



Developing Spark Applications

Slide Guide

Version 5.1 – Summer 2016

For use with the following courses:

- DEV 3600 – Developing Spark Applications
- DEV 360 – Apache Spark Essentials
- DEV 361 – Build and Monitor Apache Spark Applications
- DEV 362 – Create Data Pipeline Applications Using Apache Spark

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2016, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.





DEV 360 - Apache Spark Essentials

Slide Guide

Version 5.1 – Summer 2016

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2016, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.



 MAPR Academy

Apache Spark Essentials

Getting Started with Apache Spark Essentials

© 2015 MapR Technologies  MAPR

1

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





Learning Goals

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

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

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





Prerequisites

Required

- Basic to intermediate Linux knowledge, including:
 - The ability to use a text editor, such as vi
 - Familiarity with basic command-line options such as mv, cp, ssh, grep, cd, useradd
- Knowledge of application development principles
- A Linux, Windows or MacOS computer with the MapR Sandbox installed (On-demand course)
- Connection to a Hadoop cluster via SSH and web browser (for the ILT and vILT course)

Recommended

- Knowledge of functional programming
- Knowledge of Scala or Python
- Beginner fluency with SQL
- HDE 100 - Hadoop Essentials

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



 Course Materials

- DEV360 SlideGuide.pdf
- DEV360 LabGuide.pdf
- DEV360 LabFiles.zip
- DEV360DATA.zip
- Connect to MapR Sandbox or AWS.pdf

The following course materials are provided:

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





Next Steps



Lesson 1

Introduction to Apache
Spark

© 2015 MapR Technologies  MAPR

5

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



 MAPR Academy

Apache Spark Essentials

Lesson 1: Introduction to Apache Spark

© 2015 MapR Technologies  MAPR®

1

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



 Learning Goals

- ▶ Describe Features of Apache Spark
 - How Spark fits in Big Data ecosystem
 - Why Spark & Hadoop fit together
- ▶ Define Spark Components

© 2015 MapR Technologies  MAPR®

2

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

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

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



2



What is Apache Spark?



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

© 2015 MapR Technologies  MAPR®

3

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



 Why Apache Spark?A large, bold number '1' is centered within a white circle, which is set against a light blue square background.**Fast**

- 10x faster on disk
- 100x in memory

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



 Why Apache Spark?

Ease of Development

- Write programs quickly
- More operators
- Interactive Shell
- Less code

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



 Why Apache Spark?

Deployment Flexibility

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

© 2015 MapR Technologies  MAPR®

6

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



 Why Apache Spark?A large, bold, dark blue number '4' is centered within a white circle, which is itself centered within a larger light blue square.

Unified Stack

Builds applications combining different processing models

- Batch
- Streaming
- Interactive Analytics

© 2015 MapR Technologies  MAPR®

7

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

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



 Why Apache Spark?**Multi-language support**

- Scala
- Python
- Java
- SparkR

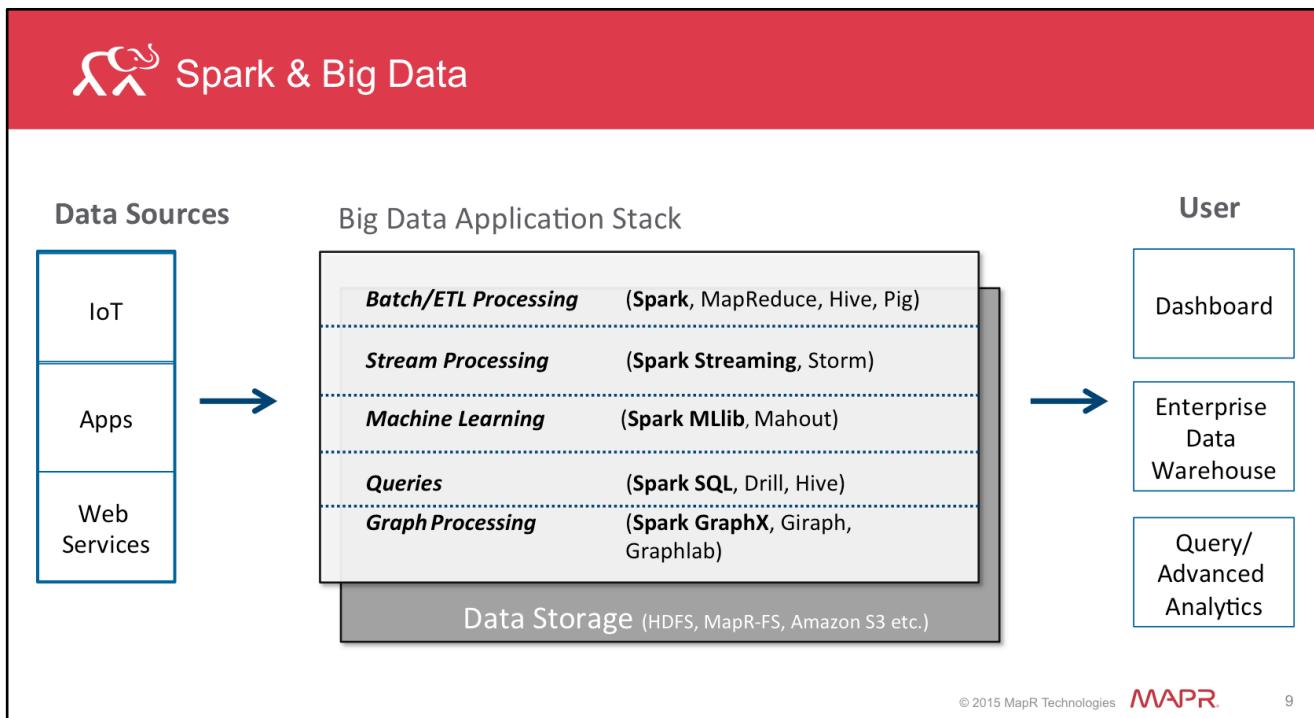
© 2015 MapR Technologies  MAPR®

8

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



8



This graphic depicts where Spark fits in the Big Data Application stack.

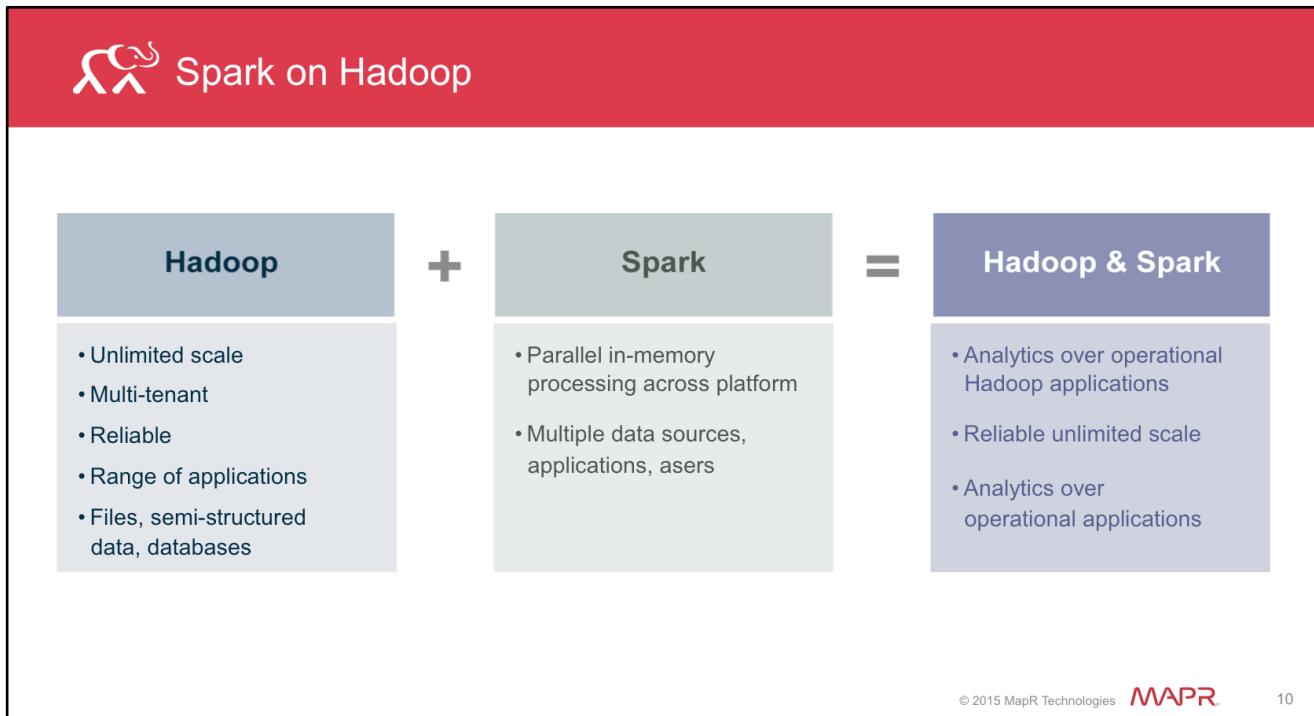
On the left we see different data sources. There are multiple ways of getting data in using different industry standards such as NFS or existing Hadoop tools.

The stack in the middle represents various Big Data processing workflows and tools that are commonly used. You may have just one of these workflows in your application, or a combination of many. Any of these workflows could read/write to or from the storage layer.

As you can see here, with Spark, you have a unified stack. You can use Spark for any of the workflows.

The output can then be used to create real-time dashboards and alerting systems for querying and advanced analytics; and loading into an enterprise data warehouse.





Hadoop

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

Spark

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

Hadoop + Spark

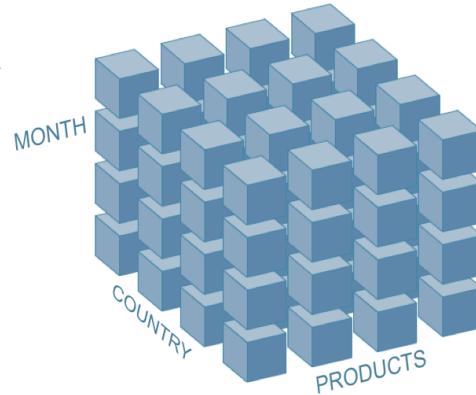
The combination of Spark and Hadoop takes advantage of both platforms, providing reliable, scalable, and fast parallel, in-memory processing. Additionally, you can easily combine different kinds of workflows to provide analytics over your Hadoop and other operational applications.



 Use Case

OLAP Analytics

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



© 2015 MapR Technologies 

11

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



 Use Case

Operational Analytics

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



© 2015 MapR Technologies 

12

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





Complex Data Pipelining

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



© 2015 MapR Technologies **MAPR**.

13

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



 Knowledge Check**What are the advantages of using Apache Spark?**

1. Compatible with Hadoop
2. Ease of development
3. Fast
4. Multiple Language support
5. Unified stack
6. All of the above

Answer: 6



 Learning Goals

Describe Features of Apache Spark

- How Spark fits in Big Data ecosystem
 - Why Spark & Hadoop fit together
- ▶ Define Spark Components

© 2015 MapR Technologies  MAPR®

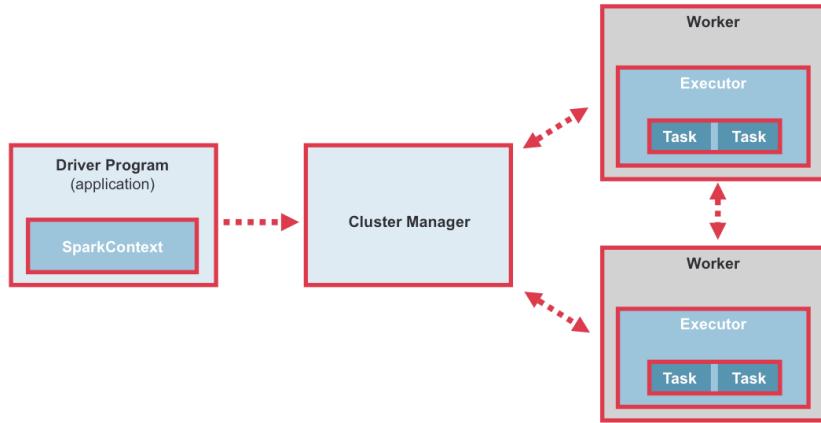
15

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





Spark Components



© 2015 MapR Technologies 

16

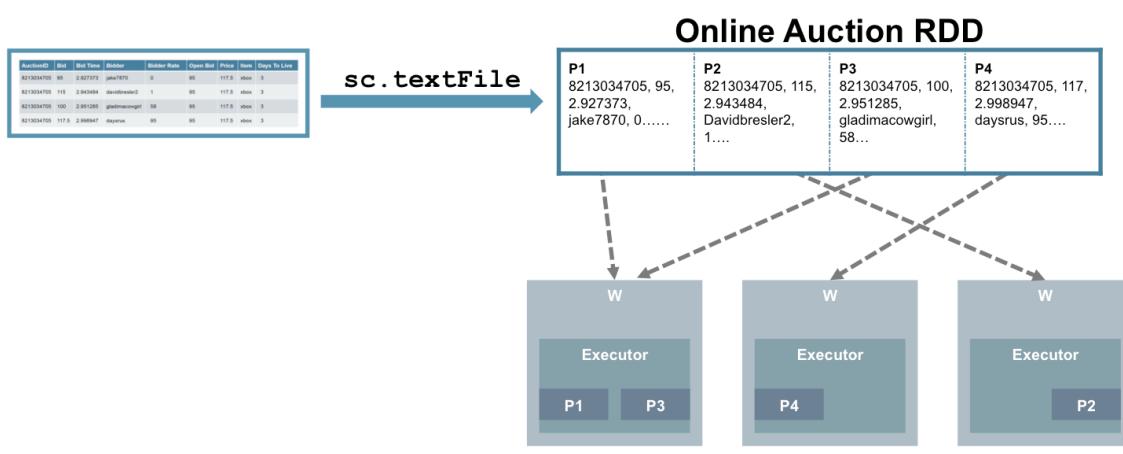
A Spark cluster consists of two processes, a driver program and multiple workers nodes each running an executor process. The driver program runs on the driver machine, the worker programs run on cluster nodes or in local threads.

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





Spark Resilient Distributed Datasets



© 2015 MapR Technologies

17

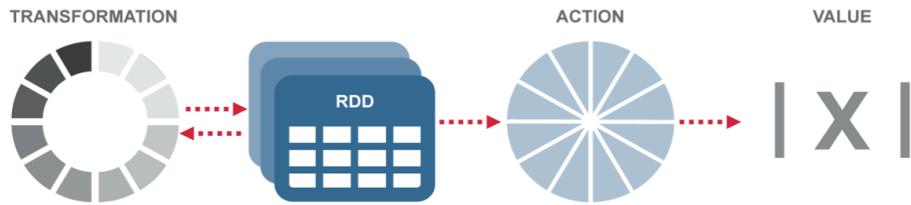
Resilient distributed datasets, or RDD, are the primary abstraction in Spark. They are a collection of objects that is distributed across nodes in a cluster, and data operations are performed on RDD.

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





Spark Resilient Distributed Datasets



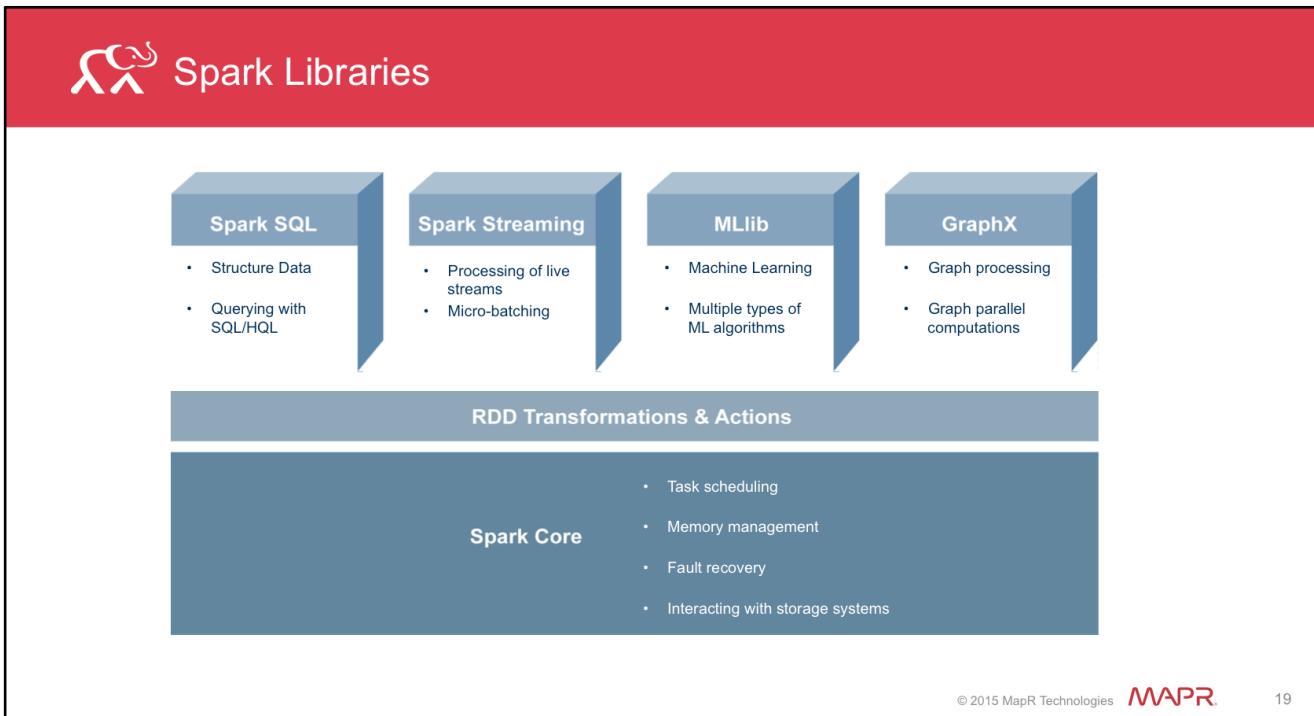
© 2015 MapR Technologies **MAPR**.

18

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

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





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

The higher level components are integrated into this stack.

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





Knowledge Check

Match the following:

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

1

SparkContext

2

RDD

3

Spark Core

1-b, 2-c, 3-a



 Next Steps

Lesson 2

Loading & Inspecting Data

© 2015 MapR Technologies  MAPR®

21

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



 MAPR Academy

Apache Spark Essentials

Lesson 2: Load & Inspect Data

© 2015 MapR Technologies  MAPR.

1

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



 Learning Goals

- ▶ Describe the different data sources and formats

Create & use Resilient Distributed Datasets (RDD)

Apply operations to RDDs

Cache intermediate RDD

Create & use DataFrames

© 2015 MapR Technologies  MAPR.

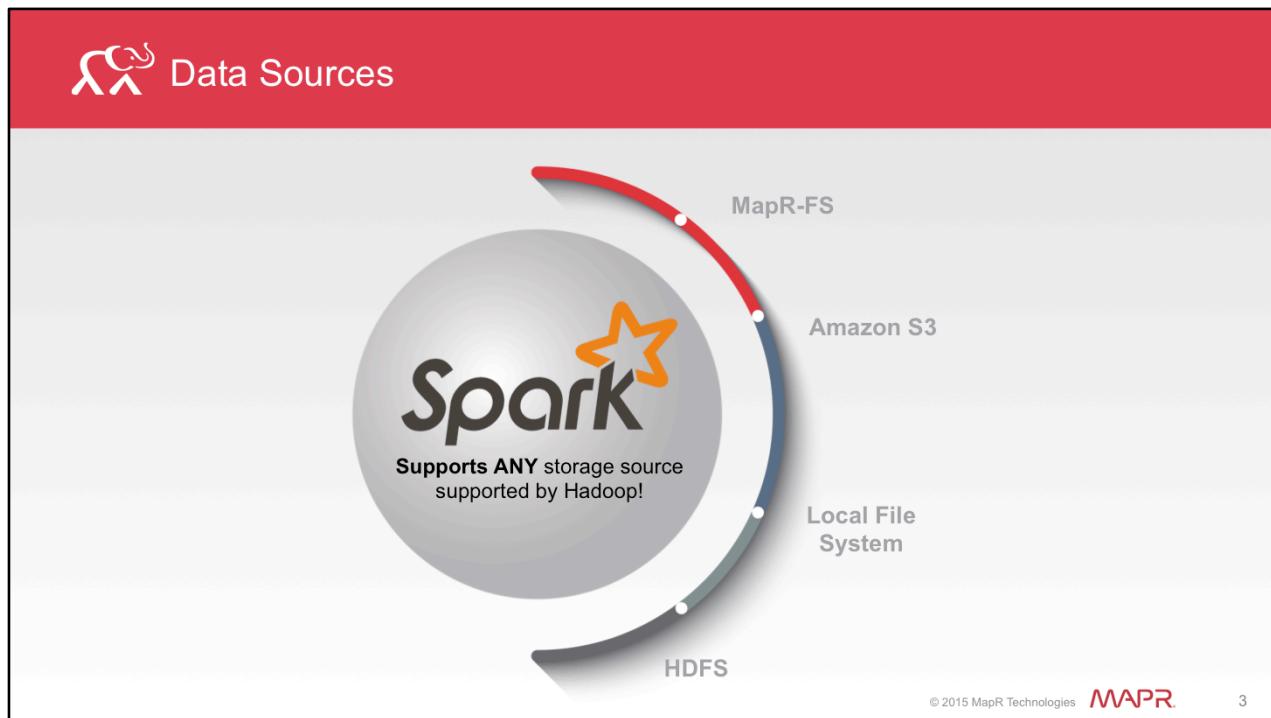
2

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

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

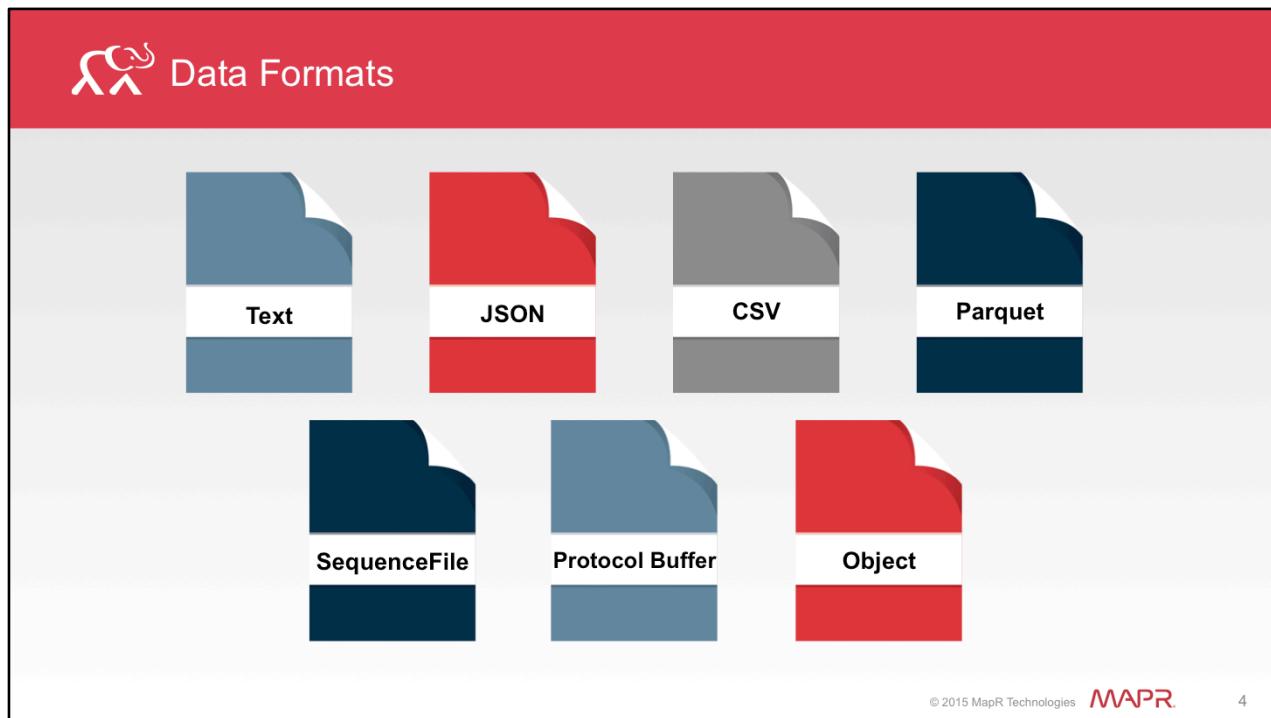
In the first section, we will look at the different data sources available to use with Spark.





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

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



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

 Learning Goals

Describe the different data sources and formats

▶ **Create & use Resilient Distributed Datasets (RDDs)**

Apply operations on RDDs

Cache intermediate RDD

Use Spark DataFrames for simple queries

© 2015 MapR Technologies  MAPR.

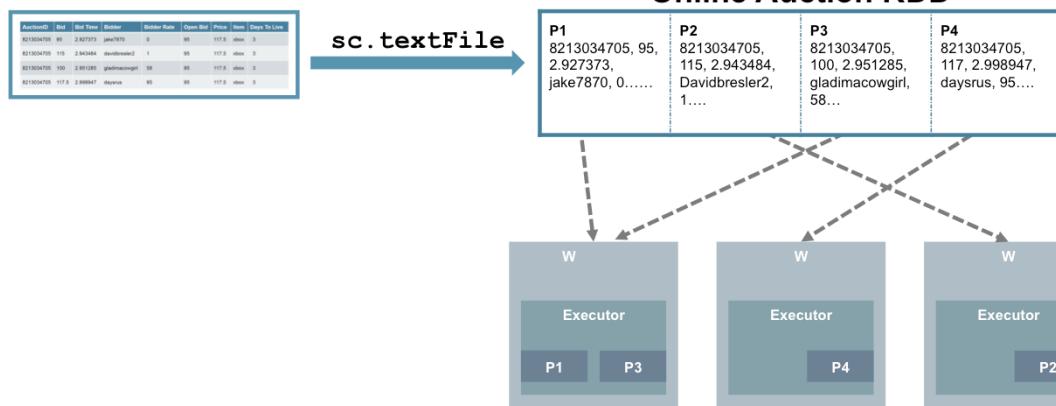
5

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





Review: Spark Resilient Distributed Datasets



6

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

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

Transformations are lazily evaluated i.e. They're not computed immediately and will only execute when an action runs on it.

You can also persist, or cache, RDDs in memory or on disk.

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





Scenario: Online Auction Data

AuctionID	Bid	Bid Time	Bidder	Bidder Rate	Open Bid	Price	Item	Days To Live
8213034705	95	2.927373	jake7870	0	95	117.5	xbox	3
8213034705	115	2.943484	davidbresler2	1	95	117.5	xbox	3
8213034705	100	2.951285	gladimacowgirl	58	95	117.5	xbox	3
8213034705	117.5	2.998947	daysrus	95	95	117.5	xbox	3

© 2015 MapR Technologies 

7

We are working with online Auction data that is in a CSV file in the local file system.

The data contains three types of items – xbox, cartier and palm.

Each row in this file represents an individual bid.

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





Scenario: Online Auction Data

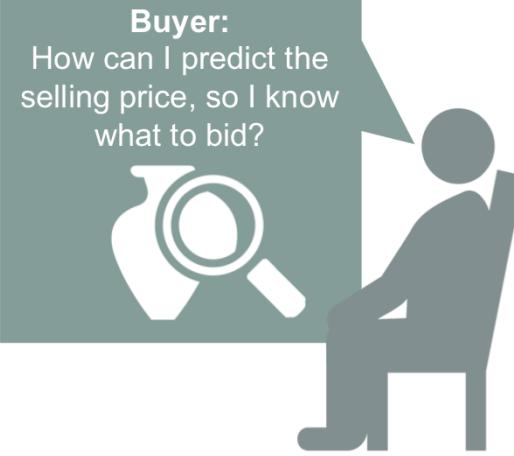


Seller:
What should be my starting price to get the highest selling price?



© 2015 MapR Technologies **MAPR**.

8



Buyer:
How can I predict the selling price, so I know what to bid?



These are some common questions to ask of auction data.

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

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





You can create an RDD from an existing collection or from external data sources.

When you load the data into Spark, you are creating an RDD.

The first RDD you create is usually called the **input RDD** or **base RDD**.

We are going to use the Spark Interactive Shell to load the auction data into Spark.



Spark Interactive Shell

- Write programs **INTERACTIVELY!**
- Scala or Python
- **INSTANT** feedback as code is entered
- **SparkContext** initialized on Shell start up



A screenshot of a computer monitor displaying the Spark Interactive Shell. The shell interface shows a command-line prompt and several lines of Scala code being typed and executed. The code includes importing the Spark Context, reading a text file from HDFS, and performing a count operation. The output shows the number of lines in the file.

© 2015 MapR Technologies  10

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





Creating RDD for Auction Data

1. Start Spark Interactive Shell



Scala /opt/mapr/spark/spark-<version>/bin/spark-shell



python™ /opt/mapr/spark/spark-<version>/bin/pyspark-shell

2. Load data into Spark using `SparkContext.textFile`

```
val auctionRDD = sc.textFile("path to file/auctiondata.csv")
```

© 2015 MapR Technologies 

11

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

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

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





Creating RDD from Different Data Sources

- **Text Files (returns one record per line)**
 - `SparkContext.textFile()`

- **SequenceFiles**
 - `SparkContext.sequenceFile[K,V]`

- **Other Hadoop InputFormats**

- `SparkContext.hadoopRDD`

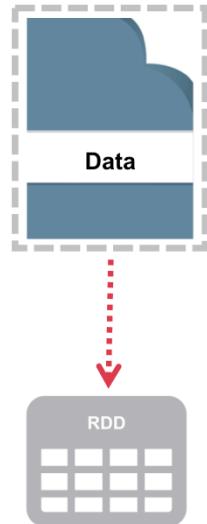
© 2015 MapR Technologies  MAPR.

12

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

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



 Lazy Evaluation

© 2015 MapR Technologies 

13

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

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





Knowledge Check

Which of the following is true of the Spark Interactive Shell?

1. Initializes SparkContext and makes it available
2. Available in Java
3. Provides instant feedback as code is entered
4. Allows you to write programs interactively

Answers: 1,3,4



 Learning Goals

Describe the different data sources and formats

Create & use Resilient Distributed Datasets (RDDs)

▶ **Apply operations on RDDs**

Cache intermediate RDD

Create & use DataFrames

© 2015 MapR Technologies  MAPR.

15

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



The slide has a red header bar with the title "Dataset Operations" and a logo. Below the header are two main sections: "Transformations" and "Actions". Each section contains a bulleted list of points and a decorative circular graphic at the bottom.

Transformations	Actions
<ul style="list-style-type: none">Creates new dataset from existing oneTransformed RDD executed only when action runs on itExamples: filter(), map(), flatMap()	<ul style="list-style-type: none">Return a value to driver program after computation on datasetExamples: count(), reduce(), take(), collect()

16

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

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

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

Some examples of Transformations are filter and map

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

Let us talk about transformations first.





Commonly Used Transformations

map	Returns new RDD by applying func to each element of source
filter	Returns new RDD consisting of elements from source on which function is true
groupByKey	Returns dataset (K, Iterable<V>) pairs on dataset of (K,V)
reduceByKey	Returns dataset (K, V) pairs where value for each key aggregated using the given reduce function
flatMap	Similar to map, but function should return a sequence rather than a single item
distinct	Returns new dataset containing distinct elements of source

© 2015 MapR Technologies 

17

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





Scenario: Want all Bids on XBox

```
8214889177,61,6.359155,714ark,15,0.01,90.01,xbox,7  
8214889177,76,6.359294,rjdorman,1,0.01,90.01,xbox,7  
8214889177,90,6.428738,baylorjeep,3,0.01,90.01,xbox,7  
8214889177,88,6.760081,jasonjasonparis,18,0.01,90.01,xbox,7  
8214889177,90.01,6.988831,gpgtpse,268,0.01,90.01,xbox,7  
1638893549,175,2.230949,schadenfreud,0,99,177.5,cartier,3  
1638893549,100,2.600116,chuik,0,99,177.5,cartier,3  
1638893549,120,2.60081,kiwisstuff,2,99,177.5,cartier,3  
1638893549,150,2.601076,kiwisstuff,2,99,177.5,cartier,3
```

© 2015 MapR Technologies  MAPR.

18

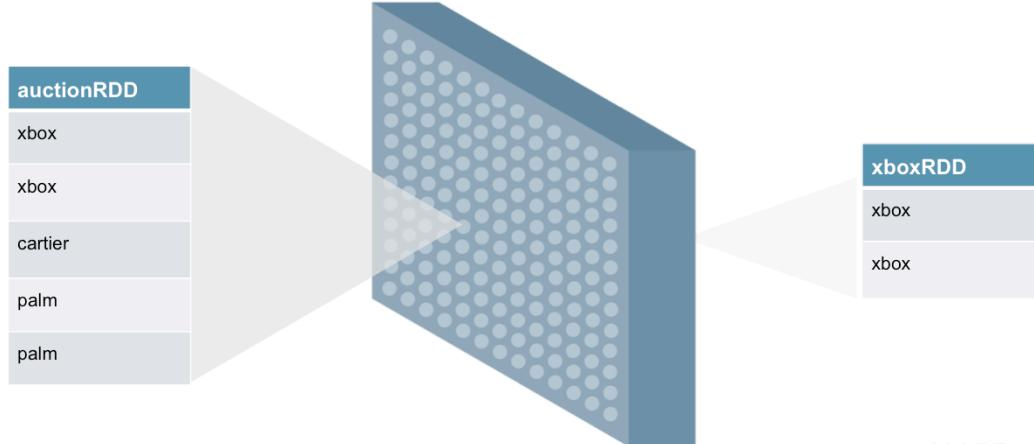
A sample of data in auctiondata.csv is shown here. Each line represents a bid on a particular item such as xbox, cartier, palm, etc. We only want to look at the bids on xbox.

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



 Transformation: filter()

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



© 2015 MapR Technologies 

19

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

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



 Anonymous Function

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

"=>"

Anonymous Syntax



© 2015 MapR Technologies 

20

The filter transformation is applying an anonymous function to each element of the RDD which is a line.

This is the Scala syntax for an anonymous function, which is a function that is not a named method.



20

 Anonymous Function

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

"line"
Input Variable

© 2015 MapR Technologies  MAPR.

21

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



 Anonymous Function

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

line.contains
("xbox")
Condition

© 2015 MapR Technologies  MAPR.

22

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

Anonymous functions can be used for short pieces of code.

We will now take a look at applying actions.





Commonly Used Actions

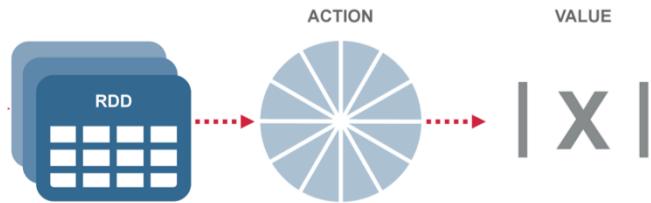
count()	Returns the number of elements in the dataset
reduce(func)	Aggregate elements of dataset using function func
collect()	Returns all elements of dataset as an array to driver program
take(n)	Returns an array with first n elements
first()	Returns the first element of the dataset
takeOrdered(n, [ordering])	Return first n elements of RDD using natural order or custom operator

© 2015 MapR Technologies  MAPR®

23

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



 Actions on RDD

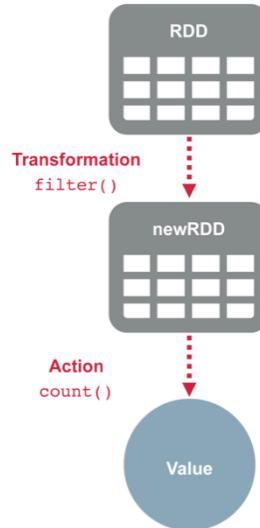
© 2015 MapR Technologies  MAPR.

24

An action on an RDD returns values to the driver program, after running the computation on the dataset.

As mentioned earlier, transformations are lazy. They are only computed when an action requires a result to be returned to the Driver program.



 Actions on RDD

© 2015 MapR Technologies 

25

Here the baseRDD and new RDD are not materialized till we add an action – count().

When we run the command for the action, Spark will read from the text file and create the base RDD. Then the filter transformation is applied to the baseRDD to produce the newRDD. Then count runs on the newRDD sending the value of the total number of elements in the RDD to the driver program.



 Actions on RDD

© 2015 MapR Technologies  MAPR.

26

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





Knowledge Check

Match the statements to the appropriate box:

**Actions**

2,4

OR

**Transformations**

1,3,5

1. Returns RDD
2. Returns a value
3. Computed lazily
4. Examples include count, take
5. Examples include filter, map

Actions: 2,4

Transformations: 1,3,5





Knowledge Check

Match the action to the query:

Query	Action
1. Total # of all bids	A. xboxRDD.take(10)
2. Total # of all bids on Xboxes	B. auctionRDD.count()
3. First 10 bids on Xboxes	C. xboxRDD.count()

© 2015 MapR Technologies  MAPR.

28

1 → b
2 → c
3 → a





Inspecting the Online Auction Data

- How many items were sold?
- How many bids per item type?
- How many different kinds of item types?
- What was the minimum number of bids?
- What was the maximum number of bids?
- What was the average number of bids?

© 2015 MapR Technologies  MAPR.

29

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





1. Set up Variables to Map Input

```
val auctionid = 0  
val bid = 1  
val bidtime = 2  
val bidder = 3  
val bidderrate = 4  
val openbid = 5  
val price = 6  
val itemtype = 7  
val daystolive = 8
```

© 2015 MapR Technologies  MAPR.

30

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



 2. Load the Data

```
val auctionRDD = sc.textFile("/user/user01/data/  
auctiondata.csv").map(_.split(","))  
  
line=>line.split(",")
```

© 2015 MapR Technologies 

31

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





Inspect Data: How Many Items Were Sold?

```
val items_sold = auctionRDD  
  .map(bid=>bid(auctionid))  
  .distinct  
  .count
```

Answer:

628 items were sold

© 2015 MapR Technologies  MAPR.

32

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

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

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

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





Inspecting Data: How Many Bids Per Item Type?

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

Answer:

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

© 2015 MapR Technologies  MAPR.

33

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

This line of code is explained in more detail next.

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





Walkthrough: itemtype()

```
val bids_item = auctionRDD  
.map(bid=>(bid(itemtype), 1))
```



TIP:
To see what the data looks like at this point, use take(1)
bids_item.take(1)

Results:

```
Array[(String, Int)] = Array((xbox,1), (xbox,1), (xbox,1),  
(xbox,1), (xbox,1).....(cartier,1),(cartier,  
1).....(palm,1),(palm,1).....(palm,1))
```

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



 Walkthrough: `reduceByKey()`

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

Results:

`Array[(String, Int)] = Array((palm,5917),.....)`



NOTE:
reduceByKey not the same as reducer in MapReduce.
reduceByKey – receives two values associated with key.
reducer in MapReduce receives list of values associated with key

© 2015 MapR Technologies 

35

In this example, `reduceByKey` runs on the key-value pairs from the map transformation – (`itemtype,1`) and aggregates on the key based on the function. The function that we have defined here for the `reduceByKey` transformation is sum of the values i.e `reduceByKey` returns key-value pairs which consists of the sum of the values by key.

Note that `reduceByKey` here does not work the same way as the reducer in MapReduce. `reduceByKey` receives two values associated with a key. The reducer in MapReduce receives a list of values associated with the key.





Walkthrough: collect()

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

Results:

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

© 2015 MapR Technologies 

36

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

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





Knowledge Check

Match the Data Operation to the result:

Data Operation	Result
1. <code>auctionRDD.first()</code>	A. Returns the total number of bids
2. <code>auctionRDD.map(func)</code>	B. Returns the func applied to each bid
3. <code>auctionRDD.count()</code>	C. Returns the first bid

© 2015 MapR Technologies  MAPR.

37

1 → c
2 → b
3 → a





Lab 2.1: Load & Inspect Online Auction Data



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



 Learning Goals

Describe the different data sources and formats

Create & use Resilient Distributed Datasets (RDD)

Apply operations on RDDs

▶ **Cache intermediate RDD**

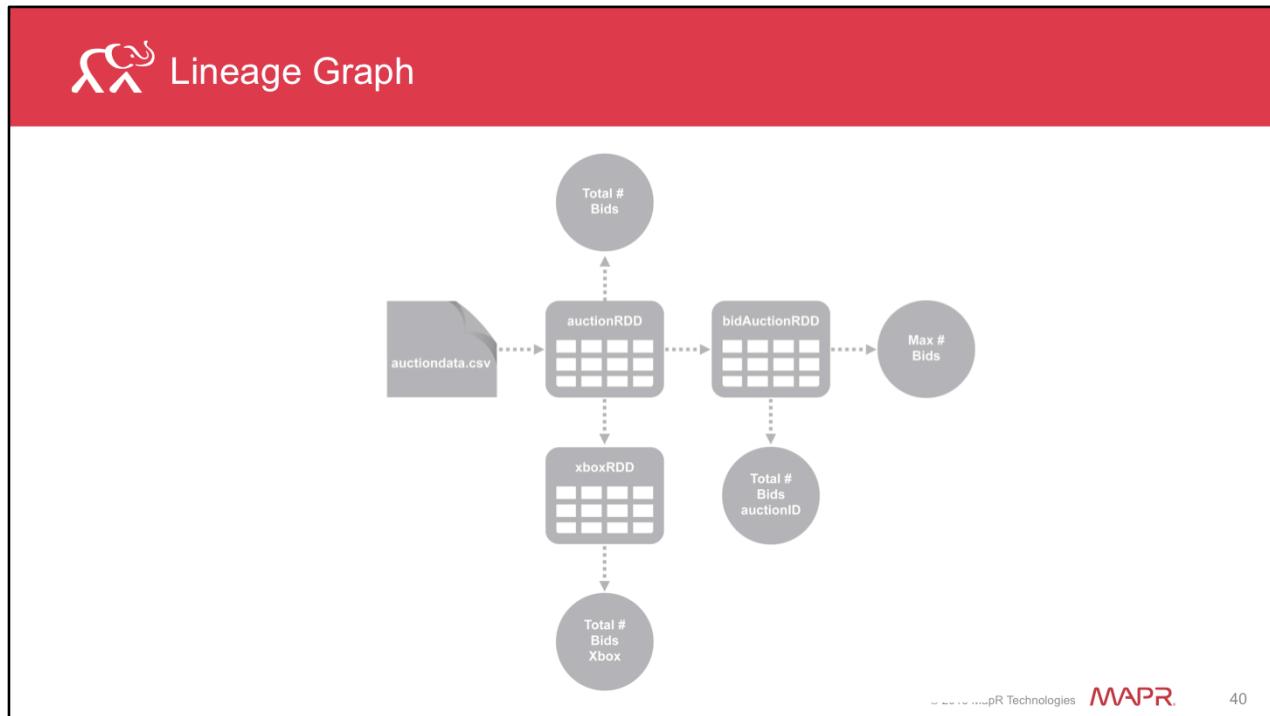
Create & use DataFrames

© 2015 MapR Technologies  MAPR.

39

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





© 2014 MapR Technologies MAPR.

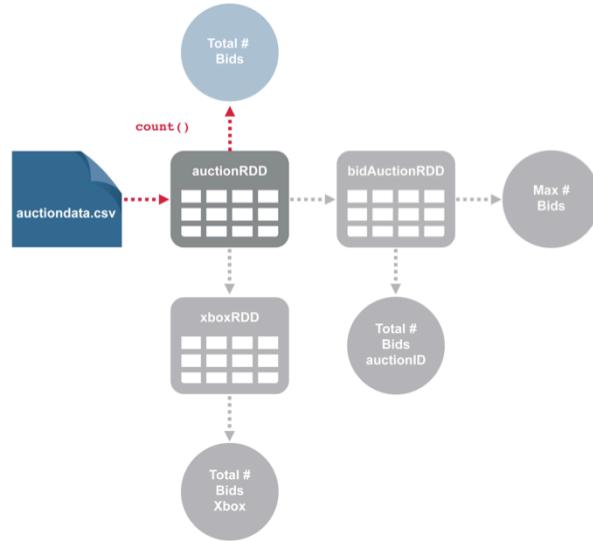
40

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

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



Lineage Graph: count()



© 2014 MapR Technologies

MAPR

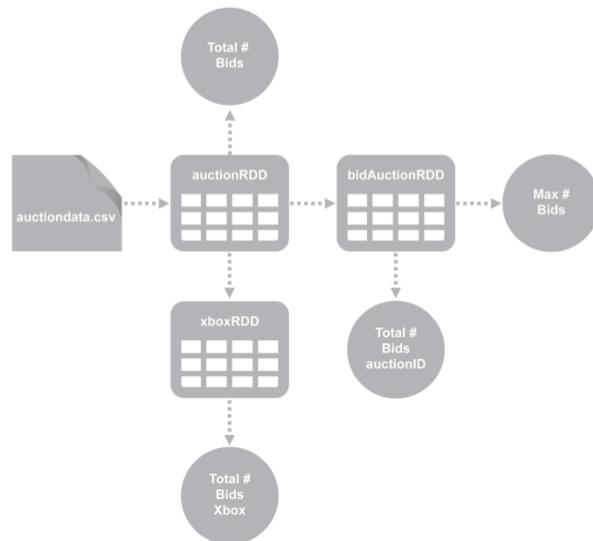
41

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





Lineage Graph: count()



© 2014 MapR Technologies

MAPR

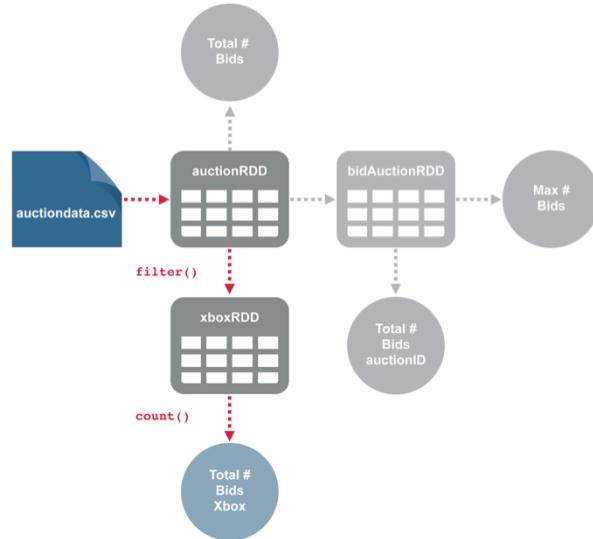
42

The data is no longer in memory.





Lineage Graph: count()



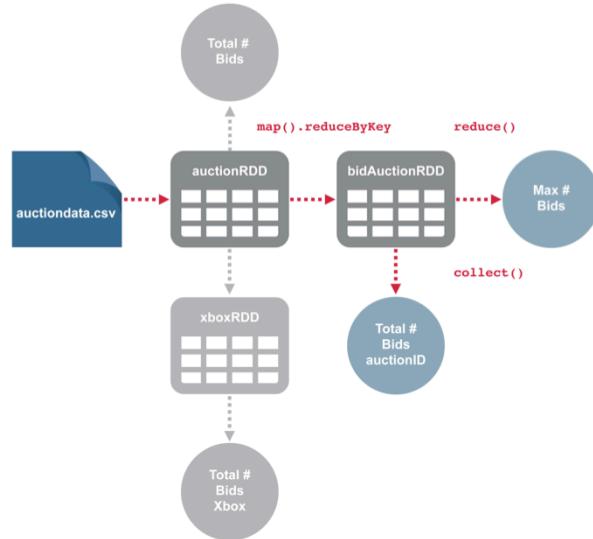
© 2014 MapR Technologies

MAPR.

43

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



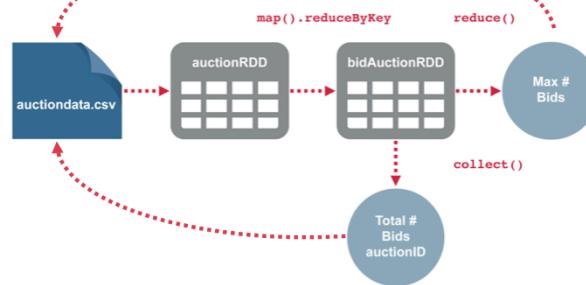
Lineage Graph: `collect()` & `reduce()`

© 2014 MapR Technologies

44

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



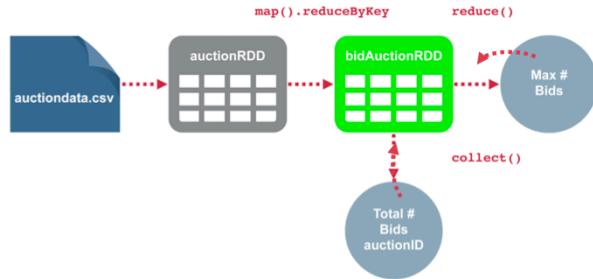
 Why Cache RDD

© 2014 MapR Technologies MAPR.

45

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



 Why Cache RDD© 2015 MapR Technologies 

46

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

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



 Caching the RDD

```
val auctionRDD = sc.textFile("/path/auctiondata.csv")

val bidAuctionRDD = auctionRDD
    .map(x=>(x(auctionid),1))
    .reduceByKey((x,y)=>x+y)

bidAuctionRDD.cache

bidAuctionRDD.collect
```

© 2015 MapR Technologies  MAPR.

47

1. The first line defines the instructions on how to create the RDD, though the file is not, yet read.
2. The second line is says to apply a transformation to the base RDD.
3. The third line says the cache the contents.
4. Again, nothing will happen until you to get the fourth line, the collect action. At this point, the auctiondata.csv is read, the transformation is applied, and the data is cached and collected

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





Knowledge Check

Which of the following is true of caching the RDD?

1. When there is branching in lineage, it is advisable to cache the RDD
2. Use rdd.cache() to cache the RDD
3. Cache behavior depends on available memory. If not enough memory, then action will reload from file instead of from cache
4. rdd.persist(MEMORY_ONLY) is the same as rdd.cache()
5. All of the above

Answers: 5



 Learning Goals

Describe the different data sources and formats available to use with Apache Spark

Create & use Resilient Distributed Datasets (RDDs)

Apply dataset operations on RDDs

Cache intermediate RDD

▶ **Create & use DataFrames**

© 2015 MapR Technologies  MAPR.

49

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





What is a Spark DataFrame?

- Programming abstraction in SparkSQL
- Distributed collection of data organized into named columns
- Scales from KBs to PBs
- Supports wide array of data formats & storage systems
- Works in Scala, Python, Java

© 2015 MapR Technologies  MAPR.

50

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

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



 Creating DataFrames

```
//1.Starting point is SQLContext
val sqlContext = new
org.apache.spark.sql.SQLContext(sc)

//2.Used to convert RDD implicitly into a DataFrame
import sqlContext.implicits._

//3. Define schema using a Case class
case class Auction(auctionid: String, bid: Float,
bidtime: Float, bidder: String, biderrate: Int,
openbid: Float, finprice: Float, itemtype: String,
dtl: Int)
```

© 2015 MapR Technologies  MAPR.

51

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

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

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



 Creating DataFrames**//4. Create the RDD**

```
val auctionRDD=sc.textFile("/user/user01/data/  
ebay.csv").map(_.split(","))
```

//5. Map the data to the Auctions class

```
val auctions=auctionRDD.map(a=>Auction(a(0),  
a(1).toFloat, a(2).toFloat, a(3), a(4).toInt,  
a(5).toFloat, a(6).toFloat, a(7), a(8).toInt))
```

© 2015 MapR Technologies  MAPR.

52

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

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



 Creating DataFrames

```
//6. Convert RDD to DataFrame  
val auctionsDF=auctions.toDF()  
  
//7. Register the DF as a table  
auctionsDF.registerTempTable("auctionsDF")
```

© 2015 MapR Technologies  MAPR.

53

6. We then convert the RDD to a DataFrame using the toDF() method.
7. In the last step, we register the DataFrame as a table. Registering it as a table allows us to use it in subsequent SQL statements.

Now we can inspect the data.





Inspecting Data: Examples

```
//To see the data  
auctionsDF.show()  
  
//To see the schema  
auctionsDF.printSchema()  
  
//Total number of bids  
val totbids=auctionsDF.count()  
  
//Show the columns in the DataFrame  
auctionsDF.columns
```

© 2015 MapR Technologies  MAPR.

54

Here are some examples of using DataFrame functions and actions.



 Commonly Use Actions

collect	Returns an array that contains all of rows in this DataFrame.
count	Returns the number of rows in the DataFrame
describe(cols)	Computes statistics for numeric columns, including count, mean, stddev, min, and max.
first(); head()	Returns first row
show()	Displays the first 20 rows of DataFrame in tabular form
take(n)	Returns the first n rows

© 2015 MapR Technologies  MAPR.

55

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





Commonly Used DataFrame Functions

cache()	Cache this DataFrame
columns	Returns all column names as an array
explain()	Only prints the physical plan to the console for debugging purposes
printSchema()	Prints the schema to the console in a tree format
registerTempTable(tableName)	Registers this DataFrame as a temporary table using the given name
toDF()	Returns a new DataFrame

© 2015 MapR Technologies  MAPR.

56

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



 Language Integrated Queries

agg(expr, exprs)	Aggregates on the entire DataFrame without groups
distinct	Returns a new DataFrame that contains only unique rows
filter(conditionExpr)	Filters based on given SQL expression
groupBy(col1, cols)	Groups DataFrame using the specified columns so we can run aggregation on them
select(cols)	Selects a set of columns based on expressions

© 2015 MapR Technologies 

57

This table lists commonly used language integrated queries. Refer to the [link here](#) for more information.



 Inspecting the Data

1. How many bids per item type?
2. What is the max, min and average number of bids per item?
3. How many distinct item types?
4. Get all the bids with closing price over 150?

```
auctionsDF.filter(auctionsDF("price")>150)  
.count()
```

© 2015 MapR Technologies  MAPR.

58

Can you answer the following?

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





Knowledge Check

Match the Data Operation to the result:

Query	DataFrame Operation
1. Distinct item types?	A. aDF.count()
2. Total number of bids?	B. aDF.show()
3. See 20 rows in DataFrame?	C. aDF.select("itemtype") .distinct().count()

© 2015 MapR Technologies  59

- 1 → c
2 → a
3 → b





Lab 2.2: Load & Inspect Data - DataFrames



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



 Next Steps

Lesson 3

Build a Simple Spark Application

© 2015 MapR Technologies  MAPR.

61

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



 MAPR Academy

Apache Spark Essentials

Lesson 3: Build a Simple Spark Application

© 2015 MapR Technologies  MAPR.

1

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



 Learning Goals

- ▶ Define the Spark program lifecycle
- ▶ Define the function of SparkContext
- ▶ Describe ways to launch Spark applications
- ▶ Launch a Spark application

© 2015 MapR Technologies  MAPR.

2

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

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





Lifecycle of a Spark Program

1. Create input RDDs in your driver program
2. Use lazy transformations to define new RDDs
3. Cache() any RDDs that are reused
4. Kick off computations using actions

© 2015 MapR Technologies  MAPR.

3

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

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

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

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





Lifecycle of a Spark Program – Step 1

1. Create input RDDs in your driver program

```
val auctionRDD = sc  
.textFile("/user/user01/data/auctiondata.csv")  
.map(line=>line.split(","))
```

© 2015 MapR Technologies  MAPR.

4

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





Lifecycle of a Spark Program – Step 2

2. Use lazy transformations to define new RDDs

```
val auctionRDD = sc  
  .textFile("/user/user01/data/auctiondata.csv")  
  .map(line=>line.split(","))  
val bidsitemRDD = auctionRDD.map(bid=>(bid(itemtype),1))  
.reduceByKey((x,y)=>x+y)
```

© 2015 MapR Technologies  MAPR.

5

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





Lifecycle of a Spark Program – Step 3

3. Cache() any RDDs that are reused

```
val auctionRDD = sc  
.textFile("/user/user01/data/auctiondata.csv")  
.map(line=>line.split(","))  
val bidsitemRDD = auctionRDD.map(bid=>(bid(item),1))  
.reduceByKey((x,y)=>x+y)  
bidsitemRDD.cache()
```

© 2015 MapR Technologies  MAPR.

6

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

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





Lifecycle of a Spark Program – Step 4

4. Kick off computations using actions

```
val auctionRDD = sc  
.textFile("/user/user01/data/auctiondata.csv")  
.map(line=>line.split(","))  
val bidsitemRDD = auctionRDD.map(bid=>(bid(item),1))  
.reduceByKey((x,y)=>x+y)  
bidsitemRDD.cache()  
bidsitemRDD.count()
```

© 2015 MapR Technologies  MAPR.

7

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

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





Knowledge Check

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

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

1

1,2,4,3

2

2,1,4,3

3

3,1,4,2

© 2015 MapR Technologies  MAPR.

8

Answers:



 Learning Goals

Define the Spark program lifecycle

- ▶ Define function of SparkContext

Describe ways to launch Spark applications

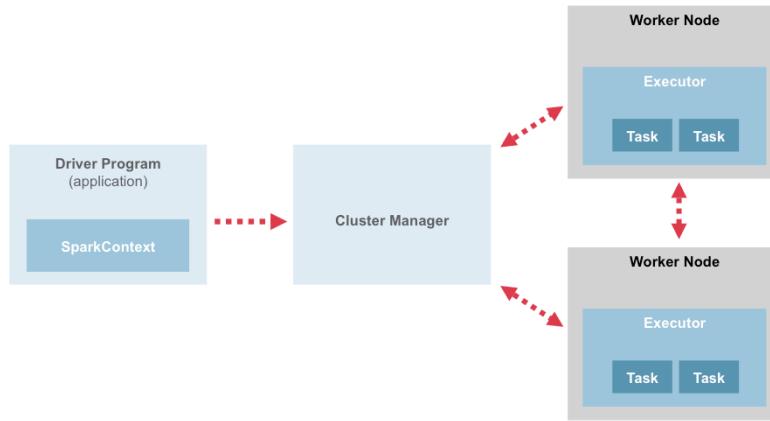
Launch a Spark application

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





Spark Components - Review



© 2015 MapR Technologies MAPR.

10

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.



 SparkContext

SparkContext

- Starting point of any Spark program
- Has methods to create, manipulate RDDs
- Is initialized by interactive shell and available as **sc**
- Needs to be initialized in standalone app
- Initialized with instance of `SparkConf` object

© 2015 MapR Technologies  MAPR.

11

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.



 Creating New SparkContext

```
import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext._  
import org.apache.spark.SparkConf  
  
val conf= new SparkConf().setAppName("AuctionsApp")  
val sc= new SparkContext(conf)
```

© 2015 MapR Technologies  MAPR.

12

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.





Start Building Application

```
/* AuctionsApp.scala - Simple App to inspect Auction data */
/* The following import statements are importing SparkContext, all subclasses and
SparkConf*/
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object AuctionApp{
    def main(args:Array[String]){
        val conf = newSparkConf().setAppName("AuctionsApp")
        val sc = new SparkContext(conf)
        /* Add location of input file */
        val aucFile ="/user/user01/data/auctiondata.csv"
        //build the inputRDD
        val auctionRDD = sc.textFile(aucFile).map(_.split(", ")).cache()
        ....
    }
}
```

© 2015 MapR Technologies  MAPR.

13

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 add 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`

Answers: 1,2,4





Lab 3.1 – Build the Spark Application



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.



 Learning Goals

Define the Spark program lifecycle

Define function of SparkContext

- ▶ Describe ways to launch Spark applications

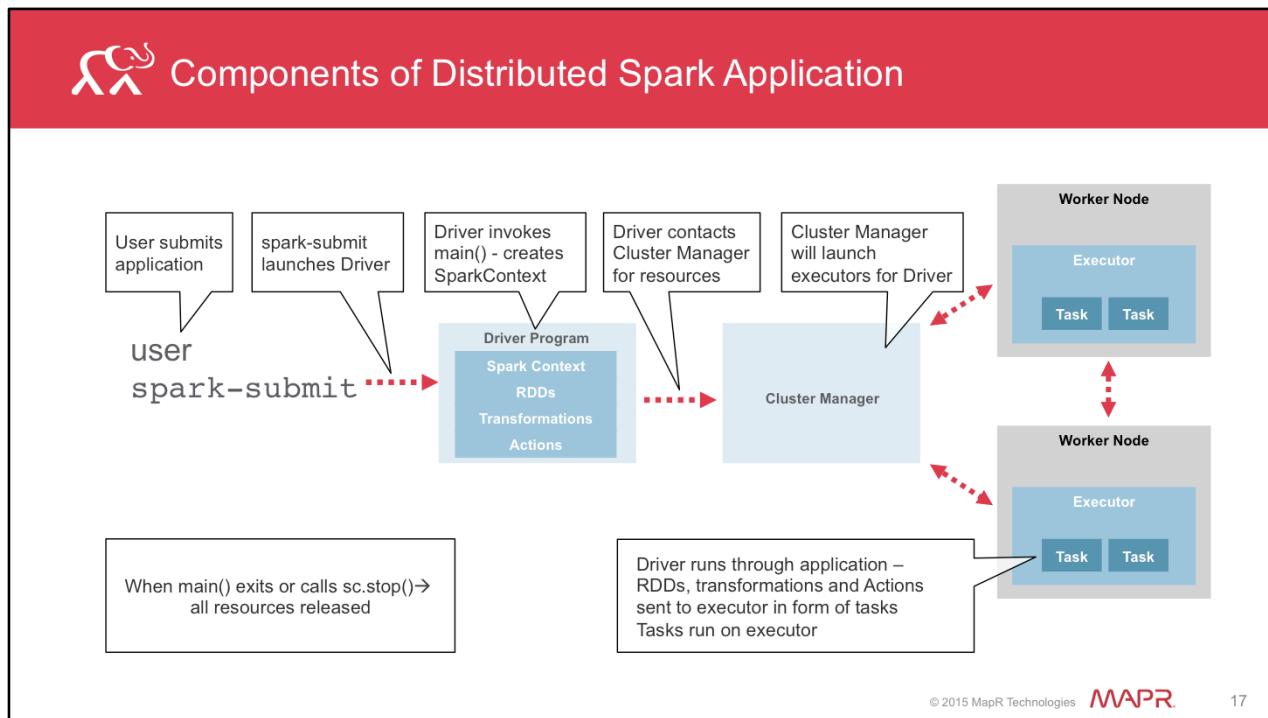
Launch a Spark application

© 2015 MapR Technologies  MAPR.

16

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



Ways to Run a Spark Application

1. Local – runs in same JVM
2. Standalone – simple cluster manager
3. Hadoop YARN – resource manager in Hadoop 2
4. Apache Mesos – general cluster manager

© 2015 MapR Technologies  MAPR.

18

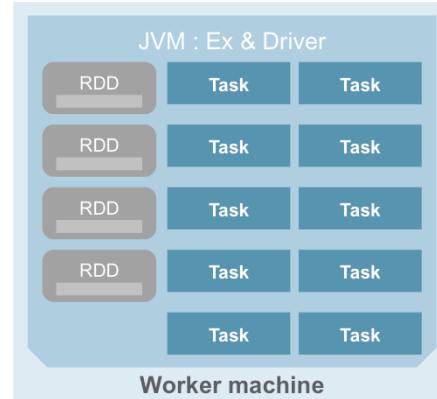
We have seen what occurs when you run a Spark application on a cluster. Spark uses the cluster manager to launch executors. The cluster manager is a pluggable component in Spark, which is why Spark can run on various external managers such as Hadoop YARN, Apache Mesos. Spark also has its own built-in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



© 2015 MapR Technologies MAPR.

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.





Running in Standalone Mode

- Simple standalone deploy mode
- Launch
 - Manually
 - Use provided scripts
- Install by placing compiled version of Spark on each cluster node
- Two deploy modes
 - Cluster mode
 - Client mode

© 2015 MapR Technologies  MAPR.

20

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](#).





Standalone – Cluster and Client modes

Cluster Mode	Client Mode
Driver launched from worker process inside cluster	Driver launches in the client process that submitted the job
Can quit without waiting for job results (async)	Need to wait for result when job finishes (sync)

If using spark-submit → application automatically distributed to all worker nodes

© 2015 MapR Technologies  MAPR.

21

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.



 Hadoop YARN

- Run on YARN if you have Hadoop cluster
 - Uses existing Hadoop cluster
 - Uses features of YARN scheduler
- Can connect to YARN cluster
 - Cluster mode
 - Client mode

© 2015 MapR Technologies  MAPR.

22

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 the 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.





Hadoop YARN – Cluster and Client modes

Cluster Mode	Client Mode
Driver launched in Application Master in cluster or worker	Driver launched in the client process that submitted the job
Can quit without waiting for job results (async)	Need to wait for result when job finishes (sync)
Suitable for production deployments	Useful for Spark interactive shell or debugging

© 2015 MapR Technologies 

23

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.

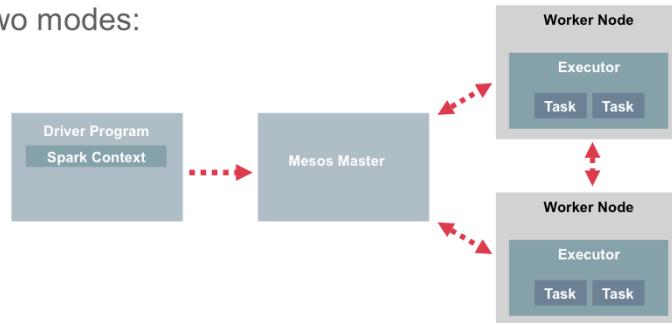
There are two deploy modes that can be used to launch Spark applications on YARN. In yarn-cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster. In yarn-client mode, the driver runs in the client process, and the application master is only used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





Apache Mesos Cluster Manager

- Mesos master replaces Spark Master as Cluster Manager
- Driver creates job
 - Mesos determines what machines handle what tasks
- Multiple frameworks can coexist on same cluster
- Spark can run in two modes:
 - Fine-grained
 - Coarse



© 2015 MapR Technologies MAPR.

24

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”.





Apache Mesos – Coarse and Fine-grained

Fine-grained (Default mode)	Coarse
Each Spark task runs as a separate Mesos task	Launches only one long-running task on each Mesos machine
Useful for sharing	No sharing Uses cluster for complete duration of application
Additional overhead at launching each task	Much lower startup overhead

© 2015 MapR Technologies 

25

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.





Summary – Deploy Modes

Mode	Purpose
Local	Good for prototyping & testing
Standalone	Easiest to set up & useful if only running Spark
YARN	Resource scheduling features when using Spark & other applications Advantageous for existing Hadoop cluster
Apache Mesos	Resource scheduling features when using Spark & other applications Fine-grained option useful for multiple users running interactive shell

© 2015 MapR Technologies 

26

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.



 spark-submit

/spark-home/bin/spark-submit – used to run in any mode

```
./bin/spark-submit \
--class <main-class>
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \
... # other options
<application-jar> \
[ application-arguments ]
```

© 2015 MapR Technologies 

27

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.



 spark-submit – Examples (2)**Run on YARN cluster:**

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

Run on YARN client

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

© 2015 MapR Technologies 

29

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

Answer: 2



 Learning Goals

- Define the Spark program lifecycle
- Define function of SparkContext
- Describe ways to launch Spark applications
 - ▶ Launch a Spark application

© 2015 MapR Technologies  MAPR.

31

We will now launch our Spark application.





Launch a Program

1. Package application & dependencies into a .jar file (SBT or Maven)
2. Use `./bin/spark-submit` to launch application

© 2015 MapR Technologies  MAPR.

32

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





1. Package the AuctionsApp Application

Use sbt to package the app:

- i. Create auctions.sbt file

```
name := "Auctions Project"
version:= "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.3.1"
```

- ii. Create the following folder structure under your working folder (Lab3)

```
[user01@maprdemo LAB3]$ find .
.
./auctions.sbt
./src
./src/main
./src/main/scala
./src/main/scala/AuctionsApp.scala
```

- iii. From working directory sbt package

```
[user01@maprdemo ~]$ cd Lab3
[user01@maprdemo Lab3]$ sbt package
[info] Set current project to Auctions Project (in build file:/user/user01/Lab3/)
[info] Compiling 1 Scala source to /user/user01/Lab3/target/scala-2.10/classes...
[info] Packaging /user/user01/Lab3/target/scala-2.10/auctions-project_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time: 17 s, completed Jul 20, 2015 9:05:46 AM
```

© 2015 MapR Technologies 

33

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.



 2. spark-submit

Use `spark-submit` to launch application

```
$ /opt/mapr/spark/spark-1.3.1/bin/spark-submit \
--class "AuctionsApp" \
--master local \
target/scala-2.10/auctions-project_2.10-1.0.jar
```

For Python applications, pass .py file directly to `spark-submit` instead of .jar

© 2015 MapR Technologies  MAPR.

34

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`.



The screenshot shows the MapR Control System (MCS) interface. The top navigation bar has tabs for 'Dashboard' and 'SparkHistoryServer'. The left sidebar navigation includes sections for Cluster, MapR FS, NFS HA, Alarms, System Settings, and various monitoring tools like HBase, Job Tracker, CLDB, and Nagios. The 'SparkHistoryServer' option is highlighted with a red box. The main content area displays the 'Spark History Server' page with the title 'Spark 1.3.1 History Server'. It shows the event log directory as 'maprfs://apps/spark' and lists one application: 'Showing 1-1 of 1'. The table shows the following data:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
application_1435684715185_0001	AuctionsApp	2015/07/01 21:27:04	2015/07/01 21:27:41	38 s	user01	2015/07/01 21:27:41

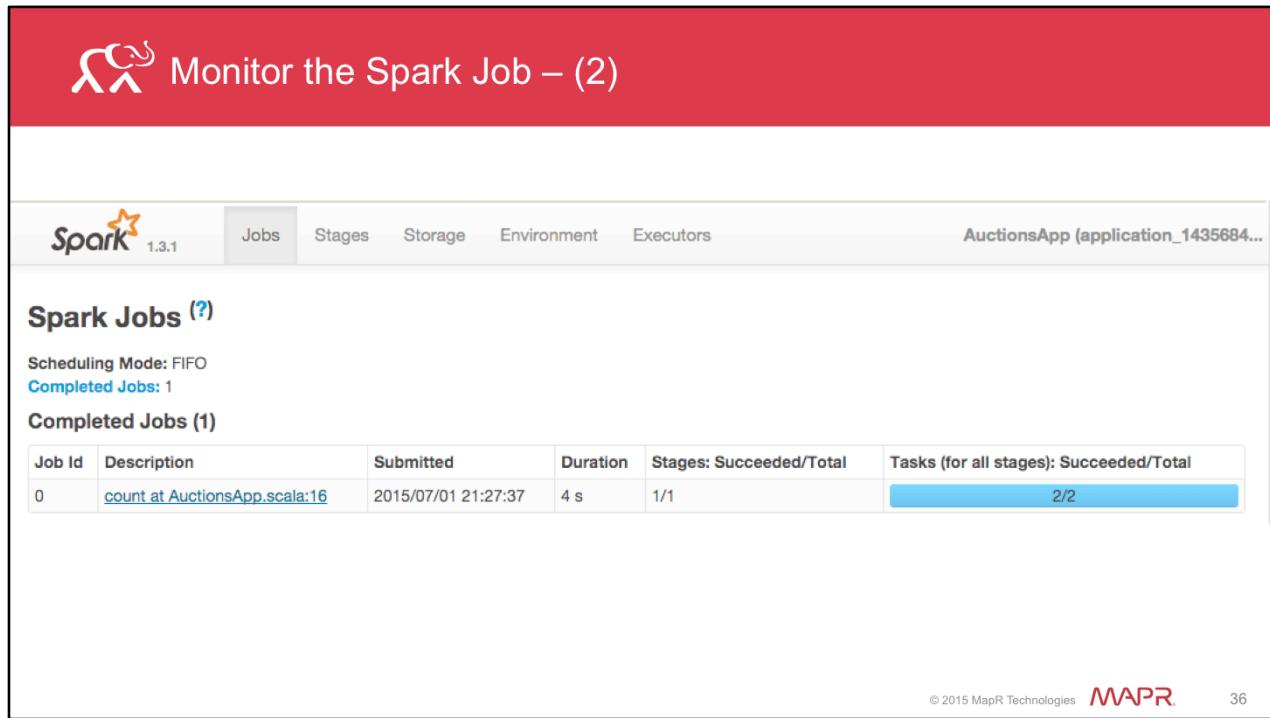
Below the table, there is a link to 'Show incomplete applications'. To the right of the table, instructions are provided to 'Launch MapR Control System (MCS)' with two steps:

- <http://<ip address>:8443>
- Select SparkHistoryServer

At the bottom right of the interface, it says '© 2015 MapR Technologies' and 'MapR'.

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.





The screenshot shows the MapR Spark UI interface. At the top, there's a red header bar with the text "Monitor the Spark Job – (2)" and a spark icon. Below the header is a navigation bar with tabs: "Spark 1.3.1" (selected), "Jobs" (selected), "Stages", "Storage", "Environment", and "Executors". To the right of the tabs, it says "AuctionsApp (application_1435684...)". The main content area is titled "Spark Jobs (?)". It shows "Scheduling Mode: FIFO" and "Completed Jobs: 1". A table titled "Completed Jobs (1)" lists one job: "Job Id: 0, Description: count at AuctionsApp.scala:16, Submitted: 2015/07/01 21:27:37, Duration: 4 s, Stages: Succeeded/Total: 1/1, Tasks (for all stages): Succeeded/Total: 2/2". At the bottom right of the content area, it says "© 2015 MapR Technologies" and "MAPR".

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

© 2015 MapR Technologies  MAPR®

37





Lab 3.2 – Run a Standalone Spark Application



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.





DEV 361

Develop, Deploy and Monitor
Apache Spark Applications

© 2015 MapR Technologies  MAPR.

39

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

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2016, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.



Lesson 4: Work with Pair RDD



DEV 361: Build and Monitor Apache Spark Applications

Lesson 4: Work with Pair RDD

© 2015 MapR Technologies 1

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.

 Learning Goals

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

© 2015 MapR Technologies 

2

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



 Review

- Create RDD
- Apply transformations to RDD
- Apply actions to RDD

© 2015 MapR Technologies 

3

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.





Scenario: SFPD Incidents Data

Note:
This information is from the file sfpd.csv

IncidentNum → Incident number	150599321
Category	OTHER_OFFENSES
Description	POSSESSION_OF_BURGLARY_TOOLS
DayOfWeek	Thursday
Date	7/9/15
Time	23:45
PdDistrict → PD District	CENTRAL
Resolution	ARREST/BOOKED
Address	JACKSON_ST/POWELL_ST
X → Longitudinal Coordinate	-122.4099006
Y → Latitudinal Coordinate	37.79561712
PdID → Police Department ID	15059900000000

© 2015 MapR Technologies 

4

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.





Review: Map Input Fields

1. Map input fields

```
val IncidntNum = 0  
val Category = 1  
val Descript = 2  
val DayOfWeek = 3  
val Date = 4  
val Time = 5  
val PdDistrict = 6  
val Resolution = 7  
val Address = 8  
val X = 9  
val Y = 10  
val PdId = 11
```

© 2015 MapR Technologies 

5

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.



 Review: Load Data into Spark

2. Load data and split on separator

```
val sfpd = sc.textFile("/path/to/file/  
sfpd.csv").map(line=>line.split(","))
```

© 2015 MapR Technologies 

6

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.





Review: Transformations & Actions

How do you know what the data looks like in the RDD?

```
sfpd.first()
```

What is the total number of incidents?

```
val totincs = sfpd.count()
```

What are the Categories?

```
val cat = sfpd.map(inc=>inc(Category)).distinct.collect()
```

© 2015 MapR Technologies 

7

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.





Review: Transformations

Note:
Transformations
are lazily evaluated



bayviewRDD; note this is a **new RDD**

```
val bayviewRDD = sfpd.filter(incident=>incident.contains("BAYVIEW"))  
filter()
```

© 2015 MapR Technologies 

8

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.



 Review: Actions

Note:
Actions evaluated immediately. Every action will compute from start.

```
val numTenderloin=sfpd  
.filter(incident=>incident.contains("TENDERLOIN")).count()
```

Returns a value of the
count() to driver

© 2015 MapR Technologies 

9

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.





Lab 4.1 – REVIEW: Load Data & Explore Data

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.



 Learning Goals

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

© 2015 MapR Technologies 

11

Notes on Slide 11:

In this section we will describe key-value pairs as data sets and create pair RDDs.



The diagram illustrates a **Pair RDD** as a collection of **2-Tuples**. A central blue box labeled "Pair RDD" contains four entries: (CENTRAL,2), (TENDERLOIN, 1), (RICHMOND, 0), and (BAYVIEW, 6). Red callout boxes point to these elements: one points to the first entry with the label "Key Identifier", another points to the second entry with the label "Value", and a third points to the entire list with the label "2-Tuple".

Note:
Pair RDD is a type
of RDD. Each
element of Pair
RDD is a 2-tuple

© 2015 MapR Technologies **MAPR** 12

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





Motivation for Pair RDD

Useful for:

- Aggregating data across network
- Regrouping data across network

Example:

Want to view all SFPD incident categories together by district

Key-Value pairs used in Map - Reduce algorithms

© 2015 MapR Technologies

13

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 an 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.





Common Example: Pair RDD in Word Count

```
val textFile=spark.textFile("hdfs://...")  
val count=textFile.flatMap(line=>line.split(","))  
    .map(word=>(word,1))  
    .reduceByKey(_+_)  
count.saveAsTextFile("hdfs://...")
```

© 2015 MapR Technologies

14

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.





Word Count: Apache Spark vs Apache Hadoop MapReduce

Spark Word Count

```
val textFile=spark.textFile("hdfs://...")
val counts=textFile.flatMap(line=>line.split(","))
    .map(word=>(word,1))
    .reduceByKey{+_}
counts.saveAsTextFile("hdfs://...")
```

MapReduce Word Count

```
import java.io.IOException
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.Path
import org.apache.hadoop.io.IntWritable
import org.apache.hadoop.io.Text
import org.apache.hadoop.mapreduce.Mapper
import org.apache.hadoop.mapreduce.Reducer
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
public class WordCountMapper extends Mapper<Text, IntWritable>
{
    private Text word = new Text();
    private IntWritable value = new IntWritable();
    public void map(Text key, IntWritable value, Context context
    ) throws IOException, InterruptedException {
        StringTokenizer it = new StringTokenizer(key.toString());
        while(it.hasMoreTokens()){
            word.set(it.nextToken());
            value.set(1);
            context.write(word, value);
        }
    }
    public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable>
    {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values,
                           Context context
                           ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true));
    }
}
```

© 2015 MapR Technologies 15

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.





Apache Spark vs Apache Hadoop MapReduce

Apache Spark

- Spark tries to keep everything in memory
- Chaining multiple jobs in faster
- A combination of any number of map & reduce operations
- map() & reduce() already implemented in Spark
- Support for the Interactive Shell

Apache Hadoop MapReduce

- Read & write from/to disk after every job
- One map & one reduce per job
- Have to implement these methods in Hadoop MapReduce
- Jobs are all batch (non-interactive)

© 2015 MapR Technologies



16

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 from Scala and Python that allows you to write your code interactively. Apache Hadoop MapReduce jobs on the other hand are all batch.

We will now take a look at creating pair RDDs.





Create Pair RDD

1. Create from existing non-pair RDD
2. Create pair RDD by loading certain formats
3. Create pair RDD from in-memory collection of pairs

© 2015 MapR Technologies 

17

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.





1. Create From Existing Non-Pair RDD

- Different ways to create from existing non-pair RDD
- Most common way is use `map()` on existing RDD
- Building key-value RDDs differs by language

Example:

```
val incByCat = sfpd.map(incident=>(incident(Category),1))
```

Results:

```
Array((LARCENY/THEFT, 1), ...)
```

© 2015 MapR Technologies



18

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.





2. Create Pair RDD By Loading Certain Formats

- Many formats will directly return pair RDD
- Examples
 - SequenceFiles create pair RDD
 - sc.wholeTextFile on small files creates pair RDD

rdd from SequenceFile

```
Array((OTHER_OFFENSES,1), (LARCENY/THEFT,1),  
(OTHER_OFFENSES,1), ... )
```

© 2015 MapR Technologies  19

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.





3. Create Pair RDD From In-Memory Collection of Pairs

Call `SparkContext.parallelize()` on collection of pairs

```
//creating a collection of pairs in memory
val dist1 = Array(("INGLESIDE",1), ("SOUTHERN",1), ("PARK",
1), ("NORTHERN",1))
// Use parallelize() to create a Pair RDD
val distRDD = SparkContext.parallelize(dist1)

Array[(String, Int)] = Array((INGLESIDE,1), (SOUTHERN,1), (PARK,1)
(NORTHERN,1))
```

© 2015 MapR Technologies 

20

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 pairs, `dist1`, in memory. We create a pair RDD, `distRDD`, by applying the method `SparkContext.parallelize` to `dist1`.



 Use Case

Which three districts have the highest number of incidents?

1. Create a pair RDD
 - Use existing sfpd RDD (non-pair)
 - Apply `map()`
2. "Combine/collect" by district
3. Sort & show top 3

```
val top3Dists = sfpd.map(inc=>(inc(PdDistrict),1))
```

Result:

```
Array((CENTRAL,1), (CENTRAL,1), (CENTRAL,1), (PARK,1),...)
```

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.



 Knowledge Check

Given the sfpd RDD, to create a Pair RDD consisting of tuples of the form (Category, 1), in Scala, use:

1. val pairs = sfpd.parallelize()
2. val pairs = sfpd.map(x=>(x(Category),1))
3. val pairs=sfpd.map(x=>x.parallelize())

Notes on Slide 22:

Answer: 2



 Learning Goals

1. Describe & create pair RDD
- ▶ **2. Apply transformations & actions to pair RDD**
3. Control partitioning across nodes

© 2015 MapR Technologies  23

Notes on Slide 23:

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





Transformation on Pair RDDs

- **Transformations specific to pair RDD**
 - `reduceByKey()`, `groupByKey()`, `aggregateByKey()`,
`combineByKey()`
- **Transformations that work on two pair RDD**
 - `join()`, `cogroup()`, `subtractByKey()`
- **Transformations that apply to RDD**
 - `filter()`, `map()`

© 2015 MapR Technologies 

24

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()`.





Apply Transformation to Pair RDD

Which three districts have the highest number of incidents?

1. Create a pair RDD
2. “Combine” or “add” by key
3. Sort
4. Show top 3

```
val top3Dists=sfpd.map(incident=>(incident(PdDistrict),1))  
.reduceByKey (x,y)=>x+y)
```

© 2015 MapR Technologies

MAPR

25

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.





Apply Transformation to Pair RDD

Which three districts have the highest number of incidents?

1. Create a pair RDD
2. “Combine” by key
3. Sort
4. Show top 3

```
val top3Dists=sfpd.map(incident=>(incident(PdDistrict),1))
    .reduceByKey((x,y)=>x+y)
    .map(x=>(x._2,x._1))
    .sortByKey(false).take(3)
```

© 2015 MapR Technologies

26

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(_.value))
```





Note:
You can use
groupByKey instead
of reduceByKey



Which three districts have the highest number of incidents?

```
val pddists = sfpd.map(x=> (x(PdDistrict),1))
  .groupByKey.map(x => (x._1, x._2.size))
  .map(x=>(x._2,x._1))
  .sortByKey(false).take(3)
```

© 2015 MapR Technologies

27

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.

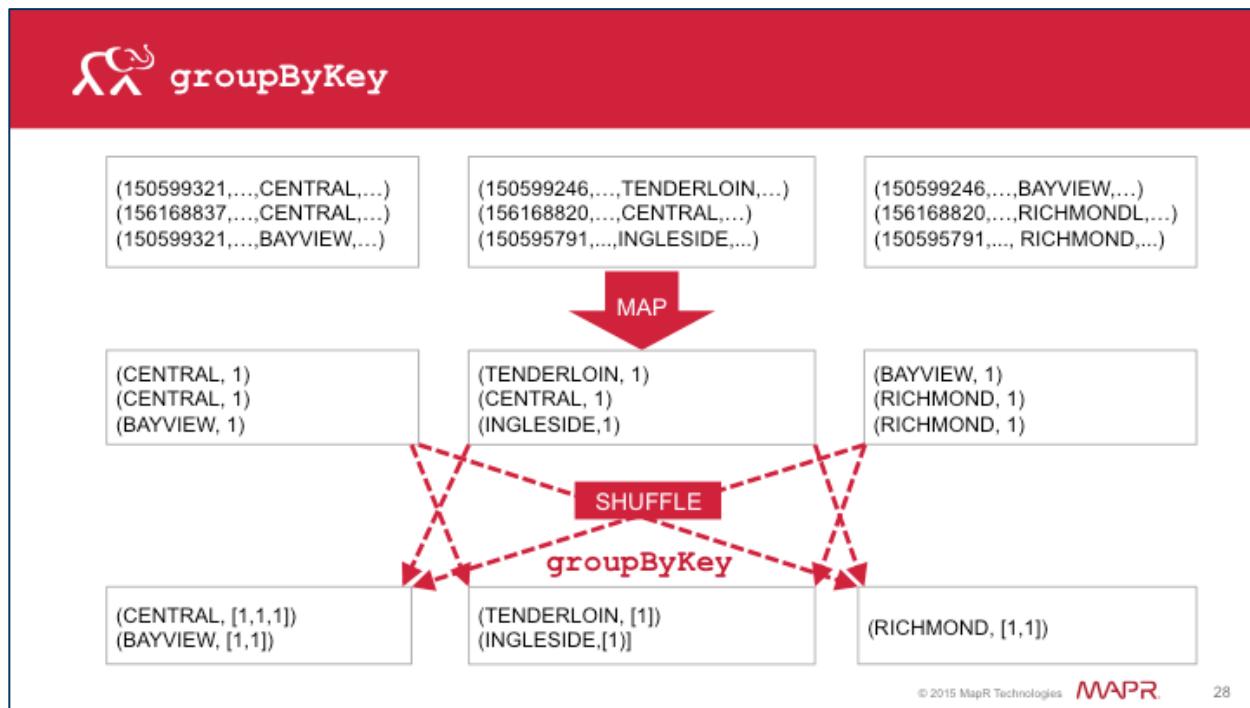
It groups all values that have the same key. As a result, it takes no argument. It results in a dataset of key/value pairs where the value is an iterable list.

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))

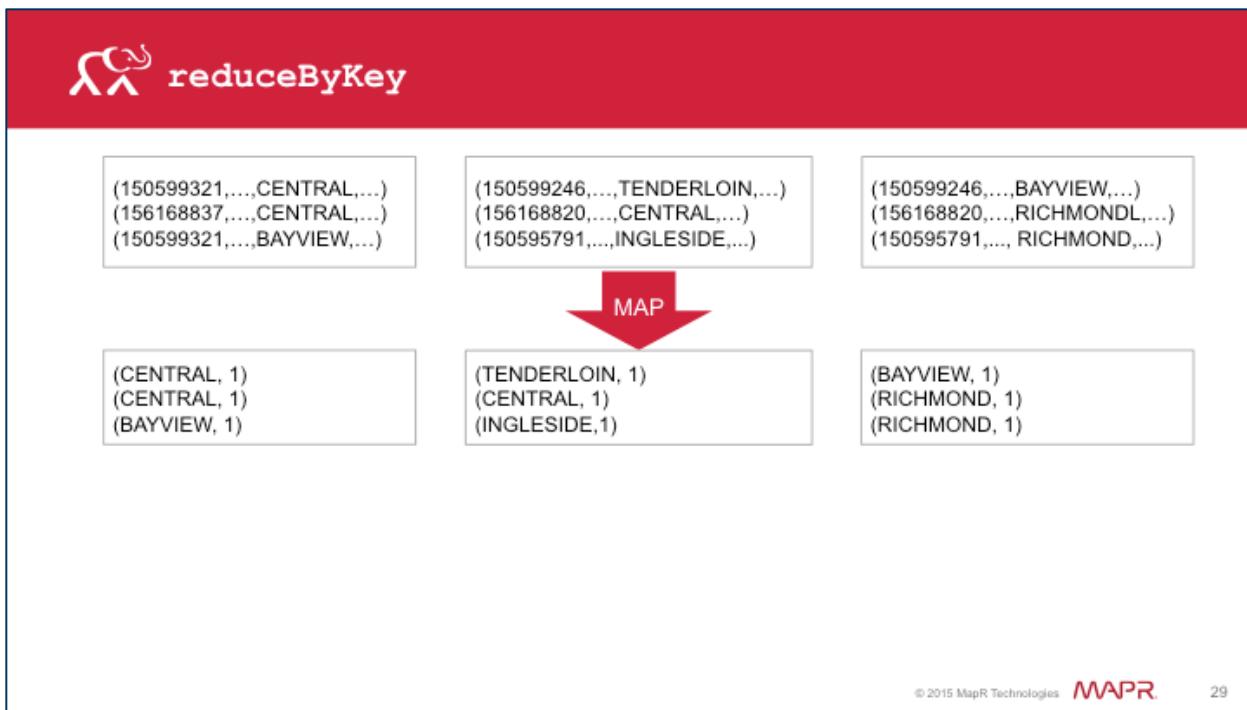




Notes on Slide 28:

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.



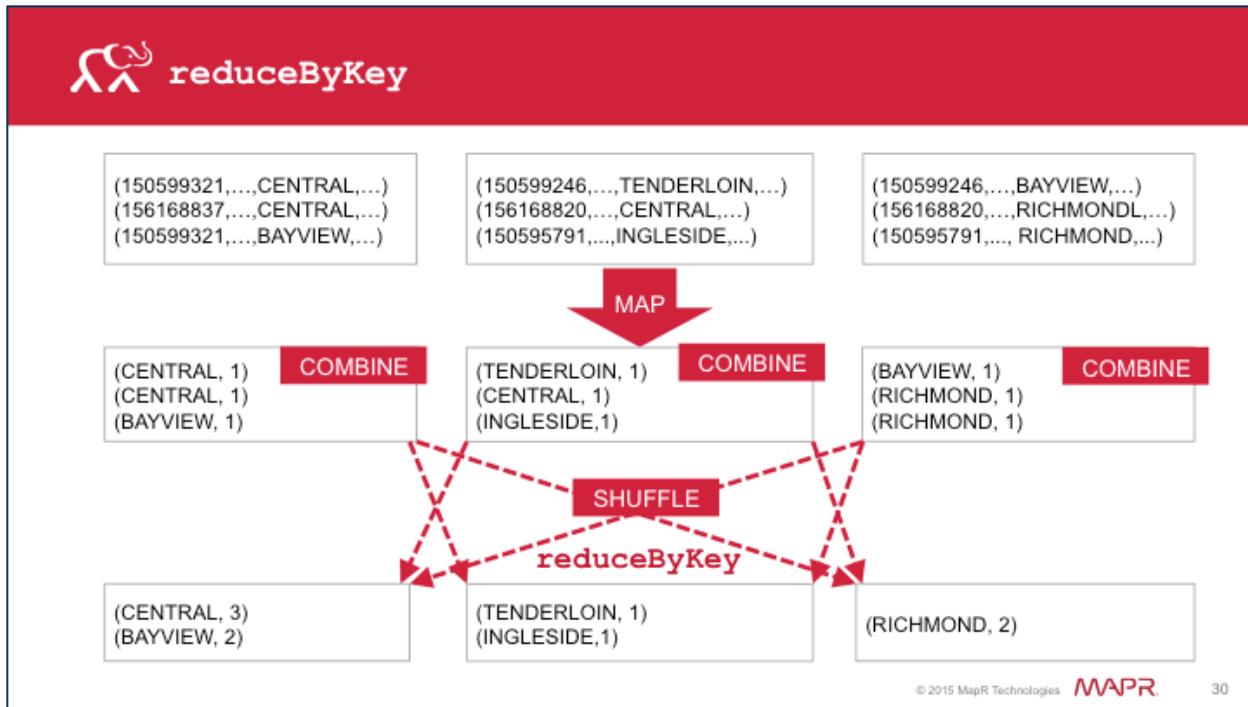


Notes on Slide 29:

`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 because `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



 Knowledge Check

groupByKey is less efficient than reduceByKey on large datasets because:

- A. groupByKey will group all values with the same key on one machine
- B. With groupByKey, if a single key has more key-value pairs than can fit in memory, an out of memory exception occurs
- C. reduceByKey combines locally first and then reduces after the shuffle
- D. All of the above

© 2015 MapR Technologies 

31

Notes on Slide 31:

Answer: 4



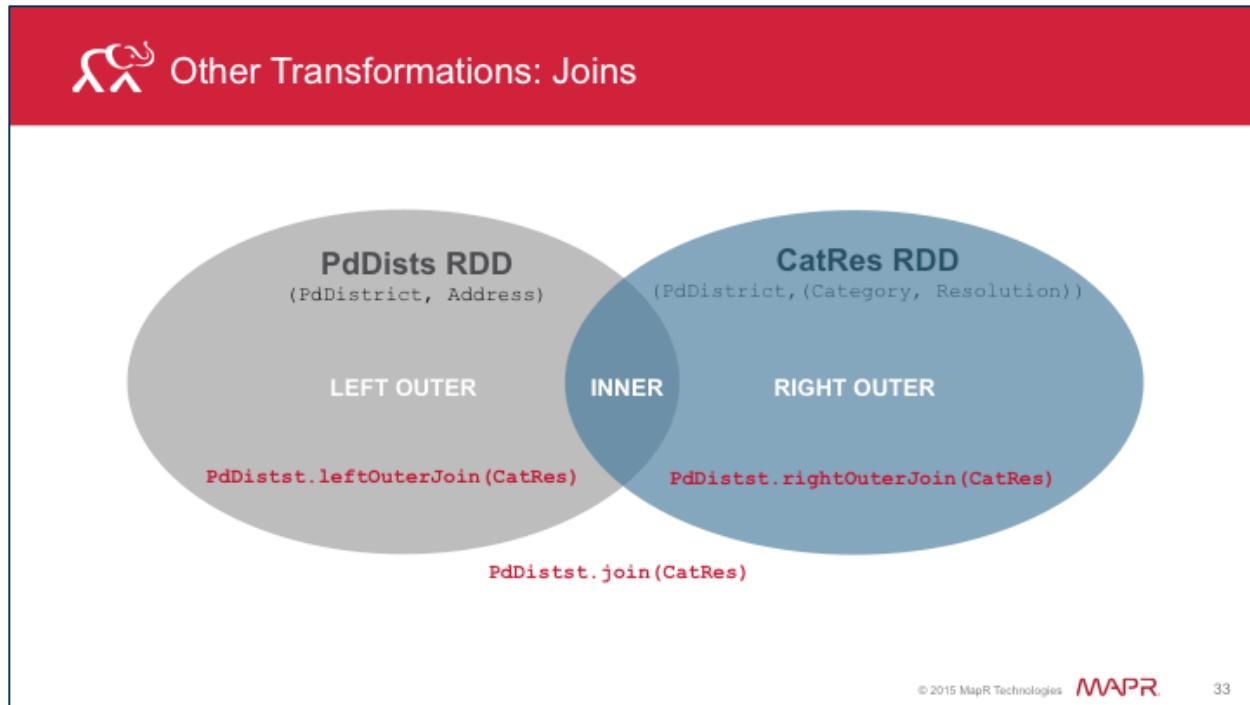


Lab 4.2.1: Create & Explore Pair RDD

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.



 Example: Joins

```
PdDists → Array((PdDistrict,Address))
CatRes → Array((PdDistrict,(Category,Resolution))
```

```
PdDists.join(CatRes)
Array((INGLESIDE,(DELTA_ST/RAYMOND_AV,(ASSAULT,ARREST/
BOOKED)))...)
```

```
PdDists.leftOuterJoin(CatRes)
Array((INGLESIDE,(DELTA_ST/RAYMOND_AV,Some ((ASSAULT,ARREST/
BOOKED))),...))
```

```
PdDists.rightOuterJoin(CatRes)
Array((INGLESIDE,(Some(DELTA_ST/RAYMOND_AV),(ASSAULT,ARREST/
BOOKED))),...))
```

© 2015 MapR Technologies 

34

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.



 Example: Joins

```
PdDists → Array((PdDistrict,Address))
IncCatRes → Array((IncidntNum, (Category,Descript,Resolution)))
```

```
PdDists.join(IncCatRes)
empty collection
```

```
PdDists.leftOuterJoin(IncCatRes)
Array((TENDERLOIN, (LEAVENWORTH_ST/TURK_ST,None)),...)
```

```
PdDists.rightOuterJoin(IncCatRes)
Array((130959836, (None, (VANDALISM,MALICIOUS_MISCHIEF/
BREAKING_WINDOWS,NONE))),...)
```

© 2015 MapR Technologies  35

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).



 Discussion: Joining Pair RDD

Can you think of examples in your organization data where you would use “joins”?





Actions on Pair RDDs

All traditional actions on RDD available to Pair RDD

- **countByKey ()**
 - Count the number of elements for each key
- **collectAsMap ()**
 - Collect the result as a map to provide easy lookup
- **lookup (key)**
 - Return all values associated with the provided key

© 2015 MapR Technologies

37

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.





Actions on Pair RDDs: Example

Caution:
Use this action
only if dataset is
small enough to
fit in memory



countByKey

```
val num_inc_dist=sfpd  
.map(incident=>(incident(PdDistrict),1))  
.countByKey()
```

```
res10:scala.collection.Map[String,Long] = Map(SOUTHERN -> 73308,  
INGLESIDE -> 33159, TENDERLOIN -> 30174, MISSION -> 50164,  
TARAVAL -> 27470, RICHMOND -> 21221, NORTHERN -> 46877, PARK ->  
23377, CENTRAL -> 41914, BAYVIEW -> 36111)
```

© 2015 MapR Technologies 38

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.





Lab 4.2.2: Join Pair RDD



Notes on Slide 39:

In this activity you will join Pair RDD.





Learning Goals

1. Create pair RDD
2. Apply transformations & actions to pair RDD
3. Control partitioning across nodes

© 2015 MapR Technologies **MAPR** 40

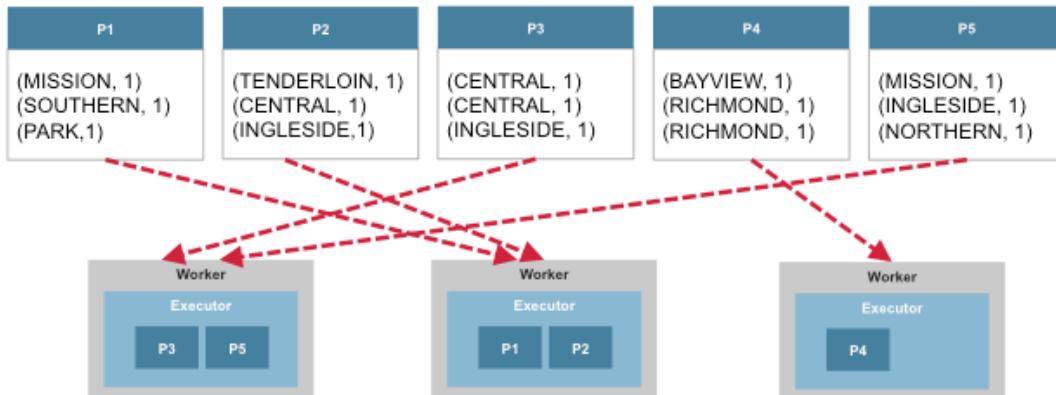
Notes on Slide 40:

In this section, we will look at the partitioning of RDDs and how to control the partitioning of RDDs across nodes.



 Why Partition

More partitions → More parallelism



© 2015 MapR Technologies 

41

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.



 RDD Partitions**The data within an RDD is split into several partitions.**

- System groups elements based on a function of each key
 - Set of keys appear together on the same node
- Each machine in cluster contains one or more partitions
 - Number of partitions is at least as large as number of cores in the cluster

© 2015 MapR Technologies 42Notes 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





Types of Partitioning

Two kinds of partitioning:

- Hash partitioning
- Range partitioning

© 2015 MapR Technologies **MAPR** 43

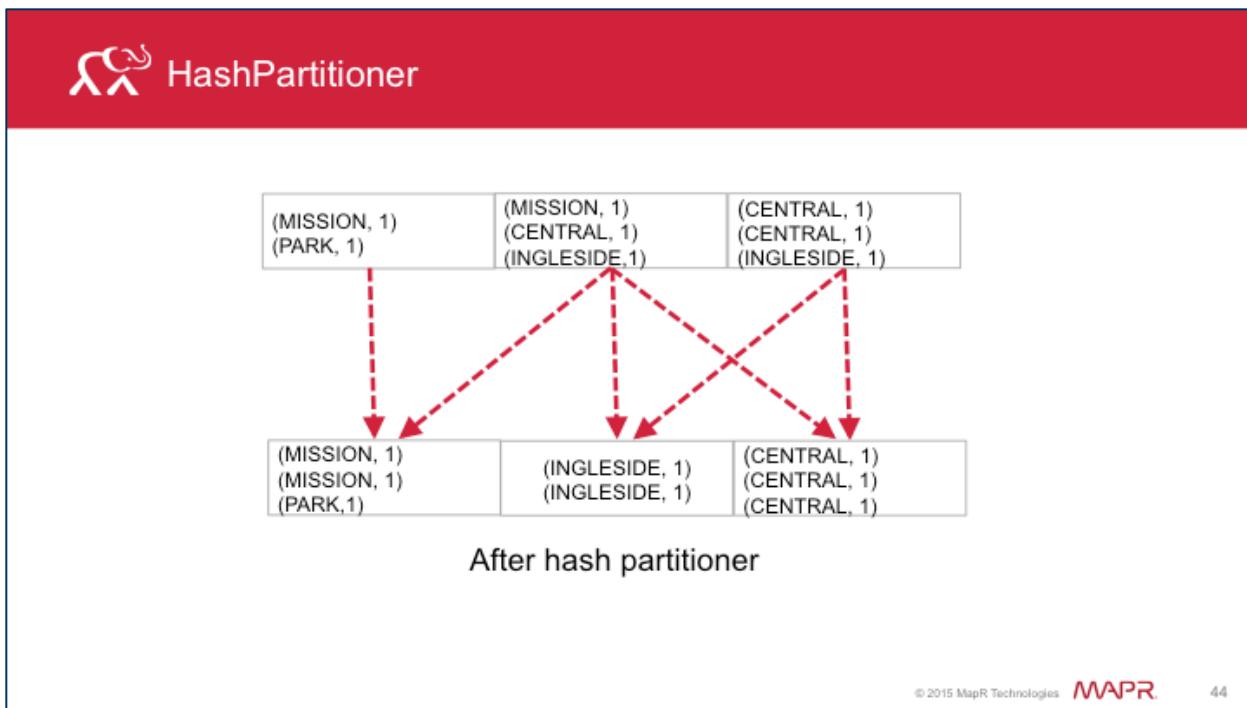
Notes on Slide 43:

There are two kinds of partitioning available in Spark:

1. Hash partitioning and
2. Range partitioning

Customizing partitioning is only possible on Pair RDDs.





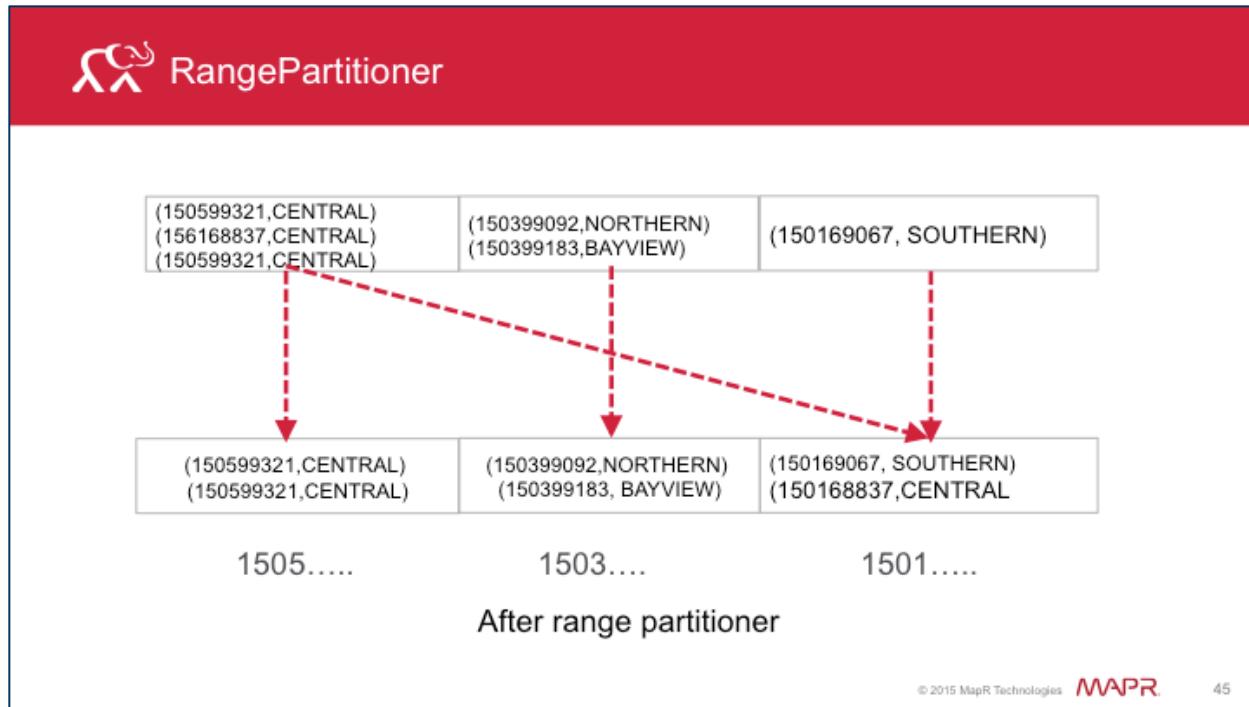
© 2015 MapR Technologies

44

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.



 Specifying Partitions**There are two ways to specify partitioning**

- `partitionBy` with `partitioner` explicitly specified
- Specify partition in transformation

© 2015 MapR Technologies  46

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





Using partitionBy: RangePartitioner

partitionBy is transformation → returns RDD with specific partitioning

To create a RangePartitioner:

1. Specify the desired number of partitions.
2. Provide a Pair RDD with ordered keys.

```
val pair1 = sfpd.map(x=>(x(PdDistrict),  
  (x(Resolution),x(Category))))  
val rpart1 = new RangePartitioner(4, pair1)  
val partrdd1 = pair1.partitionBy(rpart1).persist()
```

© 2015 MapR Technologies

47

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.





Using partitionBy: HashPartitioner

To create a HashPartitioner:

Specify the desired number of partitions.

Note:

Result of partitionBy should be persisted to prevent the partitioning from being applied each time the partitioned RDD is used.

```
val hpart1 = new HashPartitioner(100)
val partrdd2 = pair1.partitionBy(hpart1).persist()
```

© 2015 MapR Technologies 

48

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.





Specify Partitions in Transformations

- **Some operations accept additional argument**
 - numPartitions or type or partitioner
 - Examples: reduceByKey; aggregateByKey
- **Some operations automatically result in RDD with known partitioner**
 - sortByKey → RangePartitioner
 - groupByKey → HashPartitioner

© 2015 MapR Technologies 

49

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.





Change Partitions

Note:
Repartitioning your
data can be fairly
expensive

To change partitioning outside of aggregations and grouping operations:

- `repartition()`
Shuffles data across network to create new set of partitions
- `coalesce()`
Decreases the number of partitions

© 2015 MapR Technologies 

50

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.



Change Partitions: `coalesce()`

Initial RDD (5 partitions):

(MISSION, 1)	(TENDERLOIN, 1)	(CENTRAL, 1)	(BAYVIEW, 1)	(MISSION, 1)
(SOUTHERN, 1)	(CENTRAL, 1)	(CENTRAL, 1)	(RICHMOND, 1)	(INGLESIDE, 1)
(PARK,1)	(INGLESIDE,1)	(INGLESIDE,1)	(RICHMOND, 1)	(NORTHERN, 1)

Code:

```
val dists=sfpd.map(x=>(x(PdDistrict), 1)).reduceByKey(_+_ ,5)
```

Intermediate RDD (3 partitions):

(MISSION, 2)	(TENDERLOIN, 1)		(BAYVIEW, 1)	(NORTHERN, 1)
(SOUTHERN, 1)	(CENTRAL, 3)		(RICHMOND, 2)	
(PARK,1)	(INGLESIDE,3)			

Code:

```
dists.partitions.size  
dists.coalesce(3)
```

Final RDD (3 partitions):

(TENDERLOIN, 1)	(MISSION, 2)	(BAYVIEW,)
(CENTRAL, 3)	(SOUTHERN, 1)	(RICHMOND, 2)
(INGLESIDE,3)	(PARK,1)	(NORTHERN, 1)

© 2015 MapR Technologies  51

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.





Determine the Partitioner

Use `partitioner()` method to determine the RDD partitioning

```
val pairl = sfpd.map(x=>(x(PdDistrict), (x(Resolution), x(Category))))  
val rpart1 = new RangePartitioner(4, pairl)  
val partrdd1 = pairl.partitionBy(rpart1).persist()  
partrdd1.partition  
  
res3: Option[org.apache.spark.Partitioner] =  
Some(org.apache.spark.RangePartitioner@8f60c3c6)
```

© 2015 MapR Technologies  52

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.partition`, it returns the type of partitioning – in this case `RangePartitioner`.





Operations that Benefit from Partitioning

- cogroup()
- groupWith()
- join()
- leftOuterJoin()
- rightOuterJoin()
- groupByKey()
- reduceByKey()
- combineByKey()
- lookup()

© 2015 MapR Technologies 

53

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.



 Operations that Affect Partitioning

- cogroup()
- groupByKey()
- groupWith()
- reduceByKey()
- join()
- combineByKey()
- leftOuterJoin()
- partitionBy()
- rightOuterJoin()
- sort()

© 2015 MapR Technologies 

54

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.





Best Practices

- Too few partitions → idle resources
- Too many partitions → too much overhead
- Use numPartitions parameter in operations to specify partitions OR
- Use `repartition()` to increase or decrease # of partitions
- To decrease number of partitions → `coalesce()`
 - More efficient than `repartition()`
 - No shuffle

© 2015 MapR Technologies 55

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.





Knowledge Check

How can you create an RDD with specific partitioning?

1. `partitionBy()`
2. `rdd.partitioner = Hash`
3. Specify partition in transformation

Notes on Slide 56:

Answer: 1, 3





Lab 4.3: Explore Partitioning

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.



 Next Steps



Lesson 5

Work with Spark
DataFrames

© 2015 MapR Technologies 

58

Notes on Slide 58:

Congratulations! You have completed Lesson 4 of this course.





DEV 361: Build & Monitor Apache Spark Applications

Lesson 5: Work with DataFrames

© 2015 MapR Technologies  MAPR®

1

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





Learning Goals

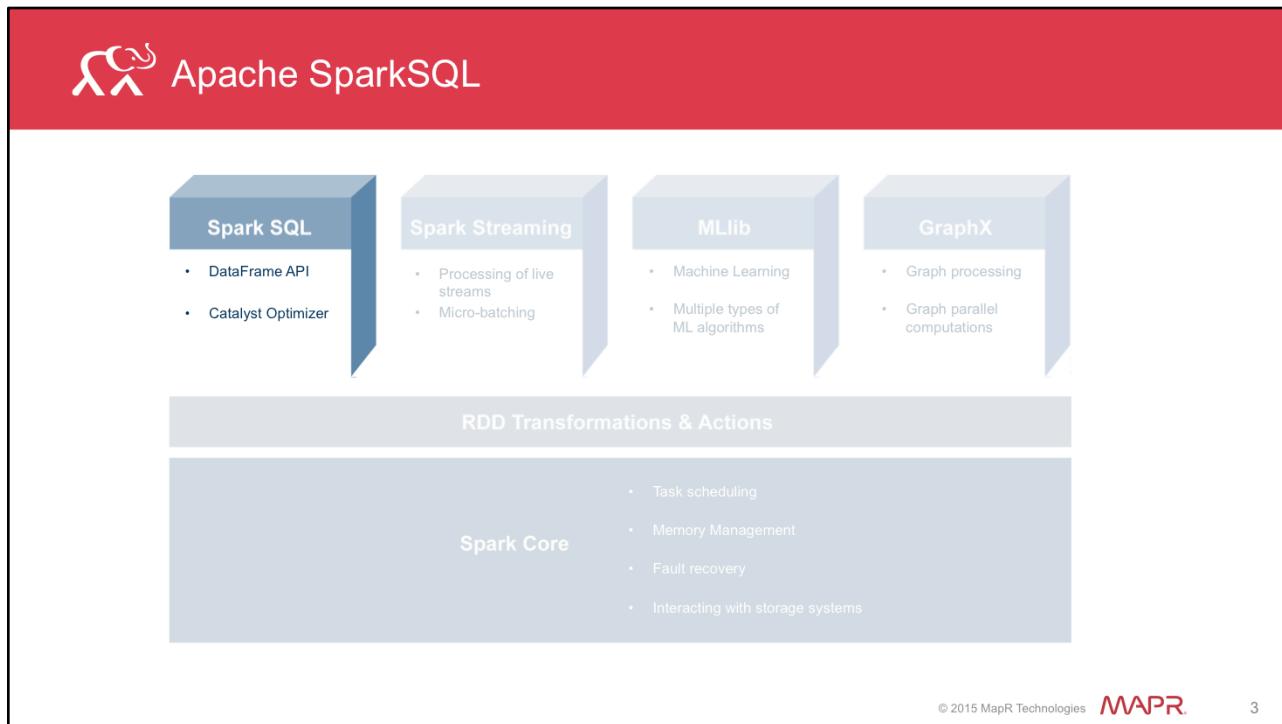
- ▶ 1. Create Apache Spark DataFrames
- 2. Explore Data in DataFrames
- 3. Create User Defined Functions
- 4. Repartition DataFrames

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.





- Programming abstraction in SparkSQL
- Distributed collection of data organized into named columns
- Supports wide array of data formats & storage systems
- Works in Scala, Python, Java & R

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.





Spark DataFrames vs Spark RDD

Spark DataFrames	Spark RDD
Collection with schema	Opaque collection of objects with no information of underlying data type
Can query data using SQL	Cannot query using SQL

5

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.



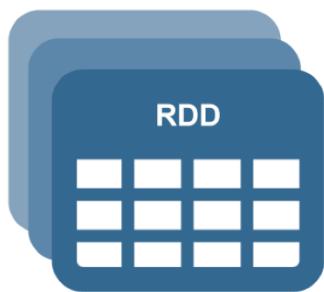


What is the starting point for creating DataFrames?

(HINT: Think back to DEV 360: Lesson 2. It is the entry to Spark SQL)

SQLContext – you must create a new SQLContext.



 Creating DataFrames**Data Sources**

- Parquet
- JSON
- Hive tables
- Relational Databases

© 2015 MapR Technologies  MAPR®

7

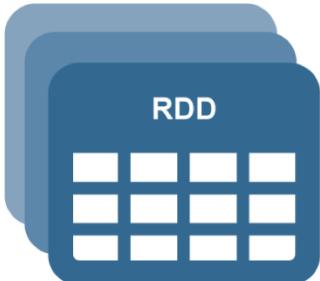
There are two ways to create Spark DataFrames. One way is to create it from existing RDDs and the other from data sources.

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

© 2015 MapR Technologies

8

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.





Infer Schema by Reflection: Case Class

- **Defines table schema**
 - Names of arguments to case class read using reflection
 - Names become name of column
- **Can be**
 - Nested
 - Contain complex data (Sequences or Arrays)

© 2015 MapR Technologies  MAPR®

9

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.





Infer Schema by Reflection

1. Import necessary classes
2. Create RDD
3. Define case class
4. Convert RDD into RDD of case objects
5. Implicitly convert resulting RDD of case objects into DataFrame
 - Apply DataFrame operations & functions to DataFrame
6. Register DataFrame as table
 - Run SQL queries on table

Note:
Create
SQLContext
before importing
classes

© 2015 MapR Technologies MAPR

10

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")
```





Example: Infer Schema by Reflection

1. Import necessary classes

```
import org.apache.spark.sql._  
val sqlContext = new org.apache.spark.sql.SQLContext(sc)  
import sqlContext.implicits._
```

2. Create RDD

```
val sfpdRDD = sc.textFile("/path/to/file/  
sfpd.csv").map(inc=>inc.split(",") )
```

© 2015 MapR Technologies  MAPR®

11

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.





Example: Infer Schema by Reflection:

3. Define case class

```
case class Incidents(incidentnum:String,  
category:String, description:String, dayofweek:String,  
date:String, time:String, pdistrict:String,  
resolution:String, address:String, X:Float, Y:Float,  
pdid:String)
```

4. Convert RDD into RDD of case objects

```
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)))
```

© 2015 MapR Technologies  MAPR®

12

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.





Example: Infer Schema by Reflection

5. Implicitly convert resulting RDD of case objects into DataFrame

```
val sfpdDF = sfpdCase.toDF()
```

6. Register DataFrame as table

```
sfpdDF.registerTempTable("sfpd")
```

© 2015 MapR Technologies  MAPR

13

5. We then implicitly convert sfpdCase into a DataFrame 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.





Construct Schema Programmatically

1. Create a Row RDD from original RDD
2. Create schema separately using:
 - StructType → table
 - StructField → field
3. Create DataFrame by applying schema to Row RDD

Note:

Used when:

- Case class cannot be defined ahead of time
- More than 22 fields as case classes (Scala)

© 2015 MapR Technologies 

14

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.





Example: Construct Schema Programmatically

Sample Data

```
150599321 Thursday 7/9/15 23:45 CENTRAL
156168837 Thursday 7/9/15 23:45 CENTRAL
150599321 Thursday 7/9/15 23:45 CENTRAL
```

Note:
Sample data is
from the file
test.txt

Import Classes

```
import org.apache.spark.sql._
import org.apache.spark.sql.types._
...
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.implicits._
```

© 2015 MapR Technologies MAPR

15

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")
```





Example: Construct Schema Programmatically

1. Create Row RDD from input RDD

```
val rowRDD = sc.textFile("/user/user01/data/test.txt")  
    .map(x=>x.split(" "))  
    .map(p=>Row(p(0),p(2),p(4)))
```

© 2015 MapR Technologies  MAPR

16

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.





Example: Construct Schema Programmatically

2. Create schema separately

```
val testsch = StructType(Array(StructField("IncNum",  
StringType,true), StructField("Date",StringType,true),  
StructField("District",StringType,true)))
```

© 2015 MapR Technologies  MAPR

17

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).





Example: Construct Schema Programmatically

3. Create DataFrame

```
val testDF = sqlContext.createDataFrame(rowRDD, schema)
```

Register the DataFrame as a table

```
testDF.registerTempTable("test")
```

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

© 2015 MapR Technologies  MAPR®

18

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.





Discussion

Can you think of situations in your organization for each method?

- Infer schema by reflection
- Programmatic

© 2015 MapR Technologies  MAPR

19

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

Answer: A





Create DataFrames from Data Sources

Operate on variety of data sources through DataFrame interface

- Parquet
- JSON
- Hive tables
- Relational Databases (in Spark 1.4)

© 2015 MapR Technologies  MAPR

21

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.





Generic “load” Method

- **Generic load method**

```
sqlContext.load("/path/to/sfpd.parquet")
```

Note:

Default data source assumed to be parquet unless configured otherwise



- **Specify format manually**

```
sqlContext.load("/path/to/sfpdjson", "json")
```

© 2015 MapR Technologies 

22

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)





Methods to Load Specific Data Sources

DataFrame from database table:

```
sqlContext.jdbc
```

DataFrame from JSON file:

```
sqlContext.jsonFile
```

DataFrame from RDD containing JSON objects:

```
sqlContext.jsonRDD
```

DataFrame from parquet file:

```
sqlContext.parquetFile
```

© 2015 MapR Technologies  MAPR®

23

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.





Methods to Load Data Sources: Spark 1.4 Onwards

```
sqlContext.read.load("path/to/file/filename.parquet")
```

```
sqlContext.read.format("json").load("/path/to/file/filename.json")
```

© 2015 MapR Technologies  MAPR

24

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

Answer: A,B,D





Lab 5.1: Create DataFrame Using Reflection



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





Learning Goals

1. Create Apache Spark DataFrames
- ▶ **2. Explore Data in DataFrames**
3. Create User Defined Functions
4. Repartition DataFrames

© 2015 MapR Technologies  MAPR

27

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





Exploring the Data

- What are the top five addresses with most incidents?
- What are the top five districts with most incidents?
- What are the top 10 resolutions?
- What are the top 10 categories of incidents?

© 2015 MapR Technologies  MAPR®

28

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.





DataFrame Operations

- DataFrame Actions
- DataFrame Functions
- Language Integrated Queries

© 2015 MapR Technologies  MAPR®

29

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.



 DataFrame Actions

Action	Description
<code>collect()</code>	Returns array containing all rows in DataFrame
<code>count()</code>	Returns number of rows in DataFrame
<code>describe(cols:String*)</code>	Computes statistics for numeric columns (count, mean, stddev, min and max)

30

This table lists DataFrame actions.





DataFrame Functions

Function	Description
<code>cache()</code>	Cache this DataFrame
<code>columns</code>	Returns an array of all column names
<code>printSchema()</code>	Prints schema to console in tree format

31

This table lists DataFrame functions.





Language Integrated Queries

L I Query	Description
<code>agg(expr, exprs)</code>	Aggregates on entire DataFrame
<code>distinct</code>	Returns new DataFrame with unique rows
<code>except(other)</code>	Returns new DataFrame with rows from this DataFrame not in other DataFrame
<code>filter(expr)</code>	Filter based on the SQL expression or condition

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.





Top Five Addresses with Most Incidents

1. val incByAdd=sfpdDF.groupBy("address")

2. val numAdd=incByAdd.count

3. val numAddDesc=numAdd.sort(\$"count".desc)

4. val top5Add=numAddDesc.show(5)

© 2015 MapR Technologies  MAPR

33

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.





Top Five Addresses with Most Incidents

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

address	count
800_Block_of_BRYA...	10852
800_Block_of_MARK...	3671
1000_Block_of_POT...	2027
2000_Block_of_MIS...	1585
16TH_ST/MISSION_ST	1512

© 2015 MapR Technologies 

34

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")





Top Five Addresses with Most Incidents: SQL

```
val top5Addresses = sqlContext.sql("SELECT address,  
count(incidentnum) AS inccount FROM sfpd GROUP BY address ORDER  
BY inccount DESC LIMIT 5") .show
```

address	inccount
800_Block_of_BRYA...	10852
800_Block_of_MARK...	3671
1000_Block_of_POT...	2027
2000_Block_of_MIS...	1585
16TH_ST/MISSION_ST	1512

© 2015 MapR Technologies

35

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`”.





Exploring the Data

- ✓ What are the top five addresses with most incidents?
 - What are the top five districts with most incidents?
 - What are the top 10 resolutions?
 - What are the top 10 categories of incidents?

© 2015 MapR Technologies  MAPR

36

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.



Output Operations: **save()**

To save contents of top5Addresses DataFrame:

```
top5Addresses.toJSON.saveAsTextFile("/user/user01/test")
```

```
{"address":"800_Block_of_BRYANT_ST","inccount":10852}  
 {"address":"800_Block_of_MARKET_ST","inccount":3671}  
 {"address":"1000_Block_of_POTRERO_AV","inccount":2027}  
 {"address":"2000_Block_of_MISSION_ST","inccount":1585}  
 {"address":"16TH_ST/MISSION_ST","inccount":1512}
```

© 2015 MapR Technologies MAPR

38

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>





Output Operations – Spark 1.4 Onwards

To save contents of top5Addresses DataFrame:

```
top5Addresses.write.format("json").mode("overw  
rite").save("/user/user01/test")
```

© 2015 MapR Technologies  MAPR

39

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)`

© 2015 MapR Technologies  MAPR

40

Answer: A, C

A → using SQL

C → using DataFrame operations





Lab 5.2: Explore Data in DataFrames



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



 Learning Goals

1. Create Apache Spark DataFrames
2. Explore data in DataFrames
- ▶ **3. Create User Defined Functions**
4. Repartition DataFrames

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





User Defined Functions (UDF)

- UDFs allow developers to define custom functions
- In Spark, can define UDF inline
- No complicated registration or packaging process
- Two types of UDF:
 - To use with Scala DSL (DataFrame operations)
 - To use with SQL

© 2015 MapR Technologies  MAPR®

43

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.





User Defined Functions (Scala DSL)

- Inline creation
 - Use `udf()`
- Function can be used with DF operations

```
val func1 = udf((arguments) => {function definition})
```

© 2015 MapR Technologies  MAPR

44

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





User Defined Functions in SQL Query

```
def funcname  
sqlContext.udf.register("funcname", funcname _)
```

partially applied
function in Scala

Inline registration and creation:

```
sqlContext.udf.register("funcname", func def)
```

© 2015 MapR Technologies 

45

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.





Example: Want to Find Incidents by Year

- Date in the format: “dd/mm/yy”
- Need to extract string after last slash
- Can then compute incidents by year

© 2015 MapR Technologies  MAPR

46

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.





Example: Want to Find Incidents by Year

- **Defining UDF**

```
val getStr = udf((s:String)=>{  
    val lastS = s.substring(s.lastIndexOf('/')+1)  
    lastS  
})
```

Note:

This is
Scala DSL



- **Use UDF in DataFrame Operations**

```
val yy = sfpdDF.groupBy(getStr(sfpdDF("date")))  
    .count  
    .show
```

© 2015 MapR Technologies 

47

In this example, we are defining a function.





Example: Want to Find Incidents by Year

Note:
This is
Scala DSL

```
scalaUDF(date) count
13          152830
14          150185
15          80760
```

© 2015 MapR Technologies  MAPR

48

This is the number of incidents by year.





Example: Want to Find Incidents by Year

- **Define UDF**

```
def getStr(s:String) = {val  
strAfter=s.substring(s.lastIndexOf('/')+1)  
strafter}
```

Note:
This is SQL



- **Register UDF**

```
sqlContext.udf.register("getStr", getStr _)
```

© 2015 MapR Technologies MAPR

49

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 getStr and then registering it as a user defined function.





Example: Want to Find Incidents by Year

- **Define & register UDF**

```
sqlContext.udf.register("getStr", (s:String)=>{
    val strAfter=s.substring(s.lastIndexOf('/')+1)
    strAfter
})
```

Note:
This is SQL



- **Use in SQL statement**

```
val numIncByYear = sqlContext.sql("SELECT getStr(date),
    count(incidentnum) AS countbyyear
    FROM sfpd GROUP BY getStr(date)
    ORDER BY countbyyear DESC
    LIMIT 5")
```

© 2015 MapR Technologies MAPR

50

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





Example: Want to Find Incidents by Year

Note:
This is SQL

```
numIncByYear.foreach(println)
[13,152830]
[14,150185]
[15,80760]
```

© 2015 MapR Technologies  MAPR

51

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

Answers: 2





Lab 5.3: Create & Use User Defined Functions



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



 Learning Goals

1. Create Apache Spark DataFrames
2. Explore data in DataFrames
3. Create User Defined Functions
- ▶ **4. Repartition DataFrames**

© 2015 MapR Technologies  MAPR

54

In this section, we look at repartitioning DataFrames.



 Partition DataFrames

DataFrame with 4 Partitions

P1			P2			P3			P4		
Incnum (Str)	Category (Str)	PdDistrict (Str)									
150598981	ASSAULT	CENTRAL	150599183	ASSAULT	SOUTHERN	150597701	ASSAULT	MISSION	150597400	ROBBERY	TARAVAL
150599161	BURGLARY	PARK	150599246	ASSAULT	CENTRAL	150597701	ROBBERY	INGLESIDE	150596468	FRAUD	SOUTHERN
150599127	SUSPICIOUS	SOUTHERN	150599246	WARRANTS	CENTRAL	150597701	ASSAULT	SOUTHERN	150597234	BURGLARY	SOUTHERN
150603455	VANDALISM	NORTHERN	150599246	WARRANTS	CENTRAL	150597591	ROBBERY	SOUTHERN	150596468	FRAUD	TARAVAL

© 2015 MapR Technologies 

55

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





Partition DataFrames

- Sets number of partitions in DataFrame after shuffle:
`spark.sql.shuffle.partitions`
- Default value set to 200
- Can change parameter using:
`sqlContext.setConf(key, value)`

© 2015 MapR Technologies  MAPR

56

By default in Spark SQL, there is a parameter called `spark.sql.shuffle.partitions`, 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)`.





Why Repartition

- Internally SparkSQL partitions data for joins and aggregations
- If applying other RDD operations on result of DataFrame operations, can manually control partitioning

© 2015 MapR Technologies  MAPR

57

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.





repartition(numPartitions)

- To repartition, use
 - `df.repartition(numPartitions)`
- To determine current number of partitions:
 - `df.rdd.partitions.size`

© 2015 MapR Technologies  MAPR

58

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.





Best Practices

- Specifying the number of partitions:
- Want each partition to be 50 MB – 200 MB
- Small dataset → few partitions
- Large cluster with 100 nodes → at least 100 partitions
- Example:
 - 100 nodes with 10 slots in each executor
 - Want 1000 partitions to use all executor slots

© 2015 MapR Technologies  MAPR

59

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 partitions, 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





Next Steps



Lesson 6

Monitor Apache Spark Applications

© 2015 MapR Technologies 

61

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



Lesson 6: Monitor Apache Spark Applications



DEV 361: Build & Monitor Apache Spark Applications

Lesson 6: Monitor Apache Spark Applications

© 2015 MapR Technologies 1

Notes on Slide 1:

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



Learning Goals

- ▶ 1. Describe Components of Spark Execution Model
- 2. Use Spark Web UI to Monitor Spark Applications
- 3. Debug & Tune Spark Applications

© 2015 MapR Technologies **MAPR**2**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.





Spark Execution Model: Logical Plan

```
1. val inputRDD = sc.textFile("/.../sfpd.csv")  
2. val sfpdRDD = input.map(x=>x.split(", " ))  
3. val catRDD = sfpdRDD.map(x=>(x(Category),  
1)).reduceByKey((a,b)=>a+b)
```

© 2015 MapR Technologies 

3

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.

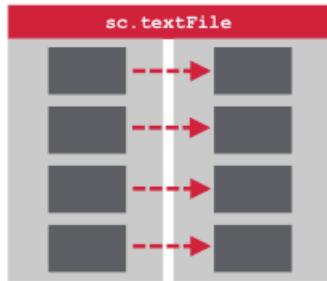
Consider the example from an earlier lesson where we load SFPD data from a csv file. We will use this as an example to walk through the components of the Spark execution model.





Spark Execution Model: Logical Plan

```
1. val inputRDD = sc.textFile("/.../sfpd.csv")
```



© 2015 MapR Technologies

4

Notes on Slide 4:

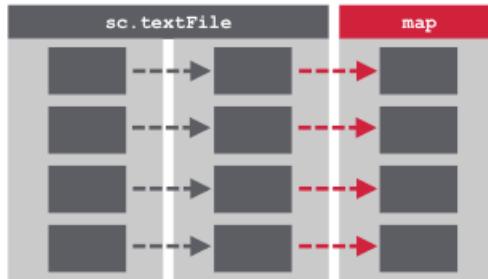
- The first line creates an RDD called inputRDD from the sfpd.csv file.





Spark Execution Model: Logical Plan

```
2. val sfpdRDD = input.map(x=>x.split(","))
```



© 2015 MapR Technologies 

5

Notes on Slide 5:

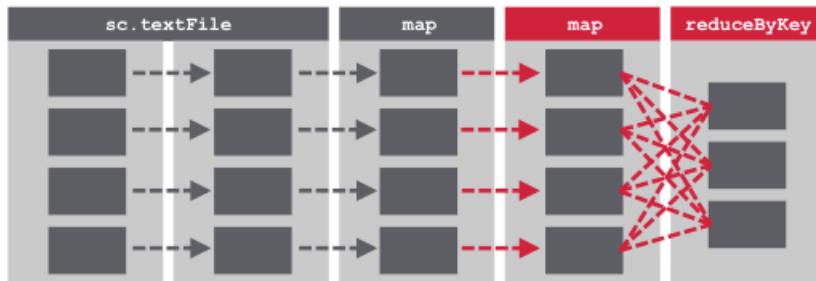
- The second line creates an RDD – sfpdRDD which splits the data in the input RDD based on the comma separator.





Spark Execution Model: Logical Plan

```
3. val catRDD = sfpdRDD.map(x=>(x(Category),
  1)).reduceByKey((a,b)=>a+b)
```



© 2015 MapR Technologies

6

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.





Spark Execution Model: Display Lineage

```
catRDD.toDebugString()
```

```
(2) ShuffledRDD[20] at reduceByKey at <console>:27 []
+- (2) MapPartitionsRDD[19] at map at <console>:27 []
  |  MapPartitionsRDD[16] at map at <console>:23 []
  |  /user/user01/data/sfpd.csv MapPartitionsRDD[15] at textFile at <console>:21 []
  |  /user/user01/data/sfpd.csv HadoopRDD[14] at textFile at <console>:21 []
```

© 2015 MapR Technologies 7

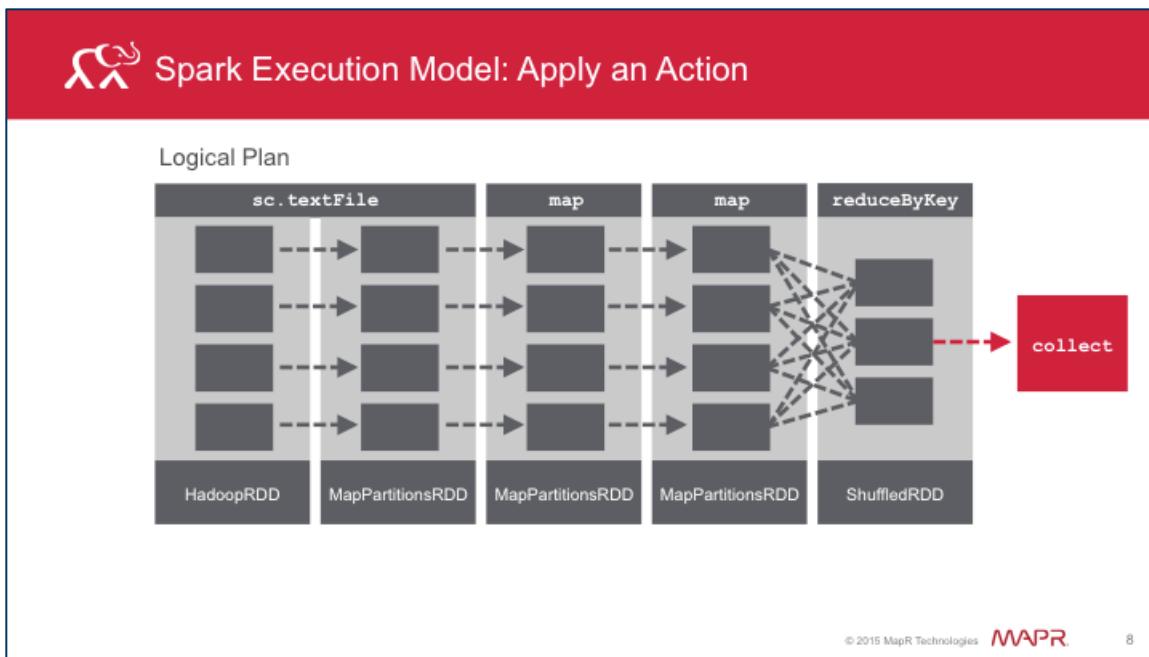
Notes on Slide 7:

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.





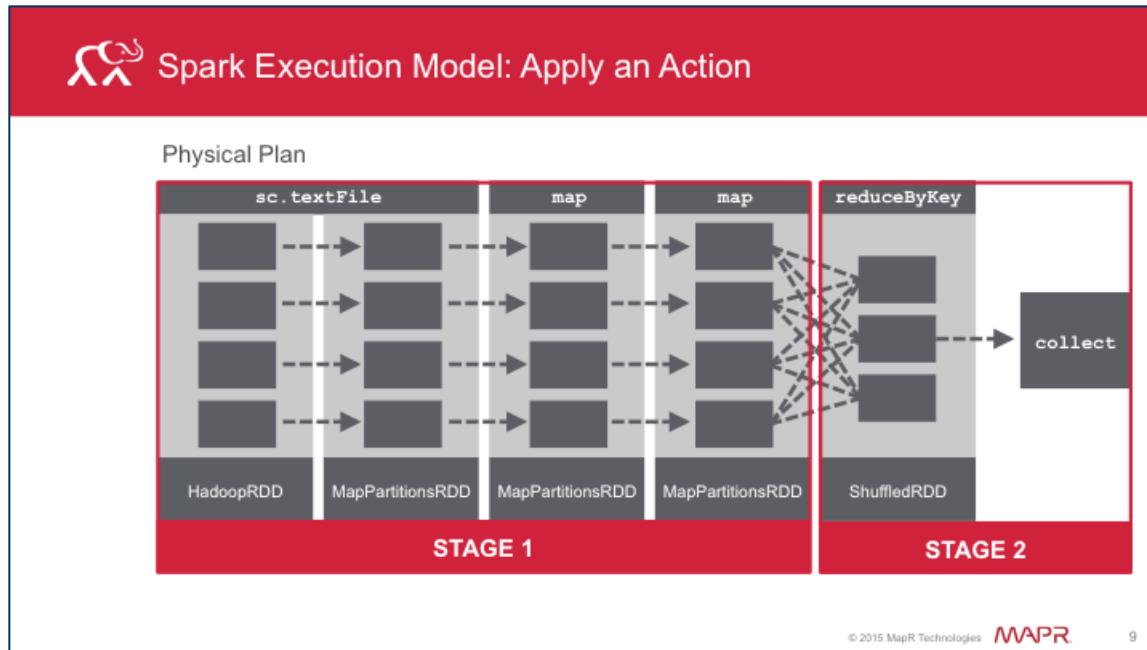
© 2015 MapR Technologies **MAPR**

8

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.





Situations When Lineage Truncated

1. Pipelining
2. RDD persisted in cluster memory or on disk
3. RDD materialized due to earlier shuffle

© 2015 MapR Technologies 

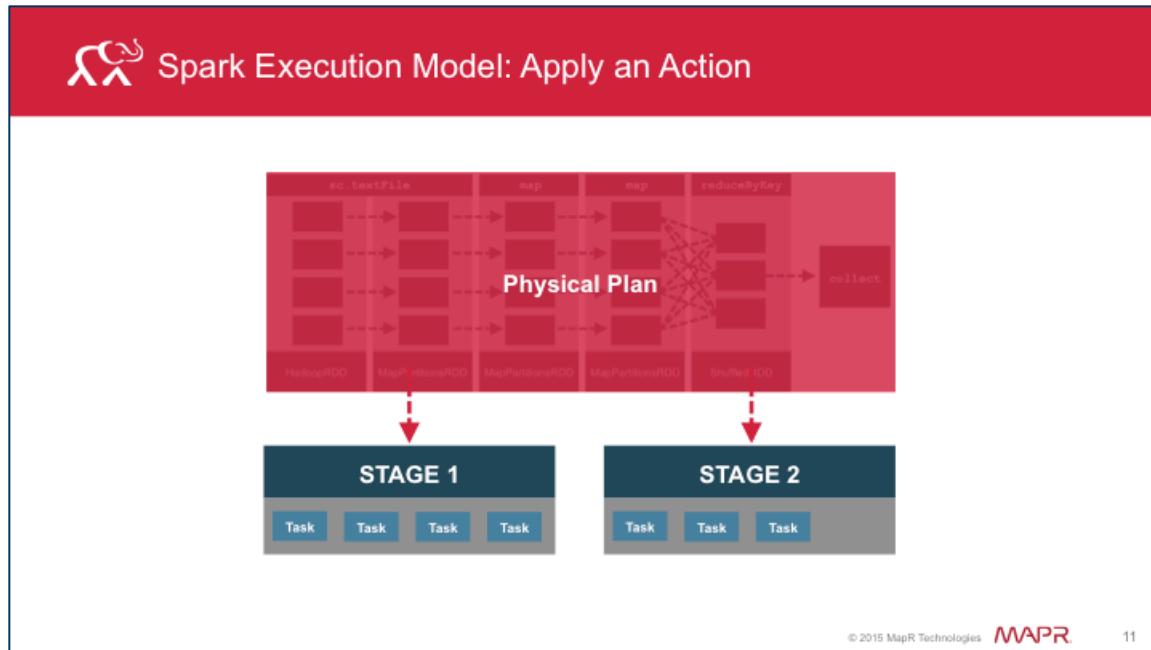
10

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.

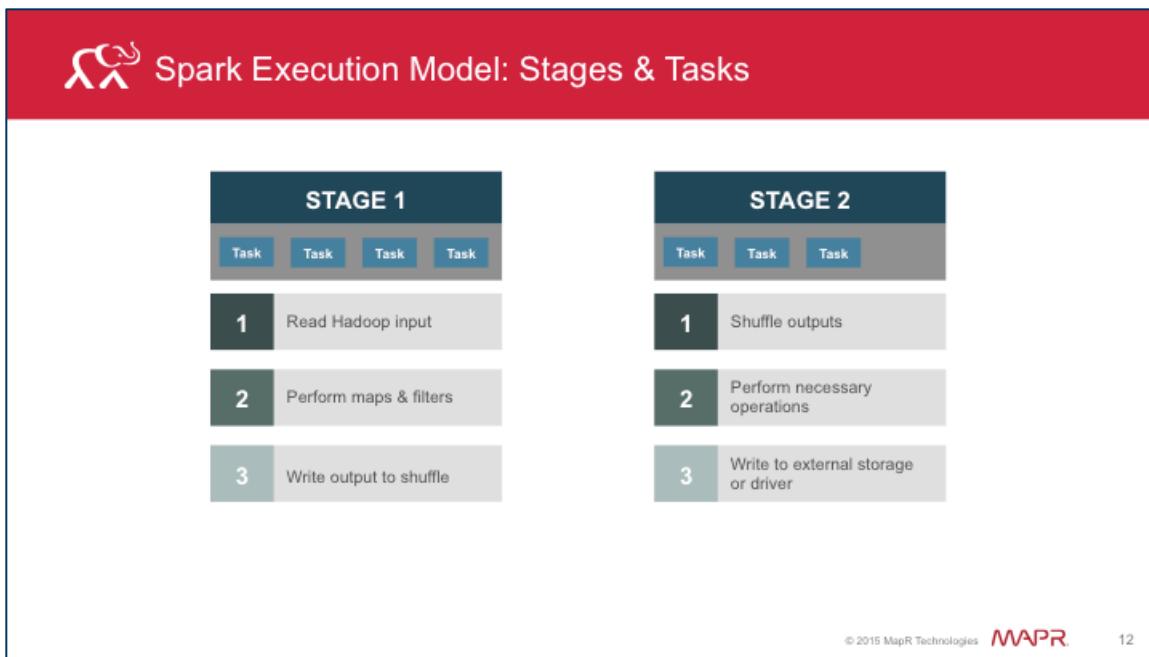




Notes on Slide 11:

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)





Spark Execution Model: Components of Spark Execution

Component	Description
Tasks	Unit of work within a stage corresponds to one RDD partition
Stages	Group of tasks which perform the same computation in parallel
Shuffle	Transfer of data between stages
Jobs	Work required to compute RDD; has one or more stages
Pipelining	Collapsing of RDDs into a single stage when RDD transformations can be computed without data movement
Directed Acyclic Graph (DAG)	Logical graph of RDD operations
Resilient Distributed Dataset (RDD)	Parallel dataset with partitions

© 2015 MapR Technologies

13

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.



Phases During Spark Execution

Note: In an application, a sequence may occur many times as new RDD created

PHASE 1	User code defines the DAG or RDDs
PHASE 2	Actions responsible for translating DAG into physical execution plan
PHASE 3	Tasks scheduled and executed on cluster

© 2015 MapR Technologies **MAPR** 14

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

© 2015 MapR Technologies **MAPR**

15

Notes on Slide 15:

Answer:A,C, D





Learning Goals

1. Describe Components of Spark Execution Model
- ▶ **2. Use Spark Web UI to Monitor Spark Applications**
3. Debug & Tune Spark Applications

© 2015 MapR Technologies **MAPR**

16

Notes on Slide 16:

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



What is Spark Web UI

Spark Jobs [?]

Total Duration: 1.8 min
Scheduling Mode: FIFO
Active Jobs: 2
Completed Jobs: 8

Active Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	print at <console>-39	2015/10/21 18:08:24	51 ms	0/2	0/4
0	start at <console>-33	2015/10/21 18:08:16	7 s	0/1	0/1

Completed Jobs (8)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8	print at <console>-39	2015/10/21 18:08:22	26 ms	1/1 (1 skipped)	1/1 (3 skipped)
7	print at <console>-39	2015/10/21 18:08:22	0.1 s	2/2	4/4
6	print at <console>-39	2015/10/21 18:08:21	23 ms	1/1 (1 skipped)	1/1 (1 skipped)
5	print at <console>-39	2015/10/21 18:08:21	44 ms	2/2	3/2
4	print at <console>-39	2015/10/21 18:08:21	19 ms	1/1 (1 skipped)	1/1 (1 skipped)
3	print at <console>-39	2015/10/21 18:08:21	47 ms	2/2	2/2
2	print at <console>-39	2015/10/21 18:08:21	26 ms	1/1 (1 skipped)	1/1 (1 skipped)

Note:
In YARN cluster mode, access Spark web UI through YARN ResourceManager

http://<driver-node>:4040

© 2015 MapR Technologies **MAPR** 17

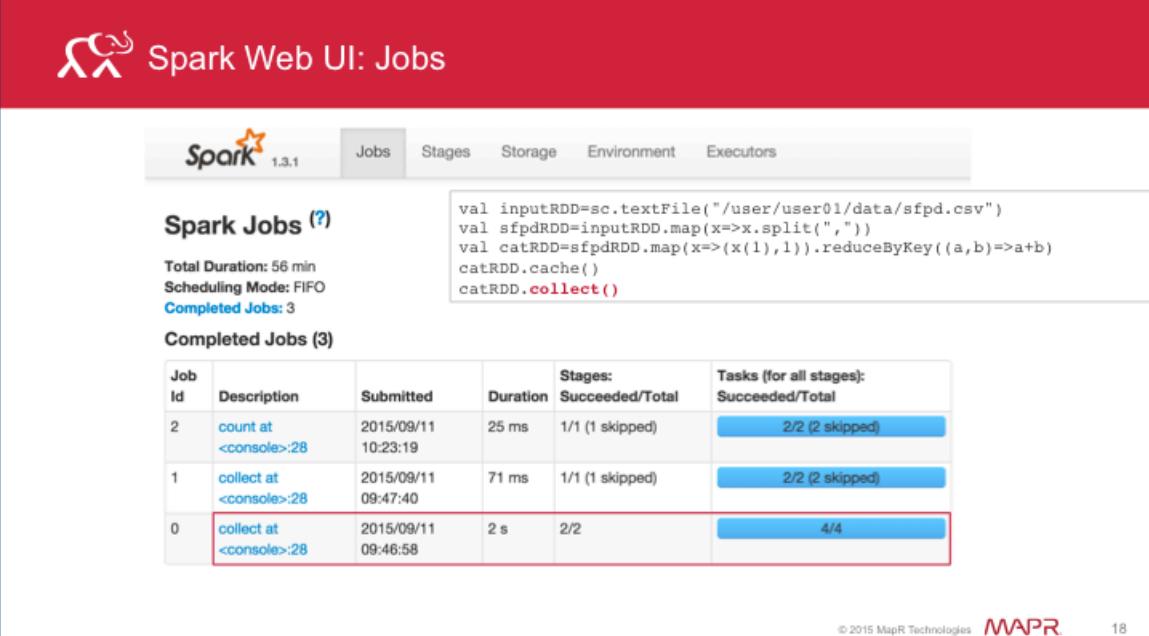
Notes on Slide 17:

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.





The screenshot shows the Spark Web UI 'Jobs' page. At the top, there's a navigation bar with tabs for 'Jobs', 'Stages', 'Storage', 'Environment', and 'Executors'. Below the navigation is a code snippet window containing Scala code for reading a CSV file, splitting it by commas, and performing a reduceByKey operation. This is followed by a section titled 'Completed Jobs (3)' which lists three completed jobs in a table.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

At the bottom right of the page, there's a copyright notice: © 2015 MapR Technologies and the MapR logo.

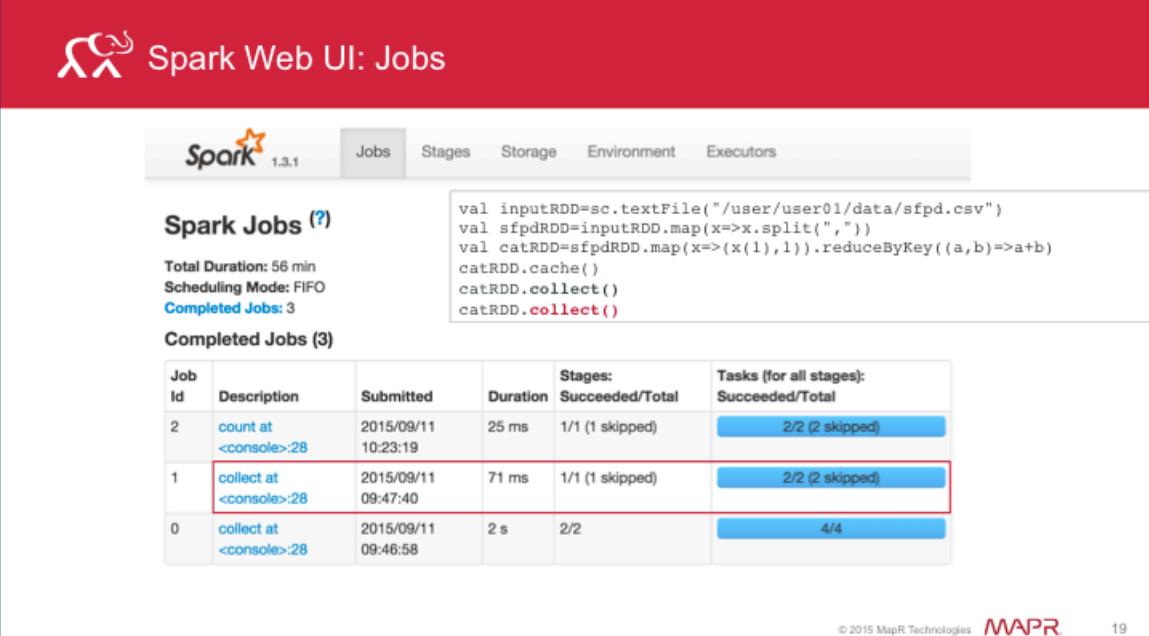
Notes on Slide 18:

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.





The screenshot shows the Spark Web UI interface. At the top, there's a red header bar with the Spark logo and the text "Spark Web UI: Jobs". Below the header, there's a navigation bar with tabs: "Jobs" (which is selected), "Stages", "Storage", "Environment", and "Executors".

Spark Jobs (?)

Total Duration: 56 min
Scheduling Mode: FIFO
Completed Jobs: 3

```
val inputRDD=sc.textFile("/user/user01/data/sfpd.csv")
val sfpdRDD=inputRDD.map(x=>x.split(","))
val catRDD=sfpdRDD.map(x=>(x(1),1)).reduceByKey((a,b)=>a+b)
catRDD.cache()
catRDD.collect()
catRDD.collect()
```

Completed Jobs (3)

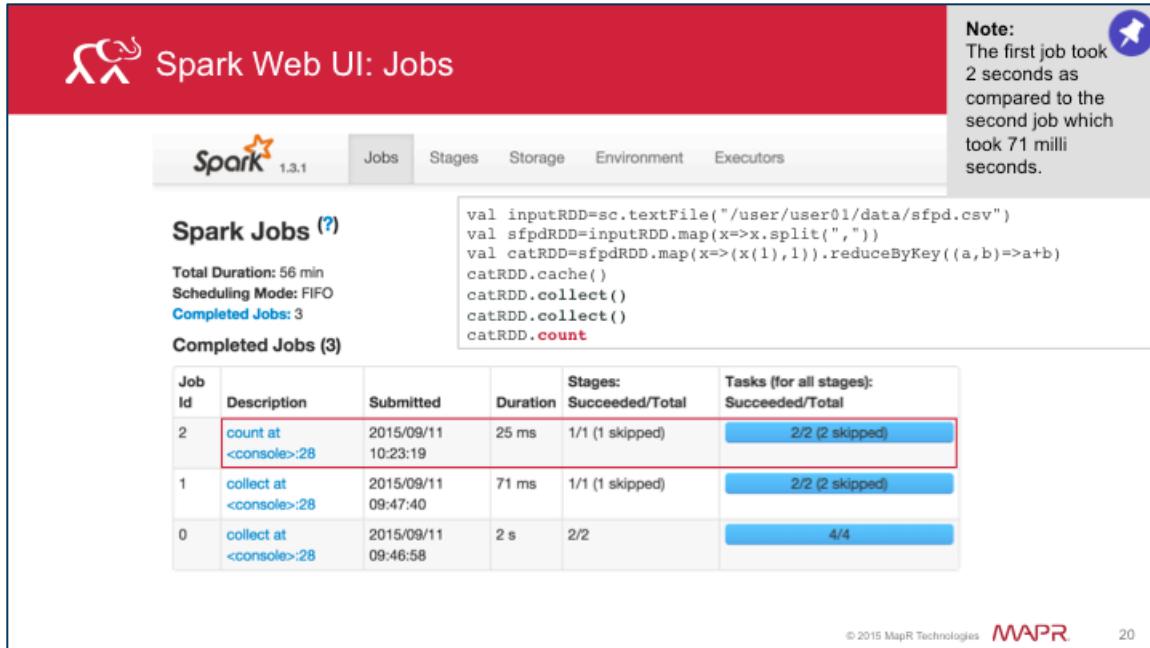
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

© 2015 MapR Technologies **MAPR** 19

Notes on Slide 19:

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





Spark Web UI: Jobs

Spark 1.3.1

Spark Jobs (?)

Total Duration: 56 min
Scheduling Mode: FIFO
Completed Jobs: 3

```
val inputRDD=sc.textFile("/user/user01/data/sfpd.csv")
val sfpdRDD=inputRDD.map(x=>x.split(","))
val catRDD=sfpdRDD.map(x=>(x(1),1)).reduceByKey((a,b)=>a+b)
catRDD.cache()
catRDD.collect()
catRDD.collect()
catRDD.count
```

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

Note: The first job took 2 seconds as compared to the second job which took 71 milliseconds.

© 2015 MapR Technologies **MapR** 20

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 milliseconds.



 Discussion: Skipped Stages

The second collect & count jobs only consist of ONE stage (skipped 1 stage) each. Why was one stage “skipped”?

Spark 1.3.1 Jobs Stages Storage Environment Executors

Spark Jobs (?)

Total Duration: 56 min
Scheduling Mode: FIFO
Completed Jobs: 3

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>-28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>-28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>-28	2015/09/11 09:46:58	2 s	2/2	4/4

© 2015 MapR Technologies  21

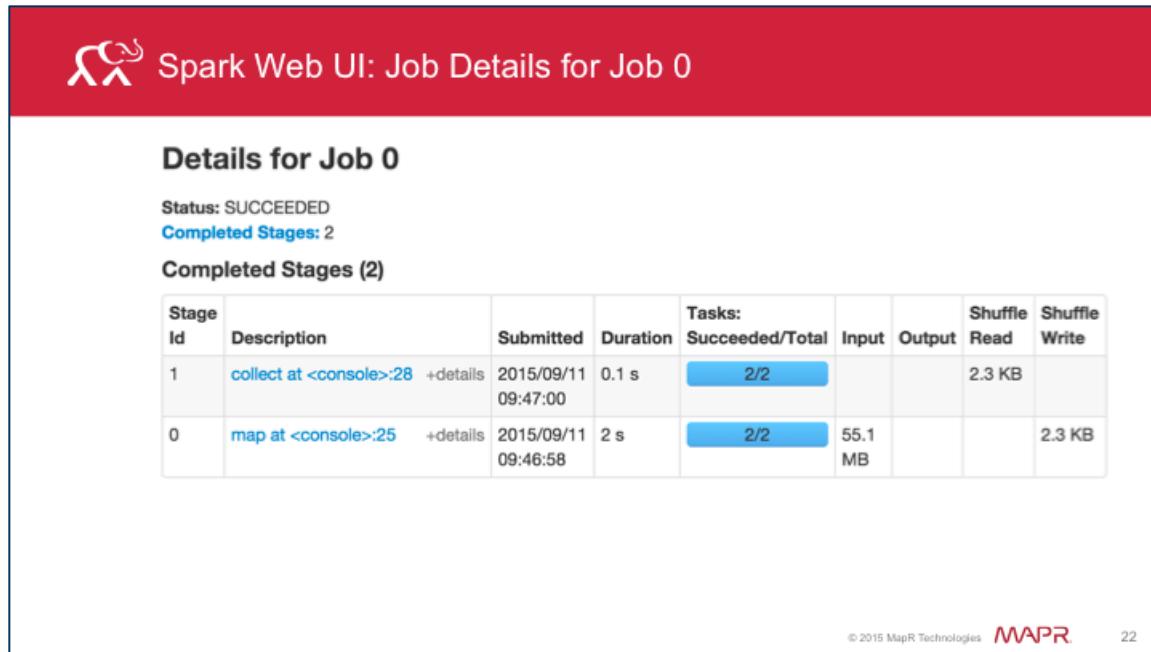
Notes on Slide 21:

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).





Spark Web UI: Job Details for Job 0

Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at <console>:28 +details	2015/09/11 09:47:00	0.1 s	2/2			2.3 KB	
0	map at <console>:25 +details	2015/09/11 09:46:58	2 s	2/2	55.1 MB			2.3 KB

© 2015 MapR Technologies  22

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.





Spark Web UI: Job Details for Job 1

Details for Job 1

Status: SUCCEEDED
Completed Stages: 1
Skipped Stages: 1

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	collect at <console>:28 +details	2015/09/11 09:47:40	31 ms	2/2	4.6 KB			

Skipped Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	map at <console>:25 +details	Unknown	Unknown	0/2				

© 2015 MapR Technologies  23

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.





The screenshot shows the Spark Web UI for Job 1. At the top, there's a red header bar with the MapR logo and the text "Spark Web UI: Stage Details for Job 1". Below the header, there's a section titled "Details for Stage 0" which includes summary metrics like total task time, input size, and shuffle write. There's also a link to "Show additional metrics". The main content area has a title "Summary Metrics for 2 Completed Tasks" with a table showing metrics for Duration, GC Time, Input Size / Records, and Shuffle Write Size / Records. Below this is another section titled "Aggregated Metrics by Executor" with a table showing metrics for Executor ID, Address, Task Time, Total Tasks, Failed Tasks, Succeeded Tasks, Input Size / Records, and Shuffle Write Size / Records. The table shows one row for the driver executor. At the bottom, there's a section titled "Tasks" with a table showing columns for Index, ID, Attempt, Status, Locality, Executor ID / Host, Launch Time, GC Duration, Input Size / Records, Write Time, Shuffle Write Size / Records, and a link. The footer of the page includes the copyright notice "© 2015 MapR Technologies" and the MapR logo.

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.



Spark Web UI: Storage

Jobs Stages Storage Environment Executors

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

Storage

RDD Name	Storage Level
4	Replicated

RDD Storage Info for 4

Storage Level: Memory Deserialized 1x Replicated
 Cached Partitions: 2
 Total Partitions: 2
 Memory Size: 4.6 KB
 Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	Memory Usage	Disk Usage
localhost:53831	4.6 KB (246.0 MB Remaining)	0.0 B

2 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_4_0	Memory Deserialized 1x Replicated	2.5 KB	0.0 B	localhost:53831
rdd_4_1	Memory Deserialized 1x Replicated	2.1 KB	0.0 B	localhost:53831

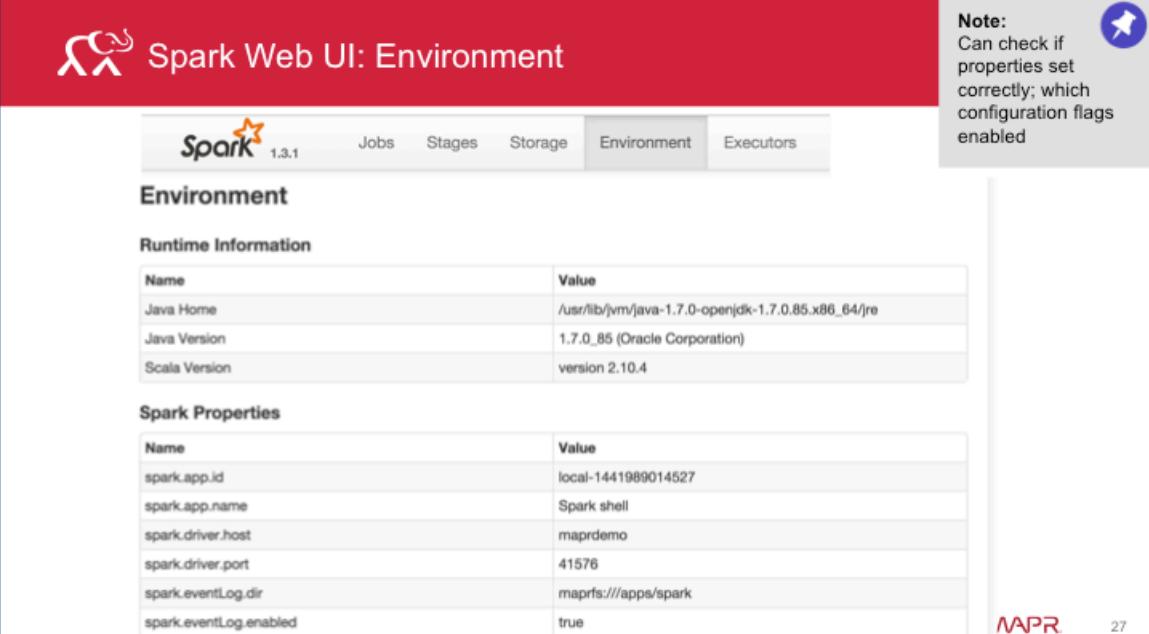
© 2015 MapR Technologies **MAPR** 25

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.





Spark Web UI: Environment

Environment

Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.85.x86_64/jre
Java Version	1.7.0_85 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.app.id	local-1441989014527
spark.app.name	Spark shell
spark.driver.host	maprdemo
spark.driver.port	41576
spark.eventLog.dir	maprfs:///apps/spark
spark.eventLog.enabled	true

Note:
Can check if properties set correctly; which configuration flags enabled

MAPR 27

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.



Executors (1)

Memory: 4.6 KB Used (246.0 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Thread Dump
<driver>	localhost:53831	2	4.6 KB / 246.0 MB	0.0 B	0	0	8	8	4.3 s	55.1 MB	0.0 B	2.3 KB	Thread Dump

Access executors stack trace

© 2015 MapR Technologies **MAPR** 26

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.



The screenshot shows the MapR Control System (MCS) interface. At the top, there's a red header bar with the MapR logo and the text "Spark Web UI via MCS". Below this, the main interface has a green navigation bar on the left with items like Cluster, MapR FS, NFS HA, Alarms, System Settings, Hbase, Job Tracker, CLDB, SparkHistoryServer, ResourceManager, JobHistoryServer, and Nagios. The main content area has tabs for Dashboard, SparkHistoryServer, Jobs, Stages, Storage, Environment, and Executors. The "Jobs" tab is selected, showing the "Spark Jobs" section with a table of completed jobs. The table has columns for Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. There are two rows of data:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

At the bottom right of the interface, it says "© 2015 MapR Technologies" and "MAPR".

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.





Knowledge Check

What are some of the things you can monitor in the Spark Web UI?

- A. Which stages are running slow
- B. Your application has the resources as expected
- C. If the datasets are fitting into memory
- D. All of the above

© 2015 MapR Technologies **MAPR**

29

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.





Learning Goals

1. Describe Components of Spark Execution Model
2. Use Spark Web UI to Monitor Spark Applications
- ▶ **3. Debug & Tune Spark Applications**

© 2015 MapR Technologies **MAPR**

30

Notes on Slide 30:

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





Detect Performance Problems

Spark Web UI: Jobs, Stages & Stage Details

- Are there any tasks that are significantly slow?
- Is issue because of skew?
 - Do some tasks read or write much more data than others?
- Are tasks running on certain nodes slow?
- How much time tasks spend on each phase: read, compute, write?

© 2015 MapR Technologies 

31

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 there are any expensive operations.



Common Slow Performance Issues



Level of Parallelism Serialization Format Memory Management

© 2015 MapR Technologies  32

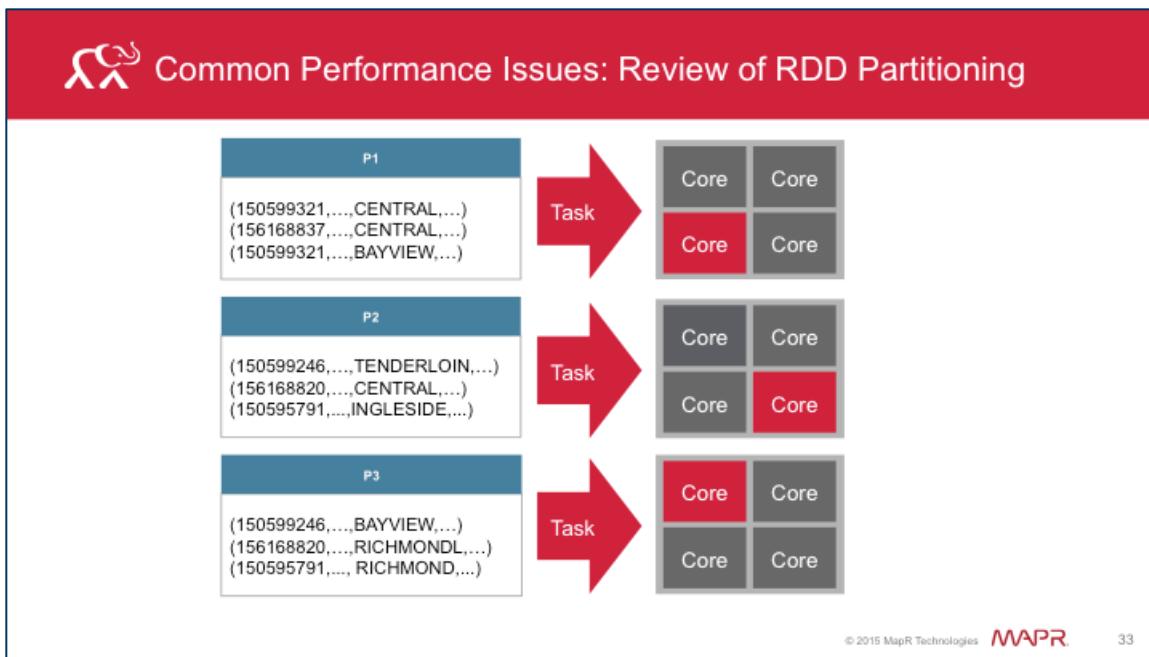
Notes on Slide 32:

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.



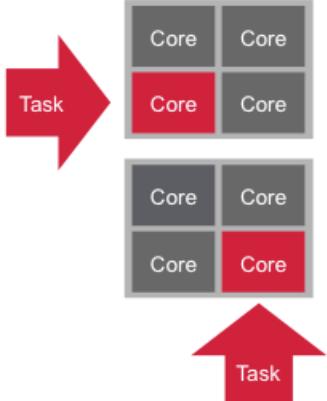


Notes on Slide 33:

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.



Common Performance Issues: Level of Parallelism



© 2015 MapR Technologies  34

Notes on Slide 34:

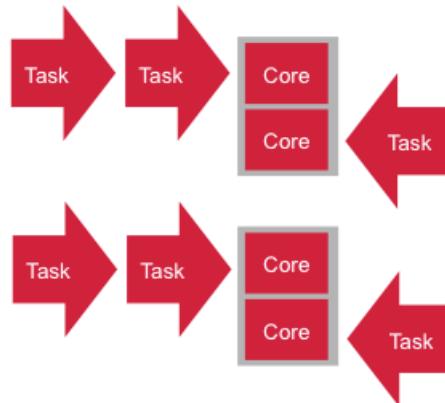
How does the level of parallelism affect performance?

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





Common Performance Issues: Level of Parallelism



© 2015 MapR Technologies 

35

Notes on Slide 35:

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





Common Performance Issues: Tuning Level of Parallelism

Check the number of partitions

1. Use Spark Web UI Stages.

Total tasks = number of partitions

2. Use `rdd.partitions.size()`

Tasks:	Succeeded/Total
	2/2
	2/2
	2/2
	2/2

© 2015 MapR Technologies 

36

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.





Common Performance Issues: Tuning Level of Parallelism

Tune level of parallelism:

1. Specify degree of parallelism for operations that shuffle data
 - For example, `reduceByKey(func, numPartitions)`
2. Change the number of partitions (fewer or more) in an RDD
 - `repartition()`
 - `coalesce()`

© 2015 MapR Technologies 

37

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.





Common Performance Issues: Serialization Format

- During data transfer, Spark serializes data
- Happens during shuffle operations
- Java built in serializer is default
- Use Kryo serialization – often more efficient

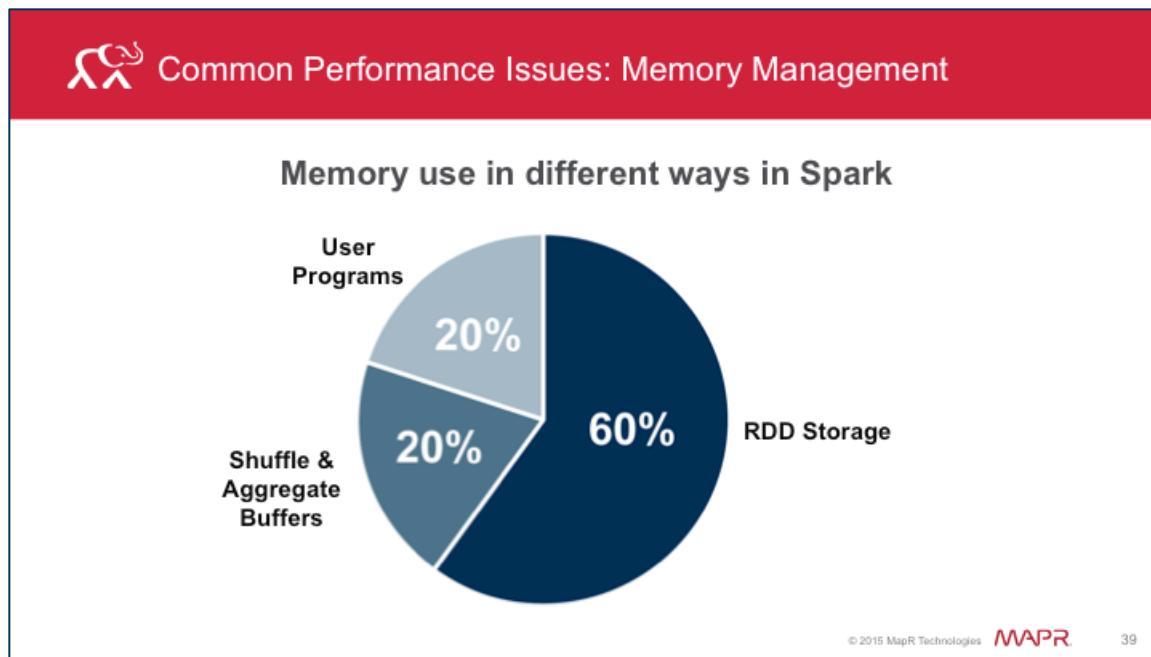
© 2015 MapR Technologies 

38

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





Common Performance Issues: Tuning Memory Usage

- RDD storage: `cache()` and `persist()`
- Various options: `persist()`
- By default: `persist(MEMORY_ONLY)` level
- Reduce expensive computations: `persist(MEMORY_AND_DISK)`
- Reduce garbage collection: `(MEMORY_ONLY_SER)`

© 2015 MapR Technologies

40

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.





Spark logging subsystem based on log4j

Deployment Mode	Location
Spark Standalone	work/ directory of distribution on each worker
Mesos	work/ directory of Mesos slave
YARN	Use YARN log collection tool

© 2015 MapR Technologies  41

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.





Best Practices & Tips

- Avoid shuffling large amounts of data
- Do not copy all elements of RDD to driver
- Filter sooner rather than later
- If you have many idle tasks, `coalesce()`
- If not using all slots in cluster, repartition

© 2015 MapR Technologies

42

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





Knowledge Check

Some ways to improve performance of your Spark application include:

- A. Using Kyro serialization
- B. Tune the degree of parallelism
- C. Avoid shuffling large amounts of data
- D. Don't use `collect()` on large datasets
- E. All of the above

© 2015 MapR Technologies **MAPR**

43

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.



**DEV 362**

Create Data Pipelines
Using Apache Spark

© 2015 MapR Technologies **MAPR**

45

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

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2016, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.





DEV362 – Create Data Pipelines With Apache Spark

Lesson 7: Introduction to Apache Spark Data Pipelines

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



Learning Goals



Learning Goals



- Identify Spark Unified Stack Components
- List Benefits of Apache Spark over Hadoop Ecosystem
- Describe Spark Data Pipeline Use Cases

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

- Identify the components of the Spark unified stack
- List the benefits of Apache Spark over the Hadoop ecosystem
- Describe Apache Spark data pipeline use cases

Learning Goals

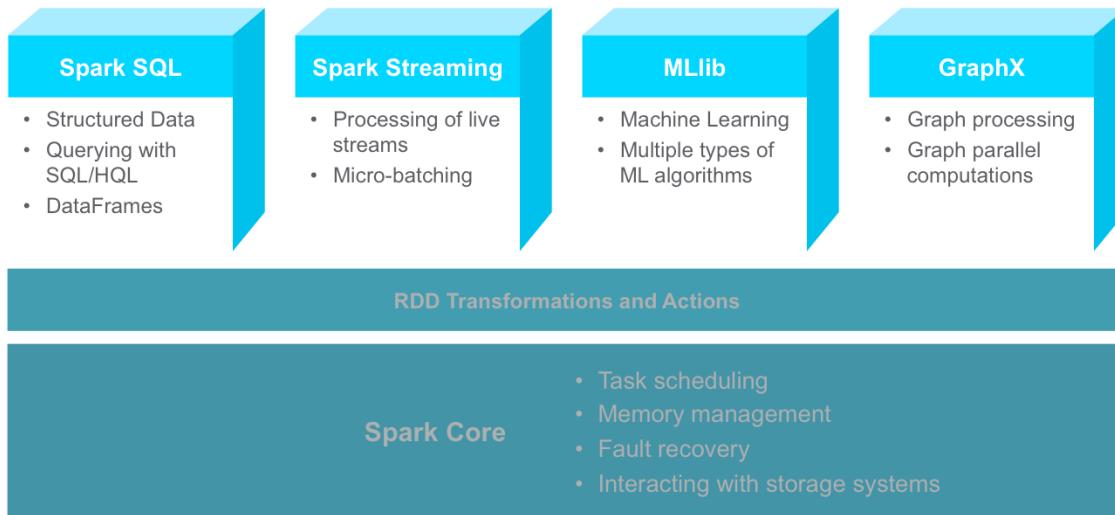


- **Identify Spark Unified Stack Components**
- List Benefits of Apache Spark over Hadoop Ecosystem
- Describe Spark Data Pipeline Use Cases

In this first section, we will discuss the components of the Apache Spark unified stack.



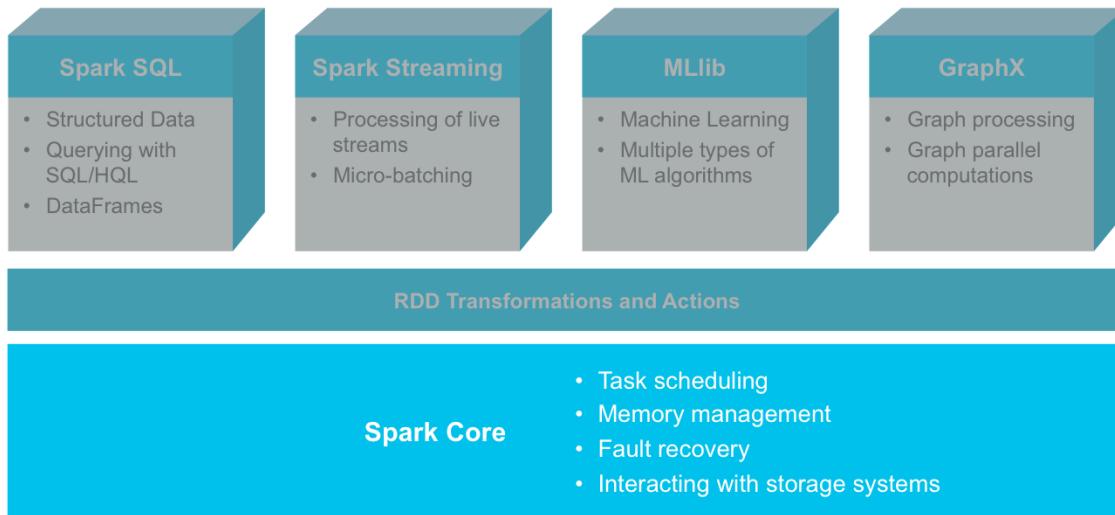
Apache Spark Unified Stack



Apache Spark has an integrated framework for advanced analytics like graph processing, advanced queries, stream processing and machine learning. You can combine these libraries into the same application and use a single programming language through the entire workflow.



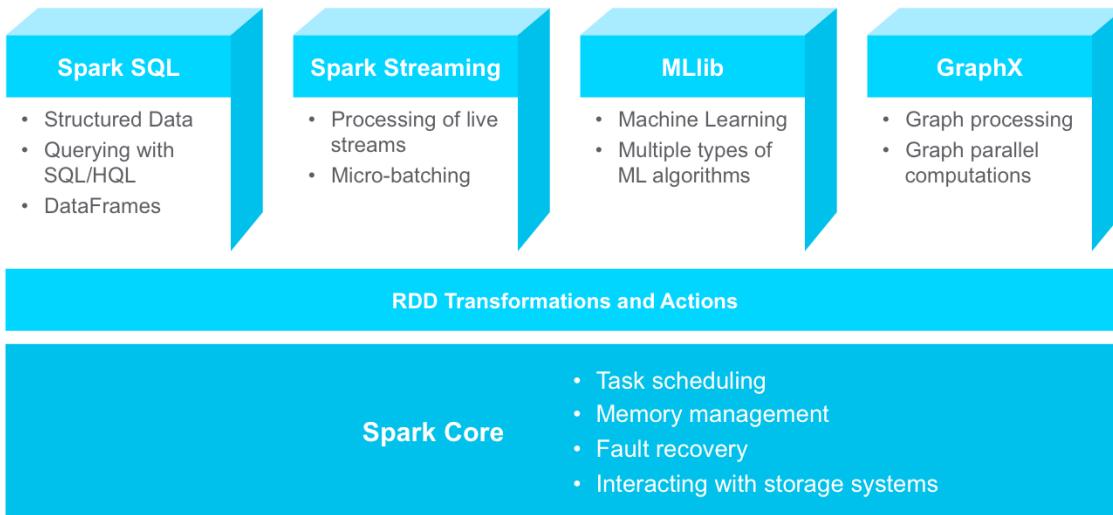
Apache Spark Unified Stack



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



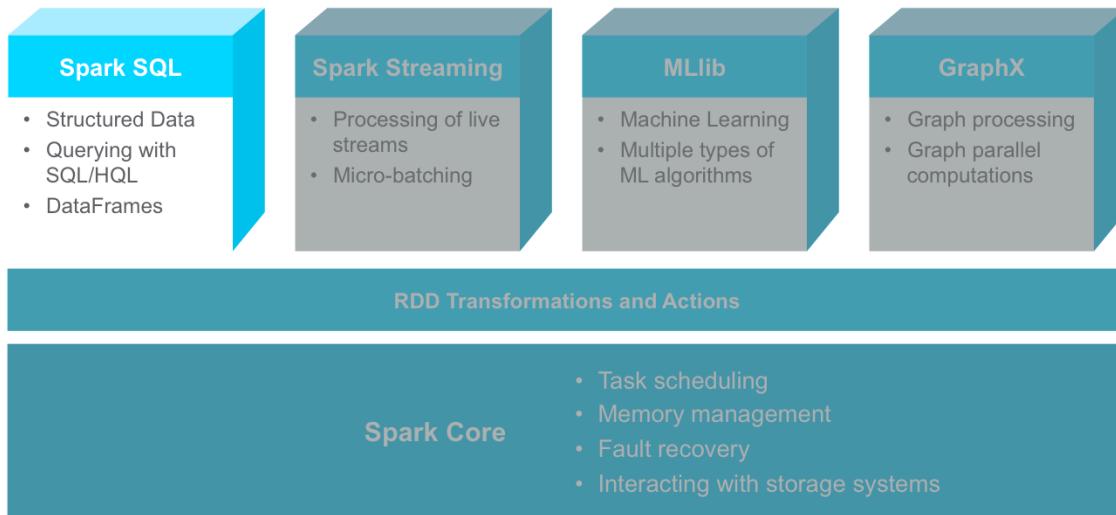
Apache Spark Unified Stack



The other Spark modules shown here are tightly integrated with the Spark core.



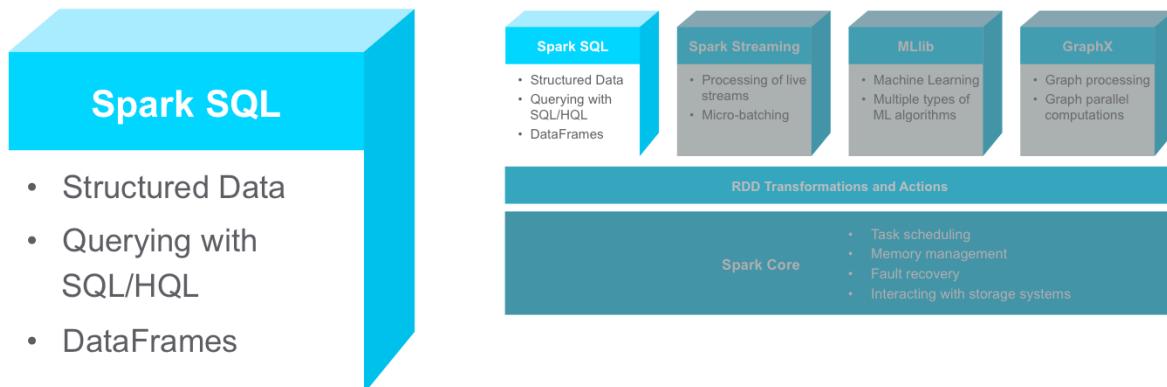
Apache Spark Unified Stack – Spark SQL



Spark SQL is used to process structured data.

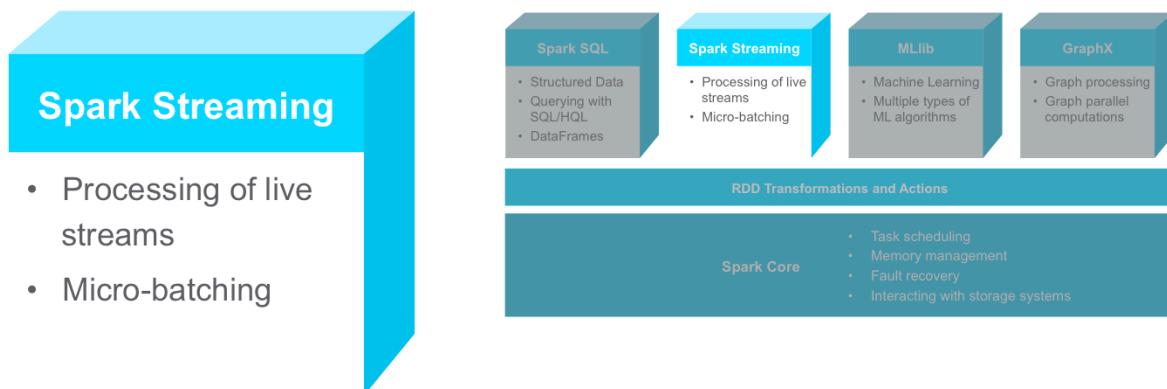


Apache Spark Unified Stack – Spark SQL



Spark provides native support for SQL queries. You can query data via SQL or HiveQL. Spark SQL supports many types of data sources, such as structured Hive tables and complex JSON data. The primary abstraction of Spark SQL is Spark DataFrames.

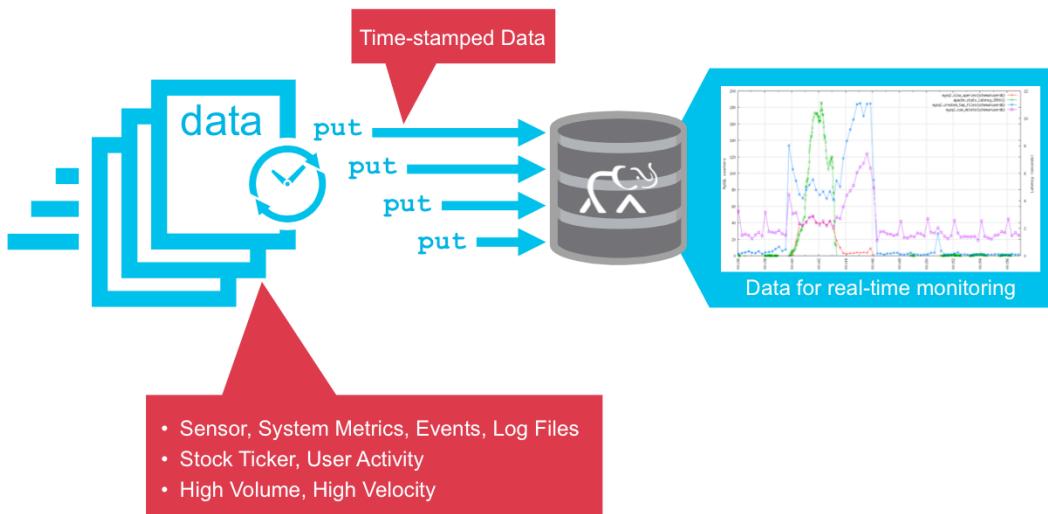
Apache Spark Unified Stack – Spark Streaming



Spark streaming enables processing of live streams of data.

When performing analytics on the live data streams, Spark streaming uses a micro-batch execution model.

Apache Spark Unified Stack – Spark Streaming



Many applications need to process streaming data with the following requirements:

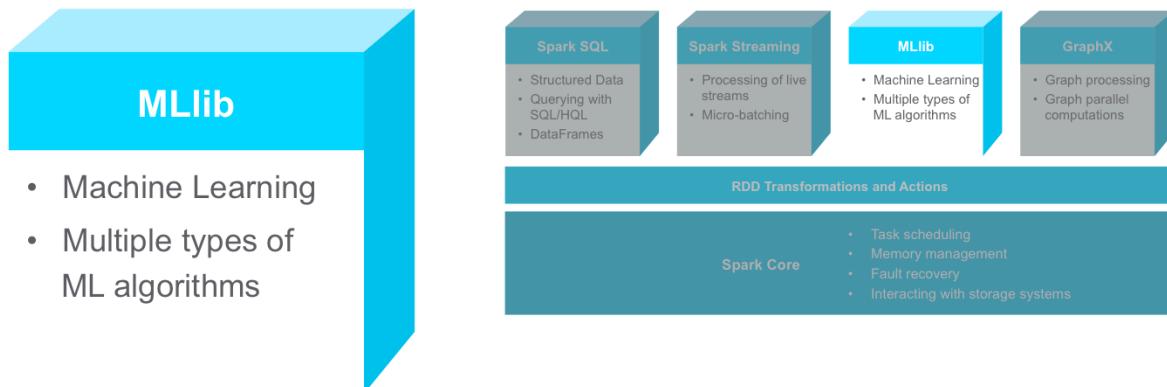
- The results should be in near-real-time
- Be able to handle large workloads and
- Have latencies of few seconds

Batch processing on the other hand processes large volumes of data collected over a period of time, where the latency is in terms of minutes.

Common examples of streaming data include activity stream data from a web or mobile application, time stamped log data, transactional data and event streams from sensor device networks.

Real-time applications of stream processing include website monitoring, network monitoring, fraud detection, web clicks and advertising.

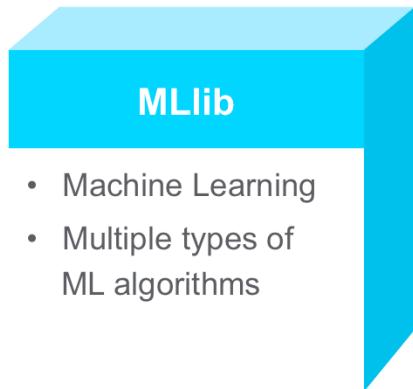
Apache Spark Unified Stack – Spark MLlib



MLlib is a Spark machine learning library that provides multiple types of machine learning algorithms such as classification, regression, clustering, collaborative filtering and optimization primitives.



Apache Spark Unified Stack – Spark MLlib



- Machine Learning
- Multiple types of ML algorithms

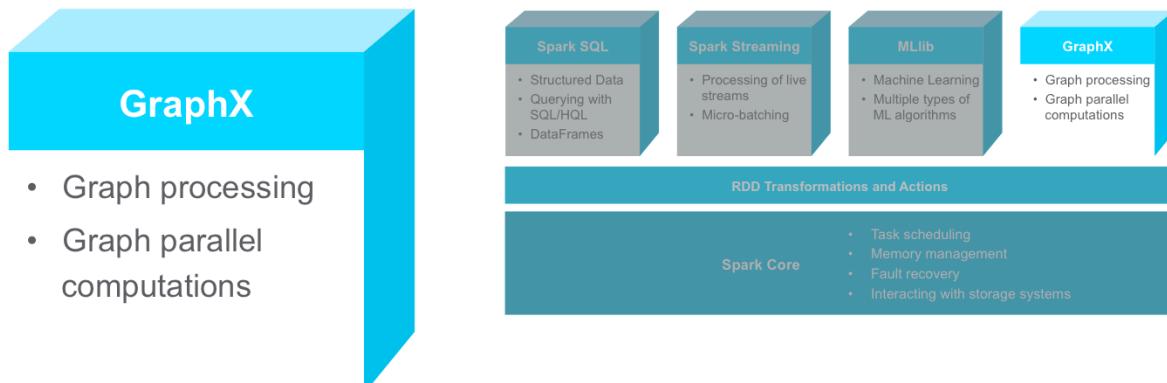
- Spark Machine Learning Library
- Common learning algorithms
 - Classification and Regressions
 - Clustering
 - Collaborative filtering
 - Dimensionality Reduction
- Two packages
 - spark.mllib
 - spark.ml

MLlib is the Spark machine learning library. It supports common learning algorithms such as classification, regressions, clustering, collaborative filtering and dimensionality reduction.

There are two packages available:

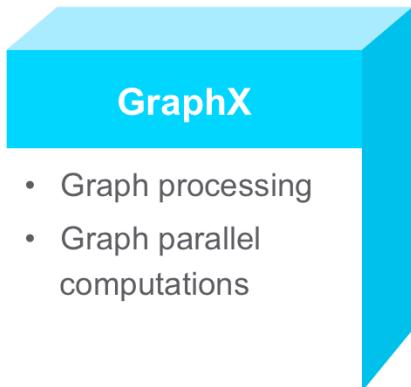
- spark.mllib – original API that can be used on RDDs
- spark.ml – higher level API that can be used on DataFrames

Apache Spark Unified Stack – Spark GraphX



GraphX is a library for manipulating graphs and performing graph-parallel computations.

Apache Spark Unified Stack – Spark GraphX



- Graph processing
- Graph parallel computations

- Graph abstraction extends Spark RDD
- Can work seamlessly with both graphs and collections
- Operations for graph computation
 - Includes optimized version of Pregel
- Provides graph algorithms & builders

The primary abstraction is a graph that extends the Spark RDD. A Graph is a directed multigraph wherein each vertex and edge has properties attached. GraphX has a number of operators that can be used for graph computations including an optimized version of Pregel. GraphX also provides graph algorithms and builders for graph analytics.

We can work seamlessly with both graphs and collections; view same data as both graphs and collections without duplication or movement of data.
We can also write custom iterative graph algorithms

Learning Goals



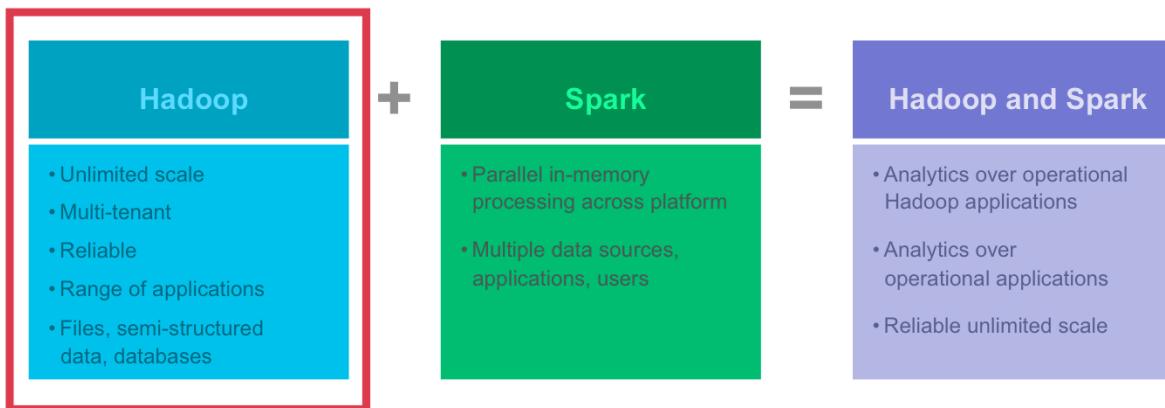
Learning Goals



- Identify Spark Unified Stack Components
- **List Benefits of Apache Spark over Hadoop Ecosystem**
- Describe Spark Data Pipeline Use Cases

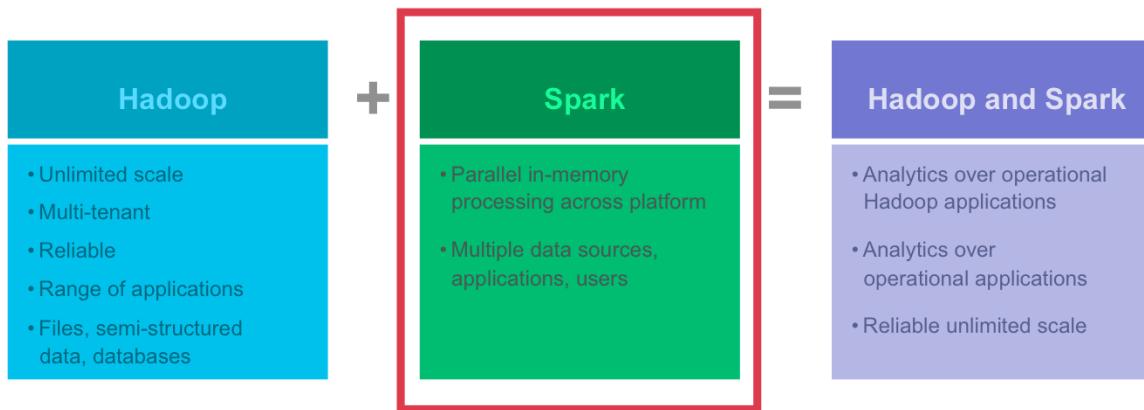
Now let us take a look at how Spark fits in with Hadoop, and the benefits of the Spark unified stack over the Hadoop Ecosystem.

Apache Spark and Apache Hadoop



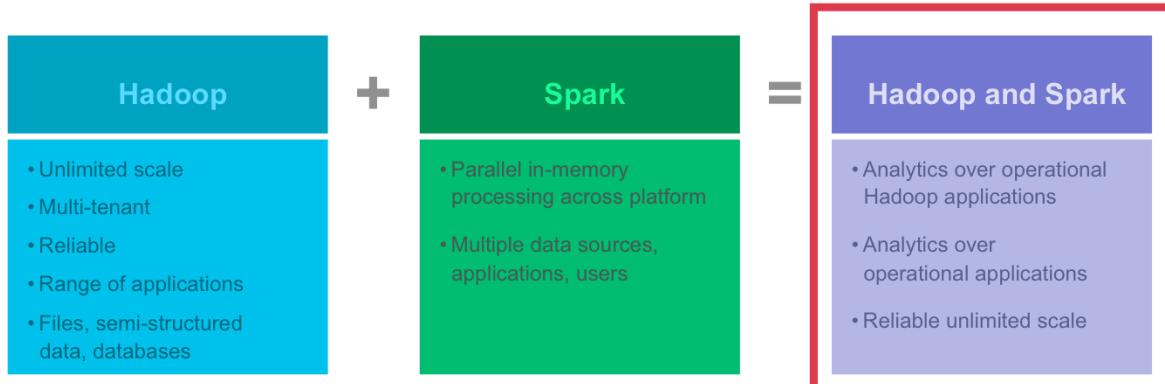
Hadoop has grown into a multi-tenant, reliable, enterprise-grade platform with a wide range of applications that can handle files, databases, and semi-structured data.

Apache Spark and Apache Hadoop



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

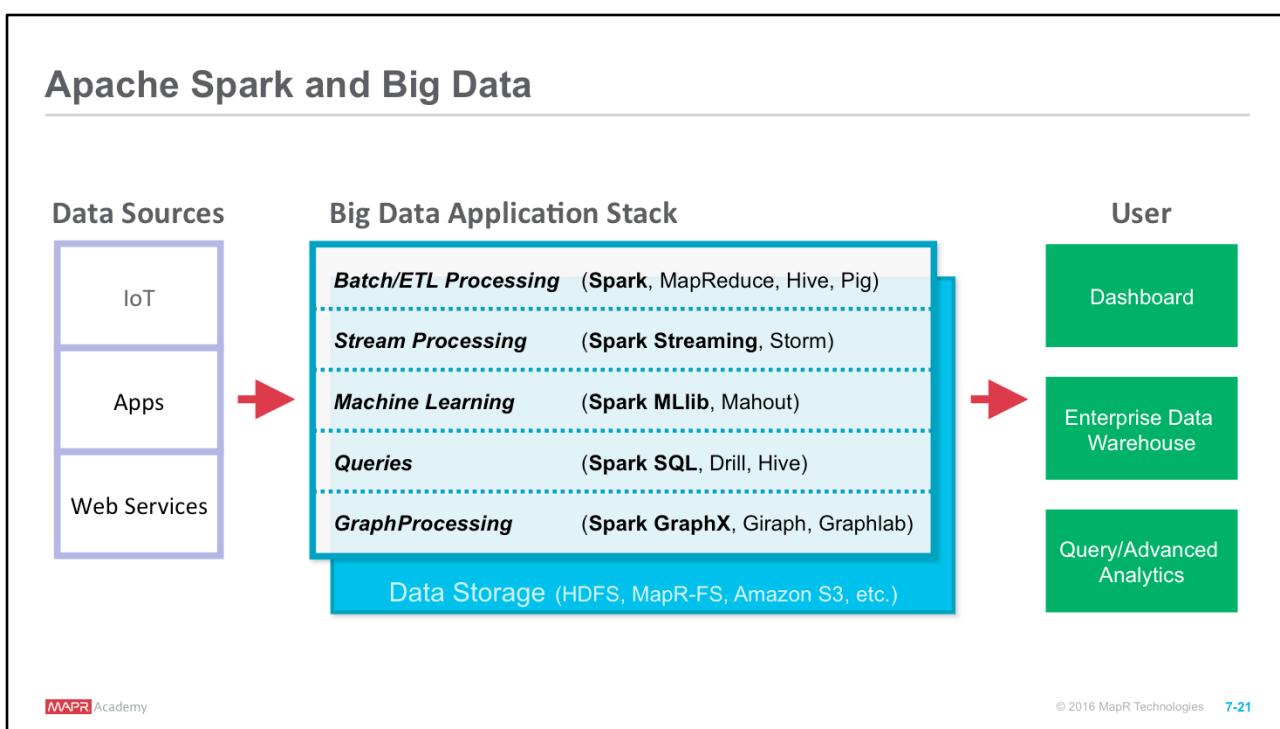
Apache Spark and Apache Hadoop



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



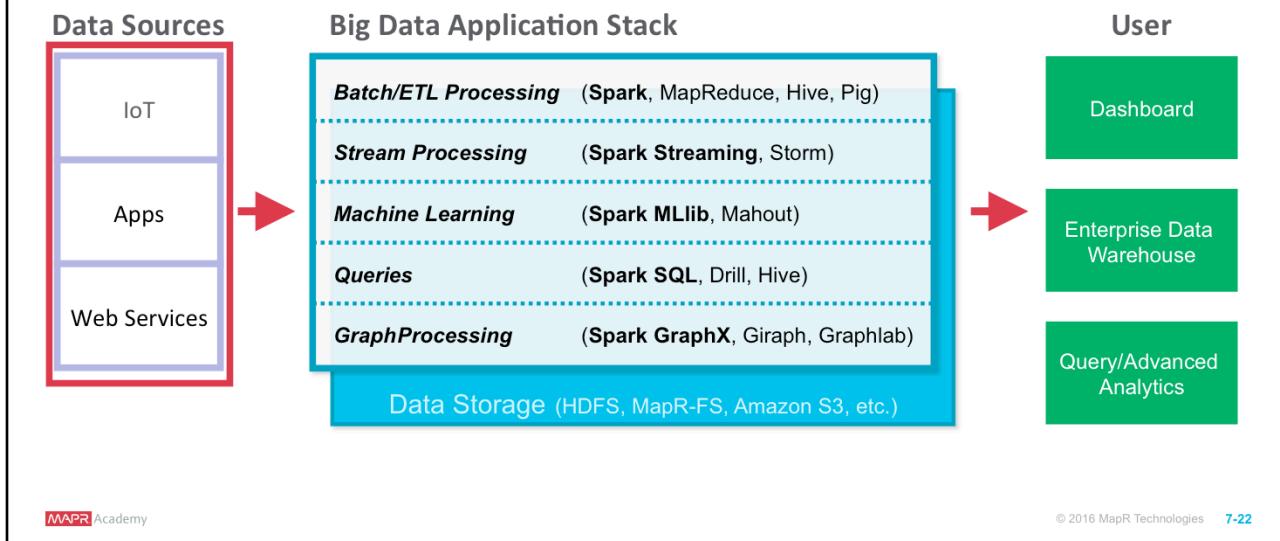
Apache Spark and Big Data



This graphic depicts where Spark fits in the big data application stack.



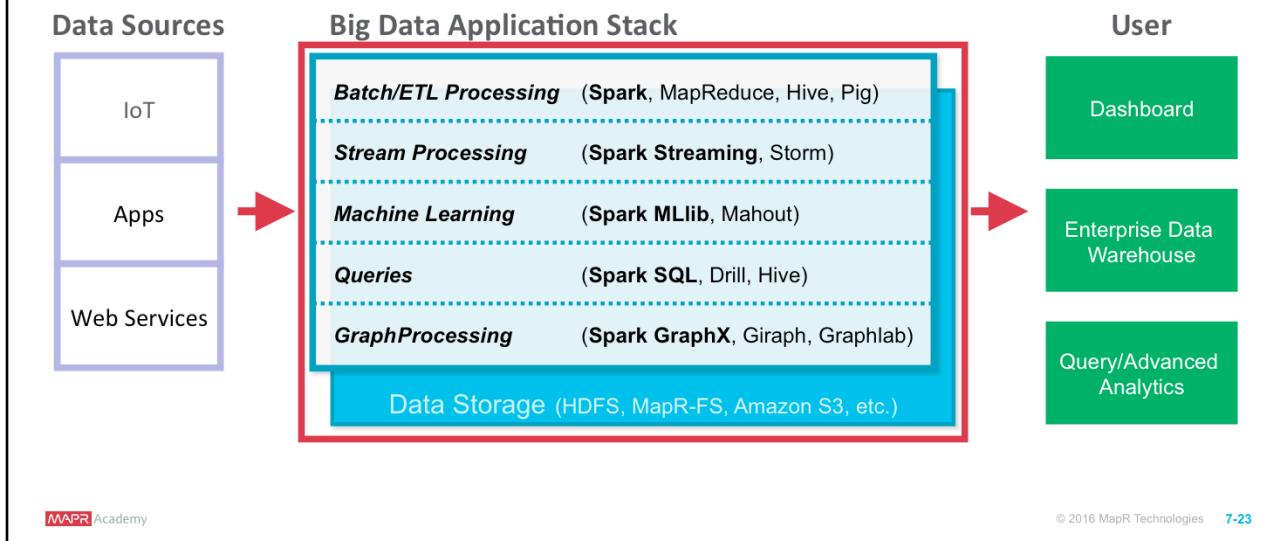
Apache Spark and Big Data



On the left we see different data sources. There are multiple ways of ingesting data, using different industry standards such as NFS or existing Hadoop tools.



Apache Spark and Big Data

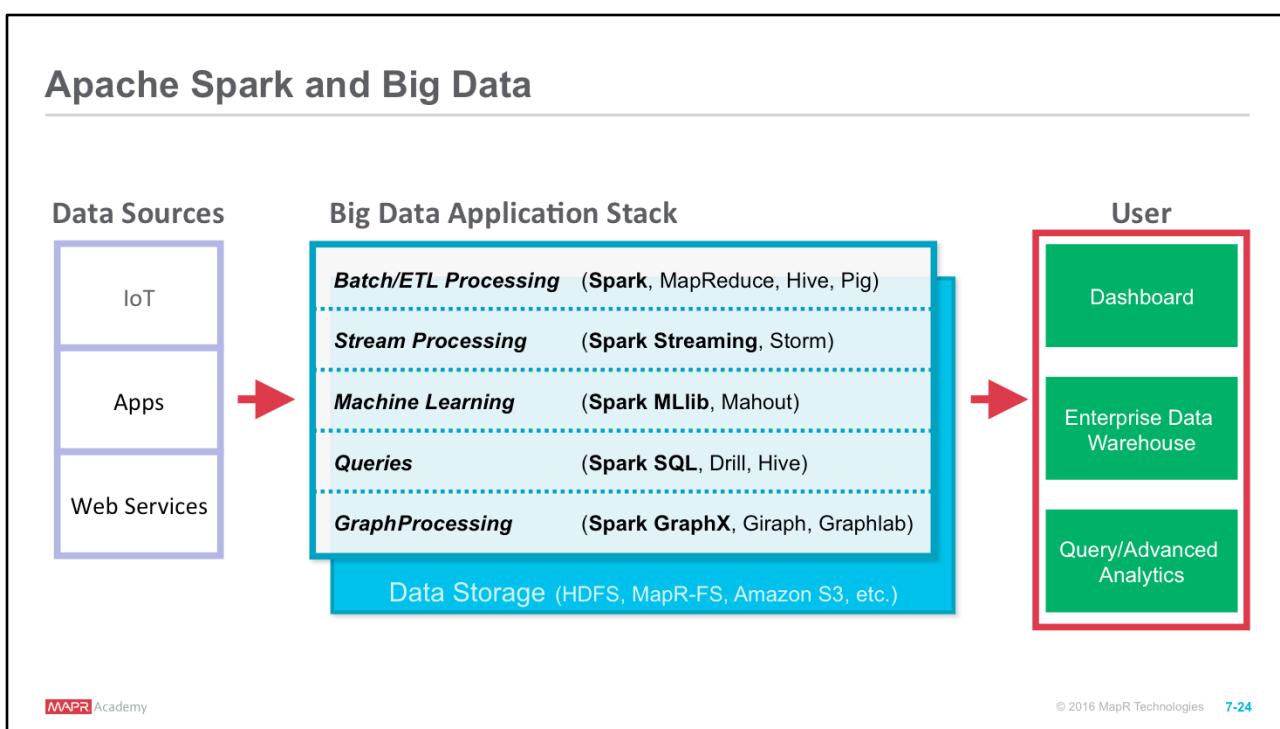


The stack in the middle represents various big data processing workflows and tools that are commonly used. You may have just one of these workflows in your application, or a combination of many. Any of these workflows could read/write to or from the storage layer.

While you can combine various workflows in Hadoop, it requires using different languages and different tools. As you can see here, with Spark, you can use Spark for any of the workflows. You can build applications that ingest streaming data and apply machine learning and graph analysis to the live streams. All this can be done in the same application using one language.



Apache Spark and Big Data



The output can then be used to create real-time dashboards and alerting systems for querying and advanced analytics.



Learning Goals



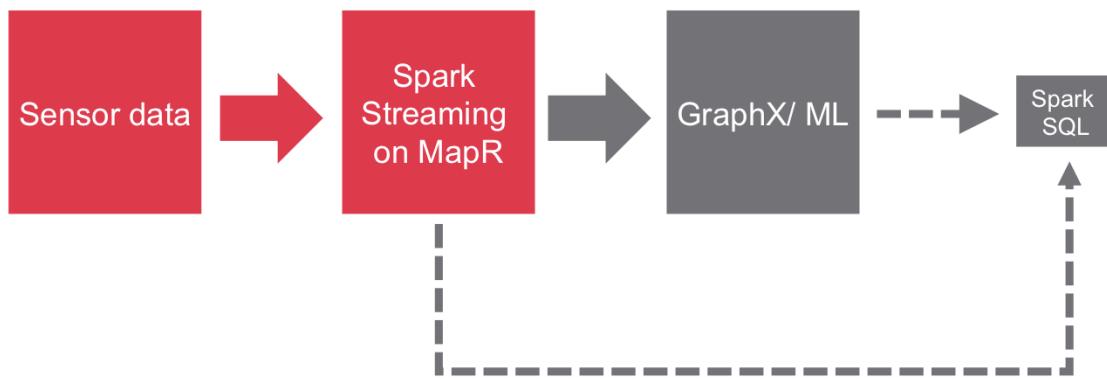
Learning Goals



- Identify Spark Unified Stack Components
- List Benefits of Apache Spark over Hadoop Ecosystem
- **Describe Spark Data Pipeline Use Cases**

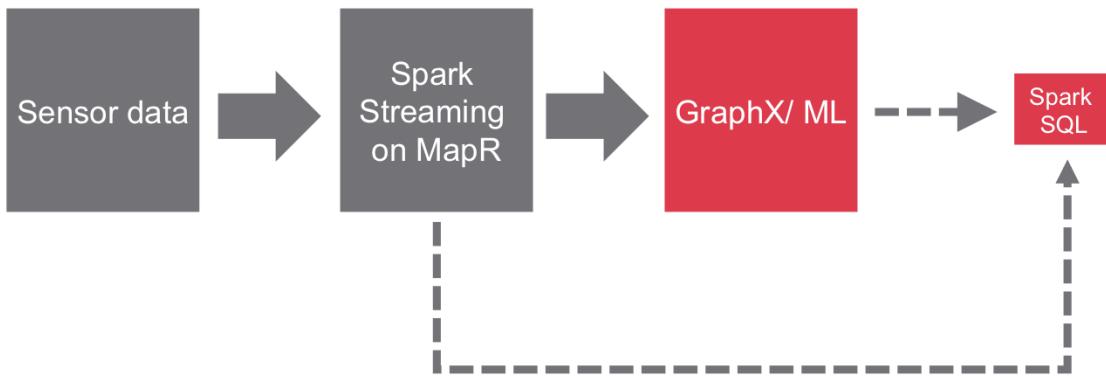
This section provides some examples of data pipeline applications using Apache Spark.

Use Case: Managed Security Services Provider



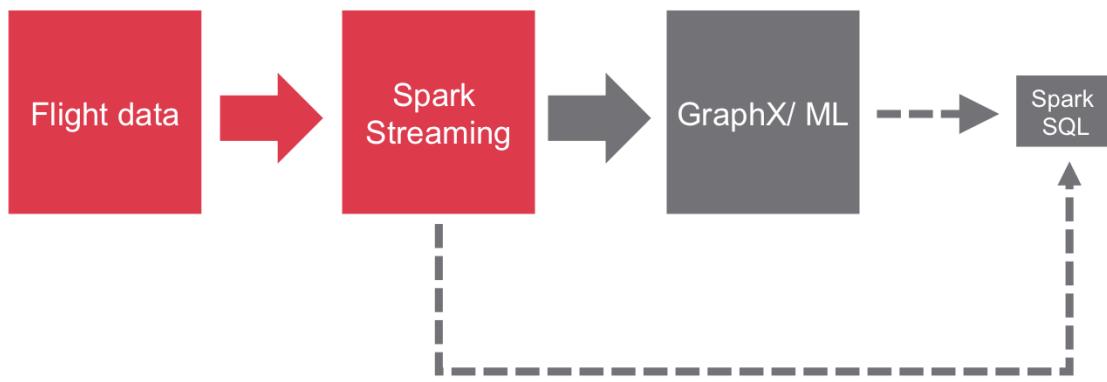
In our first use case, sensor data is streamed in using Spark Streaming on MapR and is checked for first known threats.

Use Case: Managed Security Services Provider



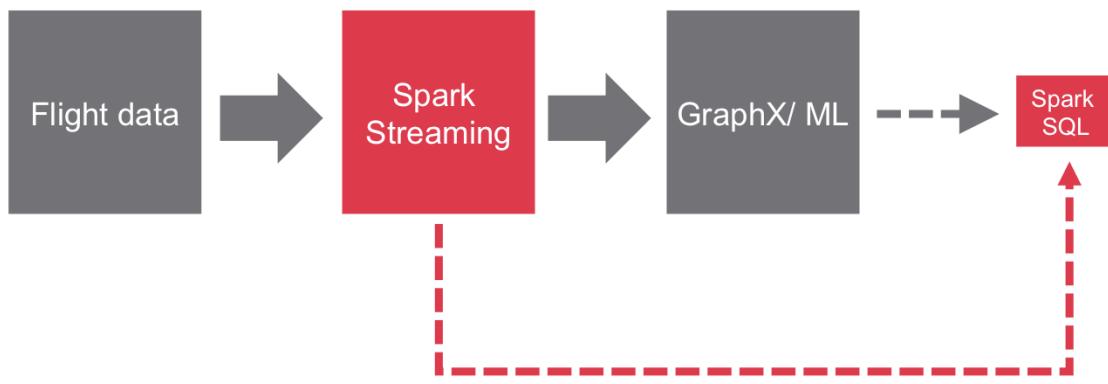
The data then goes through graph processing and machine learning for predictions. Additional querying such as results of graph algorithms, predictive models and summary/aggregate data is done using Spark SQL.

Use Case: Logistics Optimization



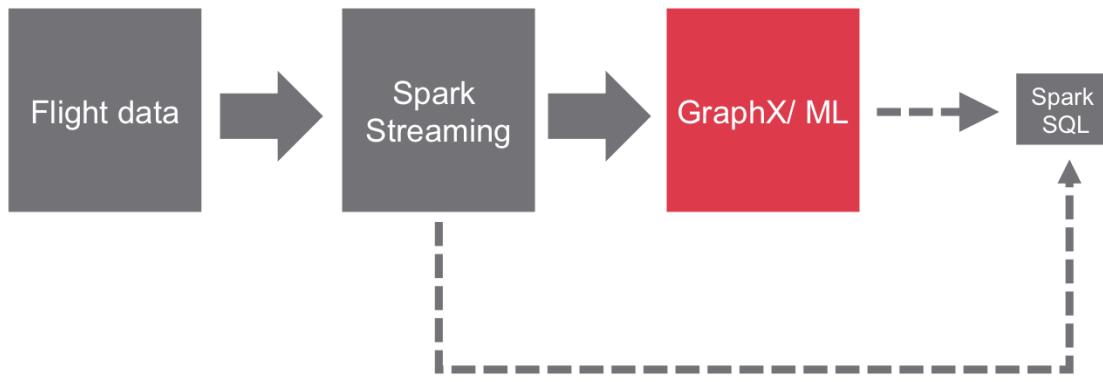
In a second use case, we begin with flight data being streamed in.

Use Case: Logistics Optimization



We use Spark SQL to do some analytics on the streaming data.

Use Case: Logistics Optimization



We use GraphX to analyze airports and routes that serve them. Machine learning is used to predict delays using a classification or decision tree algorithms.

Knowledge Check



Knowledge Check



We have real-time twitter feed. We need to build an application that is near real-time and classifies the twitter feeds based on relevant and not relevant, where “relevant” means that it contains the words “FIFA”, “Women’s”, and “World Cup”. Which of the following Apache Spark libraries could we use in the application?

- A. Spark SQL
- B. Spark Streaming
- C. Spark MLlib
- D. Spark GraphX

Use Spark SQL to query the data in DataFrames, Spark Streaming to ingest the live feeds and Spark MLlib to do the classification.



Next Steps

DEV362 – Create Data Pipelines With Apache Spark

Lesson 8: Create an Apache Spark Streaming Application

Congratulations, you have completed DEV 362 Lesson 7. Continue on to lesson 8 to learn about how to create an Apache Spark Streaming application.





DEV362 – Create Data Pipelines With Apache Spark

Lesson 8: Create an Apache Spark Streaming Application

Welcome to DEV 362 lesson 8, create an Apache Spark Streaming application.



Learning Goals



Learning Goals



- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

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

- Describe the Apache Spark Streaming architecture
- Create DStreams and a Spark Streaming application
- Apply operations on DStreams
- Define windowed operations and
- Describe how streaming applications are fault tolerant

Learning Goals



- **Describe Spark Streaming Architecture**
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In the first section we will describe the Spark streaming architecture.



Stream Processing Architecture



MAPR Academy

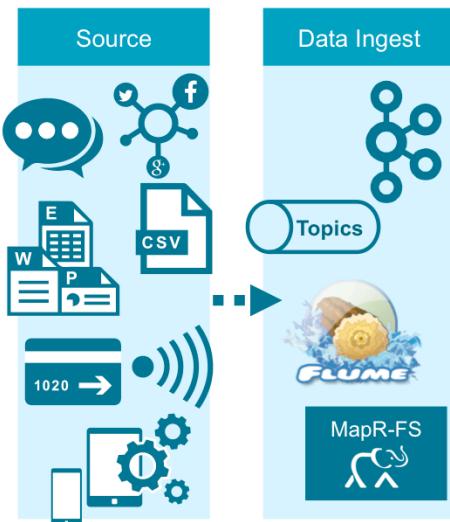
© 2016 MapR Technologies 8-5

A stream processing architecture is typically made of the following components. First, the data we want to process must come from somewhere.

Sources refers to the source of data streams. Examples include sensor networks, mobile applications, web clients, logs from a server, or even a “Thing” from the Internet of Things.



Stream Processing Architecture



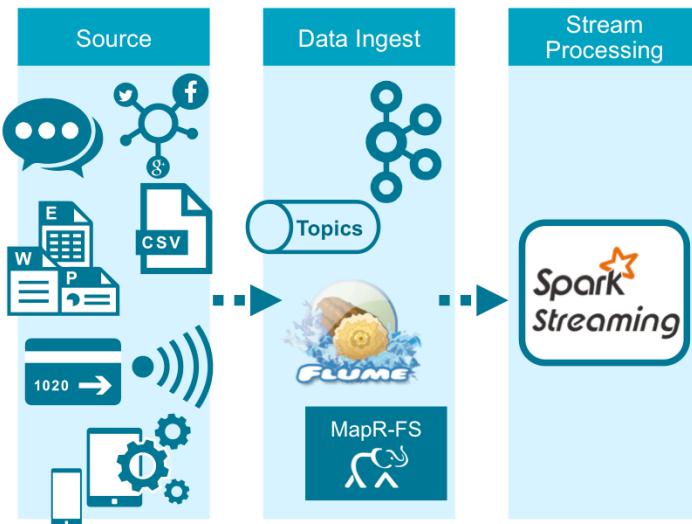
MAPR Academy

© 2016 MapR Technologies 8-6

This data is delivered through messaging systems such as MapR Streams, Kafka, Flume, or deposited in a file system.



Stream Processing Architecture

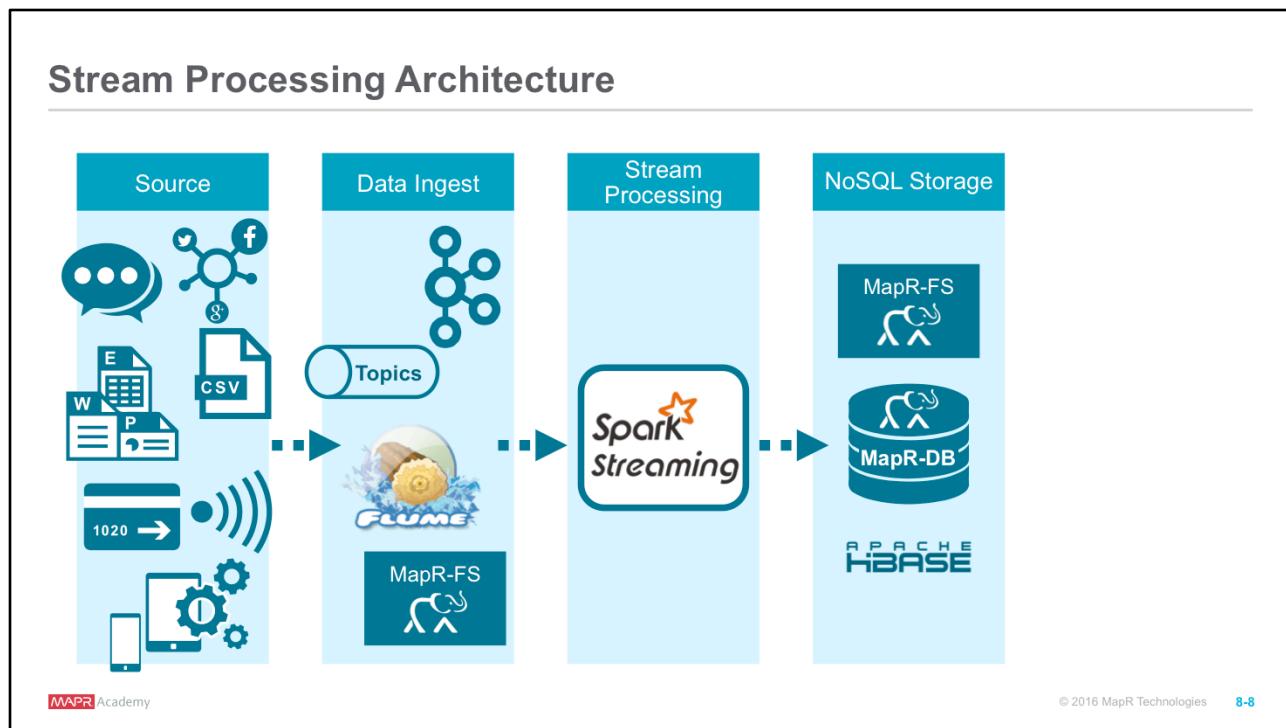


MAPR Academy

© 2016 MapR Technologies 8-7

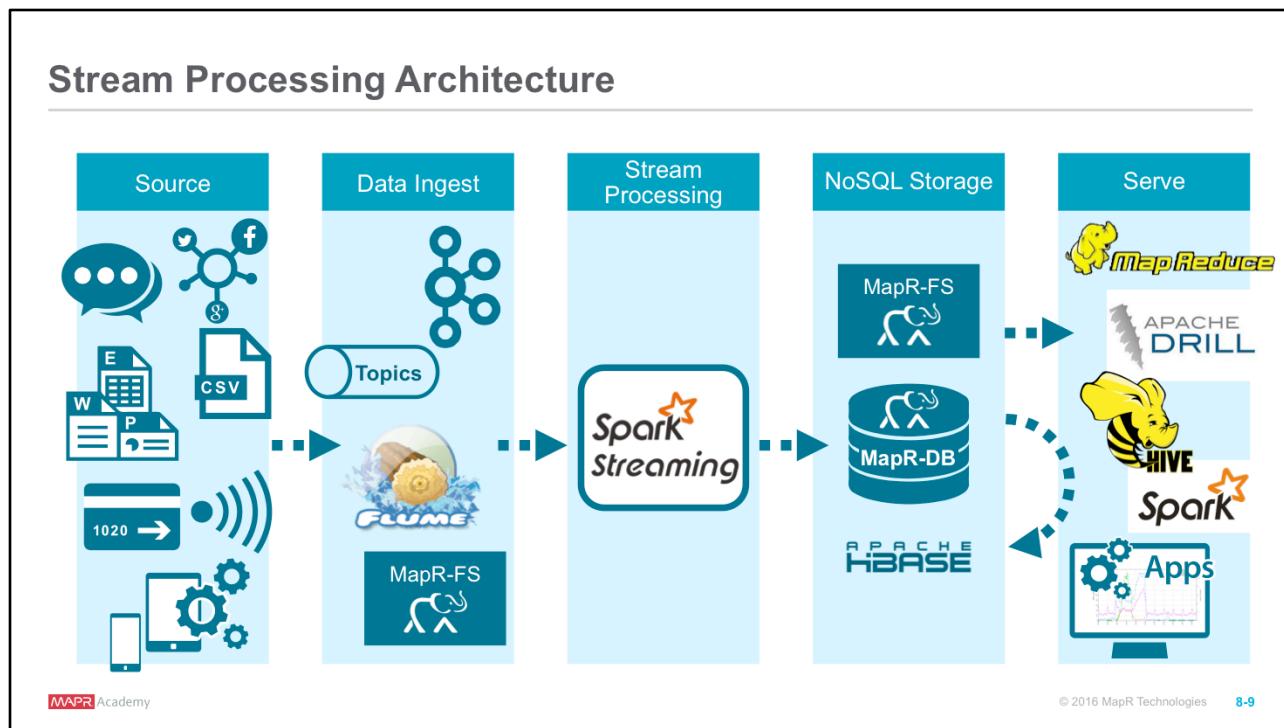
The data is then processed by a stream processing system like Spark Streaming, which is a framework for processing data streams.



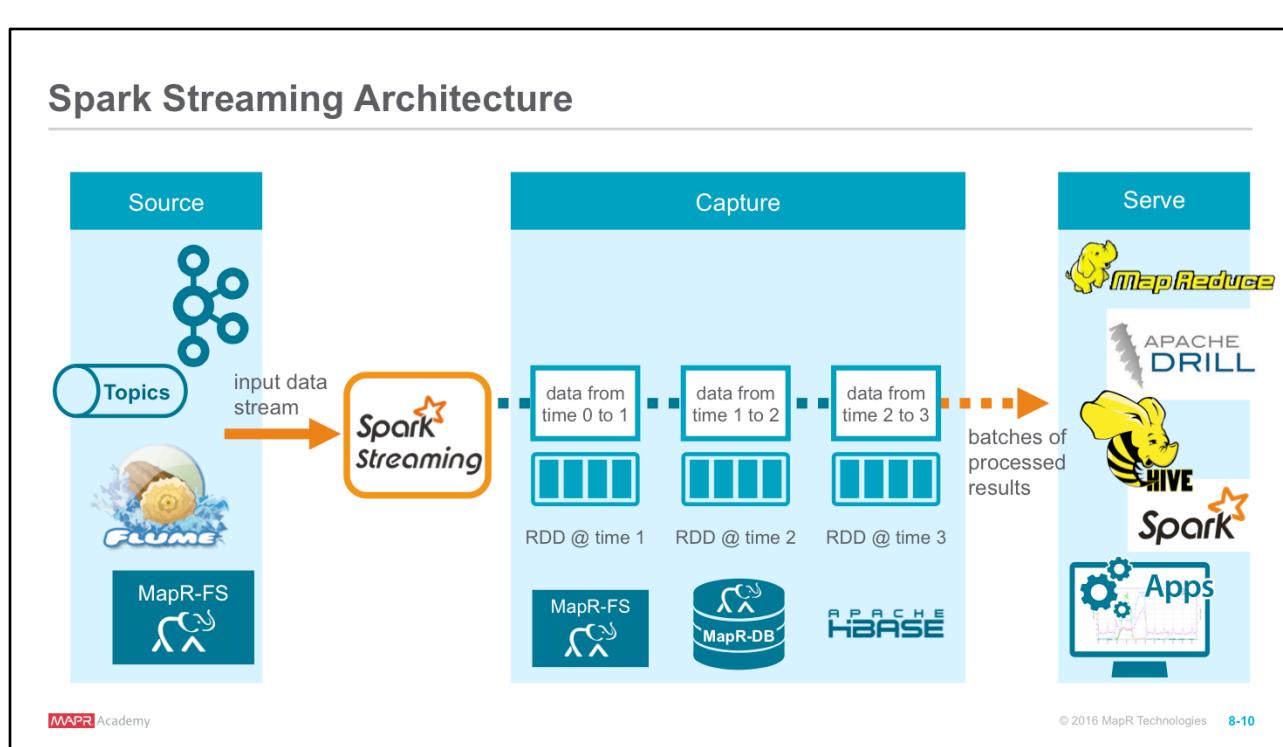


A NoSQL database, such as HBase, MapR-DB, or MapR-FS is used for storing processed data. This system must be capable of low latency, fast read and writes.





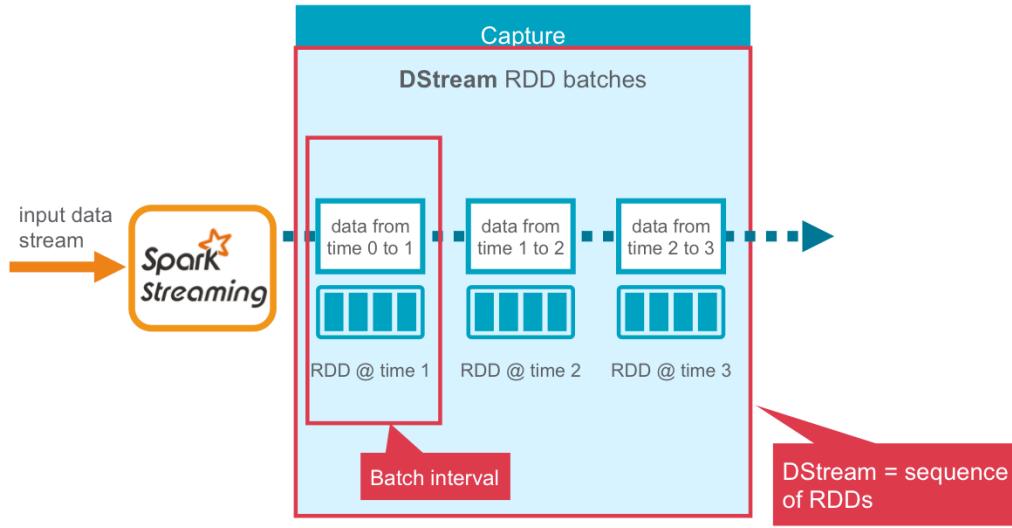
End applications like dashboards, business intelligence tools and other applications use the processed data. The output can also be stored back in our database for further processing later.



Possible input sources include Flume, MapR Streams, and HDFS. Streaming data is continuous, but to process the data stream, it needs to be batched.

Processing Spark DStreams

Data stream divided into batches of X milliseconds = DStreams



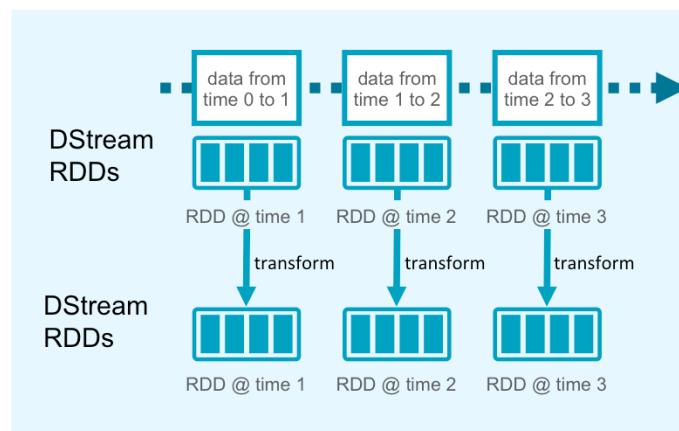
Spark Streaming divides the data stream into batches of X milliseconds called discretized streams, or DStreams.

A DStream is a sequence of mini-batches, where each mini-batch is represented as a Spark RDD. The stream is broken up into time periods equal to the batch interval.

Each RDD in the stream will contain the records that are received by the Spark Streaming application during the batch interval.

Processing Spark DStreams

Process using transformations → creates new RDDs

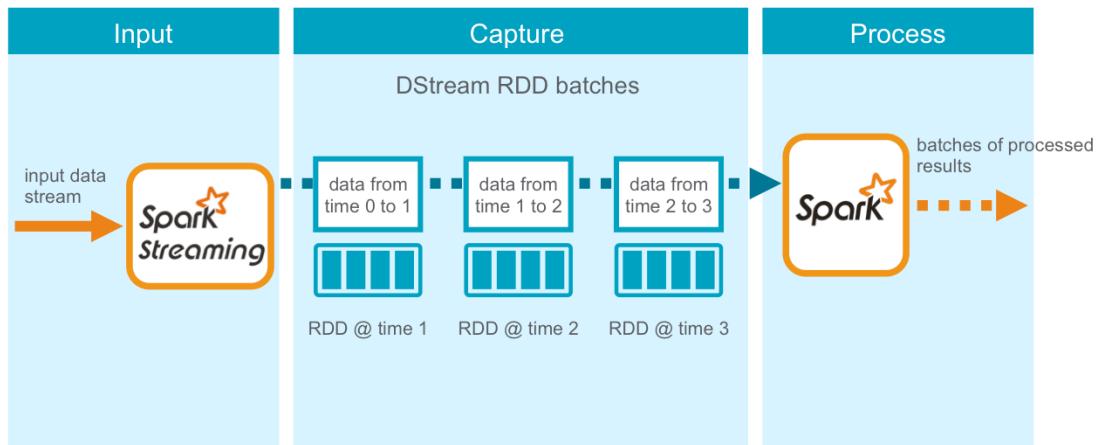


There are two types of operations on DStreams: *transformations* and *output operations*.

Your Spark application processes the DStream RDDs using Spark transformations like map, reduce, and join, which create new RDDs. Any operation applied on a DStream translates to operations on the underlying RDDs, which in turn, applies the transformation to the elements of the RDD.

Processing Spark DStreams

Processed results are pushed out in batches



Output operations are similar to RDD actions in that they write data to an external system, but in Spark Streaming they run periodically on each time step, producing output in batches.

Spark StreamingContext

- Entry point to all streaming functionality
- Can create new StreamingContext from:

- Existing SparkContext sc

```
val ssc = new StreamingContext(sc, Seconds(5))
```

- Existing SparkConf conf

```
val ssc = new StreamingContext(sparkConf, Seconds(5))
```

Spark StreamingContext is the entry point to all streaming functionality. It gives access to methods for creating DStreams from various input sources.

The StreamingContext can be created from an existing SparkContext or SparkConf, as shown in these examples. It can also be created by specifying a Spark master URL, and an app name.



Spark StreamingContext

- Entry point to all streaming functionality
- Can create new StreamingContext from:

- Existing SparkContext sc

```
val ssc = new StreamingContext(sc, Seconds(5))
```

batchDuration
Time interval at which
streaming data divided
into batches

- Existing SparkConf conf

```
val ssc = new StreamingContext(sparkConf, Seconds(5))
```

The second parameter here is the batchDuration. This is the time interval at which the streaming data is divided into batches.

Whether using the Spark Interactive Shell or creating a standalone application, we need to create a new StreamingContext.



Knowledge Check



Knowledge Check



Spark Streaming converts streaming data into DStreams. Which the statements below about DStreams are true?

- A DStream is a sequence of mini batches of streamed content
- Each mini-batch in a DStream is represented as a Spark RDD
- You can run Spark transformations and output operations on a DStream
- Spark StreamingContext gives access to methods for creating DStreams from various input sources
- StreamingContext can be created from an existing SparkContext or SparkConf

True

False – each complete DStream, sequence of mini-batches, is represented by an RDD

True

True

True

Learning Goals



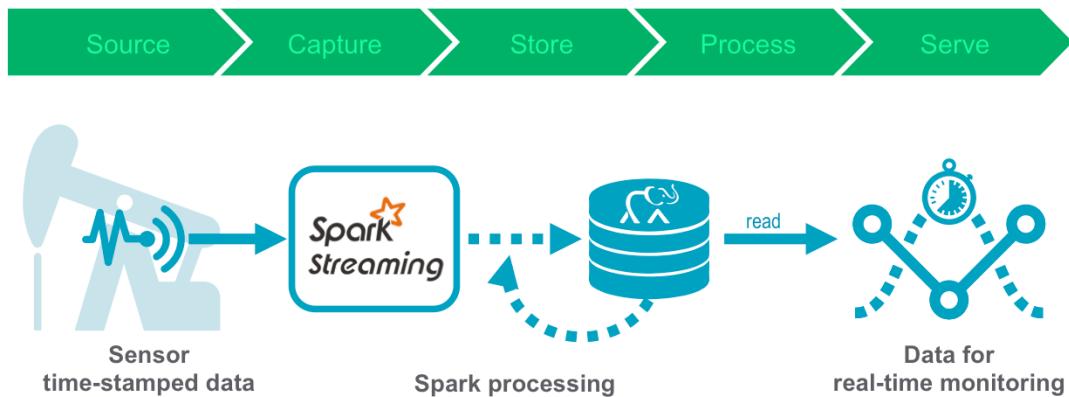
Learning Goals



- Describe Spark Streaming Architecture
- **Create DStreams and a Spark Streaming Application**
 - Define Use Case
 - Basic Steps
 - Save Data to HBase
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In this section, you will create DStreams and save the output to HBase tables. First, we will define our use case and application needs.

Use Case: Time Series Data

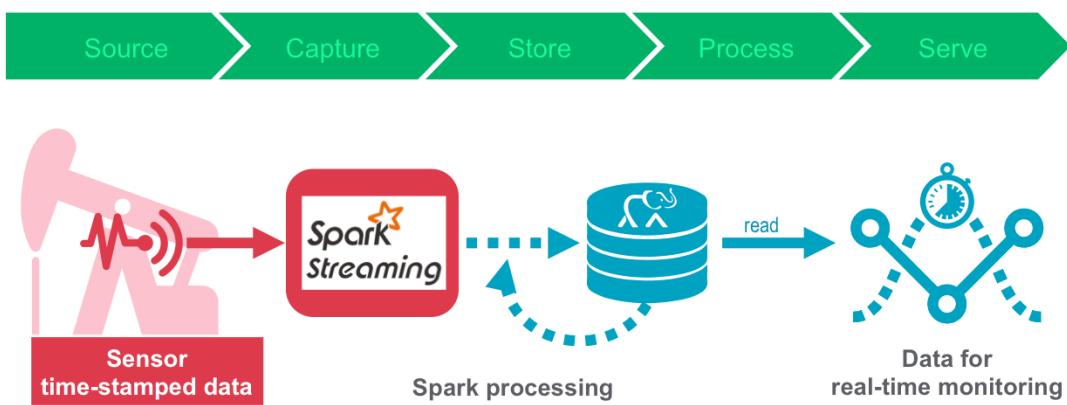


The example use case we will look at here is an application that monitors oil wells.

Sensors in oil rigs generate streaming data, which is processed by Spark and stored in HBase, for use by various analytical and reporting tools.

We want to store every single event in HBase as it streams in. We also want to filter for, and store alarms. Daily Spark processing will store aggregated summary statistics.

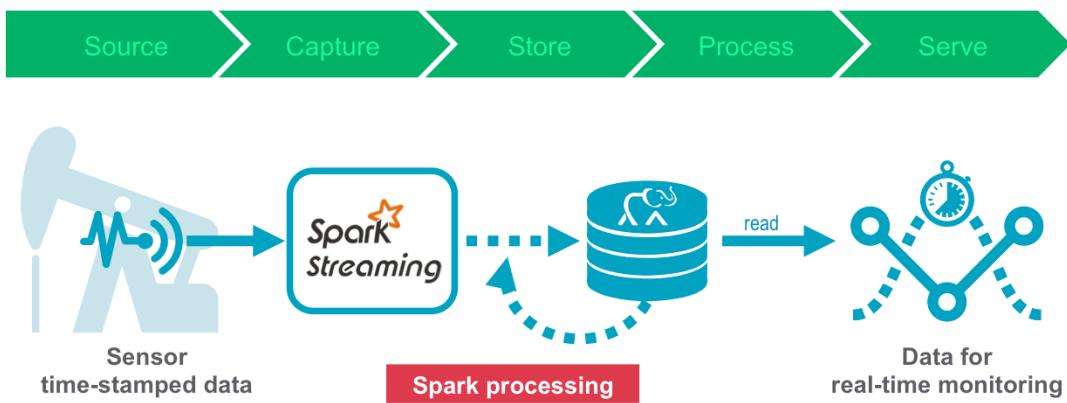
Use Case: Time Series Data



Our Spark Streaming example flow then, first reads streaming data...



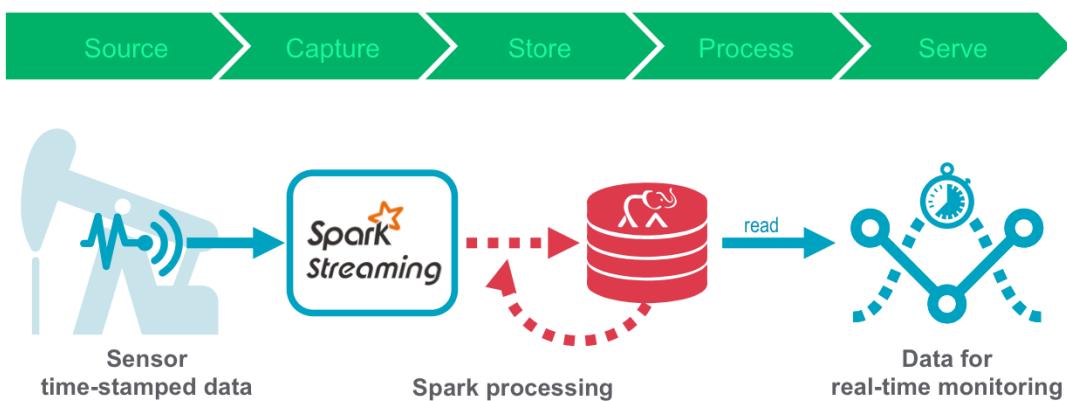
Use Case: Time Series Data



Processes the streaming data...

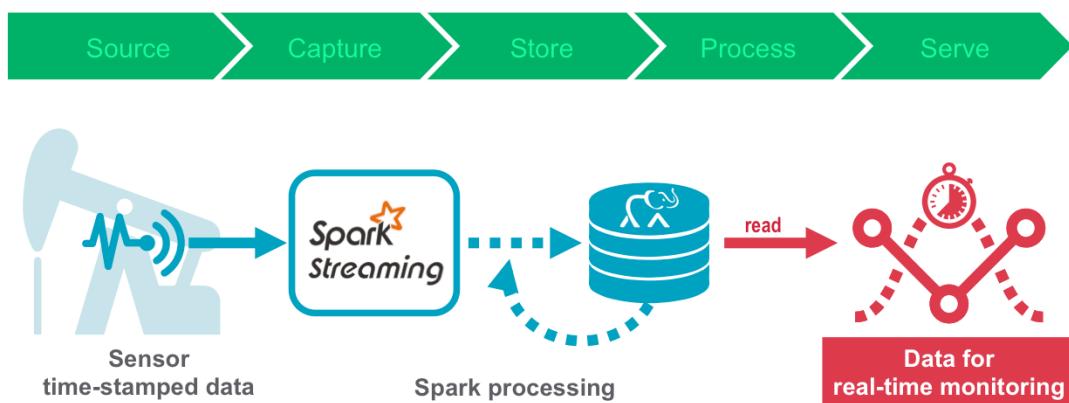


Use Case: Time Series Data



And writes the processed data to an HBase table.

Use Case: Time Series Data



Our non Streaming Spark code:

- Reads the HBase table data written by the streaming code
- Calculates daily summary statistics, and
- Writes summary statistics to the HBase table, Column Family stats

Convert Line of CSV Data to Sensor Object

```
sensordata.csv x
1 ResourceID,Date,Time,HZ,Displace,Flow,SedimentPPM,PressureLbs,ChlorinePPM
2 COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
3 COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
4 COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
```

```
case class Sensor(resid: String, date: String, time: String,
  hz: Double, disp: Double, flo: Double, sedPPM: Double,
  psi: Double, chlPPM: Double)

def parseSensor(str: String): Sensor = {
  val p = str.split(",")
  Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,
    p(6).toDouble, p(7).toDouble, p(8).toDouble)
}
```

The oil pump sensor data comes in as comma separated value files, or CSV, saved to a directory. Spark Streaming will monitor the directory and process any files created into that directory. Spark Streaming supports many different streaming data sources. For simplicity, in this example we will use the MapR file system.

Unlike other Hadoop distributions that only allow cluster data import, or import as a batch operation, MapR lets you mount the cluster itself via NFS, so that your applications can read and write data directly. MapR allows direct file modification with multiple concurrent reads and writes via POSIX semantics. With an NFS-mounted cluster, you can read and write data directly with standard tools, applications, and scripts.

This means that MapR-FS is really easy to use with Spark streaming by putting data files into a directory.



Convert Line of CSV Data to Sensor Object

```
sensordata.csv  ×  
1 ResourceID,Date,Time,HZ,Displace,Flow,SedimentPPM,PressureLbs,ChlorinePPM  
2 COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94  
3 COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79  
4 COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
```

```
case class Sensor(resid: String, date: String, time: String,  
                  hz: Double, disp: Double, flo: Double, sedPPM: Double,  
                  psi: Double, chlPPM: Double)  
  
def parseSensor(str: String): Sensor = {  
    val p = str.split(",")  
    Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,  
           p(6).toDouble, p(7).toDouble, p(8).toDouble)  
}
```

We use a Scala case class to define the Sensor schema corresponding to the sensor data CSV files...



Convert Line of CSV Data to Sensor Object

```
sensordata.csv      x
1 ResourceID,Date,Time,HZ,Displace,Flow,SedimentPPM,PressureLbs,ChlorinePPM
2 COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
3 COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
4 COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
```

```
case class Sensor(resid: String, date: String, time: String,
                  hz: Double, disp: Double, flo: Double, sedPPM: Double,
                  psi: Double, chlPPM: Double)

def parseSensor(str: String): Sensor = {
    val p = str.split(",")
    Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,
           p(6).toDouble, p(7).toDouble, p(8).toDouble)
}
```

...and a parseSensor function to parse the comma separated values into the sensor case class.



Learning Goals



Learning Goals



- Describe Spark Streaming Architecture
- **Create DStreams and a Spark Streaming Application**
 - Define Use Case
 - Basic Steps
 - Save Data to HBase
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In this section, you will create the DStreams.

Basic Steps for Spark Streaming Code

1. Initialize a Spark StreamingContext object
2. Using context, create a DStream
 - Represents streaming data from a source
 - 1. Apply transformations
 - Creates new DStreams
 - 2. And/or apply output operations
 - Persists or outputs data
3. Start receiving and processing data
 - Using `streamingContext.start()`
4. Wait for the processing to be stopped
 - Using `streamingContext.awaitTermination()`

Shown here are the basic steps for Spark Streaming code:

First, initialize a Spark StreamingContext object.

Using this context, create a Dstream, which represents streaming data from a source.
Apply transformations and/or output operations to the Dstreams.

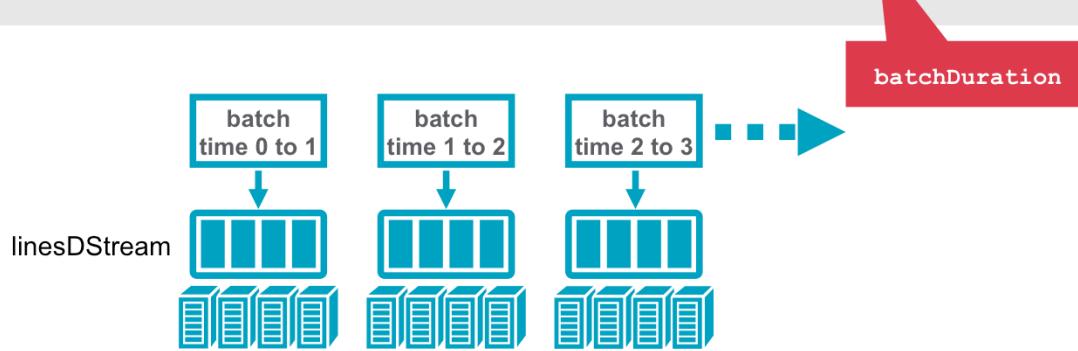
We can then start receiving data and processing it using `streamingContext.start()`.

And finally we wait for the processing to be stopped using
`streamingContext.awaitTermination()`

We will go through these steps showing code from our use case example.
Spark Streaming programs are best run as standalone applications built using Maven
or sbt. In the lab you will use the shell for simplicity, and build a streaming application.

Create a DStream

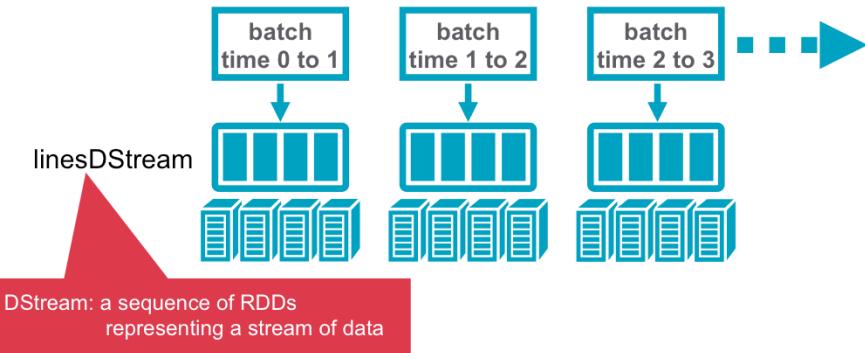
```
val ssc = new StreamingContext(sparkConf, Seconds(2))  
val linesDStream = ssc.textFileStream("/mapr/stream")
```



The first step is to create a [StreamingContext](#), which is the main entry point for streaming functionality. In this example we will use a 2 second [batch interval](#).

Create a DStream

```
val ssc = new StreamingContext(sparkConf, Seconds(2))  
val linesDStream = ssc.textFileStream("/mapr/stream")
```



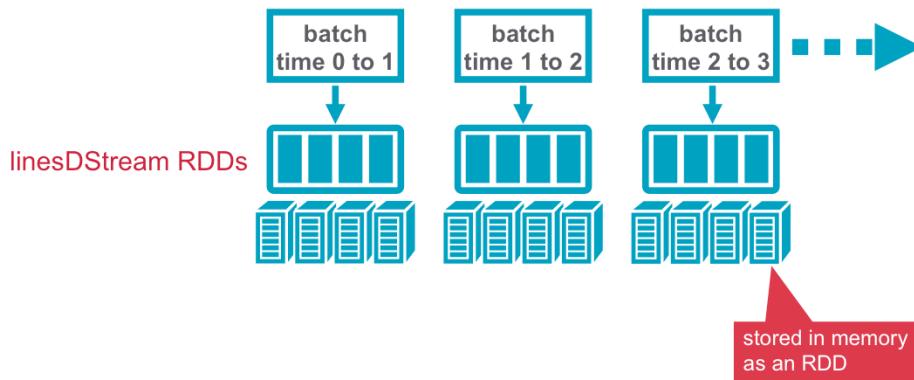
Using this context, we can create a DStream that represents streaming data from a source.

In this example we use the StreamingContext textFileStream method to create an input stream that monitors a Hadoop-compatible file system for new files, and processes any files created in that directory.

This ingestion type supports a workflow where new files are written to a landing directory and Spark Streaming is used to detect them, ingest them, and process the data. Only use this ingestion type with files that are moved or copied into a directory.

Create a DStream

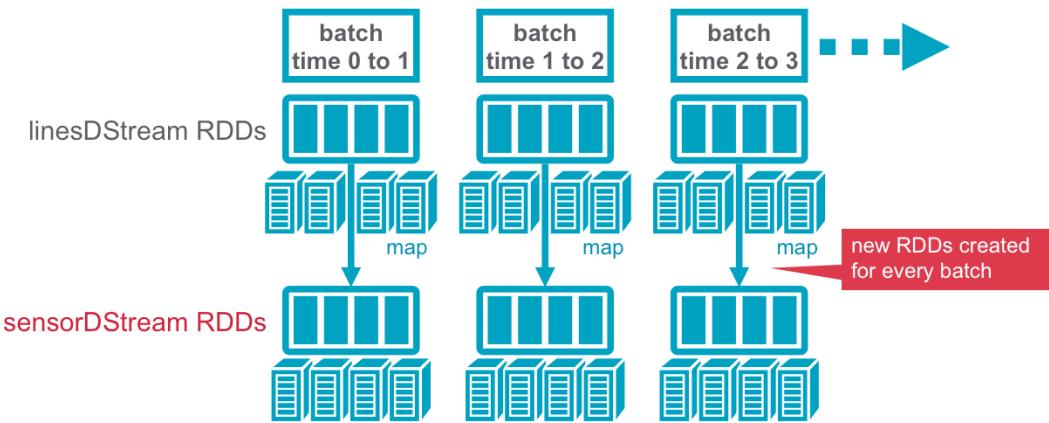
```
val ssc = new StreamingContext(sparkConf, Seconds(2))  
val linesDStream = ssc.textFileStream("/mapr/stream")
```



linesDStream represents the stream of incoming data, where each record is a line of text. Internally a DStream is a sequence of RDDs, one RDD per batch interval.

Process DStream

```
val linesDStream = ssc.textFileStream("directory path")
val sensorDStream = linesDStream.map(parseSensor)
```



Next we parse the lines of data into Sensor objects, with the map operation on the linesDStream.

The map operation applies the Sensor.parseSensor function on the RDDs in the linesDStream, resulting in RDDs of Sensor objects.

Any operation applied on a DStream translates to operations on the underlying RDDs. the map operation is applied on each RDD in the linesDStream to generate the sensorDStream RDDs.

Process DStream

```
// for Each RDD
sensorDStream.foreachRDD { rdd =>
    // filter sensor data for low psi
    val alertRDD = sensorRDD.filter(sensor => sensor.psi < 5.0)
    . . .
}
```

Next we use the DStream [foreachRDD](#) method to apply processing to each RDD in this DStream. We filter the sensor objects for low PSI to create an RDD of alert sensor objects.



Start Receiving Data

```
sensorDStream.foreachRDD { rdd =>
    . . .
}
// Start the computation
ssc.start()
// Wait for the computation to terminate
ssc.awaitTermination()
```

To start receiving data, we must explicitly call `start()` on the `StreamingContext`, then call `awaitTermination` to wait for the streaming computation to finish.



Streaming Application Output

```
-----
Time: 1452886488000 ms

COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
COHUTTA,3/10/14,1:05,9.56,1.734,883,1.35,99,0.68
COHUTTA,3/10/14,1:06,9.74,1.736,884,1.27,92,0.73
COHUTTA,3/10/14,1:08,10.44,1.737,885,1.34,93,1.54
COHUTTA,3/10/14,1:09,9.83,1.738,885,0.06,76,1.44
COHUTTA,3/10/14,1:11,10.49,1.739,886,1.51,81,1.83
COHUTTA,3/10/14,1:12,9.79,1.739,886,1.74,82,1.91
COHUTTA,3/10/14,1:13,10.02,1.739,886,1.24,86,1.79
...
low pressure alert
Sensor(NANTAHALLA,3/13/14,2:05,0.0,0.0,0.0,1.73,0.0,1.51)
Sensor(NANTAHALLA,3/13/14,2:07,0.0,0.0,0.0,1.21,0.0,1.51)
-----
Time: 1452886490000 ms
-----
```

The output from our application will show the name of the oil rig, and streamed data.



Knowledge Check



Knowledge Check



Using the Spark Streaming code here:

```
val ssc = new StreamingContext(sparkConf, Seconds(30))
val linesDStream = ssc.textFileStream("/mapr/source/webstream")
```

- **What is the batch interval?**
- **What method is used to create an input stream?**
- **Where is the streaming data saved?**

Batch interval = 30 seconds

ssc.textFileStream is the method used to create the input stream

Streaming data is saved into the directory: /mapr/source/webstream

Learning Goals



Learning Goals



- Describe Spark Streaming Architecture
- **Create DStreams and a Spark Streaming Application**
 - Define Use Case
 - Basic Steps
 - Save Data to HBase
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In this section, you will save the process stream data into an HBase table.

Configure to Write to HBase

```
// set JobConfiguration variables for writing to HBase
val jobConfig: JobConf = new JobConf(conf, this.getClass)
jobConfig.setOutputFormat(classOf[TableOutputFormat])
jobConfig.set(TableOutputFormat.OUTPUT_TABLE, tableName)
```

We register the DataFrame as a table, which allows us to use it in subsequent SQL statements.

Now we can inspect the data.



Schema

- All events stored, CF data could be set to expire data
- Filtered alerts put in CF alerts
- Daily summaries put in CF stats

Row key	CF data			CF alerts		CF stats		
	hz	...	psi	psi	...	hz_avg	...	psi_min
COHUTTA_3/10/14_1:01	10.37		84	0				
COHUTTA_3/10/14						10		0

Row Key contains oil pump name, date, and a time stamp

The HBase table schema for the streaming data is as follows:

The row key is a composite of the oil pump name, date, and a time stamp

Schema

- All events stored, CF data could be set to expire data
- Filtered alerts put in CF alerts
- Daily summaries put in CF stats

Row key	CF data			CF alerts		CF stats		
	hz	...	psi	psi	...	hz_avg	...	psi_min
COHUTTA_3/10/14_1:01	10.37		84	0				
COHUTTA_3/10/14					10		0	

The Column Family “data” contains columns corresponding to the input data fields.

The Column Family “alerts” contains columns corresponding to any filters for alarm values.

Note that the data and alert column families could be set to expire values after a certain amount of time.



Schema

- All events stored, CF data could be set to expire data
- Filtered alerts put in CF alerts
- Daily summaries put in CF stats

Row key	CF data			CF alerts		CF stats		
	hz	...	psi	psi	...	hz_avg	...	psi_min
COHUTTA 3/10/14 1:01	10.37		84	0				
COHUTTA_3/10/14						10		0

The Schema for the daily statistics summary rollups is as: the composite row key of the pump name and date, and then the Column Family “stats”



Function to Convert Sensor Data to HBase Put Object

```
// Convert a row of sensor object data to an HBase put object
def convertToPut(sensor: Sensor): (ImmutableBytesWritable, Put) = {
    val put = new Put(Bytes.toBytes(rowkey))
    // add column values to put object
    . . .
    put.add(Bytes.toBytes("data"), Bytes.toBytes("psi"),
            Bytes.toBytes(sensor.psi))
    return (new ImmutableBytesWritable(Bytes.toBytes(rowkey)), put)
}
```

This function converts a Sensor object into an HBase Put object, which is used to insert a row into HBase.



Save to HBase

```
// for Each RDD parse into a sensor object filter
sensorDStream.foreachRDD { rdd =>
    . . .
    // convert alert to put object write to HBase alerts
    rdd.map(Sensor.convertToPutAlert)
        .saveAsHadoopDataset(jobConfig)
}
```

We use the DStream [foreachRDD](#) method to apply processing to each RDD in the DStream.

We filter the sensor objects for low PSI values, to create alerts, and then convert the data to HBase Put objects using the convertToPutAlert function.

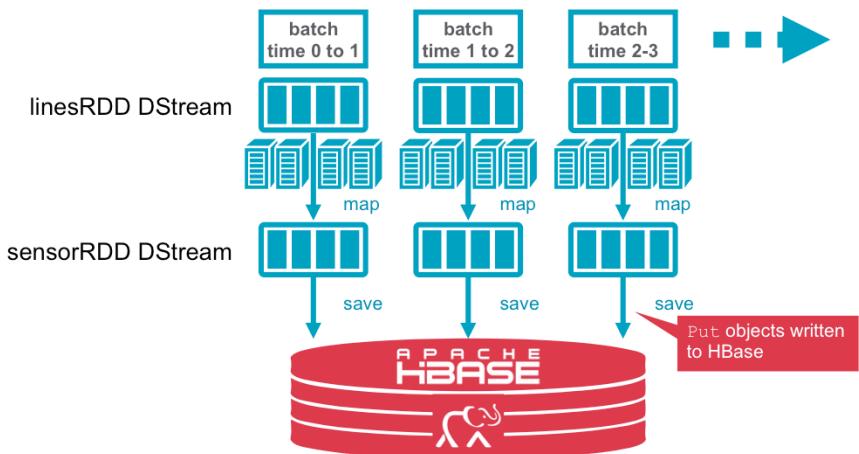
Next we write the sensor and alert data to HBase using the PairRDDFunctions [saveAsHadoopDataset](#) method. This outputs the RDD to any Hadoop-supported storage system using a Hadoop Configuration object for that storage system.



Save to HBase

```
rdd.map(Sensor.convertToPut).saveAsHadoopDataset(jobConfig)
```

output operation: persist data to external storage



MAPR Academy

© 2016 MapR Technologies 8-48

The sensorRDD objects are converted to Put objects, and then written to HBase, using the [saveAsHadoopDataset](#) method.

Learning Goals



Learning Goals



- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- **Apply Operations on DStreams**
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

This section you will learn how to apply operations to your DStreams.

Operations on DStreams

- What is the pump vendor and maintenance information for sensors with low pressure alerts?
- What is the Max, Min, and Average for sensor attributes?

Now that we have our input data stream, we would like to answer some questions such as:

What is the pump vendor and maintenance information for sensors with low pressure alerts?

What is the max, min, and average for sensor attributes?

To answer these questions, we will use operations on the DStream we just created.

DataFrame and SQL Operations

```
// put pump vendor and maintenance data in temp table
sc.textFile("vendor.csv").map(parsePump).toDF().registerTempTable("pump")
sc.textFile("maint.csv").map(parseMaint).toDF().registerTempTable("maint")
// for Each RDD
sensorDStream.foreachRDD { rdd =>
    val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
    rdd.filter(sensor => sensor.psi < 5.0).toDF.registerTempTable("alert")
    val alertpumpmaint = sqlContext.sql("select s.resid, s.date, s.psi,
        p.pumpType, p.vendor, m.eventDate, m.technician from alert s join
        pump p on s.resid = p.resid join maint m on p.resid=m.resid")
    alertpumpmaint.show
}
```

Here we join the filtered alert data with pump vendor and maintenance information, which was read in and cached before streaming. Each RDD is converted into a DataFrame, registered as a temporary table and then queried using SQL.

This Query answers our first question:

What is the pump vendor and maintenance information for sensors with low pressure alerts?



Streaming Application Output

```
alert pump maintenance data
resid date psi pumpType purchaseDate serviceDate vendor eventDate technician description
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 10/2/09 T. LaBou Install
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 10/5/09 W.Stevens Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 11/24/09 W.Stevens Tighten Mounts
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 6/10/10 T. LaBou Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 1/7/11 T. LaBou Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 9/30/11 W.Stevens Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 10/3/11 T. LaBou Bearing Seal
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 11/5/11 D.Pitre Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 5/22/12 D.Pitre Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 12/15/12 W.Stevens Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 6/18/13 T. LaBou Vane clearance ad...
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 7/11/13 W.Stevens Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 2/5/14 D.Pitre Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 3/14/14 D.Pitre Shutdown Main Fee...
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 10/2/09 T. LaBou Install
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 10/5/09 W.Stevens Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 11/24/09 W.Stevens Tighten Mounts
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 6/10/10 T. LaBou Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 1/7/11 T. LaBou Inspect
LAGNAPPE 3/14/14 0.0 HYDROPUMP 5/25/08 9/26/09 GENPUMP 9/30/11 W.Stevens Inspect
-----
Time: 1452887522000 ms
```

MAPR Academy © 2016 MapR Technologies 8-53

Here we see some sample output, answering our first question.

DataFrame and SQL Operations

```
// for Each RDD
sensorDStream.foreachRDD { rdd =>
    val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
    rdd.toDF().registerTempTable("sensor")
    val res = sqlContext.sql( "SELECT resid, date,
        max(hz) as maxhz, min(hz) as minhz, avg(hz) as avghz,
        max(disp) as maxdisp, min(disp) as mindisp, avg(disp) as avgdisp,
        max(flo) as maxflo, min(flo) as minflo, avg(flo) as avgflo,
        max(psi) as maxpsi, min(psi) as minpsi, avg(psi) as avgpsi
        FROM sensor GROUP BY resid,date")
    res.show()
}
```

Our second question:

What is the max, min, and average for sensor attributes?

Is addressed by the query shown here.



Streaming Application Output

sensor max, min, averages												
resid	date	maxhz	minhz	avghz	maxdisp	mindisp	avgdisp	maxflo	minflo	avgflo		
LAGNAPPE	3/12/14	10.5	9.5	9.988799582463459	3.235	1.623	2.4714206680584563	1570.0	788.0	1199.1022964509395		
CHER	3/13/14	10.5	9.5	10.005271398747391	3.552	1.857	2.732156576200417	1670.0	873.0	1284.660751565762		
BBKING	3/13/14	10.5	9.5	9.996833402922755	1.58	0.902	1.26099164926931	1822.0	1041.0	1454.0240083507306		
MOJO	3/12/14	10.5	9.5	10.006482254697294	3.589	2.143	2.888004175365341	1899.0	1134.0	1528.2724425887266		
CARGO	3/10/14	10.5	9.5	9.998507306889353	3.752	1.903	2.8525417536534423	1533.0	778.0	1165.5302713987473		
LAGNAPPE	3/11/14	10.5	9.5	10.009540709812102	3.092	1.454	2.307491649269313	1500.0	706.0	1119.5647181628392		
CHER	3/12/14	10.5	9.5	10.008256784968692	3.443	1.728	2.6184937369519825	1619.0	812.0	1231.230688935282		
BBKING	3/12/14	10.5	9.5	10.006795407098123	1.556	0.862	1.2138110647181624	1794.0	994.0	1399.608559498956		
MOJO	3/11/14	10.5	9.5	10.0029331941545	3.513	1.979	2.8009843423799587	1859.0	1047.0	1482.2160751565762		
LAGNAPPE	3/10/14	10.5	9.5	10.00329853862213	3.116	1.453	2.349840292275576	1512.0	705.0	1140.1022964509395		
CHER	3/11/14	10.5	9.5	9.990396659707717	3.325	1.691	2.545035490605431	1564.0	795.0	1196.6837160751566		
BBKING	3/11/14	10.5	9.5	9.997348643006282	1.49	0.821	1.1701722338204592	1718.0	947.0	1349.2849686847599		
MOJO	3/10/14	10.5	9.5	9.999457202505194	3.345	1.828	2.6188089770354868	1770.0	967.0	1385.8131524008352		
CHER	3/10/14	10.5	9.5	9.998726513569954	3.172	1.653	2.4233079331941516	1492.0	777.0	1139.4488517745303		
BBKING	3/10/14	10.5	9.5	10.001409185803748	1.502	0.821	1.18567223382046	1732.0	947.0	1367.1711899791233		
THERMALITO	3/14/14	10.5	9.5	9.983862212943635	3.784	2.135	3.0065960334029254	1680.0	948.0	1335.1002087682673		
THERMALITO	3/13/14	10.5	9.5	9.999311064718169	3.896	2.116	3.069195198329852	1730.0	940.0	1362.910229645094		
ANDOUILLE	3/14/14	10.5	9.5	10.011388308977041	2.146	1.139	1.6609373695198306	1592.0	845.0	1232.2296450939457		
THERMALITO	3/12/14	10.5	9.5	10.007526096033398	3.823	1.986	2.9294561586638808	1697.0	882.0	1300.8622129436326		
ANDOUILLE	3/13/14	10.5	9.5	9.9904384133618	2.174	1.104	1.656331941544886	1613.0	819.0	1228.8371607515658		

And output from that query shows the max, min, and average output from our sensors.



Key Concepts

- **Data sources:**
 - File based: HDFS
 - Network based: TCP sockets
 - Twitter, Kafka, Flume, ZeroMQ, Akka Actor
- Transformations: create new DStream
 - Standard RDD operations: `map`, `countByValue`, `reduce`, `join`, ...
 - Stateful operations: `UpdateStateByKey(function)`, `countByValueAndWindow`, ...
- Output operations: trigger computation
 - `print` – prints first 10 elements
 - `saveAsObjectFile`, `saveAsTextFiles`, `saveAsHadoopFiles` – save to HDFS
 - `foreachRDD` – do anything with each batch of RDDs

Spark Streaming supports various input sources, including file-based sources and network-based sources such as socket-based sources, the Twitter API stream, Akka actors, or message queues and distributed stream and log transfer frameworks, such Flume, Kafka, and Amazon Kinesis.



Key Concepts

- Data sources:
 - File based: HDFS
 - Network based: TCP sockets
 - Twitter, Kafka, Flume, ZeroMQ, Akka Actor
- **Transformations: create new DStream**
 - Standard RDD operations: `map`, `countByValue`, `reduce`, `join`, ...
 - Stateful operations: `UpdateStateByKey(function)`, `countByValueAndWindow`, ...
- Output operations: trigger computation
 - `print` – prints first 10 elements
 - `saveAsObjectFile`, `saveAsTextFiles`, `saveAsHadoopFiles` – save to HDFS
 - `foreachRDD` – do anything with each batch of RDDs

Spark Streaming provides a set of transformations available on DStreams. These transformations are similar to those available on RDDs. They include `map`, `flatMap`, `filter`, `join`, and `reduceByKey`. Streaming also provides operators such as `reduce` and `count`.

These operators return a DStream made up of a single element. Unlike the `reduce` and `count` operators on RDDs, these do not trigger computation on DStreams. They are not actions, they return another DStream.

Stateful transformations maintain state across batches, they use data or intermediate results from previous batches to compute the results of the current batch. They include transformations based on sliding windows and on tracking state across time. `updateStateByKey()` is used to track state across events for each key. For example this could be used to maintain stats (`state`) by a key `accessLogDStream.reduceByKey(SUM_REDUCER)`.

Streaming transformations are applied to each RDD in the DStream, which, in turn, applies the transformation to the elements of the RDD.

Want to see more? Find a full list of transformations at: <http://spark.apache.org/docs/latest/streaming-programming-guide.html>



Key Concepts

- Data sources:
 - File based: HDFS
 - Network based: TCP sockets
 - Twitter, Kafka, Flume, ZeroMQ, Akka Actor
- Transformations: create new DStream
 - Standard RDD operations: `map`, `countByValue`, `reduce`, `join`, ...
 - Stateful operations: `UpdateStateByKey(function)`, `countByValueAndWindow`, ...
- **Output operations: trigger computation**
 - `print` – prints first 10 elements
 - `saveAsObjectFile`, `saveAsTextFiles`, `saveAsHadoopFiles` – save to HDFS
 - `foreachRDD` – do anything with each batch of RDDs

Actions are output operators that, when invoked, trigger computation on the DStream. They include:

`print`, which prints the first 10 elements of each batch to the console, and is typically used for debugging and testing.

The `saveAsObjectFile`, `saveAsTextFiles`, and `saveAsHadoopFiles` functions output each batch to a Hadoop-compatible file system.

And the `foreachRDD` operator applies any arbitrary processing to the RDDs within each batch of a DStream.



Knowledge Check



Knowledge Check



Indicate whether the statements below are TRUE or FALSE:

- Transformations on a DStream return another DStream
- Stateful transformations maintain their state across batches
- Transformations trigger computations
- `foreachRDD` will apply processing to the RDDs within each batch of a DStream
- DStreams can only be made from streaming sources

True

True

False - output actions trigger actions. Transformations return another DStream.

True

False – DStreams can be made from file-based sources and network-based sources such as socket-based sources.

Learning Goals



Learning Goals



- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- **Define Windowed Operations**
- Describe How Streaming Applications are Fault Tolerant

This section discusses windowed operations.

Sliding Windows

- Can compute results across longer time period
- Example – want oil pressure data across last 12 intervals (60 seconds) computed after every 6 intervals (30 seconds)
 - Create a sliding window of the last 60 seconds
 - Computed every 30 seconds

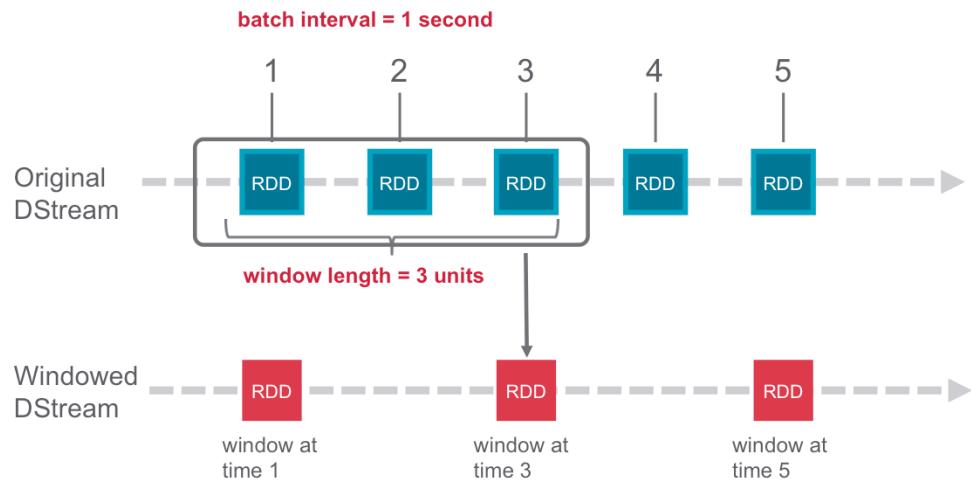
With window operations you can apply transformations over a sliding window of data. This allows us to perform computations over a longer time period than that specified in the StreamingContext batch Interval, by combining results from multiple batches.

For example, our data is coming in at 5-second intervals. The DStream computations are performed every 5 seconds across the data in that 5-second interval.

Say, we want to compute the oil pressure at a specific station every 30 seconds over the last 60 seconds. We create a sliding window of the last 60 seconds and compute every 30 seconds.

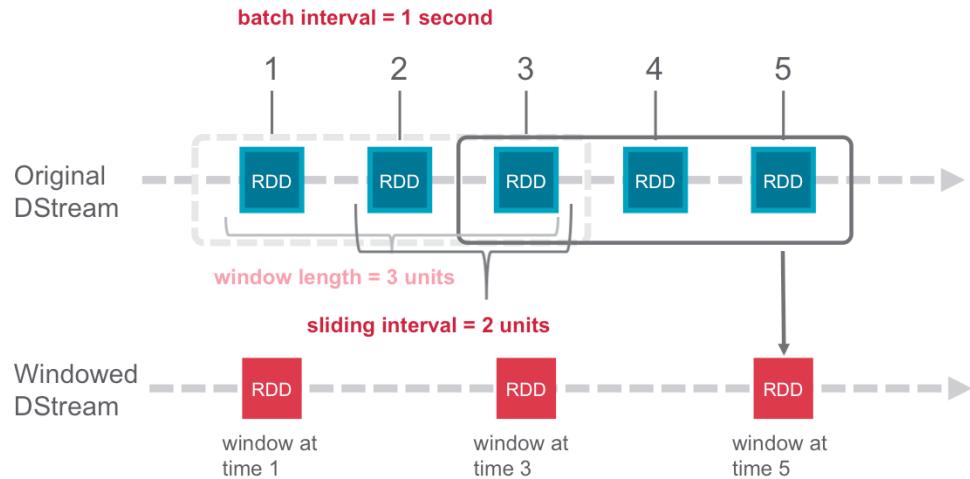


Windowed Computations



In this illustration, the original stream is coming in at one second intervals. The length of the sliding window is specified by the window length, in this case 3 units.

Windowed Computations



The window slides over the source DStream at the specified sliding interval, in this case 2 units.

Both the window length and the sliding interval must be multiples of the batch interval of the source DStream, which in this case is 1 second. When the window slides over the source DStream, all the RDDs that fall in that window are combined. The operation is applied to the combined RDDs resulting in RDDs in the windowed stream.

All window operations require two parameters:

The window length, is the duration of the window. In this example the window length is 3 units.

And the sliding interval, which is the interval at which the operation window is performed. In this example the sliding interval is 2 units.

Again, both of these parameters must be multiples of the batch interval of the original DStream.

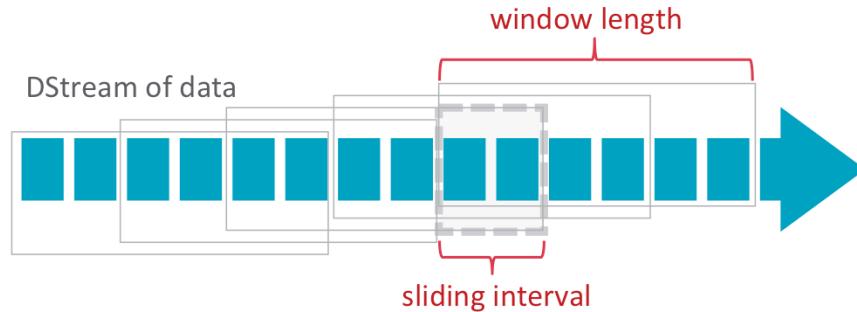
Window-based Transformations

```
val windowedWordCounts = pairs.reduceByKeyAndWindow(  
    (a:Int,b:Int) => (a + b), Seconds(12), Seconds(4))
```

sliding window operation

window length

sliding interval



For example we want to generate word counts every 4 seconds, over the last 6 seconds of data. To do this, we apply the reduceByKey operation on the DStream of paired 2-tuples (word and the number 1), over the last 6 seconds of data using the operation reduceByKeyAndWindow.

Window Operations

Window Operation	Description
<code>window(windowLength, slideInterval)</code>	Returns new DStream computed based on windowed batches of source DStream.
<code>countByWindow(windowLength, slideInterval)</code>	Returns a sliding window count of elements in the stream
<code>reduceByWindow(func, windowLength, slideInterval)</code>	Returns a new single-element stream created by aggregating elements over sliding interval using <code>func</code> .
<code>reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])</code>	Returns a new DStream of (K,V) pairs from DStream of (K,V) pairs; aggregates using given reduce function <code>func</code> over batches of sliding window.
<code>countByValueAndWindow(windowLength, slideInterval, [numTasks])</code>	Returns new DStream of (K,V) pairs where value of each key is its frequency within a sliding window; it acts on DStreams of (K,V) pairs

Output operations are used to push the data in DStreams to an external system such as a database or file system. Some commonly used output operations are listed here.



Window Operations on DStreams

With a windowed stream :

- What is the count of sensor events by pump ID?
- What is the Max, Min, and Average for PSI?

We would like to answer some questions such as:

With a windowed stream of 6 seconds of data, every 2 seconds:

What is the count of sensor events by pump ID?

What is the Max, Min, and Average pump PSI?

To answer these questions, we will use operations on a windowed stream.



DataFrame and SQL Operations

```
// Windowed 6 seconds of data, every 2 seconds
sensorDStream.window(Seconds(6), Seconds(2))
// for Each RDD
sensorDStream.foreachRDD { rdd =>
    val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
    rdd.toDF().registerTempTable("sensor")
    val res = sqlContext.sql("SELECT resid, date, count(resid) as total
        FROM sensor GROUP BY resid, date")

    res.show
}
```

The code here performs operations on 6 seconds worth of data, on a 2 second window.



DataFrame and SQL Operations

```
// Windowed 6 seconds of data, every 2 seconds
sensorDStream.window(Seconds(6), Seconds(2))
// for Each RDD
sensorDStream.foreachRDD { rdd =>
    val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
    rdd.toDF().registerTempTable("sensor")
    val res = sqlContext.sql("SELECT resid, date, count(resid) as total
        FROM sensor GROUP BY resid, date")

    res.show
}
```

We answer our first question:

What is the count of sensor events by pump ID?

By counting the ID on each sensor DStream RDD.



Streaming Application Output

resid	date	total
LAGNAPPE	3/12/14	958
CHER	3/13/14	958
BBKING	3/13/14	958
MOJO	3/12/14	958
CARGO	3/10/14	958
LAGNAPPE	3/11/14	958
CHER	3/12/14	958
BBKING	3/12/14	958
MOJO	3/11/14	958
LAGNAPPE	3/10/14	958
CHER	3/11/14	958
BBKING	3/11/14	958
MOJO	3/10/14	958
CHER	3/10/14	958
BBKING	3/10/14	958
THERMALITO	3/14/14	958
THERMALITO	3/13/14	958
ANDOUILLE	3/14/14	958
THERMALITO	3/12/14	958
ANDOUILLE	3/13/14	958

This output shows the answer for our first question.

DataFrame and SQL Operations

```
// Windowed 6 seconds of data, every 2 seconds
sensorDStream.window(Seconds(6), Seconds(2))
// for Each RDD
sensorDStream.foreachRDD { rdd =>
    val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
    rdd.toDF().registerTempTable("sensor")
    val res = sqlContext.sql("SELECT resid, date, max(psi) as maxpsi,
        min(psi) as minpsi, avg(psi) as avgpsi
        FROM sensor GROUP BY resid,date")
    res.show
}
```

To answer our second question:

What is the Max, Min, and Average pump PSI?

We use the same data window, and then collect the PSI data on each sensor RDD.



Streaming Application Output

sensor max, min, averages				
resid	date	maxpsi	minpsi	avgpsi
LAGNAPPE	3/12/14	100.0	75.0	87.30793319415449
CHER	3/13/14	100.0	75.0	88.16597077244259
BBKING	3/13/14	100.0	75.0	87.73903966597078
MOJO	3/12/14	100.0	75.0	87.55219206680584
CARGO	3/10/14	100.0	75.0	87.39352818371607
LAGNAPPE	3/11/14	100.0	75.0	87.37473903966597
CHER	3/12/14	100.0	75.0	87.13883089770354
BBKING	3/12/14	100.0	75.0	87.79749478079331
MOJO	3/11/14	100.0	75.0	87.74739039665971
LAGNAPPE	3/10/14	100.0	75.0	87.60647181628393
CHER	3/11/14	100.0	75.0	87.74008350730689
BBKING	3/11/14	100.0	75.0	87.71398747390397
MOJO	3/10/14	100.0	75.0	87.20876826722338
CHER	3/10/14	100.0	75.0	87.44885177453027
BBKING	3/10/14	100.0	75.0	87.20146137787056
THERMALITO	3/14/14	100.0	75.0	87.48225469728601
THERMALITO	3/13/14	100.0	75.0	87.45093945720251
ANDOUILLE	3/14/14	100.0	75.0	87.7223382045929
THERMALITO	3/12/14	100.0	75.0	87.39770354906054
ANDOUILLE	3/13/14	100.0	75.0	87.78496868475992

The output from our application gives us the requested PSI data for each pump.

Class Discussion



Class Discussion



What are some scenarios where you would find it useful to use windowed operations?

Learning Goals



Learning Goals



- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- Define Windowed Operations
- **Describe How Streaming Applications are Fault Tolerant**

This section describes what makes Spark streaming applications fault tolerant.

Fault Tolerance in Spark RDDs

- RDD is immutable
- Each RDD remembers lineage
- If RDD partition is lost due to worker node failure, partition recomputed
- Data in final transformed RDD always the same
- Data comes from fault-tolerant systems → RDDs from fault-tolerant data are also fault tolerant

As a review, an RDD is an immutable, distributed dataset that is deterministically re-computable.

Each RDD remembers its lineage of operations.

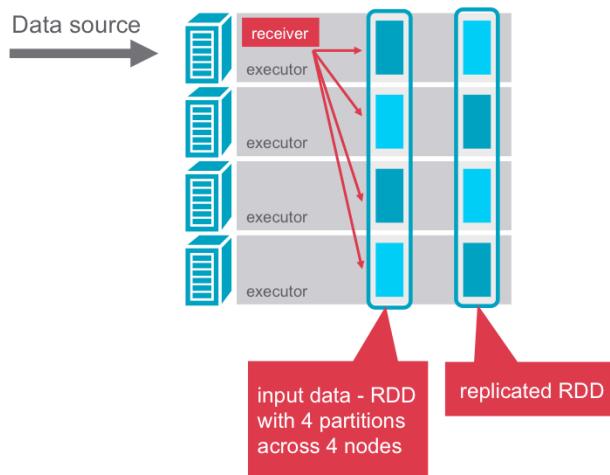
If any RDD partition is lost due to worker node failure, the partition can be recomputed from the original data using the lineage. The data in the final transformed RDD is always the same provided the RDD transformation are deterministic.

Since RDDs are operating on data from fault tolerant systems such as HDFS, S3 and MapR, the RDDs generated from fault tolerant data are also fault tolerant.

This is not the case for Spark Streaming, however, since data is received over the network, except when using fileStream. Next we will see how Spark achieves the same fault-tolerance with streaming.



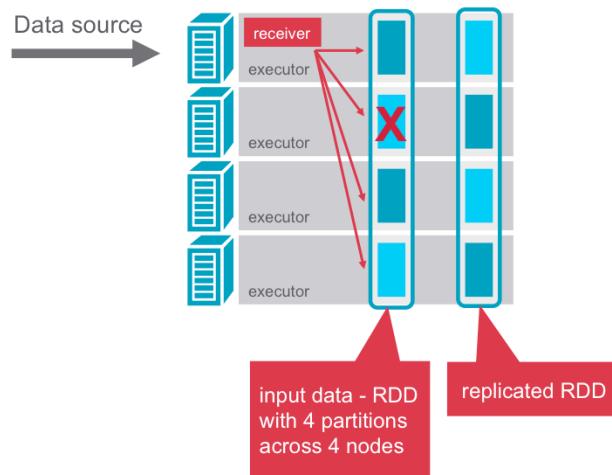
Fault Tolerance in Spark Streaming



Spark Streaming launches receivers within an executor for each input source. The receiver receives input data that is saved as RDDs and then replicated to other executors for fault tolerance. The default replication factor is 2.

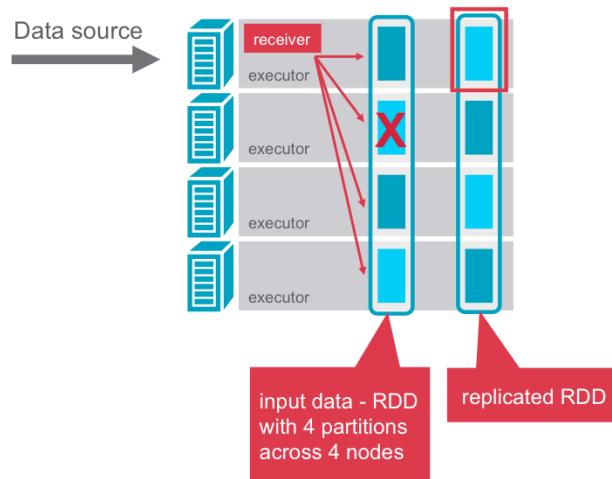
The data is stored in memory of the executors as cached RDDs. In this example, the input data source is partitioned by the Spark receiver into a DStream of four RDDs and the RDD partitions are replicated to different executors.

Fault Tolerance in Spark Streaming



When receiving data from network-based sources, you can have a worker node fail.

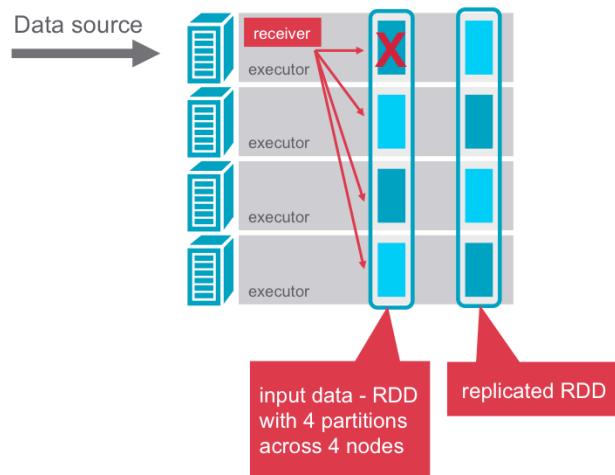
Fault Tolerance in Spark Streaming



Since the data is replicated in memory on the worker node, in the event of failure of a single worker node, the data exists on another node.

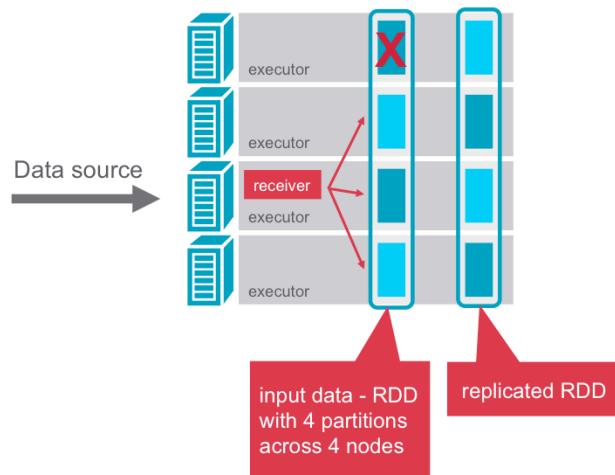
In this example, if the second node fails, there is a replica of the data on node 1.

Fault Tolerance in Spark Streaming



If instead the node with the receiver fails, then there may be a loss of the data that was received by the system but not yet replicated to other nodes.

Fault Tolerance in Spark Streaming



In this case the receiver will be started on a different node and it will continue to receive data.

Fault Tolerance in Spark RDDs – Write Ahead Logs

- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors

Spark also supports write ahead logs for Spark Streaming, to improve the recovery mechanism.

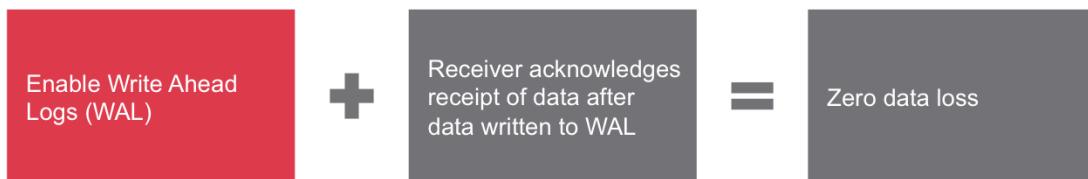
Sources like Flume, MapR Streams and Kafka use receivers to receive data.

The received data is stored in the executor memory, and the driver runs tasks on executors.



Fault Tolerance in Spark RDDs – Write Ahead Logs

- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors



When write ahead logs is enabled, all of the received data is saved to log files in a fault tolerant system. This makes the received data durable in the event of any failure in Spark Streaming.

Fault Tolerance in Spark RDDs – Write Ahead Logs

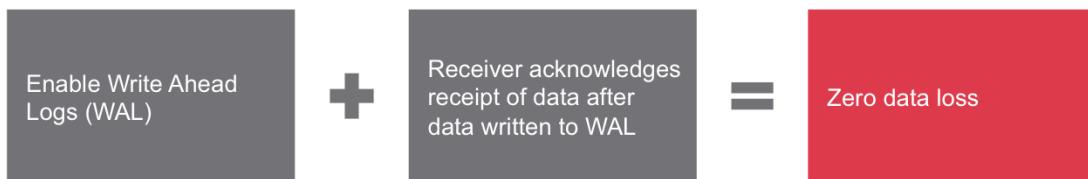
- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors



Furthermore, if the receiver acknowledges the receipt of the data after it has been written to the WAL, then in the event of a failure, the data that is buffered but not saved can be resent by the source once the driver is restarted.

Fault Tolerance in Spark RDDs – Write Ahead Logs

- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors



These two factors provide zero data loss.

Refer to the Spark documentation on how to enable and configure write ahead logs for Spark Streaming.

Checkpointing

- Provides fault tolerance for driver
- Periodically saves data to fault tolerant system
- Two types of checkpointing:
 - Metadata – for recovery from driver failures
 - Data checkpointing – for basic functioning if using stateful transformations
- Enable checkpointing
 - `ssc.checkpoint("hdfs://...")`

For a streaming application to be available 24/7, and be resilient to failures, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system for recovery.

There are two types of checkpointing- metadata checkpointing and data checkpointing.

Metadata checkpointing is for recovery from driver failures.

Data checkpointing is needed for basic functioning if using stateful transformations.

You can enable checkpointing by calling the `checkpoint` method on the `StreamingContext`.

If you use sliding windows, you need to enable checkpointing.



Knowledge Check



Knowledge Check



Spark Streaming uses several techniques to achieve fault tolerance on streaming data. Match the techniques listed below with the way in which they help ensure fault tolerance.

Data Replication

Writes data to a fault tolerant system, and sends acknowledgement of receipt, to protect against receiver failure

Write Ahead Logs

Save operational and RDD data to a fault tolerant system for recovery from a driver failure

Checkpointing

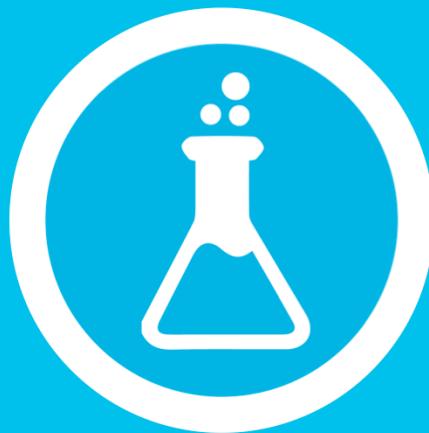
Saves data on multiple nodes for recovery should a single node fail

Data Replication - Saves data on multiple nodes for recovery should a single node fail

Write Ahead Logs - Writes data to a fault tolerant system, and sends acknowledgement of receipt, to protect against receiver failure

Checkpointing - Save operational and RDD data to a fault tolerant system for recovery from a driver failure

Lab 8: Create a Streaming Application



In these activities, you will create a Spark Streaming application. There is also a follow up lab where you save Spark Streaming data to an HBase table.



Next Steps

DEV362 – Create Data Pipelines With Apache Spark

Lesson 9: Use GraphX to Analyze Flight Data

Congratulations! You have completed Lesson 8.





DEV362 – Create Data Pipelines With Apache Spark

Lesson 9: Use Apache Spark GraphX

Welcome to DEV 362 lesson 9, Use Apache Spark GraphX.



Learning Goals



Learning Goals



- Describe GraphX
- Define Regular, Directed, and Property Graphs
- Create a Property Graph
- Perform Operations on Graphs

When you have finished with this lesson, you will be able to:

- Describe GraphX
- Define a regular, directed and property graph
- Create a property graph and
- Perform operations on graphs

Learning Goals



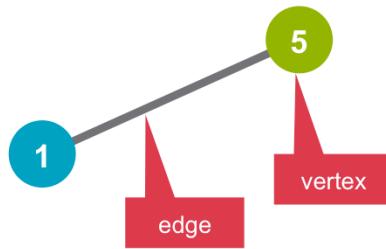
- **Describe GraphX**
- Define Regular, Directed, and Property Graphs
- Create a Property Graph
- Perform Operations on Graphs

The first section introduces Apache Spark GraphX.



What is a Graph?

A way to represent a set of vertices that may be connected by edges.

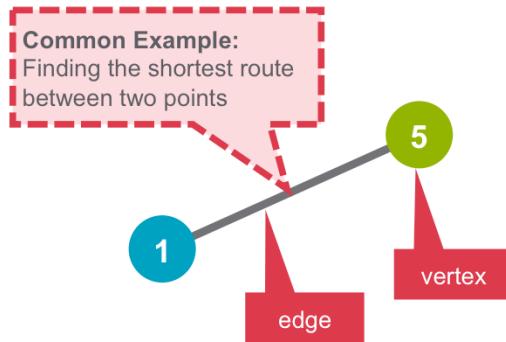


A graph is a way of representing a set of vertices that may be connected by edges.



What is a Graph?

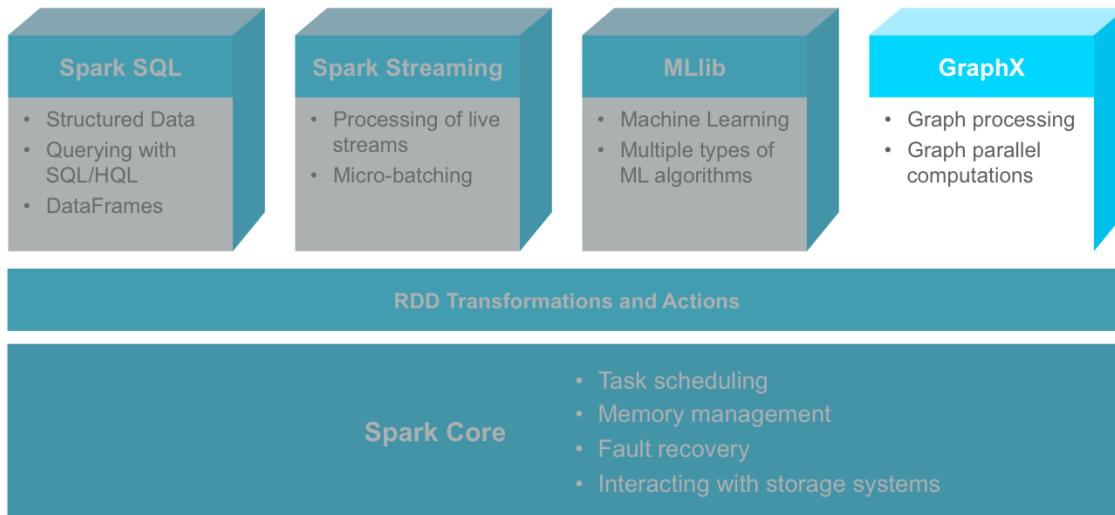
A way to represent a set of vertices that may be connected by edges.



A common example of graph processing is to find the shortest route between two points where the points are the vertices and the routes are edges.

Need more info? Learn about graph theory: https://en.wikipedia.org/wiki/Graph_theory

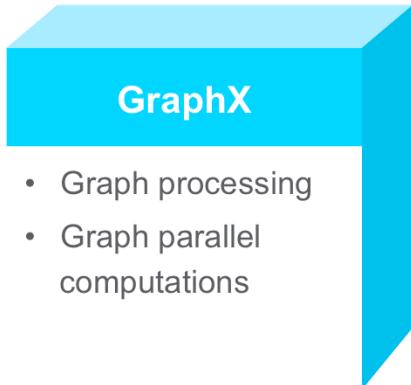
What is GraphX?



GraphX is the Apache Spark component for graphs and graph-parallel computations. It is a distributed graph processing framework that sits on top of Spark core.



Apache Spark GraphX



- Graph processing
- Graph parallel computations

- Spark component for graphs and graph-parallel computations
- Combines data parallel and graph parallel processing in single API
- View data as graphs and as collections (RDD)
 - no duplication or movement of data
- Operations for graph computation
- Provides graph algorithms and builders

GraphX combines data parallel and graph parallel processing in a single API.

We can view data as graphs and as collections without the need for duplication or movement of data.

GraphX has a number of operators that can be used for graph computations.

GraphX also provides graph algorithms and builders for graph analytics.

Knowledge Check



Knowledge Check



GraphX is the Apache Spark component for graphs and graph parallel computations. Which of the characteristics listed below are true of GraphX?

1. GraphX sits on top of Spark Core
2. GraphX provides distinct APIs for data parallel and graph parallel processing
3. GraphX provides multiple operators, including an optimized version of Pragel
4. When using GraphX, we can view data as graphs and as collections without the need for duplication or movement of data

True

False – these are available in a single API

True

True

Learning Goals



Learning Goals

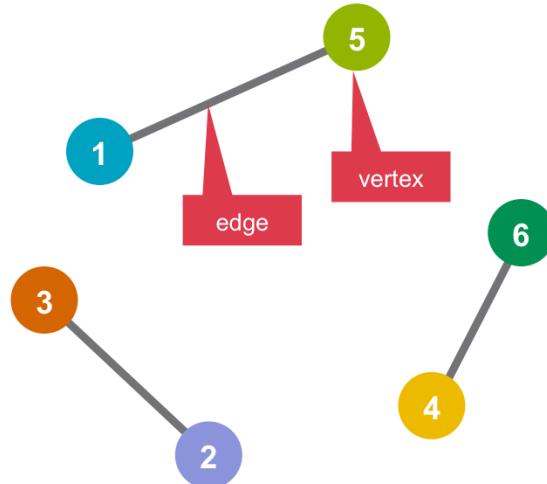


- Describe GraphX
- **Define Regular, Directed, and Property Graphs**
- Create a Property Graph
- Perform Operations on Graphs

In this section, we will define and regular, directed and property graphs.

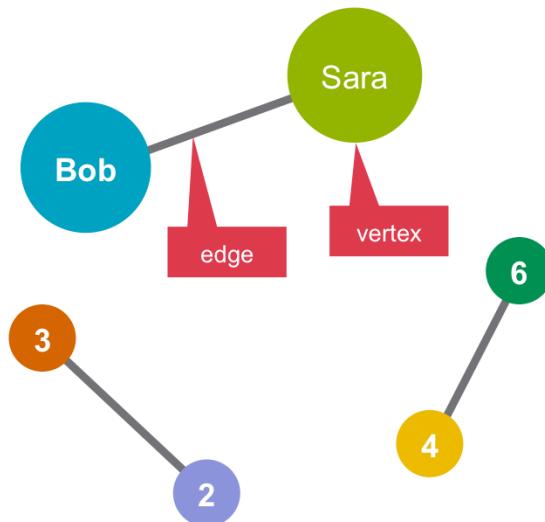


Regular Graphs vs Directed Graphs



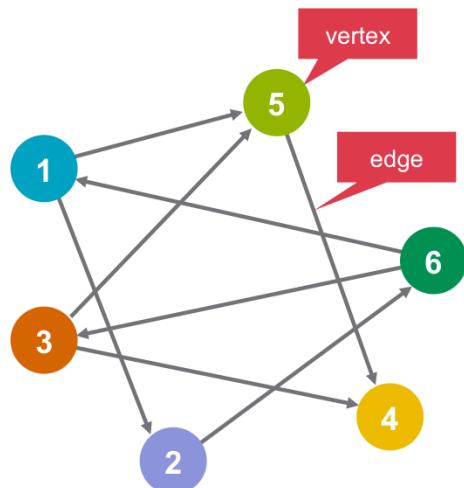
A regular graph is one where each vertex has the same number of edges.

Regular Graphs vs Directed Graphs



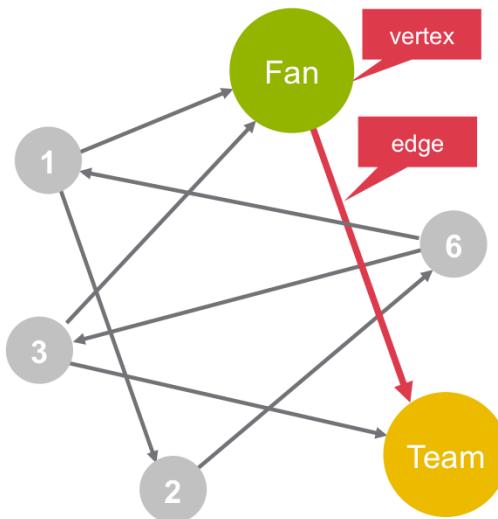
An example of a regular graph, is where the users on Facebook are vertices. If Bob is a friend of Sara, then Sara is also a friend of Bob.

Regular Graphs vs Directed Graphs



A directed graph or digraph...

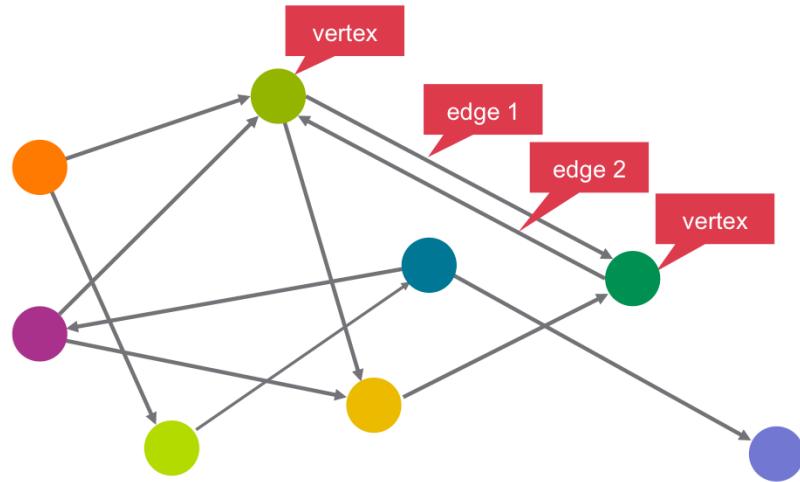
Regular Graphs vs Directed Graphs



is one in which edges run in one direction from vertex Fan to vertex Team.

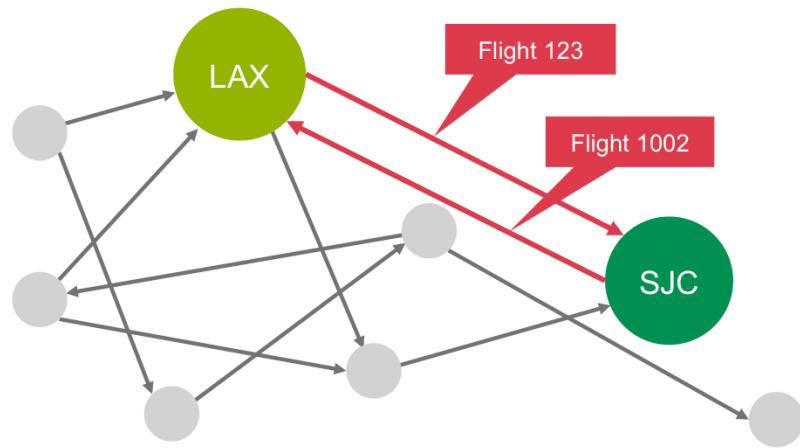
An example of a directed graph is a Twitter follower. User Fan can follow user Team without implying that user Team follows user Fan.

Property Graph



A property graph is the primary abstraction of Spark GraphX.

Property Graph



A property graph is a directed multigraph which can have multiple edges in parallel. Every edge and vertex has user defined properties associated with it, and therefore is unique.

Property graphs are immutable, distributed and fault-tolerant.

Knowledge Check



Knowledge Check



The primary abstraction in Spark GraphX is the property graph. Which characteristics below are true of a property graph?

1. Property graphs are bidirectional
2. Edges and vertices have user-defined properties associated with them
3. Property graphs are directional
4. Every edge and vertex is unique
5. Property graphs are immutable

False
True
True
True
True

Learning Goals



Learning Goals



- Define GraphX
- Define Regular, Directed, and Property Graphs
- **Create a Property Graph**
- Perform Operations on Graphs

In this section, we will create a property graph.



Create a Property Graph

- 1 Import required classes
- 2 Create vertex RDD
- 3 Create edge RDD
- 4 Create graph



To create a property graph, follow the steps listed here. We will describe each step in more detail, using flight data.

Create a Property Graph: Data Set

Airports

Vertex ID	Property (V)
org_id	(id, name, numAirlines, numFlights, numDestinations)

Routes

Source ID	Dest ID	Property (E)
org_id	dest_id	(src, dst, name, numAirlines, numFlights, distance, crsTime, majorLines)

We have the flight data streaming in. Airports are the vertices in our graph, with routes being the edges. Listed here are the properties we assign to the vertices (airports) and edges (routes).



Create a Property Graph

1 Import required classes

```
import org.apache.spark._  
import org.apache.spark.graphx._  
import org.apache.spark.rdd.RDD
```

When creating a property graph, first we import the GraphX related classes listed here.



Create a Property Graph

2 Create vertex RDD

```
//defining the case class first
case class Airport(id:Int, name:String, numAirlines:Int,
numFlights:Int, numDestinations:Int)
```

id	(id, name, numAirlines, numFlights, numDestinations)
14843	Airport(14843, SJU, 7, 2309, 22)

Next we define the case class, which includes the properties that we define for the vertices.

The airports have the following properties:

- id is the ID of the airport where the flight originates
- name is the name of the airport where the flight originates
- numAirlines is the number of airlines at that airport
- numFlights is the number of flights at the airport
- And numDestinations is the number of destinations that flights go to



Create a Property Graph

2 Create vertex RDD

```
//create the vertex rdd

val airports = ardd.map(x => (x(org_id), x)).groupByKey.map(x => (x._1.toLong, {
    val id = x._1.toInt
    val name = x._2.map(y=>y(origin)).head
    val numAirlines = x._2.map(y=>y(carrier)).toList.distinct.size
    val numFlights = x._2.size
    val numDestinations = x._2.map(y=>y(dest_id)).toList.distinct.size
    Airport(id, name, numAirlines, numFlights, numDestinations)
})))
```

In this example, we are making the assumption that the data has been loaded into an RDD called ardd.



Create a Property Graph

2 Create vertex RDD

```
//create the vertex rdd
```

```
val airports = ardd.map(x => (x.org_id, x)).groupByKey.map(x => (x._1.toLong, {  
    val id = x._1.toInt      airport ID  
    val name = x._2.map(y=>y.origin).head   origin  
    val numAirlines = x._2.map(y=>y(carrier)).toList.distinct.size  distinct airlines  
    val numFlights = x._2.size     total number of flights  
    val numDestinations = x._2.map(y=>y(dest_id)).toList.distinct.size  
    Airport(id, name, numAirlines, numFlights, numDestinations)  number of destinations  
}))
```

We build the vertex RDD as shown here.

The result is a tuple with the origin ID, followed by an array of the properties of the vertex.



Create a Property Graph

2 Create vertex RDD

```
//create the vertex rdd

val airports = ardd.map(x => (x.org_id, x)).groupByKey.map(x => (x._1.toLong, {
  val id = x._1.toInt
  val name = x._2.map(y=>y(origin)).head
  val org_id = x._2.map(y=>y(carrier)).toList.distinct.size
  val numAirlines = x._2.size
  val numFlights = x._2.size
  val numDestinations = x._2.map(y=>y(dest_id)).toList.distinct.size
  Airport(id, name, numAirlines, numFlights, numDestinations)
}))
```

org_id	(id, name, numAirlines, numFlights, numDestinations)
14843	Airport(14843, SJU, 7, 2309, 22)

Each vertex must have a vertex ID, which in this case is the origin airport ID.



Create a Property Graph

2 Create vertex RDD

```
//create the vertex rdd
```

```
val airports = ardd.map(x => (x.org_id, x)).groupByKey.map(x => (x._1.toLong, {
  val id = x._1.toInt      airport ID
  val name = x._2.map(y=>y.origin).head    origin
  val numAirlines = x._2.map(y=>y(carrier)).toList.distinct.size  distinct airlines
  val numFlights = x._2.size     total number of flights
  val numDestinations = x._2.map(y=>y(dest_id)).toList.distinct.size
  Airport(id, name, numAirlines, numFlights, numDestinations)
}))
```

number of destinations

org_id	(id, name, numAirlines, numFlights, numDestinations)
14843	Airport(14843, SJU, 7, 2309, 22)

We can define properties for the vertices. In our example, the properties for the vertices are shown in the table. We compute the distinct list of airlines from the carrier field, total number of flights, and the number of distinct destinations.



Create a Property Graph

3 Create edge RDD

```
case class Route(src:Int, dst:Int, name:String,  
numAirlines:Int, numFlights:Int, distance:Int,  
crsTime:Double, majorLines:Boolean)
```

org_id	dest_id	(src, dst, name, numAirlines, numFlights, distance, crsTime, majorLines)
11618	13495	Route(11618,13495,EWR->MSY,3,69,1167,50.0,true)

Now, we define the Route case class, and then create the routes RDD.



Create a Property Graph

3 Create edge RDD

```
//create the routes rdd
```

```
val routes = ardd.map(x => ((x.org_id),x.dest_id), x).groupByKey.map(x => (x._1, {
    val src = x._1._1.toInt      source ID
    val dst = x._1._2.toInt      destination ID
    val name = x._2.map(y=>y(origin)).head + "-" + x._2.map(y=>y(dest)).head   route name
    val numAirlines = x._2.map(y=>y(carrier)).toList.distinct.size   distinct airlines
    val numFlights = x._2.size   total number of flights
    val distance = x._2.map(y=>y(dist)).head.toFloat.toInt   distance
    val crsTime = x._2.map(y=>y(crselapsedtime)).flatMap(y=>y).head   crs elapsed time
    val majorLines = x._2.map(y=>y(carrier)).map(y => List("AA", "UA").contains(y)).reduce(_ || _)
    Route(src, dst, name, numAirlines,numFlights,distance,crsTime, majorLines)
}))
```

creating list of
major airlines

MAPR Academy

© 2016 MapR Technologies 9-32

An edge in a property graph will have the source ID, destination ID, and properties. The edge RDD must contain a source ID, destination ID, and edge properties. Before we create the edge RDD, we first create the routes RDD that contains the required information.

The source ID and destination IDs are numbers, and we are therefore converting them to integers. We construct the route name as shown in the example, with the origin airport code to the destination airport code. As before, we compute the distinct list of airlines and the total number of flights. The distance and elapsed time are both part of the data. We define a list of major airlines. In this case, carriers AA and UA are defined as major airlines. We check to see if the carrier is a major airline and return true or false accordingly.

Next we will use this routes RDD to create the edge RDD.



Create a Property Graph

3 Create edge RDD

```
//create the edge rdd
val edges = routes.map(x => Edge(x._1._1.toLong, x._1._2.toLong, x._2))

edges.first
Res: Edge(11618,13495,Route(11618,13495,EWR->MSY,3,69,1167,50.0,true))
```

We now use the Edge case class to map the routes RDD to an Edge RDD. The Edge class stores the edge property. To see what this RDD looks like, use edges.first, as shown here.



Create a Property Graph

4 Create graph

```
// define default airport  
val nowhere = Airport(0,0,0,0,0)  
  
//build initial graph  
val graph = Graph(airports, edges, nowhere)
```

Once the vertex and edge RDDs are defined, we can create the property graph. To create the graph use the Graph method which accepts (vertex, edge, default vertex). We define a default vertex as shown and then create the graph.



Knowledge Check



Knowledge Check



The steps to create a Property graph are listed below. List these steps in the correct order.

Import required classes

Create graph

Create edge RDD

Create vertex RDD

1. Import required classes
2. Create vertex RDD
3. Create edge RDD
4. Create graph

Knowledge Check



The steps to create a Property graph are listed below. List these steps in the correct order.

1

Import required classes

4

Create graph

3

Create edge RDD

2

Create vertex RDD

1. Import required classes
2. Create vertex RDD
3. Create edge RDD
4. Create graph

Learning Goals



Learning Goals



- Define GraphX
- Define Regular, Directed, and Property Graphs
- Create a Property Graph
- **Perform Operations on Graphs**

In this section, we will look at the different types of operators on property graphs.

Graph Operators

To find information about the graph

Operator	Description
numEdges	number of edges (Long)
numVertices	number of vertices (Long)
inDegrees	The in-degree of each vertex (VertexRDD[Int])
outDegrees	The out-degree of each vertex (VertexRDD[Int])
degrees	The degree of each vertex (VertexRDD[Int])

Here are some operators that we can use to find more information about the graph, such as the number of edges and number of vertices.



Graph Operators

Caching Graphs

Operator	Description
<code>cache()</code>	Caches the vertices and edges; default level is <code>MEMORY_ONLY</code>
<code>persist(newLevel)</code>	Caches the vertices and edges at specified storage level; returns a reference to this graph
<code>unpersist(blocking)</code>	Uncaches both vertices and edges of this graph
<code>unpersistVertices(blocking)</code>	Uncaches only the vertices, leaving edges alone

The operators listed here can be used for caching graphs.



Graph Operators

Other Operators – Collection and Structural

Operator	Description
vertices	An RDD containing the vertices and associated attributes
edges	An RDD containing the edges and associated attributes
subgraph(epred, vpred)	Restricts graph to only vertices and edges satisfying the predicates
reverse	Reverses all edges in the graph

Some of the operators listed here can be used to view graphs as collections such as vertices and edges. The other operators listed here can be used to modify the graph structure. There are many more operators that can be used on graphs. Refer to the Spark GraphX API documentation.

--

Need more info?

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.package>
<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph>



Graph Operators

Property operators:

- Similar to map on RDD
- Result in new graph

Operator	Description
mapVertices	Transforms each vertex attribute using map function
mapEdges	Transforms each edge attribute, a partition at a time, using map function – does not pass the vertex value to the edge
mapTriplets	Transforms each edge attribute, a partition at a time, using the map function – does pass the vertex value to the edge

Property operators are similar to the map operators on RDDs, and result in a new graph.



Graph Operators

To answer questions such as:

- How many airports are there?
- How many routes for flights?
- Which airport has the most incoming flights?
- Which airports are covered by major airlines?



We can use the graph operators to answer the questions listed here.

Class Discussion



Class Discussion



1. How many airports are there?
 - In our graph, what represents airports?
 - Which operator could you use to find the number of airports?
2. How many routes are there?
 - In our graph, what represents routes?
 - Which operator could you use to find the number of routes?

How Many Airports are There?

How many airports are there?

- In our graph, what represents airports?

Vertices

- Which operator could you use to find the number of airports?

`graph.numVertices`

The first question we want to ask our data is, “How many airports are there?”



PageRank

```
graph.pageRank(tolerance).vertices
```

- Measure the importance of vertices
- **tolerance** is a measure of convergence
- Returns a graph with vertex attributes

Another operator is PageRank. It is based on the Google PageRank algorithm. It can be used to measure the importance of the vertices. We have to specify the tolerance, which is the measure of convergence.

PageRank on a graph will return a graph with vertex attributes containing the normalized edge weight.

In our example, this would be to answer the question – Which are the most important airports?



Use Case

Monitor air traffic at airports



Monitor delays



Analyze airport and routes overall



Analyze airport and routes by airline



You can use graphs in these different situations. For security reasons, on a particular day, you want to monitor the data in real time. In this case, you may add graph processing to a streaming application and send the output to a visualization tool.

If you want to do a longer term analysis, you may save the data over a longer period of time such as a month or quarter, and then do the analysis by airline, by route, by week and so on. In this case, you may create a standalone graph processing application.

In our activity, we will look at an example of including graph processing in our streaming application and also a stand alone graph processing application.

Lab 9 – Use Apache Spark GraphX



In this activity, you will define a graph and apply graph operators to it. In the first part of the activity, you will create the property graph and apply operators to it in the Interactive shell. In the second part, you will add some of this to the streaming application created in Lesson 8.



Next Steps

DEV362 – Create Data Pipelines With Apache Spark

Lesson 10: Use Apache Spark MLlib

Congratulations! You have completed Lesson 9; Use Apache Spark GraphX.
Continue on to lesson 10 to learn about the Apache Spark Machine Learning Library.



 Academy

DEV362 – Create Data Pipelines With Apache Spark

Lesson 10: Use Apache Spark MLlib

Welcome to DEV 362 Lesson 10: Use Apache Spark machine learning library



Learning Goals



Learning Goals



- Describe Apache Spark MLlib
- Describe the Machine Learning Techniques:
 - Classification
 - Clustering
 - Collaborative Filtering
- Use Collaborative Filtering to Predict User Choice

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

- Define the Apache Spark machine learning library
- Describe three popular machine learning techniques: classification, clustering, and collaborative filtering, commonly known as recommendation
- And use collaborative filtering to predict what a user will like

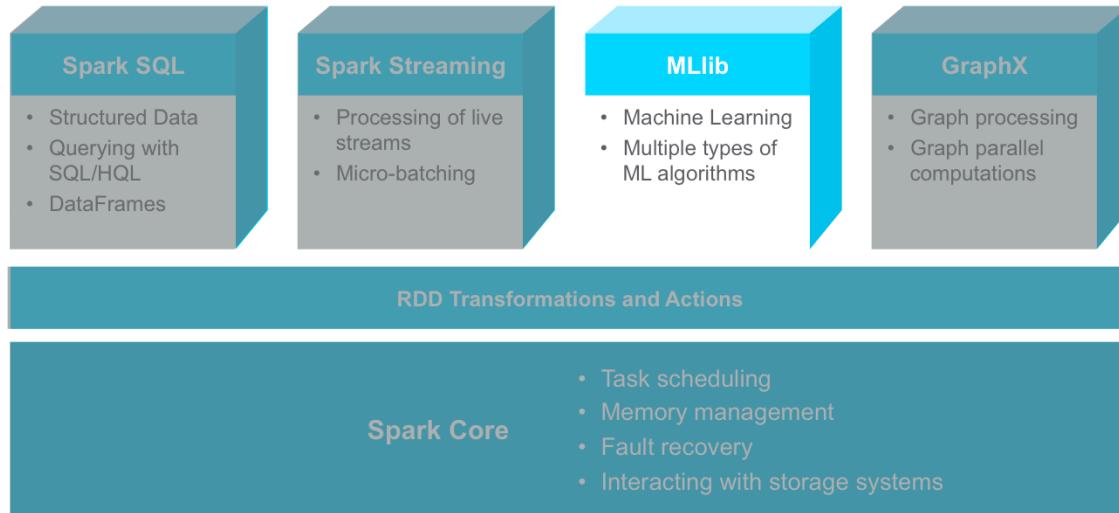
Learning Goals



- **Describe Apache Spark MLlib**
- Describe the Machine Learning Techniques:
 - Classification
 - Clustering
 - Collaborative Filtering
- Use Collaborative Filtering to Predict User Choice

The first section introduces the Apache Spark machine learning library.

What is MLlib?



MLlib is the Apache Spark library of machine learning functions. It contains only those machine learning algorithms that are parallel and run on clusters. Those algorithms that were not designed to run on parallel platforms are not included.

Apache Spark machine learning library algorithms are most suitable for running on a single large data set.

MLlib Algorithms and Utilities

Algorithms and Utilities	Description
Basic statistics	Includes summary statistics, correlations, hypothesis testing, random data generation
Classification and regression	Includes methods for linear models, decision trees and Naïve Bayes
Collaborative filtering	Supports model-based collaborative filtering using alternating least squares (ALS) algorithm
Clustering	Supports K-means clustering
Dimensionality reduction	Supports dimensionality reduction on the RowMatrix class; singular value decomposition (SVD) and principal component analysis (PCA)
Feature extraction and transformation	Contains several classes for common feature transformations

MLlib provides the machine learning algorithms and utilities listed here.



MLlib Algorithms and Utilities

Algorithms and Utilities	Description
Basic statistics	Includes summary statistics, correlations, hypothesis testing, random data generation
Classification and regression	Includes methods for linear models, decision trees and Naïve Bayes
Collaborative filtering	Supports model-based collaborative filtering using alternating least squares (ALS) algorithm
Clustering	Supports K-means clustering
Dimensionality reduction	Supports dimensionality reduction on the RowMatrix class; singular value decomposition (SVD) and principal component analysis (PCA)
Feature extraction and transformation	Contains several classes for common feature transformations

MLlib includes functions for summary statistics on RDDs. There are operations for calculating correlations between two series of data using either the Pearson or Spearman methods. MLlib also provides support for hypothesis testing and random data generation.



MLlib Algorithms and Utilities

Algorithms and Utilities	Description
Basic statistics	Includes summary statistics, correlations, hypothesis testing, random data generation
Classification and regression	Includes methods for linear models, decision trees and Naïve Bayes
Collaborative filtering	Supports model-based collaborative filtering using alternating least squares (ALS) algorithm
Clustering	Supports K-means clustering
Dimensionality reduction	Supports dimensionality reduction on the RowMatrix class; singular value decomposition (SVD) and principal component analysis (PCA)
Feature extraction and transformation	Contains several classes for common feature transformations

MLlib has various methods for binary, and for multiple classification and regression problems. These include methods for linear models, decision trees and Naïve Bayes.

There is support for model-based collaborative filtering using the Alternating Least Squares or ALS algorithm.

Later in this lesson, we will use collaborative filtering to predict what movies a user will like.



MLlib Algorithms and Utilities

Algorithms and Utilities	Description
Basic statistics	Includes summary statistics, correlations, hypothesis testing, random data generation
Classification and regression	Includes methods for linear models, decision trees and Naïve Bayes
Collaborative filtering	Supports model-based collaborative filtering using alternating least squares (ALS) algorithm
Clustering	Supports K-means clustering
Dimensionality reduction	Supports dimensionality reduction on the RowMatrix class; singular value decomposition (SVD) and principal component analysis (PCA)
Feature extraction and transformation	Contains several classes for common feature transformations

MLlib also provides support for K-means clustering, often used for data mining of large data sets.



MLlib Algorithms and Utilities

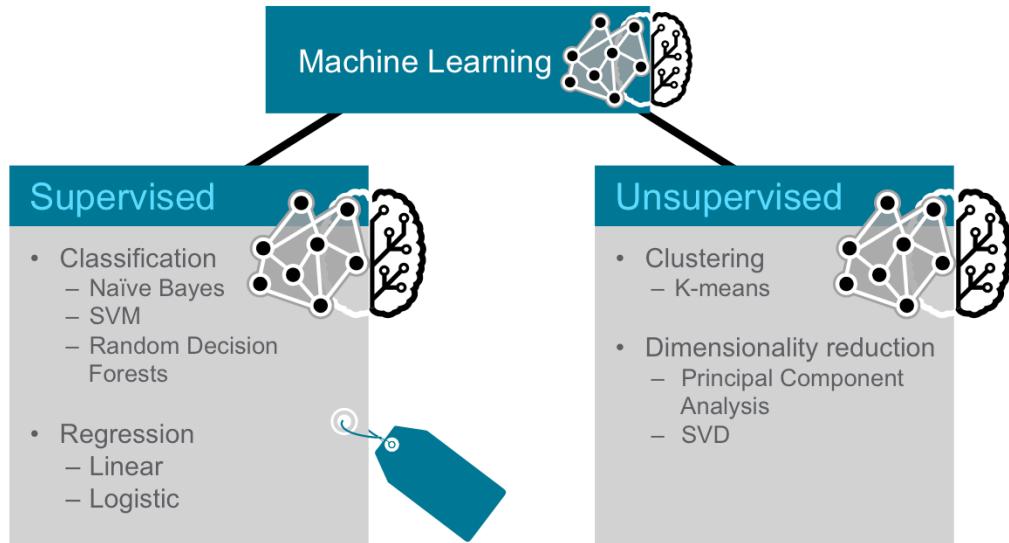
Algorithms and Utilities	Description
Basic statistics	Includes summary statistics, correlations, hypothesis testing, random data generation
Classification and regression	Includes methods for linear models, decision trees and Naïve Bayes
Collaborative filtering	Supports model-based collaborative filtering using alternating least squares (ALS) algorithm
Clustering	Supports K-means clustering
Dimensionality reduction	Supports dimensionality reduction on the RowMatrix class; singular value decomposition (SVD) and principal component analysis (PCA)
Feature extraction and transformation	Contains several classes for common feature transformations

MLlib supports dimensionality reduction on the RowMatrix class. It provides functionality for Singular Value Decompositions, shown here as SVD, and Principal Component Analysis, shown as PCA.

There are also several classes for common feature transformations.



Examples of ML Algorithms



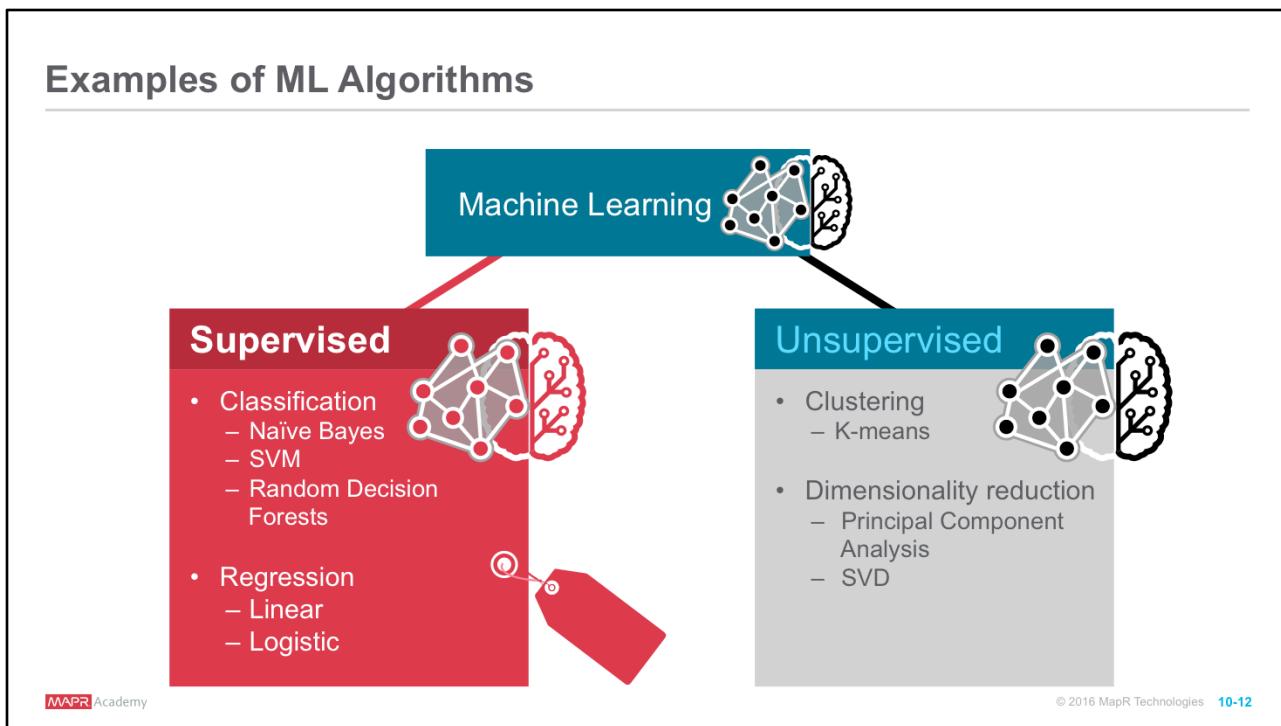
MAPR Academy

© 2016 MapR Technologies 10-11

In general, machine learning may be broken down into two classes of algorithms: supervised and unsupervised.



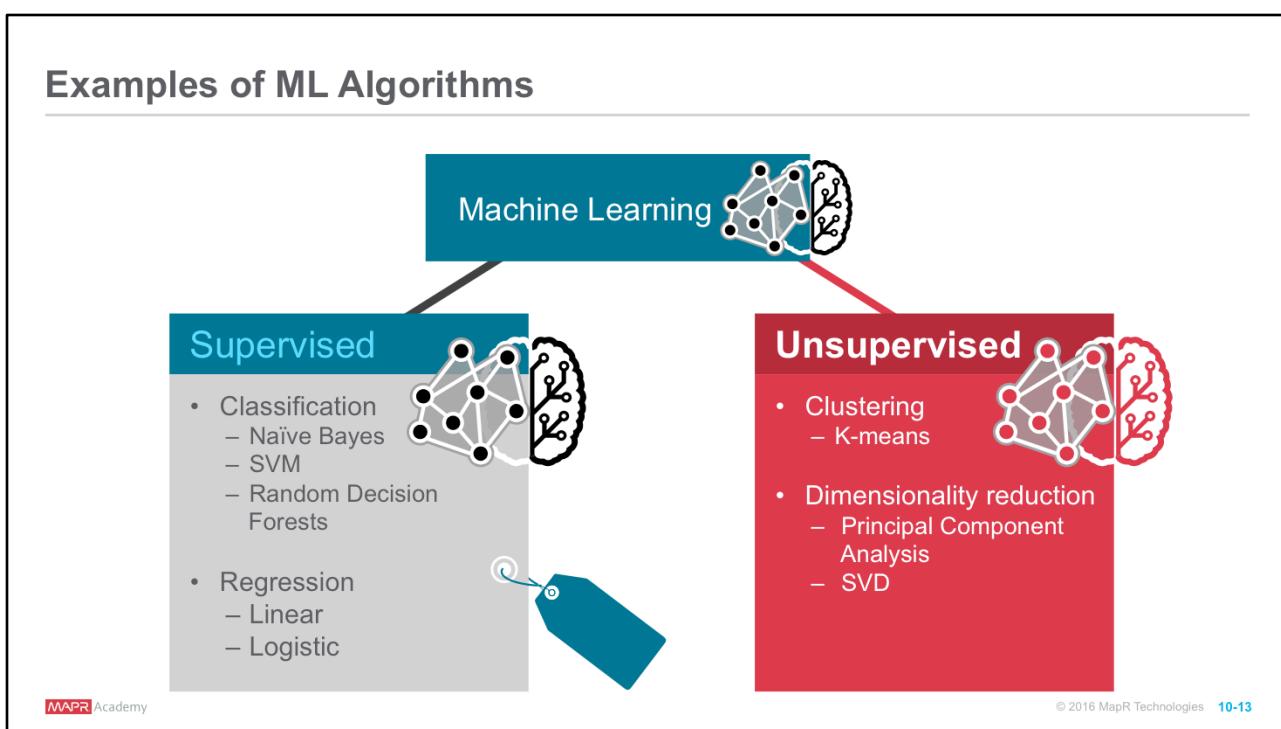
Examples of ML Algorithms



Supervised algorithms use labeled data in which both the input and output are provided to the algorithm.



Examples of ML Algorithms



MAPR Academy

© 2016 MapR Technologies 10-13

Unsupervised algorithms do not have the outputs in advance. These algorithms are left to make sense of the data without any hints.



Class Discussion



Class Discussion



What is the difference between supervised and unsupervised machine learning algorithms?

Why is classification considered supervised, while clustering is considered unsupervised?

Supervised algorithms require known information about the results, to which to compare the sample data.

Unsupervised algorithms have no known information, and must determine all of their results from the sample data.

Classification has a known set of definitions that the sample data is compared against. The sample data is determined to either be the same as, or different from the known definitions.

Clustering organizes a set of data into groups, based only on the data itself. It does not require any known prior information about the data.

Learning Goals



Learning Goals



- Describe Apache Spark MLlib
- **Describe the Machine Learning Techniques:**
 - Classification
 - Clustering
 - Collaborative Filtering
- Use Collaborative Filtering to Predict User Choice

In the next section, we will describe three common categories of machine learning techniques, starting with Classification.

Machine Learning: Classification

Classification

The screenshot shows a Gmail inbox with a search bar containing 'in:spam'. Below the search bar, several spam emails are listed. A red box highlights the search term 'in:spam' in the search bar, and another red box highlights the first email in the list. To the right of the list, a text box states 'Identifies category for item'.

Identifies category for item

© 2016 MapR Technologies 10-18

Gmail uses a machine learning technique called classification to designate if an email is spam or not, based on the data of an email: the sender, recipients, subject, and message body.

Classification is a supervised algorithm meaning it uses labeled data in which both the input and output are provided to the algorithm.

Classification takes a set of data with known labels and learns how to label new records based on that information. The algorithm identifies which category an item belongs to, based on labeled examples of known items. For example, it identifies whether an email is spam or not based on emails known to be spam or not.

Classification: Definition

Form of ML that:

- Identifies which category an item belongs to
- Uses supervised learning algorithms
 - Data is labeled

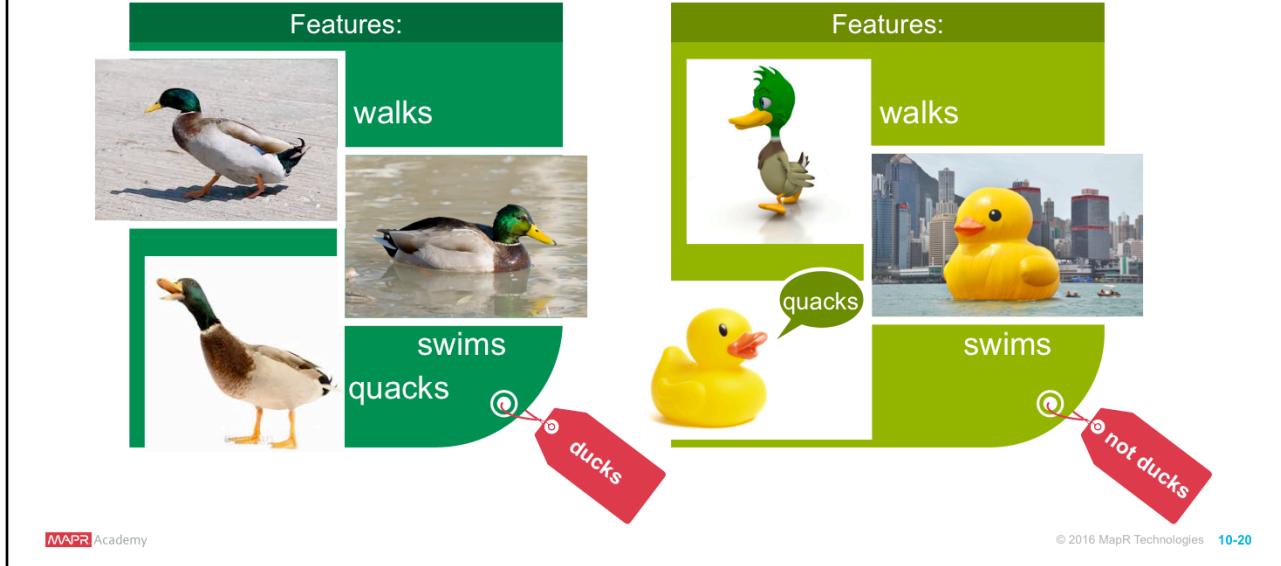


Classification is a family of supervised machine learning algorithms that designate input as belonging to one of several pre-defined classes.

Some common use cases for classification include credit card fraud detection and email spam detection, both of which are binary classification problems.

Classification data is labeled, for example, as spam/non-spam or fraud/non-fraud. Machine learning assigns a label or class to new data.

If it Walks/Swims/Quacks Like a Duck Then It Must Be a Duck



You can classify something based on pre-determined features. Features are the “if questions” that you ask. The label is the answer to those questions. In this example, if it walks, swims, and quacks like a duck, then the label is “duck”.

In this simplistic example, the classification is binary but classification can extend to any number of pre-defined classes.

Building and Deploying a Classifier Model

Spam:

free money now!
buy this money
free savings \$\$\$

Non-spam:

how are you?
that Spark job
that Spark job

Training Data

MAPR Academy

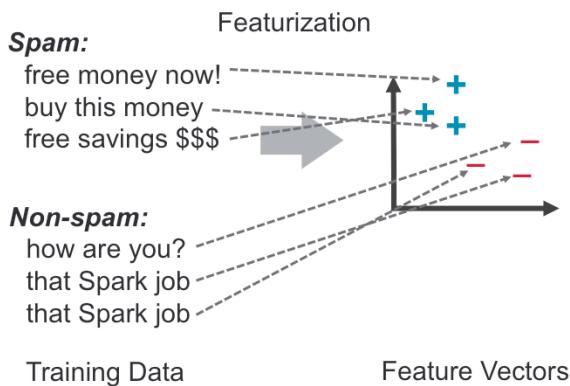
Image reference O'Reilly Learning Spark

© 2016 MapR Technologies 10-21

To build a classifier model, first extract the features that most contribute to the classification. In our email example, we find features that define an email as spam, or not spam.



Building and Deploying a Classifier Model



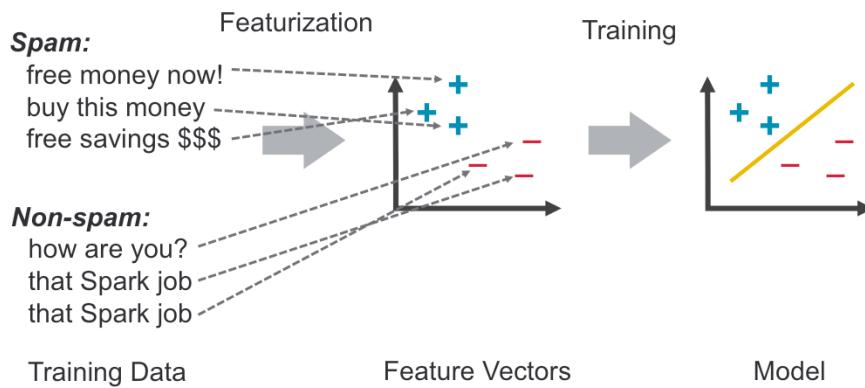
MAPR Academy

Image reference O'Reilly Learning Spark

© 2016 MapR Technologies 10-22

The features are transformed and put into Feature Vectors, which is a vector of numbers representing the value for each feature. We rank our features by how strongly they define an email as spam, or not spam.

Building and Deploying a Classifier Model



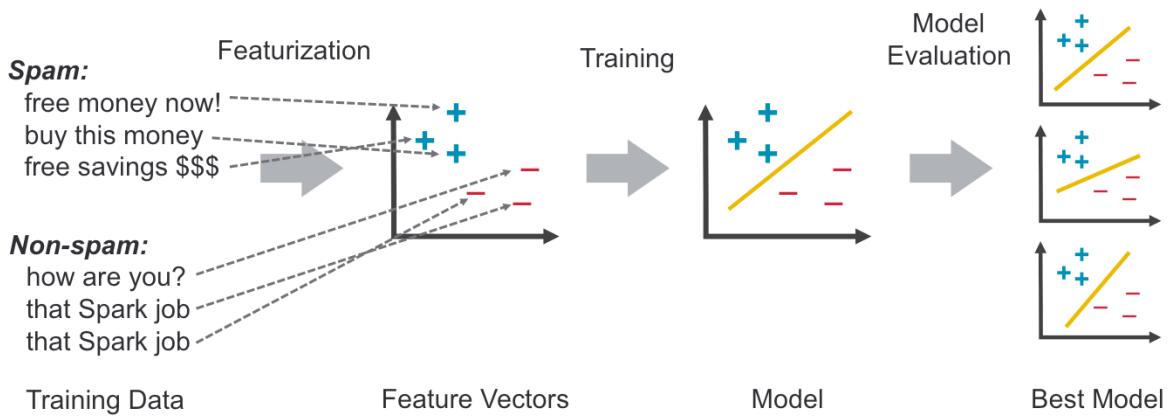
MAPR Academy

Image reference O'Reilly Learning Spark

© 2016 MapR Technologies 10-23

We train our model by making associations between the input features and the labeled output associated with those features.

Building and Deploying a Classifier Model



MAPR Academy

Image reference O'Reilly Learning Spark

© 2016 MapR Technologies 10-24

Then at runtime, we deploy our model by extracting the same set of features, and ask the model to classify that set of features as one of the pre-defined set of labeled classes.

Learning Goals



Learning Goals

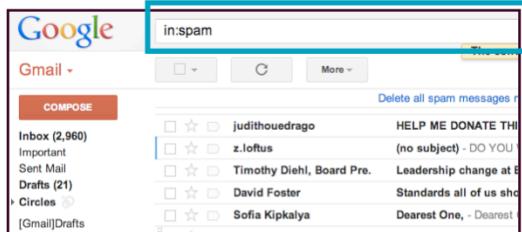


- Describe Apache Spark MLlib
- **Describe the Machine Learning Techniques:**
 - Classification
 - Clustering
 - Collaborative Filtering
- Use Collaborative Filtering to Predict User Choice

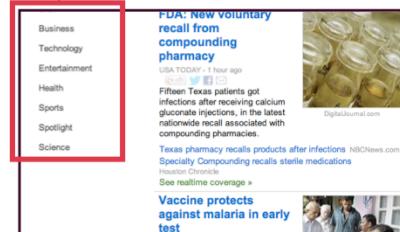
Next, we will discuss Clustering as a machine learning model.

Machine Learning: Clustering

Classification



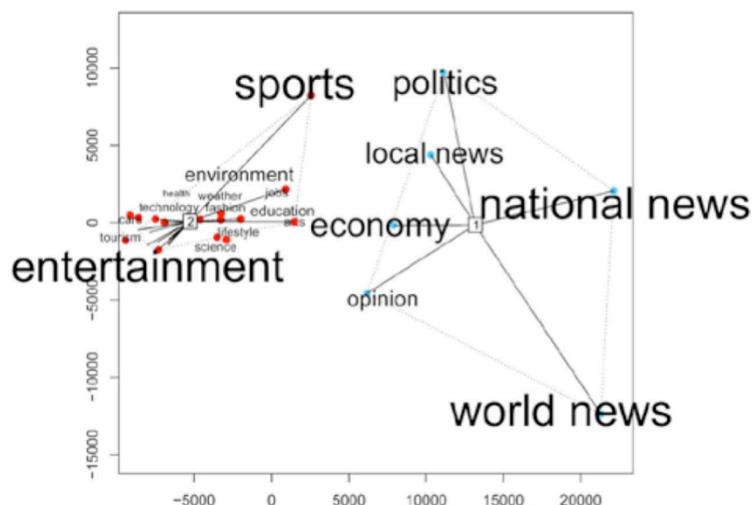
Clustering



Google News uses a technique called clustering to group news articles into different categories, based on title and content.

Clustering algorithms discover groupings that occur in collections of data.

Clustering: Definition



Marco Toledo Bastos, and Gabriela Zago SAGE
Copyright © by a Creative Commons Attribution License, unless otherwise noted.

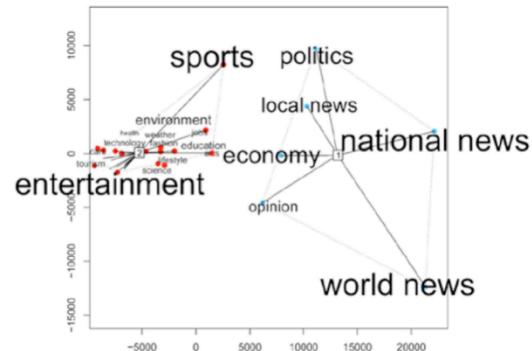
© 2016 MapR Technologies 10-28

In clustering, an algorithm classifies inputs into categories by analyzing similarities between input examples.



Clustering: Definition

- Unsupervised learning task
- Groups objects into **clusters of high similarity**



MAPR Academy

© 2016 MapR Technologies 10-29

Clustering uses unsupervised algorithms, which do not have the outputs in advance.
No known classes are used as a reference, as with a supervised algorithm like classification.

Clustering: Definition

- Unsupervised learning task
- Groups objects into **clusters of high similarity**
 - Search results grouping
 - Grouping of customers
 - Anomaly detection
 - Text categorization



MAPR Academy

© 2016 MapR Technologies 10-30

Clustering can be used for many purposes, for example, search results grouping.

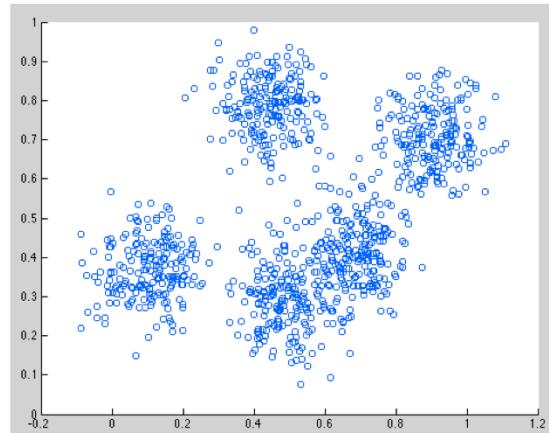
Other examples include:

- partitioning data into groups, such as grouping similar customers
- anomaly detection, such as fraud detection
- and text categorization, such as sorting books into genres



Clustering: Example

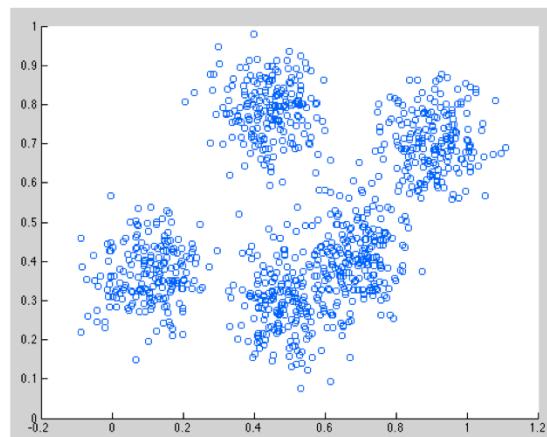
- Group similar objects



In this example, we are given a set of raw data points. The objective is to create some number of clusters that group these data points with those that are most similar, or closest.

Clustering: Example

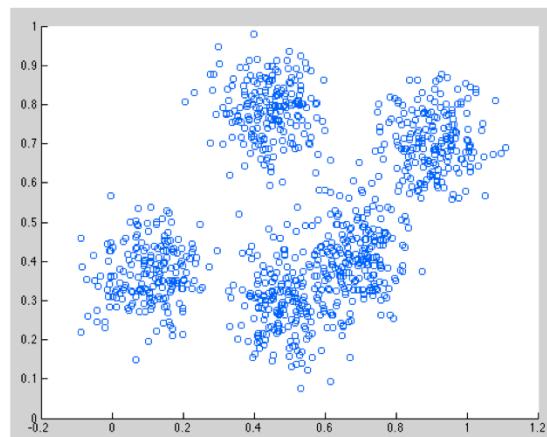
- Group similar objects
 - Use MLlib K-means algorithm
1. Initialize coordinates to center of clusters (centroid)



Clustering using the K-means algorithm begins by initializing all the coordinates to centroids. There are various ways you can initialize the points, including randomly.

Clustering: Example

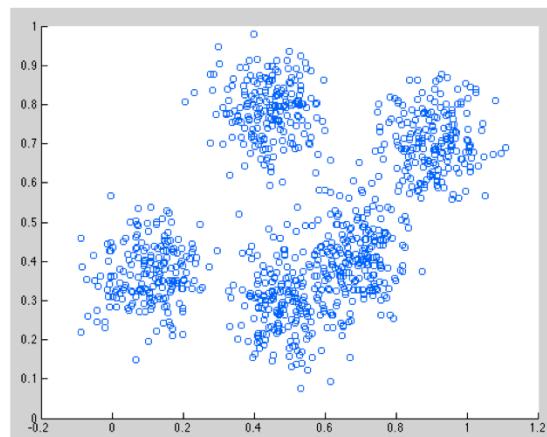
- Group similar objects
- Use MLlib K-means algorithm
 1. Initialize coordinates to center of clusters (centroid)
 2. Assign all points to nearest centroid



With every pass of the algorithm, each point is assigned to its nearest centroid based on some distance metric, usually Euclidean distance.

Clustering: Example

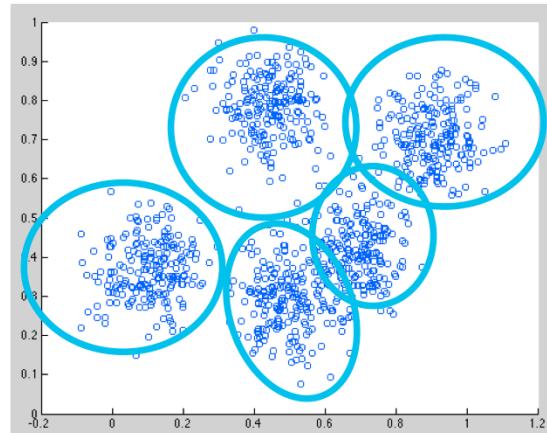
- Group similar objects
- Use MLlib K-means algorithm
 - 1. Initialize coordinates to center of clusters (centroid)
 - 2. Assign all points to nearest centroid
 - 3. Update centroids to center of points



The centroids are then updated to be the “centers” of all the points assigned to it in that pass.

Clustering: Example

- Group similar objects
- Use MLlib K-means algorithm
 - 1. Initialize coordinates to center of clusters (centroid)
 - 2. Assign all points to nearest centroid
 - 3. Update centroids to center of points
 - 4. Repeat until conditions met



The algorithm stops according to various programmable conditions, such as:

- A minimum change of median from the last iteration
- A sufficiently small intra-cluster distance
- A sufficiently large inter-cluster distance

Learning Goals





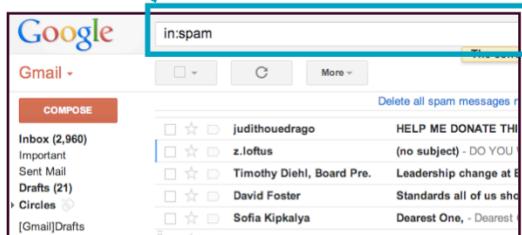
Learning Goals

- Describe Apache Spark MLlib
- **Describe the Machine Learning Techniques:**
 - Classification
 - Clustering
 - Collaborative Filtering
- Use Collaborative Filtering to Predict User Choice

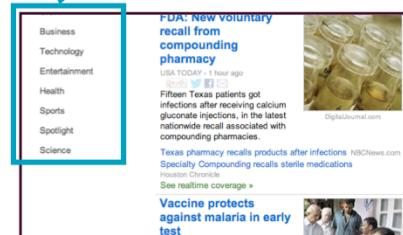
The final machine learning technique we will look at is collaborative filtering, more commonly known as recommendation. We will describe collaborative filtering briefly here, and then show an example of a movie recommendation engine.

Three Categories of Techniques for Machine Learning

Classification



Clustering



Collaborative Filtering
(Recommendation)

Customers Who Bought This Item Also Bought



Amazon uses a machine learning technique called collaborative filtering commonly referred to as recommendation, to determine products users will like based on their history and similarity to other users.

Collaborative Filtering with Spark

- Recommend items
 - (Filtering)
- Based on user preferences data
 - (Collaborative)



Collaborative filtering algorithms recommend items based on preference information from many users. The collaborative filtering approach is based on similarity; people who liked similar items in the past will like similar items in the future.

In the example shown, Ted likes movies A, B, and C. Carol likes movies B and C. Bob likes movie B. To recommend a movie to Bob, we calculate that users who liked B also liked C, so C is a possible recommendation for Bob.

Train a Model to Make Predictions

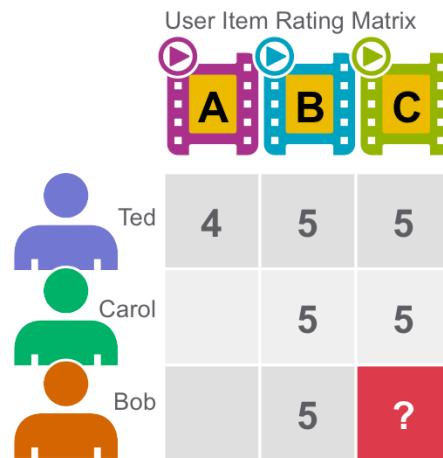
Ted and Carol like movies B and C



Bob likes movie B, what might he like?



Bob likes movie B, predict C



The goal of a collaborative filtering algorithm is to take preferences data from users, and to create a model that can be used for recommendations or predictions.

Ted likes movies A, B, and C. Carol likes movies B and C. We take this data and run it through an algorithm to build a model.

Then when we have new data such as Bob likes movie B, we use the model to predict that C is a possible recommendation for Bob.

Knowledge Check



Knowledge Check



Match the scenarios listed here with the machine learning technique that would provide the best results:

Classification, Clustering or Collaborative Filtering

You want to determine what restaurant someone may like, based on restaurants they have previously liked.

You want to detect fraudulent attempts to log into your website.

You need to list which students have passed, and which have failed an exam.

You want to organize your music collection based on genre metadata.

Restaurant recommendations::Collaborative Filtering. Suggesting restaurants that people liked, who also like the restaurants that our user liked.

Login fraud detection::Clustering. Looking for anomalies in failed login attempts.

List passed and failed students::Classification

Organizing music::Clustering.

Class Discussion



Class Discussion



Given the three machine learning techniques we've discussed:

- Classification
- Clustering
- Collaborative Filtering

What are some scenarios at your company that use one or more of these techniques? Fraud detection? Organization of data? Spam filters?

Learning Goals



Learning Goals



- Describe Apache Spark MLlib
- Describe the Machine Learning Techniques:
 - Classification
 - Clustering
 - Collaborative Filtering
- **Use Collaborative Filtering to Predict User Choice**

In this next section, we will use collaborative filtering to predict what a user will like.

Train a Model to Make Predictions

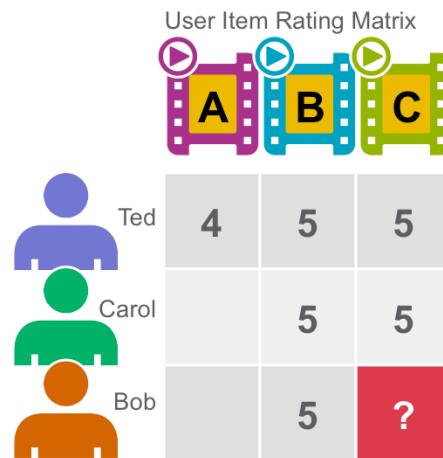
Ted and Carol like movies B and C



Bob likes movie B, what might he like?



Bob likes movie B, predict C



Let's look more deeply at our movie recommendation example.

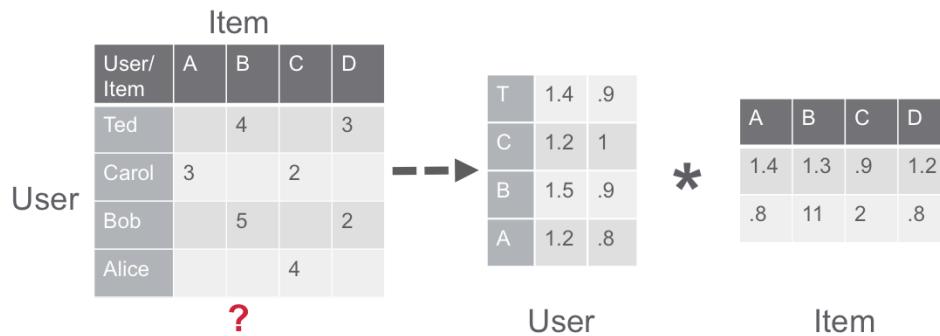
To quickly review, the goal of a collaborative filtering algorithm is to take preferences data from users, and to create a model that can be used for recommendations or predictions.

Ted likes movies A, B, and C. Carol likes movies B and C. We use this data to build a model, and predict what Bob will like.

Alternating Least Squares

Approximates **sparse** user item rating matrix

- as product of **two dense** matrices, User and Item factor matrices



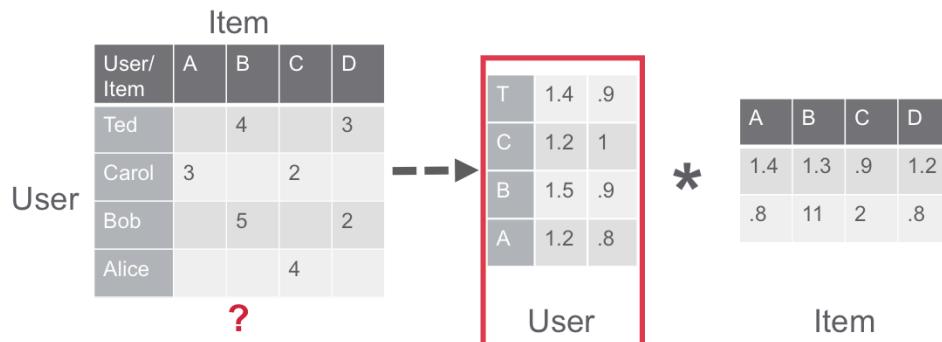
Alternating Least Squares, or ALS, approximates a sparse user item rating matrix of dimension K, as the product of two dense matrices.

We start with User and Item factor matrices of size $U \times K$ and $I \times K$. The factor matrices are also called latent feature models, and represent hidden features which the algorithm tries to discover.

Alternating Least Squares

Approximates **sparse** user item rating matrix

- as product of **two dense** matrices, User and Item factor matrices
- tries to **learn the hidden features** of each user and item

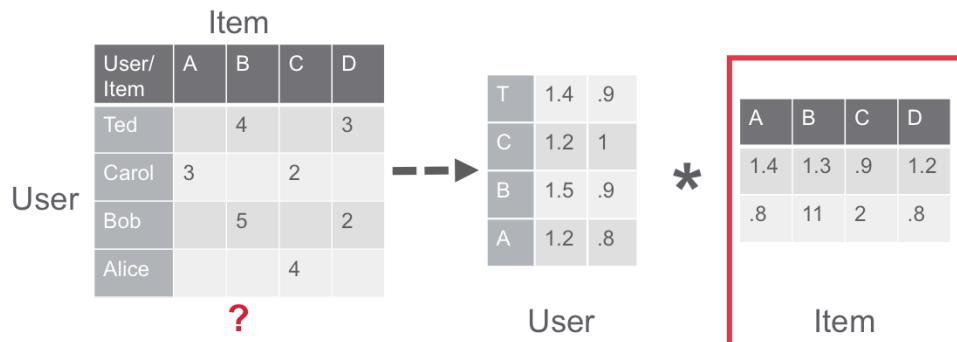


One matrix tries to describe the latent or hidden features of each user

Alternating Least Squares

Approximates **sparse** user item rating matrix

- as product of **two dense** matrices, User and Item factor matrices
- tries to **learn the hidden features** of each user and item

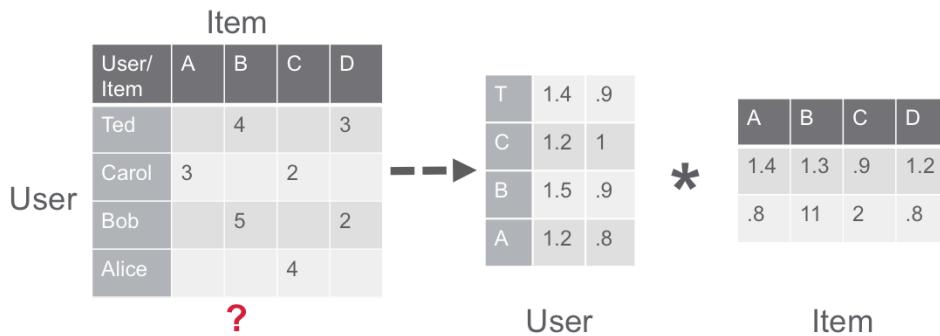


and one tries to describe latent properties of each movie.

Alternating Least Squares

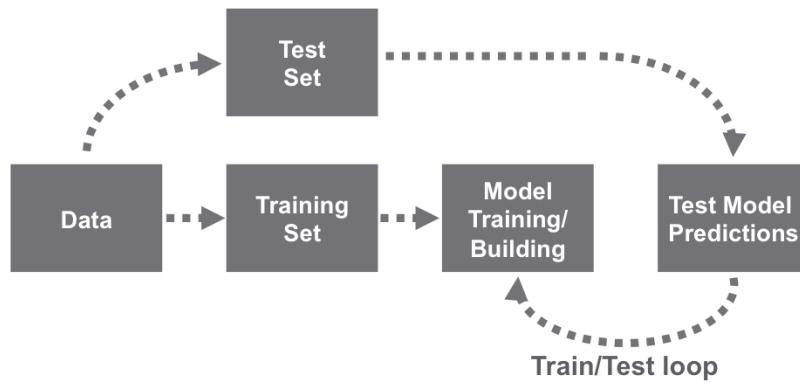
Approximates **sparse** user item rating matrix

- as product of **two dense** matrices, User and Item factor matrices
- tries to **learn the hidden features** of each user and item
- algorithm **alternatively fixes** one factor matrix and **solves** for the other



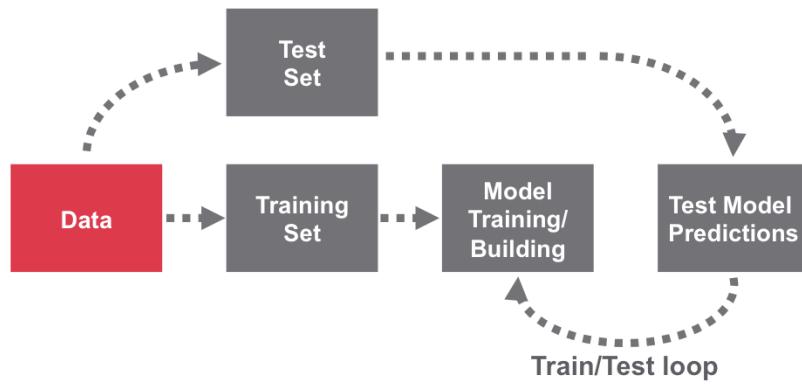
ALS is an iterative algorithm. During each iteration, the algorithm alternatively fixes one factor matrix and solves for the other. This process continues until it converges. The practice of alternating between which matrix to optimize is why the process is called Alternating Least Squares.

ML Cross-Validation Process



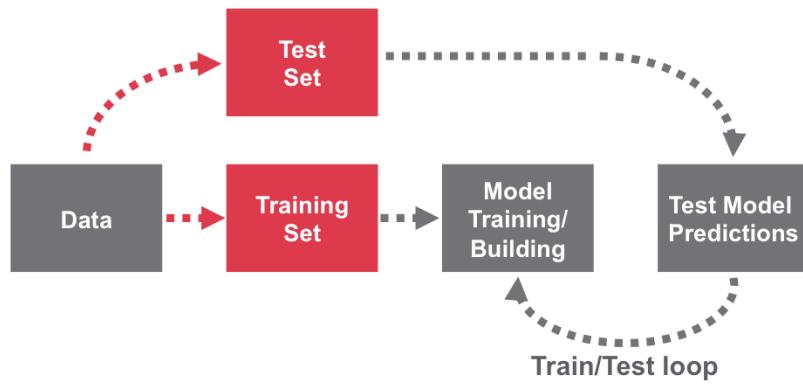
A typical machine learning workflow is shown here. To make our predictions, we will perform the following steps:

ML Cross-Validation Process



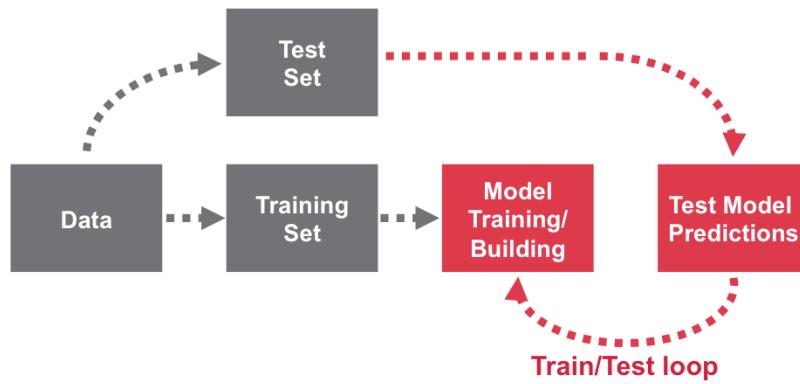
Load the sample data, and parse the data into the input format for the ALS algorithm.

ML Cross-Validation Process



Split the data into two parts, one for building the model and one for testing the model.

ML Cross-Validation Process



We then run the ALS algorithm to build and train a user product matrix model.

We make predictions with the training data, and observe the results.

Then, test the model with the test data.

Parse Input

```
// load ratings data into a RDD  
val ratingText = sc.textFile("/data/ratings.dat")  
  
// create an RDD of Ratings objects  
val ratingsRDD = ratingText.map(parseRating).cache()
```

In the first step, we load the ratings data into the ratingText RDD.

Then we use the map transformation on the ratingText RDD. This will apply the parseRating function to each element in ratingText, and return a new RDD of Rating objects. We cache ratingsRDD, since we will use this data to build the matrix model.



Parse Input Lines into Ratings Objects

```
// Import mllib recommendation data types
import org.apache.spark.mllib.recommendation.{ALS,
MatrixFactorizationModel, Rating}

// parse string: UserID::MovieID::Rating
def parseRating(str: String): Rating= {
    val fields = str.split("::")
    Rating(fields(0).toInt, fields(1).toInt,
        fields(2).toDouble)
}
```

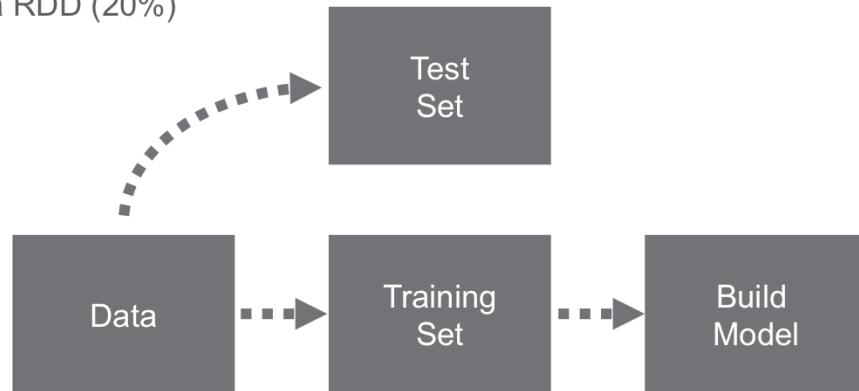
The parseRating function parses a line from the ratings data file into the MLlib Rating class. We will use this as input for the ALS run method.



Build Model

Split ratings RDD into:

- Training data RDD (80%)
- Test data RDD (20%)



Next we split the data into two parts, one for building the model and one for testing the model.

Build Model

```
// Randomly split ratings RDD into training data RDD (80%)
// and test data RDD (20%)
val splits = ratingsRDD.randomSplit(Array(0.8, 0.2), 0L)

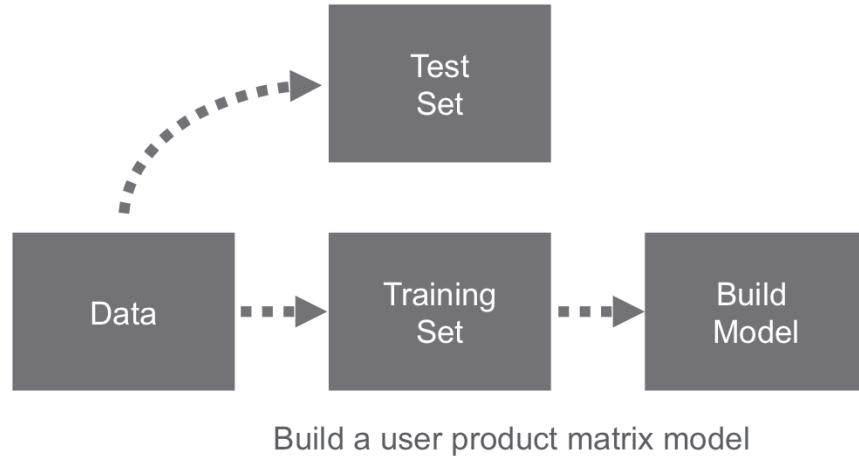
val trainingRatingsRDD = splits(0).cache()
val testRatingsRDD = splits(1).cache()

// build a ALS user product matrix model with rank=20,
iterations=10
val model = (new ALS().setRank(20).setIterations(10)
    .run(trainingRatingsRDD))
```

The top line in the code shown here applies the 80-20 split to our data.



Build Model



Then run the ALS algorithm to build and train a user product matrix model.

Build Model

```
// Randomly split ratings RDD into training data RDD (80%)
and test data RDD (20%)
val splits = ratingsRDD.randomSplit(Array(0.8, 0.2), 0L)

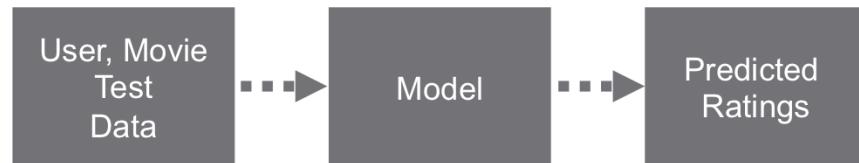
val trainingRatingsRDD = splits(0).cache()
val testRatingsRDD = splits(1).cache()

// build a ALS user product matrix model with rank=20,
iterations=10
val model = (new ALS().setRank(20).setIterations(10)
    .run(trainingRatingsRDD))
```

The last line builds the new product matrix model.



Get Predictions



Now that we have trained our model, we want to get predicted movie ratings for the test data.

Get Predictions

```
// get predicted ratings to compare to test ratings
val testUserProductRDD = testRatingsRDD.map {
    case Rating(user, product, rating) => (user, product)
}
// call model.predict with test Userid, MovieId input data
val predictionsForTestRDD = model.predict(testUserProductRDD)
```

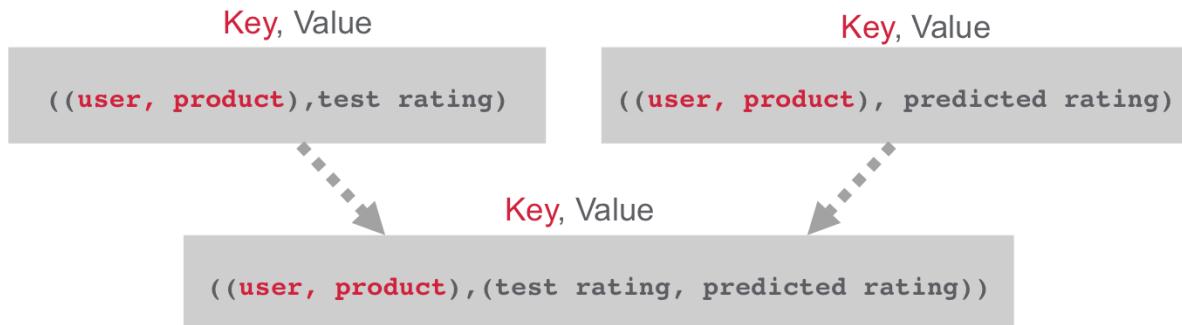
First, we use a map on the testRatingsRDD to create a new RDD of test User IDs and Movie IDs, without any ratings.

We then call the model.predict method, with the new testUserProductRDD, to get predicted ratings for each test User ID and Movie ID pair.



Compare Predictions to Tests

Join predicted ratings to test ratings in order to compare



Next we compare test User ID, Movie ID rating pairs to the predicted User ID, Movie ID rating pairs.

Test Model

```
// prepare predictions for comparison
val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}

// prepare test for comparison
val testKeyedByUserProductRDD = testRatingsRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}

//Join the test with predictions
val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD
  .join(predictionsKeyedByUserProductRDD)
```

Here we create User ID, Movie ID ratings key-value pairs, so that we can compare the test ratings to the predicted ratings.



Compare Predictions to Tests

Find false positives where:

```
test rating <= 1 and predicted rating >= 4
```

Key, Value

```
((user, product),(test rating, predicted rating))
```

We compare the ratings predictions to the actual rating to find false positives where the predicted rating was high, and the actual rating was low.



Test Model

```
val falsePositives =(testAndPredictionsJoinedRDD.filter{  
    case ((user, product), (ratingT, ratingP)) =>  
        (ratingT <= 1 && ratingP >=4)  
    })  
falsePositives.take(2)  
  
Array[((Int, Int), (Double, Double))] =  
((3842,2858),(1.0,4.106488210964762)),  
((6031,3194),(1.0,4.790778049100913))
```

The code shown here compares actual ratings to predicted ratings by filtering on ratings where the test rating<=1, and the predicted rating is >=4.



Test Model Mean Absolute Error

```
//Evaluate the model using Mean Absolute Error (MAE) between
//test and predictions
val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
  case ((user, product), (testRating, predRating)) =>
    val err = (testRating - predRating)
    Math.abs(err)
}.mean()
meanAbsoluteError: Double = 0.7244940545944053
```

The model can also be evaluated by calculating the mean absolute error, or MAE. The MAE is the mean of the absolute error between the actual test rating and the predicted rating.



Get Predictions for New User

```
val newRatingsRDD=sc.parallelize(Array(Rating(0,260,4),Rating(0,1,3))
// union
val unionRatingsRDD = ratingsRDD.union(newRatingsRDD)
// build a ALS user product matrix model
val model = (new ALS().setRank(20).setIterations(10)
    .run(unionRatingsRDD))

// get 5 recs for userid 0
val topRecsForUser = model.recommendProducts(0, 5)
```

In this example, we get ratings for a new user, with the ID of 0.

First, we create an RDD with the User ID, Movie ID, and ratings for some movies.

Then, we add this RDD to the existing ratings data RDD with a union. We call an ALS run using the new unionRatingsRDD, which will return a new recommendation model.

We can now use this model to get recommendations by calling model.recommendProducts with the userid 0, and the number of recommendations desired.



Knowledge Check



Knowledge Check



Place the steps below in the order to correctly describe a supervised learning workflow.

Use the validated model
with new data

Load sample data

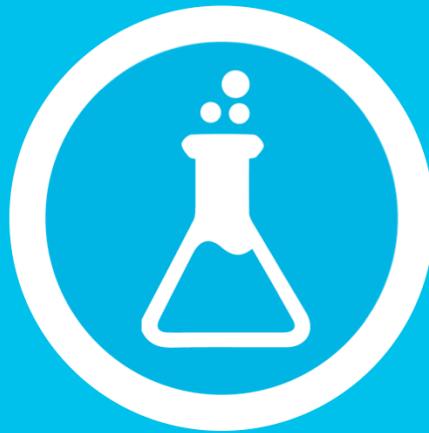
Build, test and tune the
model

Extract features

Split the data into
training and data sets

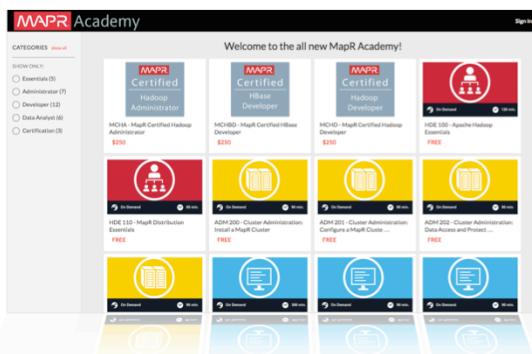
1. Load sample data
2. Extract features
3. Split the data into training and test sets.
4. Build, Test, and Tune the model
5. Use the validated model with new data

Lab 10 – Use Apache Spark MLlib to Make Recommendations



Load and inspect data using the Spark shell
Use Spark to make movie recommendations

Congratulations!



You have completed this course.

Register for more at: learn.mapr.com

Congratulations! You have completed DEV 362 Create Data Pipelines with Apache Spark. Visit the MapR Academy to find more courses about big data processing and analysis, and the Hadoop ecosystem.