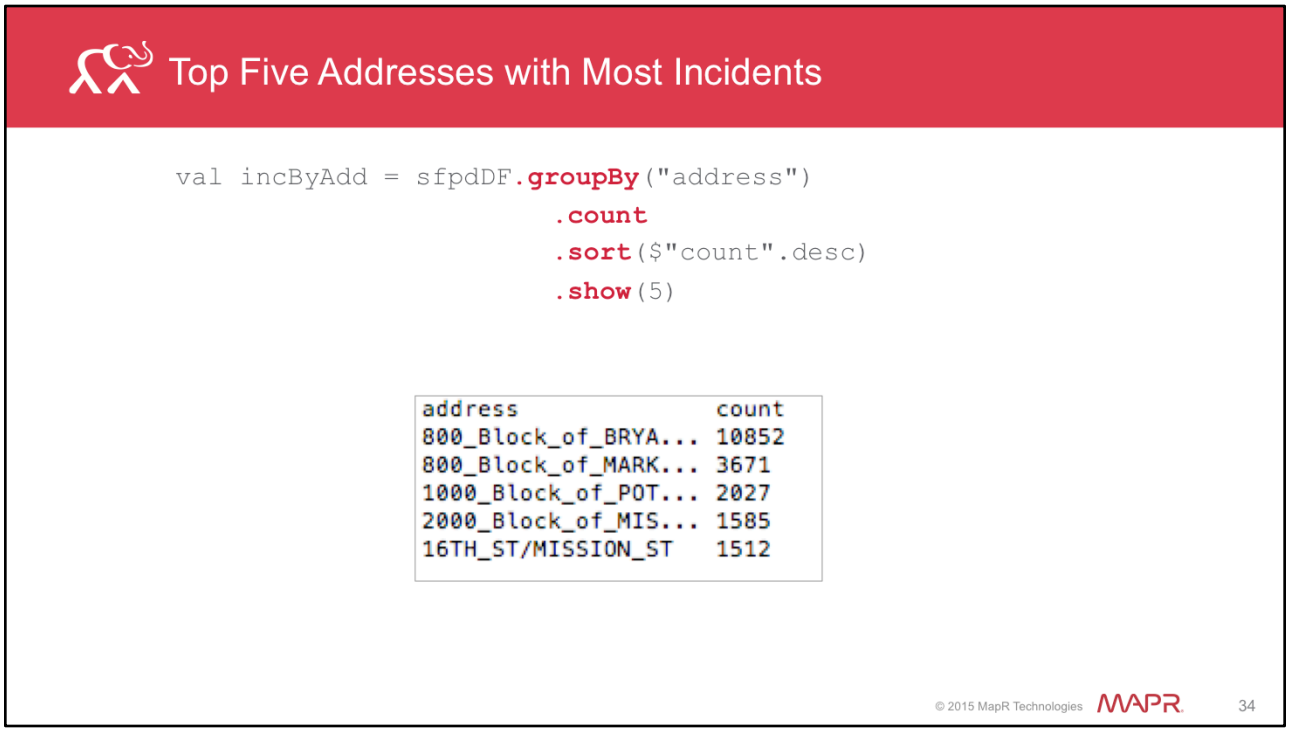
This table lists Language Integrated queries.

Now let us use these actions, functions and language integrated queries to find the answers to questions posed earlier.

Which are the five addresses with the most number of incidents?

To answer this:

1. Create a DataFrame by grouping the incidents by **address**.
2. Count the number of incidents for each address
3. Sort the result of previous step in descending order
4. Show the first five which is the top five addresses with the most incidents.



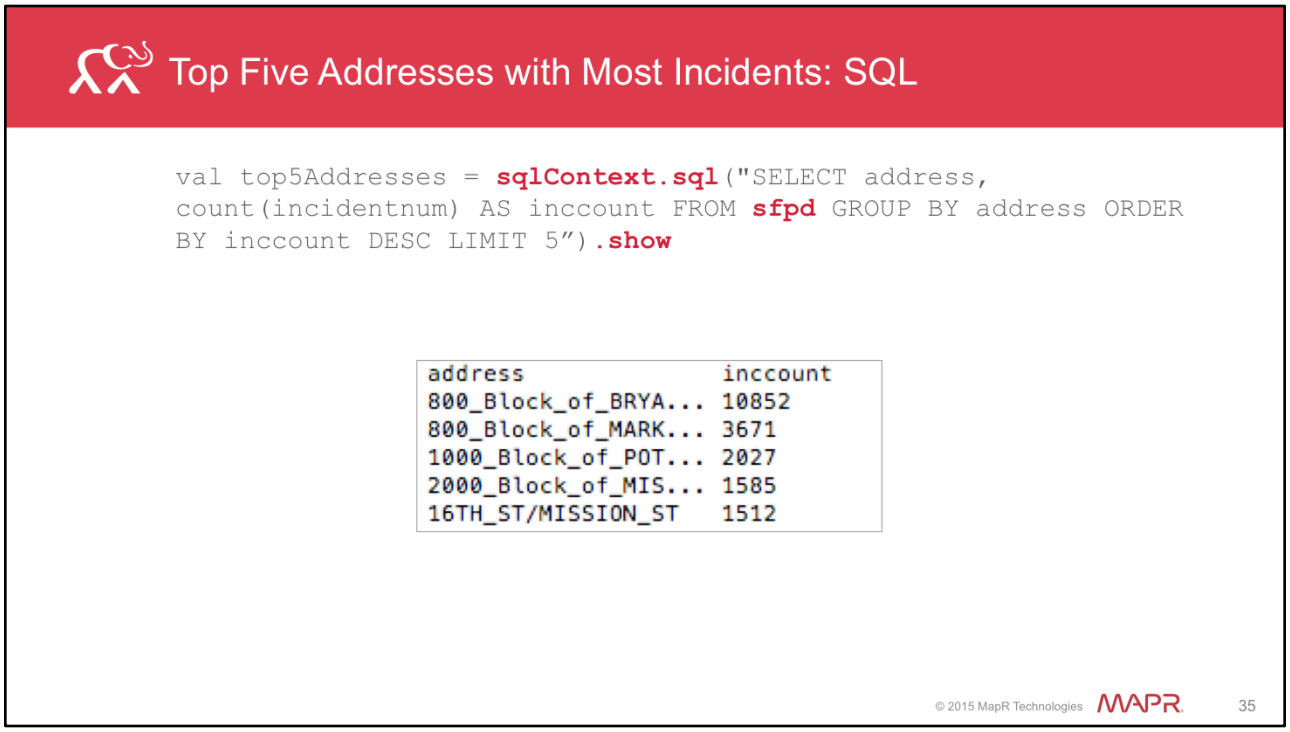
You can combine the statements into one statement. The result is shown here.

DEMO

```
val incByAdd =
sfpdDF.groupBy("address”).count .sort($"count".desc).show(5)

Ask: “What if you want the top 10?”
A: change show(5) to show(10)

Ask: “What if you want categories?”
A: groupBy(“category”)
```



We can answer the same question using SQL as shown here. Note that we are selecting from "sfpd" which is the table that the DataFrame is registered as.

NOTE TO INSTRUCTOR
The key points here are:
1. You need sqlContext.sql
2. You can use SQL queries against the DF registered as a table -hence "sfpd" here.

In the previous statement we used "sfpdDF".

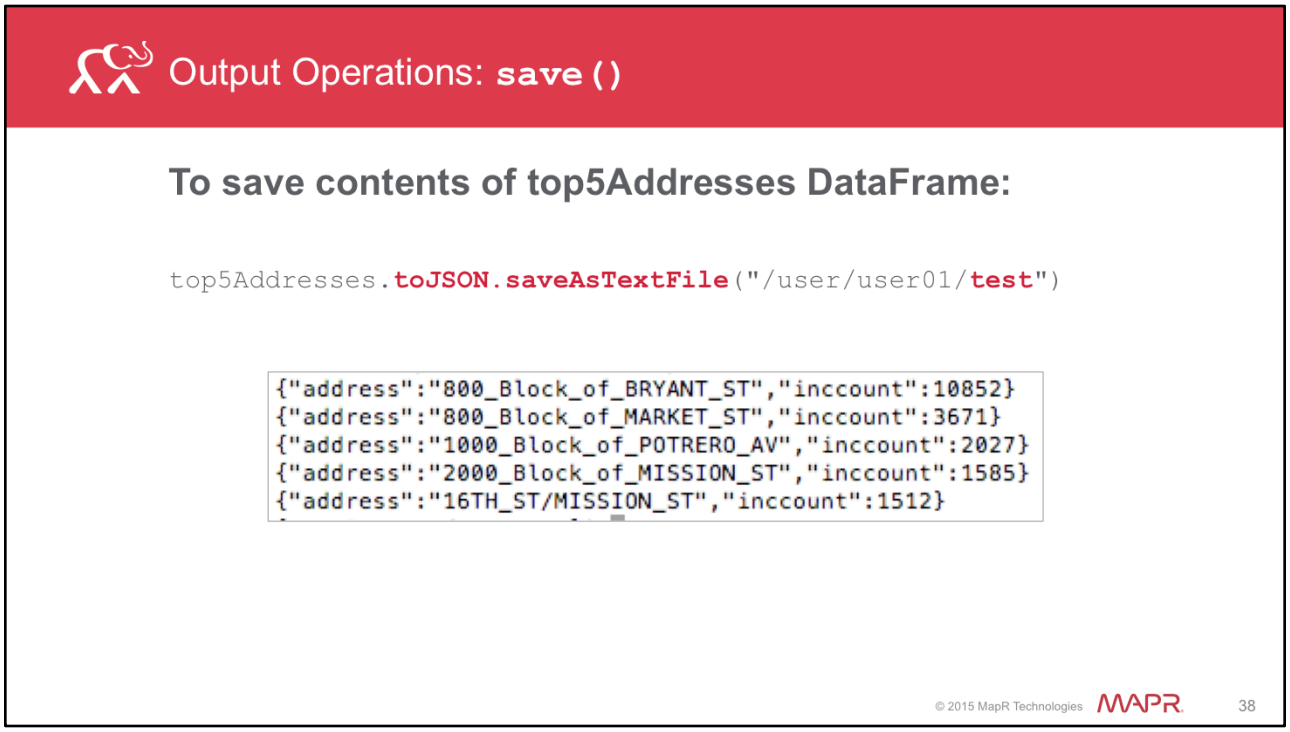
We have just answered the first question. We can use similar logic to answer the remaining questions which will be done in the Lab.

Output Operations: `save()`

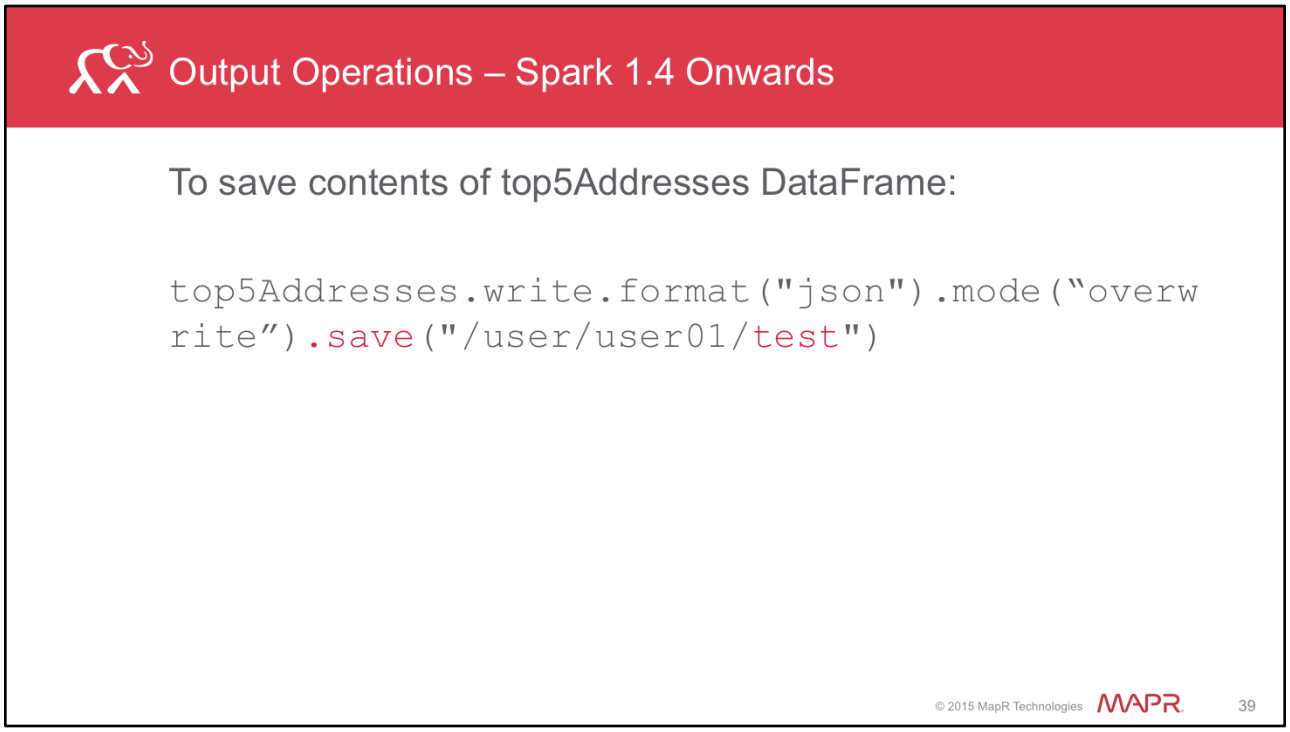
Operation	Description
`Save` (source, mode, options)	Saves contents of DataFrame based on given data sources, savemode and set of options
`insertIntoJDBC` (url,name, overwrite)	Saves contents of DataFrame to JDBC at url under table name table

37

There are times when we want to save the results of our queries. We can use save operations to save data from resulting DataFrames using Spark Data Sources.



The contents of the top5Addresses DataFrame is saved in JSON format in a folder called test.
In this statement, if the folder exists, you will get an error.
In Spark 1.4, the methods have been deprecated. See the link here.
https://spark.apache.org/docs/1.4.1/sql-programming-guide.html#generic-loadsave-functions



In Spark 4.x onwards, save operations have been replaced by the “write” method.
The contents of the top5Addresses DataFrame is saved in JSON format in a folder called test that is created. If the folder exists, you will get an error.

Knowledge Check

Which of the following (in Scala) will give the top 10 resolutions to the console assuming that sfpdDF is the DataFrame registered as a table – sfpd?

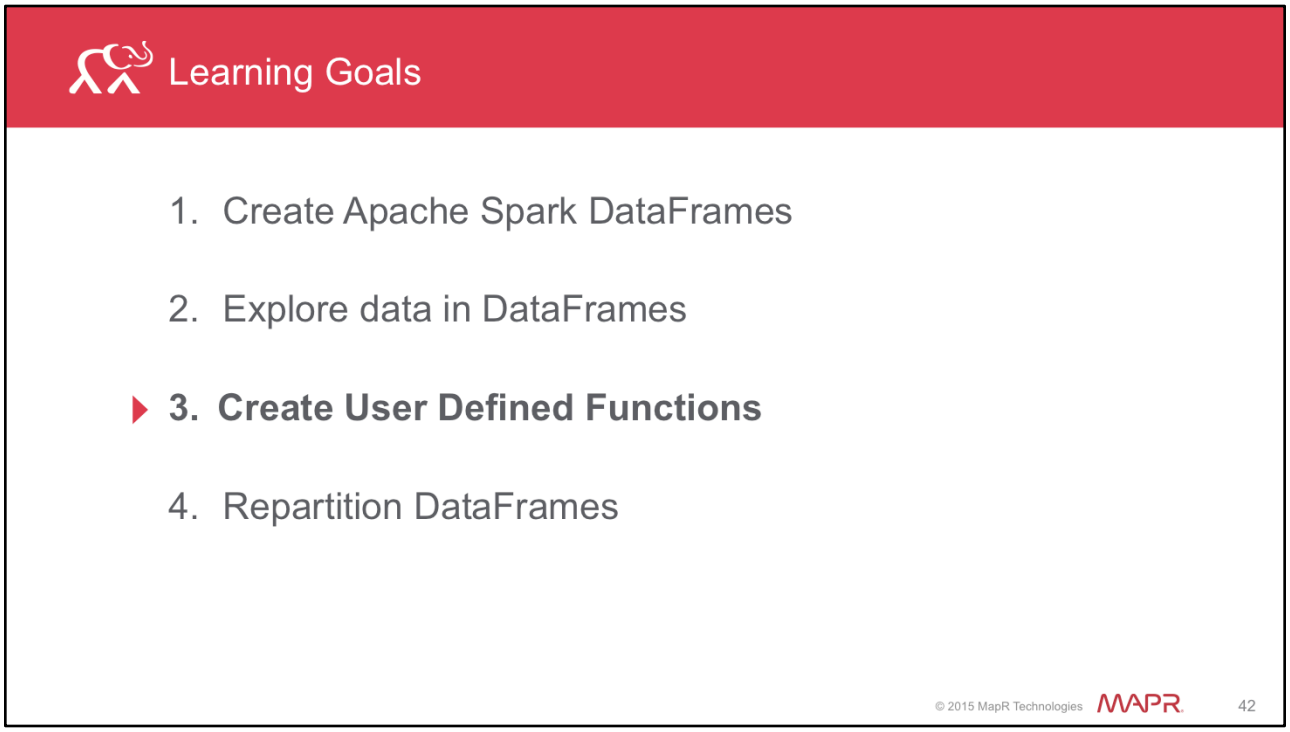
A. `sqlContext.sql("SELECT resolution, count(incidentnum) AS inccount FROM sfpd GROUP BY resolution ORDER BY inccount DESC LIMIT 10")`

B. `sfpdDF.select("resolution").count.sort($"count".desc).show(10)`

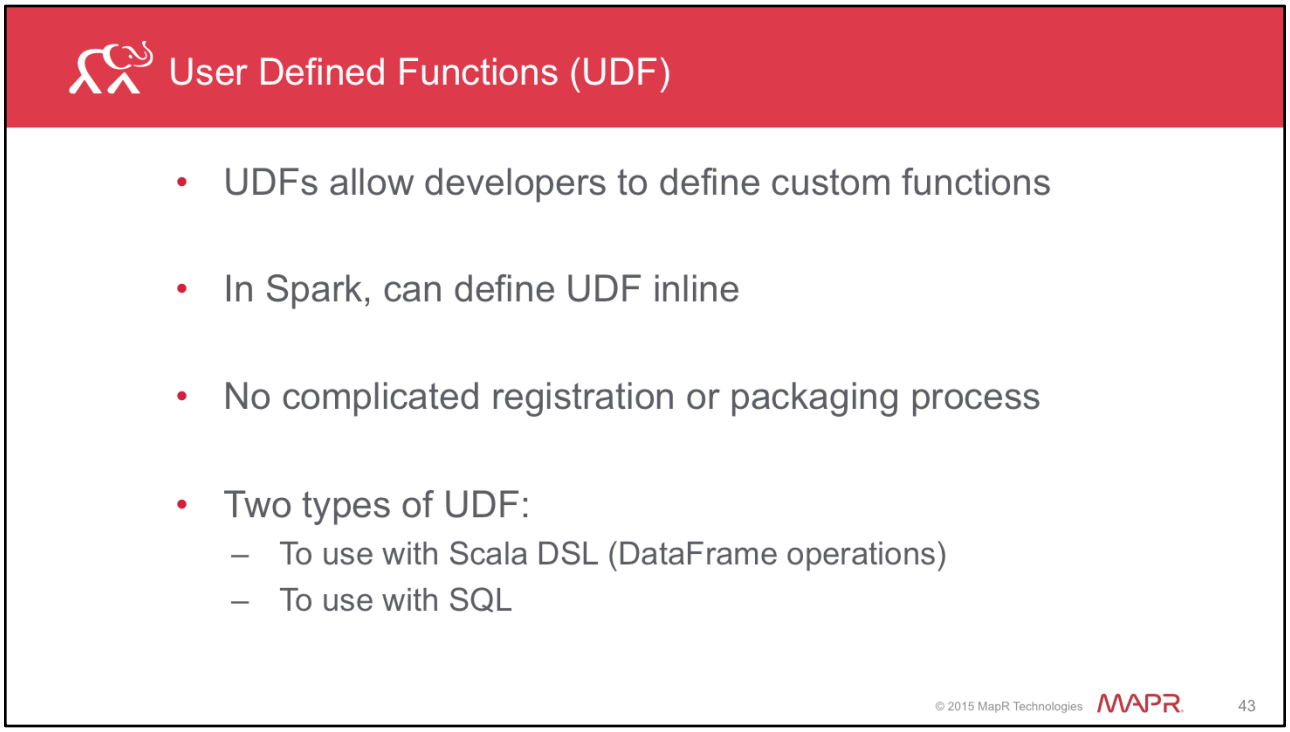
C. `sfpdDF.groupBy("resolution").count.sort($"count".desc).show(10)`

Answer: A, C
A→ using SQL
C→ using DataFrame operations

In this lab, you will explore the dataset using DataFrame operations and SQL queries. You will also save from the DataFrame.



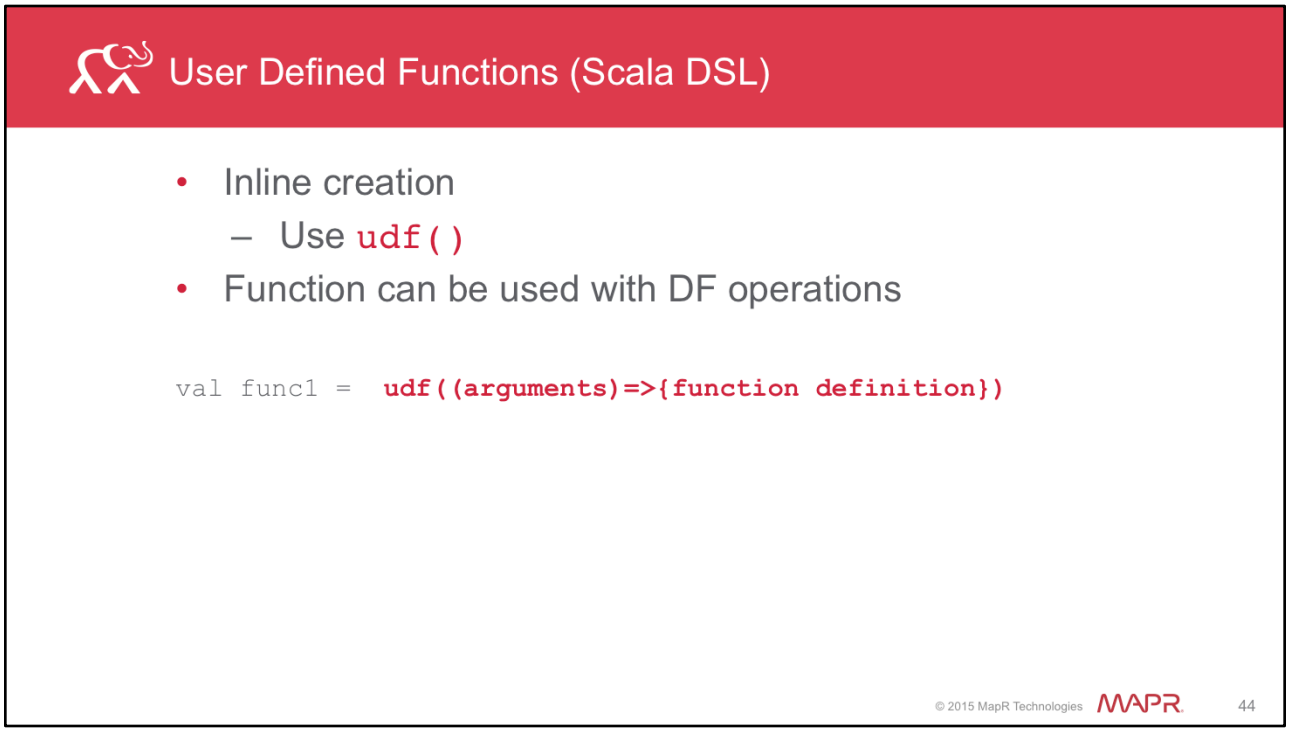
In this section, we will create and use User Defined functions.



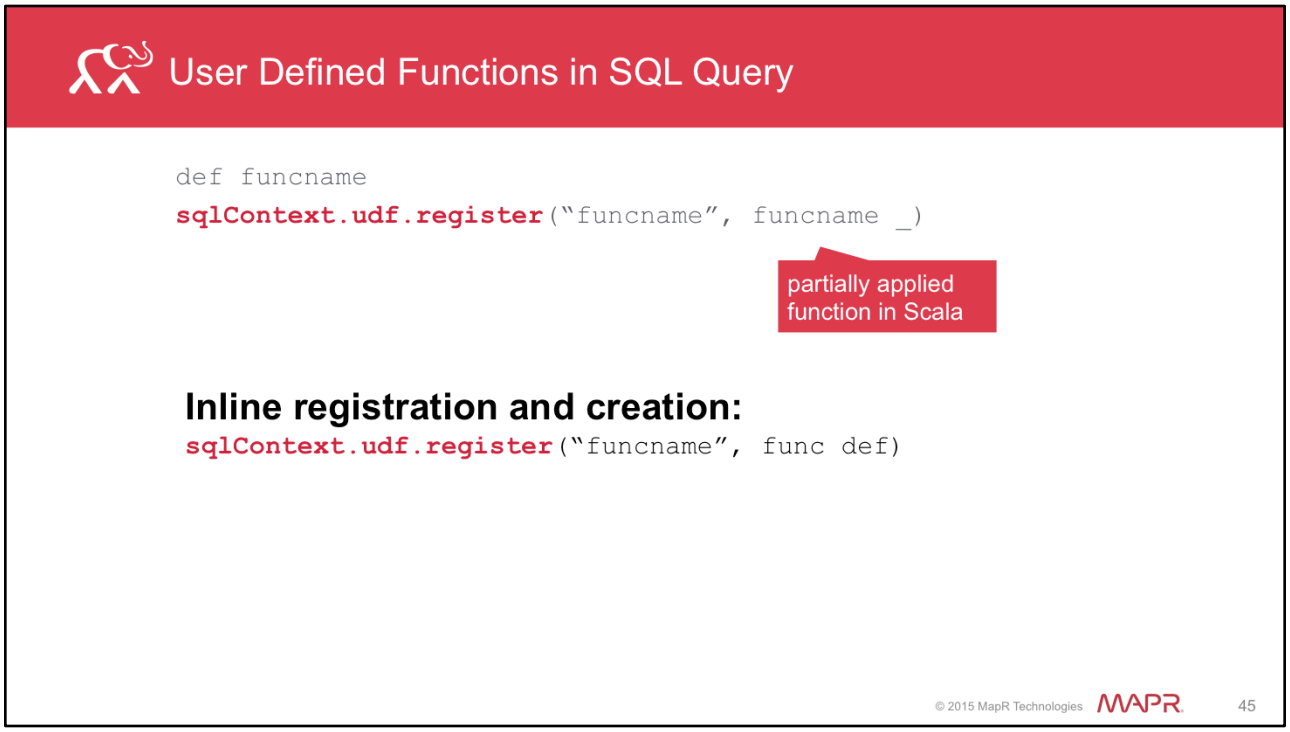
User defined functions allow developers to define custom functions. Spark provides the ability to create UDFs like other query engines.

In Spark, you can define the UDF inline. There is no complicated registration or packaging process.

There are two types of UDFs – one that can be used with the Scala DSL (with the DataFrame operations) and the other that can be used with SQL.

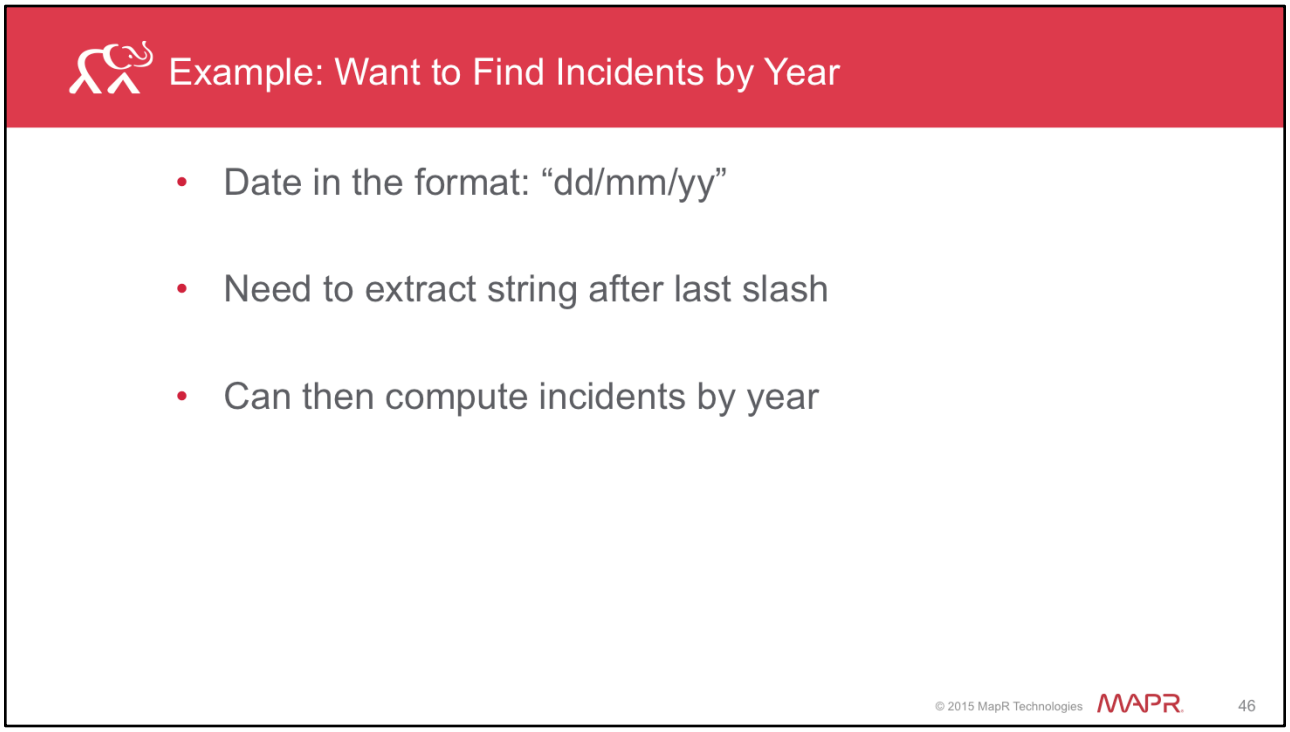


You can define the function as a udf inline. Here we are creating a user defined function called func1 by using udf.

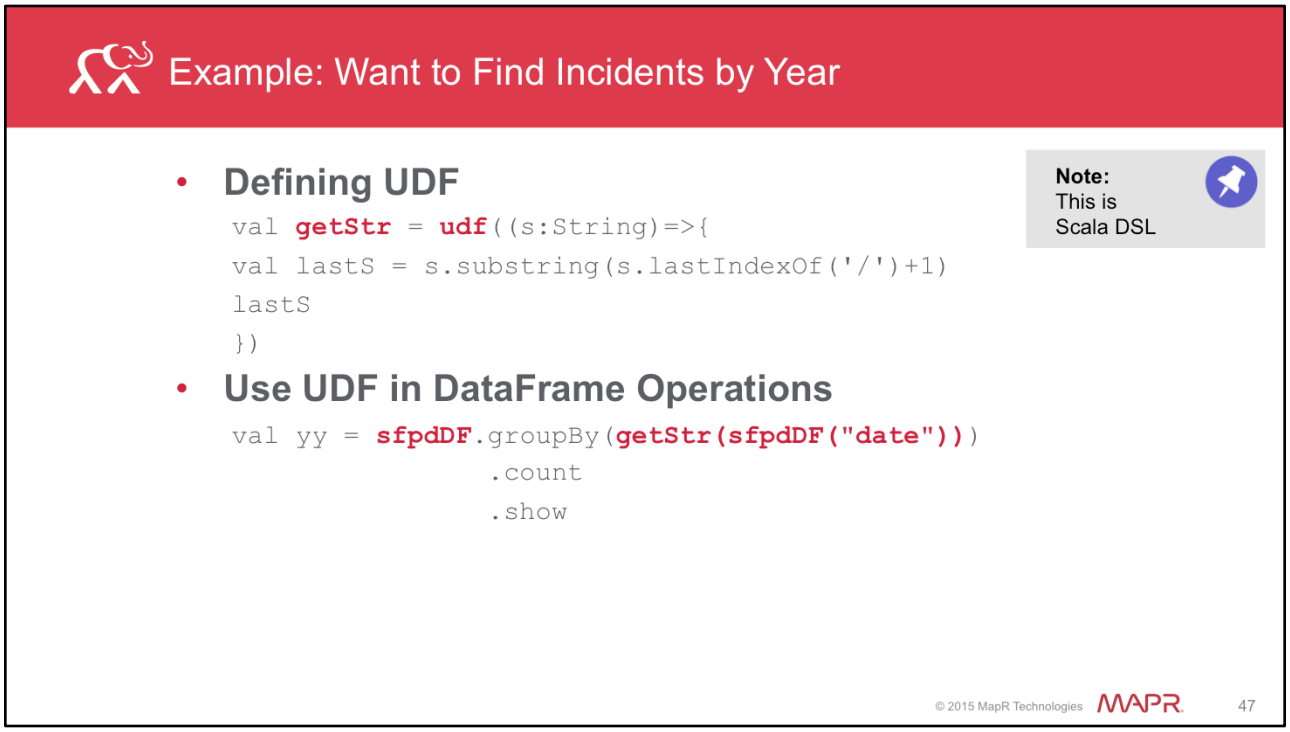


There are two ways to register the udf to be used with SQL.

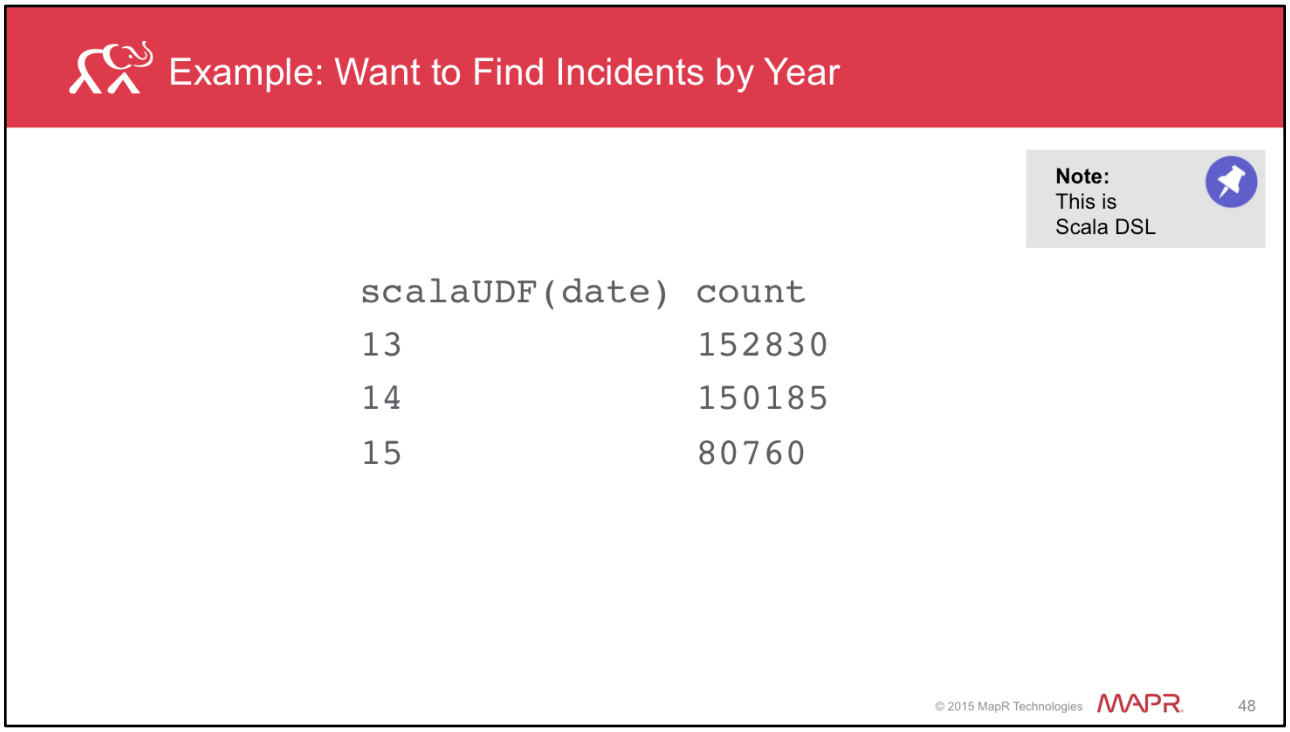
1. You can define the function first and then register using sqlContext.udf.register. This accepts two arguments: the function name as a literal, and the second parameter is the function name followed by an underscore as shown. The underscore (must have a space in between function and underscore) turns the function into a partially applied function that can be passed to register. The underscore tells Scala that we don't know the parameter yet but want the function as a value. The required parameter will be supplied later.
2. The second way is to define the function inline in the registration.



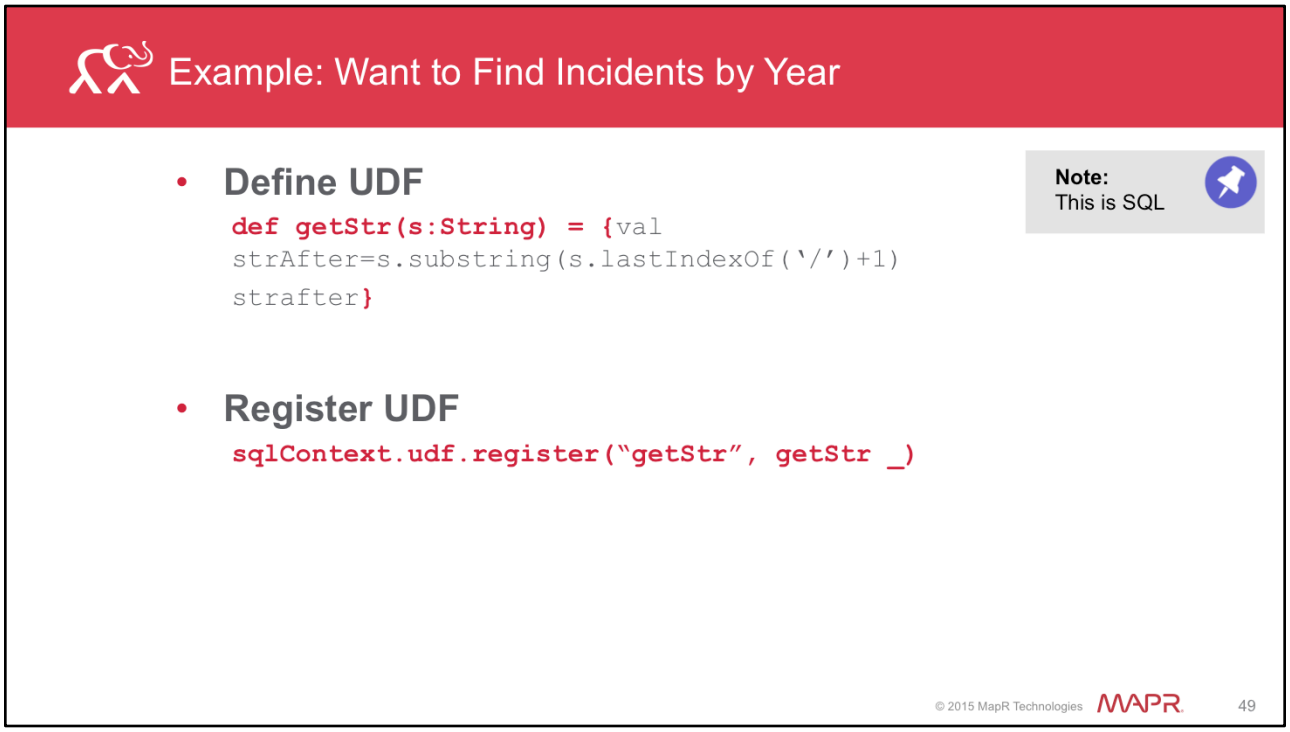
The date in the SFPD data is a string of the form “dd/mm/yy”. In order to group or do any aggregation by year, we have to extract the year from the string.



In this example, we are defining a function.

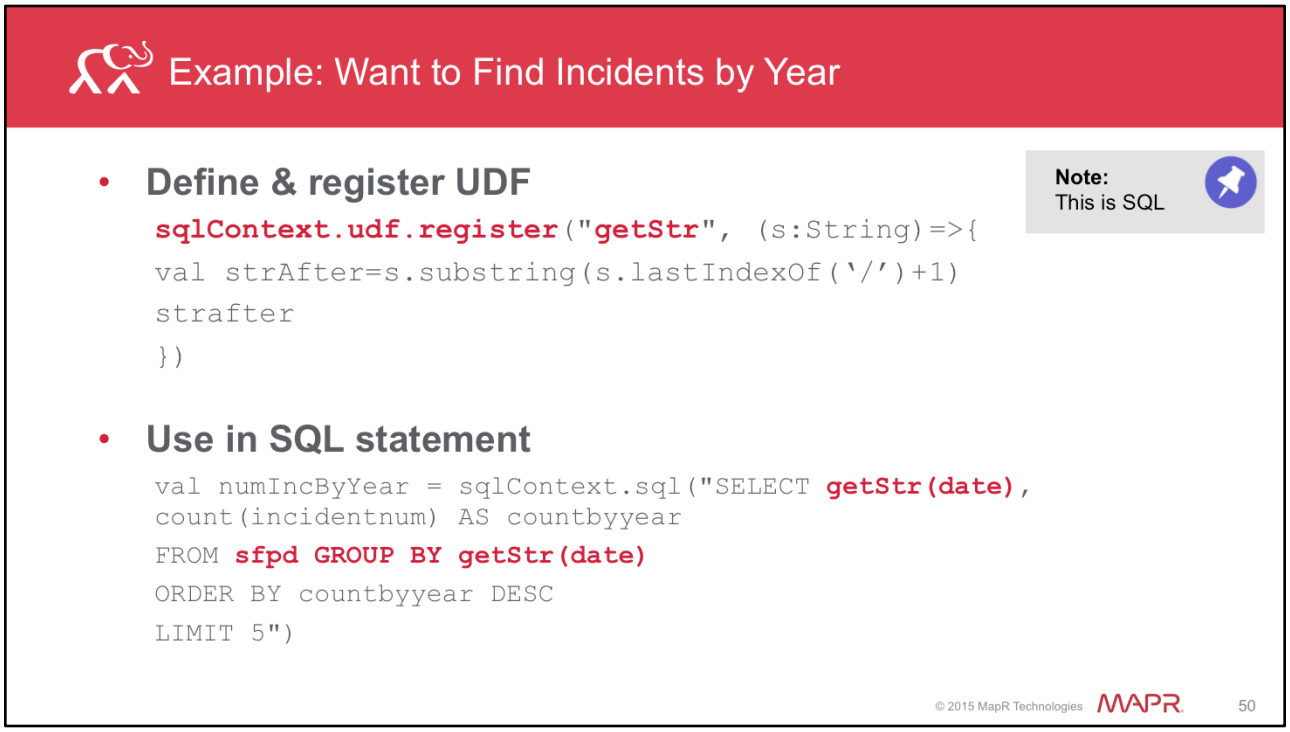


This is the number of incidents by year.

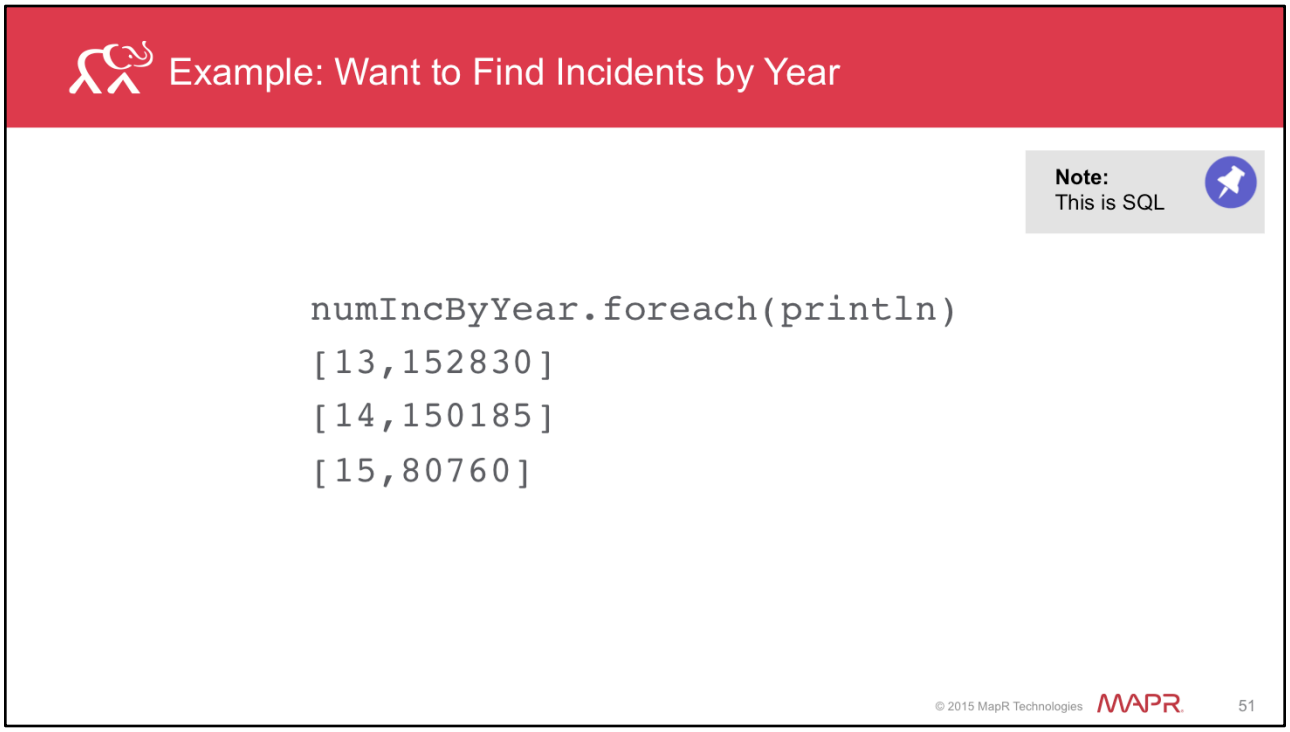


The date is the SFPD data is a string of the form “dd/mm/yy”. In order to group or do any aggregation by year, we have to extract the year from the string.

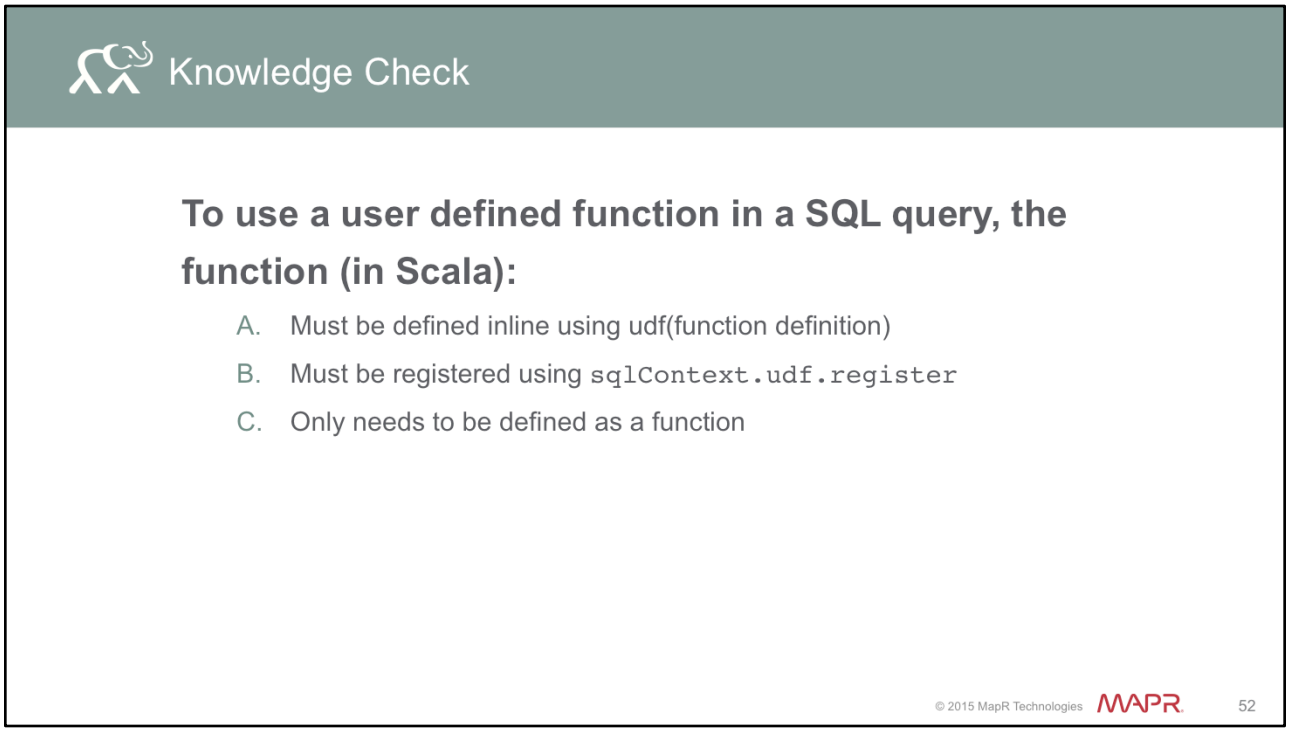
In this example, we are defining a function getStr and then registering it as a user defined function.



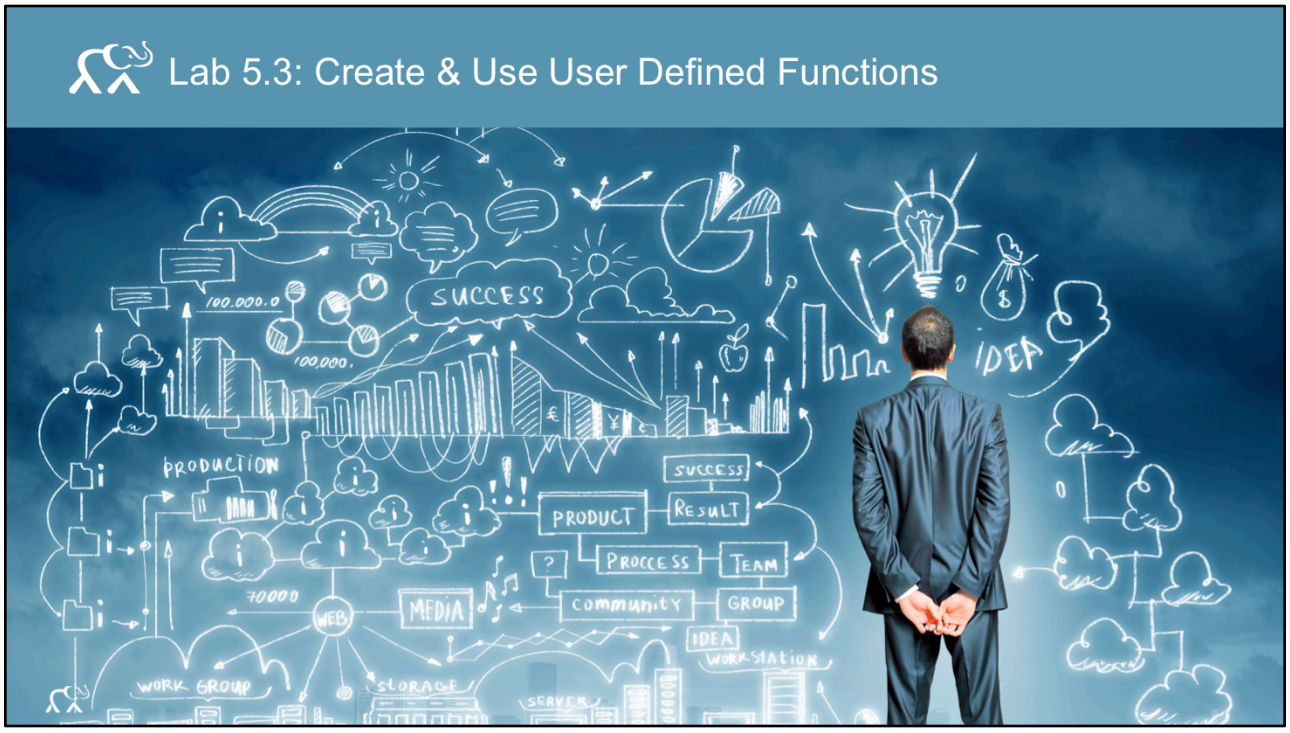
This is the same function now defined and registered as a udf to be used in SQL statements.



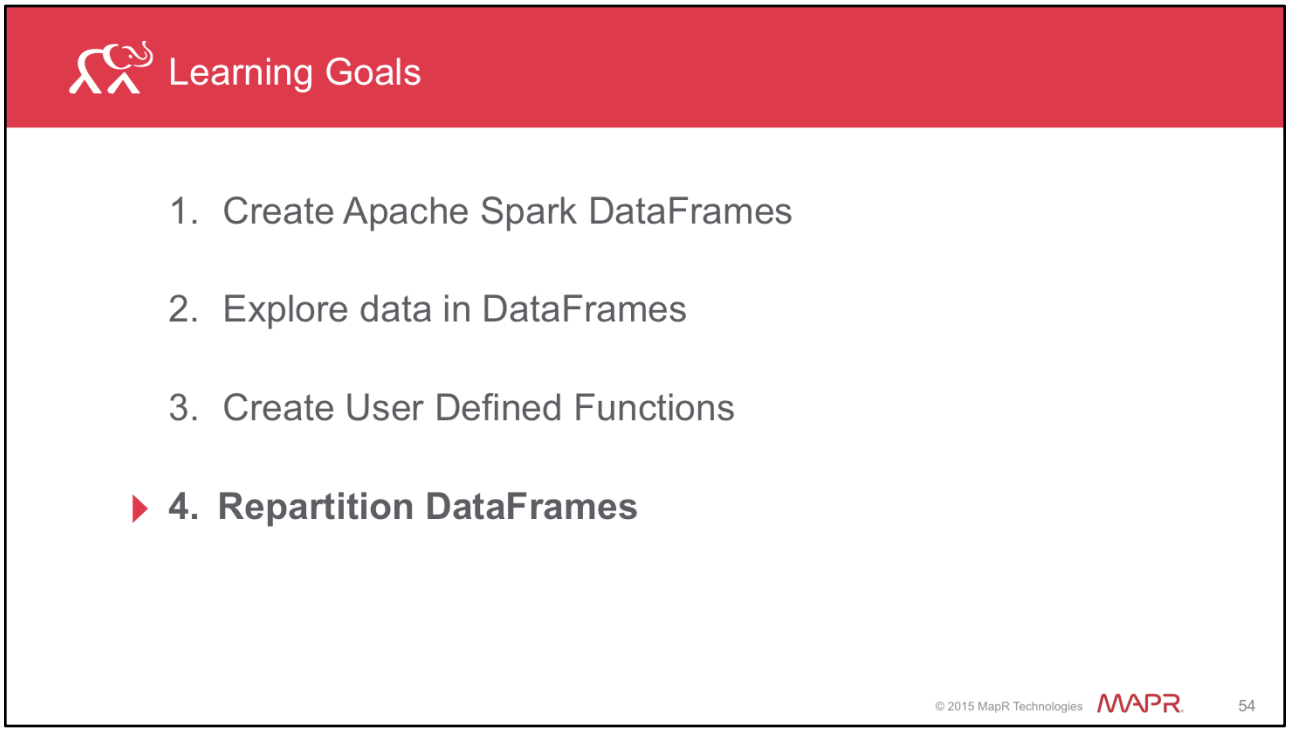
As you can see, we get the same result as before for the number of incidents by year.


Knowledge Check
To use a user defined function in a SQL query, the function (in Scala):
A. Must be defined inline using udf(function definition)
B. Must be registered using sqlContext.udf.register
C. Only needs to be defined as a function
© 2015 MapR Technologies
52

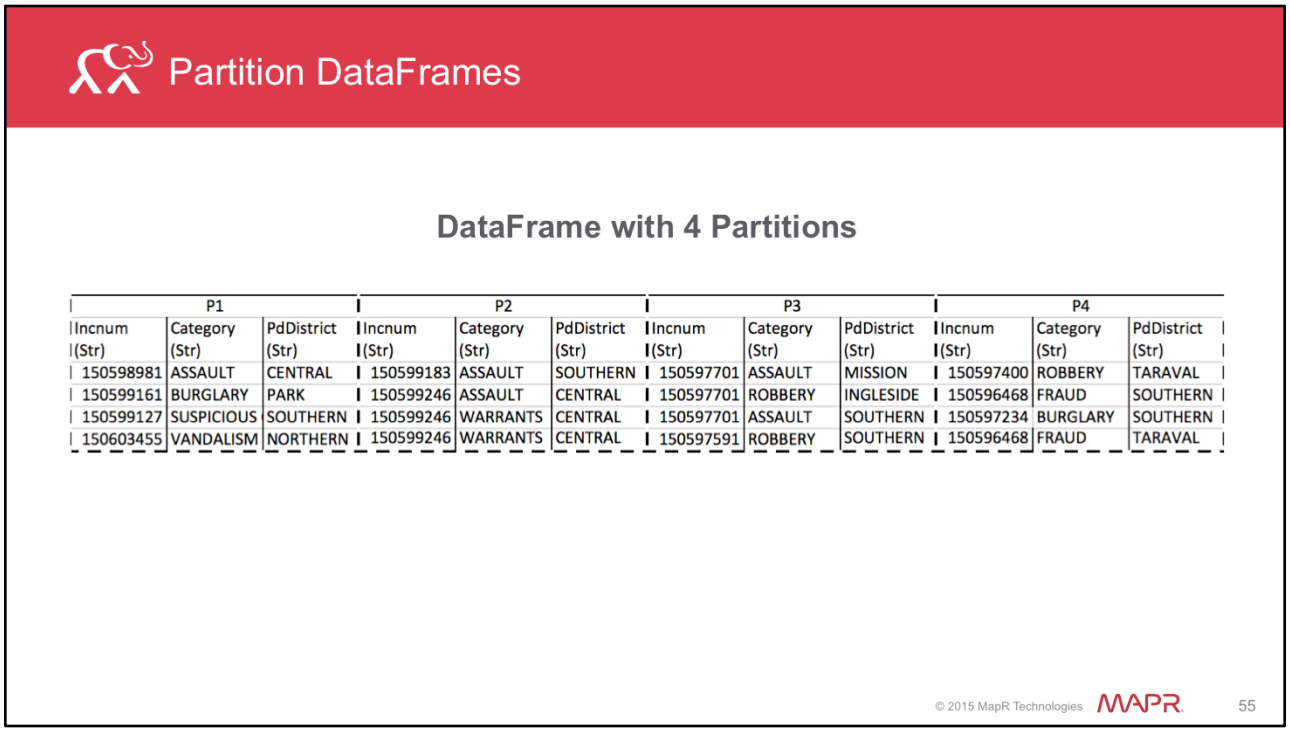
Answers: 2



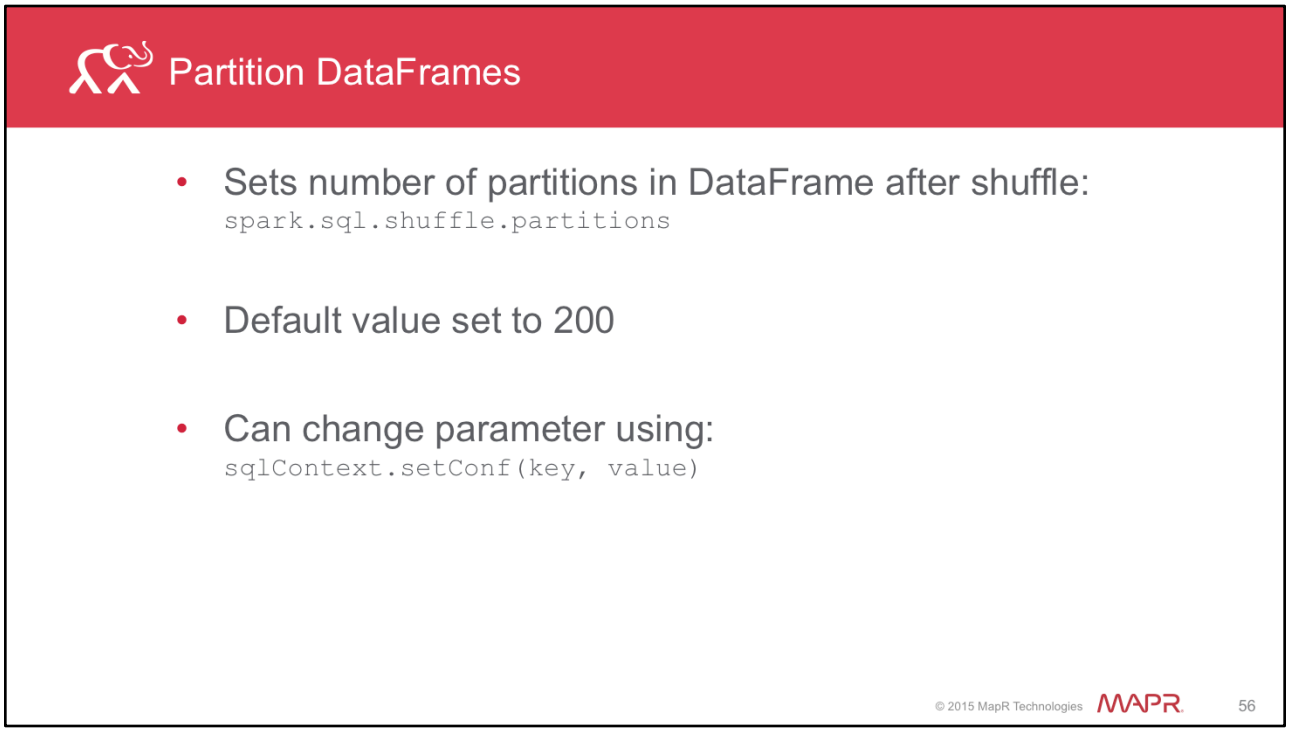
In this activity, you will create and use a user defined function.



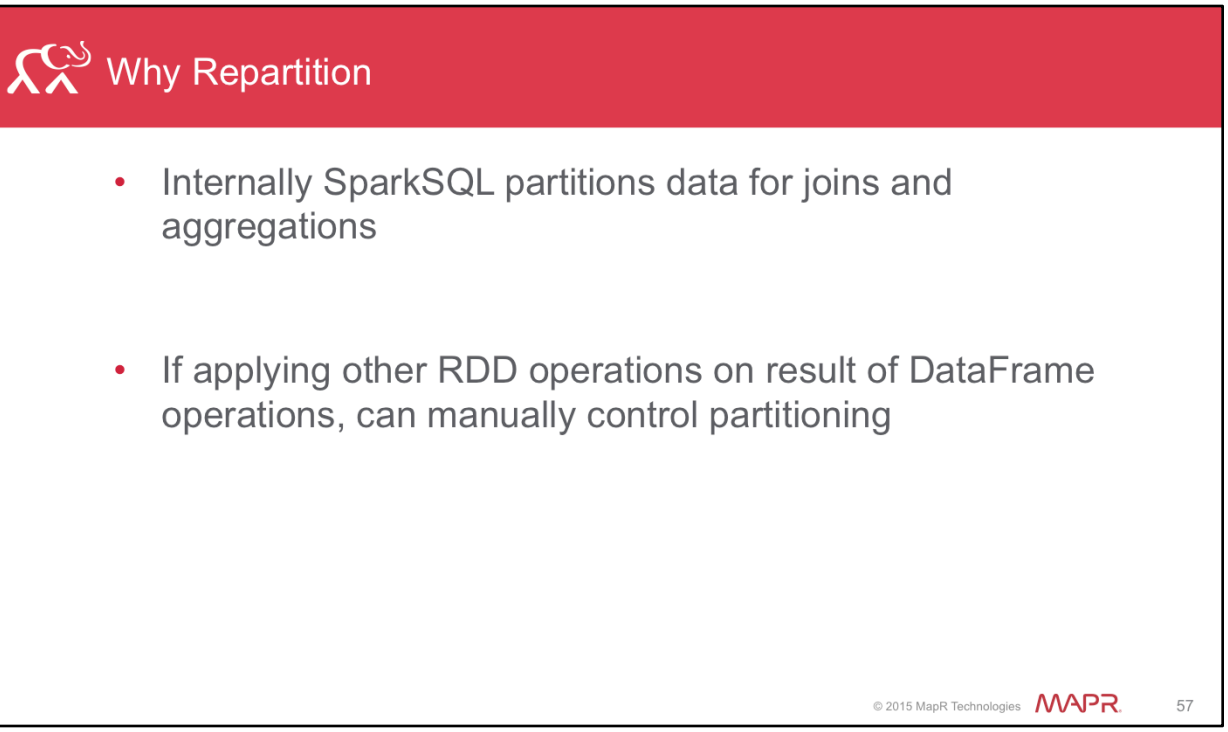
In this section, we look at repartitioning DataFrames.



This is a DataFrame containing the SFPD data. This DataFrame has four partitions.

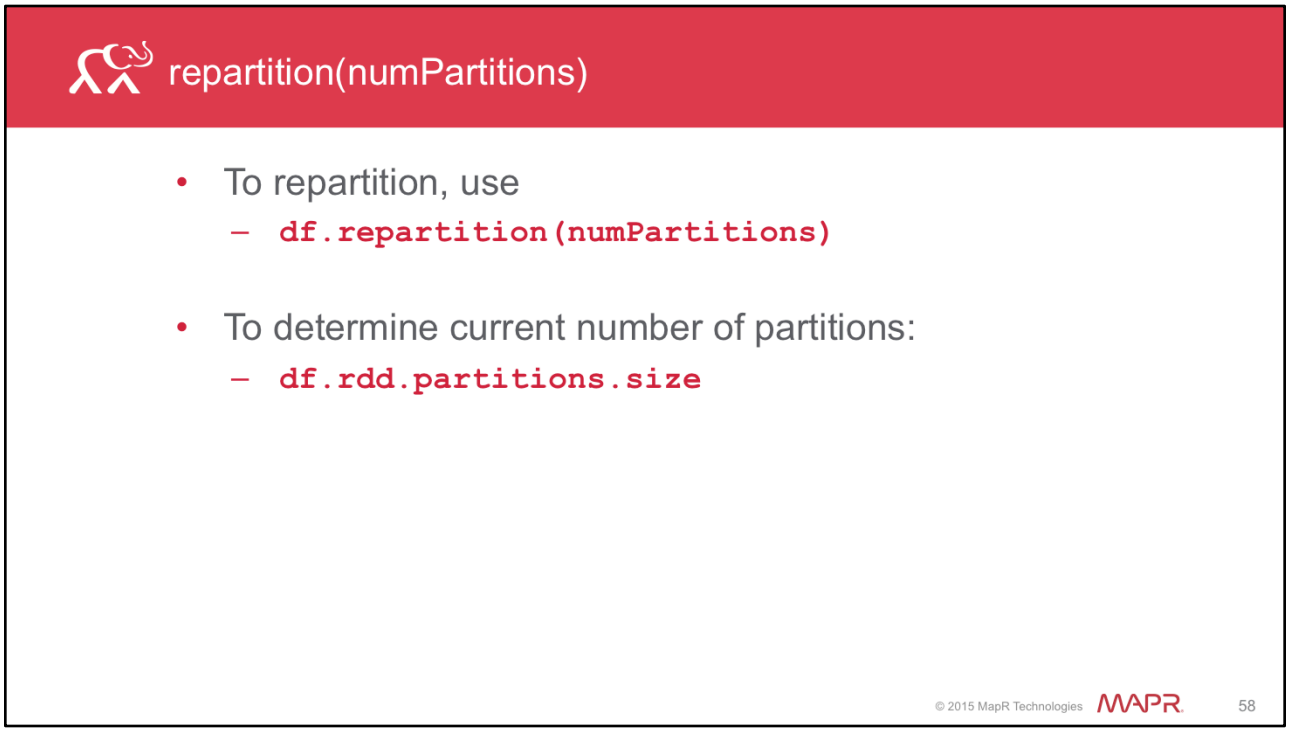


By default in Spark SQL, there is a parameter called spark.sql.shuffle.partitititions, which sets the number of partitions in a DataFrame after a shuffle (in case the user hasn't manually specified it). Currently, Spark does not do any automatic determination of partitions, it just uses the number in that parameter. We can change this parameter using: sqlContext.setConf(key, value).

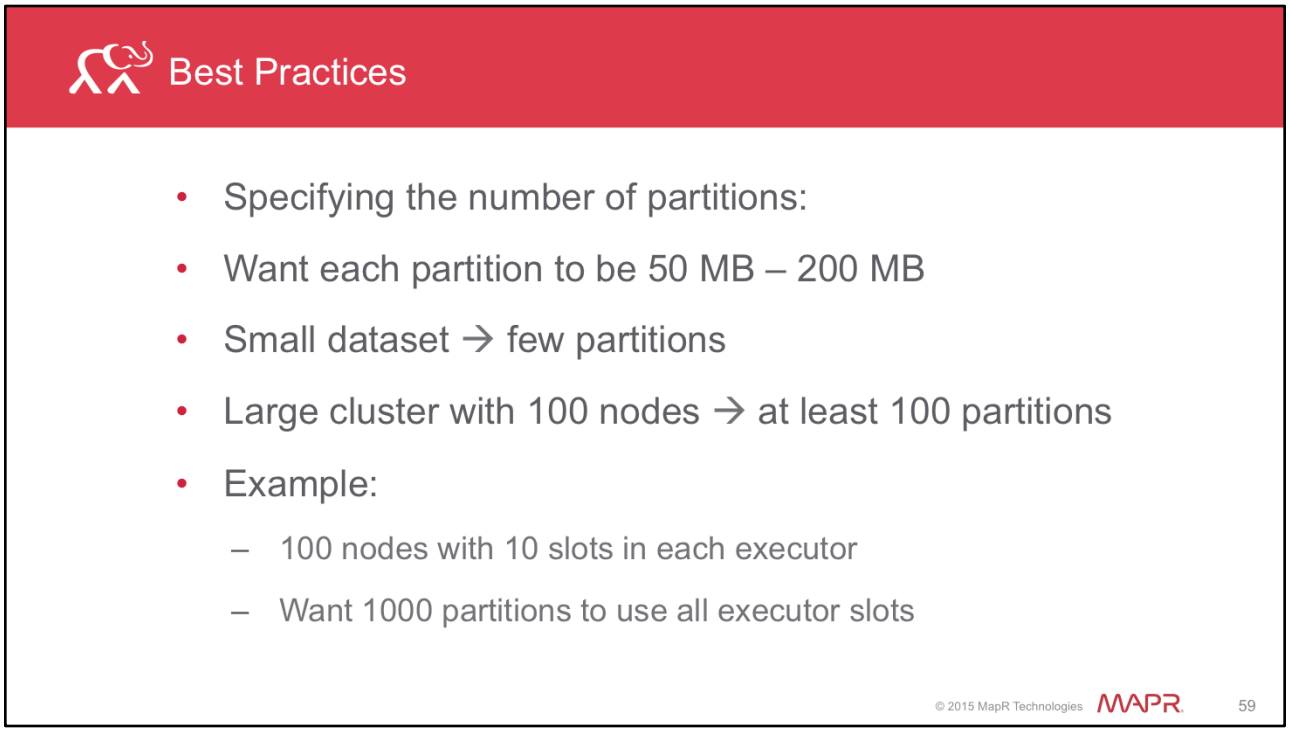


Internally Spark SQL will add exchange operators to make sure that data is partitioned correctly for joins and aggregations. Optimizing partitioning can improve performance.

If we want to apply other RDD operations on the result of the DataFrame operations, we have the option to manually control the partitioning.



You can use the DataFrame.repartition(numPartitions) method to repartition a DataFrame. To determine the current number of partitions, use the method, df.rdd.partitions.size.



Ideally we want each partition to be around 50 MB - 200 MB. If the dataset is really small, then having just a few partitions may be fine.

If we have a large cluster with 100 nodes and 10 slots in each Executor, then we want the DataFrame to have 1,000 partitions to use all of the Executor slots to process it simultaneously. In this case, it's also fine to have a DataFrame with thousands of paritititions, and a 1,000 partitions will be processed at a time.

Knowledge Check

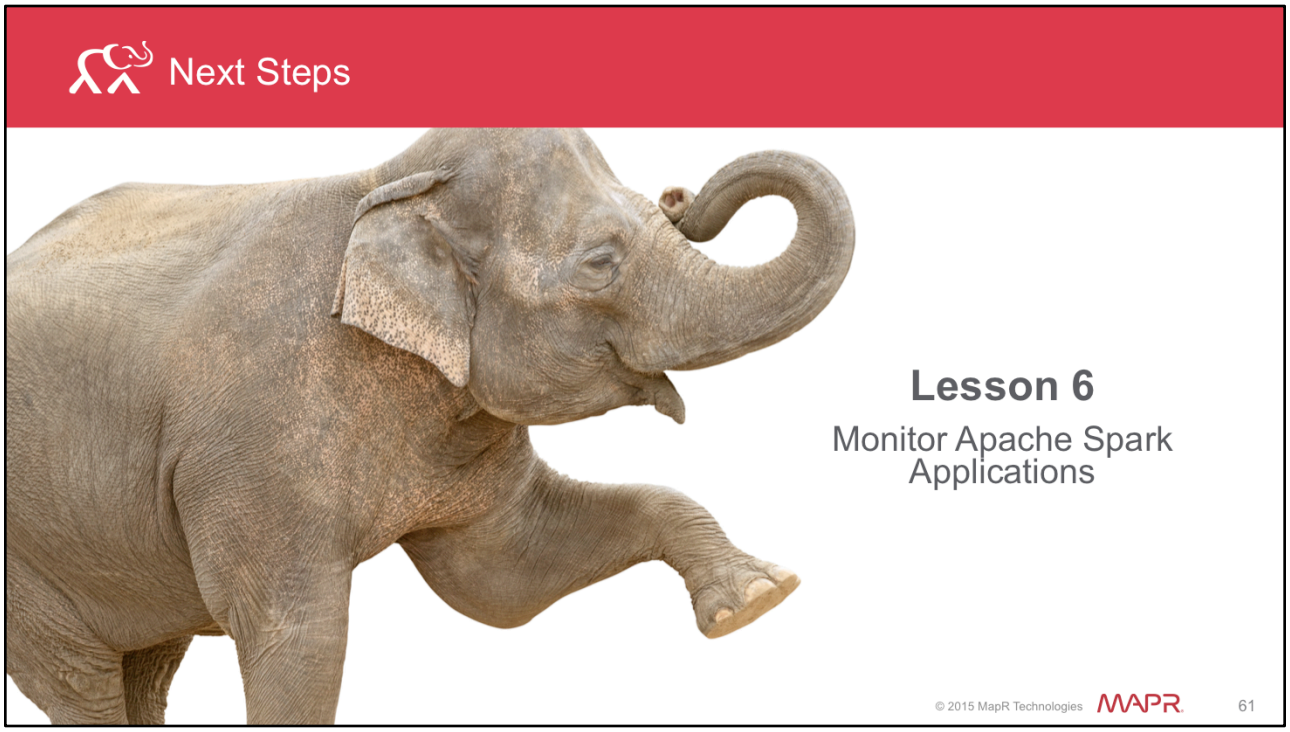
To find the number of partitions in the DataFrame, use (in Scala)

A. `df.numPartitions()`

B. `df.rdd.partitions.size()`

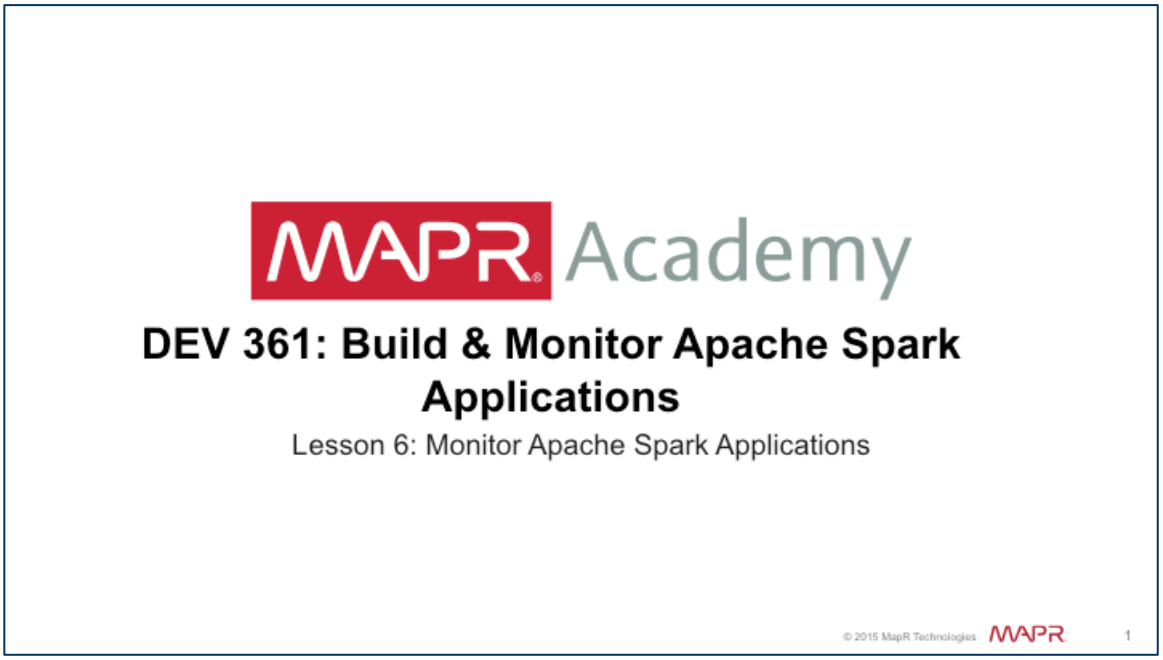
C. `rdd.partitions.size()`

Answers: B



Congratulations! You have completed Lesson 5 of this course– DEV 361 – Build and Monitor Apache Spark Applications.

Lesson 6: Monitor Apache Spark Applications



Notes on Slide 1:

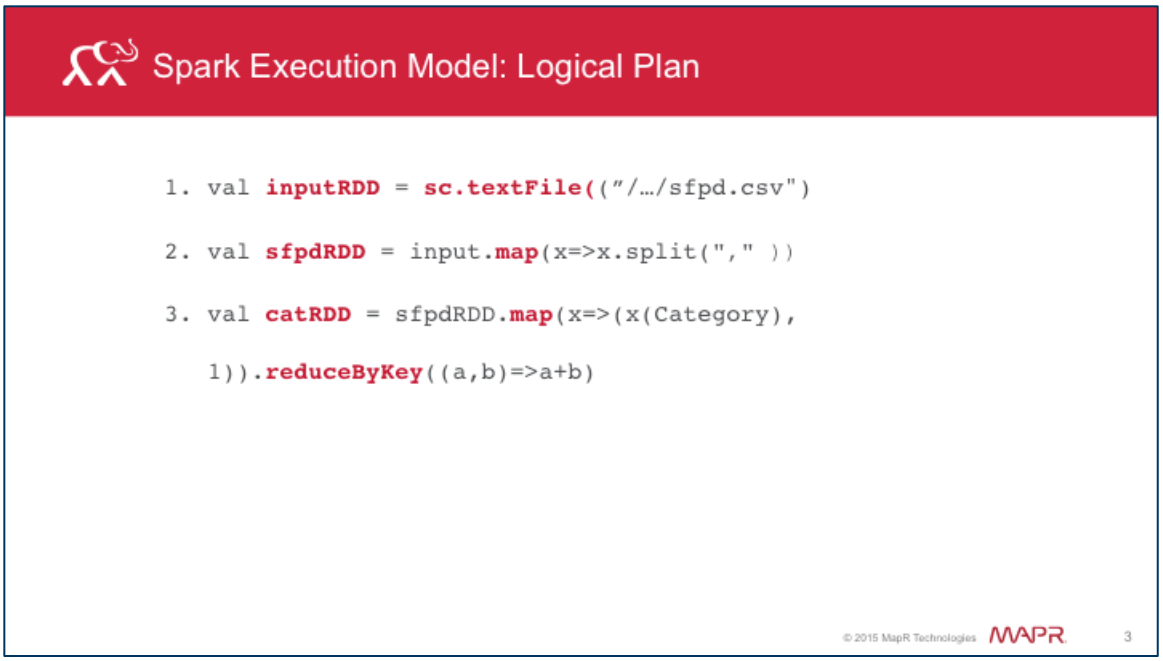
Welcome to Build and Monitor Apache Spark Applications, Lesson 6- Monitor Apache Spark Applications.

Notes on Slide 2:

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

1. Describe components of the Spark execution model
2. Use Spark Web UI to monitor Spark applications
3. Debug and tune Spark Applications

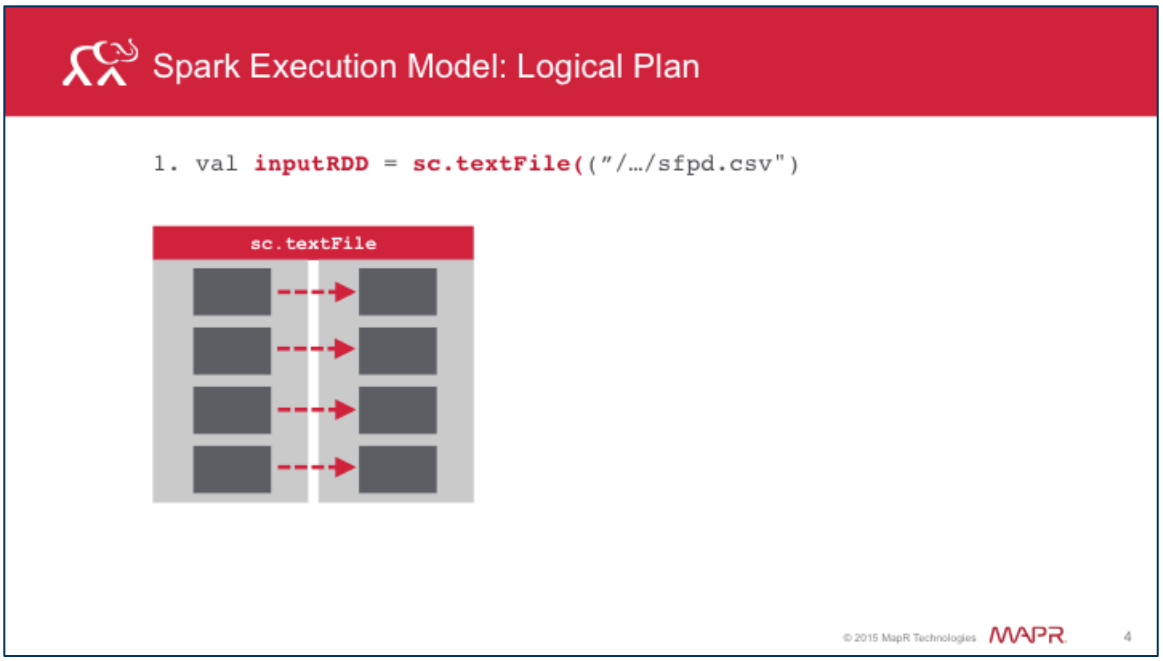
In this section, we will look at the components of the Spark execution model.



Notes on Slide 3:

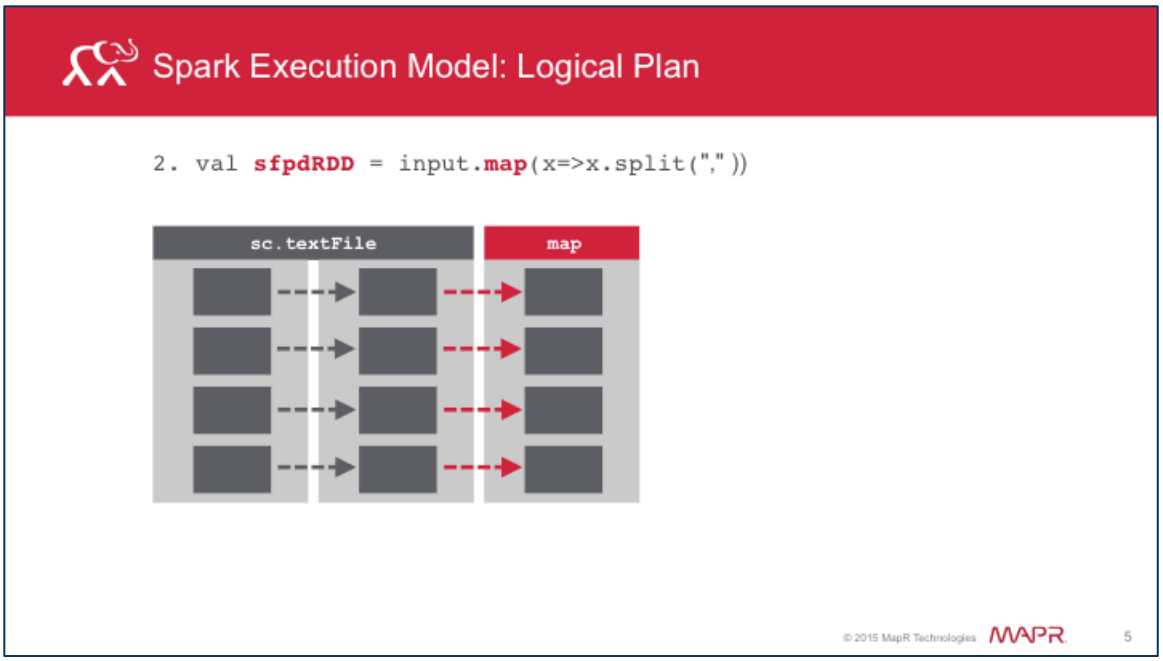
We are going to see how a user program translates into the units of physical execution. Let us first take a look at the logical plan.

Consider the example from an earlier lesson where we load SFPD data from a csv file. We will use this as an example to walk through the components of the Spark execution model.



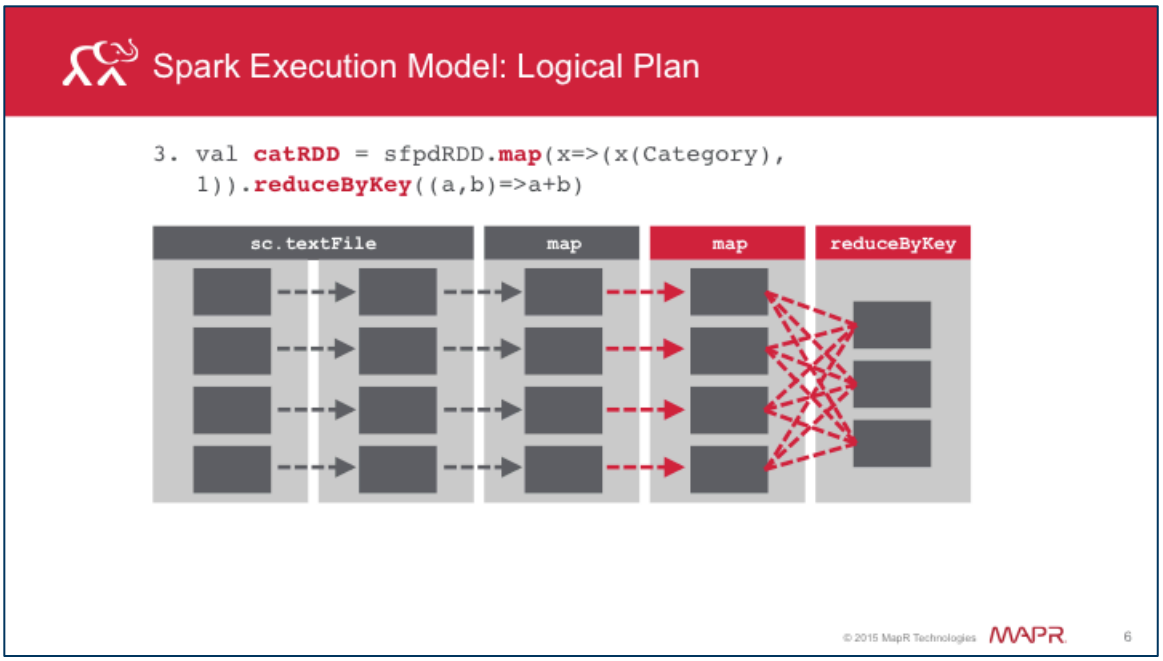
Notes on Slide 4:

- The first line creates an RDD called inputRDD from the sfpd.csv file.



Notes on Slide 5:

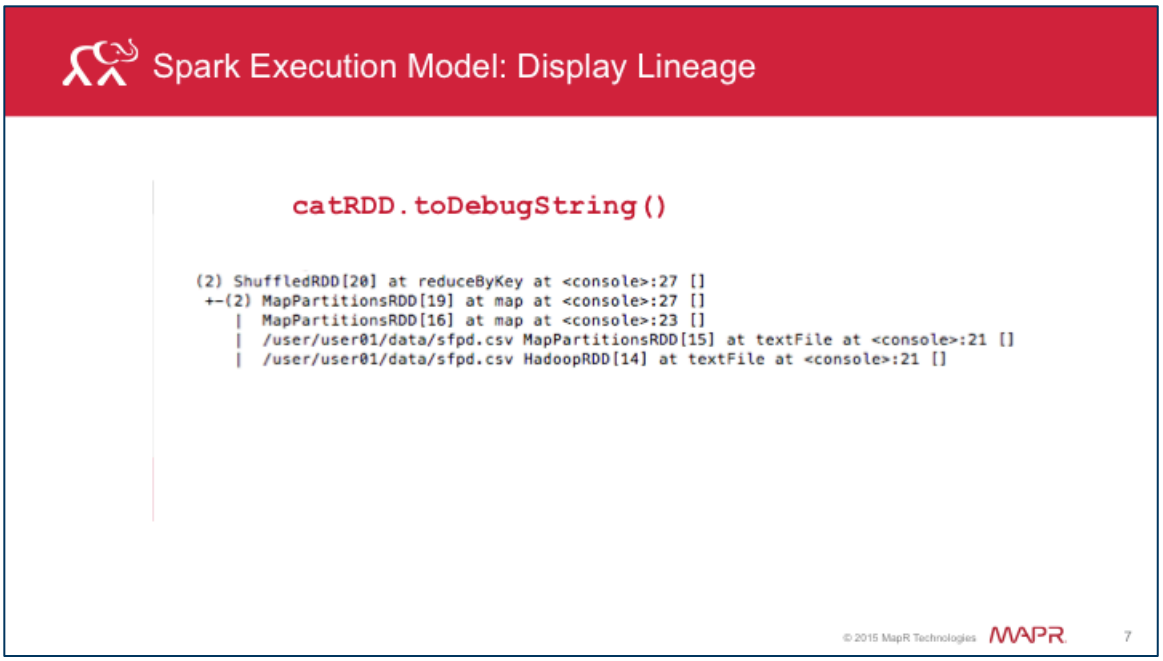
- The second line creates an RDD – sfpdRDD which splits the data in the input RDD based on the comma separator.



Notes on Slide 6:

- The third statement creates the catRDD by applying the map and reduceByKey transformations.

No actions have been performed yet. Once Spark executes these lines, it defines a Directed Acyclic Graph (DAG) of these RDD objects. Each RDD maintains a pointer to its parent(s) along with the metadata about the type of relationship. RDDs use these pointer to trace its ancestors.

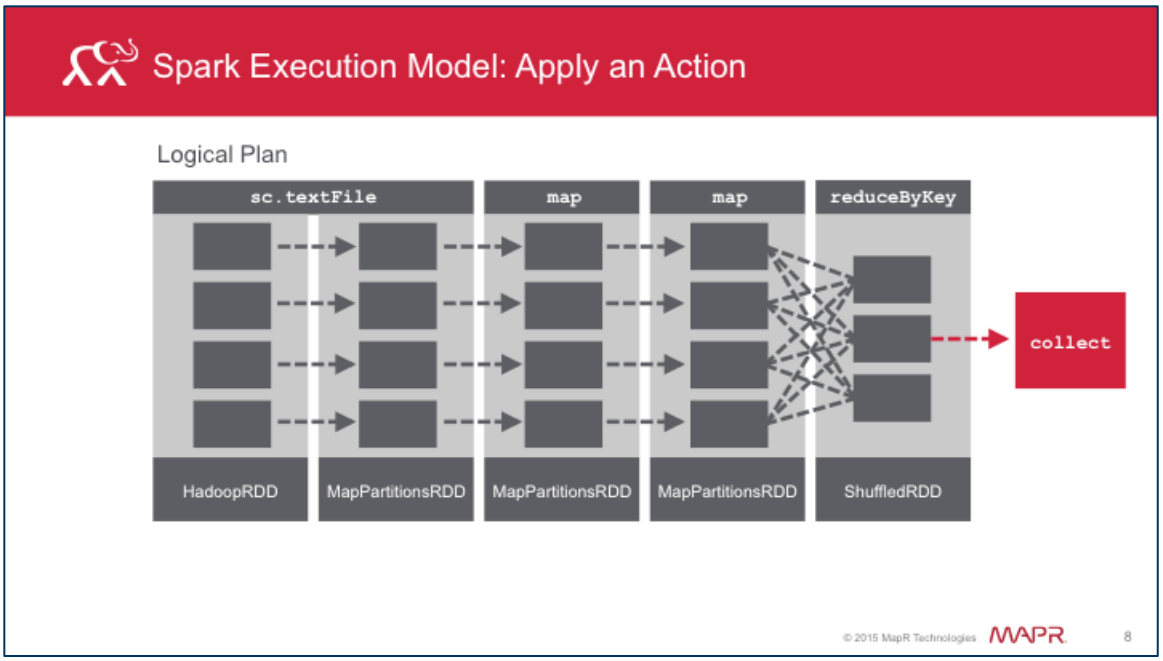


Notes on Slide 7:

To display the lineage for an RDD, use rdd.toDebugString. In this example, to display the lineage for cat RDD, use catRDD.toDebugString. The lineage displays all the ancestors of cat RDD.

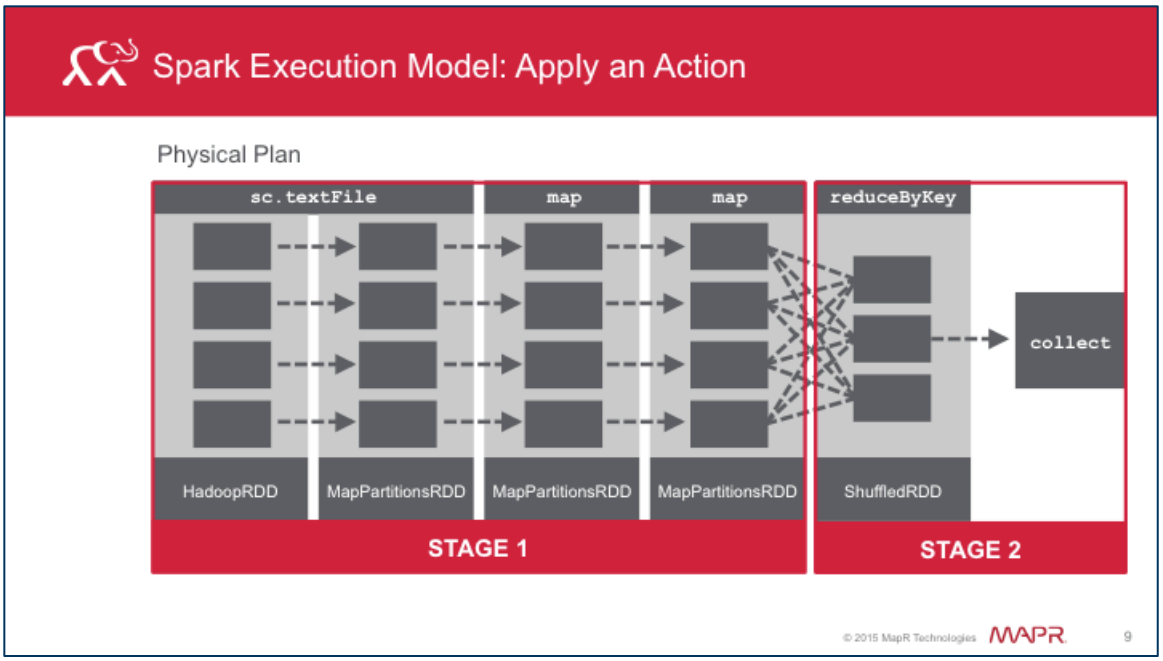
sc.textFile first creates a HadoopRDD and then the MapPartitions RDD. Each time we apply the map transformation, it results in a MapPartitions RDD. When we apply the reduceByKey transformation, it results in a ShuffledRDD.

No computation has taken place yet as we have not performed any actions.



Notes on Slide 8:

Now we add a collect action on the catRDD. The collect action triggers a computation. The Spark scheduler creates a physical plan to compute the RDDs needed for the computation. When the collect action is called, every partition of the RDD is materialized and transferred to the driver program. The Spark scheduler then works backwards from the catRDD to create the physical plan necessary to compute all ancestor RDDs.

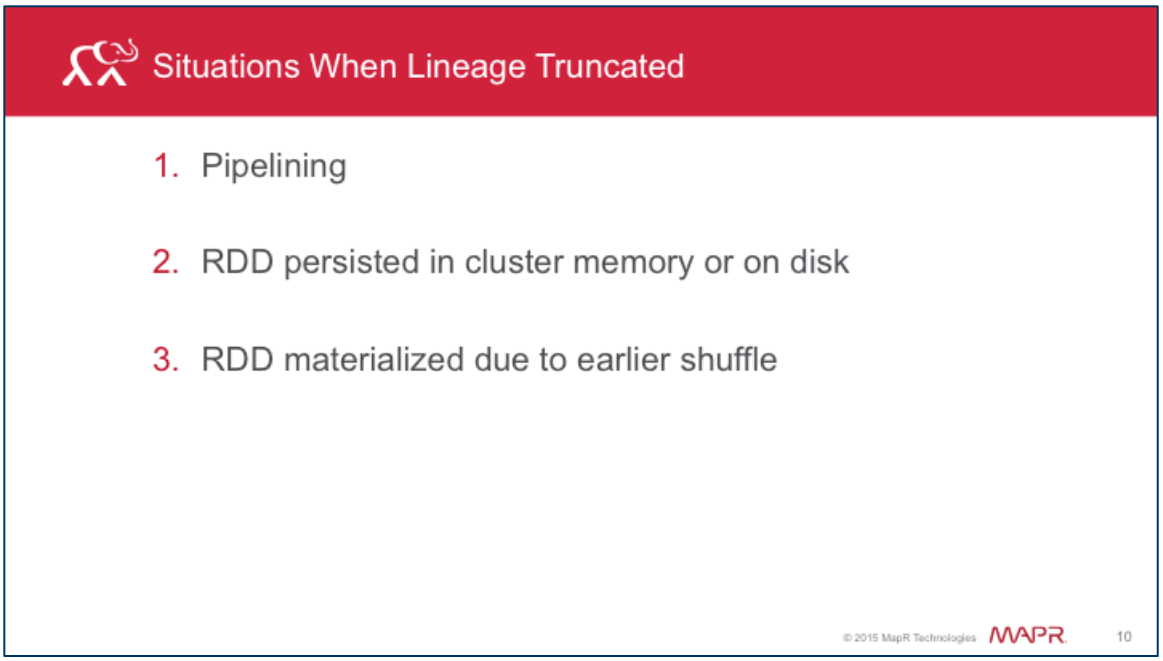


Notes on Slide 9:

The scheduler usually outputs a computation stage for each RDD in the graph. However, when an RDD can be computed from its parent without movement of data, multiple RDDs are collapsed into a single stage. The collapsing of RDDs into one stage is called **pipelining**.

In the example, the map operations are not moving any data and hence the RDDs have been pipelined into stage 1. Since the reduceByKey does a shuffle, it is in the next stage.

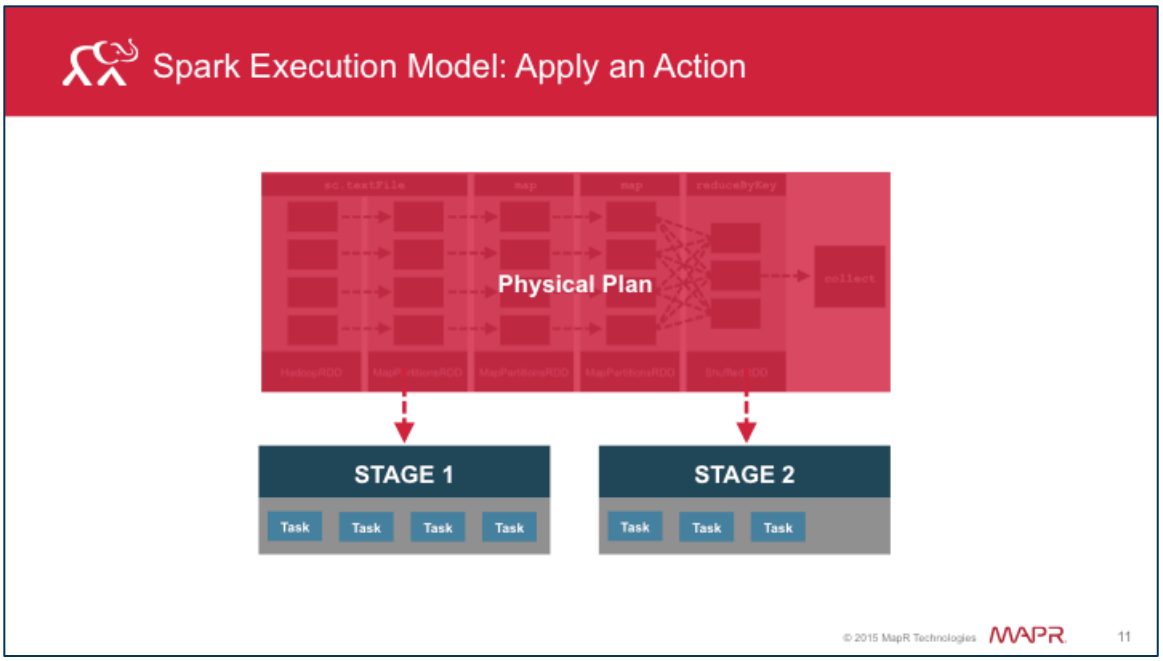
There are other reasons as to why the scheduler may truncate the lineage which are discussed next.



Notes on Slide 10:

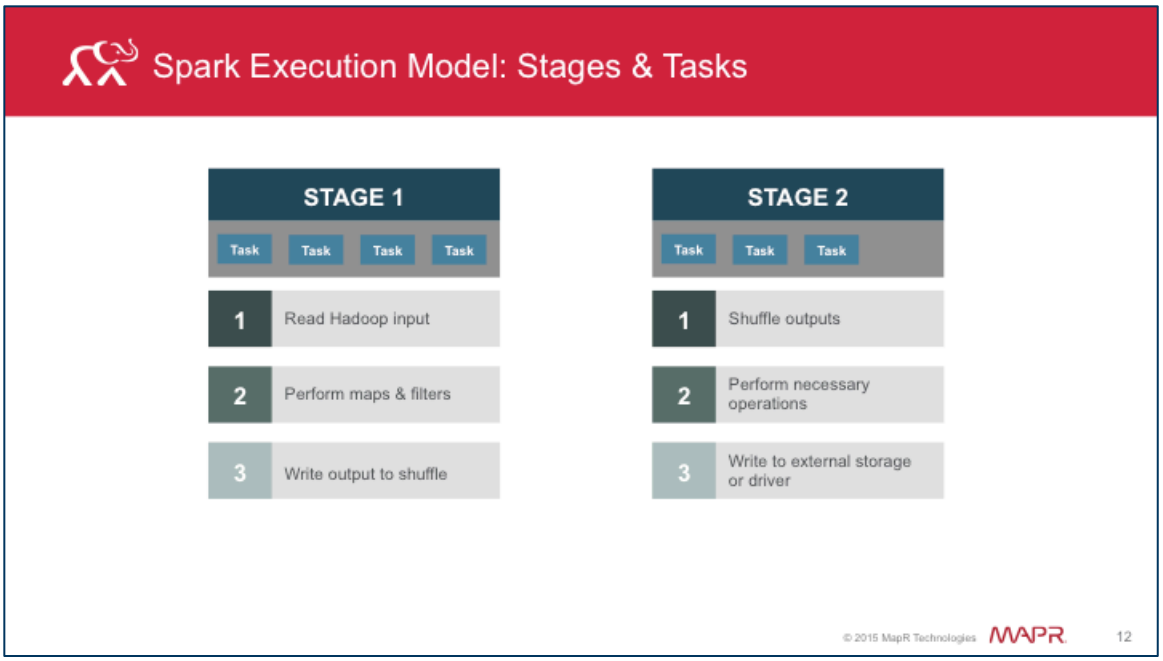
The situations in which the scheduler can truncate the lineage of an RDD graph are listed here.

1. Pipelining: When there is no movement of data from the parent RDD, the scheduler will pipeline the RDD graph collapsing multiple RDDs into a single stage.
2. When an RDD is persisted to cluster memory or disk, the Spark scheduler will truncate the lineage of the RDD graph. It will begin computations based on the persisted RDD.
3. Another situation when the Spark scheduler will truncate the lineage is if an RDD is already materialized due to an earlier shuffle, since shuffle outputs in Spark are written to disk. This optimization is built into Spark.



Notes on Slide 11:

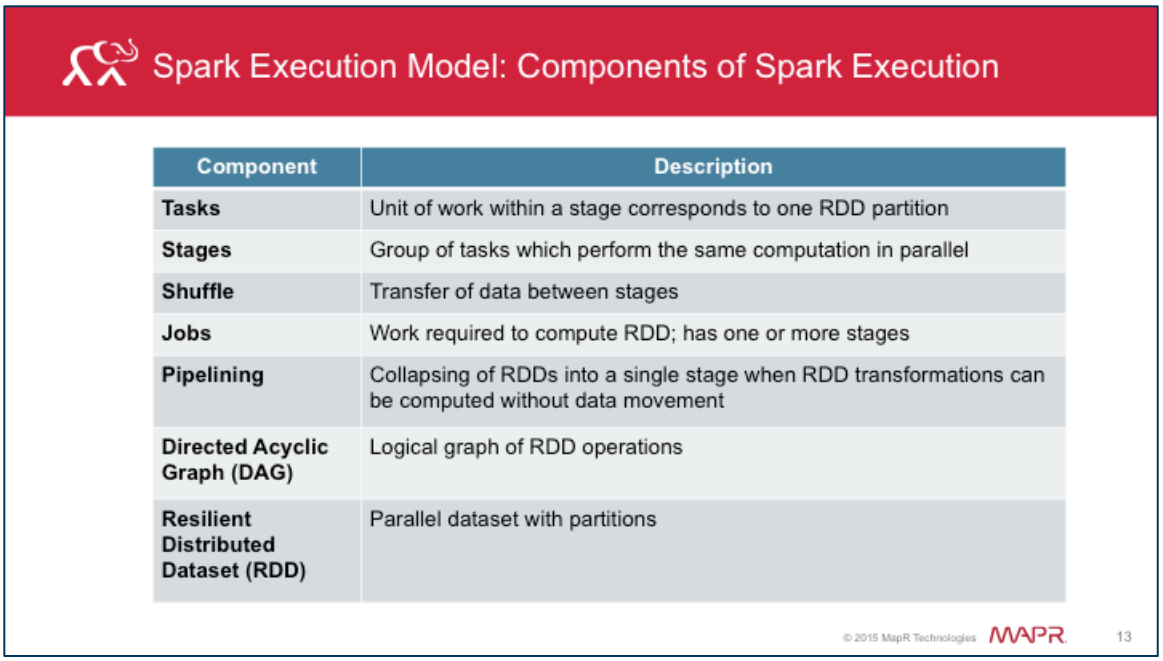
When an action is encountered, the DAG is translated into a Physical plan to compute the RDDs needed for performing the action. The Spark scheduler submits a job to compute all the necessary RDDs. Each job is made up of one or more stages and each stage is composed of tasks. Stages are processed in order and individual tasks are scheduled and executed on the cluster.



Notes on Slide 12:

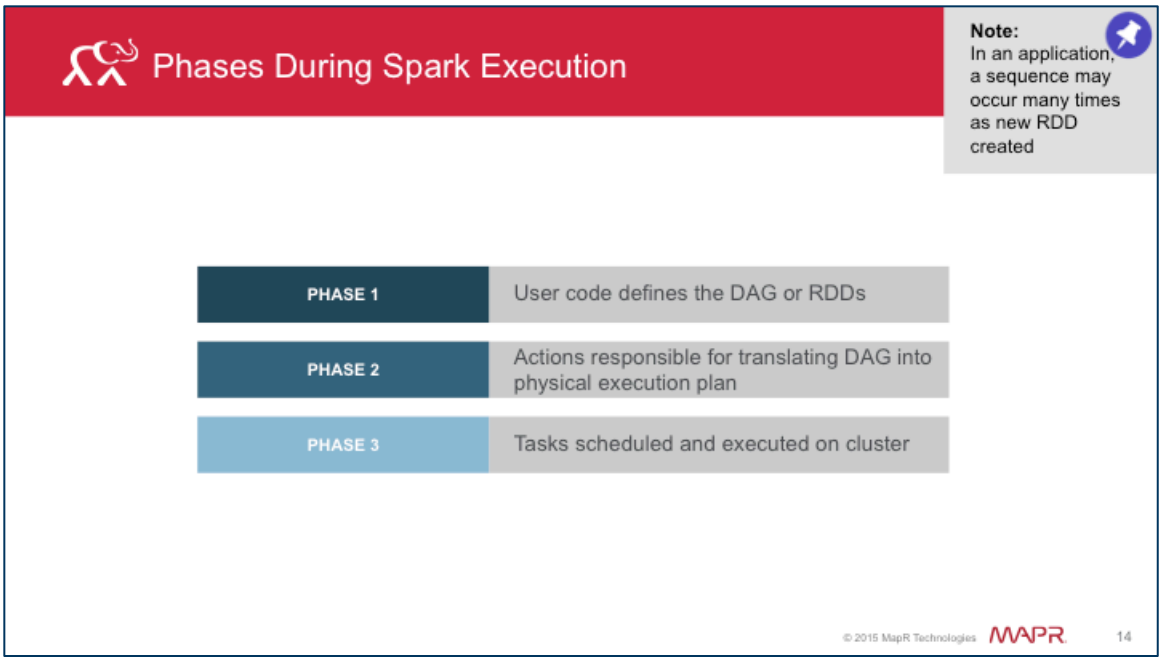
The stage has tasks for each partition in that RDD. A stage launches tasks that do the same thing but on specific partitions of data. Each task performs the same steps:

1. Fetch input (from data storage or existing RDD or shuffle outputs)
2. Perform necessary operations to compute required RDDs
3. Write output to shuffle, external storage or back to driver (for example, count, collect)



Notes on Slide 13:

- A task is a unit of work within a stage corresponding to one RDD partition.
- A stage is a group of tasks which perform the same computation in parallel.
- A shuffle is the transferring of data between stages.
- A set of stages for a particular action is a job.
- When an RDD is computed from the parent without movement of data, the scheduler will pipeline or collapse RDDs into single stage
- DAG or Directed Acyclic Graph is the logical graph of RDD operations.
- RDD or Resilient Distributed Dataset is a parallel dataset with partitions.



Notes on Slide 14:

1. User code defines the DAG or RDDs

The user code defines RDDs and operations on RDDs. When you apply transformations on RDDs, new RDDs are created that point to their parents resulting in a DAG.

2. Actions are responsible for translating the DAG into a physical execution plan

The RDD must be computed when an action is called on it. This results in computing the ancestor(s). The scheduler will submit a job per action to compute all the required RDDs. This job has one or more stages, which in turn is made up of tasks that operate in parallel on partitions. A stage corresponds to one RDD unless the lineage is truncated due to pipelining.

3. Tasks are scheduled and executed on the cluster

Stages are executed in order. The action is considered complete when the final stage in a job completes.

This sequence can occur many times when new RDDs are created.

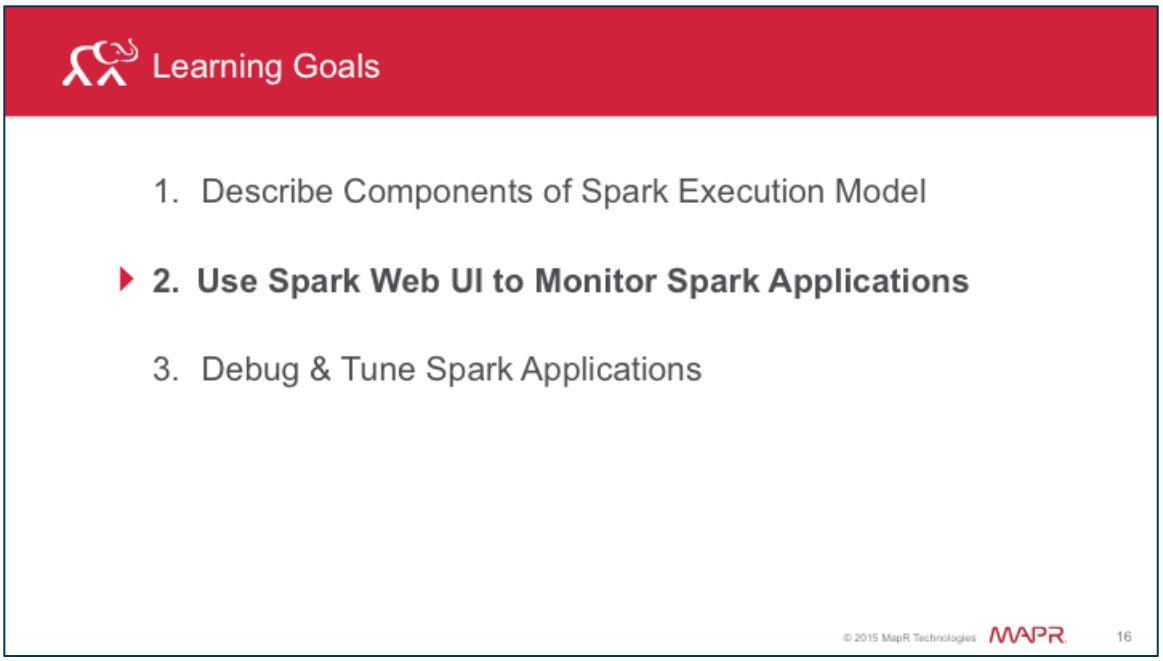
Knowledge Check

The number of stages in a job is usually equal to the number of RDDs in the DAG. However, the scheduler can truncate the lineage when:

A. There is no movement of data from the parent RDD
B. There is a shuffle.
C. The RDD is cached or persisted
D. The RDD was materialized due to an earlier shuffle

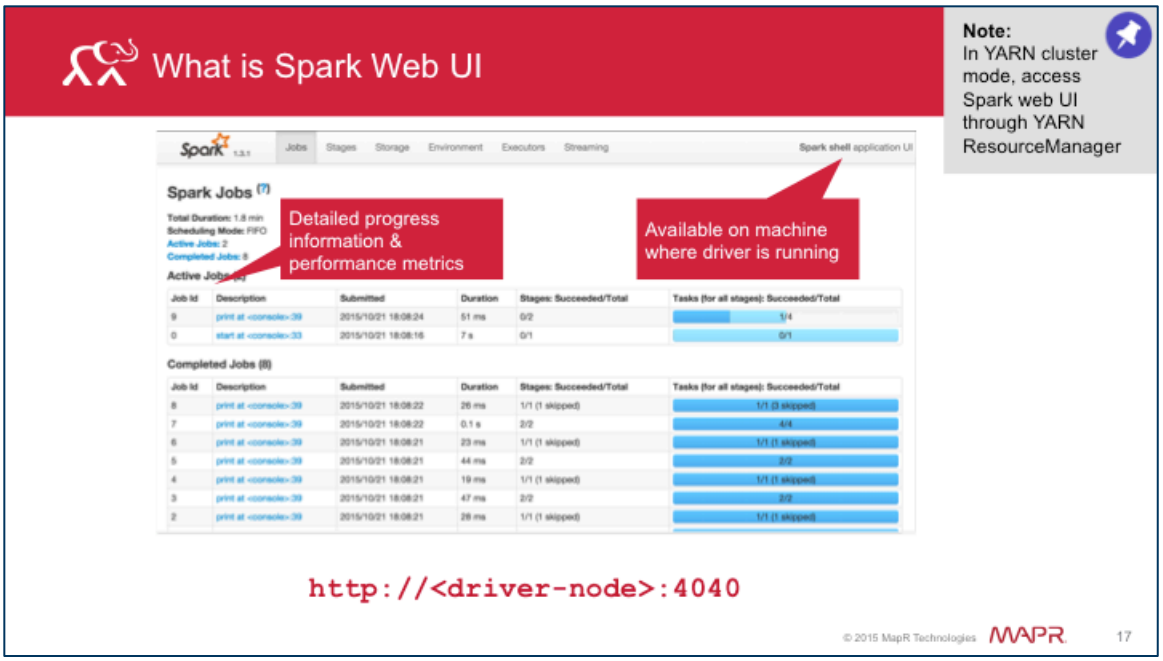
 MAPR 15

Notes on Slide 15:
Answer:A,C, D



Notes on Slide 16:

In this section, we will use the Spark Web UI to monitor Apache Spark Applications.

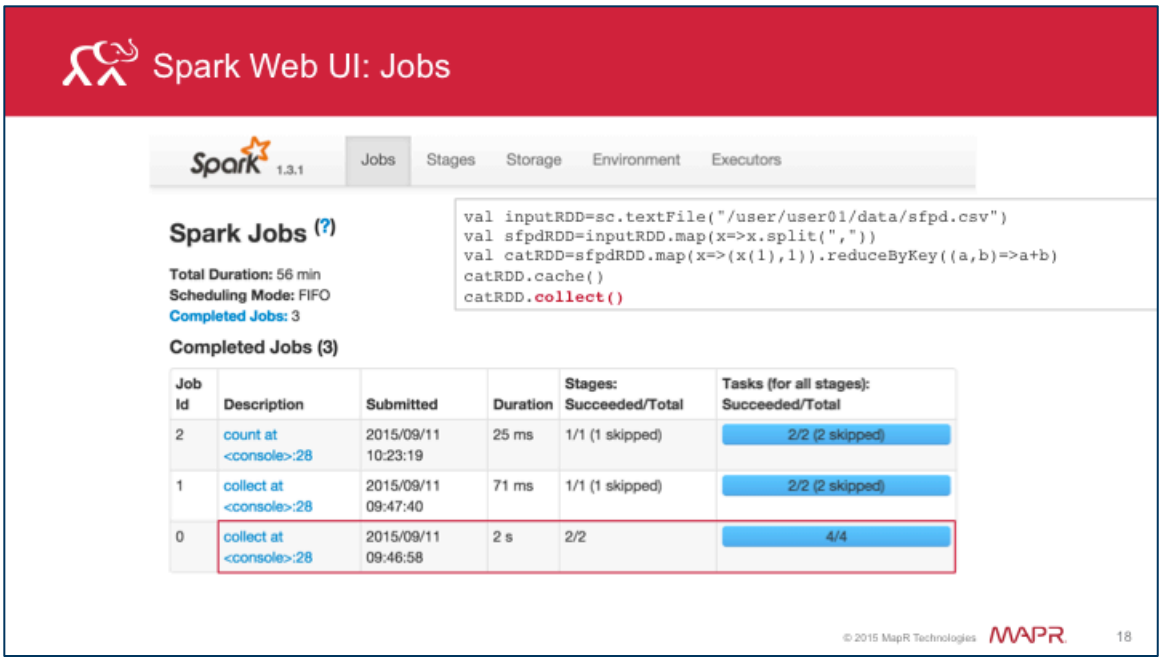


Notes on Slide 17:

The Spark Web UI provides detailed information about the progress and performance for Spark jobs. By default, this information is only available for the duration of the application. You can view the web UI after the event by setting spark.eventLog.enabled to true before starting the application.

The Spark Web UI is available on the machine where the driver is running on port 4040 by default. If multiple SparkContexts are running on the same host, they will bind to successive ports beginning with 4040, then going to 4041, 4042, and so on.

Note that in YARN cluster mode, the UI can be accessed through the YARN ResourceManager which proxies requests directly to the driver.

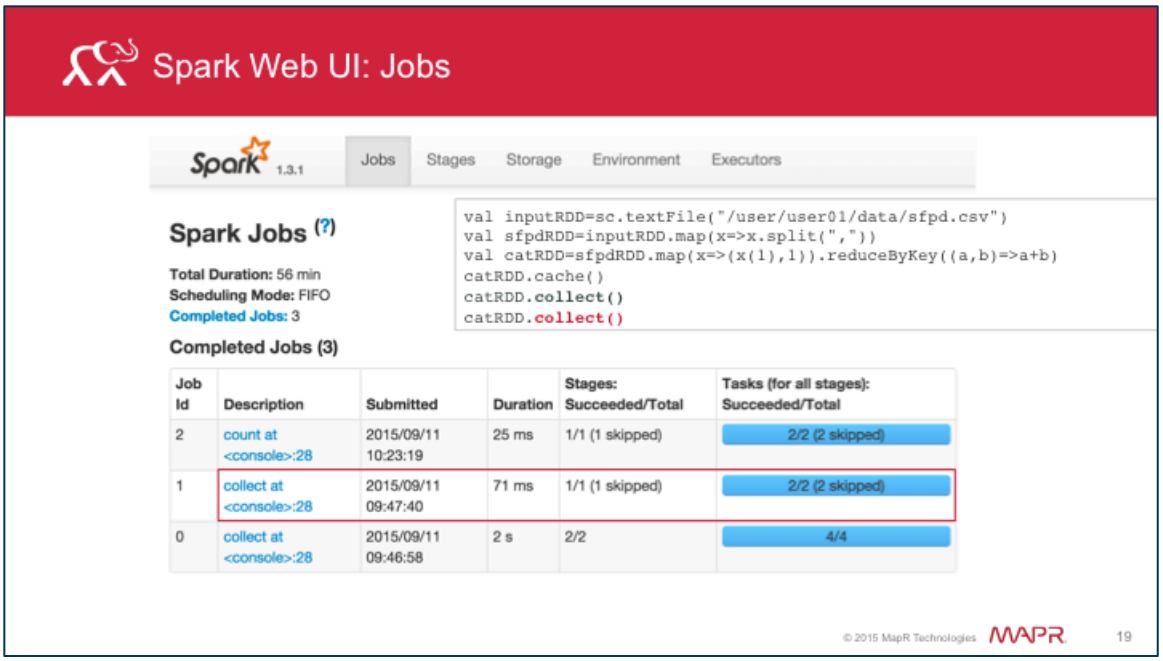


Notes on Slide 18:

To access the Spark Web UI, go to a web browser. Use the ipaddress of the driver and port 4040. The Jobs page gives you detailed execution information for active and recently completed Spark jobs. It gives you the performance of a job, and also the progress of running jobs, stages and tasks.

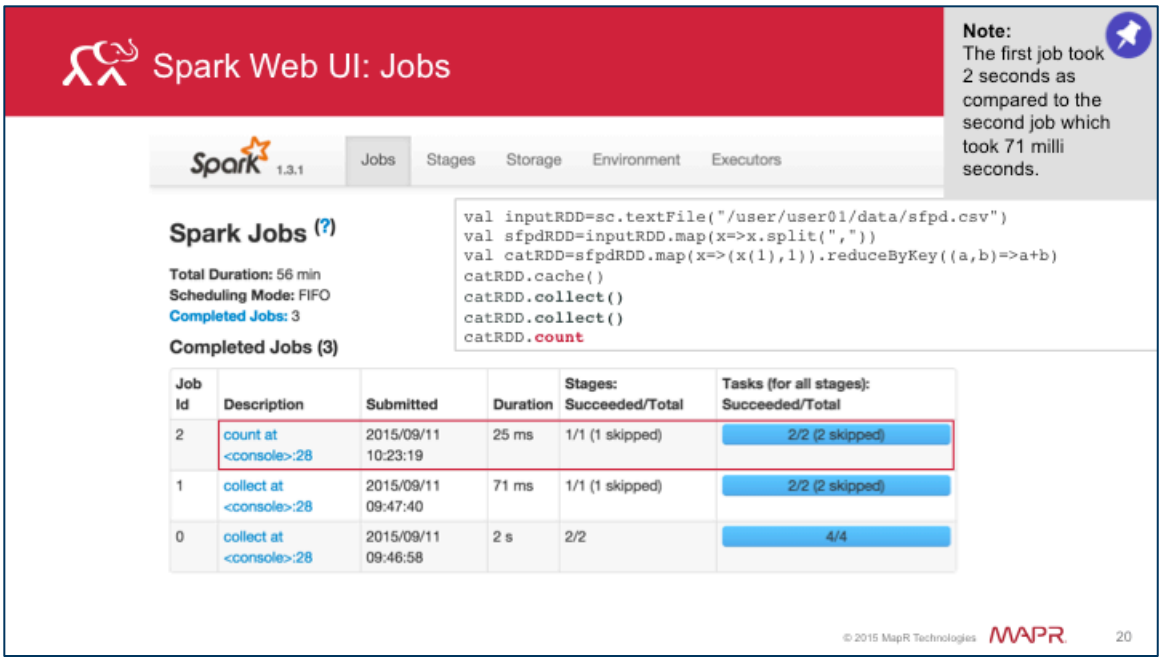
In this example you see the jobs and stages based on the code sample shown.

- job 0 is the first job that was executed and corresponds to the first collect() action. It consists of 2 stages and each stage consists of 4 tasks.



Notes on Slide 19:

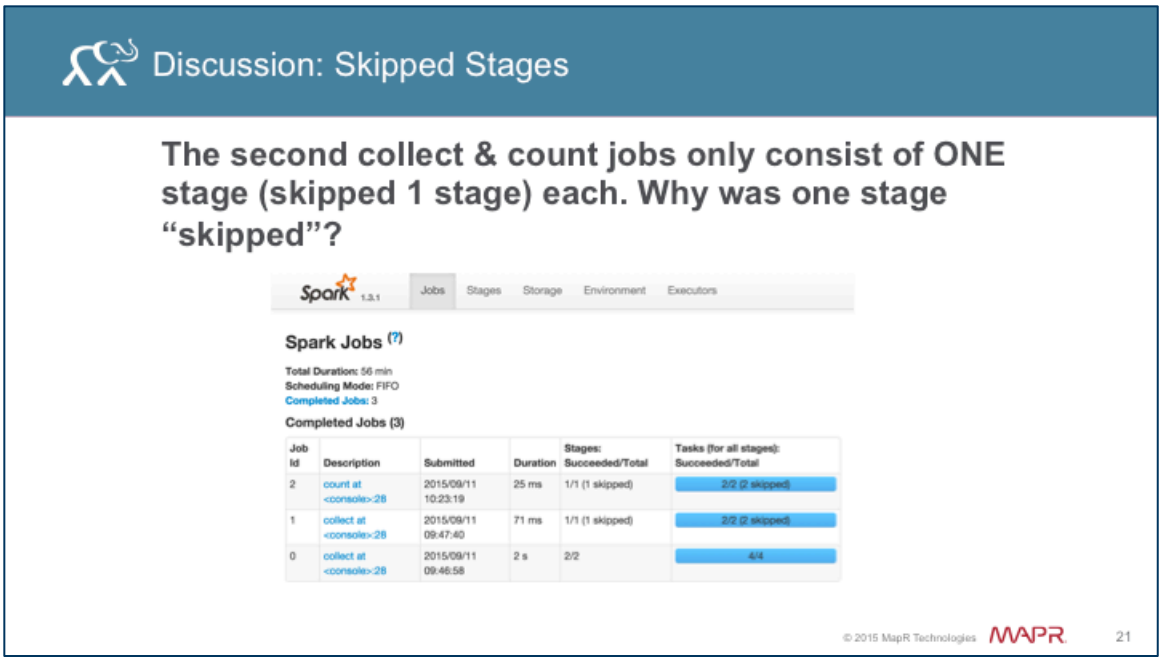
- job 1 corresponds to the second collect() action. It consists of 1 stage which is made up of two tasks.



Notes on Slide 20:

- job 2 corresponds to the count() action and is also consists of 1 stage containing two tasks.

Note that the first job took 2 seconds as compared to the second job which took 71 milli seconds.

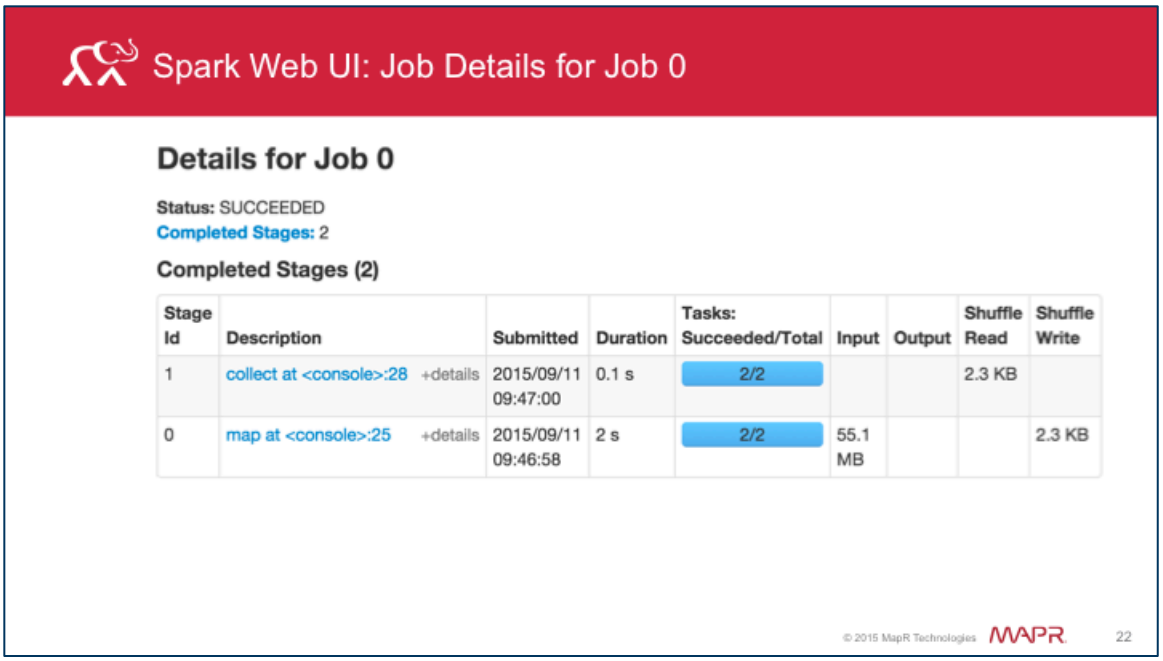


Notes on Slide 21:

Answer:

The first collect computes all RDDs and then caches catRDD and therefore has two stages.

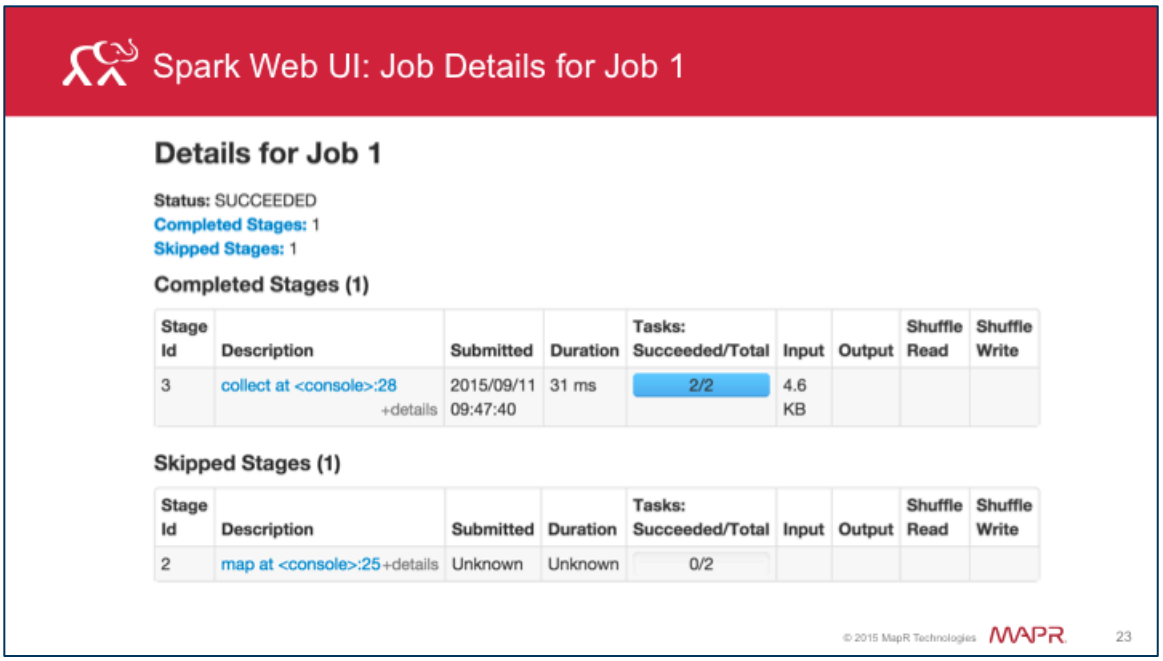
The second **`collect()`** & the **`count()`** use the cached RDD and the scheduler truncates the lineage resulting in a skipped stage. This also results in job 1 (71 ms) being faster than job 0 (2 s).



Notes on Slide 22:

Clicking the link in the Description column on the Jobs page takes you to the Job Details page. This page also gives you the progress of running jobs, stages and tasks.

Note that the collect job here takes 1 s.



Notes on Slide 23:

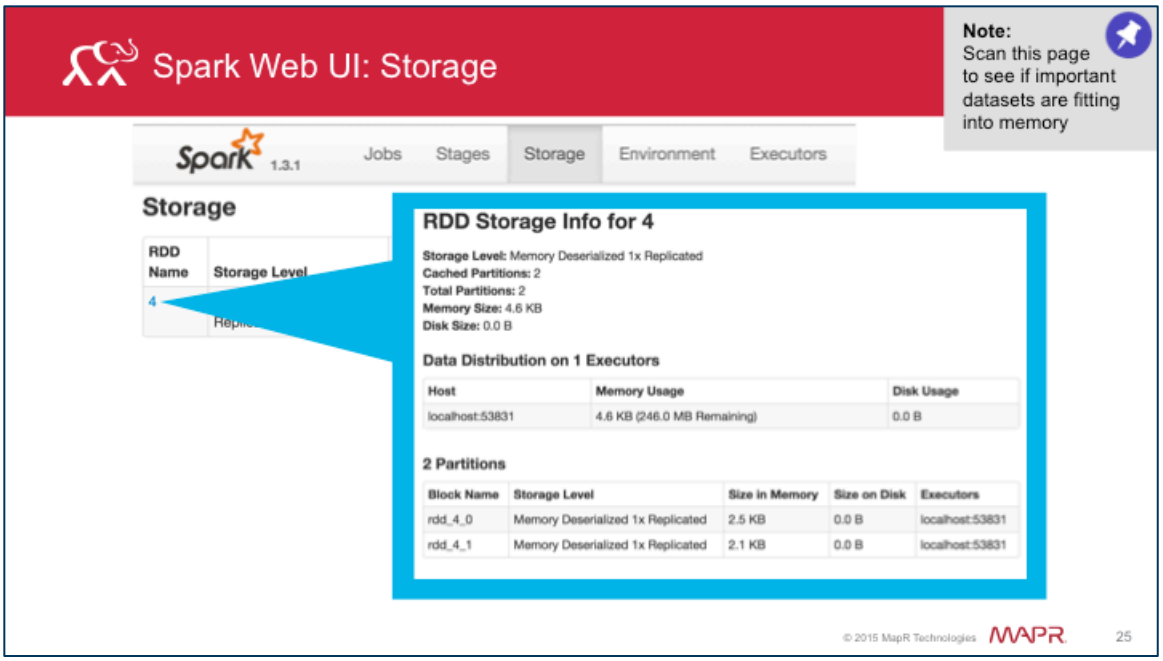
This page provides the details for Job 1, which had the skipped stage. In this page you can see the details for the completed stage and the skipped stage.

Note that the collect here took only 31 ms.

Notes on Slide 24:

Once you have identified the stage in which you are interested, you can click the link to drill down to the Stage details page.

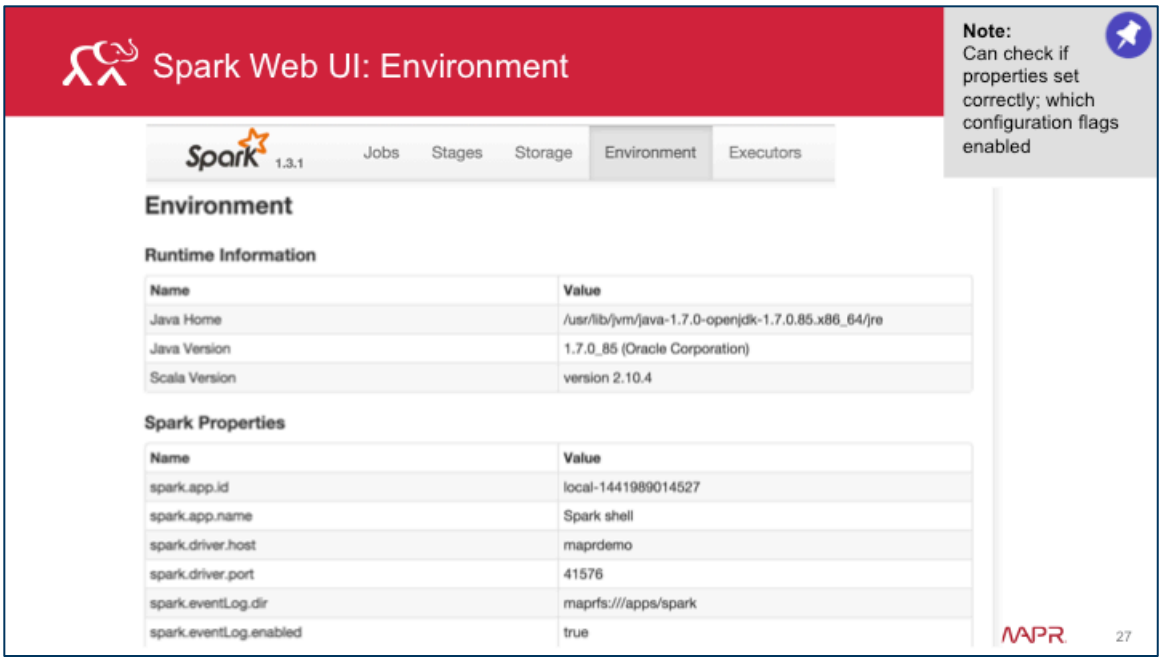
Here we have summary metrics and aggregated metrics for completed tasks and metrics on all tasks.



Notes on Slide 25:

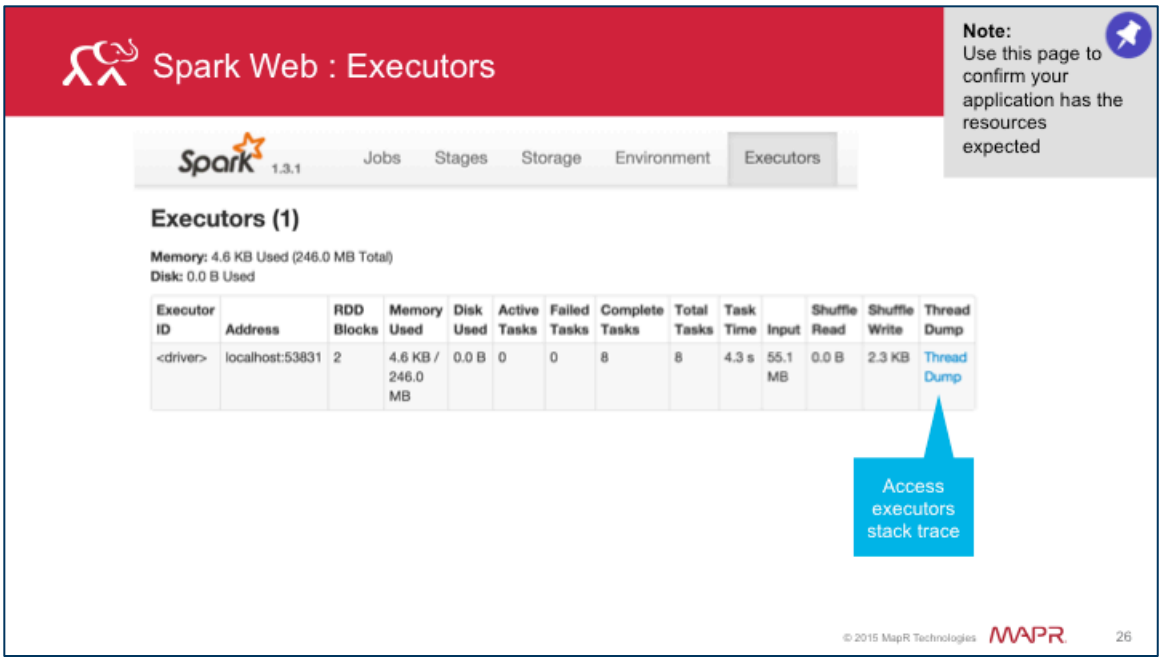
The Storage page provides information about persisted RDDs. The RDD is persisted if you called persist() or cache() on the RDD followed by an action to compute on that RDD. This page tells you which fraction of the RDD is cached and the quantity of data cached in various storage media.

Scan this page to see if important datasets are fitting into memory.



Notes on Slide 27:

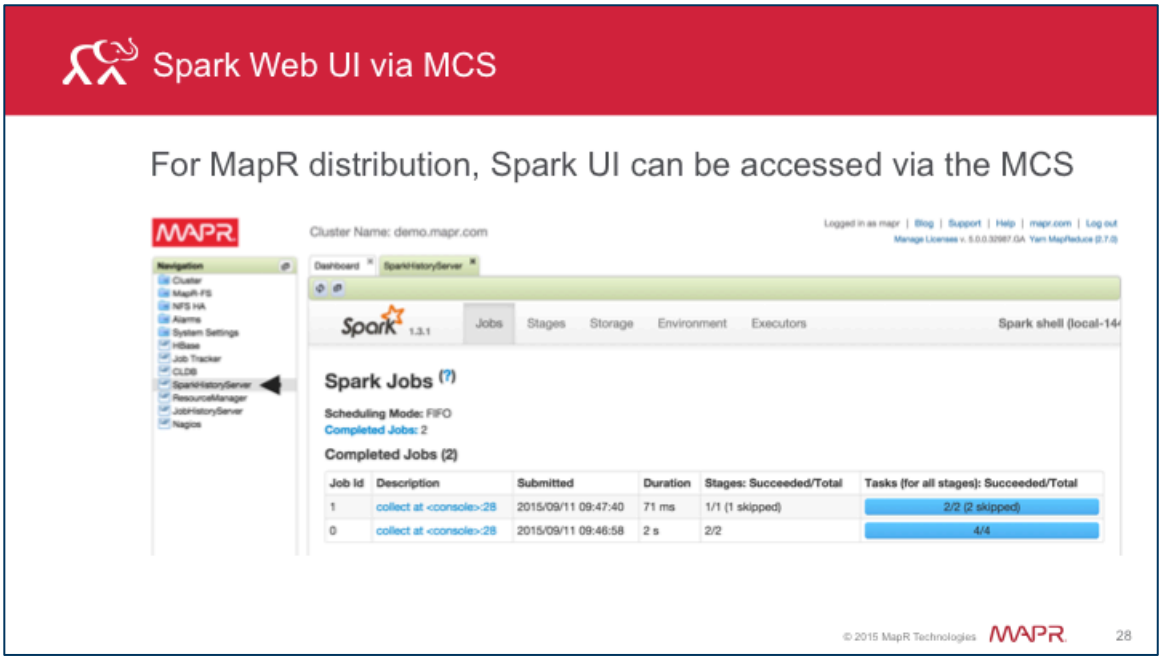
The Environments page lists all the active properties of your Spark application environment.
Use this page when you want to see which configuration flags are enabled.
Note that only values specified through spark-defaults.conf, SparkConf, or the command line will be displayed here. For all other configuration properties, the default value is used.



Notes on Slide 26:

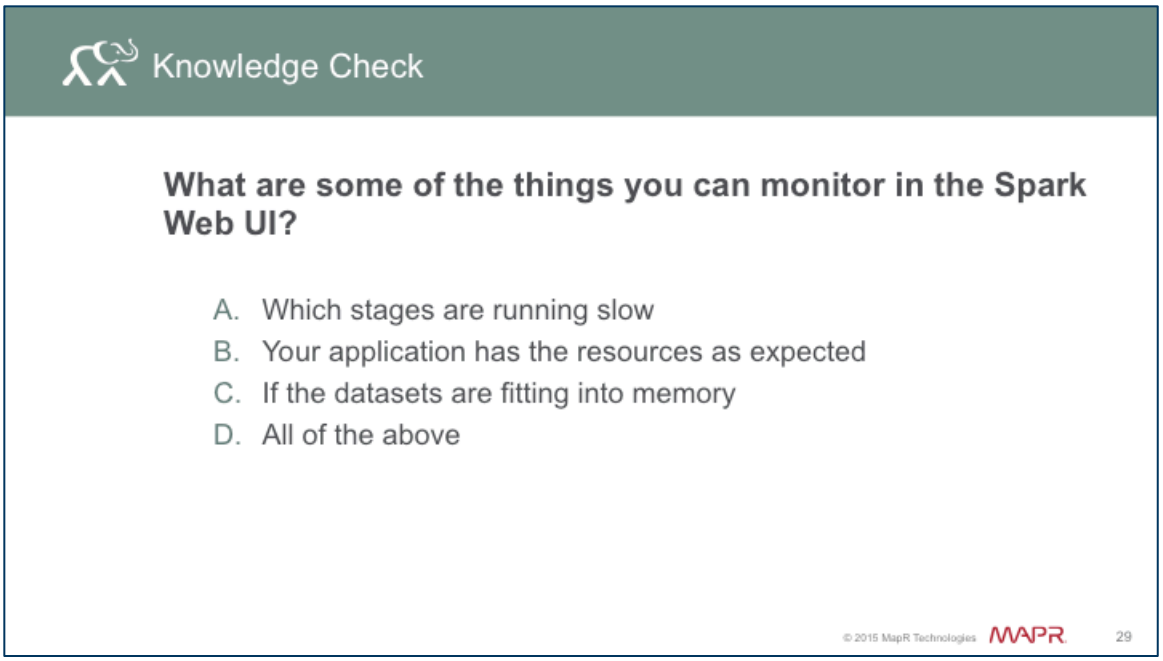
The Executors page lists the active executors in the application. It also includes some metrics about the processing and storage on each executor.

Use this page to confirm that your application has the amount of resources that you were expecting.



Notes on Slide 28:

If you have a MapR distribution, you can access the Spark Web UI through the MapR Control System (MCS). Once you log into the MCS, you will see the Spark History Server in the left navigation pane as shown here.



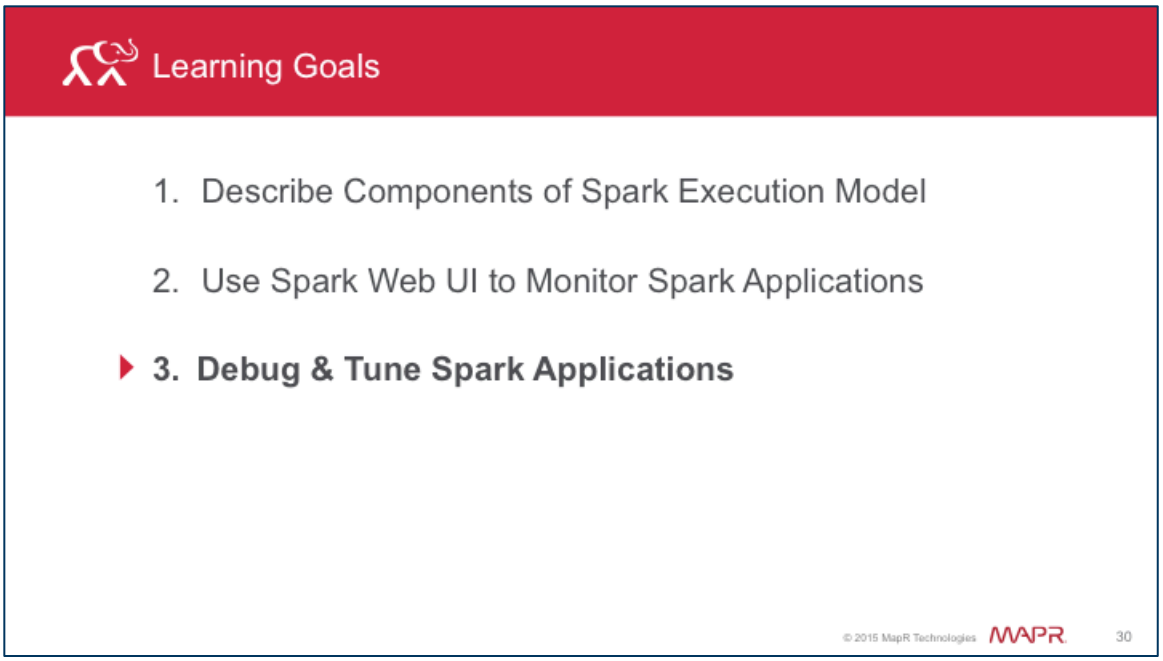
Notes on Slide 29:

Answer: D

You can use the Job details page and the stages tab to see which stages are running slow; to compare the metrics for a stage; look at each task.

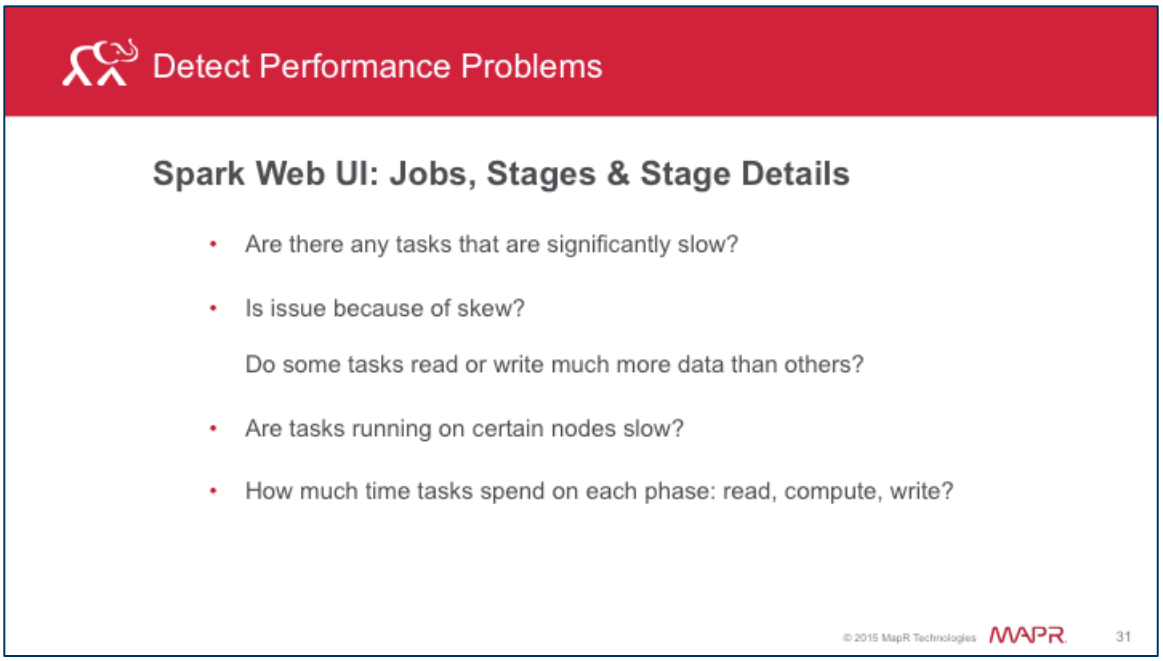
Look at the Executors tab to see if your application has the resources as expected

Use the Storage tab to see if the datasets are fitting into memory and what fraction is cached.



Notes on Slide 30:

We are now going to see how to debug and tune our Spark applications.



Notes on Slide 31:

Here are some ways to debug performance issues.

To detect shuffle problems, look at the Spark Web UI for any tasks that are significantly slow.

A common source of performance problems in data-parallel systems that occurs when some small tasks take much longer than others is called skew. To see if skew is the problem, look at the Stages Details page and see if there are some tasks that are running significantly slower than others. Drill down to see if there is a small number of tasks that read or write more data than others.

From the Stages Details page, you can also determine if tasks are running slow on certain nodes.

From the Spark Web UI, you can also find those tasks that are spending too much time on reading, computing and/or writing.

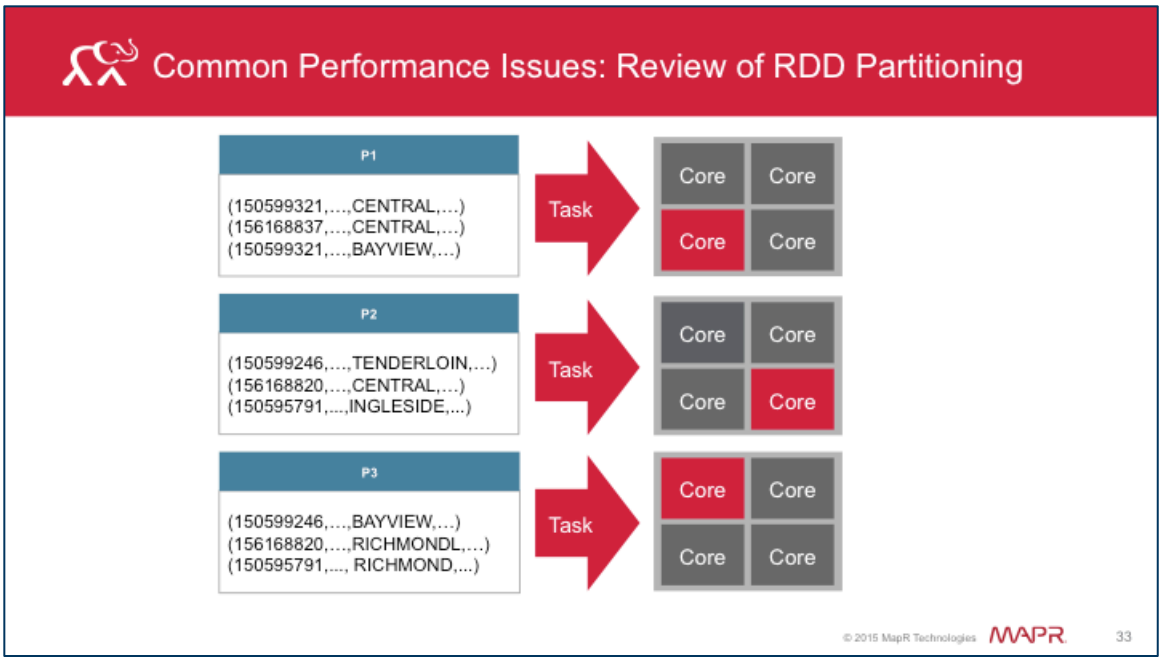
In this case, look at the code to see if if you any expensive operations.

Notes on Slide 32:

Common issues leading to slow performance are:

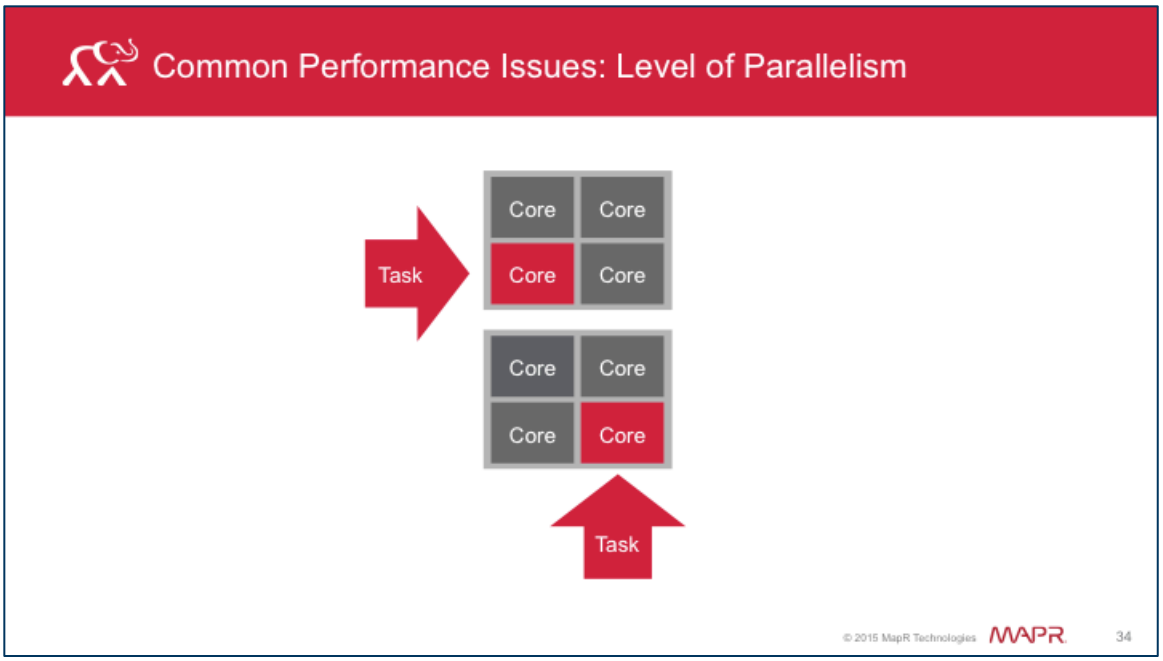
- The level of parallelism
- The serialization format used during shuffle operations
- Managing memory to optimize your application

We will look at each one of these in more depth.



Notes on Slide 33:

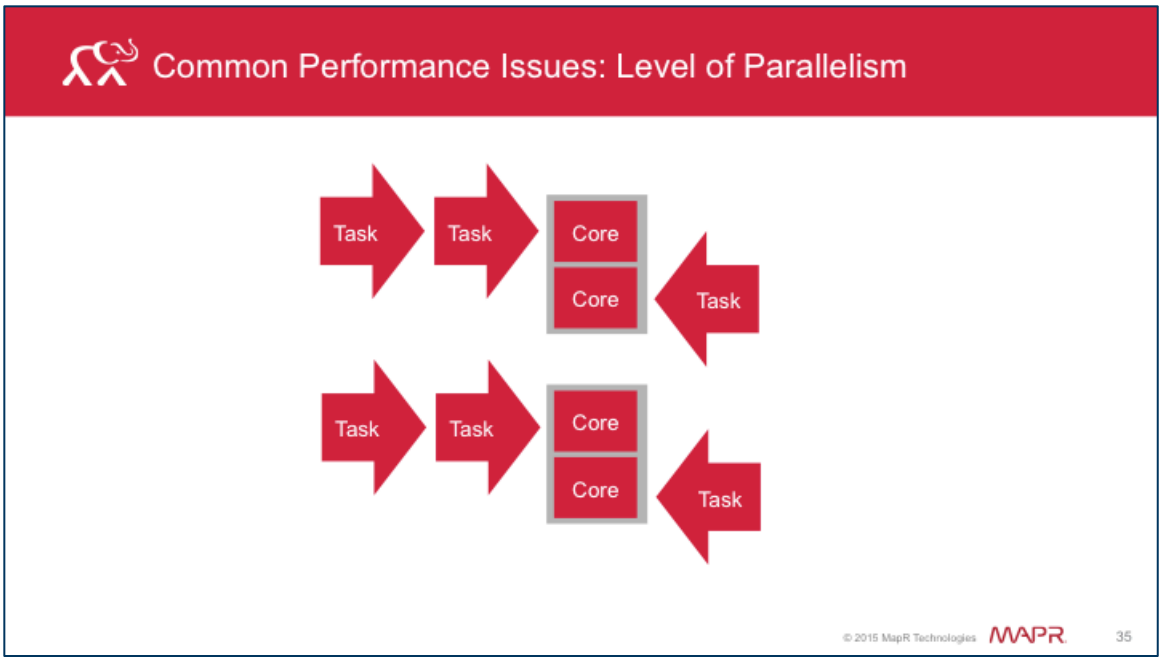
An RDD is divided into a set of partitions where each partition contains a subset of the data. The scheduler will create a task for each partition. Each task requires a single core in the cluster. Spark by default will partition based on what it considers to be the best degree of parallelism.



Notes on Slide 34:

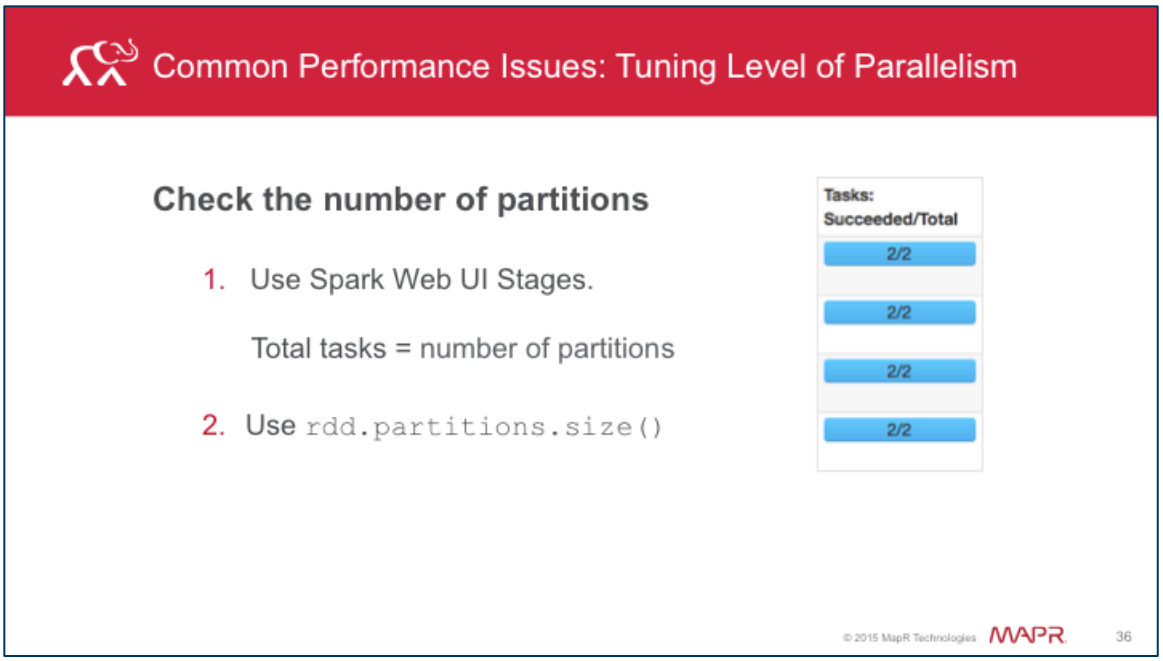
How does the level of parallelism affect performance?

If there is too little parallelism – Spark might leave resources idle.



Notes on Slide 35:

If there is too much parallelism, overheads associated with each partition add up to become significant.

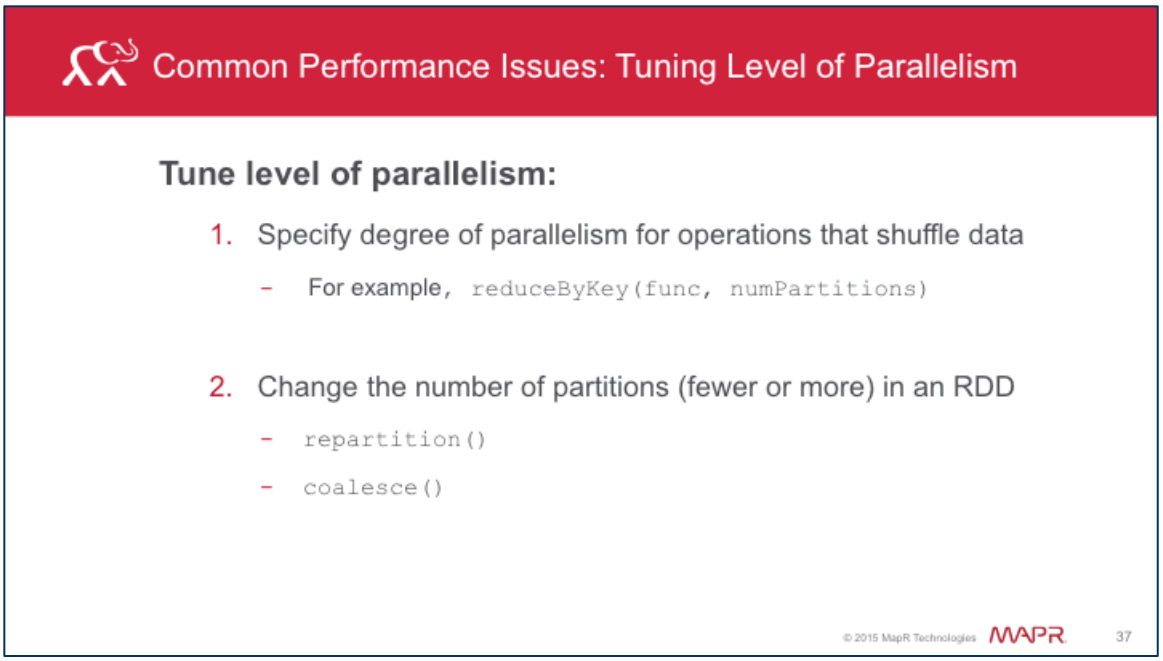


Notes on Slide 36:

How do you find the number of partitions?

You can do this through the Spark Web UI in the stages tab. Since a task in a stage maps to a single partition in the RDD, the total number of tasks will give you the number of partitions.

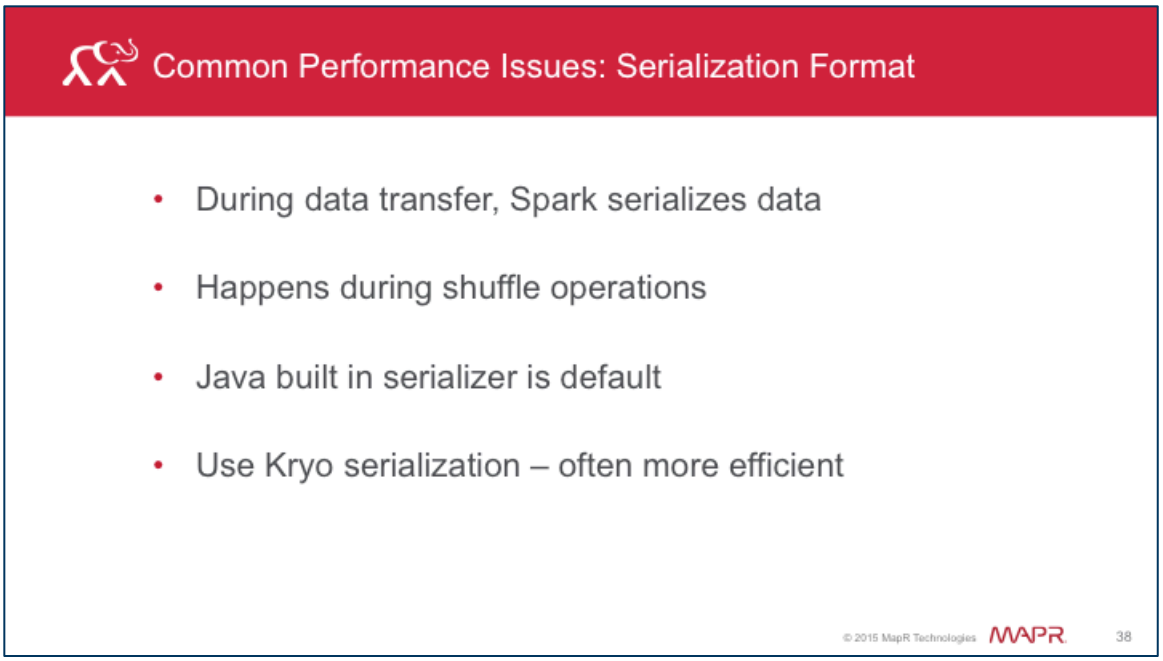
You can also use `rdd.partitions.size()` to get the number of partitions in the RDD.



Notes on Slide 37:

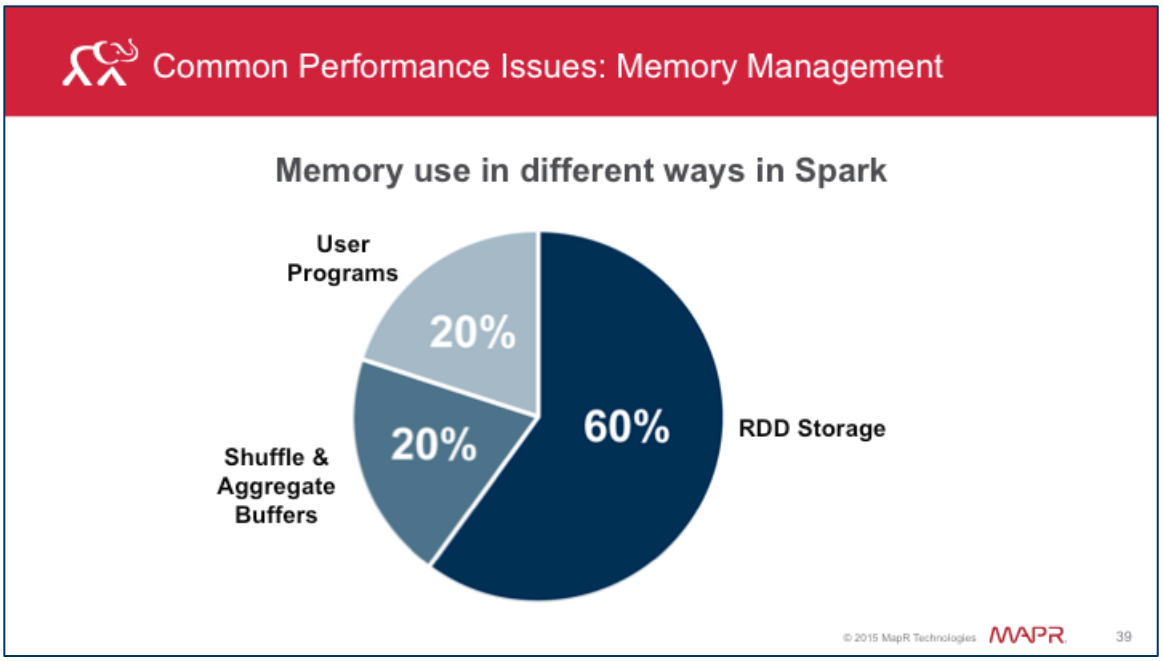
To tune the level of parallelism:

- Specify the number of partitions, when you call operations that shuffle data, for example, reduceByKey.
- Redistribute the data in the RDD. This can be done by increasing or decreasing the number of partitions. You can use the repartition() method to specify the number of partitions or coalesce() to decrease the number of partitions.



Notes on Slide 38:

When a large amount of data is transferred over the network during shuffle operations, Spark serializes objects into binary format. This can sometimes cause a bottleneck. By default Spark uses the Java built-in serializer. However, it is often more efficient to use Kryo serialization.

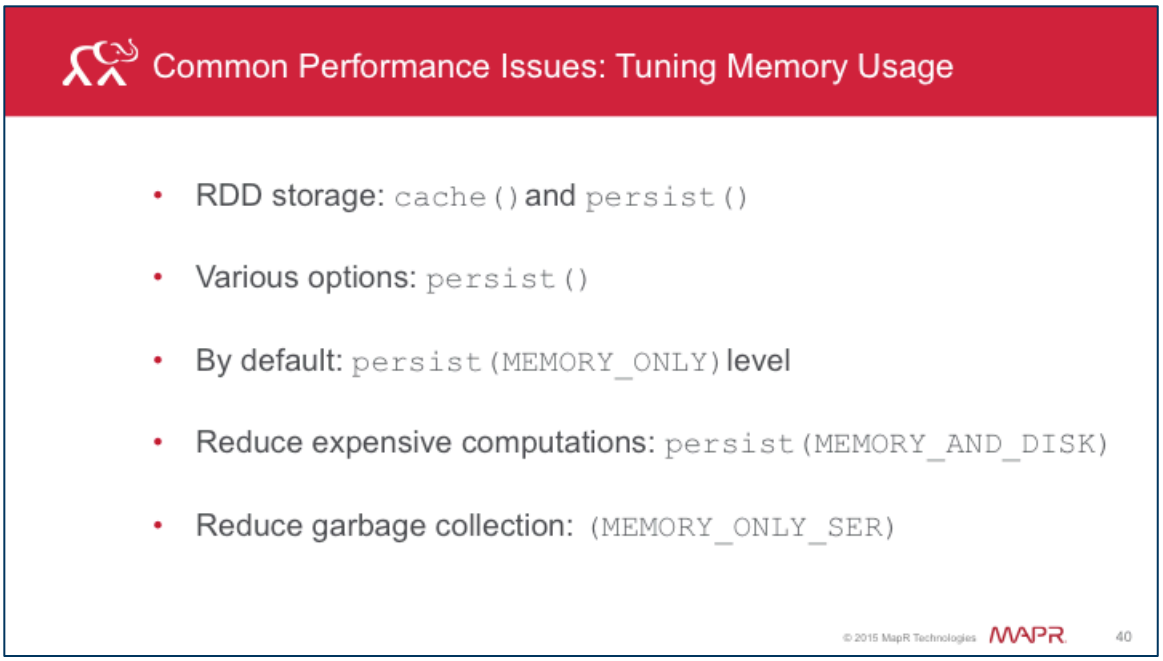


Notes on Slide 39:

Memory can be used in different ways in Spark. Tuning Spark's use of memory can help optimize your application.

By default, Spark will use:

- 60% of space for RDD storage
- 20% for shuffle
- 20% for user programs



Notes on Slide 40:

You can tune the memory usage by adjusting the memory regions used for RDD storage, shuffle and user programs.

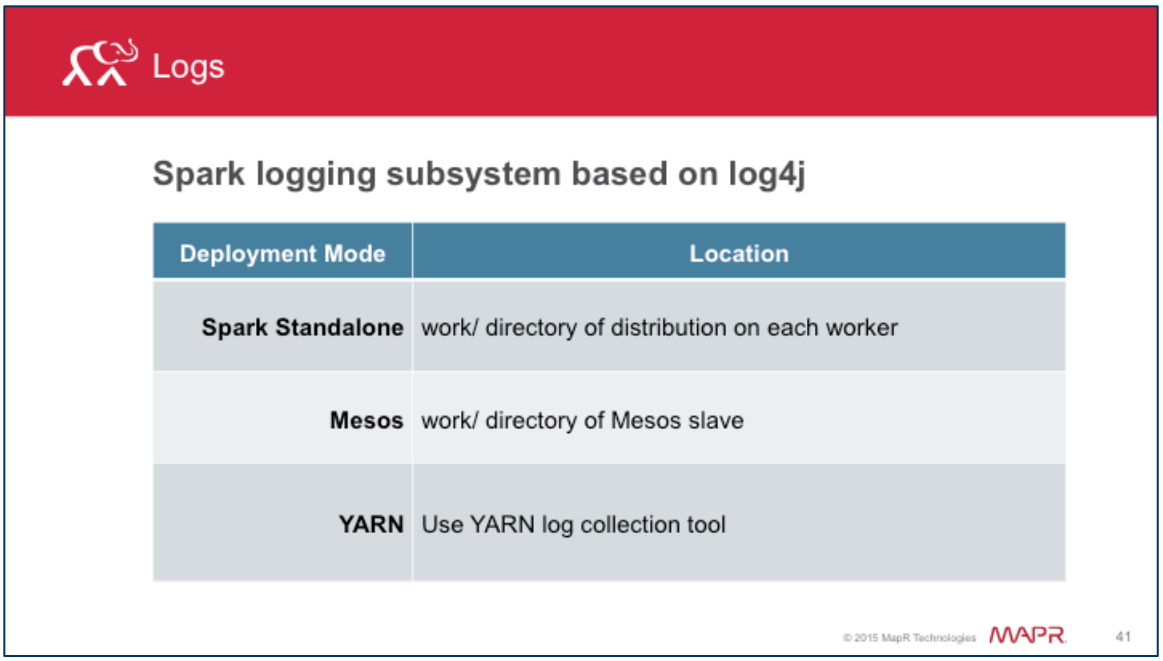
Use cache() or persist() on RDDs. Using cache() on an RDD will store the RDD partitions in memory buffers.

There are various options for persist(). Refer to Apache Spark documentation on persist options for RDD.

By default, persist() is the same as cache() or persist(MEMORY_ONLY). If there is not enough space to cache new RDD partitions, old ones are deleted and recomputed when needed.

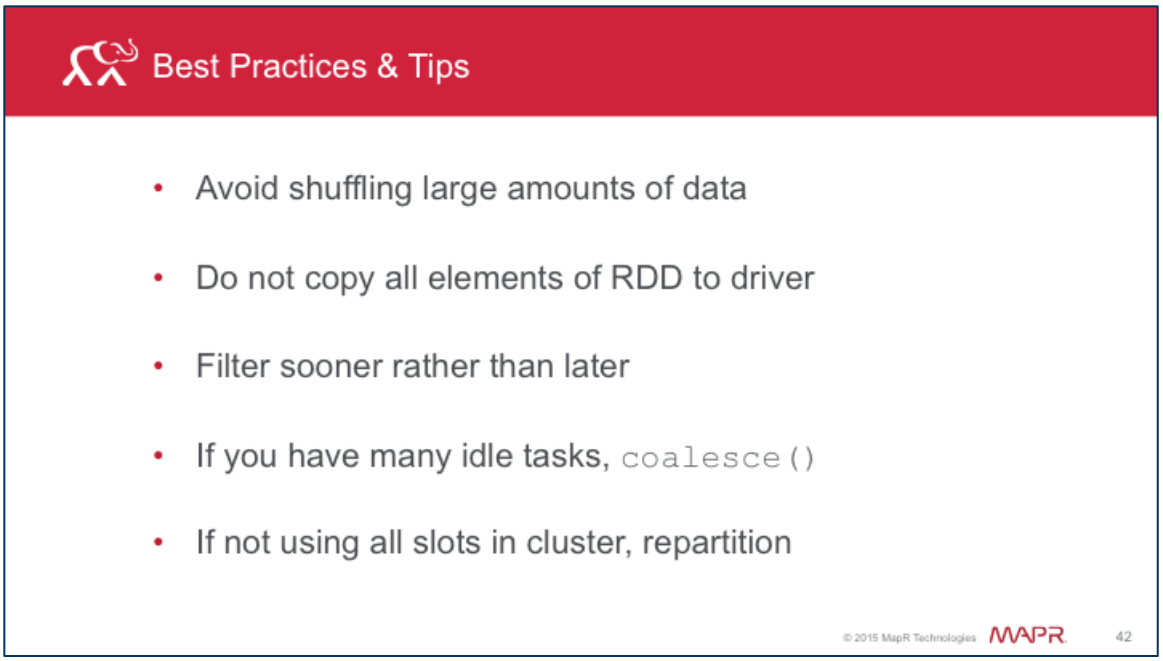
It is better to use persist(MEMORY_AND_DISK). This will store the data on disk and load it into memory when needed. This cuts down expensive computations.

Using MEMORY_ONLY_SER will cut down on garbage collection. Caching serialized objects may be slower than caching raw objects. However, it does decrease the time spent on garbage collection.



Notes on Slide 41:

- The Spark logging subsystem is based on log4j. The logging level or log output can be customized. An example of the log4j configuration properties is provided in the Spark conf directory which can be copied and suitably edited.
- The location of the Spark log files depends on the deployment mode.
- In the Spark Standalone mode, the log files are located in the work/ directory of the Spark distribution on each worker.
- In Mesos, the log files are in work/ directory of the Mesos slave and is accessible from the Mesos master UI.
- To access the logs in YARN, use the YARN log collection tool.

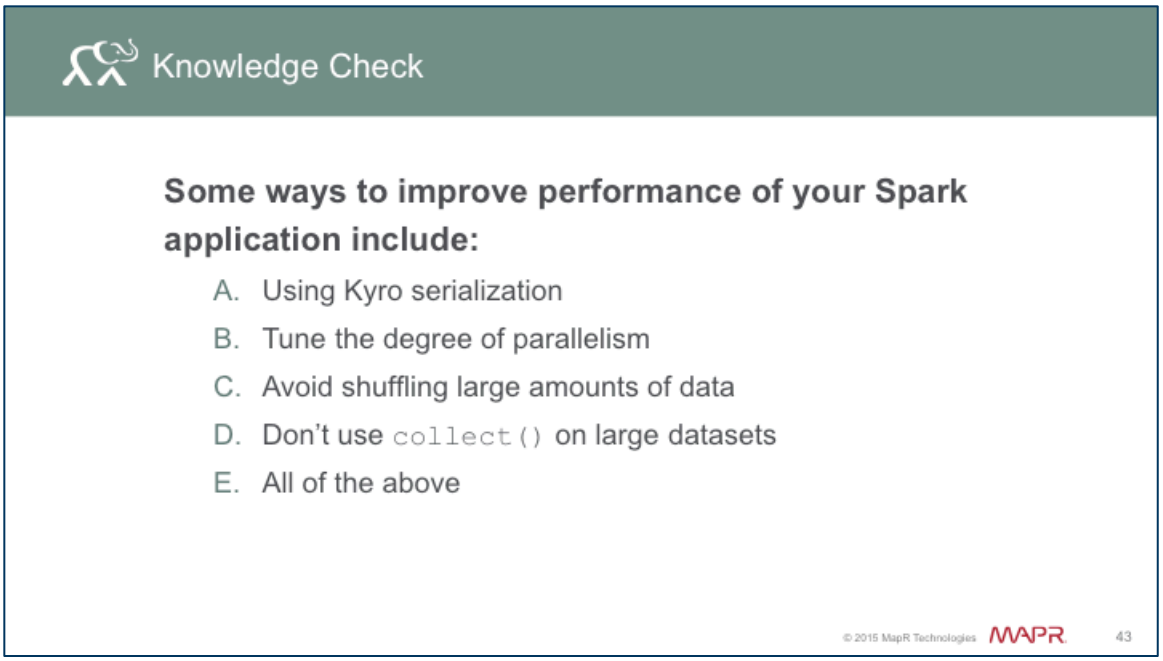


Notes on Slide 42:

- If possible avoid having to shuffle large amounts of data. Use aggregateByKey when possible for aggregations.

groupByKey on large dataset results in shuffling large amounts of data. If possible use reduceByKey. You can also use combineByKey or foldByKey.

- collect() action tries to copy every single element in the RDD to the single driver program. If you have a very large RDD, this can result in the driver crashing. The same problem occurs with countByKey, countByValue and collectAsMap.
- Filter out as much as you can to have smaller datasets
- If you have many idle tasks (~ 10k) , then coalesce
- If you are not using all the slots in the cluster, then repartition



Notes on Slide 43:
Answer: E

Notes on Slide 44:

In this Lab, you will create RDDs, apply transformations and actions; print the DAG to console; use the Spark UI to monitor the Spark execution components.

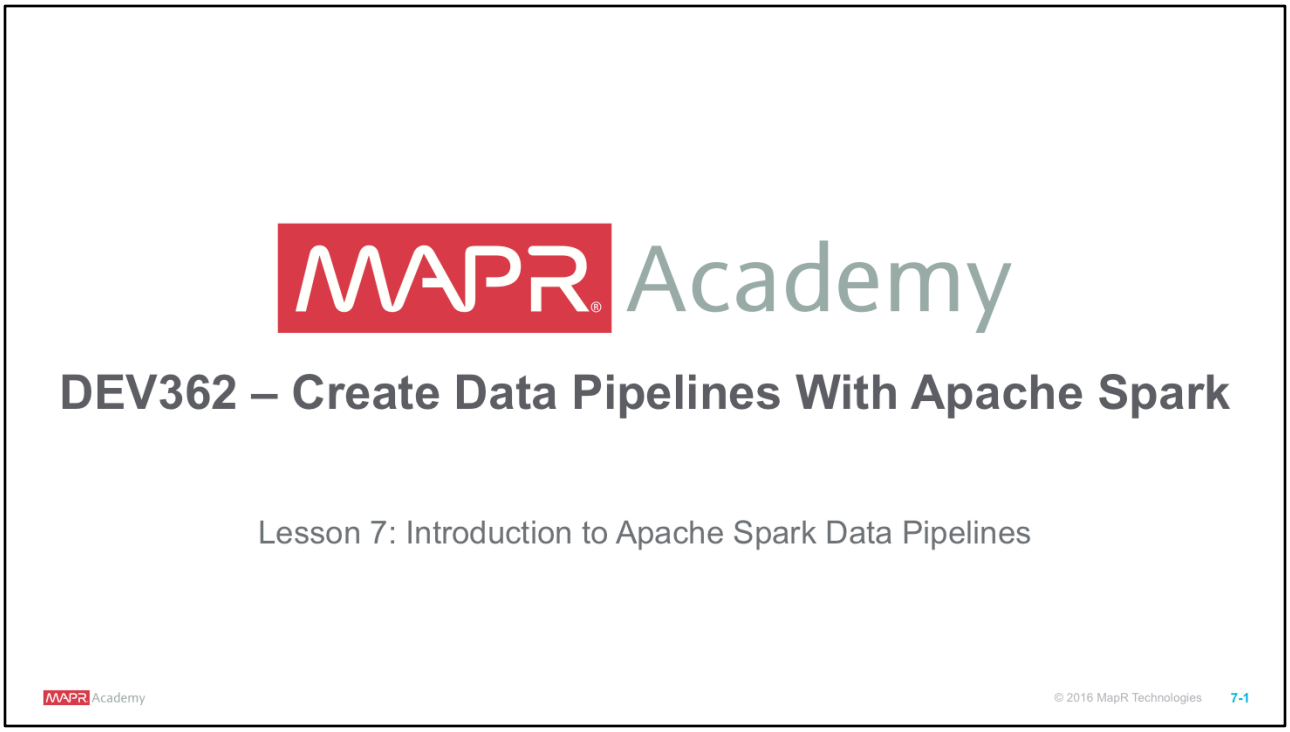
Notes on Slide 45:

Congratulations! You have completed Lesson 6 and this course – DEV 361 – Build and Monitor Apache Spark Applications. Go to doc.mapr.com for more information on Hadoop, Spark and other eco system components. To learn more about other courses, visit the MapR academy.

DEV 362 - Apache Spark Essentials

Slide Guide

Version 5.1 – Summer 2016



Welcome to DEV 362, Lesson 7, Introduction to Apache Spark Data Pipelines.

Learning Goals